man pages section 3: Extended Library Functions

Part No: 816–5172–17
August 2011

ORACLE

111219@25097

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 6 contains available games and demos.
- 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 9 provides reference information needed to write device drivers in the kernel environment. It describes two device driver interface specifications: the Device Driver Interface (DDI) and the Driver/Kernel Interface (DKI).
- Section 9E describes the DDI/DKI, 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 - Part 1

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_TYPE_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	Evolving
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
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.

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:

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

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

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

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

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	Evolving
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 acos, acosf, acosl – arc cosine functions

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double acos(double x);

float acosf(float x);

long double acosl(long double x);
```

Description These functions compute the principal value of the arc cosine of *x*. The value of *x* should be in the range [−1,1].

Return Values Upon successful completion, these functions return the arc cosine of *x* in the range [0, π] radians.

For finite values of *x* not in the range [−1,1], a domain error occurs and NaN is returned.

If *x* is NaN, NaN is returned.

If *x* is +1, +0 is returned.

If *x* is ±Inf, a domain error occurs and NaN is returned.

For exceptional cases, `matherr(3M)` tabulates the values to be returned by `acos()` as specified by SVID3 and XPG3.

Errors These functions will fail if:

Domain Error — The *x* argument is finite and not in the range [-1,1], or is ±Inf.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, the invalid floating-point exception is raised.

The `acos()` function sets `errno` to `EDOM` if *x* is not ±Inf or NaN and is not in the range [−1,1].

Usage An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. An application should either examine the return value or check the floating point exception flags to detect exceptions.

An application can also set `errno` to 0 before calling `acos()`. On return, if `errno` is non-zero, an error has occurred. The `acosf()` and `acosl()` functions do not set `errno`.

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

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

See Also cos(3M), feclearexcept(3M), fetestexcept(3M), isnan(3M), math.h(3HEAD), matherr(3M), attributes(5), standards(5)

Name acosh, acoshf, acoshl – inverse hyperbolic cosine functions

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double acosh(double x);

float acoshf(float x);

long double acoshl(long double x);
```

Description These functions compute the inverse hyperbolic cosine of their argument *x*.

Return Values Upon successful completion, these functions return the inverse hyperbolic cosine of their argument.

For finite values of $x < 1$, a domain error occurs and NaN is returned.

If *x* is NaN, NaN is returned.

If *x* is +1, +0 is returned.

If *x* is +Inf, +Inf is returned.

If *x* is –Inf, a domain error occurs and NaN is returned.

For exceptional cases, `matherr(3M)` tabulates the values to be returned by `acosh()` as specified by SVID3 and XPG3.

Errors These functions will fail if:

Domain Error — The *x* argument is finite and less than 1.0, or is –Inf.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, the invalid floating-point exception is raised.

The `acosh()` function sets `errno` to `EDOM` if *x* is less than 1.0.

Usage An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. An application should either examine the return value or check the floating point exception flags to detect exceptions.

An application can also set `errno` to 0 before calling `acosh()`. On return, if `errno` is non-zero, an error has occurred. The `acoshf()` and `acoshl()` functions do not set `errno`.

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

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

See Also cosh(3M), feclearexcept(3M), fetestexcept(3M), math.h(3HEAD), matherr(3M), attributes(5), standards(5)

Name asin, asinf, asinl – arc sine function

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double asin(double x);

float asinf(float x);

long double asinl(long double x);
```

Description These functions compute the principal value of the arc sine of their argument *x*. The value of *x* should be in the range [−1,1].

Return Values Upon successful completion, these functions return the arc sine of *x* in the range [−π/2, π/2] radians.

For finite values of *x* not in the range [−1,1], a domain error occurs and a NaN is returned.

If *x* is NaN, NaN is returned.

If *x* is ±0, *x* is returned.

If *x* is ±Inf, a domain error occurs and a NaN is returned.

For exceptional cases, `matherr(3M)` tabulates the values to be returned by `asin()` as specified by SVID3 and XPG3.

Errors These functions will fail if:

Domain Error — The *x* argument is finite and not in the range [−1,1], or is ±Inf.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, the invalid floating-point exception is raised.

The `asin()` function sets `errno` to `EDOM` if *x* is not ±Inf or NaN and is not in the range [−1,1].

Usage An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. An application should either examine the return value or check the floating point exception flags to detect exceptions.

An application can also set `errno` to 0 before calling `asin()`. On return, if `errno` is non-zero, an error has occurred. The `asinf()` and `asinl()` functions do not set `errno`.

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

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

See Also isnan(3M), feclearexcept(3M), fetestexcept(3M), math.h(3HEAD), matherr(3M), sin(3M), attributes(5), standards(5)

Name asinh, asinhf, asinhl – inverse hyperbolic sine functions

Synopsis

```
cc [ flag... ] file... -lm [ library... ]
#include <math.h>

double asinh(double x);

float asinhf(float x);

long double asinhl(long double x);
```

Description These functions compute the inverse hyperbolic sine of their argument *x*.

Return Values Upon successful completion, these functions return the inverse hyperbolic sine of their argument.

If *x* is NaN, NaN is returned.

If *x* is ±0 or ±Inf, *x* is returned.

Errors No errors are defined.

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

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

See Also math.h(3HEAD), sinh(3M), attributes(5), standards(5)

Name atan2, atan2f, atan2l – arc tangent function

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double atan2(double y, double x);

float atan2f(float y, float x);

long double atan2l(long double y, long double x);
```

Description These functions compute the principal value of the arc tangent of y/x, using the signs of both arguments to determine the quadrant of the return value.

Return Values Upon successful completion, these functions return the arc tangent of y/x in the range $[-\pi,\pi]$ radians.

If y is ± 0 and x is < 0, $\pm\pi$ is returned.

If y is ± 0 and x is > 0, ± 0 is returned.

If y is < 0 and x is ± 0, $-\pi/2$ is returned.

If y is > 0 and x is ± 0, $\pi/2$ is returned.

If x is 0, a pole error does not occur.

If either x or y is NaN, a NaN is returned.

If y is ± 0 and x is -0, $\pm\pi$ is returned.

If y is ± 0 and x is $+0$, ± 0 is returned.

For finite values of $\pm y > 0$, if x is $-$Inf, $\pm\pi$ is returned.

For finite values of $\pm y > 0$, if x is $+$Inf, ± 0 is returned.

For finite values of x, if y is $\pm$Inf, $\pm\pi/2$ is returned.

If y is $\pm$Inf and x is $-$Inf, $\pm 3\pi/4$ is returned.

If y is $\pm$Inf and x is $+$Inf, $\pm\pi/4$ is returned.

If both arguments are 0, a domain error does not occur.

Errors No errors are defined.

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

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

See Also atan(3M), isnan(3M), math.h(3HEAD)tan(3M), attributes(5), standards(5)

Name atan, atanf, atanl – arc tangent function

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double atan(double x);

float atanf(float x);

long double atanl(long double x);
```

Description These functions compute the principal value of the arc tangent of *x*.

Return Values Upon successful completion, these functions return the arc tangent of *x* in the range [−π/2,π/2] radians.

If *x* is NaN, NaN is returned.

If x is ±0, *x* is returned.

If *x* is ±Inf, ±π/2 is returned.

Errors No errors are defined.

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

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

See Also atan2(3M), isnan(3M), math.h(3HEAD), tan(3M), attributes(5), standards(5)

Name atanh, atanhf, atanhl – inverse hyperbolic tangent functions

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double atanh(double x);

float atanhf(float x);

long double atanhl(long double x);
```

Description These functions compute the inverse hyperbolic tangent of their argument *x*.

Return Values Upon successful completion, these functions return the inverse hyperbolic tangent of their argument.

If *x* is ±1, a pole error occurs and `atanh()`, `atanhf()`, and `atanhl()` return the value of the macro `HUGE_VAL`, `HUGE_VALF`, and `HUGE_VALL`, respectively, with the same sign as the correct value of the function.

For finite $|x| > 1$, a domain error occurs and a NaN is returned.

If *x* is NaN, NaN is returned.

If *x* is +0, *x* is returned.

If *x* is +Inf, a domain error occurs and a NaN is returned.

For exceptional cases, matherr(3M) tabulates the values to be returned by `atanh()` as specified by SVID3 and XPG3.

Errors These functions will fail if:

Domain Error — The *x* argument is finite and not in the range [-1,1], or is ±Inf.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, the invalid floating-point exception is raised.

The `atanh()` function sets `errno` to `EDOM` if the absolute value of *x* is greater than 1.0.

Pole Error — The *x* argument is ±1.

If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the divide-by-zero floating-point exception is raised.

The `atanh()` function sets `errno` to `ERANGE` if the absolute value of *x* is equal to 1.0.

Usage An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. An application should either examine the return value or check the floating point exception flags to detect exceptions.

An application can also set `errno` to 0 before calling `atanh()`. On return, if `errno` is non-zero, an error has occurred. The `atanhf()` and `atanhl()` functions do not set `errno`.

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

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

See Also `feclearexcept(3M)`, `fetestexcept(3M)`, `math.h(3HEAD)`, `matherr(3M)`, `tanh(3M)`, `attributes(5)`, `standards(5)`

Name au_open, au_close, au_write – construct and write audit records

Synopsis

```
cc [ flag... ] file... -lbsm -lsocket -lnsl [ library... ]
#include <bsm/libbsm.h>

int au_close(int d, int keep, short event);

int au_open(void);

int au_write(int d, token_t *m);
```

Description The `au_open()` function returns an audit record descriptor to which audit tokens can be written using `au_write()`. The audit record descriptor is an integer value that identifies a storage area where audit records are accumulated.

The `au_close()` function terminates the life of an audit record *d* of type *event* started by `au_open()`. If the *keep* parameter is `AU_TO_NO_WRITE`, the data contained therein is discarded. If the *keep* parameter is `AU_TO_WRITE`, the additional parameters are used to create a header token. Depending on the audit policy information obtained by `auditon(2)`, additional tokens such as *sequence* and *trailer* tokens can be added to the record. The `au_close()` function then writes the record to the audit trail by calling `audit(2)`. Any memory used is freed by calling `free(3C)`.

The `au_write()` function adds the audit token pointed to by *m* to the audit record identified by the descriptor *d*. After this call is made the audit token is no longer available to the caller.

Return Values Upon successful completion, `au_open()` returns an audit record descriptor. If a descriptor could not be allocated, `au_open()` returns −1 and sets `errno` to indicate the error.

Upon successful completion, `au_close()` returns `0`. If *d* is an invalid or corrupted descriptor or if `audit()` fails, `au_close()` returns −1 without setting `errno`. If `audit()` fails, `errno` is set to one of the error values described on the `audit(2)` manual page.

Upon successful completion, `au_write()` returns `0`. If *d* is an invalid descriptor or *m* is an invalid token, or if `audit()` fails, `au_write()` returns −1 without setting `errno`. If `audit()` fails, `errno` is set to one of the error values described on the `audit(2)` manual page.

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

`ENOMEM` The physical limits of the system have been exceeded such that sufficient memory cannot be allocated.

`EAGAIN` There is currently insufficient memory available. The application can try again later.

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

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

See Also bsmconv(1M), audit(2), auditon(2), au_preselect(3BSM), au_to(3BSM), free(3C), attributes(5)

Notes The functionality described on this manual page is available only if the Basic Security Module (BSM) has been enabled. See bsmconv(1M) for more information.

Name au_preselect – preselect an audit event

Synopsis

```
cc [ flag... ] file... -lbsm  -lsocket   -lnsl   [ library... ]
#include <bsm/libbsm.h>

int au_preselect(au_event_t event, au_mask_t *mask_p, int sorf, int flag);
```

Description The `au_preselect()` function determines whether the audit event *event* is preselected against the binary preselection mask pointed to by *mask_p* (usually obtained by a call to `getaudit(2)`). The `au_preselect()` function looks up the classes associated with *event* in `audit_event(4)` and compares them with the classes in *mask_p*. If the classes associated with *event* match the classes in the specified portions of the binary preselection mask pointed to by *mask_p*, the event is said to be preselected.

The *sorf* argument indicates whether the comparison is made with the success portion, the failure portion, or both portions of the mask pointed to by *mask_p*.

The following are the valid values of *sorf*:

`AU_PRS_SUCCESS`	Compare the event class with the success portion of the preselection mask.
`AU_PRS_FAILURE`	Compare the event class with the failure portion of the preselection mask.
`AU_PRS_BOTH`	Compare the event class with both the success and failure portions of the preselection mask.

The *flag* argument tells `au_preselect()` how to read the `audit_event(4)` database. Upon initial invocation, `au_preselect()` reads the `audit_event(4)` database and allocates space in an internal cache for each entry with `malloc(3C)`. In subsequent invocations, the value of *flag* determines where `au_preselect()` obtains audit event information. The following are the valid values of *flag*:

`AU_PRS_REREAD`	Get audit event information by searching the `audit_event(4)` database.
`AU_PRS_USECACHE`	Get audit event information from internal cache created upon the initial invocation. This option is much faster.

Return Values Upon successful completion,`au_preselect()` returns 0 if *event* is not preselected or 1 if *event* is preselected. If `au_preselect()` could not allocate memory or could not find *event* in the `audit_event(4)` database, −1 is returned.

Files

`/etc/security/audit_class`	file mapping audit class number to audit class names and descriptions
`/etc/security/audit_event`	file mappint audit even number to audit event names and associates

Attributes See attributes(5) for a description of the following attributes:

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

See Also bsmconv(1M), getaudit(2), au_open(3BSM), getauclassent(3BSM), getauevent(3BSM), malloc(3C), audit_class(4), audit_event(4), attributes(5)

Notes The au_preselect() function is normally called prior to constructing and writing an audit record. If the event is not preselected, the overhead of constructing and writing the record can be saved.

The functionality described on this manual page is available only if the Basic Security Module (BSM) has been enabled. See bsmconv(1M) for more information.

Name au_to, au_to_arg, au_to_arg32, au_to_arg64, au_to_attr, au_to_cmd, au_to_data, au_to_groups, au_to_in_addr, au_to_ipc, au_to_iport, au_to_me, au_to_newgroups, au_to_opaque, au_to_path, au_to_process, au_to_process_ex, au_to_return, au_to_return32, au_to_return64, au_to_socket, au_to_subject, au_to_subject_ex, au_to_text – create audit record tokens

Synopsis

```
cc [ flag... ] file... -lbsm  -lsocket   -lnsl   [ library... ]
#include <sys/types.h>
#include <sys/vnode.h>
#include <netinet/in.h>
#include <bsm/libbsm.h>

token_t *au_to_arg(char n, char *text, uint32_t v);

token_t *au_to_arg32(char n, char *text, uint32_t v);

token_t *au_to_arg64(char n, char *text, uint64_t v);

token_t *au_to_attr(struct vattr *attr);

token_t *au_to_cmd(uint_t argc, char **argv, char **envp);

token_t *au_to_data(char unit_print, char unit_type, char unit_count,
     char *p);

token_t *au_to_groups(int *groups);

token_t *au_to_in_addr(struct in_addr *internet_addr);

token_t *au_to_ipc(char type, int id);

token_t *au_to_iport(u_short_t iport);

token_t *au_to_me(void);

token_t *au_to_newgroups(int n, gid_t *groups);

token_t *au_to_opaque(char *data, short bytes);

token_t *au_to_path(char *path);

token_t *au_to_process(au_id_t auid, uid_t euid, gid_t egid,
     uid_t ruid, gid_t rgid, pid_t pid, au_asid_t sid, au_tid_t *tid);

token_t *au_to_process_ex(au_id_t auid, uid_t euid, gid_t egid,
     uid_t ruid, gid_t rgid, pid_t pid, au_asid_t sid, au_tid_addr_t *tid);

token_t *au_to_return(char number, uin32t_t value);

token_t *au_to_return32(char number, uin32t_t value);

token_t *au_to_return64(char number, uin64t_t value);

token_t *au_to_socket(struct oldsocket *so);

token_t *au_to_subject(au_id_t auid, uid_t euid, gid_t egid,
     uid_t ruid, gid_t rgid, pid_t pid, au_asid_t sid, au_tid_t *tid);
```

```
token_t *au_to_subject_ex(au_id_t auid, uid_t euid, gid_t egid,
     uid_t ruid, gid_t rgid, pid_t pid, au_asid_t sid, au_tid_addr_t *tid);

token_t *au_to_text(char *text);
```

Description The `au_to_arg()`, `au_to_arg32()`, and `au_to_arg64()` functions format the data in *v* into an "argument token". The *n* argument indicates the argument number. The *text* argument is a null-terminated string describing the argument.

The `au_to_attr()` function formats the data pointed to by *attr* into a "vnode attribute token".

The `au_to_cmd()` function formats the data pointed to by *argv* into a "command token". A command token reflects a command and its parameters as entered. For example, the `pfexec(1)` utility uses `au_to_cmd()` to record the command and arguments it reads from the command line.

The `au_to_data()` function formats the data pointed to by *p* into an "arbitrary data token". The *unit_print* parameter determines the preferred display base of the data and is one of `AUP_BINARY`, `AUP_OCTAL`, `AUP_DECIMAL`, `AUP_HEX`, or `AUP_STRING`. The *unit_type* parameter defines the basic unit of data and is one of `AUR_BYTE`, `AUR_CHAR`, `AUR_SHORT`, `AUR_INT`, or `AUR_LONG`. The *unit_count* parameter specifies the number of basic data units to be used and must be positive.

The `au_to_groups()` function formats the array of 16 integers pointed to by *groups* into a "groups token". The `au_to_newgroups()` function (see below) should be used in place of this function.

The `au_to_in_addr()` function formats the data pointed to by *internet_addr* into an "internet address token".

The `au_to_ipc()` function formats the data in the *id* parameter into an "interprocess communications ID token".

The `au_to_iport()` function formats the data pointed to by *iport* into an "ip port address token".

The `au_to_me()` function collects audit information from the current process and creates a "subject token" by calling `au_to_subject()`.

The `au_to_newgroups()` function formats the array of *n* integers pointed to by *groups* into a "newgroups token". This function should be used in place of `au_to_groups()`.

The `au_to_opaque()` function formats the *bytes* bytes pointed to by *data* into an "opaque token". The value of *size* must be positive.

The `au_to_path()` function formats the path name pointed to by *path* into a "path token."

The `au_to_process()` function formats an *auid* (audit user ID), an *euid* (effective user ID), an *egid* (effective group ID), a *ruid* (real user ID), a *rgid* (real group ID), a *pid* (process ID), an *sid* (audit session ID), and a *tid* (audit terminal ID containing an IPv4 IP address), into a "process token". A process token should be used when the process is the object of an action (ie. when the process is the receiver of a signal). The `au_to_process_ex()` function (see below) should be used in place of this function.

The `au_to_process_ex()` function formats an *auid* (audit user ID), an *euid* (effective user ID), an *egid* (effective group ID), a *ruid* (real user ID), a *rgid* (real group ID), a *pid* (process ID), an *sid* (audit session ID), and a *tid* (audit terminal ID containing an IPv4 or IPv6 IP address), into a "process token". A process token should be used when the process is the object of an action (that is, when the process is the receiver of a signal). This function should be used in place of `au_to_process()`.

The `au_to_return()`, `au_to_return32()`, and `au_to_return64()` functions format an error number *number* and a return value *value* into a "return value token".

The `au_to_socket()` function format the data pointed to by *so* into a "socket token."

The `au_to_subject()` function formats an *auid* (audit user ID), an *euid* (effective user ID), an *egid* (effective group ID), a *ruid* (real user ID), an *rgid* (real group ID), a *pid* (process ID), an *sid* (audit session ID), an *tid* (audit terminal ID containing an IPv4 IP address), into a "subject token". The `au_to_subject_ex()` function (see below) should be used in place of this function.

The `au_to_subject_ex()` function formats an *auid* (audit user ID), an *euid* (effective user ID), an *egid* (effective group ID), a *ruid* (real user ID), an *rgid* (real group ID), a *pid* (process ID), an *sid* (audit session ID), an *tid* (audit terminal ID containing an IPv4 or IPv6 IP address), into a "subject token". This function should be used in place of `au_to_subject()`.

The `au_to_text()` function formats the null-terminated string pointed to by *text* into a "text token".

Return Values

These functions return `NULL` if memory cannot be allocated to put the resultant token into, or if an error in the input is detected.

Attributes

See `attributes(5)` for a description of the following attributes:

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

See Also bsmconv(1M), au_open(3BSM), attributes(5)

Notes The functionality described on this manual page is available only if the Basic Security Module (BSM) has been enabled. See bsmconv(1M) for more information.

Name auto_ef, auto_ef_file, auto_ef_str, auto_ef_free, auto_ef_get_encoding, auto_ef_get_score – auto encoding finder functions

Synopsis

```
cc [ flag ... ] file... -lauto_ef [ library... ]
#include <auto_ef.h>

size_t auto_ef_file(auto_ef_t **info, const char *filename, int flags);

size_t auto_ef_str(auto_ef_t **info, const char *buffer, size_t bufsize,
     int flags);

void auto_ef_free(auto_ef_t *info);

char *auto_ef_get_encoding(auto_ef_t info);

double auto_ef_get_score(auto_ef_t info);
```

Description Auto encoding finder provides functions that find the encoding of given file or string.

The `auto_ef_file()` function examines text in the file specified with *filename* and returns information on possible encodings.

The *info* argument is a pointer to a pointer to an `auto_ef_t`, the location at which the pointer to the `auto_ef_t` array is stored upon return.

The *flags* argument specifies the level of examination. Currently only one set of flags, exclusive each other, is available: `AE_LEVEL_0`, `AE_LEVEL_1`, `AE_LEVEL_2`, and `AE_LEVEL_3`. The `AE_LEVEL_0` level is fastest but the result can be less accurate. The `AE_LEVEL_3` level produces best result but can be slow. If the *flags* argument is unspecified, the default is `AE_LEVEL_0`. When another flag or set of flags are defined in the future, use the inclusive-bitwise OR operation to specify multiple flags.

Information about encodings are stored in data type`auto_ef_t` in the order of possibility with the most possible encoding stored first. To examine the information, use the `auto_ef_get_encoding()` and `auto_ef_get_score()` access functions. For a list of encodings with which `auto_ef_file()` can examine text, see `auto_ef(1)`.

If `auto_ef_file()` cannot determine the encoding of text, it returns 0 and stores `NULL` at the location pointed by *info*.

The `auto_ef_get_encoding()` function returns the name of the encoding. The returned string is vaild until until the location pointed to by *info* is freed with `auto_ef_free()`. Applications should not use `free(3C)` to free the pointer returned by `auto_ef_get_encoding()`.

The `auto_ef_get_score()` function returns the score of this encoding in the range between 0.0 and 1.0.

The `auto_ef_str()` function is identical to `auto_ef_file()`, except that it examines text in the buffer specified by *buffer* with a maximum size of *bufsize* bytes, instead of text in a file.

The `auto_ef_free()` function frees the area allocated by `auto_ef_file()` or by `auto_ef_str()`, taking as its argument the pointer stored at the location pointed to by *info*.

Return Values Upon successful completion, the `auto_ef_file()` and `auto_ef_str()` functions return the number of possible encodings for which information is stored. Otherwise, –1 is returned.

The `auto_ef_get_encoding()` function returns the string that stores the encoding name.

the `auto_ef_get_score()` function returns the score value for encoding the name with the examined text data.

Errors The `auto_ef_file()` and `auto_ef_str()` will fail if:

`EACCES` Search permission is denied on a component of the path prefix, the file exists and the permissions specified by mode are denied, the file does not exist and write permission is denied for the parent directory of the file to be created, or `libauto_ef` cannot find the internal hashtable.

`EINTR` A signal was caught during the execution.

`ENOMEM` Failed to allocate area to store the result.

`EMFILE` Too many files descriptors are currently open in the calling process.

`ENFILE` Too many files are currently open in the system.

Examples **EXAMPLE 1** Specify the array index to examine stored information.

Since `auto_ef_file()` stores the array whose elements hold information on each possible encoding, the following example specifies the array index to examine the stored information.

```
#include <auto_ef.h>
auto_ef_t       *array_info;
size_t          number;
char            *encoding;

number = auto_ef_file(&array_info, filename, flags);
encoding = auto_ef_get_encoding(array_info[0]);
auto_ef_free(array_info);
```

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

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

See Also auto_ef(1), libauto_ef(3LIB), attributes(5)

Name au_user_mask – get user's binary preselection mask

Synopsis

```
cc [ flag... ] file... -lbsm  -lsocket  -lnsl  [ library... ]
#include <bsm/libbsm.h>

int au_user_mask(char *username, au_mask_t *mask_p);
```

Description The `au_user_mask()` function reads the default, system wide audit classes from `audit_control(4)`, combines them with the per-user audit classes from the `audit_user(4)` database, and updates the binary preselection mask pointed to by *mask_p* with the combined value.

The audit flags in the *flags* field of the `audit_control(4)` database and the *always-audit-flags* and *never-audit-flags* from the `audit_user(4)` database represent binary audit classes. These fields are combined by `au_preselect(3BSM)` as follows:

mask = (*flags* + *always-audit-flags*) − *never-audit-flags*

The `au_user_mask()` function fails only if both the both the `audit_control(4)` and the `audit_user(4)` database entries could not be retrieved. This allows for flexible configurations.

Return Values Upon successful completion, `au_user_mask()` returns 0. It fails and returns −1 if both the `audit_control(4)` and the `audit_user(4)` database entries could not be retrieved.

Files

`/etc/security/audit_control`	file containing default parameters read by the audit daemon, `auditd(1M)`
`/etc/security/audit_user`	file that stores per-user audit event mask

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

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

See Also `login(1)`, `bsmconv(1M)`, `getaudit(2)`, `setaudit(2)`, `au_preselect(3BSM)`, `getacinfo(3BSM)`, `getauusernam(3BSM)`, `audit_control(4)`, `audit_user(4)`, `attributes(5)`

Notes The `au_user_mask()` function should be called by programs like `login(1)` which set a process's preselection mask with `setaudit(2)`. `getaudit(2)` should be used to obtain audit characteristics for the current process.

The functionality described on this manual page is available only if the Basic Security Module (BSM) has been enabled. See `bsmconv(1M)` for more information.

Name bgets – read stream up to next delimiter

Synopsis

```
cc [ flag ... ] file ... -lgen [ library ... ]
#include <libgen.h>

char *bgets(char *buffer, size_t count, FILE  *stream,
     const char *breakstring);
```

Description The `bgets()` function reads characters from *stream* into *buffer* until either *count* is exhausted or one of the characters in *breakstring* is encountered in the stream. The read data is terminated with a null byte ('`\0`') and a pointer to the trailing null is returned. If a *breakstring* character is encountered, the last non-null is the delimiter character that terminated the scan.

Note that, except for the fact that the returned value points to the end of the read string rather than to the beginning, the call

```
bgets(buffer, sizeof buffer, stream, "\n");
```

is identical to

```
fgets (buffer, sizeof buffer, stream);
```

There is always enough room reserved in the buffer for the trailing null character.

If *breakstring* is a null pointer, the value of *breakstring* from the previous call is used. If *breakstring* is null at the first call, no characters will be used to delimit the string.

Return Values `NULL` is returned on error or end-of-file. Reporting the condition is delayed to the next call if any characters were read but not yet returned.

Examples **EXAMPLE 1** Example of the `bgets()` function.

The following example prints the name of the first user encountered in `/etc/passswd`, including a trailing ":"

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

int main()
{
    char buffer[8];
    FILE *fp;

    if ((fp = fopen("/etc/passwd","r")) == NULL) {
        perror("/etc/passwd");
        return 1;
    }
    if (bgets(buffer, 8, fp, ":") == NULL) {
        perror("bgets");
        return 1;
```

EXAMPLE 1 Example of the `bgets()` function. *(Continued)*

```
    }
    (void) puts(buffer);
    return 0;
}
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also

`gets(3C)`, `attributes(5)`

Notes

When compiling multithread applications, the `_REENTRANT` flag must be defined on the compile line. This flag should only be used in multithreaded applications.

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 *Oracle 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
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 bufsplit – split buffer into fields

Synopsis

```
cc [ flag ... ] file ... -lgen [ library ... ]
#include <libgen.h>

size_t bufsplit(char *buf, size_t n, char **a);
```

Description `bufsplit()` examines the buffer, *buf*, and assigns values to the pointer array, *a*, so that the pointers point to the first *n* fields in *buf* that are delimited by TABs or NEWLINEs.

To change the characters used to separate fields, call `bufsplit()` with *buf* pointing to the string of characters, and *n* and *a* set to zero. For example, to use colon (:), period (.), and comma (,), as separators along with TAB and NEWLINE:

```
bufsplit (":.,\t\n", 0, (char**)0 );
```

Return Values The number of fields assigned in the array *a*. If *buf* is zero, the return value is zero and the array is unchanged. Otherwise the value is at least one. The remainder of the elements in the array are assigned the address of the null byte at the end of the buffer.

Examples **EXAMPLE 1** Example of `bufsplit()` function.

```
/*
 * set a[0] = "This", a[1] = "is", a[2] = "a",
 * a[3] = "test"
 */
bufsplit("This\tis\ta\ttest\n", 4, a);
```

Notes `bufsplit()` changes the delimiters to null bytes in *buf*.

When compiling multithreaded applications, the `_REENTRANT` flag must be defined on the compile line. This flag should only be used in multithreaded applications.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `attributes(5)`

Name cabs, cabsf, cabsl – return a complex absolute value

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <complex.h>

double cabs(double complex z);

float cabsf(float complex z);

long double cabsl(long double complex z);
```

Description These functions compute the complex absolute value (also called norm, modulus, or magnitude) of *z*.

Return Values These functions return the complex absolute value.

Errors No errors are defined.

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

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

See Also complex.h(3HEAD), attributes(5), standards(5)

Name cacos, cacosf, cacosl – complex arc cosine functions

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <complex.h>

double complex cacos(double complex z);

float complex cacosf(float complex z);

long double complex cacosl(long double complex z);
```

Description These functions compute the complex arc cosine of *z*, with branch cuts outside the interval [-1, +1] along the real axis.

Return Values These functions return the complex arc cosine value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval [0, π] along the real axis.

Errors No errors are defined.

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

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

See Also ccos(3M), complex.h(3HEAD), attributes(5), standards(5)

Name cacosh, cacoshf, cacoshl – complex arc hyperbolic cosine functions

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <complex.h>

double complex cacosh(double complex z);

float complex cacoshf(float complex z);

long double complex cacoshl(long double complex z);
```

Description These functions compute the complex arc hyperbolic cosine of *z*, with a branch cut at values less than 1 along the real axis.

Return Values These functions return the complex arc hyperbolic cosine value, in the range of a half-strip of non-negative values along the real axis and in the interval [$-i\pi$, $+i\pi$] along the imaginary axis.

Errors No errors are defined.

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

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

See Also `ccosh(3M)`, `complex.h(3HEAD)`, `attributes(5)`, `standards(5)`

Name carg, cargf, cargl – complex argument functions

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <complex.h>

double carg(double complex z);

float cargf(float complex z);

long double cargl(long double complex z);
```

Description These functions compute the argument (also called phase angle) of *z*, with a branch cut along the negative real axis.

Return Values These functions return the value of the argument in the interval [$-\pi$, $+\pi$].

Errors No errors are defined.

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

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

See Also cimag(3M), complex.h(3HEAD), conj(3M), cproj(3M), attributes(5), standards(5)

Name casin, casinf, casinl – complex arc sine functions

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <complex.h>

double complex casin(double complex z);

float complex casinf(float complex z);

long double complex casinl(long double complex z);
```

Description These functions compute the complex arc sine of *z*, with branch cuts outside the interval [−1, +1] along the real axis.

Return Values These functions return the complex arc sine value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval [$-\pi/2$, $+\pi/2$] along the real axis.

Errors No errors are defined.

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

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

See Also complex.h(3HEAD), csin(3M), attributes(5), standards(5)

Name casinh, casinhf, casinhl – complex arc hyperbolic sine functions

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <complex.h>

double complex casinh(double complex z);

float complex casinhf(float complex z);

long double complex casinhl(long double complex z);
```

Description These functions compute the complex arc hyperbolic sine of *z*, with branch cuts outside the interval [-*i*, +*i*] along the imaginary axis.

Return Values These functions return the complex arc hyperbolic sine value, in the range of a strip mathematically unbounded along the real axis and in the interval [$-i\pi/2$, $+i\pi/2$] along the imaginary axis.

Errors No errors are defined.

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

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

See Also complex.h(3HEAD), csinh(3M), attributes(5), standards(5)

Name catan, catanf, catanl – complex arc tangent functions

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <complex.h>

double complex catan(double complex z);

float complex catanf(float complex z);

long double complex catanl(long double complex z);
```

Description These functions compute the complex arc tangent of *z*, with branch cuts outside the interval [$-i$, $+\infty i$] along the imaginary axis.

Return Values These functions return the complex arc tangent value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval [$-\pi/2$, $+\pi/2$] along the real axis.

Errors No errors are defined.

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

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

See Also complex.h(3HEAD), ctan(3M), attributes(5), standards(5)

Name catanh, catanhf, catanhl – complex arc hyperbolic tangent functions

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <complex.h>

double complex catanh(double complex z);

float complex catanhf(float complex z);

long double complex catanhl(long double complex z);
```

Description These functions compute the complex arc hyperbolic tangent of z, with branch cuts outside the interval [−1, +1] along the real axis.

Return Values These functions return the complex arc hyperbolic tangent value, in the range of a strip mathematically unbounded along the real axis and in the interval [$-i\pi/2$, $+i\pi/2$] along the imaginary axis.

Errors No errors are defined.

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

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

See Also complex.h(3HEAD), ctanh(3M), attributes(5), standards(5)

Name cbrt, cbrtf, cbrtl – cube root functions

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double cbrt(double x);

float cbrtf(float x);

long double cbrtl(long double x);
```

Description These functions compute the real cube root of their argument *x*.

Return Values On successful completion, these functions return the cube root of *x*.

If *x* is NaN, a NaN is returned.

If *x* is ±0 or ±Inf, *x* is returned.

Errors No errors are defined.

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

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

See Also math.h(3HEAD), attributes(5), standards(5)

Name ccos, ccosf, ccosl – complex cosine functions

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <complex.h>

double complex ccos(double complex z);

float complex ccosf(float complex z);

long double complex ccosl(long double complex z);
```

Description These functions compute the complex cosine of *z*.

Return Values These functions return the complex cosine value.

Errors No errors are defined.

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

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

See Also cacos(3M), complex.h(3HEAD), attributes(5), standards(5)

Name ccosh, ccoshf, ccoshl – complex hyperbolic cosine functions

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <complex.h>

double complex ccosh(double complex z);

float complex ccoshf(float complex z);

long double complex ccoshl(long double complex z);
```

Description These functions compute the complex hyperbolic cosine of *z*.

Return Values These functions return the complex hyperbolic cosine value.

Errors No errors are defined.

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

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

See Also cacosh(3M), complex.h(3HEAD), attributes(5), standards(5)

Name ceil, ceilf, ceill – ceiling value function

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double ceil(double x);

float ceilf(float x);

long double ceill(long double x);
```

Description These functions compute the smallest integral value not less than *x*.

Return Values Upon successful completion, the `ceil()`, `ceilf()`, and `ceill()` functions return the smallest integral value not less than *x*, expressed as a type `double`, `float`, or `long double`, respectively.

If *x* is NaN, a NaN is returned.

If *x* is ±0 or ±Inf, *x* is returned.

Usage The integral value returned by these functions need not be expressible as an `int` or `long int`. The return value should be tested before assigning it to an integer type to avoid the undefined results of an integer overflow.

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

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

See Also `feclearexcept(3M)`, `fetestexcept(3M)`, `floor(3M)`, `isnan(3M)`, `math.h(3HEAD)`, `attributes(5)`, `standards(5)`

Name cexp, cexpf, cexpl – complex exponential functions

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <complex.h>

double complex cexp(double complex z);

float complex cexpf(float complex z);

long double complex cexpl(long double complex z);
```

Description These functions compute the complex exponent of *z*, defined as e^*z*.

Return Values These functions return the complex exponential value of *z*.

Errors No errors are defined.

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

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

See Also clog(3M), complex.h(3HEAD), attributes(5), standards(5)

Name cimag, cimagf, cimagl – complex imaginary functions

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <complex.h>

double cimag(double complex z);

float cimagf(float complex z);

long double cimagl(long double complex z);
```

Description These functions compute the imaginary part of *z*.

Return Values These functions return the imaginary part value (as a real).

Errors No errors are defined.

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

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

See Also carg(3M), complex.h(3HEAD), conj(3M), cproj(3M), creal(3M), attributes(5), standards(5)

Name clog, clogf, clogl – complex natural logarithm functions

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <complex.h>

double complex clog(double complex z);

float complex clogf(float complex z);

long double complex clogl(long double complex z);
```

Description These functions compute the complex natural (base *e*) logarithm of *z*, with a branch cut along the negative real axis.

Return Values These functions return the complex natural logarithm value, in the range of a strip mathematically unbounded along the real axis and in the interval [$-i$, $+i$] along the imaginary axis.

Errors No errors are defined.

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

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

See Also cexp(3M), complex.h(3HEAD), attributes(5), standards(5)

Name config_admin, config_change_state, config_private_func, config_test, config_stat, config_list, config_list_ext, config_ap_id_cmp, config_unload_libs, config_strerror – configuration administration interface

Synopsis

```
cc [ flag... ] file... -lcfgadm [ library... ]
#include <config_admin.h>
#include <sys/param.h>

cfga_err_t config_change_state(cfga_cmd_t state_change_cmd,
     int num_ap_ids, char * const *ap_ids, const char *options,
     struct cfga_confirm *confp, struct cfga_msg *msgp,
     char **errstring, cfga_flags_t flags);

cfga_err_t config_private_func(const char *function, int num_ap_ids,
     char * const *ap_ids, const char *options,
     struct cfga_confirm *confp, msgp, char **errstring,
     cfga_flags_t flags);

cfga_err_t config_test(int num_ap_ids, char * const *ap_ids,
     const char *options, struct cfga_msg *msgp,
     char **errstring, cfga_flags_t flags);

cfga_err_t config_list_ext(int num_ap_ids, char * const *ap_ids,
     struct cfga_list_data **ap_id_list, int *nlist,
     const char *options, const char *listops,
     char **errstring, cfga_flags_t flags);

int config_ap_id_cmp(const cfga_ap_id_t ap_id1,
     const cfga_ap_id_t ap_id2);

void config_unload_libs(void);

const char *config_strerror(cfga_err_t cfgerrnum);
```

Deprecated Interfaces The following interfaces have been deprecated and their use is strongly discouraged:

```
cfga_err_t config_stat(int num_ap_ids, char * const *ap_ids,
     struct cfga_stat_data *buf, const char *options, char **errstring);

cfga_err_t config_list(struct cfga_stat_data **ap_id_list,
      int *nlist, const char *options, char **errstring);
```

Hardware Dependent Library Synopsis The `config_admin` library is a generic interface that is used for dynamic configuration, (DR). Each piece of hardware that supports DR must supply a hardware-specific *plugin* library that contains the entry points listed in this subsection. The generic library will locate and link to the appropriate library to effect DR operations. The interfaces specified in this subsection are really "hidden" from users of the generic libraries. It is, however, necessary that writers of the hardware-specific plug in libraries know what these interfaces are.

```
cfga_err_t cfga_change_state(cfga_cmd_t state_change_cmd,
     const char *ap_id, const char *options, struct cfga_confirm *confp,
     struct cfga_msg *msgp, char **errstring, cfga_flags_t flags);
```

```
cfga_err_t cfga_private_func(const char *function,
      const char *ap_id, const char *options, struct cfga_confirm *confp,
      struct cfga_msg *msgp, char **errstring, cfga_flags_t flags);

cfga_err_t cfga_test(const char *ap_id, const char *options,
      struct cfga_msg *msgp, char **errstring, cfga_flags_t flags);

cfga_err_t cfga_list_ext(const char *ap_id,
      struct cfga_list_data **ap_id_list, nlist, const char *options,
      const char *listopts, char **errstring, cfga_flags_t flags);

cfga_err_t cfga_help(struct cfga_msg *msgp, const char *options,
      cfga_flags_t flags);

int cfga_ap_id_cmp(const cfga_ap_id_t ap_id1, const cfga_ap_id_t ap_id2);
```

Deprecated Interfaces

The following interfaces have been deprecated and their use is strongly discouraged:

```
cfga_err_t cfga_stat(const char *ap_id, struct cfga_stat_data *buf,
     const char *options, char **errstring);

cfga_err_t cfga_list(const char *ap_id,
     struct cfga_stat_data **ap_id_list, int *nlist, const char *options,
     char **errstring);
```

Description

The `config_*()` functions provide a hardware independent interface to hardware-specific system configuration administration functions. The `cfga_*()` functions are provided by hardware-specific libraries that are dynamically loaded to handle configuration administration functions in a hardware-specific manner.

The `libcfgadm` library is used to provide the services of the `cfgadm(1M)` command. The hardware-specific libraries are located in `/usr/platform/${machine}/lib/cfgadm`, `/usr/platform/${arch}/lib/cfgadm`, and `/usr/lib/cfgadm`. The hardware-specific library names are derived from the driver name or from class names in device tree nodes that identify attachment points.

The `config_change_state()` function performs operations that change the state of the system configuration. The *state_change_cmd* argument can be one of the following: `CFGA_CMD_INSERT`, `CFGA_CMD_REMOVE`, `CFGA_CMD_DISCONNECT`, `CFGA_CMD_CONNECT`, `CFGA_CMD_CONFIGURE`, or `CFGA_CMD_UNCONFIGURE`. The *state_change_cmd* `CFGA_CMD_INSERT` is used to prepare for manual insertion or to activate automatic hardware insertion of an occupant. The *state_change_cmd* `CFGA_CMD_REMOVE` is used to prepare for manual removal or activate automatic hardware removal of an occupant. The *state_change_cmd* `CFGA_CMD_DISCONNECT` is used to disable normal communication to or from an occupant in a receptacle. The *state_change_cmd* `CFGA_CMD_CONNECT` is used to enable communication to or from an occupant in a receptacle. The *state_change_cmd* `CFGA_CMD_CONFIGURE` is used to bring the hardware resources contained on, or attached to, an occupant into the realm of Solaris, allowing use of the occupant's hardware resources by the system. The *state_change_cmd* `CFGA_CMD_UNCONFIGURE` is used to remove the hardware resources

contained on, or attached to, an occupant from the realm of Solaris, disallowing further use of the occupant's hardware resources by the system.

The *flags* argument may contain one or both of the defined flags, `CFGA_FLAG_FORCE` and `CFGA_FLAG_VERBOSE`. If the `CFGA_FLAG_FORCE` flag is asserted certain safety checks will be overridden. For example, this may not allow an occupant in the failed condition to be configured, but might allow an occupant in the failing condition to be configured. Acceptance of a force is hardware dependent. If the `CFGA_FLAG_VERBOSE` flag is asserted hardware-specific details relating to the operation are output utilizing the `cfga_msg` mechanism.

The `config_private_func()` function invokes private hardware-specific functions.

The `config_test()` function is used to initiate testing of the specified attachment point.

The *num_ap_ids* argument specifies the number of *ap_id*s in the *ap_ids* array. The *ap_ids* argument points to an array of *ap_id*s.

The *ap_id* argument points to a single *ap_id*.

The *function* and *options* strings conform to the `getsubopt(3C)` syntax convention and are used to supply hardware-specific function or option information. No generic hardware-independent functions or options are defined.

The `cfga_confirm` structure referenced by *confp* provides a call-back interface to get permission to proceed should the requested operation require, for example, a noticeable service interruption. The `cfga_confirm` structure includes the following members:

```
int  (*confirm)(void *appdata_ptr, const char *message);
void *appdata_ptr;
```

The `confirm()` function is called with two arguments: the generic pointer *appdata_ptr* and the message detailing what requires confirmation. The generic pointer *appdata_ptr* is set to the value passed in in the `cfga_confirm` structure member `appdata_ptr` and can be used in a graphical user interface to relate the `confirm` function call to the `config_*()` call. The *confirm*() function should return 1 to allow the operation to proceed and 0 otherwise.

The `cfga_msg` structure referenced by *msgp* provides a call-back interface to output messages from a hardware-specific library. In the presence of the `CFGA_FLAG_VERBOSE` flag, these messages can be informational; otherwise they are restricted to error messages. The `cfga_msg` structure includes the following members:

```
int (*message_routine)(void *appdata_ptr, const char *message);
void *appdata_ptr;
```

The `message_routine()` function is called with two arguments: the generic pointer *appdata_ptr* and the message. The generic pointer *appdata_ptr* is set to the value passed in in the `cfga_confirm` structure member `appdata_ptr` and can be used in a graphical user interface to relate the `message_routine()` function call to the `config_*()` call. The messages must be in the native language specified by the `LC_MESSAGES` locale category; see `setlocale(3C)`.

For some generic errors a hardware-specific error message can be returned. The storage for the error message string, including the terminating null character, is allocated by the `config_*` functions using `malloc(3C)` and a pointer to this storage returned through *errstring*. If *errstring* is `NULL` no error message will be generated or returned. If *errstring* is not `NULL` and no error message is generated, the pointer referenced by *errstring* will be set to `NULL`. It is the responsibility of the function calling `config_*()` to deallocate the returned storage using `free(3C)`. The error messages must be in the native language specified by the `LC_MESSAGES` locale category; see `setlocale(3C)`.

The `config_list_ext()` function provides the listing interface. When supplied with a list of *ap_id*s through the first two arguments, it returns an array of `cfga_list_data_t` structures for each attachment point specified. If the first two arguments are 0 and `NULL` respectively, then all attachment points in the device tree will be listed. Additionally, dynamic expansion of an attachment point to list dynamic attachment points may also be requested by passing the `CFGA_FLAG_LIST_ALL` flag through the *flags* argument. Storage for the returned array of `stat` structures is allocated by the `config_list_ext()` function using `malloc(3C)`. This storage must be freed by the caller of `config_list_ext()` by using `free(3C)`.

The `cfga_list_data` structure includes the following members:

```
cfga_log_ext_t     ap_log_id;        /* Attachment point logical id */
cfga_phys_ext_t    ap_phys_id;       /* Attachment point physical id */
cfga_class_t       ap_class;         /* Attachment point class */
cfga_stat_t        ap_r_state;       /* Receptacle state */
cfga_stat_t        ap_o_state;       /* Occupant state */
cfga_cond_t        ap_cond;          /* Attachment point condition */
cfga_busy_t        ap_busy;          /* Busy indicator */
time_t             ap_status_time;   /* Attachment point last change*/
cfga_info_t        ap_info;          /* Miscellaneous information */
cfga_type_t        ap_type;          /* Occupant type */
```

The types are defined as follows:

```
typedef char cfga_log_ext_t[CFGA_LOG_EXT_LEN];
typedef char cfga_phys_ext_t[CFGA_PHYS_EXT_LEN];
typedef char cfga_class_t[CFGA_CLASS_LEN];
typedef char cfga_info_t[CFGA_INFO_LEN];
typedef char cfga_type_t[CFGA_TYPE_LEN];
typedef enum cfga_cond_t;
typedef enum cfga_stat_t;
typedef int  cfga_busy_t;
typedef int cfga_flags_t;
```

The *listopts* argument to `config_list_ext()` conforms to the `getsubopt(3C)` syntax and is used to pass listing sub-options. Currently, only the sub-option *class*=`class_name` is supported. This list option restricts the listing to attachment points of class `class_name`.

The *listopts* argument to `cfga_list_ext()` is reserved for future use. Hardware-specific libraries should ignore this argument if it is `NULL`. If *listopts* is not `NULL` and is not supported by the hardware-specific library, an appropriate error code should be returned.

The `ap_log_id` and the `ap_phys_id` members give the hardware-specific logical and physical names of the attachment point. The `ap_busy` memberd indicates activity is present that may result in changes to state or condition. The `ap_status_time` member provides the time at which either the `ap_r_state`, `ap_o_state`, or `ap_cond` field of the attachment point last changed. The `ap_info` member is available for the hardware-specific code to provide additional information about the attachment point. The `ap_class` member contains the attachment point class (if any) for an attachment point. The `ap_class` member is filled in by the generic library. If the `ap_log_id` and `ap_phys_id` members are not filled in by the hardware-specific library, the generic library will fill in these members using a generic format. The remaining members are the responsibility of the corresponding hardware-tospecific library.

All string members in the `cfga_list_data` structure are null-terminated.

The `config_stat()`, `config_list()`, `cfga_stat()`, and `cfga_list()` functions and the `cfga_stat_data` data structure are deprecated interfaces and are provided solely for backward compatibility. Use of these interfaces is strongly discouraged.

The `config_ap_id_cmp` function performs a hardware dependent comparison on two *ap_id*s, returning an equal to, less than or greater than indication in the manner of `strcmp(3C)`. Each argument is either a `cfga_ap_id_t` or can be a null-terminated string. This function can be used when sorting lists of *ap_id*s, for example with `qsort(3C)`, or when selecting entries from the result of a `config_list` function call.

The `config_unload_libs` function unlinks all previously loaded hardware-specific libraries.

The `config_strerror` function can be used to map an error return value to an error message string. See `RETURN VALUES`. The returned string should not be overwritten. `config_strerror` returns `NULL` if *cfgerrnum* is out-of-range.

The `cfga_help` function can be used request that a hardware-specific library output it's localized help message.

Return Values The `config_*()` and `cfga_*()` functions return the following values. Additional error information may be returned through *errstring* if the return code is not `CFGA_OK`. See `DESCRIPTION` for details.

`CFGA_BUSY`	The command was not completed due to an element of the system configuration administration system being busy.
`CFGA_ATTR_INVAL`	No attachment points with the specified attributes exists

`CFGA_ERROR`	An error occurred during the processing of the requested operation. This error code includes validation of the command arguments by the hardware-specific code.
`CFGA_INSUFFICIENT_CONDITION`	Operation failed due to attachment point condition.
`CFGA_INVAL`	The system configuration administration operation requested is not supported on the specified attachment point.
`CFGA_LIB_ERROR`	A procedural error occurred in the library, including failure to obtain process resources such as memory and file descriptors.
`CFGA_NACK`	The command was not completed due to a negative acknowledgement from the *confp*`->confirm` function.
`CFGA_NO_LIB`	A hardware-specific library could not be located using the supplied *ap_id*.
`CFGA_NOTSUPP`	System configuration administration is not supported on the specified attachment point.
`CFGA_OK`	The command completed as requested.
`CFGA_OPNOTSUPP`	System configuration administration operation is not supported on this attachment point.
`CFGA_PRIV`	The caller does not have the required process privileges. For example, if configuration administration is performed through a device driver, the permissions on the device node would be used to control access.
`CFGA_SYSTEM_BUSY`	The command required a service interruption and was not completed due to a part of the system that could not be quiesced.

Errors Many of the errors returned by the system configuration administration functions are hardware-specific. The strings returned in *errstring* may include the following:

`attachment point` *ap_id* `not known`
The attachment point detailed in the error message does not exist.

`unknown hardware option` *option* `for`*operation*
An unknown option was encountered in the *options* string.

`hardware option` *option* `requires a value`
An option in the *options* string should have been of the form *option=value*.

`listing option` *list_option* `requires a value`
An option in the listopts string should have been of the form *option*=`value`.

`hardware option` *option* `does not require a value`
An option in the *options* string should have been a simple option.

`attachment point` *ap_id* `is not configured`
A *config_change_state* command to `CFGA_CMD_UNCONFIGURE` an occupant was made to an attachment point whose occupant was not in the `CFGA_STAT_CONFIGURED` state.

`attachment point` *ap_id* `is not unconfigured`
A *config_change_state* command requiring an unconfigured occupant was made to an attachment point whose occupant was not in the `CFGA_STAT_UNCONFIGURED` state.

`attachment point` *ap_id* `condition not satisfactory`
A *config_change_state* command was made to an attachment point whose condition prevented the operation.

`attachment point` *ap_id* `in condition` *condition* `cannot be used`
A *config_change_state* operation with force indicated was directed to an attachment point whose condition fails the hardware dependent test.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcsu, SUNWkvm
MT-Level	Safe

See Also `cfgadm(1M)`, `devinfo(1M)`, `dlopen(3C)`, `dlsym(3C)`, `free(3C)`, `getsubopt(3C)`, `malloc(3C)`, `qsort(3C)`, `setlocale(3C)`, `strcmp(3C)`, `libcfgadm(3LIB)`, `attributes(5)`

Notes Applications using this library should be aware that the underlying implementation may use system services which alter the contents of the external variable `errno` and may use file descriptor resources.

The following code shows the intended error processing when `config_*()` returns a value other than `CFGA_OK`:

```
void
emit_error(cfga_err_t cfgerrnum, char *estrp)
{
    const char *ep;
    ep = config_strerror(cfgerrnum);
    if (ep == NULL)
        ep = gettext("configuration administration unknown error");
    if (estrp != NULL && *estrp != '\0') {
        (void) fprintf(stderr, "%s: %s\n", ep, estrp);
```

```
    } else {
        (void) fprintf(stderr, "%s\n", ep);
    }
    if (estrp != NULL)
        free((void *)estrp);
}
```

Reference should be made to the Hardware Specific Guide for details of System Configuration Administration support.

Name conj, conjf, conjl – complex conjugate functions

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <complex.h>

double complex conj(double complex z);

float complex conjf(float complex z);

long double complex conjl(long double complex z);
```

Description These functions compute the complex conjugate of z, by reversing the sign of its imaginary part.

Return Values These functions return the complex conjugate value.

Errors No errors are defined.

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

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

See Also carg(3M), cimag(3M), complex.h(3HEAD), cproj(3M), creal(3M), attributes(5), standards(5)

Name ConnectToServer – connect to a DMI service provider

Synopsis

```
cc [ flag ... ] file ... -ldmici  -ldmimi    [ library ... ]
#include <dmi/api.hh>

bool_t ConnectToServer(ConnectI *argp, DmiRpcHandle *dmi_rpc_handle);
```

Description The `ConnectToServer()` function enables a management application or a component instrumentation to connect to a DMI service provider.

The *argp* parameter is an input parameter that uses the following data structure:

```
struct ConnectIN {
        char        *host;
        const char  *nettype;
        ServerType  servertype;
        RpcType     rpctype;
}
```

The `host` member indicates the host on which the service provider is running. The default is *localhost*.

The `nettype` member specifies the type of transport RPC uses. The default is *netpath*.

The `servertype` member indicates whether the connecting process is a management application or a component instrumentation.

The `rpctype` member specifies the type of RPC, either ONC or DCE. Only ONC is supported in the Solaris 7 release.

The *dmi_rpc_handle* parameter is the output parameter that returns DMI RPC handle.

Return Values The `ConnectToServer()` function returns TRUE if successful, otherwise FALSE.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-level	Safe

See Also `DisconnectToServer(3DMI)`,`attributes(5)`

Name copylist – copy a file into memory

Synopsis

```
cc [ flag ... ] file ... -lgen [ library ... ]
#include <libgen.h>

char *copylist(const char *filenm, off_t *szptr);
```

Description The `copylist()` function copies a list of items from a file into freshly allocated memory, replacing new-lines with null characters. It expects two arguments: a pointer *filenm* to the name of the file to be copied, and a pointer *szptr* to a variable where the size of the file will be stored.

Upon success, `copylist()` returns a pointer to the memory allocated. Otherwise it returns NULL if it has trouble finding the file, calling `malloc()`, or reading the file.

Usage The `copylist()` function has a transitional interface for 64-bit file offsets. See `lf64(5)`.

Examples **EXAMPLE 1** Example of `copylist()` function.

```
/* read "file" into buf */
off_t size;
char *buf;
buf = copylist("file", &size);
if (buf) {
    for (i=0; i<size; i++)
        if (buf[i])
            putchar(buf[i]);
        else
            putchar('\n');
    }
} else {
    fprintf(stderr, "%s: Copy failed for "file".\n", argv[0]);
    exit (1);
}
```

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `malloc(3C)`, `attributes(5)`, `lf64(5)`

Notes When compiling multithreaded applications, the `_REENTRANT` flag must be defined on the compile line. This flag should only be used in multithreaded applications.

Name copysign, copysignf, copysignl – number manipulation function

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double copysign(double x, double y);

float copysignf(float x, float y);

long double copysignl(long double x, long double y);
```

Description These functions produce a value with the magnitude of *x* and the sign of *y*.

Return Values Upon successful completion, these functions return a value with the magnitude of *x* and the sign of *y*.

Errors No errors are defined.

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

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

See Also math.h(3HEAD), signbit(3M),attributes(5), standards(5)

Name cos, cosf, cosl – cosine function

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double cos(double x);

float cosf(float x);

long double cosl(long double x);
```

Description These functions compute the cosine of *x*, measured in radians.

Return Values Upon successful completion, these functions return the cosine of *x*.

If *x* is NaN, NaN is returned.

If *x* is +0, 1.0 is returned.

If *x* is ±Inf, a domain error occurs and a NaN is returned.

Errors These functions will fail if:

Domain Error — The *x* argument is ±Inf.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, the invalid floating-point exception is raised.

Usage An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. An application should either examine the return value or check the floating point exception flags to detect exceptions.

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

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

See Also `acos(3M)`, `feclearexcept(3M)`, `fetestexcept(3M)`, `isnan(3M)`, `math.h(3HEAD)`, `sin(3M)`, `tan(3M)`, `attributes(5)`, `standards(5)`

Name cosh, coshf, coshl – hyperbolic cosine function

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double cosh(double x);

float coshf(float x);

long double coshl(long double x);
```

Description These functions compute the hyperbolic cosine of their argument *x*.

Return Values Upon successful completion, these functions return the hyperbolic cosine of *x*.

If the correct value would cause overflow, a range error occurs and `cosh()`, `coshf()`, and `coshl()` return the value of the macro `HUGE_VAL`, `HUGE_VALF`, and `HUGE_VALL`, respectively.

If *x* is NaN, a NaN is returned.

If *x* is ±0, 1.0 is returned.

If *x* is ±Inf, ±Inf is returned.

For exceptional cases, `matherr(3M)` tabulates the values to be returned by `cosh()` as specified by SVID3 and XPG3.

Errors These functions will fail if:

Range Error — The result would cause an overflow.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, the overflow floating-point exception is raised.

The `cosh()` function sets `errno` to `ERANGE` if the result would cause an overflow.

Usage An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. An application should either examine the return value or check the floating point exception flags to detect exceptions.

An application can also set `errno` to 0 before calling `cosh()`. On return, if `errno` is non-zero, an error has occurred. The `coshf()` and `coshl()` functions do not set `errno`.

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

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

See Also acosh(3M), feclearexcept(3M), fetestexcept(3M), isnan(3M), math.h(3HEAD), matherr(3M), sinh(3M), tanh(3M), attributes(5), standards(5)

Name cpc – hardware performance counters

Description Modern microprocessors contain *hardware performance counters* that allow the measurement of many different hardware events related to CPU behavior, including instruction and data cache misses as well as various internal states of the processor. The counters can be configured to count user events, system events, or both. Data from the performance counters can be used to analyze and tune the behavior of software on a particular type of processor.

Most processors are able to generate an interrupt on counter overflow, allowing the counters to be used for various forms of profiling.

This manual page describes a set of APIs that allow Solaris applications to use these counters. Applications can measure their own behavior, the behavior of other applications, or the behavior of the whole system.

Shared Counters or Private Counters There are two principal models for using these performance counters. Some users of these statistics want to observe system-wide behavior. Other users want to view the performance counters as part of the register set exported by each LWP. On a machine performing more than one activity, these two models are in conflict because the counters represent a critical hardware resource that cannot simultaneously be both shared and private.

Configuration Interfaces The following configuration interfaces are provided:

`cpc_open`(3CPC)	Check the version the application was compiled with against the version of the library.
`cpc_cciname`(3CPC)	Return a printable string to describe the performance counters of the processor.
`cpc_npic`(3CPC)	Return the number of performance counters on the processor.
`cpc_cpuref`(3CPC)	Return a reference to documentation that should be consulted to understand how to use and interpret data from the performance counters.

Performance Counter Access Performance counters can be present in hardware but not acccessible because either some of the necessary system software components are not available or not installed, or the counters might be in use by other processes. The `cpc_open`(3CPC) function determines the accessibility of the counters and must be invoked before any attempt to program the counters.

Finding Events Each different type of processor has its own set of events available for measurement. The `cpc_walk_events_all`(3CPC) and `cpc_walk_events_pic`(3CPC) functions allow an application to determine the names of events supported by the underlying processor.

Using Attributes Some processors have advanced performance counter capabilities that are configured with attributes. The `cpc_walk_attrs`(3CPC) function can be used to determine the names of attributes supported by the underlying processor. The documentation referenced by `cpc_cpuref`(3CPC) should be consulted to understand the meaning of a processor's performance counter attributes.

Performance Counter Context

Each processor on the system possesses its own set of performance counter registers. For a single process, it is often desirable to maintain the illusion that the counters are an intrinsic part of that process (whichever processors it runs on), since this allows the events to be directly attributed to the process without having to make passive all other activity on the system.

To achieve this behavior, the library associates *performance counter context* with each LWP in the process. The context consists of a small amount of kernel memory to hold the counter values when the LWP is not running, and some simple kernel functions to save and restore those counter values from and to the hardware registers when the LWP performs a normal context switch. A process can only observe and manipulate its own copy of the performance counter control and data registers.

Performance Counters In Other Processes

Though applications can be modified to instrument themselves as demonstrated above, it is frequently useful to be able to examine the behavior of an existing application without changing the source code. A separate library, `libpctx`, provides a simple set of interfaces that use the facilities of `proc(4)` to control a target process, and together with functions in `libcpc`, allow `truss`-like tools to be constructed to measure the performance counters in other applications. An example of one such application is `cputrack(1)`.

The functions in `libpctx` are independent of those in `libcpc`. These functions manage a process using an event-loop paradigm — that is, the execution of certain system calls by the controlled process cause the library to stop the controlled process and execute callback functions in the context of the controlling process. These handlers can perform various operations on the target process using APIs in `libpctx` and `libcpc` that consume `pctx_t` handles.

See Also `cputrack(1)`, `cpustat(1M)`, `cpc_bind_curlwp(3CPC)`, `cpc_buf_create(3CPC)`, `cpc_enable(3CPC)`, `cpc_npic(3CPC)`, `cpc_open(3CPC)`, `cpc_set_create(3CPC)`, `cpc_seterrhndlr(3CPC)`, `libcpc(3LIB)`, `pctx_capture(3CPC)`, `pctx_set_events(3CPC)`, `proc(4)`.

Name cpc_access – test access CPU performance counters

Synopsis

```
cc [ flag... ] file... -lcpc [ library... ]
#include <libcpc.h>

int cpc_access(void);
```

Description Access to CPU performance counters is possible only on systems where the appropriate hardware exists and is correctly configured. The `cpc_access()` function *must* be used to determine if the hardware exists and is accessible on the platform before any of the interfaces that use the counters are invoked.

When the hardware is available, access to the per-process counters is always allowed to the process itself, and allowed to other processes mediated using the existing security mechanisms of `/proc`.

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

By default, two common `errno` values are decoded and cause the library to print an error message using its reporting mechanism. See `cpc_seterrfn(3CPC)` for a description of how this behavior can be modified.

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

`EAGAIN` Another process may be sampling system-wide CPU statistics.

`ENOSYS` CPU performance counters are inaccessible on this machine. This error can occur when the machine supports CPU performance counters, but some software components are missing. Check to see that all CPU Performance Counter packages have been correctly installed.

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

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

See Also `cpc(3CPC)`, `cpc_open(3CPC)`, `cpc_seterrfn(3CPC)`, `libcpc(3LIB)`, `proc(4)`, `attributes(5)`

Notes The `cpc_access()` function exists for binary compatibility only. Source containing this function will not compile. This function is obsolete and might be removed in a future release. Applications should use `cpc_open(3CPC)` instead.

Name cpc_bind_curlwp, cpc_bind_pctx, cpc_bind_cpu, cpc_unbind, cpc_request_preset, cpc_set_restart – bind request sets to hardware counters

Synopsis

```
cc [ flag... ] file... -lcpc [ library... ]
#include <libcpc.h>

int cpc_bind_curlwp(cpc_t *cpc, cpc_set_t *set, uint_t flags);

int cpc_bind_pctx(cpc_t *cpc, pctx_t *pctx, id_t id, cpc_set_t *set,
     uint_t flags);

int cpc_bind_cpu(cpc_t *cpc, processorid_t id, cpc_set_t *set,
     uint_t flags);

int cpc_unbind(cpc_t *cpc, cpc_set_t *set);

int cpc_request_preset(cpc_t *cpc, int index, uint64_t preset);

int cpc_set_restart(cpc_t *cpc, cpc_set_t *set);
```

Description These functions program the processor's hardware counters according to the requests contained in the *set* argument. If these functions are successful, then upon return the physical counters will have been assigned to count events on behalf of each request in the set, and each counter will be enabled as configured.

The `cpc_bind_curlwp()` function binds the set to the calling LWP. If successful, a performance counter context is associated with the LWP that allows the system to virtualize the hardware counters to that specific LWP.

By default, the system binds the set to the current LWP only. If the `CPC_BIND_LWP_INHERIT` flag is present in the *flags* argument, however, any subsequent LWPs created by the current LWP will inherit a copy of the request set. The newly created LWP will have its virtualized 64-bit counters initialized to the preset values specified in *set*, and the counters will be enabled and begin counting events on behalf of the new LWP. This automatic inheritance behavior can be useful when dealing with multithreaded programs to determine aggregate statistics for the program as a whole.

If the `CPC_BIND_LWP_INHERIT` flag is specified and any of the requests in the set have the `CPC_OVF_NOTIFY_EMT` flag set, the process will immediately dispatch a `SIGEMT` signal to the freshly created LWP so that it can preset its counters appropriately on the new LWP. This initialization condition can be detected using `cpc_set_sample(3CPC)` and looking at the counter value for any requests with `CPC_OVF_NOTIFY_EMT` set. The value of any such counters will be `UINT64_MAX`.

The `cpc_bind_pctx()` function binds the set to the LWP specified by the *pctx*-*id* pair, where *pctx* refers to a handle returned from `libpctx` and *id* is the ID of the desired LWP in the target process. If successful, a performance counter context is associated with the specified LWP and the system virtualizes the hardware counters to that specific LWP. The *flags* argument is reserved for future use and must always be `0`.

The `cpc_bind_cpu()` function binds the set to the specified CPU and measures events occurring on that CPU regardless of which LWP is running. Only one such binding can be active on the specified CPU at a time. As long as any application has bound a set to a CPU, per-LWP counters are unavailable and any attempt to use either `cpc_bind_curlwp()` or `cpc_bind_pctx()` returns `EAGAIN`. The first invocation of `cpc_bind_cpu()` invalidates all currently bound per-LWP counter sets, and any attempt to sample an invalidated set returns `EAGAIN`. To bind to a CPU, the library binds the calling LWP to the measured CPU with `processor_bind(2)`. The application must not change its processor binding until after it has unbound the set with `cpc_unbind()`. The *flags* argument is reserved for future use and must always be `0`.

The `cpc_request_preset()` function updates the preset and current value stored in the indexed request within the currently bound set, thereby changing the starting value for the specified request for the calling LWP only, which takes effect at the next call to `cpc_set_restart()`.

When a performance counter counting on behalf of a request with the `CPC_OVF_NOTIFY_EMT` flag set overflows, the performance counters are frozen and the LWP to which the set is bound receives a `SIGEMT` signal. The `cpc_set_restart()` function can be called from a `SIGEMT` signal handler function to quickly restart the hardware counters. Counting begins from each request's original preset (see `cpc_set_add_request(3CPC)`), or from the preset specified in a prior call to `cpc_request_preset()`. Applications performing performance counter overflow profiling should use the `cpc_set_restart()` function to quickly restart counting after receiving a `SIGEMT` overflow signal and recording any relevant program state.

The `cpc_unbind()` function unbinds the set from the resource to which it is bound. All hardware resources associated with the bound set are freed and if the set was bound to a CPU, the calling LWP is unbound from the corresponding CPU. See `processor_bind(2)`.

Return Values Upon successful completion these functions return 0. Otherwise, -1 is returned and `errno` is set to indicate the error.

Errors Applications wanting to get detailed error values should register an error handler with `cpc_seterrhndlr(3CPC)`. Otherwise, the library will output a specific error description to `stderr`.

These functions will fail if:

`EACCES` For `cpc_bind_curlwp()`, the system has Pentium 4 processors with HyperThreading and at least one physical processor has more than one hardware thread online. See NOTES.

For `cpc_bind_cpu()`, the process does not have the *cpc_cpu* privilege to access the CPU's counters.

For `cpc_bind_curlwp()`, `cpc_bind_cpc()`, and `cpc_bind_pctx()`, access to the requested hypervisor event was denied.

EAGAIN	For `cpc_bind_curlwp()` and `cpc_bind_pctx()`, the performance counters are not available for use by the application.
	For `cpc_bind_cpu()`, another process has already bound to this CPU. Only one process is allowed to bind to a CPU at a time and only one set can be bound to a CPU at a time.
EINVAL	The set does not contain any requests or `cpc_set_add_request()` was not called.
	The value given for an attribute of a request is out of range.
	The system could not assign a physical counter to each request in the system. See NOTES.
	One or more requests in the set conflict and might not be programmed simultaneously.
	The *set* was not created with the same *cpc* handle.
	For `cpc_bind_cpu()`, the specified processor does not exist.
	For `cpc_unbind()`, the set is not bound.
	For `cpc_request_preset()` and `cpc_set_restart()`, the calling LWP does not have a bound set.
ENOSYS	For `cpc_bind_cpu()`, the specified processor is not online.
ENOTSUP	The `cpc_bind_curlwp()` function was called with the `CPC_OVF_NOTIFY_EMT` flag, but the underlying processor is not capable of detecting counter overflow.
ESRCH	For `cpc_bind_pctx()`, the specified LWP in the target process does not exist.

Examples

EXAMPLE 1 Use hardware performance counters to measure events in a process.

The following example demonstrates how a standalone application can be instrumented with the `libcpc(3LIB)` functions to use hardware performance counters to measure events in a process. The application performs 20 iterations of a computation, measuring the counter values for each iteration. By default, the example makes use of two counters to measure external cache references and external cache hits. These options are only appropriate for UltraSPARC processors. By setting the EVENT0 and EVENT1 environment variables to other strings (a list of which can be obtained from the `-h` option of the `cpustat(1M)` or `cputrack(1)` utilities), other events can be counted. The `error()` routine is assumed to be a user-provided routine analogous to the familiar `printf(3C)` function from the C library that also performs an `exit(2)` after printing the message.

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

EXAMPLE 1 Use hardware performance counters to measure events in a process. *(Continued)*

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

int
main(int argc, char *argv[])
{
int iter;
char *event0 = NULL, *event1 = NULL;
cpc_t *cpc;
cpc_set_t *set;
cpc_buf_t *diff, *after, *before;
int ind0, ind1;
uint64_t val0, val1;

if ((cpc = cpc_open(CPC_VER_CURRENT)) == NULL)
        error("perf counters unavailable: %s", strerror(errno));

if ((event0 = getenv("EVENT0")) == NULL)
     event0 = "EC_ref";
if ((event1 = getenv("EVENT1")) == NULL)
     event1 = "EC_hit";

if ((set = cpc_set_create(cpc)) == NULL)
        error("could not create set: %s", strerror(errno));

if ((ind0 = cpc_set_add_request(cpc, set, event0, 0, CPC_COUNT_USER, 0,
        NULL)) == -1)
        error("could not add first request: %s", strerror(errno));

if ((ind1 = cpc_set_add_request(cpc, set, event1, 0, CPC_COUNT_USER, 0,
        NULL)) == -1)
        error("could not add first request: %s", strerror(errno));

if ((diff = cpc_buf_create(cpc, set)) == NULL)
        error("could not create buffer: %s", strerror(errno));
if ((after = cpc_buf_create(cpc, set)) == NULL)
        error("could not create buffer: %s", strerror(errno));
if ((before = cpc_buf_create(cpc, set)) == NULL)
        error("could not create buffer: %s", strerror(errno));

if (cpc_bind_curlwp(cpc, set, 0) == -1)
         error("cannot bind lwp%d: %s", _lwp_self(), strerror(errno));

for (iter = 1; iter <= 20; iter++) {
```

EXAMPLE 1 Use hardware performance counters to measure events in a process. *(Continued)*

```
        if (cpc_set_sample(cpc, set, before) == -1)
             break;

         /* ==> Computation to be measured goes here <== */

        if (cpc_set_sample(cpc, set, after) == -1)
             break;

        cpc_buf_sub(cpc, diff, after, before);
        cpc_buf_get(cpc, diff, ind0, &val0);
        cpc_buf_get(cpc, diff, ind1, &val1);

         (void) printf("%3d: %" PRId64 " %" PRId64 "\n", iter,
                val0, val1);
}

 if (iter != 21)
        error("cannot sample set: %s",  strerror(errno));

cpc_close(cpc);

return (0);
}
```

EXAMPLE 2 Write a signal handler to catch overflow signals.

The following example builds on Example 1 and demonstrates how to write the signal handler to catch overflow signals. A counter is preset so that it is 1000 counts short of overflowing. After 1000 counts the signal handler is invoked.

The signal handler:

```
cpc_t     *cpc;
cpc_set_t *set;
cpc_buf_t *buf;
int       index;

void
emt_handler(int sig, siginfo_t *sip, void *arg)
{
     ucontext_t *uap = arg;
     uint64_t val;

     if (sig != SIGEMT || sip->si_code != EMT_CPCOVF) {
         psignal(sig, "example");
```

EXAMPLE 2 Write a signal handler to catch overflow signals. *(Continued)*

```
        psiginfo(sip, "example");
        return;
    }

    (void) printf("lwp%d - si_addr %p ucontext: %%pc %p %%sp %p\n",
        _lwp_self(), (void *)sip->si_addr,
        (void *)uap->uc_mcontext.gregs[PC],
        (void *)uap->uc_mcontext.gregs[SP]);

    if (cpc_set_sample(cpc, set, buf) != 0)
        error("cannot sample: %s", strerror(errno));

    cpc_buf_get(cpc, buf, index, &val);

    (void) printf("0x%" PRIx64"\n", val);
    (void) fflush(stdout);

    /*
    * Update a request's preset and restart the counters. Counters which
    * have not been preset with cpc_request_preset() will resume counting
    * from their current value.
    */
    (cpc_request_preset(cpc, ind1, val1) != 0)
       error("cannot set preset for request %d: %s", ind1,
            strerror(errno));
       if (cpc_set_restart(cpc, set) != 0)
            error("cannot restart lwp%d: %s", _lwp_self(), strerror(errno));
}
```

The setup code, which can be positioned after the code that opens the CPC library and creates a set:

```
#define PRESET (UINT64_MAX - 999ull)

    struct sigaction act;
    ...
    act.sa_sigaction = emt_handler;
    bzero(&act.sa_mask, sizeof (act.sa_mask));
    act.sa_flags = SA_RESTART|SA_SIGINFO;
    if (sigaction(SIGEMT, &act, NULL) == -1)
        error("sigaction: %s", strerror(errno));

    if ((index = cpc_set_add_request(cpc, set, event, PRESET,
       CPC_COUNT_USER | CPC_OVF_NOTIFY_EMT, 0, NULL)) != 0)
       error("cannot add request to set: %s", strerror(errno));
```

EXAMPLE 2 Write a signal handler to catch overflow signals. *(Continued)*

```
if ((buf = cpc_buf_create(cpc, set)) == NULL)
   error("cannot create buffer: %s", strerror(errno));

if (cpc_bind_curlwp(cpc, set, 0) == -1)
    error("cannot bind lwp%d: %s", _lwp_self(), strerror(errno));

for (iter = 1; iter <= 20; iter++) {
    /* ==> Computation to be measured goes here <== */
}

cpc_unbind(cpc, set);      /* done */
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Safe

See Also

`cpustat(1M)`, `cputrack(1)`, `psrinfo(1M)`, `processor_bind(2)`, `cpc_seterrhndlr(3CPC)`, `cpc_set_sample(3CPC)`, `libcpc(3LIB)`, `attributes(5)`

Notes

When a set is bound, the system assigns a physical hardware counter to count on behalf of each request in the set. If such an assignment is not possible for all requests in the set, the bind function returns -1 and sets `errno` to `EINVAL`. The assignment of requests to counters depends on the capabilities of the available counters. Some processors (such as Pentium 4) have a complicated counter control mechanism that requires the reservation of limited hardware resources beyond the actual counters. It could occur that two requests for different events might be impossible to count at the same time due to these limited hardware resources. See the processor manual as referenced by `cpc_cpuref(3CPC)` for details about the underlying processor's capabilities and limitations.

Some processors can be configured to dispatch an interrupt when a physical counter overflows. The most obvious use for this facility is to ensure that the full 64-bit counter values are maintained without repeated sampling. Certain hardware, such as the UltraSPARC processor, does not record which counter overflowed. A more subtle use for this facility is to preset the counter to a value slightly less than the maximum value, then use the resulting interrupt to catch the counter overflow associated with that event. The overflow can then be used as an indication of the frequency of the occurrence of that event.

The interrupt generated by the processor might not be particularly precise. That is, the particular instruction that caused the counter overflow might be earlier in the instruction stream than is indicated by the program counter value in the ucontext.

When a request is added to a set with the CPC_OVF_NOTIFY_EMT flag set, then as before, the control registers and counter are preset from the 64-bit preset value given. When the flag is set, however, the kernel arranges to send the calling process a SIGEMT signal when the overflow occurs. The si_code member of the corresponding siginfo structure is set to EMT_CPCOVF and the si_addr member takes the program counter value at the time the overflow interrupt was delivered. Counting is disabled until the set is bound again.

If the CPC_CAP_OVERFLOW_PRECISE bit is set in the value returned by cpc_caps(3CPC), the processor is able to determine precisely which counter has overflowed after receiving the overflow interrupt. On such processors, the SIGEMT signal is sent only if a counter overflows and the request that the counter is counting has the CPC_OVF_NOTIFY_EMT flag set. If the capability is not present on the processor, the system sends a SIGEMT signal to the process if any of its requests have the CPC_OVF_NOTIFY_EMT flag set and any counter in its set overflows.

Different processors have different counter ranges available, though all processors supported by Solaris allow at least 31 bits to be specified as a counter preset value. Portable preset values lie in the range UINT64_MAX to UINT64_MAX-INT32_MAX.

The appropriate preset value will often need to be determined experimentally. Typically, this value will depend on the event being measured as well as the desire to minimize the impact of the act of measurement on the event being measured. Less frequent interrupts and samples lead to less perturbation of the system.

If the processor cannot detect counter overflow, bind will fail and return ENOTSUP. Only user events can be measured using this technique. See Example 2.

Pentium 4 Most Pentium 4 events require the specification of an event mask for counting. The event mask is specified with the *emask* attribute.

Pentium 4 processors with HyperThreading Technology have only one set of hardware counters per physical processor. To use cpc_bind_curlwp() or cpc_bind_pctx() to measure per-LWP events on a system with Pentium 4 HT processors, a system administrator must first take processors in the system offline until each physical processor has only one hardware thread online (See the -p option to psrinfo(1M)). If a second hardware thread is brought online, all per-LWP bound contexts will be invalidated and any attempt to sample or bind a CPC set will return EAGAIN.

Only one CPC set at a time can be bound to a physical processor with cpc_bind_cpu(). Any call to cpc_bind_cpu() that attempts to bind a set to a processor that shares a physical processor with a processor that already has a CPU-bound set returns an error.

To measure the shared state on a Pentium 4 processor with HyperThreading, the *count_sibling_usr* and *count_sibling_sys* attributes are provided for use with cpc_bind_cpu(). These attributes behave exactly as the CPC_COUNT_USER and CPC_COUNT_SYSTEM request flags, except that they act on the sibling hardware thread sharing the physical processor with the CPU measured by cpc_bind_cpu(). Some CPC sets will fail to bind due to resource

constraints. The most common type of resource constraint is an ESCR conflict among one or more requests in the set. For example, the branch_retired event cannot be measured on counters 12 and 13 simultaneously because both counters require the `CRU_ESCR2` ESCR to measure this event. To measure *branch_retired* events simultaneously on more than one counter, use counters such that one counter uses `CRU_ESCR2` and the other counter uses CRU_ESCR3. See the processor documentation for details.

Name cpc_bind_event, cpc_take_sample, cpc_rele – use CPU performance counters on lwps

Synopsis

```
cc [ flag... ] file... -lcpc [ library... ]
#include <libcpc.h>

int cpc_bind_event(cpc_event_t *event, int flags);

int cpc_take_sample(cpc_event_t *event);

int cpc_rele(void);
```

Description Once the events to be sampled have been selected using, for example, `cpc_strtoevent(3CPC)`, the event selections can be bound to the calling LWP using `cpc_bind_event()`. If `cpc_bind_event()` returns successfully, the system has associated performance counter context with the calling LWP. The context allows the system to virtualize the hardware counters to that specific LWP, and the counters are enabled.

Two flags are defined that can be passed into the routine to allow the behavior of the interface to be modified, as described below.

Counter values can be sampled at any time by calling `cpc_take_sample()`, and dereferencing the fields of the `ce_pic[]` array returned. The `ce_hrt` field contains the timestamp at which the kernel last sampled the counters.

To immediately remove the performance counter context on an LWP, the `cpc_rele()` interface should be used. Otherwise, the context will be destroyed after the LWP or process exits.

The caller should take steps to ensure that the counters are sampled often enough to avoid the 32-bit counters wrapping. The events most prone to wrap are those that count processor clock cycles. If such an event is of interest, sampling should occur frequently so that less than 4 billion clock cycles can occur between samples. Practically speaking, this is only likely to be a problem for otherwise idle systems, or when processes are bound to processors, since normal context switching behavior will otherwise hide this problem.

Return Values Upon successful completion, `cpc_bind_event()` and `cpc_take_sample()` return `0`. Otherwise, these functions return `-1`, and set `errno` to indicate the error.

Errors The `cpc_bind_event()` and `cpc_take_sample()` functions will fail if:

`EACCES` For `cpc_bind_event()`, access to the requested hypervisor event was denied.

`EAGAIN` Another process may be sampling system-wide CPU statistics. For `cpc_bind_event()`, this implies that no new contexts can be created. For `cpc_take_sample()`, this implies that the performance counter context has been invalidated and must be released with `cpc_rele()`. Robust programs should be coded to expect this behavior and recover from it by releasing the now invalid context by calling `cpc_rele()` sleeping for a while, then attempting to bind and sample the event once more.

EINVAL	The `cpc_take_sample()` function has been invoked before the context is bound.
ENOTSUP	The caller has attempted an operation that is illegal or not supported on the current platform, such as attempting to specify signal delivery on counter overflow on a CPU that doesn't generate an interrupt on counter overflow.

Usage Prior to calling `cpc_bind_event()`, applications should call `cpc_access(3CPC)` to determine if the counters are accessible on the system.

Examples **EXAMPLE 1** Use hardware performance counters to measure events in a process.

The example below shows how a standalone program can be instrumented with the `libcpc` routines to use hardware performance counters to measure events in a process. The program performs 20 iterations of a computation, measuring the counter values for each iteration. By default, the example makes the counters measure external cache references and external cache hits; these options are only appropriate for UltraSPARC processors. By setting the `PERFEVENTS` environment variable to other strings (a list of which can be gleaned from the `-h` flag of the `cpustat` or `cputrack` utilities), other events can be counted. The `error()` routine below is assumed to be a user-provided routine analogous to the familiar `printf(3C)` routine from the C library but which also performs an `exit(2)` after printing the message.

```
#include <inttypes.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <libcpc.h>
int
main(int argc, char *argv[])
{
int cpuver, iter;
char *setting = NULL;
cpc_event_t event;

if (cpc_version(CPC_VER_CURRENT) != CPC_VER_CURRENT)
    error("application:library cpc version mismatch!");

if ((cpuver = cpc_getcpuver()) == -1)
    error("no performance counter hardware!");

if ((setting = getenv("PERFEVENTS")) == NULL)
    setting = "pic0=EC_ref,pic1=EC_hit";

if (cpc_strtoevent(cpuver, setting, &event) != 0)
    error("can't measure '%s' on this processor", setting);
setting = cpc_eventtostr(&event);

if (cpc_access() == -1)
    error("can't access perf counters: %s", strerror(errno));
```

EXAMPLE 1 Use hardware performance counters to measure events in a process. *(Continued)*

```
if (cpc_bind_event(&event, 0) == -1)
    error("can't bind lwp%d: %s", _lwp_self(), strerror(errno));

for (iter = 1; iter <= 20; iter++) {
    cpc_event_t before, after;

    if (cpc_take_sample(&before) == -1)
        break;

    /* ==> Computation to be measured goes here <== */

    if (cpc_take_sample(&after) == -1)
        break;
    (void) printf("%3d: %" PRId64 " %" PRId64 "\
", iter,
        after.ce_pic[0] - before.ce_pic[0],
        after.ce_pic[1] - before.ce_pic[1]);
}

if (iter != 20)
    error("can't sample '%s': %s", setting,    strerror(errno));

free(setting);
return (0);
}
```

EXAMPLE 2 Write a signal handler to catch overflow signals.

This example builds on Example 1, but demonstrates how to write the signal handler to catch overflow signals. The counters are preset so that counter zero is 1000 counts short of overflowing, while counter one is set to zero. After 1000 counts on counter zero, the signal handler will be invoked.

First the signal handler:

```
#define PRESET0        (UINT64_MAX - UINT64_C(999))
#define PRESET1        0

void
emt_handler(int sig, siginfo_t *sip, void *arg)
{
ucontext_t *uap = arg;
cpc_event_t sample;

if (sig != SIGEMT || sip->si_code != EMT_CPCOVF) {
```

EXAMPLE 2 Write a signal handler to catch overflow signals. *(Continued)*

```
    psignal(sig, "example");
    psiginfo(sip, "example");
    return;
}

(void) printf("lwp%d - si_addr %p ucontext: %%pc %p %%sp %p\
",
    _lwp_self(), (void *)sip->si_addr,
    (void *)uap->uc_mcontext.gregs[PC],
    (void *)uap->uc_mcontext.gregs[USP]);

if (cpc_take_sample(&sample) == -1)
    error("can't sample: %s", strerror(errno));

(void) printf("0x%" PRIx64 " 0x%" PRIx64 "\
",
    sample.ce_pic[0], sample.ce_pic[1]);
(void) fflush(stdout);

sample.ce_pic[0] = PRESET0;
sample.ce_pic[1] = PRESET1;
if (cpc_bind_event(&sample, CPC_BIND_EMT_OVF) == -1)
    error("cannot bind lwp%d: %s", _lwp_self(), strerror(errno));
}
```

and second the setup code (this can be placed after the code that selects the event to be measured):

```
struct sigaction act;
cpc_event_t event;
...
act.sa_sigaction = emt_handler;
bzero(&act.sa_mask, sizeof (act.sa_mask));
act.sa_flags = SA_RESTART|SA_SIGINFO;
if (sigaction(SIGEMT, &act, NULL) == -1)
    error("sigaction: %s", strerror(errno));
event.ce_pic[0] = PRESET0;
event.ce_pic[1] = PRESET1;
if (cpc_bind_event(&event, CPC_BIND_EMT_OVF) == -1)
    error("cannot bind lwp%d: %s", _lwp_self(), strerror(errno));

for (iter = 1; iter <= 20; iter++) {
    /* ==> Computation to be measured goes here <== */
}

cpc_bind_event(NULL, 0);    /* done */
```

EXAMPLE 2 Write a signal handler to catch overflow signals. *(Continued)*

Note that a more general version of the signal handler would use `write(2)` directly instead of depending on the signal-unsafe semantics of `stderr` and `stdout`. Most real signal handlers will probably do more with the samples than just print them out.

Attributes

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

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

See Also

`cpustat(1M)`, `cputrack(1)`, `write(2)`. `cpc(3CPC)`, `cpc_access(3CPC)`, `cpc_bind_curlwp(3CPC)`, `cpc_set_sample(3CPC)`, `cpc_strtoevent(3CPC)`, `cpc_unbind(3CPC)`, `libcpc(3LIB)`, `attributes(5)`

Notes

The `cpc_bind_event()`, `cpc_take_sample()`, and `cpc_rele()` functions exist for binary compatibility only. Source containing these functions will not compile. These functions are obsolete and might be removed in a future release. Applications should use `cpc_bind_curlwp(3CPC)`, `cpc_set_sample(3CPC)`, and `cpc_unbind(3CPC)` instead.

Sometimes, even the overhead of performing a system call will be too disruptive to the events being measured. Once a call to `cpc_bind_event()` has been issued, it is possible to directly access the performance hardware registers from within the application. If the performance counter context is active, then the counters will count on behalf of the current LWP.

```
SPARC  rd %pic, %rN        ! All UltraSPARC
       wr %rN, %pic        ! (ditto, but see text)

  x86  rdpmc               ! Pentium II only
```

If the counter context is not active or has been invalidated, the `%pic` register (SPARC), and the `rdpmc` instruction (Pentium) will become unavailable.

Note that the two 32-bit UltraSPARC performance counters are kept in the single 64-bit `%pic` register so a couple of additional instructions are required to separate the values. Also note that when the `%pcr` register bit has been set that configures the `%pic` register as readable by an application, it is also writable. Any values written will be preserved by the context switching mechanism.

Pentium II processors support the non-privileged `rdpmc` instruction which requires [5] that the counter of interest be specified in `%ecx`, and returns a 40-bit value in the `%edx:%eax` register pair. There is no non-privileged access mechanism for Pentium I processors.

Handling counter overflow

As described above, when counting events, some processors allow their counter registers to silently overflow. More recent CPUs such as UltraSPARC III and Pentium II, however, are capable of generating an interrupt when the hardware counter overflows. Some processors offer more control over when interrupts will actually be generated. For example, they might allow the interrupt to be programmed to occur when only one of the counters overflows. See `cpc_strtoevent(3CPC)` for the syntax.

The most obvious use for this facility is to ensure that the full 64-bit counter values are maintained without repeated sampling. However, current hardware does not record which counter overflowed. A more subtle use for this facility is to preset the counter to a value to a little less than the maximum value, then use the resulting interrupt to catch the counter overflow associated with that event. The overflow can then be used as an indication of the frequency of the occurrence of that event.

Note that the interrupt generated by the processor may not be particularly precise. That is, the particular instruction that caused the counter overflow may be earlier in the instruction stream than is indicated by the program counter value in the ucontext.

When `cpc_bind_event()` is called with the `CPC_BIND_EMT_OVF` flag set, then as before, the control registers and counters are preset from the 64-bit values contained in `event`. However, when the flag is set, the kernel arranges to send the calling process a `SIGEMT` signal when the overflow occurs, with the `si_code` field of the corresponding `siginfo` structure set to `EMT_CPCOVF`, and the `si_addr` field is the program counter value at the time the overflow interrupt was delivered. Counting is disabled until the next call to `cpc_bind_event()`. Even in a multithreaded process, during execution of the signal handler, the thread behaves as if it is temporarily bound to the running LWP.

Different processors have different counter ranges available, though all processors supported by Solaris allow at least 31 bits to be specified as a counter preset value; thus portable preset values lie in the range `UINT64_MAX` to `UINT64_MAX–INT32_MAX`.

The appropriate preset value will often need to be determined experimentally. Typically, it will depend on the event being measured, as well as the desire to minimize the impact of the act of measurement on the event being measured; less frequent interrupts and samples lead to less perturbation of the system.

If the processor cannot detect counter overflow, this call will fail (`ENOTSUP`). Specifying a null event unbinds the context from the underlying LWP and disables signal delivery. Currently, only user events can be measured using this technique. See Example 2, above.

Inheriting events onto multiple LWPs

By default, the library binds the performance counter context to the current LWP only. If the `CPC_BIND_LWP_INHERIT` flag is set, then any subsequent LWPs created by that LWP will automatically inherit the same performance counter context. The counters will be initialized to 0 as if a `cpc_bind_event()` had just been issued. This automatic inheritance behavior can be useful when dealing with multithreaded programs to determine aggregate statistics for the program as a whole.

If the CPC_BIND_EMT_OVF flag is also set, the process will immediately dispatch a SIGEMT signal to the freshly created LWP so that it can preset its counters appropriately on the new LWP. This initialization condition can be detected using cpc_take_sample() to check that both ce_pic[] values are set to UINT64_MAX.

Name cpc_buf_create, cpc_buf_destroy, cpc_set_sample, cpc_buf_get, cpc_buf_set, cpc_buf_hrtime, cpc_buf_tick, cpc_buf_sub, cpc_buf_add, cpc_buf_copy, cpc_buf_zero – sample and manipulate CPC data

Synopsis

```
cc [ flag... ] file... -lcpc [ library... ]
#include <libcpc.h>

cpc_buf_t *cpc_buf_create(cpc_t *cpc, cpc_set_t *set);

int cpc_buf_destroy(cpc_t *cpc, cpc_buf_t *buf);

int cpc_set_sample(cpc_t *cpc, cpc_set_t *set, cpc_buf_t *buf);

int cpc_buf_get(cpc_t *cpc, cpc_buf_t *buf, int index, uint64_t *val);

int cpc_buf_set(cpc_t *cpc, cpc_buf_t *buf, int index, uint64_t val);

hrtime_t cpc_buf_hrtime(cpc_t *cpc, cpc_buf_t *buf);

uint64_t cpc_buf_tick(cpc_t *cpc, cpc_buf_t *buf);

void cpc_buf_sub(cpc_t *cpc, cpc_buf_t *ds, cpc_buf_t *a, cpc_buf_t *b);

void cpc_buf_add(cpc_t *cpc, cpc_buf_t *ds, cpc_buf_t *a, cpc_buf_t *b);

void cpc_buf_copy(cpc_t *cpc, cpc_buf_t *ds, cpc_buf_t *src);

void cpc_buf_zero(cpc_t *cpc, cpc_buf_t *buf);
```

Description Counter data is sampled into CPC buffers, which are represented by the opaque data type `cpc_buf_t`. A CPC buffer is created with `cpc_buf_create()` to hold the data for a specific CPC set. Once a CPC buffer has been created, it can only be used to store and manipulate the data of the CPC set for which it was created.

Once a set has been successfully bound, the counter values are sampled using `cpc_set_sample()`. The `cpc_set_sample()` function takes a snapshot of the hardware performance counters counting on behalf of the requests in *set* and stores the 64-bit virtualized software representations of the counters in the supplied CPC buffer. If a set was bound with `cpc_bind_curlwp(3CPC)` or `cpc_bind_curlwp(3CPC)`, the set can only be sampled by the LWP that bound it.

The kernel maintains 64-bit virtual software counters to hold the counts accumulated for each request in the set, thereby allowing applications to count past the limits of the underlying physical counter, which can be significantly smaller than 64 bits. The kernel attempts to maintain the full 64-bit counter values even in the face of physical counter overflow on architectures and processors that can automatically detect overflow. If the processor is not capable of overflow detection, the caller must ensure that the counters are sampled often enough to avoid the physical counters wrapping. The events most prone to wrap are those that count processor clock cycles. If such an event is of interest, sampling should occur frequently so that the counter does not wrap between samples.

The `cpc_buf_get()` function retrieves the last sampled value of a particular request in *buf*. The *index* argument specifies which request value in the set to retrieve. The index for each request is returned during set configuration by `cpc_set_add_request(3CPC)`. The 64-bit virtualized software counter value is stored in the location pointed to by the *val* argument.

The `cpc_buf_set()` function stores a 64-bit value to a specific request in the supplied buffer. This operation can be useful for performing calculations with CPC buffers, but it does not affect the value of the hardware counter (and thus will not affect the next sample).

The `cpc_buf_hrtime()` function returns a high-resolution timestamp indicating exactly when the set was last sampled by the kernel.

The `cpc_buf_tick()` function returns a 64-bit virtualized cycle counter indicating how long the set has been programmed into the counter since it was bound. The units of the values returned by `cpc_buf_tick()` are CPU clock cycles.

The `cpc_buf_sub()` function calculates the difference between each request in sets *a* and *b*, storing the result in the corresponding request within set *ds*. More specifically, for each request index *n*, this function performs *ds*[*n*] = *a*[*n*] - *b*[n]. Similarly, `cpc_buf_add()` adds each request in sets *a* and *b* and stores the result in the corresponding request within set *ds*.

The `cpc_buf_copy()` function copies each value from buffer *src* into buffer *ds*. Both buffers must have been created from the same `cpc_set_t`.

The `cpc_buf_zero()` function sets each request's value in the buffer to zero.

The `cpc_buf_destroy()` function frees all resources associated with the CPC buffer.

Return Values Upon successful completion, `cpc_buf_create()` returns a pointer to a CPC buffer which can be used to hold data for the set argument. Otherwise, this function returns `NULL` and sets `errno` to indicate the error.

Upon successful completion, `cpc_set_sample()`, `cpc_buf_get()`, and `cpc_buf_set()` return 0. Otherwise, they return -1 and set `errno` to indicate the error.

Errors These functions will fail if:

`EINVAL` For `cpc_set_sample()`, the set is not bound, the set and/or CPC buffer were not created with the given *cpc* handle, or the CPC buffer was not created with the supplied set.

`EAGAIN` When using `cpc_set_sample()` to sample a CPU-bound set, the LWP has been unbound from the processor it is measuring.

`ENOMEM` The library could not allocate enough memory for its internal data structures.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Safe

See Also cpc_bind_curlwp(3CPC), cpc_set_add_request(3CPC), libcpc(3LIB), attributes(5)

Notes Often the overhead of performing a system call can be too disruptive to the events being measured. Once a cpc_bind_curlwp(3CPC) call has been issued, it is possible to access directly the performance hardware registers from within the application. If the performance counter context is active, the counters will count on behalf of the current LWP.

Not all processors support this type of access. On processors where direct access is not possible, cpc_set_sample() must be used to read the counters.

SPARC

```
rd %pic, %rN        ! All UltraSPARC
wr %rN, %pic        ! (All UltraSPARC, but see text)
```

x86

```
rdpmc               ! Pentium II, III, and 4 only
```

If the counter context is not active or has been invalidated, the %pic register (SPARC), and the rdpmc instruction (Pentium) becomes unavailable.

Pentium II and III processors support the non-privileged rdpmc instruction that requires that the counter of interest be specified in %ecx and return a 40-bit value in the %edx:%eax register pair. There is no non-privileged access mechanism for Pentium I processors.

Name cpc_count_usr_events, cpc_count_sys_events – enable and disable performance counters

Synopsis

```
cc [ flag... ] file... -lcpc [ library... ]
#include <libcpc.h>

int cpc_count_usr_events(int enable);

int cpc_count_sys_events(int enable);
```

Description In certain applications, it can be useful to explicitly enable and disable performance counters at different times so that the performance of a critical algorithm can be examined. The `cpc_count_usr_events()` function can be used to control whether events are counted on behalf of the application running in user mode, while `cpc_count_sys_events()` can be used to control whether events are counted on behalf of the application while it is running in the kernel, without otherwise disturbing the binding of events to the invoking LWP. If the *enable* argument is non-zero, counting of events is enabled, otherwise they are disabled.

Return Values Upon successful completion, `cpc_count_usr_events()` and `cpc_count_sys_events()` return `0`. Otherwise, the functions return `-1` and set `errno` to indicate the error.

Errors The `cpc_count_usr_events()` and `cpc_count_sys_events()` functions will fail if:

`EAGAIN` The associated performance counter context has been invalidated by another process.

`EINVAL` No performance counter context has been created, or an attempt was made to enable system events while delivering counter overflow signals.

Examples **EXAMPLE 1** Use `cpc_count_usr_events()` to minimize code needed by application.

In this example, the routine `cpc_count_usr_events()` is used to minimize the amount of code that needs to be added to the application. The `cputrack(1)` command can be used in conjunction with these interfaces to provide event programming, sampling, and reporting facilities.

If the application is instrumented in this way and then started by `cputrack` with the `nouser` flag set in the event specification, counting of user events will only be enabled around the critical code section of interest. If the program is run normally, no harm will ensue.

```
int have_counters = 0;
int
main(int argc, char *argv[])
{
    if (cpc_version(CPC_VER_CURRENT) == CPC_VER_CURRENT &&
        cpc_getcpuver() != -1 && cpc_access() == 0)
        have_counters = 1;

    /* ... other application code */
```

EXAMPLE 1 Use `cpc_count_usr_events()` to minimize code needed by application. *(Continued)*

```
    if (have_counters)
        (void) cpc_count_usr_events(1);

    /* ==> Code to be measured goes here <== */

    if (have_counters)
        (void) cpc_count_usr_events(0);

    /* ... other application code */
}
```

Attributes

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

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

See Also

`cputrack(1)`, `cpc(3CPC)`, `cpc_access(3CPC)`, `cpc_bind_event(3CPC)`, `cpc_enable(3CPC)`, `cpc_getcpuver(3CPC)`, `cpc_pctx_bind_event(3CPC)`, `cpc_version(3CPC)`, `libcpc(3LIB)`, `attributes(5)`

Notes

The `cpc_count_usr_events()` and `cpc_count_sys_events()` functions exist for binary compatibility only. Source containing these functions will not compile. These functions are obsolete and might be removed in a future release. Applications should use `cpc_enable(3CPC)` instead.

Name cpc_enable, cpc_disable – enable and disable performance counters

Synopsis

```
cc [ flag... ] file... -lcpc [ library... ]
#include <libcpc.h>

int cpc_enable(cpc_t *cpc);

int cpc_disable(cpc_t *cpc);
```

Description In certain applications, it can be useful to explicitly enable and disable performance counters at different times so that the performance of a critical algorithm can be examined. The `cpc_enable()` and `cpc_disable()` functions can be used to enable and disable the performance counters without otherwise disturbing the invoking LWP's performance hardware configuration.

Return Values Upon successful completion, `cpc_enable()` and `cpc_disable()` return 0. Otherwise, they return -1 and set `errno` to indicate the error.

Errors These functions will fail if:

`EAGAIN` The associated performance counter context has been invalidated by another process.

`EINVAL` No performance counter context has been created for the calling LWP.

Examples **EXAMPLE 1** Use cpc_enable and cpc_disable to minimize code needed by application.

In the following example, the `cpc_enable()` and `cpc_disable()` functions are used to minimize the amount of code that needs to be added to the application. The `cputrack(1)` command can be used in conjunction with these functions to provide event programming, sampling, and reporting facilities.

If the application is instrumented in this way and then started by `cputrack` with the `nouser` flag set in the event specification, counting of user events will only be enabled around the critical code section of interest. If the program is run normally, no harm will ensue.

```
int
main(int argc, char *argv[])
{
   cpc_t *cpc = cpc_open(CPC_VER_CURRENT);
    /* ... application code ... */

   if (cpc != NULL)
           (void) cpc_enable(cpc);

    /* ==> Code to be measured goes here <== */

   if (cpc != NULL)
           (void) cpc_disable(cpc);
```

EXAMPLE 1 Use cpc_enable and cpc_disable to minimize code needed by application. *(Continued)*

```
    /* ... other application code */
}
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Safe

See Also

cputrack(1), cpc(3CPC), cpc_open(3CPC), libcpc(3LIB), attributes(5)

Name cpc_event – data structure to describe CPU performance counters

Synopsis

```
#include <libcpc.h>
```

Description The `libcpc` interfaces manipulate CPU performance counters using the `cpc_event_t` data structure. This structure contains several fields that are common to all processors, and some that are processor-dependent. These structures can be declared by a consumer of the API, thus the size and offsets of the fields and the entire data structure are fixed per processor for any particular version of the library. See `cpc_version(3CPC)` for details of library versioning.

SPARC For UltraSPARC, the structure contains the following members:

```
typedef struct {
        int ce_cpuver;
        hrtime_t ce_hrt;
        uint64_t ce_tick;
        uint64_t ce_pic[2];
        uint64_t ce_pcr;
} cpc_event_t;
```

x86 For Pentium, the structure contains the following members:

```
typedef struct {
        int ce_cpuver;
        hrtime_t ce_hrt;
        uint64_t ce_tsc;
        uint64_t ce_pic[2];
        uint32_t ce_pes[2];
#define ce_cesr ce_pes[0]
} cpc_event_t;
```

The APIs are used to manipulate the highly processor-dependent control registers (the `ce_pcr`, `ce_cesr`, and `ce_pes` fields); the programmer is strongly advised not to reference those fields directly in portable code. The `ce_pic` array elements contain 64-bit accumulated counter values. The hardware registers are virtualized to 64-bit quantities even though the underlying hardware only supports 32-bits (UltraSPARC) or 40-bits (Pentium) before overflow.

The `ce_hrt` field is a high resolution timestamp taken at the time the counters were sampled by the kernel. This uses the same timebase as `gethrtime(3C)`.

On SPARC V9 machines, the number of cycles spent running on the processor is computed from samples of the processor-dependent `%tick` register, and placed in the `ce_tick` field. On Pentium processors, the processor-dependent time-stamp counter register is similarly sampled and placed in the `ce_tsc` field.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

See Also gethrtime(3C), cpc(3CPC), cpc_version(3CPC), libcpc(3LIB), attributes(5)

Name cpc_event_diff, cpc_event_accum – simple difference and accumulate operations

Synopsis

```
cc [ flag... ] file... -lcpc [ library... ]
#include <libcpc.h>

void cpc_event_accum(cpc_event_t *accum, cpc_event_t *event);

void cpc_event_diff(cpc_event_t *diff, cpc_event_t *after,
     cpc_event_t *before);
```

Description The `cpc_event_accum()` and `cpc_event_diff()` functions perform common accumulate and difference operations on `cpc_event(3CPC)` data structures. Use of these functions increases program portability, since structure members are not referenced directly .

`cpc_event_accum()` The `cpc_event_accum()` function adds the `ce_pic` fields of *event* into the corresponding fields of *accum*. The `ce_hrt` field of *accum* is set to the later of the times in *event* and *accum*.

SPARC:

The function adds the contents of the `ce_tick` field of *event* into the corresponding field of *accum*.

x86:

The function adds the contents of the `ce_tsc` field of *event* into the corresponding field of *accum*.

`cpc_event_diff()` The `cpc_event_diff()` function places the difference between the `ce_pic` fields of *after* and *before* and places them in the corresponding field of *diff*. The `ce_hrt` field of *diff* is set to the `ce_hrt` field of *after*.

SPARC:

Additionally, the function computes the difference between the `ce_tick` fields of *after* and *before*, and places it in the corresponding field of `diff`.

x86:

Additionally, the function computes the difference between the `ce_tsc` fields of *after* and *before*, and places it in the corresponding field of *diff*.

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

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

See Also cpc(3CPC), cpc_buf_add(3CPC), cpc_buf_sub(3CPC), cpc_event(3CPC), libcpc(3LIB), attributes(5)

Notes The cpc_event_accum() and cpc_event_diff() functions exist for binary compatibility only. Source containing these functions will not compile. These functions are obsolete and might be removed in a future release. Applications should use cpc_buf_add(3CPC) and cpc_buf_sub(3CPC) instead.

Name cpc_getcpuver, cpc_getcciname, cpc_getcpuref, cpc_getusage, cpc_getnpic, cpc_walk_names – determine CPU performance counter configuration

Synopsis

```
cc [ flag... ] file... –lcpc [ library... ]
#include <libcpc.h>

int cpc_getcpuver(void);

const char *cpc_getcciname(int cpuver);

const char *cpc_getcpuref(int cpuver);

const char *cpc_getusage(int cpuver);

uint_t cpc_getnpic(int cpuver);

void cpc_walk_names(int cpuver, int regno, void *arg,
     void (*action)(void *arg, int regno,  const char *name,
     uint8_t bits));
```

Description The `cpc_getcpuver()` function returns an abstract integer that corresponds to the distinguished version of the underlying processor. The library distinguishes between processors solely on the basis of their support for performance counters, so the version returned should not be interpreted in any other way. The set of values returned by the library is unique across all processor implementations.

The `cpc_getcpuver()` function returns –1 if the library cannot support CPU performance counters on the current architecture. This may be because the processor has no such counter hardware, or because the library is unable to recognize it. Either way, such a return value indicates that the configuration functions described on this manual page cannot be used.

The `cpc_getcciname()` function returns a printable description of the processor performance counter interfaces-for example, the string *UltraSPARC I&II*. Note that this name should not be assumed to be the same as the name the manufacturer might otherwise ascribe to the processor. It simply names the performance counter interfaces as understood by the library, and thus names the set of performance counter events that can be described by that interface. If the *cpuver* argument is unrecognized, the function returns `NULL`.

The `cpc_getcpuref()` function returns a string that describes a reference work that should be consulted to (allow a human to) understand the semantics of the performance counter events that are known to the library. If the *cpuver* argument is unrecognized, the function returns `NULL`. The string returned might be substantially longer than 80 characters. Callers printing to a terminal might want to insert line breaks as appropriate.

The `cpc_getusage()` function returns a compact description of the `getsubopt()`-oriented syntax that is consumed by `cpc_strtoevent`(3CPC). It is returned as a space-separated set of tokens to allow the caller to wrap lines at convenient boundaries. If the *cpuver* argument is unrecognized, the function returns `NULL`.

The `cpc_getnpic()` function returns the number of valid fields in the `ce_pic`[] array of a `cpc_event_t` data structure.

The library maintains a list of events that it believes the processor capable of measuring, along with the bit patterns that must be set in the corresponding control register, and which counter the result will appear in. The `cpc_walk_names()` function calls the *action*( ) function on each element of the list so that an application can print appropriate help on the set of events known to the library. The *arg* parameter is passed uninterpreted from the caller on each invocation of the *action*( ) function.

If the parameters specify an invalid or unknown CPU or register number, the function silently returns without invoking the action function.

Usage Prior to calling any of these functions, applications should call `cpc_access(3CPC)` to determine if the counters are accessible on the system.

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

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

See Also `cpc(3CPC)`, `cpc_access(3CPC)`, `cpc_cciname(3CPC)`, `cpc_cpuref(3CPC)`, `cpc_npic(3CPC)`, `cpc_walk_events_all(3CPC)libcpc(3LIB)`, `attributes(5)`

Notes The `cpc_getcpuver()`, `cpc_getcciname()`, `cpc_getcpuref()`, `cpc_getusage()`, `cpc_getnpic()`, and `cpc_walk_names()` functions exist for binary compatibility only. Source containing these functions will not compile. These functions are obsolete and might be removed in a future release. Applications should use `cpc_cciname(3CPC)`, `cpc_cpuref(3CPC)`, `cpc_npic(3CPC)`, and `cpc_npic(3CPC)` instead.

Only SPARC processors are described by the SPARC version of the library, and only x86 processors are described by the x86 version of the library.

Name cpc_npic, cpc_caps, cpc_cciname, cpc_cpuref, cpc_walk_events_all, cpc_walk_events_pic, cpc_walk_attrs – determine CPU performance counter configuration

Synopsis

```
cc [ flag... ] file... -lcpc [ library... ]
#include <libcpc.h>

const char *cpc_cciname(cpc_t *cpc);

const char *cpc_cpuref(cpc_t *cpc);

uint_t cpc_npic(cpc_t *cpc);

uint_t cpc_caps(cpc_t *cpc);

void cpc_walk_events_all(cpc_t *cpc, void *arg, void (*action)(void *arg,
     const char *event));

void cpc_walk_events_pic(cpc_t *cpc, uint_t picno, void *arg,
     void (*action)(void *arg, uint_t picno, const char *event));

void cpc_walk_attrs(cpc_t *cpc, void *arg, void (*action)(void *arg,
     const char *attr));
```

Description The `cpc_cciname()` function returns a printable description of the processor performance counter interfaces, for example, the string UltraSPARC III+ & IV. This name should not be assumed to be the same as the name the manufacturer might otherwise ascribe to the processor. It simply names the performance counter interfaces as understood by the system, and thus names the set of performance counter events that can be described by that interface.

The `cpc_cpuref()` function returns a string that describes a reference work that should be consulted to (allow a human to) understand the semantics of the performance counter events that are known to the system. The string returned might be substantially longer than 80 characters. Callers printing to a terminal might want to insert line breaks as appropriate.

The `cpc_npic()` function returns the number of performance counters accessible on the processor.

The `cpc_caps()` function returns a bitmap containing the bitwise inclusive-OR of zero or more flags that describe the capabilities of the processor. If `CPC_CAP_OVERFLOW_INTERRUPT` is present, the processor can generate an interrupt when a hardware performance counter overflows. If `CPC_CAP_OVERFLOW_PRECISE` is present, the processor can determine precisely which counter overflowed, thereby affecting the behavior of the overflow notification mechanism described in `cpc_bind_curlwp(3CPC)`.

The system maintains a list of performance counter events supported by the underlying processor. Some processors are able to count all events on all hardware counters, while other processors restrict certain events to be counted only on specific hardware counters. The system also maintains a list of processor-specific attributes that can be used for advanced configuration of the performance counter hardware. These functions allow applications to

determine what events and attributes are supported by the underlying processor. The reference work pointed to by `cpc_cpuref()` should be consulted to understand the reasons for and use of the attributes.

The `cpc_walk_events_all()` function calls the *action* function on each element of a global *event* list. The *action* function is called with each event supported by the processor, regardless of which counter is capable of counting it. The *action* function is called only once for each event, even if that event can be counted on more than one counter.

The `cpc_walk_events_pic()` function calls the action *function* with each event supported by the counter indicated by the *picno* argument, where *picno* ranges from 0 to the value returned by `cpc_npic()`.

The system maintains a list of attributes that can be used to enable advanced features of the performance counters on the underlying processor. The `cpc_walk_attrs()` function calls the *action* function for each supported attribute name. See the reference material as returned by `cpc_cpuref(3CPC)` for the semantics use of attributes.

Return Values The `cpc_cciname()` function always returns a printable description of the processor performance counter interfaces.

The `cpc_cpuref()` function always returns a string that describes a reference work.

The `cpc_npic()` function always returns the number of performance counters accessible on the processor.

The `cpc_caps()` function always returns a bitmap containing the bitwise inclusive-OR of zero or more flags that describe the capabilities of the processor.

If the user-defined function specified by *action* is not called, the `cpc_walk_events_all()`, `cpc_walk_events_pic()`, and `cpc_walk_attrs()` functions set `errno` to indicate the error.

Errors The `cpc_walk_events_all()`, `cpc_walk_events_pic()`, and `cpc_walk_attrs()` functions will fail if:

`ENOMEM` There is not enough memory available.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Safe

See Also `cpc_bind_curlwp`(3CPC), `libcpc`(3LIB), `attributes`(5)

Name cpc_open, cpc_close – initialize the CPU Performance Counter library

Synopsis

```
cc [ flag... ] file... -lcpc [ library... ]
#include <libcpc.h>

cpc_t *cpc_open(int vers);

int cpc_close(cpc_t *cpc);
```

Description The `cpc_open()` function initializes `libcpc(3LIB)` and returns an identifier that must be used as the *cpc* argument in subsequent `libcpc` function calls. The `cpc_open()` function takes an interface version as an argument and returns `NULL` if that version of the interface is incompatible with the `libcpc` implementation present on the system. Usually, the argument has the value of `CPC_VER_CURRENT` bound to the application when it was compiled.

The `cpc_close()` function releases all resources associated with the *cpc* argument. Any bound counters utilized by the process are unbound. All entities of type `cpc_set_t` and `cpc_buf_t` are invalidated and destroyed.

Return Values If the version requested is supported by the implementation, `cpc_open()` returns a `cpc_t` handle for use in all subsequent `libcpc` operations. If the implementation cannot support the version needed by the application, `cpc_open()` returns `NULL`, indicating that the application at least needs to be recompiled to operate correctly on the new platform and might require further changes.

The `cpc_close()` function always returns 0.

Errors These functions will fail if:

`EINVAL` The version requested by the client is incompatible with the implementation.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Safe

See Also `libcpc(3LIB)`, `attributes(5)`

Name cpc_pctx_bind_event, cpc_pctx_take_sample, cpc_pctx_rele, cpc_pctx_invalidate – access CPU performance counters in other processes

Synopsis

```
cc [ flag... ] file... -lcpc -lpctx [ library... ]
#include <libpctx.h>
#include <libcpc.h>

int cpc_pctx_bind_event(pctx_t *pctx, id_t lwpid, cpc_event_t *event,
     int flags);

int cpc_pctx_take_sample(pctx_t *pctx, id_t lwpid, cpc_event_t *event);

int cpc_pctx_rele(pctx_t *pctx, id_t lwpid);

int cpc_pctx_invalidate(pctx_t *pctx, id_t lwpid);
```

Description These functions are designed to be run in the context of an event handler created using the libpctx(3LIB) family of functions that allow the caller, also known as the *controlling process*, to manipulate the performance counters in the context of a *controlled process*. The controlled process is described by the *pctx* argument, which must be obtained from an invocation of pctx_capture(3CPC) or pctx_create(3CPC) and passed to the functions described on this page in the context of an event handler.

The semantics of the functions cpc_pctx_bind_event(), cpc_pctx_take_sample(), and cpc_pctx_rele() are directly analogous to those of cpc_bind_event(), cpc_take_sample(), and cpc_rele() described on the cpc_bind_event(3CPC) manual page.

The cpc_pctx_invalidate() function allows the performance context to be invalidated in an LWP in the controlled process.

Return Values These functions return 0 on success. On failure, they return −1 and set errno to indicate the error.

Errors The cpc_pctx_bind_event(), cpc_pctx_take_sample(), and cpc_pctx_rele() functions return the same errno values the analogous functions described on the cpc_bind_event(3CPC) manual page. In addition, these function may fail if:

EACCES For cpc_pctx_bind_event(), access to the requested hypervisor event was denied.

ESRCH The value of the *lwpid* argument is invalid in the context of the controlled process.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe
Interface Stability	Evolving

See Also cpc(3CPC), cpc_bind_event(3CPC), libcpc(3LIB), pctx_capture(3CPC), pctx_create(3CPC), attributes(5)

Notes The `cpc_pctx_bind_event()`, `cpc_pctx_invalidate()`, `cpc_pctx_rele()`, and `cpc_pctx_take_sample()` functions exist for binary compatibility only. Source containing these functions will not compile. These functions are obsolete and might be removed in a future release. Applications should use `cpc_bind_pctx`(3CPC), `cpc_unbind`(3CPC), and `cpc_set_sample`(3CPC) instead.

The capability to create and analyze overflow events in other processes is not available, though it may be made available in a future version of this API. In the current implementation, the *flags* field must be specified as 0.

Name cpc_set_create, cpc_set_destroy, cpc_set_add_request, cpc_walk_requests – manage sets of counter requests

Synopsis

```
cc [ flag... ] file... -lcpc [ library... ]
#include <libcpc.h>

cpc_set_t *cpc_set_create(cpc_t *cpc);

int cpc_set_destroy(cpc_t *cpc, cpc_set_t *set);

int cpc_set_add_request(cpc_t *cpc, cpc_set_t *set,
     const char *event, uint64_t preset, uint_t flags,
     uint_t nattrs, const cpc_attr_t *attrs);

void cpc_walk_requests(cpc_t *cpc, cpc_set_t *set, void *arg,
     void (*action)(void *arg, int index, const char *event,
     uint64_t preset, uint_t flags, int nattrs,
     const cpc_attr_t *attrs));
```

Description The `cpc_set_create()` function returns an initialized and empty CPC set. A CPC set contains some number of requests, where a request represents a specific configuration of a hardware performance instrumentation counter present on the processor. The `cpc_set_t` data structure is opaque and must not be accessed directly by the application.

Applications wanting to program one or more performance counters must create an empty set with `cpc_set_create()` and add requests to the set with `cpc_set_add_request()`. Once all requests have been added to a set, the set must be bound to the hardware performance counters (see `cpc_bind_curlwp()`, `cpc_bind_pctx()`, and `cpc_bind_cpu()`, all described on `cpc_bind_curlwp`(3CPC)) before counting events. At bind time, the system attempts to match each request with an available physical counter capable of counting the event specified in the request. If the bind is successful, a 64-bit virtualized counter is created to store the counts accumulated by the hardware counter. These counts are stored and managed in CPC buffers separate from the CPC set whose requests are being counted. See `cpc_buf_create`(3CPC) and `cpc_set_sample`(3CPC).

The `cpc_set_add_request()` function specifies a configuration of a hardware counter. The arguments to `cpc_set_add_request()` are:

event A string containing the name of an event supported by the system's processor. The `cpc_walk_events_all()` and `cpc_walk_events_pic()` functions (both described on `cpc_npic`(3CPC)) can be used to query the processor for the names of available events. Certain processors allow the use of raw event codes, in which case a string representation of an event code in a form acceptable to `strtol`(3C) can be used as the *event* argument.

preset The value with which the system initializes the counter.

flags Three flags are defined that modify the behavior of the counter acting on behalf of this request:

`CPC_COUNT_USER`
The counter should count events that occur while the processor is in user mode.

`CPC_COUNT_SYSTEM`
The counter should count events that occur while the processor is in privileged mode.

`CPC_OVF_NOTIFY_EMT`
Request a signal to be sent to the application when the physical counter overflows. A `SIGEMT` signal is delivered if the processor is capable of delivering an interrupt when the counter counts past its maximum value. All requests in the set containing the counter that overflowed are stopped until the set is rebound.

At least one of `CPC_COUNT_USER` or `CPC_COUNT_SYSTEM` must be specified to program the hardware for counting.

nattrs, *attrs* The *nattrs* argument specifies the number of attributes pointed to by the *attrs* argument, which is an array of `cpc_attr_t` structures containing processor-specific attributes that modify the request's configuration. The `cpc_walk_attrs()` function (see `cpc_npic`(3CPC)) can be used to query the processor for the list of attributes it accepts. The library makes a private copy of the *attrs* array, allowing the application to dispose of it immediately after calling `cpc_set_add_request()`.

The `cpc_walk_requests()` function calls the action function on each request that has been added to the set. The *arg* argument is passed unmodified to the *action* function with each call.

Return Values Upon successful completion, `cpc_set_create()` returns a handle to the opaque `cpc_set_t` data structure. Otherwise, NULL is returned and `errno` is set to indicate the error.

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

Upon successful completion, `cpc_set_add_request()` returns an integer index used to refer to the data generated by that request during data retrieval. Otherwise, -1 is returned and `errno` is set to indicate the error.

Errors These functions will fail if:

`EINVAL` An event, attribute, or flag passed to `cpc_set_add_request()` was invalid.

For `cpc_set_destroy()` and `cpc_set_add_request()`, the set parameter was not created with the given cpc_t.

`ENOMEM` There was not enough memory available to the process to create the library's data structures.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also cpc_bind_curlwp(3CPC), cpc_buf_create(3CPC), cpc_npic(3CPC), cpc_seterrhndlr(3CPC), libcpc(3LIB), strtol(3C), attributes(5)

Notes The system automatically determines which particular physical counter to use to count the events specified by each request. Applications can force the system to use a particular counter by specifying the counter number in an attribute named *picnum* that is passed to `cpc_set_add_request()`. Counters are numbered from 0 to *n* - 1, where n is the number of counters in the processor as returned by cpc_npic(3CPC).

Some processors, such as UltraSPARC, do not allow the hardware counters to be programmed differently. In this case, all requests in the set must have the same configuration, or an attempt to bind the set will return `EINVAL`. If a `cpc_errhndlr_t` has been registered with cpc_seterrhndlr(3CPC), the error handler is called with subcode `CPC_CONFLICTING_REQS`. For example, on UltraSPARC `pic0` and `pic1` must both program events in the same processor mode (user mode, kernel mode, or both). For example, `pic0` cannot be programmed with `CPC_COUNT_USER` while `pic1` is programmed with `CPC_COUNT_SYSTEM`. Refer to the hardware documentation referenced by cpc_cpuref(3CPC) for details about a particular processor's performance instrumentation hardware.

Name

cpc_seterrfn – control libcpc error reporting

Synopsis

```
cc [ flag... ] file... -lcpc [ library... ]
#include <libcpc.h>

typedef void (cpc_errfn_t)(const char *fn, const char *fmt, va_list ap);

void cpc_seterrfn(cpc_errfn_t *errfn);
```

Description

For the convenience of programmers instrumenting their code, several libcpc(3LIB) functions automatically emit to `stderr` error messages that attempt to provide a more detailed explanation of their error return values. While this can be useful for simple programs, some applications may wish to report their errors differently—for example, to a window or to a log file.

The `cpc_seterrfn()` function allows the caller to provide an alternate function for reporting errors; the type signature is shown above. The *fn* argument is passed the library function name that detected the error, the format string *fmt* and argument pointer *ap* can be passed directly to vsnprintf(3C) or similar `varargs`-based routine for formatting.

The default printing routine can be restored by calling the routine with an *errfn* argument of `NULL`.

Examples

EXAMPLE 1 Debugging example.

This example produces error messages only when debugging the program containing it, or when the `cpc_strtoevent()` function is reporting an error when parsing an event specification

```
int debugging;
void
myapp_errfn(const char *fn, const char *fmt, va_list ap)
{
        if (strcmp(fn, "strtoevent") != 0 && !debugging)
            return;
        (void) fprintf(stderr, "myapp: cpc_%s(): ", fn);
        (void) vfprintf(stderr, fmt, ap);
}
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe
Interface Stability	Obsolete

See Also cpc(3CPC), cpc_seterrhndlr(3CPC), libcpc(3LIB), vsnprintf(3C), attributes(5)

Notes The cpc_seterrfn() function exists for binary compatibility only. Source containing this function will not compile. This function is obsolete and might be removed in a future release. Applications should use cpc_seterrhndlr(3CPC) instead.

Name cpc_seterrhndlr – control libcpc error reporting

Synopsis

```
cc [ flag... ] file... -lcpc [ library... ]
#include <libcpc.h>

typedef void(cpc_errhndlr_t)(cpc_t *cpc, const char *fn, int subcode,
     const char *fmt, va_list ap);

void cpc_seterrhndlr(cpc_t *cpc, cpc_errhndlr_t *errfn);
```

Description For the convenience of programmers instrumenting their code, several libcpc(3LIB) functions automatically emit to `stderr` error messages that attempt to provide a more detailed explanation of their error return values. While this can be useful for simple programs, some applications might wanat to report their errors differently, for example, to a window or to a log file.

The `cpc_seterrhndlr()` function allows the caller to provide an alternate function for reporting errors. The type signature is shown in the SYNOPSIS. The *fn* argument is passed the library function name that detected the error, an integer subcode indicating the specific error condidtion that has occurred, and the format string *fmt* that contains a textual description of the integer subcode. The format string *fmt* and argument pointer *ap* can be passed directly to vsnprintf(3C) or similar *varargs*-based function for formatting.

The integer subcodes are provided to allow programs to recognize error conditions while using `libcpc`. The *fmt* string is provided as a convenience for easy printing. The error subcodes are:

`CPC_INVALID_EVENT`	A specified event is not supported by the processor.
`CPC_INVALID_PICNUM`	The counter number does not fall in the range of available counters.
`CPC_INVALID_ATTRIBUTE`	A specified attribute is not supported by the processor.
`CPC_ATTRIBUTE_OUT_OF_RANGE`	The value of an attribute is outside the range supported by the processor.
`CPC_RESOURCE_UNAVAIL`	A hardware resource necessary for completing an operation was unavailable.
`CPC_PIC_NOT_CAPABLE`	The requested counter cannot count an assigned event.
`CPC_REQ_INVALID_FLAGS`	One or more requests has invalid flags.
`CPC_CONFLICTING_REQS`	The requests in a set cannot be programmed onto the hardware at the same time.
`CPC_ATTR_REQUIRES_PRIVILEGE`	A request contains an attribute which requires the cpc_cpu privilege, which the process does not have.

The default printing routine can be restored by calling the routine with an *errfn* argument of `NULL`.

Examples

EXAMPLE 1 Debugging example.

The following example produces error messages only when debugging the program containing it, or when the `cpc_bind_curlwp()`, `cpc_bind_cpu()`, or `cpc_bind_pctx()` functions are reporting an error when binding a cpc_set_t.

```
int debugging;
void
myapp_errfn(const char *fn, int subcode, const char *fmt, va_list ap)
{
        if (strncmp(fn, "cpc_bind", 8) != 0 && !debugging)
            return;
        (void) fprintf(stderr, "myapp: cpc_%s(): ", fn);
        (void) vfprintf(stderr, fmt, ap);
}
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Safe

See Also

`cpc_bind_curlwp(3CPC)`, `libcpc(3LIB)`, `vsnprintf(3C)`, `attributes(5)`

Name cpc_shared_open, cpc_shared_bind_event, cpc_shared_take_sample, cpc_shared_rele, cpc_shared_close – use CPU performance counters on processors

Synopsis

```
cc [ flag... ] file... -lcpc [ library... ]
#include <libcpc.h>

int cpc_shared_open(void);

int cpc_shared_bind_event(int fd, cpc_event_t *event, int flags);

int cpc_shared_take_sample(int fd, cpc_event_t *event);

int cpc_shared_rele(int fd);

void cpc_shared_close(int fd);
```

Description The `cpc_shared_open()` function allows the caller to access the hardware counters in such a way that the performance of the currently bound CPU can be measured. The function returns a file descriptor if successful. Only one such open can be active at a time on any CPU.

The `cpc_shared_bind_event()`, `cpc_shared_take_sample()`, and `cpc_shared_rele()` functions are directly analogous to the corresponding `cpc_bind_event()`, `cpc_take_sample()`, and `cpc_rele()` functions described on the `cpc_bind_event(3CPC)`manual page, except that they operate on the counters of a particular processor.

Usage If a thread wishes to access the counters using this interface, it `must` do so using a thread bound to an lwp, (see the `THR_BOUND` flag to `thr_create(3C)`), that has in turn bound itself to a processor using `processor_bind(2)`.

Unlike the `cpc_bind_event(3CPC)` family of functions, no counter context is attached to those lwps, so the performance counter samples from the processors reflects the system-wide usage, instead of per-lwp usage.

The first successful invocation of `cpc_shared_open()` will immediately `invalidate` *all* existing performance counter context on the system, and `prevent` *all* subsequent attempts to bind counter context to lwps from succeeding anywhere on the system until the last caller invokes `cpc_shared_close()`.

This is because it is impossible to simultaneously use the counters to accurately measure per-lwp and system-wide events, so there is an exclusive interlock between these uses.

Access to the shared counters is mediated by file permissions on a `cpc` pseudo device. Only a user with the {`PRIV_SYS_CONFIG`} privilege is allowed to access the shared device. This control prevents use of the counters on a per-lwp basis to other users.

The `CPC_BIND_LWP_INHERIT` and `CPC_BIND_EMT_OVF` flags are invalid for the shared interface.

Return Values On success, the functions (except for `cpc_shared_close()`) return 0. On failure, the functions return –1 and set `errno` to indicate the reason.

Errors

`EACCES` The caller does not have appropriate privilege to access the CPU performance counters system-wide.

`EAGAIN` For cpc_shared_open( ), this value implies that the counters on the bound cpu are busy because they are already being used to measure system-wide events by some other caller.

`EAGAIN` Otherwise, this return value implies that the counters are not available because the thread has been unbound from the processor it was bound to at open time. Robust programs should be coded to expect this behavior, and should invoke `cpc_shared_close( )`, before retrying the operation.

`EINVAL` The counters cannot be accessed on the current CPU because the calling thread is not bound to that CPU using `processor_bind(2)`.

`ENOTSUP` The caller has attempted an operation that is illegal or not supported on the current platform.

`ENXIO` The current machine either has no performance counters, or has been configured to disallow access to them system-wide.

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

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

See Also `processor_bind(2)`, `cpc(3CPC)`, `cpc_bind_cpu(3CPC)`, `cpc_bind_event(3CPC)`, `cpc_set_sample(3CPC)`, `cpc_unbind(3CPC)`, `libcpc(3LIB)`, `thr_create(3C)`, `attributes(5)`

Notes The `cpc_shared_open()`, `cpc_shared_bind_event()`, `cpc_shared_take_sample()`, `cpc_shared_rele()`, and `cpc_shared_close()` functions exist for binary compatibility only. Source containing these functions will not compile. These functions are obsolete and might be removed in a future release. Applications should use `cpc_bind_cpu(3CPC)`, `cpc_set_sample(3CPC)`, and `cpc_unbind(3CPC)` instead.

Name cpc_strtoevent, cpc_eventtostr – translate strings to and from events

Synopsis

```
cc [ flag... ] file... -lcpc [ library... ]
#include <libcpc.h>

int cpc_strtoevent(int cpuver, const char *spec, cpc_event_t *event);

char *cpc_eventtostr(cpc_event_t *event);
```

Description The `cpc_strtoevent()` function translates an event specification to the appropriate collection of control bits in a `cpc_event_t` structure pointed to by the *event* argument. The event specification is a `getsubopt`(3C)–style string that describes the event and any attributes that the processor can apply to the event or events. If successful, the funciton returns 0, the `ce_cpuver` field and the ISA-dependent control registers of event are initialized appropriately, and the rest of the `cpc_event_t` structure is initialized to 0.

The `cpc_eventtostr()` function takes an event and constructs a compact canonical string representation for that event.

Return Values Upon successful completion, `cpc_strtoevent()` returns 0. If the string cannot be decoded, a non-zero value is returned and a message is printed using the library's error-reporting mechanism (see `cpc_seterrfn`(3CPC)).

Upon successful completion, `cpc_eventtostr()` returns a pointer to a string. The string returned must be freed by the caller using `free`(3C). If `cpc_eventtostr()` fails, a null pointer is returned.

Usage The event selection syntax used is processor architecture-dependent. The supported processor families allow variations on how events are counted as well as what events can be counted. This information is available in compact form from the `cpc_getusage()` function (see `cpc_getcpuver`(3CPC)), but is explained in further detail below.

UltraSPARC On UltraSPARC processors, the syntax for setting options is as follows:

```
pic0=<eventspec>,pic1=<eventspec> [,sys] [,nouser]
```

This syntax, which reflects the simplicity of the options available using the `%pcr` register, forces both counter events to be selected. By default only user events are counted; however, the `sys` keyword allows system (kernel) events to be counted as well. User event counting can be disabled by specifying the `nouser` keyword.

The keywords `pic0` and `pic1` may be omitted; they can be used to resolve ambiguities if they exist.

Pentium I On Pentium processors, the syntax for setting counter options is as follows:

```
pic0=<eventspec>,pic1=<eventspec> [,sys[[0|1]]] [,nouser[[0|1]]]
[,noedge[[0|1]]] [,pc[[0|1]]]
```

The syntax and semantics are the same as UltraSPARC, except that is possible to specify whether a particular counter counts user or system events. If unspecified, the specification is presumed to apply to both counters.

There are some additional keywords. The `noedge` keyword specifies that the counter should count clocks (duration) instead of events. The `pc` keyword allows the external pin control pins to be set high (defaults to low). When the pin control register is set high, the external pin will be asserted when the associated register overflows. When the pin control register is set low, the external pin will be asserted when the counter has been incremented. The electrical effect of driving the pin is dependent uptoon how the motherboard manufacturer has chosen to connect it, if it is connected at all.

Pentium II

For Pentium II processors, the syntax is substantially more complex, reflecting the complex configuration options available:

```
pic0=<eventspec>,pic1=<eventspec> [,sys[[0|1]]]
[,nouser[[0|1]]] [,noedge[[0|1]]] [,pc[[0|1]]] [,inv[[0|1]]] [,int[[0|1]]]
[,cmask[0|1]=<maskspec>] [,umask[0|1]=<maskspec>]
```

This syntax is a straightforward extension of the earlier syntax. The additional `inv`, `int`, `cmask0`, `cmask1`, `umask0`, and `umask1` keywords allow extended counting semantics. The mask specification is a number between 0 and 255, expressed in hexadecimal, octal or decimal notation.

Examples

SPARC

EXAMPLE 1 SPARC Example.

```
cpc_event_t event;
char *setting = "pic0=EC_ref,pic1=EC_hit"; /* UltraSPARC-specific */

if (cpc_strtoevent(cpuver, setting, &event) != 0)
        /* can't measure 'setting' on this processor */
else
        setting = cpc_eventtostr(&event);
```

Attributes

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

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

See Also

`cpc(3CPC)`, `cpc_getcpuver(3CPC)`, `cpc_set_add_request(3CPC)`, `cpc_seterrfn(3CPC)`, `free(3C)`, `getsubopt(3C)`, `libcpc(3LIB)`, `attributes(5)`

Notes The `cpc_strtoevent()` and `cpc_eventtostr()` functions exist for binary compatibility only. Source containing these functions will not compile. These functions are obsolete and might be removed in a future release. Applications should use `cpc_set_add_request`(3CPC) instead.

These functions are provided as a convenience only. As new processors are usually released asynchronously with software, the library allows the `pic0` and `pic1` keywords to interpret numeric values specified directly in hexadecimal, octal, or decimal.

Name cpc_version – coordinate CPC library and application versions

Synopsis

```
cc [ flag... ] file... –lcpc [ library... ]
#include <libcpc.h>

uint_t cpc_version(uint_t version);
```

Description The `cpc_version()` function takes an interface version as an argument and returns an interface version as a result. Usually, the argument will be the value of `CPC_VER_CURRENT` bound to the application when it was compiled.

Return Values If the version requested is still supported by the implementation, `cpc_version()` returns the requested version number and the application can use the facilities of the library on that platform. If the implementation cannot support the version needed by the application, `cpc_version()` returns `CPC_VER_NONE`, indicating that the application will at least need to be recompiled to operate correctly on the new platform, and may require further changes.

If *version* is `CPC_VER_NONE`, `cpc_version()` returns the most current version of the library.

Examples **EXAMPLE 1** Protect an application from using an incompatible library.

The following lines of code protect an application from using an incompatible library:

```
if (cpc_version(CPC_VER_CURRENT) == CPC_VER_NONE) {
        /* version mismatch - library cannot translate */
        exit(1);
}
```

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Unsafe

See Also `cpc(3CPC)`, `cpc_open(3CPC)`, `libcpc(3LIB)`, `attributes(5)`

Notes The `cpc_version()` function exists for binary compatibility only. Source containing this function will not compile. This function is obsolete and might be removed in a future release. Applications should use `cpc_open(3CPC)` instead.

The version number is used only to express incompatible semantic changes in the performance counter interfaces on the given platform within a single instruction set architecture, for example, when a new set of performance counter registers are added to an existing processor family that cannot be specified in the existing `cpc_event_t` data structure.

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 cpow, cpowf, cpowl – complex power functions

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <complex.h>

double complex cpow(double complex x, double complex y);

float complex cpowf(float complex x, float complex y);

long double complex cpowl(long double complex x,
     long double complex y);
```

Description These functions compute the complex power function x^y, with a branch cut for the first parameter along the negative real axis.

Return Values These functions return the complex power function value.

Errors No errors are defined.

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

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

See Also cabs(3M), complex.h(3HEAD), csqrt(3M), attributes(5), standards(5)

Name cproj, cprojf, cprojl – complex projection functions

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <complex.h>

double complex cproj(double complex z);

float complex cprojf(float complex z);

long double complex cprojl(long double complex z);
```

Description These functions compute a projection of *z* onto the Riemann sphere: *z* projects to *z*, except that all complex infinities (even those with one infinite part and one NaN part) project to positive infinity on the real axis. If *z* has an infinite part, then `cproj`(*z*) is equivalent to:

```
INFINITY + I * copysign(0.0, cimag(z))
```

Return Values These functions return the value of the projection onto the Riemann sphere.

Errors No errors are defined.

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

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

See Also `carg(3M)`, `cimag(3M)`, `complex.h(3HEAD)`, `conj(3M)`, `creal(3M)`, `attributes(5)`, `standards(5)`

Name creal, crealf, creall – complex real functions

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <complex.h>

double creal(double complex z);

float crealf(float complex z);

long double creall(long double complex z);
```

Description These functions compute the real part of z.

Return Values These functions return the real part value.

Errors No errors are defined.

Usage For a variable *z* of complex type:

```
z == creal(z) + cimag(z)*I
```

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

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

See Also carg(3M), cimag(3M), complex.h(3HEAD), conj(3M), cproj(3M), attributes(5), standards(5)

Name csin, csinf, csinl – complex sine functions

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <complex.h>

double complex csin(double complex z);

float complex csinf(float complex z);

long double complex csinl(long double complex z);
```

Description These functions compute the complex sine of *z*.

Return Values These functions return the complex sine value.

Errors No errors are defined.

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

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

See Also casin(3M), complex.h(3HEAD), attributes(5), standards(5)

Name csinh, csinhf, csinhl – complex hyperbolic sine functions

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <complex.h>

double complex csinh(double complex z);

float complex csinhf(float complex z);

long double complex csinhl(long double complex z);
```

Description These functions compute the complex hyperbolic sine of *z*.

Return Values These functions return the complex hyperbolic sine value.

Errors No errors are defined.

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

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

See Also casinh(3M), complex.h(3HEAD), attributes(5), standards(5)

Name csqrt, csqrtf, csqrtl – complex square root functions

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <complex.h>

double complex csqrt(double complex z);

float complex csqrtf(float complex z);

long double complex csqrtl(long double complex z);
```

Description These functions compute the complex square root of *z*, with a branch cut along the negative real axis.

Return Values These functions return the complex square root value, in the range of the right half-plane (including the imaginary axis).

Errors No errors are defined.

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

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

See Also cabs(3M), complex.h(3HEAD), cpow(3M), attributes(5), standards(5)

Name ctan, ctanf, ctanl – complex tangent functions

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <complex.h>

double complex ctan(double complex z);

float complex ctanf(float complex z);

long double complex ctanl(long double complex z);
```

Description These functions compute the complex tangent of *z*.

Return Values These functions return the complex tangent value.

Errors No errors are defined.

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

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

See Also catan(3M), complex.h(3HEAD), attributes(5), standards(5)

Name ctanh, ctanhf, ctanhl – complex hyperbolic tangent functions

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <complex.h>

double complex ctanh(double complex z);

float complex ctanhf(float complex z);

long double complex ctanhl(long double complex z);
```

Description These functions compute the complex hyperbolic tangent of *z*.

Return Values These functions return the complex hyperbolic tangent value.

Errors No errors are defined.

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

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

See Also catanh(3M), complex.h(3HEAD), attributes(5), standards(5)

Name ct_ctl_adopt, ct_ctl_abandon, ct_ctl_newct, ct_ctl_ack, ct_ctl_qack – common contract control functions

Synopsis

```
cc [ flag... ] file... -D_LARGEFILE64_SOURCE -lcontract [ library... ]
#include <libcontract.h>

int ct_ctl_adopt(int fd);

int ct_ctl_abandon(int fd);

int ct_ctl_newct(int fd, uint64_t evid);

int ct_ctl_ack(int fd, uint64_t evid);

int ct_ctl_qack(int fd, uint64_t evid, int templatefd);
```

Description These functions operate on contract control file descriptors obtained from the `contract(4)` file system.

The `ct_ctl_adopt()` function adopts the contract referenced by the file descriptor *fd*. After a successful call to `ct_ctl_adopt()`, the contract is owned by the calling process and any events in that contract's event queue are appended to the process's bundle of the appropriate type.

The `ct_ctl_abandon()` function abandons the contract referenced by the file descriptor *fd*. After a successful call to `ct_ctl_abandon()` the process no longer owns the contract, any events sent by that contract are automatically removed from the process's bundle, and any critical events on the contract's event queue are automatically acknowledged. Depending on its type and terms, the contract will either be orphaned or destroyed.

The `ct_ctl_ack()` function acknowledges the critical event specified by*evid*. If the event corresponds to an exit negotiation, `ct_ctl_ack()` also indicates that the caller is prepared for the system to proceed with the referenced reconfiguration.

The `ct_ctl_qack()` function requests a new quantum of time for the negotiation specified by the event ID *evid*.

The `ct_ctl_newct()` function instructs the contract specified by the file descriptor *fd* that when the current exit negotiation completes, another contract with the terms provided by the template specified by *templatefd* should be automatically written.

Return Values Upon successful completion, `ct_ctl_adopt()`, `ct_ctl_abandon()`, `ct_ctl_newct()`, `ct_ctl_ack()`, and `ct_ctl_qack()` return 0. Otherwise, they return a non-zero error value.

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

`EBUSY` The contract is in the owned state.

`EINVAL` The contract was not inherited by the caller's process contract or was created by a process in a different zone.

The `ct_ctl_abandon()`, `ct_ctl_newct()`, `ct_ctl_ack()`, and `ct_ctl_qack()` functions will fail if:

EBUSY The contract does not belong to the calling process.

The `ct_ctl_newct()` and `ct_ctl_qack()` functions will fail if:

ESRCH The event ID specified by *evid* does not correspond to an unacknowledged negotiation event.

The `ct_ctl_newct()` function will fail if:

EINVAL The file descriptor specified by *fd* was not a valid template file descriptor.

The `ct_ctl_ack()` function will fail if:

ESRCH The event ID specified by *evid* does not correspond to an unacknowledged critical event.

The `ct_ctl_qack()` function will fail if:

ERANGE The maximum amount of time has been requested.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Safe

See Also `libcontract(3LIB)`, `contract(4)`, `attributes(5)`, `lfcompile(5)`

Name ct_event_read, ct_event_read_critical, ct_event_reset, ct_event_reliable, ct_event_free, ct_event_get_flags, ct_event_get_ctid, ct_event_get_evid, ct_event_get_type, ct_event_get_nevid, ct_event_get_newct – common contract event functions

Synopsis

```
cc [ flag... ] file... -D_LARGEFILE64_SOURCE -lcontract [ library... ]
#include <libcontract.h>

int ct_event_read(int fd, ct_evthdl_t *evthndlp);

int ct_event_read_critical(int fd, ct_evthdl_t *evthndlp);

int ct_event_reset(int fd);

int ct_event_reliable(int fd);

void ct_event_free(ct_evthdl_t evthndl);

ctid_t ct_event_get_ctid(ct_evthdl_t evthndl);

ctevid_t ct_event_get_evid(ct_evthdl_t evthndl);

uint_t ct_event_get_flags(ct_evthdl_t evthndl);

uint_t ct_event_get_type(ct_evthdl_t evthndl);

int ct_event_get_nevid(ct_evthdl_t evthndl, ctevid_t *evidp);

int ct_event_get_newct(ct_evthdl_t evthndl, ctid_t *ctidp);
```

Description These functions operate on contract event endpoint file descriptors obtained from the `contract(4)` file system and event object handles returned by `ct_event_read()` and `ct_event_read_critical()`.

The `ct_event_read()` function reads the next event from the queue referenced by the file descriptor *fd* and initializes the event object handle pointed to by *evthndlp*. After a successful call to `ct_event_read()`, the caller is responsible for calling `ct_event_free()` on this event object handle when it has finished using it.

The `ct_event_read_critical()` function behaves like `ct_event_read()` except that it reads the next critical event from the queue, skipping any intermediate events.

The `ct_event_reset()` function resets the location of the listener to the beginning of the queue. This function can be used to re-read events, or read events that were sent before the event endpoint was opened. Informative and acknowledged critical events, however, might have been removed from the queue.

The `ct_event_reliable()` function indicates that no event published to the specified event queue should be dropped by the system until the specified listener has read the event. This function requires that the caller have the {`PRIV_CONTRACT_EVENT`} privilege in its effective set.

The `ct_event_free()` function frees any storage associated with the event object handle specified by *evthndl*.

The `ct_event_get_ctid()` function returns the ID of the contract that sent the specified event.

The `ct_event_get_evid()` function returns the ID of the specified event.

The `ct_event_get_flags()` function returns the event flags for the specified event. Valid event flags are:

`CTE_INFO` The event is an informative event.

`CTE_ACK` The event has been acknowledged (for critical and negotiation messages).

`CTE_NEG` The message represents an exit negotiation.

The `ct_event_get_type()` function reads the event type. The value is one of the event types described in `contract(4)` or the contract type's manual page.

The `ct_event_get_nevid()` function reads the negotiation ID from an `CT_EV_NEGEND` event.

The `ct_event_get_newct()` function obtains the ID of the contract created when the negotiation referenced by the `CT_EV_NEGEND` event succeeded. If no contract was created, *ctidp* will be 0. If the operation was cancelled, **ctidp* will equal the ID of the existing contract.

Return Values Upon successful completion, `ct_event_read()`, `ct_event_read_critical()`, `ct_event_reset()`, `ct_event_reliable()`, `ct_event_get_nevid()`, and `ct_event_get_newct()` return 0. Otherwise, they return a non-zero error value.

The `ct_event_get_flags()`, `ct_event_get_ctid()`, `ct_event_get_evid()`, and `ct_event_get_type()` functions return data as described in the DESCRIPTION.

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

`EPERM` The caller does not have {`PRIV_CONTRACT_EVENT`} in its effective set.

The `ct_event_read()` and `ct_event_read_critical()` functions will fail if:

`EAGAIN` The event endpoint was opened `O_NONBLOCK` and no applicable events were available to be read.

The The `ct_event_get_nevid()` and `ct_event_get_newct()` functions will fail if:

`EINVAL` The *evthndl* argument is not a `CT_EV_NEGEND` event object.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also libcontract(3LIB), contract(4), attributes(5), lfcompile(5)

Name ct_pr_event_get_pid, ct_pr_event_get_ppid, ct_pr_event_get_signal, ct_pr_event_get_sender, ct_pr_event_get_senderct, ct_pr_event_get_exitstatus, ct_pr_event_get_pcorefile, ct_pr_event_get_gcorefile, ct_pr_event_get_zcorefile – process contract event functions

Synopsis

```
cc [ flag... ] file... -D_LARGEFILE64_SOURCE -lcontract [ library... ]
#include <libcontract.h>
#include <sys/contract/process.h>

int ct_pr_event_get_pid(ct_evthdl_t evthdl, pid_t *pidp);

int ct_pr_event_get_ppid(ct_evthdl_t evthdl, pid_t *pidp);

int ct_pr_event_get_signal(ct_evthdl_t evthdl, int *signalp);

int ct_pr_event_get_sender(ct_evthdl_t evthdl, pid_t *pidp);

int ct_pr_event_get_senderct(ct_evthdl_t evthdl, ctid_t *pidp);

int ct_pr_event_get_exitstatus(ct_evthdl_t evthdl, int *statusp);

int ct_pr_event_get_pcorefile(ct_evthdl_t evthdl, char **namep);

int ct_pr_event_get_gcorefile(ct_evthdl_t evthdl, char **namep);

int ct_pr_event_get_zcorefile(ct_evthdl_t evthdl, char **namep);
```

Description These functions read process contract event information from an event object returned by `ct_event_read(3CONTRACT)` or `ct_event_read_critical(3CONTRACT)`.

The `ct_pr_event_get_pid()` function reads the process ID of the process generating the event.

The `ct_pr_event_get_ppid()` function reads the process ID of the process that forked the new process causing the `CT_PR_EV_FORK` event.

The `ct_pr_event_get_signal()` function reads the signal number of the signal that caused the `CT_PR_EV_SIGNAL` event.

The `ct_pr_event_get_sender()` function reads the process ID of the process that sent the signal that caused the `CT_PR_EV_SIGNAL` event. If the signal's sender was not in the same zone as the signal's recipient, this information is available only to event consumers in the global zone.

The ct_pr_event_get_senderct function reads the contract ID of the process that sent the signal that caused the CT_PR_EV_SIGNAL event. If the signal's sender was not in the same zone as the signal's recipient, this information is available only

The `ct_pr_event_get_exitstatus()` function reads the exit status of the process generating a `CT_PR_EV_EXIT` event.

The `ct_pr_event_get_pcorefile()` function reads the name of the process core file if one was created when the `CT_PR_EV_CORE` event was generated. A pointer to a character array is stored in **namep* and is freed when `ct_event_free(3CONTRACT)` is called on the event handle.

The `ct_pr_event_get_gcorefile()` function reads the name of the zone's global core file if one was created when the `CT_PR_EV_CORE` event was generated. A pointer to a character array is stored in **namep* and is freed when `ct_event_free()` is called on the event handle.

The `ct_pr_event_get_zcorefile()` function reads the name of the system-wide core file in the global zone if one was created when the `CT_PR_EV_CORE` event was generated. This information is available only to event consumers in the global zone. A pointer to a character array is stored in **namep* and is freed when `ct_event_free()` is called on the event handle.

Return Values Upon successful completion, `ct_pr_event_get_pid()`, `ct_pr_event_get_ppid()`, `ct_pr_event_get_signal()`, `ct_pr_event_get_sender()`, `ct_pr_event_get_senderct()`, `ct_pr_event_get_exitstatus()`, `ct_pr_event_get_pcorefile()`, `ct_pr_event_get_gcorefile()`, and `ct_pr_event_get_zcorefile()` return 0. Otherwise, they return a non-zero error value.

Errors The `ct_pr_event_get_pid()`, `ct_pr_event_get_ppid()`, `ct_pr_event_get_signal()`, `ct_pr_event_get_sender()`, `ct_pr_event_get_senderct()`, `ct_pr_event_get_exitstatus()`, `ct_pr_event_get_pcorefile()`, `ct_pr_event_get_gcorefile()`, and `ct_pr_event_get_zcorefile()` functions will fail if:

`EINVAL` The *evthdl* argument is not a process contract event object.

The `ct_pr_event_get_ppid()`, `ct_pr_event_get_signal()`, `ct_pr_event_get_sender()`, `ct_pr_event_get_senderct()`, `ct_pr_event_get_exitstatus()`, `ct_pr_event_get_pcorefile()`, `ct_pr_event_get_gcorefile()`, and `ct_pr_event_get_zcorefile()` functions will fail if:

`EINVAL` The requested data do not match the event type.

The `ct_pr_event_get_sender()`a functions will fail if:

`ENOENT` The process ID of the sender was not available, or the event object was read by a process running in a non-global zone and the sender was in a different zone.

The `ct_pr_event_get_pcorefile()`, `ct_pr_event_get_gcorefile()`, and `ct_pr_event_get_zcorefile()` functions will fail if:

`ENOENT` The requested core file was not created.

The `ct_pr_event_get_zcorefile()` function will fail if:

`ENOENT` The event object was read by a process running in a non-global zone.

Examples

EXAMPLE 1 Print the instigator of all `CT_PR_EV_SIGNAL` events.

Open the process contract bundle. Loop reading events. Fetch and display the signalled pid and signalling pid for each `CT_PR_EV_SIGNAL` event encountered.

```
#include <sys/types.h>
#include <fcntl.h>
#include <stdio.h>
#include <libcontract.h>

...
int fd;
ct_evthdl_t event;
pid_t pid, sender;

fd = open("/system/contract/process/bundle", O_RDONLY);
for (;;) {
        ct_event_read(fd, &event);
        if (ct_event_get_type(event) != CT_PR_EV_SIGNAL) {
                ct_event_free(event);
                continue;
        }
        ct_pr_event_get_pid(event, &pid);
        if (ct_pr_event_get_sender(event, &sender) == ENOENT)
                printf("process %ld killed by unknown process\n",
                    (long)pid);
        else
                printf("process %ld killed by process %ld\n",
                    (long)pid, (long)sender);
        ct_event_free(event);
}
        ...
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Safe

See Also

`ct_event_free(3CONTRACT)`, `ct_event_read(3CONTRACT)`, `ct_event_read_critical(3CONTRACT)`, `libcontract(3LIB)`, `contract(4)`, `process(4)`, `attributes(5)`, `lfcompile(5)`

Name ct_pr_status_get_param, ct_pr_status_get_fatal, ct_pr_status_get_members, ct_pr_status_get_contracts – process contract status functions

Synopsis

```
cc [ flag... ] file... -D_LARGEFILE64_SOURCE -lcontract [ library... ]
#include <libcontract.h>
#include <sys/contract/process.h>

int ct_pr_status_get_param(ct_stathdl_t stathdl, uint_t *paramp);

int ct_pr_status_get_fatal(ct_stathdl_t stathdl, uint_t *eventsp);

int ct_pr_status_get_members(ct_stathdl_t stathdl,
     pid_t **pidpp, uint_t *n);

int ct_pr_status_get_contracts(ct_stathdl_t stathdl,
     ctid_t **idpp, uint_t *n);
```

Description These functions read process contract status information from a status object returned by `ct_status_read(3CONTRACT)`.

The `ct_pr_status_get_param()` function reads the parameter set term. The value is a collection of bits as described in `process(4)`.

The `ct_pr_status_get_fatal()` function reads the fatal event set term. The value is a collection of bits as described in `process(4)`.

The `ct_pr_status_get_members()` function obtains a list of the process IDs of the members of the process contract. A pointer to an array of process IDs is stored in **pidpp*. The number of elements in this array is stored in **n*. These data are freed when the status object is freed by a call to `ct_status_free(3CONTRACT)`.

The `ct_pr_status_get_contracts()` function obtains a list of IDs of contracts that have been inherited by the contract. A pointer to an array of IDs is stored in **idpp*. The number of elements in this array is stored in **n*. These data are freed when the status object is freed by a call to `ct_status_free()`.

Return Values Upon successful completion, `ct_pr_status_get_param()`, `ct_pr_status_get_fatal()`, `ct_pr_status_get_members()`, and `ct_pr_status_get_contracts()` return 0. Otherwise, they return a non-zero error value.

Errors The `ct_pr_status_get_param()`, `ct_pr_status_get_fatal()`, `ct_pr_status_get_members()`, and `ct_pr_status_get_contracts()` functions will fail if:

`EINVAL` The *stathdl* argument is not a process contract status object.

The `ct_pr_status_get_param()`, `ct_pr_status_get_fatal()`, `ct_pr_status_get_members()`, and `ct_r_status_get_contracts()` functions will fail if:

`ENOENT` The requested data were not available in the status object.

Examples

EXAMPLE 1 Print members of process contract 1.

Open the status file for contract 1, read the contract's status, obtain the list of processes, print them, and free the status object.

```
#include <sys/types.h>
#include <fcntl.h>
#include <libcontract.h>
#include <stdio.h>

...
int fd;
uint_t i, n;
pid_t *procs;
ct_stathdl_t st;

fd = open("/system/contract/process/1/status");
ct_status_read(fd, &st);
ct_pr_status_get_members(st, &procs, &n);
for (i = 0 ; i < n; i++)
        printf("%ld\n", (long)procs[i]);
ct_status_free(stat);
close(fd);
...
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Safe

See Also

ct_status_free(3CONTRACT), ct_status_read(3CONTRACT), libcontract(3LIB), contract(4), process(4), attributes(5), lfcompile(5)

Name ct_pr_tmpl_set_transfer, ct_pr_tmpl_set_fatal, ct_pr_tmpl_set_param, ct_pr_tmpl_get_transfer, ct_pr_tmpl_get_fatal, ct_pr_tmpl_get_param – process contract template functions

Synopsis

```
cc [ flag... ] file... -D_LARGEFILE64_SOURCE -lcontract [ library... ]
#include <libcontract.h>
#include <sys/contract/process.h>

int ct_pr_tmpl_set_transfer(int fd, ctid_t ctid);

int ct_pr_tmpl_set_fatal(int fd, uint_t events);

int ct_pr_tmpl_set_param(int fd, uint_t params);

int ct_pr_tmpl_get_transfer(int fd, ctid_t *ctidp);

int ct_pr_tmpl_get_fatal(int fd, uint_t *eventsp);

int ct_pr_tmpl_get_param(int fd, uint_t *paramsp);
```

Description These functions read and write process contract terms and operate on process contract template file descriptors obtained from the `contract(4)` file system.

The `ct_pr_tmpl_set_transfer()` and `ct_pr_tmpl_get_transfer()` functions write and read the transfer contract term. The value is the ID of an empty regent process contract owned by the caller whose inherited contracts are to be transferred to a newly created contract.

The `ct_pr_tmpl_set_fatal()` and `ct_pr_tmpl_get_fatal()` functions write and read the fatal event set term. The value is a collection of bits as described in `process(4)`.

The `ct_pr_tmpl_set_param()` and `ct_pr_tmpl_get_param()` functions write and read the parameter set term. The value is a collection of bits as described in `process(4)`.

Return Values Upon successful completion, `ct_pr_tmpl_set_transfer()`, `ct_pr_tmpl_set_fatal()`, `ct_pr_tmpl_set_param()`, `ct_pr_tmpl_get_transfer()`, `ct_pr_tmpl_get_fatal()`, and `ct_pr_tmpl_get_param()` return 0. Otherwise, they return a non-zero error value.

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

`EINVAL` An invalid parameter was specified.

The `ct_pr_tmpl_set_fatal()` function will fail if:

`EINVAL` An invalid event was specified.

The `ct_pr_tmpl_set_transfer()` function will fail if:

`ESRCH` The ID specified by *ctid* does not correspond to a process contract.

`EACCES` The ID specified by *ctid* does not correspond to a process contract owned by the calling process.

`ENOTEMPTY` The ID specified by *ctid* does not correspond to an empty process contract.

Examples

EXAMPLE 1 Create and activate a process contract template.

The following example opens a new template, makes hardware errors and signals fatal events, makes hardware errors critical events, and activates the template. It then forks a process in the new contract using the requested terms.

```
#include <libcontract.h>
#include <fcntl.h>
#include <unistd.h>

...
int fd;

fd = open("/system/contract/process/template", O_RDWR);
(void) ct_pr_tmpl_set_fatal(fd, CT_PR_EV_HWERR|CT_PR_EV_SIGNAL);
(void) ct_tmpl_set_critical(fd, CT_PR_EV_HWERR);
(void) ct_tmpl_activate(fd);
close(fd);

if (fork()) {
        /* parent - owns new process contract */
        ...
} else {
        /* child - in new process contract */
        ...
}
...
```

EXAMPLE 2 Clear the process contract template.

The following example opens the template file and requests that the active template be cleared.

```
#include <libcontract.h>
#include <fcntl.h>

...
int fd;

fd = open("/system/contract/process/template", O_RDWR);
(void) ct_tmpl_clear(fd);
close(fd);
...
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also libcontract(3LIB), contract(4), process(4), attributes(5), lfcompile(5)

Name ct_status_read, ct_status_free, ct_status_get_id, ct_status_get_zoneid, ct_status_get_type, ct_status_get_state, ct_status_get_holder, ct_status_get_nevents, ct_status_get_ntime, ct_status_get_qtime, ct_status_get_nevid, ct_status_get_cookie, ct_status_get_informative, ct_status_get_critical – common contract status functions

Synopsis

```
cc [ flag... ] file... -D_LARGEFILE64_SOURCE  -lcontract  [ library... ]
#include <libcontract.h>

int ct_status_read(int fd, int detail, ct_stathdl_t *stathdlp);

void ct_status_free(ct_stathdl_t stathdl);

ctid_t ct_status_get_id(ct_stathdl_t stathdl);

zoneid_t ct_status_get_zoneid(ct_stathdl_t stathdl);

char *ct_status_get_type(ct_stathdl_t stathdl);

uint_t ct_status_get_state(ct_stathdl_t stathdl);

pid_t ct_status_get_holder(ct_stathdl_t stathdl);

int ct_status_get_nevents(ct_stathdl_t stathdl);

int ct_status_get_ntime(ct_stathdl_t stathdl);

int ct_status_get_qtime(ct_stathdl_t stathdl);

ctevid_t ct_status_get_nevid(ct_stathdl_t stathdl);

uint64_t ct_status_get_cookie(ct_stathdl_t stathdl);

ctevid_t ct_status_get_informative(ct_stathdl_t stathdl);

uint_t ct_status_get_critical(ct_stathdl_t stathdl);
```

Description These functions operate on contract status file descriptors obtained from the `contract(4)` file system and status object handles returned by `ct_status_read()`.

The `ct_status_read()` function reads the contract's status and initializes the status object handle pointed to by *stathdlp*. After a successful call to `ct_status_read()`, the caller is responsible for calling `ct_status_free()` on this status object handle when it has finished using it. Because the amount of information available for a contract might be large, the *detail* argument allows the caller to specify how much information `ct_status_read()` should obtain. A value of `CTD_COMMON` fetches only those data accessible by the functions on this manual page. `CTD_FIXED` fetches `CTD_COMMON` data as well as fixed-size contract type-specific data. `CTD_ALL` fetches `CTD_FIXED` data as well as variable lengthed data, such as arrays. See the manual pages for contract type-specific status accessor functions for information concerning which data are fetched by `CTD_FIXED` and `CTD_ALL`.

The `ct_status_free()` function frees any storage associated with the specified status object handle.

The remaining functions all return contract information obtained from a status object.

The `ct_status_get_id()` function returns the contract's ID.

The `ct_status_get_zoneid()` function returns the contract's creator's zone ID, or −1 if the creator's zone no longer exists.

The `ct_status_get_type()` function returns the contract's type. The string should be neither modified nor freed.

The `ct_status_get_state()` function returns the state of the contract. Valid state values are:

`CTS_OWNED`	a contract that is currently owned by a process
`CTS_INHERITED`	a contract that has been inherited by a regent process contract
`CTS_ORPHAN`	a contract that has no owner and has not been inherited
`CTS_DEAD`	a contract that is no longer in effect and will be automatically removed from the system as soon as the last reference to it is release (for example, an open status file descriptor)

The `ct_status_get_holder()` function returns the process ID of the contract's owner if the contract is in the `CTS_OWNED` state, or the ID of the regent process contract if the contract is in the `CTS_INHERITED` state.

The `ct_status_get_nevents()` function returns the number of unacknowledged critical events on the contract's event queue.

The `ct_status_get_ntime()` function returns the amount of time remaining (in seconds) before the ongoing exit negotiation times out, or -1 if there is no negotiation ongoing.

The `ct_status_get_qtime()` function returns the amount of time remaining (in seconds) in the quantum before the ongoing exit negotiation times out, or -1 if there is no negotiation ongoing.

The `ct_status_get_nevid()` function returns the event ID of the ongoing negotiation, or 0 if there are none.

The `ct_status_get_cookie()` function returns the cookie term of the contract.

The `ct_status_get_critical()` function is used to read the critical event set term. The value is a collection of bits as described in the contract type's manual page.

The `ct_status_get_informative()` function is used to read the informative event set term. The value is a collection of bits as described in the contract type's manual page.

Return Values Upon successful completion, `ct_status_read()` returns 0. Otherwise, it returns a non-zero error value.

Upon successful completion, `ct_status_get_id()`, `ct_status_get_type()`, `ct_status_get_holder()`, `ct_status_get_state()`, `ct_status_get_nevents()`, `ct_status_get_ntime()`, `ct_status_get_qtime()`, `ct_status_get_nevid()`, `ct_status_get_cookie()`, `ct_status_get_critical()`, and `ct_status_get_informative()` return the data described in the DESCRIPTION.

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

`EINVAL` The *detail* level specified is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Safe

See Also `libcontract(3LIB)`, `contract(4)`, `attributes(5)`, `lfcompile(5)`

Name ct_tmpl_activate, ct_tmpl_clear, ct_tmpl_create, ct_tmpl_set_cookie, ct_tmpl_set_critical, ct_tmpl_set_informative, ct_tmpl_get_cookie, ct_tmpl_get_critical, ct_tmpl_get_informative – common contract template functions

Synopsis

```
cc [ flag... ] file... -D_LARGEFILE64_SOURCE -lcontract [ library... ]
#include <libcontract.h>

int ct_tmpl_activate(int fd);

int ct_tmpl_clear(int fd);

int ct_tmpl_create(int fd, ctid_t *idp);

int ct_tmpl_set_cookie(int fd, uint64_t cookie);

int ct_tmpl_set_critical(int fd, uint_t events);

int ct_tmpl_set_informative(int fd, uint_t events);

int ct_tmpl_get_cookie(int fd, uint64_t *cookiep);

int ct_tmpl_get_critical(int fd, uint_t *eventsp);

int ct_tmpl_get_informative(int fd, uint_t *eventsp);
```

Description These functions operate on contract template file descriptors obtained from the `contract(4)` file system.

The `ct_tmpl_activate()` function makes the template referenced by the file descriptor *fd* the active template for the calling thread.

The `ct_tmpl_clear()` function clears calling thread's active template.

The `ct_tmpl_create()` function uses the template referenced by the file descriptor *fd* to create a new contract. If successful, the ID of the newly created contract is placed in **idp*.

The `ct_tmpl_set_cookie()` and `ct_tmpl_get_cookie()` functions write and read the cookie term of a contract template. The cookie term is ignored by the system, except to include its value in a resulting contract's status object. The default cookie term is 0.

The `ct_tmpl_set_critical()` and `ct_tmpl_get_critical()` functions write and read the critical event set term. The value is a collection of bits as described in the contract type's manual page.

The `ct_tmpl_set_informative()` and `ct_tmpl_get_informative()` functions write and read the informative event set term. The value is a collection of bits as described in the contract type's manual page.

Return Values Upon successful completion, `ct_tmpl_activate()`, `ct_tmpl_create()`, `ct_tmpl_set_cookie()`, `ct_tmpl_get_cookie()`, `ct_tmpl_set_critical()`, `ct_tmpl_get_critical()`, `ct_tmpl_set_informative()`, and `ct_tmpl_get_informative()` return 0. Otherwise, they return a non-zero error value.

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

`ERANGE` The terms specified in the template were unsatisfied at the time of the call.

`EAGAIN` The *project.max-contracts* resource control would have been exceeded.

The `ct_tmpl_set_critical()` and `ct_tmpl_set_informative()` functions will fail if:

`EINVAL` An invalid event was specified.

The `ct_tmpl_set_critical()` function will fail if:

`EPERM` One of the specified events was disallowed given other contract terms (see `contract(4)`) and {`PRIV_CONTRACT_EVENT`} was not in the effective set for the calling process.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Safe

See Also `libcontract(3LIB)`, `contract(4)`, `attributes(5)`, `lfcompile(5)`

Name dat_cno_create – create a CNO instance

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_cno_create (
    IN    DAT_IA_HANDLE             ia_handle,
    IN    DAT_OS_WAIT_PROXY_AGENT   agent,
    OUT   DAT_CNO_HANDLE            *cno_handle
    )
```

Parameters

ia_handle Handle for an instance of DAT IA.

agent An optional OS Wait Proxy Agent that is to be invoked whenever CNO is invoked. `DAT_OS_WAIT_PROXY_AGENT_NULL` indicates that there is no proxy agent

cno_handle Handle for the created instance of CNO.

Description The `dat_cno_create()` function creates a CNO instance. Upon creation, there are no Event Dispatchers feeding it.

The *agent* parameter specifies the proxy agent, which is OS-dependent and which is invoked when the CNO is triggered. After it is invoked, it is no longer associated with the CNO. The value of `DAT_OS_WAIT_PROXY_AGENT_NULL` specifies that no OS Wait Proxy Agent is associated with the created CNO.

Upon creation, the CNO is not associated with any EVDs, has no waiters and has, at most, one OS Wait Proxy Agent.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INSUFFICIENT_RESOURCES` The operation failed due to resource limitations.

`DAT_INVALID_HANDLE` The *ia_handle* parameter is invalid.

`DAT_INVALID_PARAMETER` One of the parameters was invalid, out of range, or a combination of parameters was invalid, or the *agent* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Safe

See Also libdat(3LIB), attributes(5)

Name dat_cno_free – destroy an instance of the CNO

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_cno_free (
    IN    DAT_CNO_HANDLE     cno_handle
    )
```

Parameters *cno_handle* Handle for an instance of the CNO

Description The `dat_cno_free()` function destroys a specified instance of the CNO.

A CNO cannot be deleted while it is referenced by an Event Dispatcher or while a thread is blocked on it.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INVALID_HANDLE`	The `cno_handle()` parameter is invalid.
`DAT_INVALID_STATE`	Parameter in an invalid state. CNO is in use by an EVD instance or there is a thread blocked on it.

Usage If there is a thread blocked in `dat_cno_wait(3DAT)`, the Consumer can do the following steps to unblock the waiter:

- Create a temporary EVD that accepts software events. It can be created in advance.
- For a CNO with the waiter, attach that EVD to the CNO and post the software event on the EVD.
- This unblocks the CNO.
- Repeat for other CNOs that have blocked waiters.
- Destroy the temporary EVD after all CNOs are destroyed and the EVD is no longer needed.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Unsafe

See Also `dat_cno_wait(3DAT)`, `libdat(3LIB)`, `attributes(5)`

Name dat_cno_modify_agent – modify the OS Wait Proxy Agent

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_cno_modify_agent (
    IN    DAT_CNO_HANDLE            cno_handle,
    IN    DAT_OS_WAIT_PROXY_AGENT   agent
    )
```

Parameters

cno_handle Handle for an instance of CNO

agent Pointer to an optional OS Wait Proxy Agent to invoke whenever CNO is invoked. `DAT_OS_WAIT_PROXY_AGENT_NULL` indicates that there is no proxy agent.

Description The `dat_cno_modify_agent()` function modifies the OS Wait Proxy Agent associated with a CNO. If non-null, any trigger received by the CNO is also passed to the OS Wait Proxy Agent. This is in addition to any local delivery through the CNO. The Consumer can pass the value of `DAT_OS_WAIT_PROXY_AGENT_NULL` to disassociate the current Proxy agent from the CNO

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *cno_handle* parameter is invalid.

`DAT_INVALID_PARAMETER` One of the parameters was invalid, out of range, or a combination of parameters was invalid, or the *agent* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Unsafe

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_cno_query – provide the Consumer parameters of the CNO

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_cno_query (
    IN    DAT_CNO_HANDLE          cno_handle,
    IN    DAT_CNO_PARAM_MASK      cno_param_mask,
    OUT   DAT_CNO_PARAM           *cno_param
    )
```

Parameters

cno_handle	Handle for the created instance of the Consumer Notification Object
cno_param_mask	Mask for CNO parameters
cno_param	Pointer to a Consumer-allocated structure that the Provider fills with CNO parameters

Description The `dat_cno_query()` function provides the Consumer parameters of the CNO. The Consumer passes in a pointer to the Consumer-allocated structures for CNO parameters that the Provider fills.

The *cno_param_mask* parameter allows Consumers to specify which parameters to query. The Provider returns values for *cno_param_mask* requested parameters. The Provider can return values for any other parameters.

A value of `DAT_OS_WAIT_PROXY_AGENT_NULL` in *cno_param* indicates that there are no Proxy Agent associated with the CNO.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INVALID_PARAMETER`	The *cno_param_mask* parameter is invalid.
`DAT_INVALID_HANDLE`	The *cno_handle* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Unsafe

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_cno_wait – wait for notification events

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_cno_wait (
    IN    DAT_CNO_HANDLE    cno_handle,
    IN    DAT_TIMEOUT       timeout,
    OUT   DAT_EVD_HANDLE    *evd_handle
    )
```

Parameters

cno_handle Handle for an instance of CNO

timeout The duration to wait for a notification. The value `DAT_TIMEOUT_INFINITE` can be used to wait indefinitely.

evd_handle Handle for an instance of EVD

Description The `dat_cno_wait()` function allows the Consumer to wait for notification events from a set of Event Dispatchers all from the same Interface Adapter. The Consumer blocks until notified or the timeout period expires.

An Event Dispatcher that is disabled or in the "Waited" state does not deliver notifications. A uDAPL Consumer waiting directly upon an Event Dispatcher preempts the CNO.

The consumer can optionally specify a timeout, after which it is unblocked even if there are no notification events. On a timeout, *evd_handle* is explicitly set to a null handle.

The returned *evd_handle* is only a hint. Another Consumer can reap the Event before this Consumer can get around to checking the Event Dispatcher. Additionally, other Event Dispatchers feeding this CNO might have been notified. The Consumer is responsible for ensuring that all EVDs feeding this CNO are polled regardless of whether they are identified as the immediate cause of the CNO unblocking.

All the waiters on the CNO, including the OS Wait Proxy Agent if it is associated with the CNO, are unblocked with the NULL handle returns for an unblocking EVD *evd_handle* when the IA instance is destroyed or when all EVDs the CNO is associated with are freed.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *cno_handle* parameter is invalid.

`DAT_QUEUE_EMPTY` The operation timed out without a notification.

`DAT_INVALID_PARAMETER` One of the parameters was invalid or out of range, a combination of parameters was invalid, or the *timeout* parameter is invalid.

`DAT_INTERRUPTED_CALL` The operation was interrupted by a signal.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Safe

See Also libdat(3LIB), attributes(5)

Name dat_cr_accept – establishes a Connection between the active remote side requesting Endpoint and the passive side local Endpoint

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_cr_accept (
    IN    DAT_CR_HANDLE    cr_handle,
    IN    DAT_EP_HANDLE    ep_handle,
    IN    DAT_COUNT        private_data_size,
    IN const DAT_PVOID     private_data
    )
```

Parameters

cr_handle Handle to an instance of a Connection Request that the Consumer is accepting.

ep_handle Handle for an instance of a local Endpoint that the Consumer is accepting the Connection Request on. If the local Endpoint is specified by the Connection Request, the *ep_handle* shall be `DAT_HANDLE_NULL`.

private_data_size Size of the *private_data*, which must be nonnegative.

private_data Pointer to the private data that should be provided to the remote Consumer when the Connection is established. If *private_data_size* is zero, then *private_data* can be `NULL`.

Description The `dat_cr_accept()` function establishes a Connection between the active remote side requesting Endpoint and the passive side local Endpoint. The local Endpoint is either specified explicitly by *ep_handle* or implicitly by a Connection Request. In the second case, *ep_handle* is `DAT_HANDLE_NULL`.

Consumers can specify private data that is provided to the remote side upon Connection establishment.

If the provided local Endpoint does not satisfy the requested Connection Request, the operation fails without any effect on the local Endpoint, Pending Connection Request, private data, or remote Endpoint.

The operation is asynchronous. The successful completion of the operation is reported through a Connection Event of type `DAT_CONNECTION_EVENT_ESTABLISHED` on the *connect_evd* of the local Endpoint.

If the Provider cannot complete the Connection establishment, the connection is not established and the Consumer is notified through a Connection Event of type `DAT_CONNECTION_EVENT_ACCEPT_COMPLETION_ERROR` on the *connect_evd* of the local Endpoint. It can be caused by the active side timeout expiration, transport error, or any other

reason. If Connection is not established, Endpoint transitions into Disconnected state and all posted Recv DTOs are flushed to its *recv_evd_handle*.

This operation, if successful, also destroys the Connection Request instance. Use of the handle of the destroyed *cr_handle* in any consequent operation fails.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INVALID_HANDLE`	The *cr_handle* or *ep_handle* parameter is invalid.
`DAT_INVALID_PARAMETER`	The *private_data_size* or *private_data* parameter is invalid, out of range, or a combination of parameters was invalid

Usage

Consumers should be aware that Connection establishment might fail in the following cases: If the accepting Endpoint has an outstanding RDMA Read outgoing attribute larger than the requesting remote Endpoint or outstanding RDMA Read incoming attribute, or if the outstanding RDMA Read incoming attribute is smaller than the requesting remote Endpoint or outstanding RDMA Read outgoing attribute.

Consumers should set the accepting Endpoint RDMA Reads as the target (incoming) to a number larger than or equal to the remote Endpoint RDMA Read outstanding as the originator (outgoing), and the accepting Endpoint RDMA Reads as the originator to a number smaller than or equal to the remote Endpoint RDMA Read outstanding as the target. DAT API does not define a protocol on how remote peers exchange Endpoint attributes. The exchange of outstanding RDMA Read incoming and outgoing attributes of EPs is left to the Consumer ULP. Consumer can use Private Data for it.

If the Consumer does not care about posting RDMA Read operations or remote RDMA Read operations on the connection, it can set the two outstanding RDMA Read attribute values to 0.

If the Consumer does not set the two outstanding RDMA Read attributes of the Endpoint, the Provider is free to pick up any value for default. The Provider can change these default values during connection setup.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Unsafe

See Also

`libdat(3LIB)`, `attributes(5)`

Name dat_cr_handoff – hand off the Connection Request to another Service Point

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_cr_handoff (
    IN    DAT_CR_HANDLE    cr_handle,
    IN    DAT_CONN_QUAL    handoff
    )
```

Parameters

cr_handle Handle to an instance of a Connection Request that the Consumer is handing off.

handoff Indicator of another Connection Qualifier on the same IA to which this Connection Request should be handed off.

Description The `dat_cr_handoff()` function hands off the Connection Request to another Service Point specified by the Connection Qualifier *handoff*.

The operation is synchronous. This operation also destroys the Connection Request instance. Use of the handle of the destroyed Connection Request in any consequent operation fails.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *cr_handle* parameter is invalid.

`DAT_INVALID_PARAMETER` The *handoff* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Unsafe

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_cr_query – provide parameters of the Connection Request

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_cr_query (
    IN    DAT_CR_HANDLE        cr_handle,
    IN    DAT_CR_PARAM_MASK    cr_param_mask,
    OUT   DAT_CR_PARAM         *cr_param
    )
```

Parameters

cr_handle	Handle for an instance of a Connection Request.
cr_param_mask	Mask for Connection Request parameters.
cr_param	Pointer to a Consumer-allocated structure that the Provider fills for Consumer-requested parameters.

Description The `dat_cr_query()` function provides to the Consumer parameters of the Connection Request. The Consumer passes in a pointer to the Consumer-allocated structures for Connection Request parameters that the Provider fills.

The *cr_param_mask* parameter allows Consumers to specify which parameters to query. The Provider returns values for *cr_param_mask* requested parameters. The Provider can return values for any other parameters.

Return Values

`DAT_SUCCESS`	The operation was successful
`DAT_INVALID_HANDLE`	The *cr_handle* handle is invalid.
`DAT_INVALID_PARAMETER`	The *cr_param_mask* parameter is invalid.

Usage The Consumer uses `dat_cr_query()` to get information about requesting a remote Endpoint as well as a local Endpoint if it was allocated by the Provider for the arrived Connection Request. The local Endpoint is created if the Consumer used PSP with `DAT_PSP_PROVIDER` as the value for *psp_flags*. For the remote Endpoint, `dat_cr_query()` provides *remote_ia_address* and *remote_port_qual*. It also provides remote *peer private_data* and its size.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Safe

See Also libdat(3LIB), attributes(5)

Name dat_cr_reject – reject a Connection Request from the Active remote side requesting Endpoint

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_cr_reject (
    IN    DAT_CR_HANDLE    cr_handle
    )
```

Parameters *cr_handle* Handle to an instance of a Connection Request that the Consumer is rejecting.

Description The `dat_cr_reject()` function rejects a Connection Request from the Active remote side requesting Endpoint. If the Provider passed a local Endpoint into a Consumer for the Public Service Point-created Connection Request, that Endpoint reverts to Provider Control. The behavior of an operation on that Endpoint is undefined. The local Endpoint that the Consumer provided for Reserved Service Point reverts to Consumer control, and the Consumer is free to use in any way it wants.

The operation is synchronous. This operation also destroys the Connection Request instance. Use of the handle of the destroyed Connection Request in any consequent operation fails.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *cr_handle* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Unsafe

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_ep_connect – establish a connection between the local Endpoint and a remote Endpoint

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ep_connect (
    IN    DAT_EP_HANDLE         ep_handle,
    IN    DAT_IA_ADDRESS_PTR    remote_ia_address,
    IN    DAT_CONN_QUAL         remote_conn_qual,
    IN    DAT_TIMEOUT           timeout,
    IN    DAT_COUNT             private_data_size,
    IN const  DAT_PVOID         private_data,
    IN    DAT_QOS               qos,
    IN    DAT_CONNECT_FLAGS     connect_flags
    )
```

Parameters

ep_handle	Handle for an instance of an Endpoint.
remote_ia_address	The Address of the remote IA to which an Endpoint is requesting a connection.
remote_conn_qual	Connection Qualifier of the remote IA from which an Endpoint requests a connection.
timeout	Duration of time, in microseconds, that a Consumer waits for Connection establishment. The value of `DAT_TIMEOUT_INFINITE` represents no timeout, indefinite wait. Values must be positive.
private_data_size	Size of the *private_data*. Must be nonnegative.
private_data	Pointer to the private data that should be provided to the remote Consumer as part of the Connection Request. If *private_data_size* is zero, then *private_data* can be `NULL`.
qos	Requested quality of service of the connection.
connect_flags	Flags for the requested connection. If the least significant bit of `DAT_MULTIPATH_FLAG` is 0, the Consumer does not request multipathing. If the least significant bit of `DAT__MULTIPATH_FLAG` is 1, the Consumer requests multipathing. The default value is `DAT_CONNECT_DEFAULT_FLAG`, which is 0.

Description The `dat_ep_connect()` function requests that a connection be established between the local Endpoint and a remote Endpoint. This operation is used by the active/client side Consumer of the Connection establishment model. The remote Endpoint is identified by Remote IA and Remote Connection Qualifier.

As part of the successful completion of this operation, the local Endpoint is bound to a Port Qualifier of the local IA. The Port Qualifier is passed to the remote side of the requested connection and is available to the remote Consumer in the Connection Request of the DAT_CONNECTION_REQUEST_EVENT.

The Consumer-provided *private_data* is passed to the remote side and is provided to the remote Consumer in the Connection Request. Consumers can encapsulate any local Endpoint attributes that remote Consumers need to know as part of an upper-level protocol. Providers can also provide a Provider on the remote side any local Endpoint attributes and Transport-specific information needed for Connection establishment by the Transport.

Upon successful completion of this operation, the local Endpoint is transferred into DAT_EP_STATE_ACTIVE_CONNECTION_PENDING.

Consumers can request a specific value of *qos*. The Provider specifies which quality of service it supports in documentation and in the Provider attributes. If the local Provider or Transport does not support the requested *qos*, the operation fails and DAT_MODEL_NOT_SUPPORTED is returned synchronously. If the remote Provider does not support the requested *qos*, the local Endpoint is automatically transitioned into the DAT_EP_STATE_DISCONNECTED state, the connection is not established, and the event returned on the *connect_evd_handle* is DAT_CONNECTION_EVENT_NON_PEER_REJECTED. The same DAT_CONNECTION_EVENT_NON_PEER_REJECTED event is returned if the connection cannot be established for all reasons of not establishing the connection, except timeout, remote host not reachable, and remote peer reject. For example, remote Consumer is not listening on the requested Connection Qualifier, Backlog of the requested Service Point is full, and Transport errors. In this case, the local Endpoint is automatically transitioned into DAT_EP_STATE_DISCONNECTED state.

The acceptance of the requested connection by the remote Consumer is reported to the local Consumer through a DAT_CONNECTION_EVENT_ESTABLISHED event on the *connect_evd_handle* of the local Endpoint and the local Endpoint is automatically transitioned into a DAT_EP_STATE_CONNECTED state.

The rejection of the connection by the remote Consumer is reported to the local Consumer through a DAT_CONNECTION_EVENT_PEER_REJECTED event on the *connect_evd_handle* of the local Endpoint and the local Endpoint is automatically transitioned into a DAT_EP_STATE_DISCONNECTED state.

When the Provider cannot reach the remote host or the remote host does not respond within the Consumer requested Timeout, a DAT_CONNECTION_EVENT_UNREACHABLE event is generated on the *connect_evd_handle* of the Endpoint. The Endpoint transitions into a DAT_EP_STATE_DISCONNECTED state.

If the Provider can locally determine that the *remote_ia_address* is invalid, or that the remote_ia_address cannot be converted to a Transport-specific address, the operation can fail synchronously with a DAT_INVALID_ADDRESS return.

The local Endpoint is automatically transitioned into a DAT_EP_STATE_CONNECTED state when a Connection Request accepted by the remote Consumer and the Provider completes the Transport-specific Connection establishment. The local Consumer is notified of the established connection through a DAT_CONNECTION_EVENT_ESTABLISHED event on the *connect_evd_handle* of the local Endpoint.

When the *timeout* expired prior to completion of the Connection establishment, the local Endpoint is automatically transitioned into a DAT_EP_STATE_DISCONNECTED state and the local Consumer through a DAT_CONNECTION_EVENT_TIMED_OUT event on the *connect_evd_handle* of the local Endpoint.

Return Values

DAT_SUCCESS	The operation was successful.
DAT_INSUFFICIENT_RESOURCES	The operation failed due to resource limitations.
DAT_INVALID_PARAMETER	Invalid parameter.
DAT_INVALID_ADDRESS	Invalid address.
DAT_INVALID_HANDLE	Invalid DAT handle; Invalid Endpoint handle.
DAT_INVALID_STATE	Parameter in an invalid state. Endpoint was not in DAT_EP_STATE_UNCONNECTED state.
DAT_MODEL_NOT_SUPPORTED	The requested Model was not supported by the Provider. For example, the requested qos was not supported by the local Provider.

Usage

It is up to the Consumer to negotiate outstanding RDMA Read incoming and outgoing with a remote peer. The outstanding RDMA Read outgoing attribute should be smaller than the remote Endpoint outstanding RDMA Read incoming attribute. If this is not the case, Connection establishment might fail.

DAT API does not define a protocol on how remote peers exchange Endpoint attributes. The exchange of outstanding RDMA Read incoming and outgoing attributes of EPs is left to the Consumer ULP. The Consumer can use Private Data for it.

If the Consumer does not care about posting RDMA Read operations or remote RDMA Read operations on the connection, it can set the two outstanding RDMA Read attribute values to 0.

If the Consumer does not set the two outstanding RDMA Read attributes of the Endpoint, the Provider is free to pick up any value for default. The Provider is allowed to change these default values during connection setup.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Unsafe

See Also libdat(3LIB), attributes(5)

Name dat_ep_create – create an instance of an Endpoint

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ep_create (
    IN    DAT_IA_HANDLE    ia_handle,
    IN    DAT_PZ_HANDLE    pz_handle,
    IN    DAT_EVD_HANDLE   recv_evd_handle,
    IN    DAT_EVD_HANDLE   request_evd_handle,
    IN    DAT_EVD_HANDLE   connect_evd_handle,
    IN    DAT_EP_ATTR      *ep_attributes,
    OUT   DAT_EP_HANDLE    *ep_handle
    )
```

Parameters

ia_handle	Handle for an open instance of the IA to which the created Endpoint belongs.
pz_handle	Handle for an instance of the Protection Zone.
recv_evd_handle	Handle for the Event Dispatcher where events for completions of incoming (receive) DTOs are reported. `DAT_HANDLE_NULL` specifies that the Consumer is not interested in events for completions of receives.
request_evd_handle	Handle for the Event Dispatcher where events for completions of outgoing (Send, RDMA Write, RDMA Read, and RMR Bind) DTOs are reported. `DAT_HANDLE_NULL` specifies that the Consumer is not interested in events for completions of requests.
connect_evd_handle	Handle for the Event Dispatcher where Connection events are reported. `DAT_HANDLE_NULL` specifies that the Consumer is not interested in connection events for now.
ep_attributes	Pointer to a structure that contains Consumer-requested Endpoint attributes. Can be `NULL`.
ep_handle	Handle for the created instance of an Endpoint.

Description The `dat_ep_create()` function creates an instance of an Endpoint that is provided to the Consumer as *ep_handle*. The value of *ep_handle* is not defined if the `DAT_RETURN` is not `DAT_SUCCESS`.

The Endpoint is created in the Unconnected state.

Protection Zone *pz_handle* allows Consumers to control what local memory the Endpoint can access for DTOs and what memory remote RDMA operations can access over the connection of a created Endpoint. Only memory referred to by LMRs and RMRs that match the Endpoint Protection Zone can be accessed by the Endpoint.

The *recv_evd_handle* and *request_evd_handle* parameters are Event Dispatcher instances where the Consumer collects completion notifications of DTOs. Completions of Receive DTOs are reported in *recv_evd_handle* Event Dispatcher, and completions of Send, RDMA Read, and RDMA Write DTOs are reported in *request_evd_handle* Event Dispatcher. All completion notifications of RMR bindings are reported to a Consumer in *request_evd_handle* Event Dispatcher.

All Connection events for the connected Endpoint are reported to the Consumer through *connect_evd_handle* Event Dispatcher.

The *ep_attributes* parameter specifies the initial attributes of the created Endpoint. If the Consumer specifies `NULL`, the Provider fills it with its default Endpoint attributes. The Consumer might not be able to do any posts to the Endpoint or use the Endpoint in connection establishment until certain Endpoint attributes are set. Maximum Message Size and Maximum Recv DTOs are examples of such attributes.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INSUFFICIENT_RESOURCES`	The operation failed due to resource limitations.
`DAT_INVALID_HANDLE`	Invalid DAT handle.
`DAT_INVALID_PARAMETER`	Invalid parameter. One of the requested EP parameters or attributes was invalid or a combination of attributes or parameters is invalid.
`DAT_MODEL_NOT_SUPPORTED`	The requested Provider Model was not supported.

Usage

The Consumer creates an Endpoint prior to the establishment of a connection. The created Endpoint is in `DAT_EP_STATE_UNCONNECTED`. Consumers can do the following:

1. Request a connection on the Endpoint through `dat_ep_connect(3DAT)` or `dat_ep_dup_connect(3DAT)` for the active side of the connection model.
2. Associate the Endpoint with the Pending Connection Request that does not have an associated local Endpoint for accepting the Pending Connection Request for the passive/server side of the connection model.
3. Create a Reserved Service Point with the Endpoint for the passive/server side of the connection model. Upon arrival of a Connection Request on the Service Point, the Consumer accepts the Pending Connection Request that has the Endpoint associated with it

The Consumer cannot specify a *request_evd_handle* (*recv_evd_handle*) with Request Completion Flags (Recv Completion Flags) that do not match the other Endpoint Completion Flags for the DTO/RMR completion streams that use the same EVD. If *request_evd_handle* (*recv_evd_handle*) is used for an EVD that is fed by any event stream other than DTO or RMR completion event streams, only `DAT_COMPLETION_THRESHOLD` is valid for Request/Recv Completion Flags for the Endpoint completion streams that use that EVD. If

request_evd_handle (*recv_evd_handle*) is used for request (recv) completions of an Endpoint whose associated Request (Recv) Completion Flag attribute is `DAT_COMPLETION_UNSIGNALLED_FLAG`, the Request Completion Flags and Recv Completion Flags for all Endpoint completion streams that use the EVD must specify the same. Analogously, if *recv_evd_handle* is used for recv completions of an Endpoint whose associated Recv Completion Flags attribute is `DAT_COMPLETION_SOLICITED_WAIT`, the Recv Completion Flags for all Endpoint Recv completion streams that use the same EVD must specify the same Recv Completion Flags attribute value and the EVD cannot be used for any other event stream types.

If EP is created with `NULL` attributes, Provider can fill them with its own default values. The Consumer should not rely on the Provider-filled attribute defaults, especially for portable applications. The Consumer cannot do any operations on the created Endpoint except for `dat_ep_query`(3DAT), `dat_ep_get_status`(3DAT), `dat_ep_modify`(3DAT), and `dat_ep_free`(3DAT), depending on the values that the Provider picks.

The Provider is encouraged to pick up reasonable defaults because unreasonable values might restrict Consumers to the `dat_ep_query()`, `dat_ep_get_status()`, `dat_ep_modify()`, and `dat_ep_free()` operations. The Consumer should check what values the Provider picked up for the attributes. It is especially important to make sure that the number of posted operations is not too large to avoid EVD overflow. Depending on the values picked up by the Provider, the Consumer might not be able to do any RDMA operations; it might only be able to send or receive messages of very small sizes, or it might not be able to have more than one segment in a buffer. Before doing any operations, except the ones listed above, the Consumer can configure the Endpoint using `dat_ep_modify()` to the attributes they want.

One reason the Consumer might still want to create an Endpoint with Null attributes is for the Passive side of the connection establishment, where the Consumer sets up Endpoint attributes based on the connection request of the remote side.

Consumers might want to create Endpoints with `NULL` attributes if Endpoint properties are negotiated as part the Consumer connection establishment protocol.

Consumers that create Endpoints with Provider default attributes should always verify that the Provider default attributes meet their application's requirements with regard to the number of request/receive DTOs that can be posted, maximum message sizes, maximum request/receive IOV sizes, and maximum RDMA sizes.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Safe

See Also dat_ep_connect(3DAT), dat_ep_dup_connect(3DAT), dat_ep_free(3DAT), dat_ep_get_status(3DAT), dat_ep_modify(3DAT), dat_ep_query(3DAT), libdat(3LIB), attributes(5)

Name dat_ep_create_with_srq – create an instance of End Point with Shared Receive Queue

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ep_create_with_srq (
        IN      DAT_IA_HANDLE       ia_handle,
        IN      DAT_PZ_HANDLE       pz_handle,
        IN      DAT_EVD_HANDLE      recv_evd_handle,
        IN      DAT_EVD_HANDLE      request_evd_handle,
        IN      DAT_EVD_HANDLE      connect_evd_handle,
        IN      DAT_SRQ_HANDLE      srq_handle,
        IN      DAT_EP_ATTR         *ep_attributes,
        OUT     DAT_EP_HANDLE       *ep_handle
    )
```

Parameters

ia_handle Handle for an open instance of the IA to which the created Endpoint belongs.

pz_handle Handle for an instance of the Protection Zone.

recv_evd_handle Handle for the Event Dispatcher where events for completions of incoming (receive) DTOs are reported. `DAT_HANDLE_NULL` specifies that the Consumer is not interested in events for completions of receives.

request_evd_handle Handle for the Event Dispatcher where events for completions of outgoing (Send, RDMA Write, RDMA Read, and RMR Bind) DTOs are reported. `DAT_HANDLE_NULL` specifies that the Consumer is not interested in events for completions of requests.

connect_evd_handle Handle for the Event Dispatcher where Connection events are reported. `DAT_HANDLE_NULL` specifies that the Consumer is not interested in connection events for now.

srq_handle Handle for an instance of the Shared Receive Queue.

ep_attributes Pointer to a structure that contains Consumer-requested Endpoint attributes. Cannot be `NULL`.

ep_handle Handle for the created instance of an Endpoint.

Description The `dat_ep_create_with_srq()` function creates an instance of an Endpoint that is using SRQ for Recv buffers is provided to the Consumer as *ep_handle*. The value of *ep_handle* is not defined if the `DAT_RETURN` is not `DAT_SUCCESS`.

The Endpoint is created in the Unconnected state.

Protection Zone *pz_handle* allows Consumers to control what local memory the Endpoint can access for DTOs except Recv and what memory remote RDMA operations can access over the connection of a created Endpoint. Only memory referred to by LMRs and RMRs that match the Endpoint Protection Zone can be accessed by the Endpoint. The Recv DTO buffers PZ must match the SRQ PZ. The SRQ PZ might or might not be the same as the EP one. Check Provider attribute for the support of different PZs between SRQ and its EPs.

The *recv_evd_handle* and *request_evd_handle* arguments are Event Dispatcher instances where the Consumer collects completion notifications of DTOs. Completions of Receive DTOs are reported in *recv_evd_handle* Event Dispatcher, and completions of Send, RDMA Read, and RDMA Write DTOs are reported in *request_evd_handle* Event Dispatcher. All completion notifications of RMR bindings are reported to a Consumer in *request_evd_handle* Event Dispatcher.

All Connection events for the connected Endpoint are reported to the Consumer through *connect_evd_handle* Event Dispatcher.

Shared Receive Queue *srq_handle* specifies where the EP will dequeue Recv DTO buffers.

The created EP can be reset. The relationship between SRQ and EP is not effected by `dat_ep_reset(3DAT)`.

SRQ can not be disassociated or replaced from created EP. The only way to disassociate SRQ from EP is to destroy EP.

Receive buffers cannot be posted to the created Endpoint. Receive buffers must be posted to the SRQ to be used for the created Endpoint.

The ep_attributes parameter specifies the initial attributes of the created Endpoint. Consumer can not specify `NULL` for *ep_attributes* but can specify values only for the parameters needed and default for the rest.

For *max_request_dtos* and *max_request_iov*, the created Endpoint will have at least the Consumer requested values but might have larger values. Consumer can query the created Endpoint to find out the actual values for these attributes. Created Endpoint has the exact Consumer requested values for *max_recv_dtos*, *max_message_size*, *max_rdma_size*, *max_ rdma_read_in*, and *max_rdma_read_out*. For all other attributes, except *max_recv_iov* that is ignored, the created Endpoint has the exact values requested by Consumer. If Provider cannot satisfy the Consumer requested attribute values the operation fails.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INSUFFICIENT_RESOURCES`	The operation failed due to resource limitations.
`DAT_INVALID_HANDLE`	Invalid DAT handle.

DAT_INVALID_PARAMETER	Invalid parameter. One of the requested EP parameters or attributes was invalid or a combination of attributes or parameters is invalid. For example, *pz_handle* specified does not match the one for SRQ or the requested maximum RDMA Read IOV exceeds IA capabilities..
DAT_MODEL_NOT_SUPPORTED	The requested Provider Model was not supported.

Usage The Consumer creates an Endpoint prior to the establishment of a connection. The created Endpoint is in DAT_EP_STATE_UNCONNECTED. Consumers can do the following:

1. Request a connection on the Endpoint through dat_ep_connect(3DAT) or dat_ep_dup_connect(3DAT) for the active side of the connection model.
2. Associate the Endpoint with the Pending Connection Request that does not have an associated local Endpoint for accepting the Pending Connection Request for the passive/server side of the con-nection model.
3. Create a Reserved Service Point with the Endpoint for the passive/server side of the connection model. Upon arrival of a Connection Request on the Service Point, the Consumer accepts the Pending Connection Request that has the Endpoint associated with it.

The Consumer cannot specify a *request_evd_handle* (*recv_evd_handle*) with Request Completion Flags (Recv Completion Flags) that do not match the other Endpoint Completion Flags for the DTO/RMR completion streams that use the same EVD. If *request_evd_handle* (*recv_evd_ handle*) is used for request (recv) completions of an Endpoint whose associated Request (Recv) Completion Flag attribute is DAT_COMPLETION_UNSIGNALLED_FLAG, the Request Completion Flags and Recv Completion Flags for all Endpoint completion streams that use the EVD must specify the same. By definition, completions of all Recv DTO posted to SRQ complete with Signal. Analogously, if *recv_evd_handle* is used for recv completions of an Endpoint whose associated Recv Completion Flag attribute is DAT_COMPLETION_SOLICITED_WAIT, the Recv Completion Flags for all Endpoint Recv completion streams that use the same EVD must specify the same Recv Completion Flags attribute value and the EVD cannot be used for any other event stream types. If *recv_evd_handle* is used for Recv completions of an Endpoint that uses SRQ and whose Recv Completion Flag attribute is DAT_COMPLETION_EVD_THRESHOLD then all Endpoint DTO completion streams (request and/or recv completion streams) that use that *recv_evd_handle* must specify DAT_COMPLETION_EVD_THRESHOLD. Other event stream types can also use the same EVD.

Consumers might want to use DAT_COMPLETION_UNSIGNALLED_FLAG for Request and/or Recv completions when they control locally with posted DTO/RMR completion flag (not needed for Recv posted to SRQ) whether posted DTO/RMR completes with Signal or not. Consumers might want to use DAT_COMPLETION_SOLICITED_WAIT for Recv completions when the remote sender side control whether posted Recv competes with Signal or not or not. uDAPL

Consumers might want to use DAT_COMPLETION_EVD_THRESHOLD for Request and/or Recv completions when they control waiter unblocking with the *threshold* parameter of the dat_evd_wait(3DAT).

Some Providers might restrict whether multiple EPs that share a SRQ can have different Protection Zones. Check the *srq_ep_pz_difference_support* Provider attribute for it.

Consumers might want to have a different PZ between EP and SRQ. This allows incoming RDMA operations to be specific to this EP PZ and not the same for all EPs that share SRQ. This is critical for servers that supports multiple independent clients.

The Provider is strongly encouraged to create an EP that is ready to be connected. Any effects of previous connections or connection establishment attempts on the underlying Transport-specific Endpoint to which the DAT Endpoint is mapped to should be hidden from the Consumer. The methods described below are examples:

- The Provider does not create an underlying Transport Endpoint until the Consumer is connecting the Endpoint or accepting a connection request on it. This allows the Provider to accumulate Consumer requests for attribute settings even for attributes that the underlying transport does not allow to change after the Transport Endpoint is created.
- The Provider creates the underlying Transport Endpoint or chooses one from a pool of Provider-controlled Transport Endpoints when the Consumer creates the Endpoint. The Provider chooses the Transport Endpoint that is free from any underlying internal attributes that might prevent the Endpoint from being connected. For IB and IP, that means that the Endpoint is not in the TimeWait state. Changing of some of the Endpoint attributes becomes hard and might potentially require mapping the Endpoint to another underlying Transport Endpoint that might not be feasible for all transports.
- The Provider allocates a Transport-specific Endpoint without worrying about impact on it from previous connections or connection establishment attempts. Hide the Transport-specific TimeWait state or CM timeout of the underlying transport Endpoint within dat_ep_connect(3DAT), dat_ep_dup_connect(3DAT), or dat_cr_accept(3DAT). On the Active side of the connection establishment, if the remnants of a previous connection for Transport-specific Endpoint can be hidden within the Timeout parameter, do so. If not, generating DAT_CONNECTION_EVENT_NON_PEER_REJECTED is an option. For the Passive side, generating a DAT_CONNECTION_COMPLETION_ERROR event locally, while sending a non-peer-reject message to the active side, is a way of handling it.

Any transitions of an Endpoint into an Unconnected state can be handled similarly. One transition from a Disconnected to an Unconnected state is a special case.

For dat_ep_reset(3DAT), the Provider can hide any remnants of the previous connection or failed connection establishment in the operation itself. Because the operation is synchronous, the Provider can block in it until the TimeWait state effect of the previous connection or connection setup is expired, or until the Connection Manager timeout of an unsuccessful

connection establishment attempt is expired. Alternatively, the Provider can create a new Endpoint for the Consumer that uses the same handle.

DAT Providers are required not to change any Consumer-specified Endpoint attributes during connection establishment. If the Consumer does not specify an attribute, the Provider can set it to its own default. Some EP attributes, like outstanding RDMA Read incoming or outgoing, if not set up by the Consumer, can be changed by Providers to establish connection. It is recommended that the Provider pick the default for outstanding RDMA Read attributes as 0 if the Consumer has not specified them. This ensures that connection establishment does not fail due to insufficient outstanding RDMA Read resources, which is a requirement for the Provider.

The Provider is not required to check for a mismatch between the maximum RDMA Read IOV and maximum RDMA Read outgoing attributes, but is allowed to do so. In the later case it is allowed to return `DAT_INVALID_ PARAMETER` when a mismatch is detected. Provider must allocate resources to satisfy the combination of these two EP attributes for local RDMA Read DTOs.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.2
MT-Level	Safe

See Also `dat_ep_create(3DAT)`, `dat_srq_create(3DAT)`, `dat_srq_free(3DAT)`, `dat_srq_query(3DAT)`, `libdat(3LIB)`, `attributes(5)`

Name dat_ep_disconnect – terminate a connection or a connection establishment

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ep_disconnect (
    IN    DAT_EP_HANDLE      ep_handle,
    IN    DAT_CLOSE_FLAGS    disconnect_flags
    )
```

Parameters

ep_handle Handle for an instance of Endpoint.

disconnect_flags Flags for disconnect. Flag values are as follows:

`DAT_CLOSE_ABRUPT_FLAG`	Abrupt close. This is the default value.
`DAT_CLOSE_GRACEFUL_FLAG`	Graceful close.

Description The `dat_ep_disconnect()` function requests a termination of a connection or connection establishment. This operation is used by the active/client or a passive/server side Consumer of the connection model.

The *disconnect_flags* parameter allows Consumers to specify whether they want graceful or abrupt disconnect. Upon disconnect, all outstanding and in-progress DTOs and RMR Binds must be completed.

For abrupt disconnect, all outstanding DTOs and RMR Binds are completed unsuccessfully, and in-progress DTOs and RMR Binds can be completed successfully or unsuccessfully. If an in-progress DTO is completed unsuccessfully, all follow on in-progress DTOs in the same direction also must be completed unsuccessfully. This order is presented to the Consumer through a DTO completion Event Stream of the *recv_evd_handle* and and *request_evd_handle* of the Endpoint.

For graceful disconnect, all outstanding and in-progress request DTOs and RMR Binds must try to be completed successfully first, before disconnect proceeds. During that time, the local Endpoint is in a `DAT_EP_DISCONNECT_PENDING` state.

The Consumer can call abrupt `dat_ep_disconnect()` when the local Endpoint is in the `DAT_EP_DISCONNECT_PENDING` state. This causes the Endpoint to transition into `DAT_EP_STATE_DISCONNECTED` without waiting for outstanding and in-progress request DTOs and RMR Binds to successfully complete. The graceful `dat_ep_disconnect()` call when the local Endpoint is in the `DAT_EP_DISCONNECT_PENDING` state has no effect.

If the Endpoint is not in `DAT_EP_STATE_CONNECTED`, the semantic of the operation is the same for graceful or abrupt *disconnect_flags* value.

No new Send, RDMA Read, and RDMA Write DTOs, or RMR Binds can be posted to the Endpoint when the local Endpoint is in the `DAT_EP_DISCONNECT_PENDING` state.

The successful completion of the disconnect is reported to the Consumer through a DAT_CONNECTION_EVENT_DISCONNECTED event on *connect_evd_handle* of the Endpoint. The Endpoint is automatically transitioned into a DAT_EP_STATE_DISCONNECTED state upon successful asynchronous completion. If the same EVD is used for *connect_evd_handle* and any *recv_evd_handle* and request_evd_handle, all successful Completion events of in-progress DTOs precede the Disconnect Completion event.

Disconnecting an unconnected Disconnected Endpoint is no-op. Disconnecting an Endpoint in DAT_EP_STATE_UNCONNECTED, DAT_EP_STATE_RESERVED, DAT_EP_STATE_PASSIVE_CONNECTION_PENDING, and DAT_EP_STATE_TENTATIVE_CONNECTION_PENDING is disallowed.

Both abrupt and graceful disconnect of the Endpoint during connection establishment, DAT_EP_STATE_ACTIVE_CONNECTION_PENDING and DAT_EP_STATE_COMPLETION_PENDING, "aborts" the connection establishment and transitions the local Endpoint into DAT_EP_STATE_DISCONNECTED. That causes preposted Recv DTOs to be flushed to *recv_evd_handle*.

Return Values

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	The *ep_handle* parameter is invalid.
DAT_INSUFFICIENT_RESOURCES	The operation failed due to resource limitations.
DAT_INVALID_PARAMETER	The *disconnect_flags* parameter is invalid.
DAT_INVALID_STATE	A parameter is in an invalid state. Endpoint is not in the valid state for disconnect.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Unsafe

See Also

libdat(3LIB), attributes(5)

Name dat_ep_dup_connect – establish a connection between the local Endpoint and a remote Endpoint

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ep_dup_connect (
    IN    DAT_EP_HANDLE    ep_handle,
    IN    DAT_EP_HANDLE    dup_ep_handle,
    IN    DAT_TIMEOUT      timeout,
    IN    DAT_COUNT        private_data_size,
    IN const  DAT_PVOID    private_data,
    IN    DAT_QOS          qos
    )
```

Parameters

ep_handle	Handle for an instance of an Endpoint.
dup_ep_handle	Connected local Endpoint that specifies a requested connection remote end.
timeout:	Duration of time, in microseconds, that Consumers wait for Connection establishment. The value of `DAT_TIMEOUT_INFINITE` represents no timeout, indefinite wait. Values must be positive.
private_data_size	Size of *private_data*. Must be nonnegative.
private_data	Pointer to the private data that should be provided to the remote Consumer as part of the Connection Request. If *private_data_size* is zero, then *private_data* can be `NULL`.
qos	Requested Quality of Service of the connection.

Description The `dat_ep_dup_connect()` function requests that a connection be established between the local Endpoint and a remote Endpoint. This operation is used by the active/client side Consumer of the connection model. The remote Endpoint is identified by the *dup_ep_handle*. The remote end of the requested connection shall be the same as the remote end of the *dup_ep_handle*. This is equivalent to requesting a connection to the same remote IA, Connection Qualifier, and *connect_flags* as used for establishing the connection on duplicated Endpoints and following the same redirections.

Upon establishing the requested connection as part of the successful completion of this operation, the local Endpoint is bound to a Port Qualifier of the local IA. The Port Qualifier is passed to the remote side of the requested connection and is available to the remote Consumer in the Connection Request of the `DAT_CONNECTION_REQUEST_EVENT`.

The Consumer-provided *private_data* is passed to the remote side and is provided to the remote Consumer in the Connection Request. Consumers can encapsulate any local Endpoint attributes that remote Consumers need to know as part of an upper-level protocol. Providers

can also provide a Provider on the remote side any local Endpoint attributes and Transport-specific information needed for Connection establishment by the Transport.

Upon successful completion of this operation, the local Endpoint is transferred into `DAT_EP_STATE_ACTIVE_CONNECTION_PENDING`.

Consumers can request a specific value of *qos*. The Provider specifies which Quality of Service it supports in documentation and in the Provider attributes. If the local Provider or Transport does not support the requested *qos*, the operation fails and `DAT_MODEL_NOT_SUPPORTED` is returned synchronously. If the remote Provider does not support the requested *qos*, the local Endpoint is automatically transitioned into a `DAT_EP_STATE_UNDISCONNECTED` state, the connection is not established, and the event returned on the *connect_evd_handle* is `DAT_CONNECTION_EVENT_NON_PEER_REJECTED`. The same `DAT_CONNECTION_EVENT_NON_PEER_REJECTED` event is returned if connection cannot be established for all reasons for not establishing the connection, except timeout, remote host not reachable, and remote peer reject. For example, remote host is not reachable, remote Consumer is not listening on the requested Connection Qualifier, Backlog of the requested Service Point is full, and Transport errors. In this case, the local Endpoint is automatically transitioned into a `DAT_EP_STATE_UNDISCONNECTED` state.

The acceptance of the requested connection by the remote Consumer is reported to the local Consumer through a `DAT_CONNECTION_EVENT_ESTABLISHED` event on the *connect_evd_handle* of the local Endpoint.

The rejection of the connection by the remote Consumer is reported to the local Consumer through a `DAT_CONNECTION_EVENT_PEER_REJECTED` event on the *connect_evd_handle* of the local Endpoint and the local Endpoint is automatically transitioned into a `DAT_EP_STATE_UNDISCONNECTED` state.

When the Provider cannot reach the remote host or the remote host does not respond within the Consumer-requested *timeout*, a `DAT_CONNECTION_EVENT_UNREACHABLE` is generated on the *connect_evd_handle* of the Endpoint. The Endpoint transitions into a `DAT_EP_STATE_DISCONNECTED` state.

The local Endpoint is automatically transitioned into a `DAT_EP_STATE_CONNECTED` state when a Connection Request is accepted by the remote Consumer and the Provider completes the Transport-specific Connection establishment. The local Consumer is notified of the established connection through a `DAT_CONNECTION_EVENT_ESTABLISHED` event on the *connect_evd_handle* of the local Endpoint.

When the *timeout* expired prior to completion of the Connection establishment, the local Endpoint is automatically transitioned into a `DAT_EP_STATE_UNDISCONNECTED` state and the local Consumer through a `DAT_CONNECTION_EVENT_TIMED_OUT` event on the *connect_evd_handle* of the local Endpoint.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INSUFFICIENT_RESOURCES`	The operation failed due to resource limitations.
`DAT_INVALID_PARAMETER`	Invalid parameter.
`DAT_INVALID_HANDLE`	The *ep_handle* or *dup_ep_handle* parameter is invalid.
`DAT_INVALID_STATE`	A parameter is in an invalid state.
`DAT_MODEL_NOT_SUPPORTED`	The requested Model is not supported by the Provider. For example, requested *qos* was not supported by the local Provider.

Usage

It is up to the Consumer to negotiate outstanding RDMA Read incoming and outgoing with a remote peer. The outstanding RDMA Read outgoing attribute should be smaller than the remote Endpoint outstanding RDMA Read incoming attribute. If this is not the case, connection establishment might fail.

DAT API does not define a protocol on how remote peers exchange Endpoint attributes. The exchange of outstanding RDMA Read incoming and outgoing attributes of EPs is left to the Consumer ULP. The Consumer can use Private Data for it.

If the Consumer does not care about posting RDMA Read operations or remote RDMA Read operations on the connection, it can set the two outstanding RDMA Read attribute values to 0.

If the Consumer does not set the two outstanding RDMA Read attributes of the Endpoint, the Provider is free to pick up any values as a default. The Provider is allowed to change these default values during connection setup.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Unsafe

See Also

`libdat(3LIB)`, `attributes(5)`

Name dat_ep_free – destroy an instance of the Endpoint

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ep_free (
    IN    DAT_EP_HANDLE    ep_handle
    )
```

Parameters *ep_handle* Handle for an instance of the Endpoint.

Description The `dat_ep_free()` function destroys an instance of the Endpoint.

The Endpoint can be destroyed in any Endpoint state except Reserved, Passive Connection Pending, and Tentative Connection Pending. The destruction of the Endpoint can also cause the destruction of DTOs and RMRs posted to the Endpoint and not dequeued yet. This includes completions for all outstanding and in-progress DTOs/RMRs. The Consumer must be ready for all completions that are not dequeued yet either still being on the Endpoint *recv_evd_handle* and *request_evd_handle* or not being there.

The destruction of the Endpoint during connection setup aborts connection establishment.

If the Endpoint is in the Reserved state, the Consumer shall first destroy the associated Reserved Service Point that transitions the Endpoint into the Unconnected state where the Endpoint can be destroyed. If the Endpoint is in the Passive Connection Pending state, the Consumer shall first reject the associated Connection Request that transitions the Endpoint into the Unconnected state where the Endpoint can be destroyed. If the Endpoint is in the Tentative Connection Pending state, the Consumer shall reject the associated Connection Request that transitions the Endpoint back to Provider control, and the Endpoint is destroyed as far as the Consumer is concerned.

The freeing of an Endpoint also destroys an Event Stream for each of the associated Event Dispatchers.

Use of the handle of the destroyed Endpoint in any subsequent operation except for the `dat_ep_free()` fails.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INVALID_HANDLE`	The *ep_handle* parameter is invalid.
`DAT_INVALID_STATE`	Parameter in an invalid state. The Endpoint is in `DAT_EP_STATE_RESERVED`, `DAT_EP_STATE_PASSIVE_CONNECTION_PENDING`, or `DAT_EP_STATE_TENTATIVE_CONNECTION_PENDING`.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Unsafe

See Also libdat(3LIB), attributes(5)

Name dat_ep_get_status – provide a quick snapshot of the Endpoint

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ep_get_status (
    IN    DAT_EP_HANDLE    ep_handle,
    OUT   DAT_EP_STATE     *ep_state,
    OUT   DAT_BOOLEAN      *recv_idle,
    OUT   DAT_BOOLEAN      *request_idle
    )
```

Parameters

ep_handle Handle for an instance of the Endpoint.

ep_state Current state of the Endpoint.

recv_idle Status of the incoming DTOs on the Endpoint.

request_idle Status of the outgoing DTOs and RMR Bind operations on the Endpoint.

Description the `dat_ep_get_status()` function provides the Consumer a quick snapshot of the Endpoint. The snapshot consists of the Endpoint state and whether there are outstanding or in-progress, incoming or outgoing DTOs. Incoming DTOs consist of Receives. Outgoing DTOs consist of the Requests, Send, RDMA Read, RDMA Write, and RMR Bind.

The *ep_state* parameter returns the value of the current state of the Endpoint *ep_handle*. State value is one of the following: `DAT_EP_STATE_UNCONNECTED`, `DAT_EP_STATE_RESERVED`, `DAT_EP_STATE_PASSIVE_CONNECTION_PENDING`, `DAT_EP_STATE_ACTIVE_CONNECTION_PENDING`, `DAT_EP_STATE_TENTATIVE_CONNECTION_PENDING`, `DAT_EP_STATE_CONNECTED`, `DAT_EP_STATE_DISCONNECT_PENDING`, or `DAT_EP_STATE_DISCONNECTED`.

A *recv_idle* value of `DAT_TRUE` specifies that there are no outstanding or in-progress Receive DTOs at the Endpoint, and `DAT_FALSE` otherwise.

A *request_idle* value of `DAT_TRUE` specifies that there are no outstanding or in-progress Send, RDMA Read, and RDMA Write DTOs, and RMR Binds at the Endpoint, and `DAT_FALSE` otherwise.

This call provides a snapshot of the Endpoint status only. No heroic synchronization with DTO queuing or processing is implied.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *ep_handle* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Safe

See Also libdat(3LIB), attributes(5)

Name dat_ep_modify – change parameters of an Endpoint

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ep_modify (
    IN    DAT_EP_HANDLE        ep_handle,
    IN    DAT_EP_PARAM_MASK    ep_param_mask,
    IN    DAT_EP_PARAM         *ep_param
    )
```

Parameters

ep_handle	Handle for an instance of the Endpoint.
ep_param_mask	Mask for Endpoint parameters.
ep_param	Pointer to the Consumer-allocated structure that contains Consumer-requested Endpoint parameters.

Description The `dat_ep_modify()` function provides the Consumer a way to change parameters of an Endpoint.

The *ep_param_mask* parameter allows Consumers to specify which parameters to modify. Providers modify values for *ep_param_mask* requested parameters only.

Not all the parameters of the Endpoint can be modified. Some can be modified only when the Endpoint is in a specific state. The following list specifies which parameters can be modified and when they can be modified.

Interface Adapter
: Cannot be modified.

  Endpoint belongs to an open instance of IA and that association cannot be changed.

Endpoint State
: Cannot be modified.

  State of Endpoint cannot be changed by a `dat_ep_modify()` operation.

Local IA Address
: Cannot be modified.

  Local IA Address cannot be changed by a `dat_ep_modify()` operation.

Local Port Qualifier
: Cannot be modified.

  Local port qualifier cannot be changed by a `dat_ep_modify()` operation.

Remote IA Address
: Cannot be modified.

Remote IA Address cannot be changed by a `dat_ep_modify()` operation.

Remote Port Qualifier
Cannot be modified.

Remote port qualifier cannot be changed by a `dat_ep_modify()` operation

Protection Zone
Can be modified when in Quiescent, Unconnected, and Tentative Connection Pending states.

Protection Zone can be changed only when the Endpoint is in quiescent state. The only Endpoint states that isare quiescent isare `DAT_EP_STATE_UNCONNECTED` and `DAT_EP_STATE_TENTATIVE_CONNECTION_PENDING`. Consumers should be aware that any Receive DTOs currently posted to the Endpoint that do not match the new Protection Zone fail with a `DAT_PROTECTION_VIOLATION` return.

In DTO Event Dispatcher
Can be modified when in Unconnected, Reserved, Passive Connection Request Pending, and Tentative Connection Pending states.

Event Dispatcher for incoming DTOs (Receive) can be changed only prior to a request for a connection for an Active side or prior to accepting a Connection Request for a Passive side.

Out DTO Event Dispatcher
Can be modified when in Unconnected, Reserved, Passive Connection Pending, and Tentative Connection Pending states.

Event Dispatcher for outgoing DTOs (Send, RDMA Read, and RDMA Write) can be changed only prior to a request for a connection for an Active side or prior to accepting a Connection Request for a Passive side.

Connection Event Dispatcher
Can be modified when in Unconnected, Reserved, Passive Connection Pending, and Tentative Connection Pending states.

Event Dispatcher for the Endpoint Connection events can be changed only prior to a request for a connection for an Active side or accepting a Connection Request for a Passive side.

Service Type
Can be modified when in Unconnected, Reserved, Passive Connection Pending, and Tentative Connection Pending states.

Service Type can be changed only prior to a request for a connection for an Active side or accepting a Connection Request for a Passive side.

Maximum Message Size
Can be modified when in Unconnected, Reserved, Passive Connection Request Pending, and Tentative Connection Pending states.

Maximum Message Size can be changed only prior to a request for a connection for an Active side or accepting a Connection Request for a Passive side.

Maximum RDMA Size
Can be modified when in Unconnected, Reserved, Passive Connection Pending, and Tentative Connection Pending states.

Maximum RDMA Size can be changed only prior to a request for a connection for an Active side or accepting a Connection Request for a Passive side.

Quality of Service
Can be modified when in Unconnected, Reserved, Passive Connection Pending, and Tentative Connection Pending states.

QoS can be changed only prior to a request for a connection for an Active side or accepting a Connection Request for a Passive side.

Recv Completion Flags
Can be modified when in Unconnected, Reserved, Passive Connection Pending, and Tentative Connection Pending states.

Recv Completion Flags specifies what DTO flags the Endpoint should support for Receive DTO operations. The value can be `DAT_COMPLETION_NOTIFICATION_SUPPRESS_FLAG`, `DAT_COMPLETION_SOLICITED_WAIT_FLAG`, or `DAT_COMPLETION_EVD_THRESHOLD_FLAG`. Recv posting does not support `DAT_COMPLETION_SUPPRESS_FLAG` or `DAT_COMPLETION_BARRIER_FENCE_FLAG` dat_completion_flags values that are only applicable to Request postings. Recv Completion Flags can be changed only prior to a request for a connection for an Active side or accepting a Connection Request for a Passive side, but before posting of any Recvs.

Request Completion Flags
Can be modified when in Unconnected, Reserved, Passive Connection Request Pending, and Tentative Connection Pending states.

Request Completion Flags specifies what DTO flags the Endpoint should support for Send, RDMA Read, RDMA Write, and RMR Bind operations. The value can be: `DAT_COMPLETION_UNSIGNALLED_FLAG` or `DAT_COMPLETION_EVD_THRESHOLD_FLAG`. Request postings always support `DAT_COMPLETION_SUPPRESS_FLAG`, `DAT_COMPLETION_SOLICITED_WAIT_FLAG`, or `DAT_COMPLETION_BARRIER_FENCE_FLAG` completion_flags values. Request Completion Flags can be changed only prior to a request for a connection for an Active side or accepting a Connection Request for a Passive side.

Maximum Recv DTO
Can be modified when in Unconnected, Reserved, Passive Connection Pending, and Tentative Connection Pending states.

Maximum Recv DTO specifies the maximum number of outstanding Consumer-submitted Receive DTOs that a Consumer expects at any time at the Endpoint.

Maximum Recv DTO can be changed only prior to a request for a connection for an Active side or accepting a Connection Request for a Passive side.

Maximum Request DTO
Can be modified when in Unconnected, Reserved, Passive Connection Pending, and Tentative Connection Pending states.

Maximum Request DTO specifies the maximum number of outstanding Consumer-submitted send and RDMA DTOs and RMR Binds that a Consumer expects at any time at the Endpoint. Maximum Out DTO can be changed only prior to a request for a connection for an Active side or accepting a Connection Request for a Passive side.

Maximum Recv IOV
Can be modified when in Unconnected, Reserved, Passive Connection Pending, and Tentative Connection Pending states.

Maximum Recv IOV specifies the maximum number of elements in IOV that a Consumer specifies for posting a Receive DTO for the Endpoint. Maximum Recv IOV can be changed only prior to a request for a connection for an Active side or accepting a Connection Request for a Passive side.

Maximum Request IOV
Can be modified when in Unconnected, Reserved, Passive Connection Pending, and Tentative Connection Pending states.

Maximum Request IOV specifies the maximum number of elements in IOV that a Consumer specifies for posting a Send, RDMA Read, or RDMA Write DTO for the Endpoint. Maximum Request IOV can be changed only prior to a request for a connection for an Active side or accepting a Connection Request for a Passive side.

Maximum outstanding RDMA Read as target
Can be modified when in Unconnected, Reserved, Passive Connection Pending, and Tentative Connection Pending states.

Maximum number of outstanding RDMA Reads for which the Endpoint is the target.

Maximum outstanding RDMA Read as originator
Can be modified when in Unconnected, Reserved, Passive Connection Pending, and Tentative Connection Pending states.

Maximum number of outstanding RDMA Reads for which the Endpoint is the originator.

Num transport-specific attributes
Can be modified when in Quiescent (unconnected) state.

Number of transport-specific attributes to be modified.

Transport-specific endpoint attributes
Can be modified when in Quiescent (unconnected) state.

Transport-specific attributes can be modified only in the transport-defined Endpoint state. The only guaranteed safe state in which to modify transport-specific Endpoint attributes is the quiescent state `DAT_EP_STATE_UNCONNECTED`.

Num provider-specific attributes
Can be modified when in Quiescent (unconnected) state.

Number of Provider-specific attributes to be modified.

Provider-specific endpoint attributes
Can be modified when in Quiescent (unconnected) state.

Provider-specific attributes can be modified only in the Provider-defined Endpoint state. The only guaranteed safe state in which to modify Provider-specific Endpoint attributes is the quiescent state `DAT_EP_STATE_UNCONNECTED`.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INVALID_HANDLE`	The *ep_handle* parameter is invalid.
`DAT_INVALID_PARAMETER`	The *ep_param_mask* parameter is invalid, or one of the requested Endpoint parameters or attributes was invalid, not supported, or cannot be modified.
`DAT_INVALID_STATE`	Parameter in an invalid state. The Endpoint was not in the state that allows one of the parameters or attributes to be modified.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Unsafe

See Also

`libdat(3LIB)`, `attributes(5)`

Name dat_ep_post_rdma_read – transfer all data to the local data buffer

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ep_post_rdma_read (
    IN    DAT_EP_HANDLE        ep_handle,
    IN    DAT_COUNT            num_segments,
    IN    DAT_LMR_TRIPLET      *local_iov,
    IN    DAT_DTO_COOKIE       user_cookie,
    IN    DAT_RMR_TRIPLET      *remote_buffer,
    IN    DAT_COMPLETION_FLAGS completion_flags
    )
```

Parameters

ep_handle Handle for an instance of the Endpoint.

num_segments Number of *lmr_triplets* in *local_iov*.

local_iov I/O Vector that specifies the local buffer to fill.

user_cookie User-provided cookie that is returned to the Consumer at the completion of the RDMA Read. Can be `NULL`.

remote_buffer A pointer to an RMR Triplet that specifies the remote buffer from which the data is read.

completion_flags Flags for posted RDMA Read. The default `DAT_COMPLETION_DEFAULT_FLAG` is 0x00. Other values are as follows:

Completion Suppression	`DAT_COMPLETION_SUPPRESS_FLAG`	
	0x01	Suppress successful Completion.
Notification of Completion	`DAT_COMPLETION_UNSIGNALLED_FLAG`	
	0x04	Non-notification completion. Local Endpoint must be configured for Notification Suppression.
Barrier Fence	`DAT_COMPLETION_BARRIER_FENCE_FLAG`	
	0x08	Request for Barrier Fence.

Description The `dat_ep_post_rdma_read()` function requests the transfer of all the data specified by the *remote_buffer* over the connection of the *ep_handle* Endpoint into the *local_iov*.

The *num_segments* parameter specifies the number of segments in the *local_iov*. The *local_iov* segments are filled in the I/O Vector order until the whole message is received. This ensures

that all the "front" segments of the *local_iov* I/O Vector are completely filled, only one segment is partially filled, if needed, and all segments that follow it are not filled at all.

The *user_cookie* allows Consumers to have unique identifiers for each DTO. These identifiers are completely under user control and are opaque to the Provider. There is no requirement on the Consumer that the value *user_cookie* should be unique for each DTO. The *user_cookie* is returned to the Consumer in the Completion event for the posted RDMA Read.

A Consumer must not modify the *local_iov* or its content until the DTO is completed. When a Consumer does not adhere to this rule, the behavior of the Provider and the underlying Transport is not defined. Providers that allow Consumers to get ownership of the *local_iov* but not the memory it specifies back after the `dat_ep_post_rdma_read()` returns should document this behavior and also specify its support in Provider attributes. This behavior allows Consumers full control of the *local_iov* after `dat_ep_post_rdma_read()` returns. Because this behavior is not guaranteed by all Providers, portable Consumers should not rely on this behavior. Consumers should not rely on the Provider copying *local_iov* information.

The completion of the posted RDMA Read is reported to the Consumer asynchronously through a DTO Completion event based on the specified *completion_flags* value. The value of `DAT_COMPLETION_UNSIGNALLED_FLAG` is only valid if the Endpoint Request Completion Flags `DAT_COMPLETION_UNSIGNALLED_FLAG`. Otherwise, `DAT_INVALID_PARAMETER` is returned.

The `DAT_SUCCESS` return of the `dat_ep_post_rdma_read()` is at least the equivalent of posting an RDMA Read operation directly by native Transport. Providers should avoid resource allocation as part of `dat_ep_post_rdma_read()` to ensure that this operation is nonblocking and thread safe for an UpCall.

The operation is valid for the Endpoint in the `DAT_EP_STATE_CONNECTED` and `DAT_EP_STATE_DISCONNECTED` states. If the operation returns successfully for the Endpoint in the `DAT_EP_STATE_DISCONNECTED` state, the posted RDMA Read is immediately flushed to *request_evd_handle*.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INSUFFICIENT_RESOURCES`	The operation failed due to resource limitations.
`DAT_INVALID_PARAMETER`	Invalid parameter. For example, one of the IOV segments pointed to a memory outside its LMR.
`DAT_INVALID_HANDLE`	The *ep_handle* parameter is invalid.
`DAT_INVALID_STATE`	A parameter is in an invalid state. Endpoint was not in the `DAT_EP_STATE_CONNECTED` or `DAT_EP_STATE_DISCONNECTED` state.
`DAT_LENGTH_ERROR`	The size of the receiving buffer is too small for sending buffer data. The size of the local buffer is too small for the data of the remote buffer.

`DAT_PROTECTION_VIOLATION`	Protection violation for local or remote memory access. Protection Zone mismatch between either an LMR of one of the *local_iov* segments and the local Endpoint or the *rmr_context* and the remote Endpoint.
`DAT_PRIVILEGES_VIOLATION`	Privileges violation for local or remote memory access. Either one of the LMRs used in *local_iov* is invalid or does not have the local write privileges, or *rmr_context* does not have the remote read privileges.

Usage For best RDMA Read operation performance, the Consumer should align each buffer segment of *local_iov* to the Optimal Buffer Alignment attribute of the Provider. For portable applications, the Consumer should align each buffer segment of *local_iov* to the `DAT_OPTIMAL_ALIGNMENT`.

If connection was established without outstanding RDMA Read attributes matching on Endpoints on both sides (outstanding RDMA Read outgoing on one end is larger than the outstanding RDMA Read incoming on the other end), connection is broken when the number of incoming RDMA Read exceeds the outstanding RDMA Read incoming attribute of the Endpoint. The Consumer can use its own flow control to ensure that it does not post more RDMA Reads then the remote EP outstanding RDMA Read incoming attribute is. Thus, they do not rely on the underlying Transport enforcing it.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Unsafe

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_ep_post_rdma_write – write all data to the remote data buffer

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ep_post_rdma_read (
    IN    DAT_EP_HANDLE        ep_handle,
    IN    DAT_COUNT            num_segments,
    IN    DAT_LMR_TRIPLET      *local_iov,
    IN    DAT_DTO_COOKIE       user_cookie,
    IN    DAT_RMR_TRIPLET      *remote_buffer,
    IN    DAT_COMPLETION_FLAGS completion_flags
    )
```

Parameters

ep_handle Handle for an instance of the Endpoint.

num_segments Number of *lmr_triplets* in *local_iov*.

local_iov I/O Vector that specifies the local buffer from which the data is transferred.

user_cookie User-provided cookie that is returned to the Consumer at the completion of the RDMA Write.

remote_buffer A pointer to an RMR Triplet that specifies the remote buffer from which the data is read.

completion_flags Flags for posted RDMA read. The default `DAT_COMPLETION_DEFAULT_FLAG` is 0x00. Other values are as follows:

Completion Suppression	`DAT_COMPLETION_SUPPRESS_FLAG`	
	0x01	Suppress successful Completion.
Notification of Completion	`DAT_COMPLETION_UNSIGNALLED_FLAG`	
	0x04	Non-notification completion. Local Endpoint must be configured for Notification Suppression.
Barrier Fence	`DAT_COMPLETION_BARRIER_FENCE_FLAG`	
	0x08	Request for Barrier Fence.

Description The `dat_ep_post_rdma_write()` function requests the transfer of all the data specified by the *local_iov* over the connection of the *ep_handle* Endpoint into the *remote_buffer*.

The *num_segments* parameter specifies the number of segments in the *local_iov*. The *local_iov* segments are traversed in the I/O Vector order until all the data is transferred.

A Consumer must not modify the *local_iov* or its content until the DTO is completed. When a Consumer does not adhere to this rule, the behavior of the Provider and the underlying Transport is not defined. Providers that allow Consumers to get ownership of the *local_iov* but not the memory it specifies back after the `dat_ep_post_rdma_write()` returns should document this behavior and also specify its support in Provider attributes. This behavior allows Consumers full control of the *local_iov* after `dat_ep_post_rdma_write()` returns. Because this behavior is not guaranteed by all Providers, portable Consumers should not rely on this behavior. Consumers should not rely on the Provider copying *local_iov* information.

The `DAT_SUCCESS` return of the `dat_ep_post_rdma_write()` is at least the equivalent of posting an RDMA Write operation directly by native Transport. Providers should avoid resource allocation as part of `dat_ep_post_rdma_write()` to ensure that this operation is nonblocking and thread safe for an UpCall.

The completion of the posted RDMA Write is reported to the Consumer asynchronously through a DTO Completion event based on the specified *completion_flags* value. The value of `DAT_COMPLETION_UNSIGNALLED_FLAG` is only valid if the Endpoint Request Completion Flags `DAT_COMPLETION_UNSIGNALLED_FLAG`. Otherwise, `DAT_INVALID_PARAMETER` is returned.

The *user_cookie* allows Consumers to have unique identifiers for each DTO. These identifiers are completely under user control and are opaque to the Provider. There is no requirement on the Consumer that the value *user_cookie* should be unique for each DTO. The *user_cookie* is returned to the Consumer in the Completion event for the posted RDMA Write.

The operation is valid for the Endpoint in the `DAT_EP_STATE_CONNECTED` and `DAT_EP_STATE_DISCONNECTED` states. If the operation returns successfully for the Endpoint in the `DAT_EP_STATE_DISCONNECTED` state, the posted RDMA Write is immediately flushed to *request_evd_handle*.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INSUFFICIENT_RESOURCES`	The operation failed due to resource limitations.
`DAT_INVALID_PARAMETER`	Invalid parameter. For example, one of the IOV segments pointed to a memory outside its LMR.
`DAT_INVALID_HANDLE`	The *ep_handle* parameter is invalid.
`DAT_INVALID_STATE`	A parameter is in an invalid state. Endpoint was not in the `DAT_EP_STATE_CONNECTED` or `DAT_EP_STATE_DISCONNECTED` state.
`DAT_LENGTH_ERROR`	The size of the receiving buffer is too small for sending buffer data. The size of the remote buffer is too small for the data of the local buffer.
`DAT_PROTECTION_VIOLATION`	Protection violation for local or remote memory access. Protection Zone mismatch between either an LMR of one

	of the *local_iov* segments and the local Endpoint or the *rmr_context* and the remote Endpoint.
`DAT_PRIVILEGES_VIOLATION`	Privileges violation for local or remote memory access. Either one of the LMRs used in *local_iov* is invalid or does not have the local read privileges, or *rmr_context* does not have the remote write privileges.

Usage For best RDMA Write operation performance, the Consumer should align each buffer segment of *local_iov* to the Optimal Buffer Alignment attribute of the Provider. For portable applications, the Consumer should align each buffer segment of *local_iov* to the `DAT_OPTIMAL_ALIGNMENT`.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Unsafe

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_ep_post_recv – receive data over the connection of the Endpoint

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ep_post_recv (
    IN    DAT_EP_HANDLE        ep_handle,
    IN    DAT_COUNT            num_segments,
    IN    DAT_LMR_TRIPLET      *local_iov,
    IN    DAT_DTO_COOKIE       user_cookie,
    IN    DAT_COMPLETION_FLAGS completion_flags
    )
```

Parameters

ep_handle Handle for an instance of the Endpoint.

num_segments Number of *lmr_triplets* in *local_iov*. Can be 0 for receiving a 0 size message.

local_iov I/O Vector that specifies the local buffer to be filled. Can be `NULL` for receiving a 0 size message.

user_cookie: User-provided cookie that is returned to the Consumer at the completion of the Receive DTO. Can be `NULL`.

completion_flags Flags for posted Receive. The default `DAT_COMPLETION_DEFAULT_FLAG` is 0x00. Other values are as follows:

Notification of Completion	`DAT_COMPLETION_UNSIGNALLED_FLAG`
0x04	Non-notification completion. Local Endpoint must be configured for Unsignaled CompletionNotification Suppression.

Description The `dat_ep_post_recv()` function requests the receive of the data over the connection of the *ep_handle* Endpoint of the incoming message into the *local_iov*.

The *num_segments* parameter specifies the number of segments in the *local_iov*. The *local_iov* segments are filled in the I/O Vector order until the whole message is received. This ensures that all the "front" segments of the *local_iov* I/O Vector are completely filled, only one segment is partially filled, if needed, and all segments that follow it are not filled at all.

The *user_cookie* allows Consumers to have unique identifiers for each DTO. These identifiers are completely under user control and are opaque to the Provider. There is no requirement on the Consumer that the value *user_cookie* should be unique for each DTO. The *user_cookie* is returned to the Consumer in the Completion event for the posted Receive.

The completion of the posted Receive is reported to the Consumer asynchronously through a DTO Completion event based on the configuration of the connection for Solicited Wait and the specified *completion_flags* value for the matching Send. The value of `DAT_COMPLETION_UNSIGNALLED_FLAG` is only valid if the Endpoint Recv Completion Flags `DAT_COMPLETION_UNSIGNALLED_FLAG`. Otherwise, `DAT_INVALID_PARAMETER` is returned.

A Consumer must not modify the *local_iov* or its content until the DTO is completed. When a Consumer does not adhere to this rule, the behavior of the Provider and the underlying Transport is not defined. Providers that allow Consumers to get ownership of the *local_iov* but not the memory it specified back after the `dat_ep_post_recv()` returns should document this behavior and also specify its support in Provider attributes. This behavior allows Consumer full control of the *local_iov* content after `dat_ep_post_recv()` returns. Because this behavior is not guaranteed by all Providers, portable Consumers should not rely on this behavior. Consumers shouldnot rely on the Provider copying *local_iov* information.

The `DAT_SUCCESS` return of the `dat_ep_post_recv()` is at least the equivalent of posting a Receive operation directly by native Transport. Providers should avoid resource allocation as part of `dat_ep_post_recv()` to ensure that this operation is nonblocking and thread safe for an UpCall.

If the size of an incoming message is larger than the size of the *local_iov*, the reported status of the posted Receive DTO in the corresponding Completion DTO event is `DAT_DTO_LENGTH_ERROR`. If the reported status of the Completion DTO event corresponding to the posted Receive DTO is not `DAT_DTO_SUCCESS`, the content of the *local_iov* is not defined.

The operation is valid for all states of the Endpoint. The actual data transfer does not take place until the Endpoint is in the `DAT_EP_STATE_CONNECTED` state. The operation on the Endpoint in `DAT_EP_STATE_DISCONNECTED` is allowed. If the operation returns successfully, the posted Recv is immediately flushed to *recv_evd_handle*.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INSUFFICIENT_RESOURCES`	The operation failed due to resource limitations.
`DAT_INVALID_PARAMETER`	Invalid parameter. For example, one of the IOV segments pointed to a memory outside its LMR.
`DAT_INVALID_HANDLE`	The *ep_handle* parameter is invalid.
`DAT_PROTECTION_VIOLATION`	Protection violation for local or remote memory access. Protection Zone mismatch between an LMR of one of the *local_iov* segments and the local Endpoint.
`DAT_PRIVILEGES_VIOLATION`	Privileges violation for local or remote memory access. One of the LMRs used in *local_iov* was either invalid or did not have the local read privileges.

Usage For best Recv operation performance, the Consumer should align each buffer segment of *local_iov* to the Optimal Buffer Alignment attribute of the Provider. For portable applications, the Consumer should align each buffer segment of *local_iov* to the `DAT_OPTIMAL_ALIGNMENT`.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Unsafe

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_ep_post_send – transfer data to the remote side

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ep_post_send (
    IN    DAT_EP_HANDLE        ep_handle,
    IN    DAT_COUNT            num_segments,
    IN    DAT_LMR_TRIPLET      *local_iov,
    IN    DAT_DTO_COOKIE       user_cookie,
    IN    DAT_COMPLETION_FLAGS completion_flags
    )
```

Parameters

ep_handle Handle for an instance of the Endpoint.

num_segments Number of *lmr_triplets* in *local_iov*. Can be 0 for 0 size message.

local_iov I/O Vector that specifies the local buffer that contains data to be transferred. Can be `NULL` for 0 size message.

user_cookie: User-provided cookie that is returned to the Consumer at the completion of the send. Can be `NULL`.

completion_flags Flags for posted Send. The default `DAT_COMPLETION_DEFAULT_FLAG` is 0x00. Other values are as follows:

Completion Suppression	`DAT_COMPLETION_SUPPRESS_FLAG`	
	0x01	Suppress successful Completion.
Solicited Wait	`DAT_COMPLETION_SOLICITED_WAIT_FLAG`	
	0x02	Request for notification completion for matching receive on the other side of the connection.
Notification of Completion	`DAT_COMPLETION_UNSIGNALLED_FLAG`	
	0x04	Non-notification completion. Local Endpoint must be configured for Notification Suppression.
Barrier Fence	`DAT_COMPLETION_BARRIER_FENCE_FLAG`	
	0x08	Request for Barrier Fence.

Description The `dat_ep_post_send()` function requests a transfer of all the data from the *local_iov* over the connection of the *ep_handle* Endpoint to the remote side.

The *num_segments* parameter specifies the number of segments in the *local_iov*. The *local_iov* segments are traversed in the I/O Vector order until all the data is transferred.

A Consumer cannot modify the *local_iov* or its content until the DTO is completed. When a Consumer does not adhere to this rule, the behavior of the Provider and the underlying Transport is not defined. Providers that allow Consumers to get ownership of the *local_iov* back after the `dat_ep_post_send()` returns should document this behavior and also specify its support in Provider attributes. This behavior allows Consumers full control of the *local_iov*, but not the memory it specifies after `dat_ep_post_send()` returns. Because this behavior is not guaranteed by all Providers, portable Consumers should not rely on this behavior. Consumers should not rely on the Provider copying *local_iov* information.

The `DAT_SUCCESS` return of the `dat_ep_post_send()` is at least the equivalent of posting a Send operation directly by native Transport. Providers should avoid resource allocation as part of `dat_ep_post_send()` to ensure that this operation is nonblocking and thread safe for an UpCall.

The completion of the posted Send is reported to the Consumer asynchronously through a DTO Completion event based on the specified *completion_flags* value. The value of `DAT_COMPLETION _UNSIGNALLED_FLAG` is only valid if the Endpoint Request Completion Flags `DAT_COMPLETION_UNSIGNALLED_FLAG`. Otherwise, `DAT_INVALID_PARAMETER` is returned.

The *user_cookie* allows Consumers to have unique identifiers for each DTO. These identifiers are completely under user control and are opaque to the Provider. There is no requirement on the Consumer that the value *user_cookie* should be unique for each DTO. The *user_cookie* is returned to the Consumer in the Completion event for the posted Send.

The operation is valid for the Endpoint in the `DAT_EP_STATE_CONNECTED` and `DAT_EP_STATE_DISCONNECTED` states. If the operation returns successfully for the Endpoint in the `DAT_EP_STATE_DISCONNECTED` state, the posted Send is immediately flushed to *request_evd_handle*.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INSUFFICIENT_RESOURCES`	The operation failed due to resource limitations.
`DAT_INVALID_PARAMETER`	Invalid parameter. For example, one of the IOV segments pointed to a memory outside its LMR.
`DAT_INVALID_HANDLE`	The *ep_handle* parameter is invalid.
`DAT_INVALID_STATE`	A parameter is in an invalid state. Endpoint was not in the `DAT_EP_STATE_CONNECTED` or `DAT_EP_STATE_DISCONNECTED` state.

`DAT_PROTECTION_VIOLATION`	Protection violation for local or remote memory access. Protection Zone mismatch between an LMR of one of the *local_iov* segments and the local Endpoint.
`DAT_PRIVILEGES_VIOLATION`	Privileges violation for local or remote memory access. One of the LMRs used in *local_iov* was either invalid or did not have the local read privileges.

Usage For best Send operation performance, the Consumer should align each buffer segment of *local_iov* to the Optimal Buffer Alignment attribute of the Provider. For portable applications, the Consumer should align each buffer segment of *local_iov* to the `DAT_OPTIMAL_ALIGNMENT`.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Unsafe

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_ep_query – provide parameters of the Endpoint

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ep_query (
    IN    DAT_EP_HANDLE        ep_handle,
    IN    DAT_EP_PARAM_MASK    ep_param_mask,
    OUT   DAT_EP_PARAM         *ep_param
    )
```

Parameters

ep_handle	Handle for an instance of the Endpoint.
ep_param_mask	Mask for Endpoint parameters.
ep_param	Pointer to a Consumer-allocated structure that the Provider fills with Endpoint parameters.

Description The `dat_ep_query()` function provides the Consumer parameters, including attributes and status, of the Endpoint. Consumers pass in a pointer to Consumer-allocated structures for Endpoint parameters that the Provider fills.

The *ep_param_mask* parameter allows Consumers to specify which parameters to query. The Provider returns values for *ep_param_mask* requested parameters. The Provider can return values for any other parameters.

Some of the parameters only have values for certain Endpoint states. Specifically, the values for *remote_ia_address* and *remote_port_qual* are valid only for Endpoints in the `DAT_EP_STATE_PASSIVE_CONNECTION_PENDING`, `DAT_EP_STATE_ACTIVE_CONNECTION_PENDING`, `DAT_EP_STATE_TENTATIVE_CONNECTION_PENDING`, `DAT_EP_STATE_DISCONNECT_PENDING`, `DAT_EP_STATE_COMPLETION_PENDING`, or `DAT_EP_STATE_CONNECTED` states. The values of *local_port_qual* is valid only for Endpoints in the `DAT_EP_STATE_PASSIVE_CONNECTION_PENDING`, `DAT_EP_STATE_ACTIVE_CONNECTION_PENDING`, `DAT_EP_STATE_DISCONNECT_PENDING`, `DAT_EP_STATE_COMPLETION_PENDING`, or `DAT_EP_STATE_CONNECTED` states, and might be valid for `DAT_EP_STATE_UNCONNECTED`, `DAT_EP_STATE_RESERVED`, `DAT_EP_STATE_TENTATIVE_CONNECTION_PENDING`, `DAT_EP_STATE_PASSIVE_CONNECTION_PENDING`, and `DAT_EP_STATE_UNCONNECTED` states.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INVALID_HANDLE`	The *ep_handle* parameter is invalid.
`DAT_INVALID_PARAMETER`	The *ep_param_mask* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Unsafe

See Also libdat(3LIB), attributes(5)

Name dat_ep_recv_query – provide Endpoint receive queue consumption on SRQ

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ep_recv_query (
        IN      DAT_EP_HANDLE       ep_handle,
        OUT     DAT_COUNT           *nbufs_allocated,
        OUT     DAT_COUNT           *bufs_alloc_span
    )
```

Parameters

ep_handle Handle for an instance of the EP.

nbufs_allocated The number of buffers at the EP for which completions have not yet been generated.

bufs_alloc_span The span of buffers that EP needs to complete arriving messages.

Description The `dat_ep_recv_query()` function provides to the Consumer a snapshot for Recv buffers on EP. The values for *nbufs_allocated* and *bufs_alloc_span* are not defined when `DAT_RETURN` is not `DAT_SUCCESS`.

The Provider might not support *nbufs_allocated*, *bufs_alloc_span* or both. Check the Provider attribute for EP Recv info support. When the Provider does not support both of these counts, the return value for the operation can be `DAT_MODEL_NOT_SUPPORTED`.

If *nbufs_allocated* is not `NULL`, the count pointed to by *nbufs_allocated* will return a snapshot count of the number of buffers allocated to *ep_handle* but not yet completed.

Once a buffer has been allocated to an EP, it will be completed to the EP *recv_evd* if the EVD has not overflowed. When an EP does not use SRQ, a buffer is allocated as soon as it is posted to the EP. For EP that uses SRQ, a buffer is allocated to the EP when EP removes it from SRQ.

If *bufs_alloc_span* is not `NULL`, then the count to which *bufs_alloc_span* pointed will return the span of buffers allocated to the *ep_handle*. The span is the number of additional successful Recv completions that EP can generate if all the messages it is currently receiving will complete successfully.

If a message sequence number is assigned to all received messages, the buffer span is the difference between the latest message sequence number of an allocated buffer minus the latest message sequence number for which completion has been generated. This sequence number only counts Send messages of remote Endpoint of the connection.

The Message Sequence Number (MSN) represents the order that Send messages were submitted by the remote Consumer. The ordering of sends is intrinsic to the definition of a reliable service. Therefore every send message does have a MSN whether or not the native transport has a field with that name.

For both *nbufs_allocated* and *bufs_alloc_span*, the Provider can return the reserved value DAT_VALUE_UNKNOWN if it cannot obtain the requested count at a reasonable cost.

Return Values

DAT_SUCCESS	The operation was successful.
DAT_INVALID_PARAMETER	Invalid parameter.
DAT_INVALID_HANDLE	The DAT handle ep_handle is invalid.
DAT_MODEL_NOT_SUPPORTED	The requested Model was not supported by the Provider.

Usage

If the Provider cannot support the query for *nbufs_allocated* or *bufs_alloc_span*, the value returned for that attribute must be DAT_VALUE_UNKNOWN.

An implementation that processes incoming packets out of order and allocates from SRQs on an arrival basis can have gaps in the MSNs associated with buffers allocated to an Endpoint.

For example, suppose Endpoint X has received buffer fragments for MSNs 19, 22, and 23. With arrival ordering, the EP would have allocated three buffers from the SRQ for messages 19, 22, and 23. The number allocated would be 3, but the span would be 5. The difference of two represents the buffers that will have to be allocated for messages 20 and 21. They have not yet been allocated, but messages 22 and 23 will not be delivered until after messages 20 and 21 have not only had their buffers allocated but have also completed.

An implementation can choose to allocate 20 and 21 as soon as any higher buffer is allocated. This makes sense if you presume that this is a valid connection, because obviously 20 and 21 are in flight. However, it creates a greater vulnerability to Denial Of Service attacks. There are also other implementation tradeoffs, so the Consumer should accept that different RNICs for iWARP will employ different strategies on when to perform these allocations.

Each implementation will have some method of tracking the receive buffers already associated with an EP and knowing which buffer matches which incoming message, though those methods might vary. In particular, there are valid implementations such as linked lists, where a count of the outstanding buffers is not instantly available. Such implementations would have to scan the allocated list to determine both the number of buffers and their span. If such a scan is necessary, it is important that it be only a single scan. The set of buffers that was counted must be the same set of buffers for which the span is reported.

The implementation should not scan twice, once to count the buffers and then again to determine their span. Not only is it inefficient, but it might produce inconsistent results if buffers were completed or arrived between the two scans.

Other implementations can simply maintain counts of these values to easily filter invalid packets. If so, these status counters should be updated and referenced atomically.

The implementation must never report *n* buffers in a span that is less than *n*.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.2
MT-Level	Unsafe

See Also dat_ep_create(3DAT), dat_srq_create(3DAT), dat_srq_free(3DAT), dat_srq_query(3DAT), dat_ep_set_watermark(3DAT), libdat(3LIB), attributes(5)

Name dat_ep_reset – transition the local Endpoint from a Disconnected to an Unconnected state

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ep_reset (
    IN    DAT_EP_HANDLE    ep_handle
    )
```

Parameters *ep_handle* Handle for an instance of Endpoint.

Description The `dat_ep_reset()` function transitions the local Endpoint from a Disconnected to an Unconnected state.

The operation might cause the loss of any completions of previously posted DTOs and RMRs that were not dequeued yet.

The `dat_ep_reset()` function is valid for both Disconnected and Unconnected states. For Unconnected state, the operation is no-op because the Endpoint is already in an Unconnected state. For an Unconnected state, the preposted Recvs are not affected by the call.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INVALID_HANDLE`	ep_handle is invalid.
`DAT_INVALID_STATE`	Parameter in an invalid state. Endpoint is not in the valid state for reset.

Usage If the Consumer wants to ensure that all Completions are dequeued, the Consumer can post DTO or RMR operations as a "marker" that are flushed to *recv_evd_handle* or *request_evd_handle*. Now, when the Consumer dequeues the completion of the "marker" from the EVD, it is guaranteed that all previously posted DTO and RMR completions for the Endpoint were dequeued for that EVD. Now, it is safe to reset the Endpoint without losing any completions.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Unsafe

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_ep_set_watermark – set high watermark on Endpoint

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ep_set_watermark (
    IN      DAT_EP_HANDLE      ep_handle,
    IN      DAT_COUNT          soft_high_watermark,
    IN      DAT_COUNT          hard_high_watermark
    )
```

Parameters

ep_handle	The handle for an instance of an Endpoint.
soft_high_watermark	The soft high watermark for the number of Recv buffers consumed by the Endpoint.
hard_high_watermark	The hard high watermark for the number of Recv buffers consumed by the Endpoint.

Description The `dat_ep_set_watermark()` function sets the soft and hard high watermark values for EP and arms EP for generating asynchronous events for high watermarks. An asynchronous event will be generated for IA *async_evd* when the number of Recv buffers at EP exceeds the soft high watermark for the first time. A connection broken event will be generated for EP *connect_evd* when the number of Recv buffers at EP exceeds the hard high watermark. These can occur during this call or when EP takes a buffer from the SRQ or EP RQ. The soft and hard high watermark asynchronous event generation and setting are independent of each other.

The asynchronous event for a soft high watermark is generated only once per setting. Once an event is generated, no new asynchronous events for the soft high watermark is generated until the EP is again set for the soft high watermark. If the Consumer is once again interested in the event, the Consumer should again set the soft high watermark.

If the Consumer is not interested in a soft or hard high watermark, the value of `DAT_WATERMARK_INFINITE` can be specified for the case that is the default value. This value specifies that a non-asynchronous event will be generated for a high watermark EP attribute for which this value is set. It does not prevent generation of a connection broken event for EP when no Recv buffer is available for a message arrived on the EP connection.

The operation is supported for all states of Endpoint.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INVALID_HANDLE`	The *ep_handle* argument is an invalid DAT handle.
`DAT_INVALID_PARAMETER`	One of the parameters is invalid.
`DAT_MODEL_NOT_SUPPORTED`	The requested Model was not supported by the Provider. The Provider does not support EP Soft or Hard High

Watermarks.

Usage For a hard high watermark, the Provider is ready to generate a connection broken event as soon as the connection is established.

If the asynchronous event for a soft or hard high watermark has not yet been generated, this call simply modifies the values for these attributes. The Provider remains armed for generation of these asynchronous events.

Regardless of whether an asynchronous event for the soft and hard high watermark has been generated, this operation will set the generation of an asynchronous event with the Consumer-provided high watermark values. If the new high watermark values are below the current number of Receive DTOs at EP, an asynchronous event will be generated immediately. Otherwise the old soft or hard (or both) high watermark values are simply replaced with the new ones.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.2
MT-Level	Unsafe

See Also `dat_ep_create(3DAT)`, `dat_ep_recv_query(3DAT)`, `dat_srq_create(3DAT)`, `dat_srq_free(3DAT)`, `dat_srq_post_recv(3DAT)`, `dat_srq_query(3DAT)`, `dat_srq_resize(3DAT)`, `dat_srq_set_lw(3DAT)`, `libdat(3LIB)`, `attributes(5)`

Name dat_evd_clear_unwaitable – transition the Event Dispatcher into a waitable state

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_evd_clear_unwaitable(
    IN    DAT_EVD_HANDLE     evd_handle
    )
```

Parameters *evd_handle* Handle for an instance of Event Dispatcher.

Description The `dat_evd_clear_unwaitable()` transitions the Event Dispatcher into a waitable state. In this state, calls to `dat_evd_wait(3DAT)` are permitted on the EVD. The actual state of the Event Dispatcher is accessible through `dat_evd_query(3DAT)` and is `DAT_EVD_WAITABLE` after the return of this operation.

This call does not affect a CNO associated with this EVD at all. Events arriving on the EVD after it is set waitable still trigger the CNO (if appropriate), and can be retrieved with `dat_evd_dequeue(3DAT)`.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *evd_handle* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Safe

See Also `dat_evd_dequeue(3DAT)`, `dat_evd_query(3DAT)`, `dat_evd_set_unwaitable(3DAT)`, `dat_evd_wait(3DAT)`, `libdat(3LIB)`, `attributes(5)`

Name dat_evd_dequeue – remove the first event from the Event Dispatcher event queue

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_evd_dequeue(
    IN    DAT_EVD_HANDLE    evd_handle,
    OUT   DAT_EVENT         *event
    )
```

Parameters

evd_handle Handle for an instance of the Event Dispatcher.

event Pointer to the Consumer-allocated structure that Provider fills with the event data.

Description The `dat_evd_dequeue()` function removes the first event from the Event Dispatcher event queue and fills the Consumer allocated *event* structure with event data. The first element in this structure provides the type of the event; the rest provides the event-type-specific parameters. The Consumer should allocate an *event* structure big enough to hold any event that the Event Dispatcher can deliver.

For all events the Provider fills the `dat_event` that the Consumer allocates. So for all events, all fields of `dat_event` are OUT from the Consumer point of view. For `DAT_CONNECTION_REQUEST_EVENT`, the Provider creates a Connection Request whose *cr_handle* is returned to the Consumer in `DAT_CR_ARRIVAL_EVENT_DATA`. That object is destroyed by the Provider as part of `dat_cr_accept(3DAT)`, `dat_cr_reject(3DAT)`, or `dat_cr_handoff(3DAT)`. The Consumer should not use *cr_handle* or any of its parameters, including *private_data*, after one of these operations destroys the Connection Request.

For `DAT_CONNECTION_EVENT_ESTABLISHED` for the Active side of connection establishment, the Provider returns the pointer for *private_data* and the *private_data_size*. For the Passive side, `DAT_CONNECTION_EVENT_ESTABLISHED` event *private_data* is not defined and *private_data_size* returns zero. The Provider is responsible for the memory allocation and deallocation for *private_data*. The *private_data* is valid until the Active side Consumer destroys the connected Endpoint (`dat_ep_free(3DAT)`), or transitions the Endpoint into Unconnected state so it is ready for the next connection. So while the Endpoint is in Connected, Disconnect Pending, or Disconnected state, the private_data of `DAT_CONNECTION_REQUEST_EVENT` is still valid for Active side Consumers.

Provider must pass to the Consumer the entire Private Data that the remote Consumer provided for `dat_ep_connect(3DAT)`, `dat_ep_dup_connect(3DAT)`, and `dat_cr_accept()`. If the Consumer provides more data than the Provider and Transport can support (larger than IA Attribute of *max_private_data_size*), `DAT_INVALID_PARAMETER` is returned for that operation.

The returned event that was posted from an Event Stream guarantees Consumers that all events that were posted from the same Event Stream prior to the returned event were already returned to a Consumer directly through a `dat_evd_dequeue()` or `dat_evd_wait(3DAT)` operation.

The ordering of events dequeued by overlapping calls to `dat_evd_wait()` or `dat_evd_dequeue()` is not specified.

Return Values

`DAT_SUCCESS`	The operation was successful. An event was returned to a Consumer.
`DAT_INVALID_HANDLE`	Invalid DAT handle; evd_handle is invalid.
`DAT_QUEUE_EMPTY`	There are no entries on the Event Dispatcher queue.
`DAT_INVALID_STATE`	One of the parameters was invalid for this operation. There is already a waiter on the EVD.

Usage

No matter how many contexts attempt to dequeue from an Event Dispatcher, each event is delivered exactly once. However, which Consumer receives which event is not defined. The Provider is not obligated to provide the first caller the first event unless it is the only caller. The Provider is not obligated to ensure that the caller receiving the first event executes earlier than contexts receiving later events.

Preservation of event ordering within an Event Stream is an important feature of the DAT Event Model. Consumers are cautioned that overlapping or concurrent calls to `dat_evd_dequeue()` from multiple contexts can undermine this ordering information. After multiple contexts are involved, the Provider can only guarantee the order that it delivers events into the EVD. The Provider cannot guarantee that they are processed in the correct order.

Although calling `dat_evd_dequeue()` does not cause a context switch, the Provider is under no obligation to prevent one. A context could successfully complete a dequeue, and then reach the end of its timeslice, before returning control to the Consumer code. Meanwhile, a context receiving a later event could be executing.

The Event ordering is preserved when dequeueing is serialized. Potential Consumer serialization methods include, but are not limited to, performing all dequeueing from a single context or protecting dequeueing by way of lock or semaphore.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Safe

See Also dat_cr_accept(3DAT), dat_cr_handoff(3DAT), dat_cr_reject(3DAT), dat_ep_connect(3DAT), dat_ep_dup_connect(3DAT), dat_ep_free(3DAT), dat_evd_wait(3DAT)libdat(3LIB), attributes(5)

Name dat_evd_disable – disable the Event Dispatcher

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_evd_disable(
    IN    DAT_EVD_HANDLE     evd_handle
    )
```

Parameters *evd_handle* Handle for an instance of Event Dispatcher.

Description The `dat_evd_disable()` function disables the Event Dispatcher so that the arrival of an event does not affect the associated CNO.

If the Event Dispatcher is already disabled, this operation is no-op.

Events arriving on this EVD might cause waiters on the associated CNO to be awakened after the return of this routine because an unblocking a CNO waiter is already "in progress" at the time this routine is called or returned.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *evd_handle* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Safe

See Also `dat_evd_enable(3DAT)`, `libdat(3LIB)`, `attributes(5)`

Name dat_evd_enable – enable the Event Dispatcher

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_evd_enable(
    IN    DAT_EVD_HANDLE     evd_handle
    )
```

Parameters *evd_handle* Handle for an instance of Event Dispatcher.

Description The `dat_evd_enable()` function enables the Event Dispatcher so that the arrival of an event can trigger the associated CNO. The enabling and disabling EVD has no effect on direct waiters on the EVD. However, direct waiters effectively take ownership of the EVD, so that the specified CNO is not triggered even if is enabled.

If the Event Dispatcher is already enabled, this operation is no-op.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *evd_handle* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Safe

See Also `dat_evd_disable(3DAT)`, `libdat(3LIB)`, `attributes(5)`

Name dat_evd_free – destroy an instance of the Event Dispatcher

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_evd_free (
    IN    DAT_EVD_HANDLE    evd_handle
    )
```

Parameters *evd_handle* Handle for an instance of the Event Dispatcher.

Description The `dat_evd_free()` function destroys a specified instance of the Event Dispatcher.

All events on the queue of the specified Event Dispatcher are lost. The destruction of the Event Dispatcher instance does not have any effect on any DAT Objects that originated an Event Stream that had fed events to the Event Dispatcher instance. There should be no event streams feeding the Event Dispatcher and no threads blocked on the Event Dispatcher when the EVD is being closed as at the time when it was created.

Use of the handle of the destroyed Event Dispatcher in any consequent operation fails.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INVALID_HANDLE`	The *evd_handle* parameter is invalid
`DAT_INVALID_STATE`	Invalid parameter. There are Event Streams associated with the Event Dispatcher feeding it.

Usage Consumers are advised to destroy all Objects that originate Event Streams that feed an instance of the Event Dispatcher before destroying it. An exception to this rule is Event Dispatchers of an IA.

Freeing an IA automatically destroys all Objects associated with it directly and indirectly, including Event Dispatchers.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Unsafe

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_evd_modify_cno – change the associated CNO for the Event Dispatcher

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_evd_modify_cno (
    IN    DAT_EVD_HANDLE    evd_handle,
    IN    DAT_CNO_HANDLE    cno_handle
    )
```

Parameters

evd_handle Handle for an instance of the Event Dispatcher.

cno_handle Handle for a CNO. The value of `DAT_NULL_HANDLE` specifies no CNO.

Description The `dat_evd_modify_cno()` function changes the associated CNO for the Event Dispatcher.

A Consumer can specify the value of `DAT_HANDLE_NULL` for *cno_handle* to associate not CNO with the Event Dispatcher instance.

Upon completion of the `dat_evd_modify_cno()` operation, the passed IN new CNO is used for notification. During the operation, an event arrival can be delivered to the old or new CNO. If Notification is generated by EVD, it is delivered to the new or old CNO.

If the EVD is enabled at the time `dat_evd_modify_cno()` is called, the Consumer must be prepared to collect a notification event on the EVD's old CNO as well as the new one. Checking immediately prior to calling `dat_evd_modify_cno()` is not adequate. A notification could have been generated after the prior check and before the completion of the change.

The Consumer can avoid the risk of missed notifications either by temporarily disabling the EVD, or by checking the prior CNO after invoking this operation. The Consumer can disable EVD before a `dat_evd_modify_cno()` call and enable it afterwards. This ensures that any notifications from the EVD are delivered to the new CNO only.

If this function is used to disassociate a CNO from the EVD, events arriving on this EVD might cause waiters on that CNO to awaken after returning from this routine because of unblocking a CNO waiter already "in progress" at the time this routine is called. If this is the case, the events causing that unblocking are present on the EVD upon return from the `dat_evd_modify_cno()` call and can be dequeued at that time

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` Invalid DAT handle.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Unsafe

See Also libdat(3LIB), attributes(5)

Name dat_evd_post_se – post Software event to the Event Dispatcher event queue

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_evd_post_se(
    IN        DAT_EVD_HANDLE     evd_handle,
    IN const  DAT_EVENT          *event
    )
```

Parameters

evd_handle Handle for an instance of the Event Dispatcher

event A pointer to a Consumer created Software Event.

Description The `dat_evd_post_se()` function posts Software events to the Event Dispatcher event queue. This is analogous to event arrival on the Event Dispatcher software Event Stream. The *event* that the Consumer provides adheres to the event format as defined in <`dat.h`>. The first element in the *event* provides the type of the event (`DAT_EVENT_TYPE_SOFTWARE`); the rest provide the event-type-specific parameters. These parameters are opaque to a Provider. Allocation and release of the memory referenced by the *event* pointer in a software event are the Consumer's responsibility.

There is no ordering between events from different Event Streams. All the synchronization issues between multiple Consumer contexts trying to post events to an Event Dispatcher instance simultaneously are left to a Consumer.

If the event queue is full, the operation is completed unsuccessfully and returns `DAT_QUEUE_FULL`. The *event* is not queued. The queue overflow condition does takes place and, therefore, the asynchronous Event Dispatcher is not effected.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *evd_handle* parameter is invalid.

`DAT_INVALID_PARAMETER` The *event* parameter is invalid.

`DAT_QUEUE_FULL` The Event Dispatcher queue is full.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Unsafe

See Also libdat(3LIB), attributes(5)

Name dat_evd_query – provide parameters of the Event Dispatcher,

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_evd_query (
    IN    DAT_EVD_HANDLE        evd_handle,
    IN    DAT_EVD_PARAM_MASK    evd_param_mask,
    OUT   DAT_EVD_PARAM         *evd_param
    )
```

Parameters

evd_handle	Handle for an instance of Event Dispatcher.
evd_param_mask	Mask for EVD parameters
evd_param	Pointer to a Consumer-allocated structure that the Provider fills for Consumer-requested parameters.

Description The `dat_evd_query()` function provides to the Consumer parameters of the Event Dispatcher, including the state of the EVD (enabled/disabled). The Consumer passes in a pointer to the Consumer-allocated structures for EVD parameters that the Provider fills.

The *evd_param_mask* parameter allows Consumers to specify which parameters to query. The Provider returns values for *evd_param_mask* requested parameters. The Provider can return values for any of the other parameters.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INVALID_HANDLE`	The *evd_handle* parameter is invalid.
`DAT_INVALID_PARAMETER`	The *evd_param_mask* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Unsafe

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_evd_resize – modify the size of the event queue of Event Dispatcher

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_evd_resize(
    IN    DAT_EVD_HANDLE     evd_handle,
    IN    DAT_COUNT          evd_min_qlen
    )
```

Parameters

evd_handle Handle for an instance of Event Dispatcher.

evd_min_qlen New number of events the Event Dispatcher event queue must hold.

Description The `dat_evd_resize()` function modifies the size of the event queue of Event Dispatcher.

Resizing of Event Dispatcher event queue should not cause any incoming or current events on the event queue to be lost. If the number of entries on the event queue is larger then the requested evd_min_qlen, the operation can return `DAT_INVALID_STATE` and not change an instance of Event Dispatcher

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INVALID_HANDLE`	The *evd_handle* parameter is invalid.
`DAT_INVALID_PARAMETER`	The *evd_min_qlen* parameter is invalid
`DAT_INSUFFICIENT_RESOURCES`	The operation failed due to resource limitations
`DAT_INVALID_STATE`	Invalid parameter. The number of entries on the event queue of the Event Dispatcher exceeds the requested event queue length.

Usage This operation is useful when the potential number of events that could be placed on the event queue changes dynamically.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Unsafe

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_evd_set_unwaitable – transition the Event Dispatcher into an unwaitable state

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_evd_set_unwaitable(
    IN    DAT_EVD_HANDLE    evd_handle
    )
```

Parameters *evd_handle* Handle for an instance of Event Dispatcher.

Description The `dat_evd_set_unwaitable()` transitions the Event Dispatcher into an unwaitable state. In this state, calls to `dat_evd_wait(3DAT)` return synchronously with a `DAT_INVALID_STATE` error, and threads already blocked in `dat_evd_wait()` are awakened and return with a `DAT_INVALID_STATE` error without any further action by the Consumer. The actual state of the Event Dispatcher is accessible through `dat_evd_query(3DAT)` and is `DAT_EVD_UNWAITABLE` after the return of this operation.

This call does not affect a CNO associated with this EVD at all. Events arriving on the EVD after it is set unwaitable still trigger the CNO (if appropriate), and can be retrieved with `dat_evd_dequeue(3DAT)`. Because events can arrive normally on the EVD, the EVD might overflow; the Consumer is expected to protect against this possibility.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *evd_handle* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Safe

See Also `dat_evd_clear_unwaitable(3DAT)`, `dat_evd_dequeue(3DAT)`, `dat_evd_query(3DAT)`, `dat_evd_wait(3DAT)`, `libdat(3LIB)`, `attributes(5)`

Name dat_evd_wait – remove first event from the Event Dispatcher event queue

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_evd_wait(
    IN    DAT_EVD_HANDLE     evd_handle,
    IN    DAT_TIMEOUT        timeout,
    IN    DAT_COUNT          threshold,
    OUT   DAT_EVENT          *event,
    OUT   DAT_COUNT          *nmore
    )
```

Parameters

evd_handle Handle for an instance of the Event Dispatcher.

timeout The duration of time, in microseconds, that the Consumer is willing to wait for the event.

threshold The number of events that should be on the EVD queue before the operation should return with `DAT_SUCCESS`. The threshold must be at least 1.

event Pointer to the Consumer-allocated structure that the Provider fills with the event data.

nmore The snapshot of the queue size at the time of the operation return.

Description The `dat_evd_wait()` function removes the first event from the Event Dispatcher event queue and fills the Consumer-allocated *event* structure with event data. The first element in this structure provides the type of the event; the rest provides the event type-specific parameters. The Consumer should allocate an event structure big enough to hold any event that the Event Dispatcher can deliver.

For all events, the Provider fills the *dat_event* that the Consumer allocates. Therefore, for all events, all fields of *dat_event* are OUT from the Consumer point of view. For `DAT_CONNECTION_REQUEST_EVENT`, the Provider creates a Connection Request whose *cr_handle* is returned to the Consumer in `DAT_CR_ARRIVAL_EVENT_DATA`. That object is destroyed by the Provider as part of `dat_cr_accept(3DAT)`, `dat_cr_reject(3DAT)`, or `dat_cr_handoff(3DAT)`. The Consumer should not use *cr_handle* or any of its parameters, including *private_data*, after one of these operations destroys the Connection Request.

For `DAT_CONNECTION_EVENT_ESTABLISHED` for the Active side of connection establishment, the Provider returns the pointer for *private_data* and the *private_data_size*. For the Passive side, `DAT_CONNECTION_EVENT_ESTABLISHED` event *private_data* is not defined and *private_data_size* returns zero. The Provider is responsible for the memory allocation and deallocation for *private_data*. The *private_data* is valid until the Active side Consumer destroys the connected Endpoint (`dat_ep_free(3DAT)`), or transitions the Endpoint into Unconnected state so it is ready for the next connection. So, while the Endpoint is in

Connected, Disconnect Pending, or Disconnected state, the *private_data* of DAT_CONNECTION_REQUEST_EVENT is still valid for Active side Consumers.

Provider must pass to the Consumer the entire Private Data that the remote Consumer provided for dat_ep_connect(3DAT), dat_ep_dup_connect(3DAT), and dat_cr_accept(). If the Consumer provides more data than the Provider and Transport can support (larger than IA Attribute of *max_private_data_size*), DAT_INVALID_PARAMETER is returned for that operation.

A Consumer that blocks performing a dat_evd_wait() on an Event Dispatcher effectively takes exclusive ownership of that Event Dispatcher. Any other dequeue operation (dat_evd_wait() or dat_evd_dequeue(3DAT)) on the Event Dispatcher is rejected with a DAT_INVALID_STATE error code.

The CNO associated with the evd_handle() is not triggered upon event arrival if there is a Consumer blocked on dat_evd_wait() on this Event Dispatcher.

The *timeout* allows the Consumer to restrict the amount of time it is blocked waiting for the event arrival. The value of DAT_TIMEOUT_INFINITE indicates that the Consumer waits indefinitely for an event arrival. Consumers should use extreme caution in using this value.

When *timeout* value is reached and the number of events on the EVD queue is below the *threshold* value, the operation fails and returns DAT_TIMEOUT_EXPIRED. In this case, no event is dequeued from the EVD and the return value for the *event* argument is undefined. However, an *nmore* value is returned that specifies the snapshot of the number of the events on the EVD queue that is returned.

The *threshold* allows the Consumer to wait for a requested number of event arrivals prior to waking the Consumer. If the value of the *threshold* is larger than the Event Dispatcher queue length, the operation fails with the return DAT_INVALID_PARAMETER. If a non-positive value is specified for *threshold*, the operation fails and returns DAT_INVALID_PARAMETER.

If EVD is used by an Endpoint for a DTO completion stream that is configured for a Consumer-controlled event Notification (DAT_COMPLETION_UNSIGNALLED_FLAG or DAT_COMPLETION_SOLICITED_WAIT_FLAG for Receive Completion Type for Receives; DAT_COMPLETION_UNSIGNALLED_FLAG for Request Completion Type for Send, RDMA Read, RDMA Write and RMR Bind), the *threshold* value must be 1. An attempt to specify some other value for *threshold* for this case results in DAT_INVALID_STATE.

The returned value of *nmore* indicates the number of events left on the Event Dispatcher queue after the dat_evd_wait() returns. If the operation return value is DAT_SUCCESS, the *nmore* value is at least the value of (*threshold* -1). Notice that *nmore* is only a snapshot and the number of events can be changed by the time the Consumer tries to dequeue events with dat_evd_wait() with timeout of zero or with dat_evd_dequeue().

For returns other than DAT_SUCCESS, DAT_TIMEOUT_EXPIRED, and DAT_INTERRUPTED_CALL, the returned value of *nmore* is undefined.

The returned event that was posted from an Event Stream guarantees Consumers that all events that were posted from the same Event Stream prior to the returned event were already returned to a Consumer directly through a dat_evd_dequeue() or dat_evd_wait() operation.

If the return value is neither DAT_SUCCESS nor DAT_TIMEOUT_EXPIRED, then returned values of *nmore* and event are undefined. If the return value is DAT_TIMEOUT_EXPIRED, then the return value of event is undefined, but the return value of *nmore* is defined. If the return value is DAT_SUCCESS, then the return values of *nmore* and *event* are defined.

If this function is called on an EVD in an unwaitable state, or if dat_evd_set_unwaitable(3DAT) is called on an EVD on which a thread is blocked in this function, the function returns with DAT_INVALID_STATE.

The ordering of events dequeued by overlapping calls to dat_evd_wait() or dat_evd_dequeue() is not specified.

Return Values

DAT_SUCCESS	The operation was successful. An event was returned to a Consumer.
DAT_INVALID_HANDLE	The *evd_handle* parameter is invalid.
DAT_INVALID_PARAMETER	The *timeout* or *threshold* parameter is invalid. For example, *threshold* is larger than the EVD's *evd_min_qlen*.
DAT_ABORT	The operation was aborted because IA was closed or EVD was destroyed
DAT_INVALID_STATE	One of the parameters was invalid for this operation. There is already a waiter on the EVD, or the EVD is in an unwaitable state.
DAT_TIMEOUT_EXPIRED	The operation timed out.
DAT_INTERRUPTED_CALL	The operation was interrupted by a signal.

Usage

Consumers should be cautioned against using threshold combined with infinite *timeout*.

Consumers should not mix different models for control of unblocking a waiter. If the Consumer uses Notification Suppression or Solicited Wait to control the Notification events for unblocking a waiter, the *threshold* must be set to 1. If the Consumer uses *threshold* to control when a waiter is unblocked, DAT_COMPLETION_UNSIGNALLED_FLAG locally and DAT_COMPLETION_SOLICITED_WAIT remotely shall not be used. By default, all completions are Notification events.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Safe

See Also dat_cr_accept(3DAT), dat_cr_handoff(3DAT), dat_cr_reject(3DAT), dat_ep_connect(3DAT), dat_ep_dup_connect(3DAT),dat_ep_free(3DAT), dat_evd_dequeue(3DAT), dat_evd_set_unwaitable(3DAT), libdat(3LIB), attributes(5)

Name dat_get_consumer_context – get Consumer context

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_get_consumer_context (
    IN    DAT_HANDLE    dat_handle,
    OUT   DAT_CONTEXT   *context
    )
```

Parameters

dat_handle Handle for a DAT Object associated with *context*.

context Pointer to Consumer-allocated storage where the current value of the *dat_handle* context will be stored.

Description The `dat_get_consumer_context()` function gets the Consumer context from the specified *dat_handle*. The *dat_handle* can be one of the following handle types: `DAT_IA_HANDLE`, `DAT_EP_HANDLE`, `DAT_EVD_HANDLE`, `DAT_CR_HANDLE`, `DAT_RSP_HANDLE`, `DAT_PSP_HANDLE`, `DAT_PZ_HANDLE`, `DAT_LMR_HANDLE`, `DAT_RMR_HANDLE`, or `DAT_CNO_HANDLE`.

Return Values

`DAT_SUCCESS` The operation was successful. The Consumer context was successfully retrieved from the specified handle.

`DAT_INVALID_HANDLE` The *dat_handle* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Unsafe

See Also `dat_set_consumer_context(3DAT)`, `libdat(3LIB)`, `attributes(5)`

Name dat_get_handle_type – get handle type

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_get_handle_typet (
    IN    DAT_HANDLE        dat_handle,
    OUT   DAT_HANDLE_TYPE   *handle_type
    )
```

Parameters

dat_handle Handle for a DAT Object.

handle_type Type of the handle of *dat_handle*.

Description The `dat_get_handle_type()` function allows the Consumer to discover the type of a DAT Object using its handle.

The *dat_handle* can be one of the following handle types: `DAT_IA_HANDLE`, `DAT_EP_HANDLE`, `DAT_EVD_HANDLE`, `DAT_CR_HANDLE`, `DAT_RSP_HANDLE`, `DAT_PSP_HANDLE`, `DAT_PZ_HANDLE`, `DAT_LMR_HANDLE`, or `DAT_RMR_HANDLE`.

The *handle_type* is one of the following handle types: `DAT_HANDLE_TYPE_IA`, `DAT_HANDLE_TYPE_EP`, `DAT_HANDLE_TYPE_EVD`, `DAT_HANDLE_TYPE_CR`, `DAT_HANDLE_TYPE_PSP`, `DAT_HANDLE_TYPE_RSP`, `DAT_HANDLE_TYPE_PZ`, `DAT_HANDLE_TYPE_LMR`, `DAT_HANDLE_TYPE_RMR`, or `DAT_HANDLE_TYPE_CNO`.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *dat_handle* parameter is invalid.

Usage Consumers can use this operation to determine the type of Object being returned. This is needed for calling an appropriate query or any other operation on the Object handle.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Unsafe

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_ia_close – close an IA

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ia_close (
    IN    DAT_IA_HANDLE     ia_handle,
    IN    DAT_CLOSE_FLAGS   ia_flags
    )
```

Parameters

ia_handle Handle for an instance of a DAT IA.

ia_flags Flags for IA closure. Flag definitions are:

`DAT_CLOSE_ABRUPT_FLAG`	Abrupt close. Abrupt cascading close of IA including all Consumer created DAT objects.
`DAT_CLOSE_GRACEFUL_FLAG`	Graceful close. Closure is successful only if all DAT objects created by the Consumer have been freed before the graceful closure call.

Default value of `DAT_CLOSE_DEFAULT` = `DAT_CLOSE_ABRUPT_FLAG` represents abrupt closure of IA.

Description The `dat_ia_close()` function closes an IA (destroys an instance of the Interface Adapter).

The *ia_flags* specify whether the Consumer wants abrupt or graceful close.

The abrupt close does a phased, cascading destroy. All DAT Objects associated with an IA instance are destroyed. These include all the connection oriented Objects: public and reserved Service Points; Endpoints, Connection Requests, LMRs (including lmr_contexts), RMRs (including rmr_contexts), Event Dispatchers, CNOs, and Protection Zones. All the waiters on all CNOs, including the OS Wait Proxy Agents, are unblocked with the DAT_HANDLE_NULL handle returns for an unblocking EVD. All direct waiters on all EVDs are also unblocked and return with DAT_ABORT.

The graceful close does a destroy only if the Consumer has done a cleanup of all DAT objects created by the Consumer with the exception of the asynchronous EVD. Otherwise, the operation does not destroy the IA instance and returns the `DAT_INVALID_STATE`.

If async EVD was created as part of the of `dat_ia_open(3DAT)`, `dat_ia_close()` must destroy it. If *async_evd_handle* was passed in by the Consumer at `dat_ia_open()`, this handle is not destroyed. This is applicable to both abrupt and graceful *ia_flags* values.

Because the Consumer did not create async EVD explicitly, the Consumer does not need to destroy it for graceful close to succeed.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INSUFFICIENT_RESOURCES`	The operation failed due to resource limitations. This is a catastrophic error.
`DAT_INVALID_HANDLE`	Invalid DAT handle; *ia_handle* is invalid.
`DAT_INVALID_PARAMETER`	Invalid parameter; *ia_flags* is invalid.
`DAT_INVALID_STATE`	Parameter in an invalid state. IA instance has Consumer-created objects associated with it.

Usage

The `dat_ia_close()` function is the root cleanup method for the Provider, and, thus, all Objects.

Consumers are advised to explicitly destroy all Objects they created prior to closing the IA instance, but can use this function to clean up everything associated with an open instance of IA. This allows the Consumer to clean up in case of errors.

Note that an abrupt close implies destruction of EVDs and CNOs. Just as with explicit destruction of an EVD or CNO, the Consumer should take care to avoid a race condition where a Consumer ends up attempting to wait on an EVD or CNO that has just been deleted.

The techniques described in `dat_cno_free(3DAT)` and `dat_evd_free(3DAT)` can be used for these purposes.

If the Consumer desires to shut down the IA as quickly as possible, the Consumer can call dat_ia_close(abrupt) without unblocking CNO and EVD waiters in an orderly fashion. There is a slight chance that an invalidated DAT handle will cause a memory fault for a waiter. But this might be an acceptable behavior, especially if the Consumer is shutting down the process.

No provision is made for blocking on event completion or pulling events from queues.

This is the general cleanup and last resort method for Consumer recovery. An implementation must provide for successful completion under all conditions, avoiding hidden resource leakage (dangling memory, zombie processes, and so on) eventually leading to a reboot of the operating system.

The `dat_ia_close()` function deletes all Objects that were created using the IA handle.

The `dat_ia_close()` function can decrement a reference count for the Provider Library that is incremented by `dat_ia_open()` to ensure that the Provider Library cannot be removed when it is in use by a DAT Consumer.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Unsafe

See Also dat_cno_free(3DAT), dat_evd_free(3DAT), dat_ia_open(3DAT), libdat(3LIB), attributes(5)

Name dat_ia_open – open an Interface Adapter (IA)

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ia_open (
    IN const DAT_NAME_PTR       ia_name_ptr,
    IN       DAT_COUNT          async_evd_min_qlen,
    INOUT    DAT_EVD_HANDLE     *async_evd_handle,
    OUT      DAT_IA_HANDLE      *ia_handle
    )
```

Parameters

ia_name_ptr — Symbolic name for the IA to be opened. The name should be defined by the Provider registration.

If the name is prefixed by the string `RO_AWARE_`, then the prefix is removed prior to being passed down and the existence of the prefix indicates that the application has been coded to correctly deal with relaxed ordering constraints. If the prefix is not present and the platform on which the application is running is utilizing relaxed ordering, the open will fail with `DAT_INVALID_PARAMETER` (with `DAT_SUBTYPE_STATUS` of `DAT_INVALID_RO_COOKIE`). This setting also affects `dat_lmr_create(3DAT)`.

async_evd_min_qlen — Minimum length of the Asynchronous Event Dispatcher queue.

async_evd_handle — Pointer to a handle for an Event Dispatcher for asynchronous events generated by the IA. This parameter can be `DAT_EVD_ASYNC_EXISTS` to indicate that there is already EVD for asynchronous events for this Interface Adapter or `DAT_HANDLE_NULL` for a Provider to generate EVD for it.

ia_handle — Handle for an open instance of a DAT IA. This handle is used with other functions to specify a particular instance of the IA.

Description The `dat_ia_open()` function opens an IA by creating an IA instance. Multiple instances (opens) of an IA can exist.

The value of `DAT_HANDLE_NULL` for *async_evd_handle* (**async_evd_handle* == `DAT_HANDLE_NULL`) indicates that the default Event Dispatcher is created with the requested *async_evd_min_qlen*. The *async_evd_handle* returns the handle of the created Asynchronous Event Dispatcher. The first Consumer that opens an IA must use `DAT_HANDLE_NULL` because no EVD can yet exist for the requested *ia_name_ptr*.

The Asynchronous Event Dispatcher (*async_evd_handle*) is created with no CNO (DAT_HANDLE_NULL). Consumers can change these values using dat_evd_modify_cno(3DAT). The Consumer can modify parameters of the Event Dispatcher using dat_evd_resize(3DAT) and dat_evd_modify_cno().

The Provider is required to provide a queue size at least equal to *async_evd_min_qlen*, but is free to provide a larger queue size or dynamically enlarge the queue when needed. The Consumer can determine the actual queue size by querying the created Event Dispatcher instance.

If *async_evd_handle* is not DAT_HANDLE_NULL, the Provider does not create an Event Dispatcher for an asynchronous event and the Provider ignores the *async_evd_min_qlen* value. The *async_evd_handle* value passed in by the Consumer must be an asynchronous Event Dispatcher created for the same Provider (*ia_name_ptr*). The Provider does not have to check for the validity of the Consumer passed in *async_evd_handle*. It is the Consumer responsibility to guarantee that *async_evd_handle* is valid and for this Provider. How the *async_evd_handle* is passed between DAT Consumers is out of scope of the DAT specification. If the Provider determines that the Consumer-provided *async_evd_handle* is invalid, the operation fails and returns DAT_INVALID_HANDLE. The *async_evd_handle* remains unchanged, so the returned *async_evd_handle* is the same the Consumer passed in. All asynchronous notifications for the open instance of the IA are directed by the Provider to the Consumer passed in Asynchronous Event Dispatcher specified by *async_evd_handle*.

Consumer can specify the value of DAT_EVD_ASYNC_EXISTS to indicate that there exists an event dispatcher somewhere else on the host, in user or kernel space, for asynchronous event notifications. It is up to the Consumer to ensure that this event dispatcher is unique and unambiguous. A special handle may be returned for the Asynchronous Event Dispatcher for this scenario, DAT_EVD_OUT_OF_SCOPE, to indicate that there is a default Event Dispatcher assigned for this Interface Adapter, but that it is not in a scope where this Consumer may directly invoke it.

The Asynchronous Event Dispatcher is an Object of both the Provider and IA. Each Asynchronous Event Dispatcher bound to an IA instance is notified of all asynchronous events, such that binding multiple Asynchronous Event Dispatchers degrades performance by duplicating asynchronous event notifications for all Asynchronous Event Dispatchers. Also, transport and memory resources can be consumed per Event Dispatcher bound to an IA

As with all Event Dispatchers, the Consumer is responsible for synchronizing access to the event queue.

Valid IA names are obtained from dat_registry_list_providers(3DAT).

Return Values

DAT_SUCCESS	The operation was successful.
DAT_INSUFFICIENT_RESOURCES	The operation failed due to resource limitations.

DAT_INVALID_PARAMETER	Invalid parameter.
DAT_PROVIDER_NOT_FOUND	The specified provider was not registered in the registry.
DAT_INVALID_HANDLE	Invalid DAT handle; async_evd_handle is invalid.

Usage The dat_ia_open() function is the root method for the Provider, and, thus, all Objects. It is the root handle through which the Consumer obtains all other DAT handles. When the Consumer closes its handle, all its DAT Objects are released.

The dat_ia_open() function is the workhorse method that provides an IA instance. It can also initialize the Provider library or do any other registry-specific functions.

The dat_ia_open() function creates a unique handle for the IA to the Consumer. All further DAT Objects created for this Consumer reference this handle as their owner.

The dat_ia_open() function can use a reference count for the Provider Library to ensure that the Provider Library cannot be removed when it is in use by a DAT Consumer.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	uDAPL, 1.1, 1.2 (except RO_AWARE_)

See Also dat_evd_modify_cno(3DAT), dat_evd_resize(3DAT), dat_ia_close(3DAT), dat_registry_list_providers(3DAT), libdat(3LIB), attributes(5)

Name dat_ia_query – query an IA

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ia_query (
    IN    DAT_IA_HANDLE           ia_handle,
    OUT   DAT_EVD_HANDLE          *async_evd_handle,
    IN    DAT_IA_ATTR_MASK        ia_attr_mask,
    OUT   DAT_IA_ATTR             *ia_attributes,
    IN    DAT_PROVIDER_ATTR_MASK  provider_attr_mask,
    OUT   DAT_PROVIDER_ATTR       *provider_attributes
    )
```

Parameters

ia_handle	Handle for an open instance of an IA.
async_evd_handle	Handle for an Event Dispatcher for asynchronous events generated by the IA.
ia_attr_mask	Mask for the *ia_attributes*.
ia_attributes	Pointer to a Consumer-allocated structure that the Provider fills with IA attributes.
provider_attr_mask	Mask for the *provider_attributes*.
provider_attributes	Pointer to a Consumer-allocated structure that the Provider fills with Provider attributes.

Description The `dat_ia_query()` functions provides the Consumer with the IA parameters, as well as the IA and Provider attributes. Consumers pass in pointers to Consumer-allocated structures for the IA and Provider attributes that the Provider fills.

The *ia_attr_mask* and *provider_attr_mask* parameters allow the Consumer to specify which attributes to query. The Provider returns values for requested attributes. The Provider can also return values for any of the other attributes.

Interface Adapter Attributes

The IA attributes are common to all open instances of the IA. DAT defines a method to query the IA attributes but does not define a method to modify them.

If IA is multiported, each port is presented to a Consumer as a separate IA.

Adapter name:	The name of the IA controlled by the Provider. The same as *ia_name_ptr*.
Vendor name:	Vendor if IA hardware.
HW version major:	Major version of IA hardware.

HW version minor:	Minor version of IA hardware.
Firmware version major:	Major version of IA firmware.
Firmware version minor:	Minor version of IA firmware.
IA_address_ptr:	An address of the interface Adapter.
Max EPs:	Maximum number of Endpoints that the IA can support. This covers all Endpoints in all states, including the ones used by the Providers, zero or more applications, and management.
Max DTOs per EP:	Maximum number of DTOs and RMR_binds that any Endpoint can support for a single direction. This means the maximum number of outstanding and in-progress Send, RDMA Read, RDMA Write DTOs, and RMR Binds at any one time for any Endpoint; and maximum number of outstanding and in-progress Receive DTOs at any one time for any Endpoint.
Max incoming RDMA Reads per EP:	Maximum number of RDMA Reads that can be outstanding per (connected) Endpoint with the IA as the target.
Max outgoing RDMA Reads per EP:	Maximum number of RDMA Reads that can be outstanding per (connected) Endpoint with the IA as the originator.
Max EVDs:	Maximum number of Event Dispatchers that an IA can support. An IA cannot support an Event Dispatcher directly, but indirectly by Transport-specific Objects, for example, Completion Queues for Infiniband™ and VI. The Event Dispatcher Objects can be shared among multiple Providers and similar Objects from other APIs, for example, Event Queues for uDAPL.
Max EVD queue size:	Maximum size of the EVD queue supported by an IA.
Max IOV segments per DTO:	Maximum entries in an IOV list that an IA supports. Notice that this number cannot be explicit but must be implicit to transport-specific Object entries. For example, for IB, it is the maximum number of

	scatter/gather entries per Work Request, and for VI it is the maximum number of data segments per VI Descriptor.
Max LMRs:	Maximum number of Local Memory Regions IA supports among all Providers and applications of this IA.
Max LMR block size:	Maximum contiguous block that can be registered by the IA.
Mac LMR VA:	Highest valid virtual address within the context of an LMR. Frequently, IAs on 32–bit architectures support only 32–bit local virtual addresses.
Max PZs:	Maximum number of Protection Zones that the IA supports.
Max MTU size:	Maximum message size supported by the IA
Max RDMA size:	Maximum RDMA size supported by the IA
Max RMRs:	Maximum number of RMRs an IA supports among all Providers and applications of this IA.
Max RMR target address:	Highest valid target address with the context of a local RMR. Frequently, IAs on 32–bit architectures support only 32–bit local virtual addresses.
Num transport attributes:	Number of transport-specific attributes.
Transport-specific attributes:	Array of transport-specific attributes. Each entry has the format of `DAT_NAMED_ATTR`, which is a structure with two elements. The first element is the name of the attribute. The second element is the value of the attribute as a string.
Num vendor attributes:	Number of vendor-specific attributes.
Vendor-specific attributes:	Array of vendor-specific attributes. Each entry has the format of `DAT_NAMED_ATTR`, which is a structure with two elements. The first element is the name of the attribute. The second element is the value of the attribute as a string.

DAPL Provider Attributes

The provider attributes are specific to the open instance of the IA. DAT defines a method to query Provider attributes but does not define a method to modify them.

Provider name:	Name of the Provider vendor.
Provider version major:	Major Version of uDAPL Provider.

Provider version minor:	Minor Version of uDAPL Provider.
DAPL API version major:	Major Version of uDAPL API supported.
DAPL API version minor:	Minor Version of uDAPL API supported.
LMR memory types supported:	Memory types that LMR Create supports for memory registration. This value is a union of LMR Memory Types `DAT_MEM_TYPE_VIRTUAL`, `DAT_MEM_TYPE_LMR`, and `DAT_MEM_TYPE_SHARED_VIRTUAL` that the Provider supports. All Providers must support the following Memory Types: `DAT_MEM_TYPE_VIRTUAL`, `DAT_MEM_TYPE_LMR`, and `DAT_MEM_TYPE_SHARED_VIRTUAL`.
IOV ownership:	An enumeration flag that specifies the ownership of the local buffer description (IOV list) after post DTO returns. The three values are as follows: ■ `DAT_IOV_CONSUMER` indicates that the Consumer has the ownership of the local buffer description after a post returns. ■ `DAT_IOV_PROVIDER_NOMOD` indicates that the Provider still has ownership of the local buffer description of the DTO when the post DTO returns, but the Provider does not modify the buffer description. ■ `DAT_IOV_PROVIDER_MOD` indicates that the Provider still has ownership of the local buffer description of the DTO when the post DTO returns and can modify the buffer description. In any case, the Consumer obtains ownership of the local buffer description after the DTO transfer is completed and the Consumer is notified through a DTO completion event.
QOS supported:	The union of the connection QOS supported by the Provider.
Completion flags supported:	The following values for the completion flag `DAT_COMPLETION_FLAGS` are supported by the Provider: `DAT_COMPLETION_SUPPRESS_FLAG`, `DAT_COMPLETION_UNSIGNALLED_FLAG`, `DAT_COMPLETION_SOLICITED_WAIT_FLAG`, and `DAT_COMPLETION_BARRIER_FENCE_FLAG`.
Thread safety:	Provider Library thread safe or not. The Provider Library is not required to be thread safe.
Max private data size:	Maximum size of private data the Provider supports. This value is at least 64 bytes.

Multipathing support:	Capability of the Provider to support Multipathing for connection establishment.
EP creator for PSP:	Indicator for who can create an Endpoint for a Connection Request. For the Consumer it is `DAT_PSP_CREATES_EP_NEVER`. For the Provider it is `DAT_PSP_CREATES_EP_ALWAYS`. For both it is `DAT_PSP_CREATES_EP_IFASKED`. This attribute is used for Public Service Point creation.
PZ support:	Indicator of what kind of protection the Provider's PZ provides.
Optimal Buffer Alignment:	Local and remote DTO buffer alignment for optimal performance on the Platform. The `DAT_OPTIMAL_ALIGMNEMT` must be divisible by this attribute value. The maximum allowed value is `DAT_OPTIMAL_ALIGMNEMT`, or 256.
EVD stream merging support:	A 2D binary matrix where each row and column represent an event stream type. Each binary entry is 1 if the event streams of its row and column can be fed to the same EVD, and 0 otherwise. More than two different event stream types can feed the same EVD if for each pair of the event stream types the entry is 1. The Provider should support merging of all event stream types. The Consumer should check this attribute before requesting an EVD that merges multiple event stream types.
Num provider attributes:	Number of Provider-specific attributes.
Provider-specific attributes:	Array of Provider-specific attributes. Each entry has the format of `DAT_NAMED_ATTR`, which is a structure with two elements. The first element is the name of the attribute. The second element is the value of the attribute as a string.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INVALID_PARAMETER`	Invalid parameter;
`DAT_INVALID_HANDLE`	Invalid DAT handle; ia_handle is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Safe

See Also libdat(3LIB), attributes(5)

Name dat_lmr_create – register a memory region with an IA

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_lmr_create (
    IN    DAT_IA_HANDLE          ia_handle,
    IN    DAT_MEM_TYPE           mem_type,
    IN    DAT_REGION_DESCRIPTION region_description,
    IN    DAT_VLEN               length,
    IN    DAT_PZ_HANDLE          pz_handle,
    IN    DAT_MEM_PRIV_FLAGS     mem_privileges,
    OUT   DAT_LMR_HANDLE         *lmr_handle,
    OUT   DAT_LMR_CONTEXT        *lmr_context,
    OUT   DAT_RMR_CONTEXT        *rmr_context,
    OUT   DAT_VLEN               *registered_size,
    OUT   DAT_VADDR              *registered_address
    )
```

Parameters

ia_handle

Handle for an open instance of the IA.

mem_type

Type of memory to be registered. The following list outlines the memory type specifications.

`DAT_MEM_TYPE_VIRTUAL`

Consumer virtual memory.

Region description: A pointer to a contiguous user virtual range.

Length: Length of the Memory Region.

`DAT_MEM_TYPE_SO_VIRTUAL`

Consumer virtual memory with strong memory ordering. This type is a Solaris specific addition. If the *ia_handle* was opened without `RO_AWARE_` (see `dat_ia_open(3DAT)`), then type `DAT_MEM_TYPE_VIRTUAL` is implicitly converted to this type.

Region description: A pointer to a contiguous user virtual range.

Length: Length of the Memory Region.

`DAT_MEM_TYPE_LMR`

LMR.

Region description: An LMR_handle.

Length: Length parameter is ignored.

`DAT_MEM_TYPE_SHARED_VIRTUAL`
Shared memory region. All DAT Consumers of the same uDAPL Provider specify the same Consumer cookie to indicate who is sharing the shared memory region. This supports a peer-to-peer model of shared memory. All DAT Consumers of the shared memory must allocate the memory region as shared memory using Platform-specific primitives.

Region description: A structure with 2 elements, where the first one is of type `DAT_LMR_COOKIE` and is a unique identifier of the shared memory region, and the second one is a pointer to a contiguous user virtual range.

Length: Length of the Memory Region

region_description
Pointer to type-specific data describing the memory in the region to be registered. The type is derived from the *mem_type* parameter.

length
Length parameter accompanying the region_description.

pz_handle
Handle for an instance of the Protection Zone.

mem_privileges:
Consumer-requested memory access privileges for the registered local memory region. The Default value is `DAT_MEM_PRIV_NONE_FLAG`. The constant value `DAT_MEM_PRIV_ALL_FLAG` = 0x33, which specifies both Read and Write privileges, is also defined. Memory privilege definitions are as follows:

Local Read	`DAT_MEM_PRIV_LOCAL_READ_FLAG`	
	0x01	Local read access requested.
Local Write	`DAT_MEM_PRIV_LOCAL_WRITE_FLAG`	
	0x10	Local write access requested.
Remote Read	`DAT_MEM_PRIV_REMOTE_READ_FLAG`	
	0x02	Remote read access requested.
Remote Write	`DAT_MEM_PRIV_REMOTE_WRITE_FLAG`	
	0x20	Remote write access requested.

lmr_handle
Handle for the created instance of the LMR.

lmr_context
Context for the created instance of the LMR to use for DTO local buffers.

registered_size
Actual memory size registered by the Provider.

registered_address
Actual base address of the memory registered by the Provider.

Description The `dat_lmr_create()` function registers a memory region with an IA. The specified buffer must have been previously allocated and pinned by the uDAPL Consumer on the platform. The Provider must do memory pinning if needed, which includes whatever OS-dependent steps are required to ensure that the memory is available on demand for the Interface Adapter. uDAPL does not require that the memory never be swapped out; just that neither the hardware nor the Consumer ever has to deal with it not being there. The created *lmr_context* can be used for local buffers of DTOs and for binding RMRs, and *lmr_handle* can be used for creating other LMRs. For uDAPL the scope of the *lmr_context* is the address space of the DAT Consumer.

The return values of *registered_size* and *registered_address* indicate to the Consumer how much the contiguous region of Consumer virtual memory was registered by the Provider and where the region starts in the Consumer virtual address.

The *mem_type* parameter indicates to the Provider the kind of memory to be registered, and can take on any of the values defined in the table in the PARAMETERS section.

The *pz_handle* parameter allows Consumers to restrict local accesses to the registered LMR by DTOs.

`DAT_LMR_COOKIE` is a pointer to a unique identifier of the shared memory region of the `DAT_MEM_TYPE_SHARED_VIRTUAL` DAT memory type. The identifier is an array of 40 bytes allocated by the Consumer. The Provider must check the entire 40 bytes and shall not interpret it as a null-terminated string.

The return value of *rmr_context* can be transferred by the local Consumer to a Consumer on a remote host to be used for an RDMA DTO.

If *mem_privileges* does not specify remote Read and Write privileges, *rmr_context* is not generated and `NULL` is returned. No remote privileges are given for Memory Region unless explicitly asked for by the Consumer.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_UNSUFFICIENT_RESOURCES`	The operation failed due to resource limitations.
`DAT_INVALID_PARAMETER`	Invalid parameter.
`DAT_INVALID_HANDLE`	Invalid DAT handle.
`DAT_INVALID_STATE`	Parameter in an invalid state. For example, shared virtual buffer was not created shared by the platform.

DAT_MODEL_NOT_SUPPORTED	The requested Model was not supported by the Provider. For example, requested Memory Type was not supported by the Provider.

Usage Consumers can create an LMR over the existing LMR memory with different Protection Zones and privileges using previously created IA translation table entries.

The Consumer should use *rmr_context* with caution. Once advertised to a remote peer, the *rmr_context* of the LMR cannot be invalidated. The only way to invalidate it is to destroy the LMR with dat_lmr_free(3DAT).

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	uDAPL, 1.1, 1.2 (except DAT_MEM_TYPE_SO_VIRTUAL)

See Also dat_lmr_free(3DAT), libdat(3LIB), attributes(5)

Name dat_lmr_free – destroy an instance of the LMR

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_lmr_free (
    IN    DAT_LMR_HANDLE    lmr_handle
    )
```

Parameters *lmr_handle:* Handle for an instance of LMR to be destroyed.

Description The `dat_lmr_free()` function destroys an instance of the LMR. The LMR cannot be destroyed if it is in use by an RMR. The operation does not deallocate the memory region or unpin memory on a host.

Use of the handle of the destroyed LMR in any subsequent operation except for `dat_lmr_free()` fails. Any DTO operation that uses the destroyed LMR after the `dat_lmr_free()` is completed shall fail and report a protection violation. The use of *rmr_context* of the destroyed LMR by a remote peer for an RDMA DTO results in an error and broken connection on which it was used. Any remote RDMA operation that uses the destroyed LMR rmr_context, whose Transport-specific request arrived to the local host after the `dat_lmr_free()` has completed, fails and reports a protection violation. Remote RDMA operation that uses the destroyed LMR *rmr_context*, whose Transport-specific request arrived to the local host prior to the `dat_lmr_free()` returns, might or might not complete successfully. If it fails, `DAT_DTO_ERR_REMOTE_ACCESS` is reported in `DAT_DTO_COMPLETION_STATUS` for the remote RDMA DTO and the connection is broken.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INVALID_HANDLE`	The *lmr_handle* parameter is invalid.
`DAT_INVALID_STATE`	Parameter in an invalid state; LMR is in use by an RMR instance.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Unsafe

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_lmr_query – provide LMR parameters

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_lmr_query (
    IN    DAT_LMR_HANDLE        lmr_handle,
    IN    DAT_LMR_PARAM_MASK    lmr_param_mask,
    OUT   DAT_LMR_PARAM         *lmr_param
    )
```

Parameters

lmr_handle	Handle for an instance of the LMR.
lmr_param_mask	Mask for LMR parameters.
lmr_param	Pointer to a Consumer-allocated structure that the Provider fills with LMR parameters.

Description The `dat_lmr_query()` function provides the Consumer LMR parameters. The Consumer passes in a pointer to the Consumer-allocated structures for LMR parameters that the Provider fills.

The *lmr_param_mask* parameter allows Consumers to specify which parameters to query. The Provider returns values for *lmr_param_mask* requested parameters. The Provider can return values for any other parameters.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INVALID_PARAMETER`	The *lmr_param_mask* function is invalid.
`DAT_INVALID_HANDLE`	The *lmr_handle* function is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Unsafe

See Also `libdat(3LIB)`, `attributes(5)`

REFERENCE

Extended Library Functions - Part 2

Name dat_lmr_sync_rdma_read – synchronize local memory with RDMA read on non-coherent memory

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_lmr_sync_rdma_read (
    IN DAT_IA_HANDLE ia_handle,
    IN const DAT_LMR_TRIPLET *local_segments,
    IN DAT_VLEN num_segments
    )
```

Parameters

ia_handle	A handle for an open instance of the IA.
local_segments	An array of buffer segments.
num_segments	The number of segments in the *local_segments* argument.

Description The `dat_lmr_sync_rdma_read()` function makes memory changes visible to an incoming RDMA Read operation. This operation guarantees consistency by locally flushing the non-coherent cache prior to it being retrieved by remote peer RDMA read operations.

The `dat_lmr_sync_rdma_read()` function is needed if and only if the Provider attribute specifies that this operation is needed prior to an incoming RDMA Read operation. The Consumer must call `dat_lmr_sync_rdma_read()` after modifying data in a memory range in this region that will be the target of an incoming RDMA Read operation. The `dat_lmr_sync_rdma_read()` function must be called after the Consumer has modified the memory range but before the RDMA Read operation begins. The memory range that will be accessed by the RDMA read operation must be supplied by the caller in the *local_segments* array. After this call returns, the RDMA Read operation can safely see the modified contents of the memory range. It is permissible to batch synchronizations for multiple RDMA Read operations in a single call by passing a *local_segments* array that includes all modified memory ranges. The *local_segments* entries need not contain the same LMR and need not be in the same Protection Zone.

If the Provider attribute specifying that this operation is required attempts to read from a memory range that is not properly synchronized using `dat_lmr_sync_rdma_read()`, the returned contents are undefined.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INVALID_HANDLE`	The DAT handle is invalid.
`DAT_INVALID_PARAMETER`	One of the parameters is invalid. For example, the address range for a local segment fell outside the boundaries of the corresponding Local Memory Region or the LMR handle was invalid.

Usage Determining when an RDMA Read will start and what memory range it will read is the Consumer's responsibility. One possibility is to have the Consumer that is modifying memory call `dat_lmr_sync_rdma_read()` and then post a Send DTO message that identifies the range in the body of the Send. The Consumer wanting to perform the RDMA Read can receive this message and know when it is safe to initiate the RDMA Read operation.

This call ensures that the Provider receives a coherent view of the buffer contents upon a subsequent remote RDMA Read operation. After the call completes, the Consumer can be assured that all platform-specific buffer and cache updates have been performed, and that the LMR range has consistency with the Provider hardware. Any subsequent write by the Consumer can void this consistency. The Provider is not required to detect such access.

The action performed on the cache before the RDMA Read depends on the cache type:

- I/O noncoherent cache will be invalidated.
- CPU noncoherent cache will be flushed.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.2
MT-Level	Unsafe

See Also `dat_lmr_sync_rdma_write(3DAT)`, `libdat(3LIB)`, `attributes(5)`

Name dat_lmr_sync_rdma_write – synchronize local memory with RDMA write on non-coherent memory

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_lmr_sync_rdma_write (
    IN DAT_IA_HANDLE ia_handle,
    IN const DAT_LMR_TRIPLET *local_segments,
    IN DAT_VLEN num_segments
    )
```

Parameters

ia_handle A handle for an open instance of the IA.

local_segments An array of buffer segments.

num_segments The number of segments in the *local_segments* argument.

Description The `dat_lmr_sync_rdma_write()` function makes effects of an incoming RDMA Write operation visible to the Consumer. This operation guarantees consistency by locally invalidating the non-coherent cache whose buffer has been populated by remote peer RDMA write operations.

The `dat_lmr_sync_rdma_write()` function is needed if and only if the Provider attribute specifies that this operation is needed after an incoming RDMA Write operation. The Consumer must call `dat_lmr_sync_rdma_write()` before reading data from a memory range in this region that was the target of an incoming RDMA Write operation. The `dat_lmr_sync_rdma_write()` function must be called after the RDMA Write operation completes, and the memory range that was modified by the RDMA Write must be supplied by the caller in the *local_ segments* array. After this call returns, the Consumer may safely see the modified contents of the memory range. It is permissible to batch synchronizations of multiple RDMA Write operations in a single call by passing a *local_segments* array that includes all modified memory ranges. The *local_segments* entries need not contain the same LMR and need not be in the same Protection Zone.

The Consumer must also use `dat_lmr_sync_rdma_write()` when performing local writes to a memory range that was or will be the target of incoming RDMA writes. After performing the local write, the Consumer must call `dat_lmr_sync_rdma_write()` before the RDMA Write is initiated. Conversely, after an RDMA Write completes, the Consumer must call `dat_lmr_sync_rdma_write()` before performing a local write to the same range.

If the Provider attribute specifies that this operation is needed and the Consumer attempts to read from a memory range in an LMR without properly synchronizing using `dat_lmr_sync_rdma_write()`, the returned contents are undefined. If the Consumer attempts to write to a memory range without properly synchronizing, the contents of the memory range become undefined.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INVALID_HANDLE`	The DAT handle is invalid.
`DAT_INVALID_PARAMETER`	One of the parameters is invalid. For example, the address range for a local segment fell outside the boundaries of the corresponding Local Memory Region or the LMR handle was invalid.

Usage

Determining when an RDMA Write completes and determining which memory range was modified is the Consumer's responsibility. One possibility is for the RDMA Write initiator to post a Send DTO message after each RDMA Write that identifies the range in the body of the Send. The Consumer at the target of the RDMA Write can receive the message and know when and how to call `dat_lmr_sync_rdma_write()`.

This call ensures that the Provider receives a coherent view of the buffer contents after a subsequent remote RDMA Write operation. After the call completes, the Consumer can be assured that all platform-specific buffer and cache updates have been performed, and that the LMR range has consistency with the Provider hardware. Any subsequent read by the Consumer can void this consistency. The Provider is not required to detect such access.

The action performed on the cache before the RDMA Write depends on the cache type:

- I/O noncoherent cache will be flushed.
- CPU noncoherent cache will be invalidated.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.2
MT-Level	Unsafe

See Also

`dat_lmr_sync_rdma_read(3DAT)`, `libdat(3LIB)`, `attributes(5)`

Name dat_provider_fini – disassociate the Provider from a given IA name

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

void
    dat_provider_fini (
    IN    const DAT_PROVIDER_INFO    *provider_info
    )
```

Parameters *provider_info* The information that was provided when dat_provider_init was called.

Description A destructor the Registry calls on a Provider before it disassociates the Provider from a given IA name.

The Provider can use this method to undo any initialization it performed when `dat_provider_init(3DAT)` was called for the same IA name. The Provider's implementation of this method should call `dat_registry_remove_provider(3DAT)` to unregister its IA Name. If it does not, the Registry might remove the entry itself.

This method can be called for a given IA name at any time after all open instances of that IA are closed, and is certainly called before the Registry unloads the Provider library. However, it is not called more than once without an intervening call to `dat_provider_init()` for that IA name.

Return Values No values are returned.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	

See Also `dat_provider_init(3DAT)`, `dat_registry_remove_provider(3DAT)`, `libdat(3LIB)`, `attributes(5)`

Name dat_provider_init – locate the Provider in the Static Registry

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

void
    dat_provider_init (
    IN    const DAT_PROVIDER_INFO    *provider_info,
    IN    const char *               instance_data
    )
```

Parameters

provider_info The information that was provided by the Consumer to locate the Provider in the Static Registry.

instance_data The instance data string obtained from the entry found in the Static Registry for the Provider.

Description A constructor the Registry calls on a Provider before the first call to `dat_ia_open(3DAT)` for a given IA name when the Provider is auto-loaded. An application that explicitly loads a Provider on its own can choose to use `dat_provider_init()` just as the Registry would have done for an auto-loaded Provider.

The Provider's implementation of this method must call `dat_registry_add_provider(3DAT)`, using the IA name in the `provider_info.ia_name` field, to register itself with the Dynamic Registry. The implementation must not register other IA names at this time. Otherwise, the Provider is free to perform any initialization it finds useful within this method.

This method is called before the first call to `dat_ia_open()` for a given IA name after one of the following has occurred:

- The Provider library was loaded into memory.
- The Registry called `dat_provider_fini(3DAT)` for that IA name.
- The Provider called `dat_registry_remove_provider(3DAT)` for that IA name (but it is still the Provider indicated in the Static Registry).

If this method fails, it should ensure that it does not leave its entry in the Dynamic Registry.

Return Values No values are returned.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	

See Also dat_ia_open(3DAT), dat_provider_fini(3DAT), dat_registry_add_provider(3DAT), dat_registry_remove_provider(3DAT), libdat(3LIB), attributes(5)

Name dat_psp_create – create a persistent Public Service Point

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_psp_create(
    IN    DAT_IA_HANDLE     ia_handle,
    IN    DAT_CONN_QUAL     conn_qual,
    IN    DAT_EVD_HANDLE    evd_handle,
    IN    DAT_PSP_FLAGS     psp_flags,
    OUT   DAT_PSP_HANDLE    *psp_handle
    )
```

Parameters

ia_handle Handle for an instance of DAT IA.

conn_qual Connection Qualifier of the IA on which the Public Service Point is listening.

evd_handle Event Dispatcher that provides the Connection Requested Events to the Consumer. The size of the event queue for the Event Dispatcher controls the size of the backlog for the created Public Service Point.

psp_flags Flag that indicates whether the Provider or Consumer creates an Endpoint per arrived Connection Request. The value of `DAT_PSP_PROVIDER` indicates that the Consumer wants to get an Endpoint from the Provider; a value of `DAT_PSP_CONSUMER` means the Consumer does not want the Provider to provide an Endpoint for each arrived Connection Request.

psp_handle Handle to an opaque Public Service Point.

Description The `dat_psp_create()` function creates a persistent Public Service Point that can receive multiple requests for connection and generate multiple Connection Request instances that are delivered through the specified Event Dispatcher in Notification events.

The `dat_psp_create()` function is blocking. When the Public Service Point is created, `DAT_SUCCESS` is returned and *psp_handle* contains a handle to an opaque Public Service Point Object.

There is no explicit backlog for a Public Service Point. Instead, Consumers can control the size of backlog through the queue size of the associated Event Dispatcher.

The *psp_flags* parameter allows Consumers to request that the Provider create an implicit Endpoint for each incoming Connection Request, or request that the Provider should not create one per Connection Request. If the Provider cannot satisfy the request, the operation shall fail and `DAT_MODEL_NOT_SUPPORTED` is returned.

All Endpoints created by the Provider have `DAT_HANDLE_NULL` for the Protection Zone and all Event Dispatchers. The Provider sets up Endpoint attributes to match the Active side connection request. The Consumer can change Endpoint parameters. Consumers should

change Endpoint parameters, especially PZ and EVD, and are advised to change parameters for local accesses prior to the connection request acceptance with the Endpoint.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INSUFFICIENT_RESOURCES`	The operation failed due to resource limitations.
`DAT_INVALID_HANDLE`	The *ia_handle* or *evd_handle* parameter is invalid.
`DAT_INVALID_PARAMETER`	The *conn_qual* or *psp_flags* parameter is invalid.
`DAT_CONN_QUAL_IN_USE`	The specified Connection Qualifier was in use.
`DAT_MODEL_NOT_SUPPORTED`	The requested Model was not supported by the Provider.

Usage

Two uses of a Public Service Point are as follows:

Model 1 For this model, the Provider manipulates a pool of Endpoints for a Public Service Point. The Provider can use the same pool for more than one Public Service Point.

- The DAT Consumer creates a Public Service Point with a *flag* set to `DAT_PSP_PROVIDER`.
- The Public Service Point does the following:
  - Collects native transport information reflecting a received Connection Reques
  - Creates an instance of Connection Reques
  - Creates a Connection Request Notice (event) that includes the Connection Request instance (thatwhich includes, among others, Public Service Point, its Connection Qualifier, Provider-generated Local Endpoint, and information about remote Endpoint)
  - Delivers the Connection Request Notice to the Consumer-specified target (CNO) *evd_handle*

    The Public Service Point is persistent and continues to listen for incoming requests for connection.
- Upon receiving a connection request, or at some time subsequent to that, the DAT Consumer can modify the provided local Endpoint to match the Connection Request and must either `accept()` or `reject()` the pending Connection Request.
- If accepted, the provided Local Endpoint is now in a "connected" state and is fully usable for this connection, pending only any native transport mandated RTU (ready-to-use) messages. This includes binding it to the IA port if that was not done previously. The Consumer is notified that the Endpoint is in Connected state by a Connection Established Event on the Endpoint *connect_evd_handle*.

- If rejected, control of the Local Endpoint point is returned back to the Provider and its *ep_handle* is no longer usable by the Consumer.

Model 2 For this model, the Consumer manipulates a pool of Endpoints. Consumers can use the same pool for more than one Service Point.

- DAT Consumer creates a Public Service Point with a *flag* set to `DAT_PSP_CONSUMER`.
- Public Service Point:
  - Collects native transport information reflecting a received Connection Request
  - Creates an instance of Connection Request
  - Creates a Connection Request Notice (event) that includes the Connection Request instance (which includes, among others, Public Service Point, its Connection Qualifier, Provider-generated Local Endpoint and information about remote Endpoint)
  - Delivers the Connection Request Notice to the Consumer-specified target (CNO) *evd_handle*

    The Public Service Point is persistent and continues to listen for incoming requests for connection.
- The Consumer creates a pool of Endpoints that it uses for accepting Connection Requests. Endpoints can be created and modified at any time prior to accepting a Connection Request with that Endpoint.
- Upon receiving a connection request or at some time subsequent to that, the DAT Consumer can modify its local Endpoint to match the Connection Request and must either `accept()` or `reject()` the pending Connection Request.
- If accepted, the provided Local Endpoint is now in a "connected" state and is fully usable for this connection, pending only any native transport mandated RTU messages. This includes binding it to the IA port if that was not done previously. The Consumer is notified that the Endpoint is in Connected state by a Connection Established Event on the Endpoint *connect_evd_handle*.
- If rejected, the Consumer does not have to provide any Endpoint for `dat_cr_reject(3DAT)`.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also dat_cr_reject(3DAT), libdat(3LIB), attributes(5)

Name dat_psp_create_any – create a persistent Public Service Point

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_psp_create_any(
    IN    DAT_IA_HANDLE     ia_handle,
    IN    DAT_CONN_QUAL     conn_qual,
    IN    DAT_EVD_HANDLE    evd_handle,
    IN    DAT_PSP_FLAGS     psp_flags,
    OUT   DAT_PSP_HANDLE    *psp_handle
    )
```

Parameters

ia_handle Handle for an instance of DAT IA.

conn_qual Connection Qualifier of the IA on which the Public Service Point is listening.

evd_handle Event Dispatcher that provides the Connection Requested Events to the Consumer. The size of the event queue for the Event Dispatcher controls the size of the backlog for the created Public Service Point.

psp_flags Flag that indicates whether the Provider or Consumer creates an Endpoint per arrived Connection Request. The value of `DAT_PSP_PROVIDER` indicates that the Consumer wants to get an Endpoint from the Provider; a value of `DAT_PSP_CONSUMER` means the Consumer does not want the Provider to provide an Endpoint for each arrived Connection Request.

psp_handle Handle to an opaque Public Service Point.

Description The `dat_psp_create_any()` function creates a persistent Public Service Point that can receive multiple requests for connection and generate multiple Connection Request instances that are delivered through the specified Event Dispatcher in Notification events.

The `dat_psp_create_any()` function allocates an unused Connection Qualifier, creates a Public Service point for it, and returns both the allocated Connection Qualifier and the created Public Service Point to the Consumer.

The allocated Connection Qualifier should be chosen from "nonprivileged" ports that are not currently used or reserved by any user or kernel Consumer or host ULP of the IA. The format of allocated Connection Qualifier returned is specific to IA transport type.

The `dat_psp_create_any()` function is blocking. When the Public Service Point is created, `DAT_SUCCESS` is returned, *psp_handle* contains a handle to an opaque Public Service Point Object, and *conn_qual* contains the allocated Connection Qualifier. When return is not `DAT_SUCCESS`, *psp_handle* and *conn_qual* return values are undefined.

There is no explicit backlog for a Public Service Point. Instead, Consumers can control the size of backlog through the queue size of the associated Event Dispatcher.

The *psp_flags* parameter allows Consumers to request that the Provider create an implicit Endpoint for each incoming Connection Request, or request that the Provider should not create one per Connection Request. If the Provider cannot satisfy the request, the operation shall fail and `DAT_MODEL_NOT_SUPPORTED` is returned.

All Endpoints created by the Provider have `DAT_HANDLE_NULL` for the Protection Zone and all Event Dispatchers. The Provider sets up Endpoint attributes to match the Active side connection request. The Consumer can change Endpoint parameters. Consumers should change Endpoint parameters, especially PZ and EVD, and are advised to change parameters for local accesses prior to the connection request acceptance with the Endpoint.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INSUFFICIENT_RESOURCES`	The operation failed due to resource limitations.
`DAT_INVALID_HANDLE`	The *ia_handle* or *evd_handle* parameter is invalid.
`DAT_INVALID_PARAMETER`	The *conn_qual* or *psp_flags* parameter is invalid.
`DAT_CONN_QUAL_UNAVAILABLE`	No Connection Qualifiers available.
`DAT_MODEL_NOT_SUPPORTED`	The requested Model was not supported by the Provider.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Safe

See Also

`libdat(3LIB)`, `attributes(5)`

Name dat_psp_free – destroy an instance of the Public Service Point

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_psp_free (
    IN    DAT_PSP_HANDLE    psp_handle
    )
```

Parameters *psp_handle* Handle for an instance of the Public Service Point.

Description The `dat_psp_free()` function destroys a specified instance of the Public Service Point.

Any incoming Connection Requests for the Connection Qualifier on the destroyed Service Point it had been listening on are automatically rejected by the Provider with the return analogous to the no listening Service Point.

The behavior of the Connection Requests in progress is undefined and left to an implementation. But it must be consistent. This means that either a Connection Requested Event has been generated for the Event Dispatcher associated with the Service Point, including the creation of the Connection Request instance, or the Connection Request is rejected by the Provider without any local notification.

This operation shall have no effect on previously generated Connection Requested Events. This includes Connection Request instances and, potentially, Endpoint instances created by the Provider.

The behavior of this operation with creation of a Service Point on the same Connection Qualifier at the same time is not defined. Consumers are advised to avoid this scenario.

Use of the handle of the destroyed Public Service Point in any consequent operation fails.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *psp_handle* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Unsafe

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_psp_query – provide parameters of the Public Service Point

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_psp_query (
    IN    DAT_PSP_HANDLE        psp_handle,
    IN    DAT_PSP_PARAM_MASK    psp_param_mask,
    OUT   DAT_PSP_PARAM         *psp_param
    )
```

Parameters

psp_handle	Handle for an instance of Public Service Point.
psp_param_mask	Mask for PSP parameters.
psp_param	Pointer to a Consumer-allocated structure that Provider fills for Consumer-requested parameters.

Description The `dat_psp_query()` function provides to the Consumer parameters of the Public Service Point. Consumer passes in a pointer to the Consumer allocated structures for PSP parameters that Provider fills.

The *psp_param_mask* parameter allows Consumers to specify which parameters they would like to query. The Provider will return values for *psp_param_mask* requested parameters. The Provider may return the value for any of the other parameters.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INVALID_HANDLE`	The *psp_handle* parameter is invalid.
`DAT_INVALID_PARAMETER`	The *psp_param_mask* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Safe

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_pz_create – create an instance of the Protection Zone

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_pz_create (
    IN    DAT_IA_HANDLE    ia_handle,
    OUT   DAT_PZ_HANDLE    *pz_handle
    )
```

Parameters

ia_handle Handle for an open instance of the IA.

pz_handle Handle for the created instance of Protection Zone.

Description The `dat_pz_create()` function creates an instance of the Protection Zone. The Protection Zone provides Consumers a mechanism for association Endpoints with LMRs and RMRs to provide protection for local and remote memory accesses by DTOs.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INSUFFICIENT_RESOURCES`	The operation failed due to resource limitations.
`DAT_INVALID_PARAMETER`	Invalid parameter.
`DAT_INVALID_HANDLE`	The *ia_handle* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Safe

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_pz_free – destroy an instance of the Protection Zone

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_pz_free (
    IN    DAT_PZ_HANDLE    pz_handle
    )
```

Parameters *pz_handle* Handle for an instance of Protection Zone to be destroyed.

Description The `dat_pz_free()` function destroys an instance of the Protection Zone. The Protection Zone cannot be destroyed if it is in use by an Endpoint, LMR, or RMR.

Use of the handle of the destroyed Protection Zone in any subsequent operation except for `dat_pz_free()` fails.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INVALID_STATE`	Parameter in an invalid state. The Protection Zone was in use by Endpoint, LMR, or RMR instances.
`DAT_INVALID_HANDLE`	The *pz_handle* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Unsafe

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_pz_query – provides parameters of the Protection Zone

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_pz_query (
    IN    DAT_PZ_HANDLE        pz_handle,
    IN    DAT_PZ_PARAM_MASK    pz_param_mask,
    OUT   DAT_PZ_PARAM         *pz_param
    )
```

Parameters

pz_handle: Handle for the created instance of the Protection Zone.

pz_param_mask: Mask for Protection Zone parameters.

pz_param: Pointer to a Consumer-allocated structure that the Provider fills with Protection Zone parameters.

Description The `dat_pz_query()` function provides the Consumer parameters of the Protection Zone. The Consumer passes in a pointer to the Consumer-allocated structures for Protection Zone parameters that the Provider fills.

The *pz_param_mask* parameter allows Consumers to specify which parameters to query. The Provider returns values for *pz_param_mask* requested parameters. The Provider can return values for any other parameters.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_PARAMETER` The *pz_param_mask* parameter is invalid.

`DAT_INVALID_HANDLE` The *pz_handle* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Safe

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_registry_add_provider – declare the Provider with the Dynamic Registry

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_registry_add_provider (
    IN    const DAT_PROVIDER          *provider,
    IN    const DAT_PROVIDER_INFO     *provider_info
    )
```

Parameters

provider Self-description of a Provider.

provider_info Attributes of the Provider.

Description The Provider declares itself with the Dynamic Registry. Note that the caller can choose to register itself multiple times, for example once for each port. The choice of what to virtualize is up to the Provider. Each registration provides an Interface Adapter to DAT. Each Provider must have a unique name.

The same IA Name cannot be added multiple times. An attempt to register the same IA Name again results in an error with the return value `DAT_PROVIDER_ALREADY_REGISTERED`.

The contents of provider_info must be the same as those the Consumer uses in the call to `dat_ia_open(3DAT)` directly, or the ones provided indirectly defined by the header files with which the Consumer compiled.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INSUFFICIENT_RESOURCES`	The maximum number of Providers was already registered.
`DAT_INVALID_PARAMETER`	Invalid parameter.
`DAT_PROVIDER_ALREADY_REGISTERED`	Invalid or nonunique name.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	

See Also `dat_ia_open(3DAT)`, `libdat(3LIB)`, `attributes(5)`

Name dat_registry_list_providers – obtain a list of available pProviders from the Static Registry

Synopsis

```
typedef struct dat_provider_info {
    char ia_name[DAT_NAME_MAX_LENGTH];
    DAT_UINT32      dapl_version_major;
    DAT_UINT32      dapl_version_minor;
    DAT_BOOLEAN     is_thread_safe;
    } DAT_PROVIDER_INFO;

cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_registry_list_providers (
    IN    DAT_COUNT          max_to_return,
    OUT   DAT_COUNT          *number_entries,
    OUT   DAT_PROVIDER_INFO  *(dat_provider_list[])
    )
```

Parameters

max_to_return Maximum number of entries that can be returned to the Consumer in the *dat_provider_list*.

number_entries The actual number of entries returned to the Consumer in the *dat_provider_list* if successful or the number of Providers available.

dat_provider_list Points to an array of `DAT_PROVIDER_INFO` pointers supplied by the Consumer. Each Provider's information will be copied to the destination specified.

Description The `dat_registry_list_providers()` function allows the Consumer to obtain a list of available Providers from the Static Registry. The information provided is the Interface Adapter name, the uDAPL/kDAPL API version supported, and whether the provided version is thread-safe. The Consumer can examine the attributes to determine which (if any) Interface Adapters it wants to open. This operation has no effect on the Registry itself.

The Registry can open an IA using a Provider whose *dapl_version_minor* is larger than the one the Consumer requests if no Provider entry matches exactly. Therefore, Consumers should expect that an IA can be opened successfully as long as at least one Provider entry returned by `dat_registry_list_providers()` matches the *ia_name*, *dapl_version_major*, and *is_thread_safe* fields exactly, and has a *dapl_version_minor* that is equal to or greater than the version requested.

If the operation is successful, the returned value is `DAT_SUCCESS` and *number_entries* indicates the number of entries filled by the registry in *dat_provider_list*.

If the operation is not successful, then *number_entries* returns the number of entries in the registry. Consumers can use this return to allocate *dat_provider_list* large enough for the

registry entries. This number is just a snapshot at the time of the call and may be changed by the time of the next call. If the operation is not successful, then the content of *dat_provider_list* is not defined.

If *dat_provider_list* is too small, including pointing to `NULL` for the registry entries, then the operation fails with the return `DAT_INVALID_PARAMETER`.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INVALID_PARAMETER`	Invalid parameter. For example, *dat_provider_list* is too small or `NULL`.
`DAT_INTERNAL_ERROR`	Internal error. The DAT static registry is missing.

Usage

`DAT_NAME_MAX_LENGTH` includes the null character for string termination.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Safe

See Also

`libdat(3LIB)`, `attributes(5)`

Name dat_registry_remove_provider – unregister the Provider from the Dynamic Registry

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_registry_remove_provider  (
    IN       DAT_PROVIDER         *provider
    IN const DAT_PROVIDER_INFO    *provider_info
    )
```

Parameters

provider Self-description of a Provider.

provider_info Attributes of the Provider.

Description The Provider removes itself from the Dynamic Registry. It is the Provider's responsibility to complete its sessions. Removal of the registration only prevents new sessions.

The Provider cannot be removed while it is in use. An attempt to remove the Provider while it is in use results in an error with the return code `DAT_PROVIDER_IN_USE`.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_PARAMETER` Invalid parameter. The Provider was not found.

`DAT_PROVIDER_IN_USE` The Provider was in use.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_rmr_bind – bind the RMR to the specified memory region within an LMR

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_rmr_bind(
    IN    DAT_RMR_HANDLE        rmr_handle,
    IN    DAT_LMR_TRIPLET       *lmr_triplet,
    IN    DAT_MEM_PRIV_FLAGS    mem_privileges,
    IN    DAT_EP_HANDLE         ep_handle,
    IN    DAT_RMR_COOKIE        user_cookie,
    IN    DAT_COMPLETION_FLAGS  completion_flags,
    OUT   DAT_RMR_CONTEXT       *rmr_context
    )
```

Parameters

rmr_handle Handle for an RMR instance.

lmr_triplet A pointer to an *lmr_triplet* that defines the memory region of the LMR.

mem_privileges Consumer-requested memory access privileges for the registered remote memory region. The Default value is `DAT_MEM_PRIV_NONE_FLAG`. The constant value `DAT_MEM_PRIV_ALL_FLAG` = 0x33, which specifies both Read and Write privileges, is also defined. Memory privilege definitions are as follows:

Remote Read	`DAT_MEM_PRIV_REMOTE_READ_FLAG`	
	0x02	Remote read access requested.
Remote Write	`DAT_MEM_PRIV_REMOTE_WRITE_FLAG`	
	0x20	Remote write access requested.

ep_handle Endpoint to which `dat_rmr_bind()` is posted.

user_cookie User-provided cookie that is returned to a Consumer at the completion of the `dat_rmr_bind()`. Can be `NULL`.

completion_flags Flags for RMR Bind. The default `DAT_COMPLETION_DEFAULT_FLAG` is 0. Flag definitions are as follows:

Completion Suppression	`DAT_COMPLETION_SUPPRESS_FLAG`	
	0x01	Suppress successful Completion.
Notification of Completion	`DAT_COMPLETION_UNSIGNALLED_FLAG`	
	0x04	Non-notification completion. Local Endpoint must be configured for Notification

		Suppression.
	Barrier Fence	`DAT_COMPLETION_BARRIER_FENCE_FLAG`
		0x08 Request for Barrier Fence.

rmr_context New *rmr_context* for the bound RMR suitable to be shared with a remote host.

Description The `dat_rmr_bind()` function binds the RMR to the specified memory region within an LMR and provides the new *rmr_context* value. The `dat_rmr_bind()` operation is a lightweight asynchronous operation that generates a new *rmr_context*. The Consumer is notified of the completion of this operation through a *rmr_bind* Completion event on the *request_evd_handle* of the specified Endpoint *ep_handle*.

The return value of *rmr_context* can be transferred by local Consumer to a Consumer on a remote host to be used for an RDMA DTO. The use of *rmr_context* by a remote host for an RDMA DTO prior to the completion of the `dat_rmr_bind()` can result in an error and a broken connection. The local Consumer can ensure that the remote Consumer does not have *rmr_context* before `dat_rmr_bind()` is completed. One way is to "wait" for the completion `dat_rmr_bind()` on the *rmr_bind* Event Dispatcher of the specified Endpoint *ep_handle*. Another way is to send *rmr_context* in a Send DTO over the connection of the Endpoint *ep_handle*. The barrier-fencing behavior of the `dat_rmr_bind()` with respect to Send and RDMA DTOs ensures that a Send DTO does not start until `dat_rmr_bind()` completed.

The `dat_rmr_bind()` function automatically fences all Send, RDMA Read, and RDMA Write DTOs and `dat_rmr_bind()` operations submitted on the Endpoint *ep_handle* after the `dat_rmr_bind()`. Therefore, none of these operations starts until `dat_rmr_bind()` is completed.

If the RMR Bind fails after `dat_rmr_bind()` returns, connection of *ep_handle* is broken. The Endpoint transitions into a `DAT_EP_STATE_DISCONNECTED` state and the `DAT_CONNECTION_EVENT_BROKEN` event is delivered to the *connect_evd_handle* of the Endpoint.

The `dat_rmr_bind()` function employs fencing to ensure that operations sending the RMR Context on the same Endpoint as the bind specified cannot result in an error from the peer side using the delivered RMR Context too soon. One method, used by InfiniBand, is to ensure that none of these operations start on the Endpoint until after the bind is completed. Other transports can employ different methods to achieve the same goal.

Any RDMA DTO that uses the previous value of *rmr_context* after the `dat_rmr_bind()` is completed fail and report a protection violation.

By default, `dat_rmr_bind()` generates notification completions.

The *mem_privileges* parameter allows Consumers to restrict the type of remote accesses to the registered RMR by RDMA DTOs. Providers whose underlying Transports require that privileges of the requested RMR and the associated LMR match, that is

- Set RMR's `DAT_MEM_PRIV_REMOTE_READ_FLAG` requires that LMR's `DAT_MEM_PRIV_LOCAL_READ_FLAG` is also set,
- Set RMR's `DAT_MEM_PRIV_REMOTE_WRITE_FLAG` requires that LMR's `DAT_MEM_PRIV_LOCAL_WRITE_FLAG` is also set,

or the operation fails and returns `DAT_PRIVILEGES_VIOLATION`.

In the *lmr_triplet*, the value of *length* of zero means that the Consumer does not want to associate an RMR with any memory region within the LMR and the return value of *rmr_context* for that case is undefined.

The completion of the posted RMR Bind is reported to the Consumer asynchronously through a DTO Completion event based on the specified *completion_flags* value. The value of `DAT_COMPLETION_UNSIGNALLED_FLAG` is only valid if the Endpoint Request Completion Flags `DAT_COMPLETION_UNSIGNALLED_FLAG`. Otherwise, `DAT_INVALID_PARAMETER` is returned.

The *user_cookie* parameter allows Consumers to have unique identifiers for each `dat_rmr_bind()`. These identifiers are completely under user control and are opaque to the Provider. The Consumer is not required to ensure the uniqueness of the *user_cookie* value. The *user_cookie* is returned to the Consumer in the *rmr_bind* Completion event for this operation.

The operation is valid for the Endpoint in the `DAT_EP_STATE_CONNECTED` and `DAT_EP_STATE_DISCONNECTED` states. If the operation returns successfully for the Endpoint in `DAT_EP_STATE_DISCONNECTED` state, the posted RMR Bind is immediately flushed to *request_evd_handle*.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INSUFFICIENT_RESOURCES`	The operation failed due to resource limitations.
`DAT_INVALID_PARAMETER`	Invalid parameter. For example, the *target_address* or *segment_length* exceeded the limits of the existing LMR.
`DAT_INVALID_HANDLE`	Invalid DAT handle.
`DAT_INVALID_STATE`	Parameter in an invalid state. Endpoint was not in the a `DAT_EP_STATE_CONNECTED` or `DAT_EP_STATE_DISCONNECTED` state.
`DAT_MODEL_NOT_SUPPORTED`	The requested Model was not supported by the Provider.
`DAT_PRIVILEGES_VIOLATION`	Privileges violation for local or remote memory access.
`DAT_PROTECTION_VIOLATION`	Protection violation for local or remote memory access.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Unsafe

See Also libdat(3LIB), attributes(5)

Name dat_rmr_create – create an RMR for the specified Protection Zone

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_rmr_create(
    IN    DAT_PZ_HANDLE     pz_handle,
    OUT   DAT_RMR_HANDLE    *rmr_handle
    )
```

Parameters

pz_handle Handle for an instance of the Protection Zone.

rmr_handle Handle for the created instance of an RMR.

Description The `dat_rmr_create()` function creates an RMR for the specified Protection Zone. This operation is relatively heavy. The created RMR can be bound to a memory region within the LMR through a lightweight `dat_rmr_bind(3DAT)` operation that generates *rmr_context*.

If the operation fails (does not return `DAT_SUCCESS`), the return values of *rmr_handle* are undefined and Consumers should not use them.

The *pz_handle* parameter provide Consumers a way to restrict access to an RMR by authorized connection only.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INSUFFICIENT_RESOURCES`	The operation failed due to resource limitations.
`DAT_INVALID_HANDLE`	The *pz_handle* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Safe

See Also `dat_rmr_bind(3DAT)` , `libdat(3LIB)`, `attributes(5)`

Name dat_rmr_free – destroy an instance of the RMR

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_rmr_free (
    IN    DAT_RMR_HANDLE    rmr_handle
    )
```

Parameters *rmr_handle* Handle for an instance of the RMR to be destroyed.

Description The `dat_rmr_free()` function destroys an instance of the RMR.

Use of the handle of the destroyed RMR in any subsequent operation except for the `dat_rmr_free()` fails. Any remote RDMA operation that uses the destroyed RMR *rmr_context*, whose Transport-specific request arrived to the local host after the `dat_rmr_free()` has completed, fails and reports a protection violation. Remote RDMA operation that uses the destroyed RMR *rmr_context*, whose Transport-specific request arrived to the local host prior to the `dat_rmr_free()` return, might or might not complete successfully. If it fails, `DAT_DTO_ERR_REMOTE_ACCESS` is reported in `DAT_DTO_COMPLETION_STATUS` for the remote RDMA DTO and the connection is broken.

The `dat_rmr_free()` function is allowed on either bound or unbound RMR. If RMR is bound, `dat_rmr_free()` unbinds (free HCA TPT and other resources and whatever else binds with length of 0 should do), and then free RMR.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *rmr_handle* handle is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Unsafe

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_rmr_query – provide RMR parameters

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_rmr_query (
    IN    DAT_RMR_HANDLE        rmr_handle,
    IN    DAT_RMR_PARAM_MASK    rmr_param_mask,
    OUT   DAT_RMR_PARAM         *rmr_param
    )
```

Parameters

rmr_handle	Handle for an instance of the RMR.
rmr_param_mask	Mask for RMR parameters.
rmr_param	Pointer to a Consumer-allocated structure that the Provider fills with RMR parameters.

Description The `dat_rmr_query()` function provides RMR parameters to the Consumer. The Consumer passes in a pointer to the Consumer-allocated structures for RMR parameters that the Provider fills.

The *rmr_param_mask* parameter allows Consumers to specify which parameters to query. The Provider returns values for *rmr_param_mask* requested parameters. The Provider can return values for any other parameters.

Not all parameters can have a value at all times. For example, *lmr_handle*, *target_address*, *segment_length*, *mem_privileges*, and *rmr_context* are not defined for an unbound RMR.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INVALID_PARAMETER`	The *rmr_param_mask* parameter is invalid.
`DAT_INVALID_HANDLE`	The *mr_handle* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Unsafe

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_rsp_create – create a Reserved Service Point

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_rsp_create (
    IN    DAT_IA_HANDLE    ia_handle,
    IN    DAT_CONN_QUAL    conn_qual,
    IN    DAT_EP_HANDLE    ep_handle,
    IN    DAT_EVD_HANDLE   evd_handle,
    OUT   DAT_RSP_HANDLE   *rsp_handle
    )
```

Parameters

ia_handle Handle for an instance of DAT IA.

conn_qual Connection Qualifier of the IA the Reserved Service Point listens to.

ep_handle Handle for the Endpoint associated with the Reserved Service Point that is the only Endpoint that can accept a Connection Request on this Service Point. The value `DAT_HANDLE_NULL` requests the Provider to associate a Provider-created Endpoint with this Service Point.

evd_handle The Event Dispatcher to which an event of Connection Request arrival is generated.

rsp_handle Handle to an opaque Reserved Service Point.

Description The `dat_rsp_create()` function creates a Reserved Service Point with the specified Endpoint that generates, at most, one Connection Request that is delivered to the specified Event Dispatcher in a Notification event.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INSUFFICIENT_RESOURCES`	The operation failed due to resource limitations.
`DAT_INVALID_HANDLE`	The *ia_handle*, *evd_handle*, or *ep_handle* parameter is invalid.
`DAT_INVALID_PARAMETER`	The *conn_qual* parameter is invalid.
`DAT_INVALID_STATE`	Parameter in an invalid state. For example, an Endpoint was not in the Idle state.
`DAT_CONN_QUAL_IN_USE`	Specified Connection Qualifier is in use.

Usage The usage of a Reserve Service Point is as follows:

- The DAT Consumer creates a Local Endpoint and configures it appropriately.
- The DAT Consumer creates a Reserved Service Point specifying the Local Endpoint.
- The Reserved Service Point performs the following:

- Collects native transport information reflecting a received Connection Request.
- Creates a Pending Connection Request.
- Creates a Connection Request Notice (event) that includes the Pending Connection Request (which includes, among others, Reserved Service Point Connection Qualifier, its Local Endpoint, and information about remote Endpoint).
- Delivers the Connection Request Notice to the Consumer-specified target (CNO) *evd_handle*. The Local Endpoint is transitioned from Reserved to Passive Connection Pending state.

- Upon receiving a connection request, or at some time subsequent to that, the DAT Consumer must either `accept()` or `reject()` the Pending Connection Request.
- If accepted, the original Local Endpoint is now in a *Connected* state and fully usable for this connection, pending only native transport mandated RTU messages. This includes binding it to the IA port if that was not done previously. The Consumer is notified that the Endpoint is in a *Connected* state by a Connection Established Event on the Endpoint *connect_evd_handle*.
- If rejected, the Local Endpoint point transitions into *Unconnected* state. The DAT Consumer can elect to destroy it or reuse it for other purposes.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Safe

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_rsp_free – destroy an instance of the Reserved Service Point

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_rsp_free (
    IN    DAT_RSP_HANDLE    rsp_handle
    )
```

Parameters *rsp_handle* Handle for an instance of the Reserved Service Point.

Description The `dat_rsp_free()` function destroys a specified instance of the Reserved Service Point.

Any incoming Connection Requests for the Connection Qualifier on the destroyed Service Point was listening on are automatically rejected by the Provider with the return analogous to the no listening Service Point.

The behavior of the Connection Requests in progress is undefined and left to an implementation, but it must be consistent. This means that either a Connection Requested Event was generated for the Event Dispatcher associated with the Service Point, including the creation of the Connection Request instance, or the Connection Request is rejected by the Provider without any local notification.

This operation has no effect on previously generated Connection Request Event and Connection Request.

The behavior of this operation with creation of a Service Point on the same Connection Qualifier at the same time is not defined. Consumers are advised to avoid this scenario.

For the Reserved Service Point, the Consumer-provided Endpoint reverts to Consumer control. Consumers shall be aware that due to a race condition, this Reserved Service Point might have generated a Connection Request Event and passed the associated Endpoint to a Consumer in it.

Use of the handle of the destroyed Service Point in any consequent operation fails.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *rsp_handle* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Unsafe

See Also libdat(3LIB), attributes(5)

Name dat_rsp_query – provide parameters of the Reserved Service Point

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_rsp_query (
    IN    DAT_RSP_HANDLE        rsp_handle,
    IN    DAT_RSP_PARAM_MASK    rsp_param_mask,
    OUT   DAT_RSP_PARAM         *rsp_param
    )
```

Parameters

rsp_handle	Handle for an instance of Reserved Service Point
rsp_param_mask	Mask for RSP parameters.
rsp_param	Pointer to a Consumer-allocated structure that the Provider fills for Consumer-requested parameters.

Description The `dat_rsp_query()` function provides to the Consumer parameters of the Reserved Service Point. The Consumer passes in a pointer to the Consumer-allocated structures for RSP parameters that the Provider fills.

The *rsp_param_mask* parameter allows Consumers to specify which parameters to query. The Provider returns values for *rsp_param_mask* requested parameters. The Provider can return values for any other parameters.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INVALID_HANDLE`	The *rsp_handle* parameter is invalid.
`DAT_INVALID_PARAMETER`	The *rsp_param_mask* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Safe

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_set_consumer_context – set Consumer context

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_set_consumer_context (
    IN    DAT_HANDLE    dat_handle,
    IN    DAT_CONTEXT   context
    )
```

Parameters

dat_handle Handle for a DAT Object associated with *context*.

context Consumer context to be stored within the associated *dat_handle*. The Consumer context is opaque to the uDAPL Provider. `NULL` represents no context.

Description The `dat_set_consumer_context()` function associates a Consumer context with the specified *dat_handle*. The *dat_handle* can be one of the following handle types: `DAT_IA_HANDLE`, `DAT_EP_HANDLE`, `DAT_EVD_HANDLE`, `DAT_CR_HANDLE`, `DAT_RSP_HANDLE`, `DAT_PSP_HANDLE`, `DAT_PZ_HANDLE`, `DAT_LMR_HANDLE`, `DAT_RMR_HANDLE`, or `DAT_CNO_HANDLE`.

Only a single Consumer context is provided for any *dat_handle*. If there is a previous Consumer context associated with the specified handle, the new context replaces the old one. The Consumer can disassociate the existing context by providing a NULL pointer for the *context*. The Provider makes no assumptions about the contents of *context*; no check is made on its value. Furthermore, the Provider makes no attempt to provide any synchronization for access or modification of the *context*.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_PARAMETER` The *context* parameter is invalid.

`DAT_INVALID_HANDLE` The *dat_handle* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Unsafe

See Also `dat_get_consumer_context(3DAT)`, `libdat(3LIB)`, `attributes(5)`

Name dat_srq_create – create an instance of a shared receive queue

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_srq_create (
        IN      DAT_IA_HANDLE       ia_handle,
        IN      DAT_PZ_HANDLE       pz_handle,
        IN      DAT_SRQ_ATTR        *srq_attr,
        OUT     DAT_SRQ_HANDLE      *srq_handle
    )
```

Parameters

ia_handle A handle for an open instance of the IA to which the created SRQ belongs.

pz_handle A handle for an instance of the Protection Zone.

srq_attr A pointer to a structure that contains Consumer-requested SRQ attributes.

srq_handle A handle for the created instance of a Shared Receive Queue.

Description The `dat_srq_create()` function creates an instance of a Shared Receive Queue (SRQ) that is provided to the Consumer as *srq_handle*. If the value of `DAT_RETURN` is not `DAT_SUCCESS`, the value of *srq_handle* is not defined.

The created SRQ is unattached to any Endpoints.

The Protection Zone *pz_handle* allows Consumers to control what local memory can be used for the Recv DTO buffers posted to the SRQ. Only memory referred to by LMRs of the posted Recv buffers that match the SRQ Protection Zone can be accessed by the SRQ.

The *srq_attributes* argument specifies the initial attributes of the created SRQ. If the operation is successful, the created SRQ will have the queue size at least *max_recv_dtos* and the number of entries on the posted Recv scatter list of at lease *max_recv_iov*. The created SRQ can have the queue size and support number of entries on post Recv buffers larger than requested. Consumer can query SRQ to find out the actual supported queue size and maximum Recv IOV.

The Consumer must set *low_watermark* to `DAT_SRQ_LW_DEFAULT` to ensure that an asynchronous event will not be generated immediately, since there are no buffers in the created SRQ. The Consumer should set the Maximum Receive DTO attribute and the Maximum number of elements in IOV for posted buffers as needed.

When an associated EP tries to get a buffer from SRQ and there are no buffers available, the behavior of the EP is the same as when there are no buffers on the EP Recv Work Queue.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INSUFFICIENT_RESOURCES` The operation failed due to resource limitations.

DAT_INVALID_HANDLE	Either *ia_handle* or *pz_handle* is an invalid DAT handle.
DAT_INVALID_PARAMETER	One of the parameters is invalid. Either one of the requested SRQ attributes was invalid or a combination of attributes is invalid.
DAT_MODEL_NOT_SUPPORTED	The requested Model was not supported by the Provider.

Usage SRQ is created by the Consumer prior to creation of the EPs that will be using it. Some Providers might restrict whether multiple EPs that share a SRQ can have different Protection Zones. Check the *srq_ep_pz_difference_support* Provider attribute. The EPs that use SRQ might or might not use the same *recv_evd*.

Since a Recv buffer of SRQ can be used by any EP that is using SRQ, the Consumer should ensure that the posted Recv buffers are large enough to receive an incoming message on any of the EPs.

If Consumers do not want to receive an asynchronous event when the number of buffers in SRQ falls below the Low Watermark, they should leave its value as DAT_SRQ_LW_DEFAULT. If Consumers do want to receive a notification, they can set the value to the desired one by calling dat_srq_set_lw(3DAT).

SRQ allows the Consumer to use fewer Recv buffers then posting the maximum number of buffers for each connection. If the Consumer can upper bound the number of incoming messages over all connections whose local EP is using SRQ, then instead of posting this maximum for each connection the Consumer can post them for all connections on SRQ. For example, the maximum utilized link bandwidth divided over the message size can be used for an upper bound.

Depending on the underlying Transport, one or more messages can arrive simultaneously on an EP that is using SRQ. Thus, the same EP can have multiple Recv buffers in its possession without these buffers being on SRQ or *recv_evd*.

Since Recv buffers can be used by multiple connections of the local EPs that are using SRQ, the completion order of the Recv buffers is no longer guaranteed even when they use of the same *recv_evd*. For each connection the Recv buffers completion order is guaranteed to be in the order of the posted matching Sends to the other end of the connection. There is no ordering guarantee that Receive buffers will be returned in the order they were posted even if there is only a single connection (Endpoint) associated with the SRQ. There is no ordering guarantee between different connections or between different *recv_evd*s.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.2

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also dat_srq_free(3DAT), dat_srq_post_recv(3DAT), dat_srq_query(3DAT), dat_srq_resize(3DAT), dat_srq_set_lw(3DAT), libdat(3LIB), attributes(5)

Name dat_srq_free – destroy an instance of the shared receive queue

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_srq_free (
    IN      DAT_SRQ_HANDLE      srq_handle
    )
```

Parameters *srq_handle* A handle for an instance of SRQ to be destroyed.

Description The `dat_srq_free()` function destroys an instance of the SRQ. The SRQ cannot be destroyed if it is in use by an EP.

It is illegal to use the destroyed handle in any consequent operation.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INVALID_HANDLE`	The *srq_handle* argument is an invalid DAT handle.
`DAT_SRQ_IN_USE`	The Shared Receive Queue can not be destroyed because it is in still associated with an EP instance.

Usage If the Provider detects the use of a deleted object handle, it should return `DAT_INVALID_HANDLE`. The Provider should avoid assigning the used handle as long as possible. Once reassigned the handle is no longer a handle of a destroyed object.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.2
MT-Level	Unsafe

See Also `dat_srq_create(3DAT)`, `dat_srq_post_recv(3DAT)`, `dat_srq_query(3DAT)`, `dat_srq_resize(3DAT)`, `dat_srq_set_lw(3DAT)`, `libdat(3LIB)`, `attributes(5)`

Name dat_srq_post_recv – add receive buffers to shared receive queue

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_srq_post_recv (
    IN      DAT_SRQ_HANDLE      srq_handle,
    IN      DAT_COUNT           num_segments,
    IN      DAT_LMR_TRIPLET     *local_iov,
    IN      DAT_DTO_COOKIE      user_cookie
    )
```

Parameters

srq_handle A handle for an instance of the SRQ.

num_segments The number of *lmr_triplets* in *local_iov*. Can be 0 for receiving a zero-size message.

local_iov An I/O Vector that specifies the local buffer to be filled. Can be `NULL` for receiving a zero-size message.

user_cookie A user-provided cookie that is returned to the Consumer at the completion of the Receive DTO. Can be `NULL`.

Description The `dat_srq_post_recv()` function posts the receive buffer that can be used for the incoming message into the *local_iov* by any connected EP that uses SRQ.

The *num_segments* argument specifies the number of segments in the *local_iov*. The *local_iov* segments are filled in the I/O Vector order until the whole message is received. This ensures that all the front segments of the *local_iov* I/O Vector are completely filled, only one segment is partially filled, if needed, and all segments that follow it are not filled at all. The actual order of segment fillings is left to the implementation.

The *user_cookie* argument allows Consumers to have unique identifiers for each DTO. These identifiers are completely under user control and are opaque to the Provider. There is no requirement on the Consumer that the value *user_cookie* should be unique for each DTO. The *user_cookie* is returned to the Consumer in the Completion event for the posted Receive.

The completion of the posted Receive is reported to the Consumer asynchronously through a DTO Completion event based on the configuration of the EP that dequeues the posted buffer and the specified *completion_flags* value for Solicited Wait for the matching Send. If EP Recv Completion Flag is `DAT_COMPLETION_UNSIGNALLED_FLAG`, which is the default value for SRQ EP, then all posted Recvs will generate completions with Signal Notifications.

A Consumer should not modify the *local_iov* or its content until the DTO is completed. When a Consumer does not adhere to this rule, the behavior of the Provider and the underlying Transport is not defined. Providers that allow Consumers to get ownership of the *local_iov* but not the memory it specified back after the `dat_srq_post_recv()` returns should document

this behavior and also specify its support in Provider attributes. This behavior allows Consumer full control of the *local_iov* content after `dat_srq_post_recv()` returns. Because this behavior is not guaranteed by all Providers, portable Consumers shall not rely on this behavior. Consumers shall not rely on the Provider copying *local_iov* information.

The `DAT_SUCCESS` return of the `dat_srq_post_recv()` is at least the equivalent of posting a Receive operation directly by native Transport. Providers shall avoid resource allocation as part of `dat_srq_post_recv()` to ensure that this operation is nonblocking.

The completion of the Receive posted to the SRQ is equivalent to what happened to the Receive posted to the Endpoint for the Endpoint that dequeued the Receive buffer from the Shared Receive queue.

The posted Recv DTO will complete with signal, equivalently to the completion of Recv posted directly to the Endpoint that dequeued the Recv buffer from SRQ with `DAT_COMPLETION_UNSIGNALLED_FLAG` value not set for it.

The posted Recv DTOs will complete in the order of Send postings to the other endpoint of each connection whose local EP uses SRQ. There is no ordering among different connections regardless if they share SRQ and *recv_evd* or not.

If the reported status of the Completion DTO event corresponding to the posted RDMA Read DTO is not `DAT_DTO_SUCCESS`, the content of the *local_iov* is not defined and the *transfered_length* in the DTO Completion event is not defined.

The operation is valid for all states of the Shared Receive Queue.

The `dat_srq_post_recv()` function is asynchronous, nonblocking, and its thread safety is Provider-dependent.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INVALID_HANDLE`	The *srq_handle* argument is an invalid DAT handle.
`DAT_INSUFFICIENT_RESOURCES`	The operation failed due to resource limitations.
`DAT_INVALID_PARAMETER`	Invalid parameter. For example, one of the IOV segments pointed to a memory outside its LMR.
`DAT_PROTECTION_VIOLATION`	Protection violation for local or remote memory access.
	Protection Zone mismatch between an LMR of one of the *local_iov* segments and the SRQ.
`DAT_PRIVILEGES_VIOLATION`	Privileges violation for local or remote memory access. One of the LMRs used in local_iov was either invalid or did not have the local write privileges.

Usage For the best Recv operation performance, the Consumer should align each buffer segment of *local_iov* to the Optimal Buffer Alignment attribute of the Provider. For portable applications, the Consumer should align each buffer segment of *local_iov* to the `DAT_OPTIMAL_ALIGNMENT`.

Since any of the Endpoints that use the SRQ can dequeue the posted buffer from SRQ, Consumers should post a buffer large enough to handle incoming message on any of these Endpoint connections.

The buffer posted to SRQ does not have a DTO completion flag value. Posting Recv buffer to SRQ is semantically equivalent to posting to EP with `DAT_COMPLETION_UNSIGNALLED_FLAG` is not set. The configuration of the Recv Completion flag of an Endpoint that dequeues the posted buffer defines how DTO completion is generated. If the Endpoint Recv Completion flag is `DAT_COMPLETION_SOLICITED_WAIT_FLAG` then matching Send DTO completion flag value for Solicited Wait determines if the completion will be Signalled or not. If the Endpoint Recv Completion flag is not `DAT_COMPLETION_SOLICITED_WAIT_FLAG`, the posted Recv completion will be generated with Signal. If the Endpoint Recv Completion flag is `DAT_COMPLETION_EVD_THRESHOLD_FLAG`, the posted Recv completion will be generated with Signal and *dat_evd_wait* threshold value controls if the waiter will be unblocked or not.

Only the Endpoint that is in Connected or Disconnect Pending states can dequeue buffers from SRQ. When an Endpoint is transitioned into Disconnected state, all the buffers that it dequeued from SRQ are queued on the Endpoint *recv_evd*. All the buffers that the Endpoint has not completed by the time of transition into Disconnected state and that have not completed message reception will be flushed.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.2
MT-Level	Unsafe

See Also `dat_srq_create(3DAT)`, `dat_srq_free(3DAT)`, `dat_srq_query(3DAT)`, `dat_srq_resize(3DAT)`, `dat_srq_set_lw(3DAT)`, `libdat(3LIB)`, `attributes(5)`

Name dat_srq_query – provide parameters of the shared receive queue

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_srq_query (
    IN      DAT_SRQ_HANDLE      srq_handle,
    IN      DAT_SRQ_PARAM_MASK  srq_param_mask,
    OUT     DAT_SRQ_PARAM       *srq_param
    )
```

Parameters

srq_handle	A handle for an instance of the SRQ.
srq_param_mask	The mask for SRQ parameters.
srq_param	A pointer to a Consumer-allocated structure that the Provider fills with SRQ parameters.

Description The `dat_srq_query()` function provides to the Consumer SRQ parameters. The Consumer passes a pointer to the Consumer-allocated structures for SRQ parameters that the Provider fills.

The *srq_param_mask* argument allows Consumers to specify which parameters to query. The Provider returns values for the requested *srq_param_mask* parameters. The Provider can return values for any other parameters.

In addition to the elements in SRQ attribute, `dat_srq_query()` provides additional information in the `srq_param` structure if Consumer requests it with *srq_param_mask* settings. The two that are related to entry counts on SRQ are the number of Receive buffers (*available_dto_count*) available for EPs to dequeue and the number of occupied SRQ entries (*outstanding_dto_count*) not available for new Recv buffer postings.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INVALID_PARAMETER`	The *srq_param_mask* argument is invalid.
`DAT_INVALID_HANDLE`	The *srq_handle* argument is an invalid DAT handle.

Usage The Provider might not be able to provide the number of outstanding Recv of SRQ or available Recvs of SRQ. The Provider attribute indicates if the Provider does not support the query for one or these values. Even when the Provider supports the query for one or both of these values, it might not be able to provide this value at this moment. In either case, the return value for the attribute that cannot be provided will be `DAT_VALUE_UNKNOWN`.

Example: Consumer created SRQ with 10 entries and associated 1 EP with it. 3 Recv buffers have been posted to it. The query will report:

```
max_recv_dtos=10,
available_dto_count=3,
outstanding_dto_count=3.
```

After a Send message arrival the query will report:

```
max_recv_dtos=10,
available_dto_count=2,
outstanding_dto_count=3.
```

After Consumer dequeues Recv completion the query will report:

```
max_recv_dtos=10,
available_dto_count=2,
outstanding_dto_count=2.
```

In general, each EP associated with SRQ can have multiple buffers in progress of receiving messages as well completed Recv on EVDs. The watermark setting helps to control how many Recv buffers posted to SRQ an Endpoint can own.

If the Provider cannot support the query for the number of outstanding Recv of SRQ or available Recvs of SRQ, the value return for that attribute should be `DAT_VALUE_UNKNOWN`.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.2
MT-Level	Unsafe

See Also

`dat_srq_create(3DAT)`, `dat_srq_free(3DAT)`, `dat_srq_post_recv(3DAT)`, `dat_srq_resize(3DAT)`, `dat_srq_set_lw(3DAT)`, `libdat(3LIB)`, `attributes(5)`

Name dat_srq_resize – modify the size of the shared receive queue

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_srq_resize (
    IN      DAT_SRQ_HANDLE      srq_handle,
    IN      DAT_COUNT           srq_max_recv_dto
    )
```

Parameters

srq_handle A handle for an instance of the SRQ.

srq_max_recv_dto The new maximum number of Recv DTOs that Shared Receive Queue must hold.

Description The `dat_srq_resize()` function modifies the size of the queue of SRQ.

Resizing of Shared Receive Queue should not cause any incoming messages on any of the EPs that use the SRQ to be lost. If the number of outstanding Recv buffers on the SRQ is larger then the requested *srq_max_recv_dto*, the operation returns `DAT_INVALID_STATE` and do not change SRQ. This includes not just the buffers on the SRQ but all outstanding Receive buffers that had been posted to the SRQ and whose completions have not reaped yet. Thus, the outstanding buffers include the buffers on SRQ, the buffers posted to SRQ at are at SRQ associated EPs, and the buffers posted to SRQ for which completions have been generated but not yet reaped by Consumer from recv_evds of the EPs that use the SRQ.

If the requested *srq_max_recv_dto* is below the SRQ low watermark, the operation returns `DAT_INVALID_STATE` and does not change SRQ.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INVALID_HANDLE`	The *srq_handle* argument is an invalid DAT handle.
`DAT_INVALID_PARAMETER`	The *srq_max_recv_dto* argument is invalid.
`DAT_INSUFFICIENT_RESOURCES`	The operation failed due to resource limitations.
`DAT_INVALID_STATE`	Invalid state. Either the number of entries on the SRQ exceeds the requested SRQ queue length or the requested SRQ queue length is smaller than the SRQ low watermark.

Usage The `dat_srq_resize()` function is required not to lose any buffers. Thus, it cannot shrink below the outstanding number of Recv buffers on SRQ. There is no requirement to shrink the SRQ to return `DAT_SUCCESS`.

The quality of the implementation determines how closely to the Consumer-requested value the Provider shrinks the SRQ. For example, the Provider can shrink the SRQ to the Consumer-requested value and if the requested value is smaller than the outstanding buffers

on SRQ, return DAT_INVALID_STATE; or the Provider can shrink to some value larger than that requested by the Consumer but below current SRQ size; or the Provider does not change the SRQ size and still returns DAT_SUCCESS.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.2
MT-Level	Unsafe

See Also dat_srq_create(3DAT), dat_srq_free(3DAT), dat_srq_post_recv(3DAT), dat_srq_query(3DAT), dat_srq_set_lw(3DAT), libdat(3LIB), attributes(5)

Name dat_srq_set_lw – set low watermark on shared receive queue

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_srq_set_lw (
        IN      DAT_SRQ_HANDLE      srq_handle,
        IN      DAT_COUNT           low_watermark
    )
```

Parameters

srq_handle A handle for an instance of a Shared Receive Queue.

low_watermark The low watermark for the number of Recv buffers on SRQ.

Description The `dat_srq_set_lw()` function sets the low watermark value for the SRQ and arms the SRQ for generating an asynchronous event for the low watermark. An asynchronous event will be generated when the number of buffers on the SRQ is below the low watermark for the first time. This can occur during the current call or when an associated EP takes a buffer from the SRQ.

The asynchronous event will be generated only once per setting of the low watermark. Once an event is generated, no new asynchronous events for the number of buffers inthe SRQ below the specified value will be generated until the SRQ is again set for the Low Watermark. If the Consumer is again interested in the event, the Consumer should set the low watermark again.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INVALID_HANDLE`	The *srq_handle* argument is an invalid DAT handle.
`DAT_INVALID_PARAMETER`	Invalid parameter; the value of *low_watermark* is exceeds the value of *max_recv_dtos*.
`DAT_MODEL_NOT_SUPPORTED`	The requested Model was not supported by the Provider. The Provider does not support SRQ Low Watermark.

Usage Upon receiving the asynchronous event for the SRQ low watermark, the Consumer can replenish Recv buffers on the SRQ or take any other action that is appropriate.

Regardless of whether an asynchronous event for the low watermark has been generated, this operation will set the generation of an asynchronous event with the Consumer-provided low watermark value. If the new low watermark value is below the current number of free Receive DTOs posted to the SRQ, an asynchronous event will be generated immediately. Otherwise the old low watermark value is simply replaced with the new one.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.2
MT-Level	Unsafe

See Also dat_srq_create(3DAT), dat_srq_free(3DAT), dat_srq_post_recv(3DAT), dat_srq_query(3DAT), dat_srq_resize(3DAT), libdat(3LIB), attributes(5)

Name dat_strerror – convert a DAT return code into human readable strings

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_strerror(
    IN    DAT_RETURN    return,
    OUT   const char    **major_message,
    OUT   const char    **minor_message
    )
```

Parameters

return DAT function return value.

message A pointer to a character string for the return.

Description The `dat_strerror()` function converts a DAT return code into human readable strings. The *major_message* is a string-converted `DAT_TYPE_STATUS`, while *minor_message* is a string-converted `DAT_SUBTYPE_STATUS`. If the return of this function is not `DAT_SUCCESS`, the values of *major_message* and *minor_message* are not defined.

If an undefined `DAT_RETURN` value was passed as the return parameter, the operation fails with `DAT_INVALID_PARAMETER` returned. The operation succeeds when `DAT_SUCCESS` is passed in as the return parameter.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_PARAMETER` Invalid parameter. The *return* value is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: uDAPL, 1.1, 1.2
MT-Level	Safe

See Also `libdat(3LIB)`, `attributes(5)`

Name demangle, cplus_demangle – decode a C++ encoded symbol name

Synopsis

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

#include <demangle.h>


int cplus_demangle(const char *symbol, char *prototype, size_t size);
```

Description The `cplus_demangle()` function decodes (demangles) a C++ linker symbol name (mangled name) into a (partial) C++ prototype, if possible. C++ mangled names may not have enough information to form a complete prototype.

The *symbol* string argument points to the input mangled name.

The *prototype* argument points to a user-specified output string buffer, of `size` bytes.

The `cplus_demangle()` function operates on mangled names generated by SPARCompilers C++ 3.0.1, 4.0.1, 4.1 and 4.2.

The `cplus_demangle()` function improves and replaces the `demangle()` function.

Refer to the `CC.1`, `dem.1`, and `c++filt.1` manual pages in the `/opt/SUNWspro/man/man1` directory. These pages are only available with the SPROcc package.

Return Values The `cplus_demangle()` function returns the following values:

`0`	The *symbol* argument is a valid mangled name and *prototype* contains a (partial) prototype for the symbol.
`DEMANGLE_ENAME`	The *symbol* argument is not a valid mangled name and the content of *prototype* is a copy of the symbol.
`DEMANGLE_ESPACE`	The *prototype* output buffer is too small to contain the prototype (or the symbol), and the content of *prototype* is undefined.

Name devid_get, devid_compare, devid_deviceid_to_nmlist, devid_free, devid_free_nmlist, devid_get_minor_name, devid_sizeof, devid_str_decode, devid_str_free, devid_str_encode, devid_valid – device ID interfaces for user applications

Synopsis

```
cc [ flag... ] file... -ldevid [ library... ]
#include <devid.h>

int devid_get(int fd, ddi_devid_t *retdevid);

void devid_free(ddi_devid_t devid);

int devid_get_minor_name(int fd, char **retminor_name);

int devid_deviceid_to_nmlist(char *search_path, ddi_devid_t devid,
     char *minor_name, devid_nmlist_t **retlist);

void devid_free_nmlist(devid_nmlist_t *list);

int devid_compare(ddi_devid_t devid1, ddi_devid_t devid2);

size_t devid_sizeof(ddi_devid_t devid);

int devid_valid(ddi_devid_t devid);

char *devid_str_encode(ddi_devid_t devid, char *minor_name);

int devid_str_decode(char *devidstr, ddi_devid_t *retdevid,
     char **retminor_name);

void devid_str_free(char *str);
```

Description These functions provide unique identifiers (device IDs) for devices. Applications and device drivers use these functions to identify and locate devices, independent of the device's physical connection or its logical device name or number.

The `devid_get()` function returns in *retdevid* the device ID for the device associated with the open file descriptor *fd*, which refers to any device. It returns an error if the device does not have an associated device ID. The caller must free the memory allocated for *retdevid* using the `devid_free()` function.

The `devid_free()` function frees the space that was allocated for the returned *devid* by `devid_get()` and `devid_str_decode()`.

The `devid_get_minor_name()` function returns the minor name, in *retminor_name*, for the device associated with the open file descriptor *fd*. This name is specific to the particular minor number, but is "instance number" specific. The caller of this function must free the memory allocated for the returned *retminor_name* string using `devid_str_free()`.

The `devid_deviceid_to_nmlist()` function returns an array of *devid_nmlist* structures, where each entry matches the *devid* and *minor_name* passed in. If the *minor_name* specified is one of the special values (`DEVID_MINOR_NAME_ALL`, `DEVID_MINOR_NAME_ALL_CHR`, or `DEVID_MINOR_NAME_ALL_BLK`) , then all minor names associated with *devid* which also meet

the special *minor_name* filtering requirements are returned. The *devid_nmlist* structure contains the device name and device number. The last entry of the array contains a null pointer for the *devname* and `NODEV` for the device number. This function traverses the file tree, starting at *search_path*. For each device with a matching device ID and minor name tuple, a device name and device number are added to the *retlist*. If no matches are found, an error is returned. The caller of this function must free the memory allocated for the returned array with the `devid_free_nmlist()` function. This function may take a long time to complete if called with the device ID of an unattached device.

The `devid_free_nmlist()` function frees the memory allocated by the `devid_deviceid_to_nmlist()` function.

The `devid_compare()` function compares two device IDs and determines both equality and sort order. The function returns an integer greater than 0 if the device ID pointed to by *devid1* is greater than the device ID pointed to by *devid2*. It returns 0 if the device ID pointed to by *devid1* is equal to the device ID pointed to by *devid2*. It returns an integer less than 0 if the device ID pointed to by *devid1* is less than the device ID pointed to by *devid2*. This function is the only valid mechanism to determine the equality of two devids. This function may indicate equality for arguments which by simple inspection appear different.

The `devid_sizeof()` function returns the size of *devid* in bytes.

The `devid_valid()` function validates the format of a *devid*. It returns 1 if the format is valid, and 0 if invalid. This check may not be as complete as the corresponding kernel function `ddi_devid_valid()` (see `ddi_devid_compare(9F)`).

The `devid_str_encode()` function encodes a *devid* and *minor_name* into a null-terminated ASCII string, returning a pointer to that string. To avoid shell conflicts, the *devid* portion of the string is limited to uppercase and lowercase letters, digits, and the plus (+), minus (-), period (.), equals (=), underscore (_), tilde (~), and comma (,) characters. If there is an ASCII quote character in the binary form of a *devid*, the string representation will be in `hex_id` form, not `ascii_id` form. The comma (,) character is added for "id1," at the head of the string *devid*. If both a *devid* and a *minor_name* are non-null, a slash (/)is used to separate the *devid* from the *minor_name* in the encoded string. If *minor_name* is null, only the *devid* is encoded. If the *devid* is null then the special string "id0" is returned. Note that you cannot compare the returned string against another string with `strcmp(3C)` to determine devid equality. The string returned must be freed by calling `devid_str_free()`.

The `devid_str_decode()` function takes a string previously produced by the `devid_str_encode()` or `ddi_devid_str_encode()` (see `ddi_devid_compare(9F)`) function and decodes the contained device ID and minor name, allocating and returning pointers to the extracted parts via the *retdevid* and *retminor_name* arguments. If the special *devidstr* "id0" was specified, the returned device ID and minor name will both be null. A non-null returned devid must be freed by the caller by the `devid_free()` function. A non-null returned minor name must be freed by calling `devid_str_free()`.

The `devid_str_free()` function frees the character string returned by `devid_str_encode()` and the *retminor_name* argument returned by `devid_str_decode()`.

Return Values Upon successful completion, the `devid_get()`, `devid_get_minor_name()`, `devid_str_decode()`, and `devid_deviceid_to_nmlist()` functions return `0`. Otherwise, they return −1.

The `devid_compare()` function returns the following values:

−1 The device ID pointed to by *devid1* is less than the device ID pointed to by *devid2*.

0 The device ID pointed to by *devid1* is equal to the device ID pointed to by *devid2*.

1 The device ID pointed to by *devid1* is greater than the device ID pointed to by *devid2*.

The `devid_sizeof()` function returns the size of *devid* in bytes. If *devid* is null, the number of bytes that must be allocated and initialized to determine the size of a complete device ID is returned.

The `devid_valid()` function returns 1 if the *devid* is valid and 0 if the *devid* is invalid.

The `devid_str_encode()` function returns `NULL` to indicate failure. Failure may be caused by attempting to encode an invalid string. If the return value is non-null, the caller must free the returned string by using the `devid_str_free()` function.

Examples **EXAMPLE 1** Using `devid_get()`, `devid_get_minor_name()`, and `devid_str_encode()`

The following example shows the proper use of `devid_get()`, `devid_get_minor_name()`, and `devid_str_encode()` to free the space allocated for *devid*, *minor_name* and encoded *devid*.

```
int fd;
ddi_devid_t    devid;
char        *minor_name, *devidstr;
if ((fd = open("/dev/dsk/c0t3d0s0", O_RDONLY|O_NDELAY)) < 0) {
    ...
}
if (devid_get(fd, &devid) != 0) {
    ...
}
if (devid_get_minor_name(fd, &minor_name) != 0) {
    ...
}
if ((devidstr = devid_str_encode(devid, minor_name)) == 0) {
    ...
}
printf("devid %s\n", devidstr);
devid_str_free(devidstr);
devid_free(devid);
devid_str_free(minor_name);
```

EXAMPLE 2 Using `devid_deviceid_to_nmlist()` and `devid_free_nmlist()`

The following example shows the proper use of `devid_deviceid_to_nmlist()` and `devid_free_nmlist()`:

```
devid_nmlist_t  *list = NULL;
int    err;
if (devid_deviceid_to_nmlist("/dev/rdsk", devid,
    minor_name, &list))
        return (-1);
/* loop through list and process device names and numbers */
devid_free_nmlist(list);
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT–Safe
Interface Stability	Stable

See Also

`free(3C)`, `libdevid(3LIB)`, `attributes(5)`, `ddi_devid_compare(9F)`

Name di_binding_name, di_bus_addr, di_compatible_names, di_devid, di_driver_name, di_driver_ops, di_driver_major, di_instance, di_nodeid, di_node_name – return libdevinfo node information

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

char *di_binding_name(di_node_t node);

char *di_bus_addr(di_node_t node);

int di_compatible_names(di_node_t node, char **names);

ddi_devid_t di_devid(di_node_t node);

char *di_driver_name(di_node_t node);

uint_t di_driver_ops(di_node_t node);

int di_driver_major(di_node_t node);

int di_instance(di_node_t node);

int di_nodeid(di_node_t node);

char *di_node_name(di_node_t node);
```

Parameters

names The address of a pointer.

node A handle to a device node.

Description These functions extract information associated with a device node.

Return Values The `di_binding_name()` function returns a pointer to the binding name. The binding name is the name used by the system to select a driver for the device.

The `di_bus_addr()` function returns a pointer to a null-terminated string containing the assigned bus address for the device. `NULL` is returned if a bus address has not been assigned to the device. A zero-length string may be returned and is considered a valid bus address.

The return value of `di_compatible_names()` is the number of compatible names. *names* is updated to point to a buffer contained within the snapshot. The buffer contains a concatenation of null-terminated strings, for example:

<*name1*>`/0`<*name2*>`/0...`<*namen*>`/0`

See the discussion of generic names in *Writing Device Drivers* for a description of how compatible names are used by Solaris to achieve driver binding for the node.

The `di_devid()` function returns the device ID for *node*, if it is registered. Otherwise, a null pointer is returned. Interfaces in the `libdevid(3LIB)` library may be used to manipulate the handle to the device id. This function is obsolete and might be removed from a future Solaris release. Applications should use the "devid" property instead.

The `di_driver_name()` function returns the name of the driver bound to the *node*. A null pointer is returned if *node* is not bound to any driver.

The `di_driver_ops()` function returns a bit array of device driver entry points that are supported by the driver bound to this *node*. Possible bit fields supported by the driver are `DI_CB_OPS`, `DI_BUS_OPS`, `DI_STREAM_OPS`.

The `di_driver_major()` function returns the major number associated with the driver bound to *node*. If there is no driver bound to the node, this function returns –1.

The `di_instance()` function returns the instance number of the device. A value of `-1` indicates an instance number has not been assigned to the device by the system.

The `di_nodeid()` function returns the type of device, which may be one of the following possible values: `DI_PSEUDO_NODEID`, `DI_PROM_NODEID`, and `DI_SID_NODEID`. Devices of type `DI_PROM_NODEID` may have additional properties that are defined by the PROM. See `di_prom_prop_data(3DEVINFO)` and `di_prom_prop_lookup_bytes(3DEVINFO)`.

The `di_node_name()` function returns a pointer to a null-terminated string containing the node name.

Examples See `di_init(3DEVINFO)` for an example demonstrating typical use of these functions.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving (`di_devid()` is obsolete)
MT-Level	Safe

See Also `di_init(3DEVINFO)`, `di_prom_init(3DEVINFO)`, `di_prom_prop_data(3DEVINFO)`, `di_prom_prop_lookup_bytes(3DEVINFO)`, `libdevid(3LIB)`, `libdevinfo(3LIB)`, `attributes(5)`

Writing Device Drivers

Name di_child_node, di_parent_node, di_sibling_node, di_drv_first_node, di_drv_next_node – libdevinfo node traversal functions

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

di_node_t di_child_node(di_node_t node);

di_node_t di_parent_node(di_node_t node);

di_node_t di_sibling_node(di_node_t node);

di_node_t di_drv_first_node(const char *drv_name, di_node_t root);

di_node_t di_drv_next_node(di_node_t node);
```

Parameters

drv_name The name of the driver of interest.

node A handle to any node in the snapshot.

root The handle of the root node for the snapshot returned by `di_init`(3DEVINFO).

Description The kernel device configuration data may be viewed in two ways, either as a tree of device configuration nodes or as a list of nodes associated with each driver. In the tree view, each node may contain references to its parent, the next sibling in a list of siblings, and the first child of a list of children. In the per-driver view, each node contains a reference to the next node associated with the same driver. Both views are captured in the snapshot, and the interfaces are provided for node access.

The `di_child_node()` function obtains a handle to the first child of *node*. If no child node exists in the snapshot, `DI_NODE_NIL` is returned and `errno` is set to `ENXIO` or `ENOTSUP`.

The `di_parent_node()` function obtains a handle to the parent node of *node*. If no parent node exists in the snapshot, `DI_NODE_NIL` is returned and `errno` is set to `ENXIO` or `ENOTSUP`.

The `di_sibling_node()` function obtains a handle to the next sibling node of *node*. If no next sibling node exists in the snapshot, `DI_NODE_NIL` is returned and `errno` is set to `ENXIO` or `ENOTSUP`.

The `di_drv_first_node()` function obtains a handle to the first node associated with the driver specified by `drv_name`. If there is no such driver, `DI_NODE_NIL` is returned with `errno` is set to `EINVAL`. If the driver exists but there is no node associated with this driver, `DI_NODE_NIL` is returned and `errno` is set to `ENXIO` or `ENOTSUP`.

The `di_drv_next_node()` function returns a handle to the next node bound to the same driver. If no more nodes exist, `DI_NODE_NIL` is returned.

Return Values Upon successful completion, a handle is returned. Otherwise, `DI_NODE_NIL` is returned and `errno` is set to indicate the error.

Errors These functions will fail if:

`EINVAL` The argument is invalid.

`ENXIO` The requested node does not exist.

`ENOTSUP` The node was not found in the snapshot, but it may exist in the kernel. This error may occur if the snapshot contains a partial device tree.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Safe

See Also `di_init(3DEVINFO)`, `libdevinfo(3LIB)`, `attributes(5)`

Writing Device Drivers

Name di_devfs_path, di_devfs_minor_path, di_devfs_path_free – generate and free physical path names

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

char *di_devfs_path(di_node_t node);

char *di_devfs_minor_path(di_minor_t minor);

void di_devfs_path_free(char *path_buf);
```

Parameters

node The handle to a device node in the snapshot.

minor The handle to a device minor node in the snapshot.

path_buf A pointer returned by `di_devfs_path()` or `di_devfs_minor_path()`.

Description The `di_devfs_path()` function generates the physical path of the device node specified by *node*.

The `di_devfs_minor_path()` function generates the physical path of the device minor node specified by *minor*.

The `di_devfs_path_free()` function frees memory that was allocated to store the physical path by `di_devfs_path()` and `di_devfs_minor_path()`. The caller of `di_devfs_path()` and `di_devfs_minor_path()` is responsible for freeing this memory allocated by calling `di_devfs_path_free()`.

Return Values Upon successful completion, the `di_devfs_path()` and `di_devfs_minor_path()` functions return a pointer to the string containing the physical path of a device node or a device minor node, respectively. Otherwise, they return `NULL` and `errno` is set to indicate the error.

Errors The `di_devfs_path()` and `di_devfs_minor_path()` functions will fail if:

`EINVAL` The *node* or *minor* argument is not a valid handle.

The `di_devfs_path()` and `di_devfs_minor_path()` functions can also return any error value returned by `malloc(3C)`.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Safe

See Also `di_init`(3DEVINFO), `libdevinfo`(3LIB), `malloc`(3C), `attributes`(5)

Writing Device Drivers

Name di_init, di_fini – create and destroy a snapshot of kernel device tree

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

di_node_t di_init(const char *phys_path, uint_t flags);

void di_fini(di_node_t root);
```

Parameters

flags Snapshot content specification. The possible values can be a bitwise OR of at least one of the following:

`DINFOSUBTREE`	Include subtree.
`DINFOPROP`	Include properties.
`DINFOMINOR`	Include minor data.
`DINFOCPYALL`	Include all of the above.
`DINFOLYR`	Include device layering data.
`DINFOCPYONE`	Include only a single node without properties or minor nodes.

phys_path Physical path of the *root* node of the snapshot. See `di_devfs_path`(3DEVINFO).

root Handle obtained by calling `di_init()`.

Description The `di_init()` function creates a snapshot of the kernel device tree and returns a handle of the *root* node. The caller specifies the contents of the snapshot by providing *flag* and *phys_path*.

The `di_fini()` function destroys the snapshot of the kernel device tree and frees the associated memory. All handles associated with this snapshot become invalid after the call to `di_fini()`.

Return Values Upon success, `di_init()` returns a handle. Otherwise, `DI_NODE_NIL` is returned and `errno` is set to indicate the error.

Errors The `di_init()` function can set `errno` to any error code that can also be set by `open`(2), `ioctl`(2) or `mmap`(2). The most common error codes include:

`EACCES`	Insufficient privilege for accessing device configuration data.
`ENXIO`	Either the device named by *phys_path* is not present in the system, or the `devinfo`(7D) driver is not installed properly.
`EINVAL`	Either *phys_path* is incorrectly formed or the *flags* argument is invalid.

Examples

EXAMPLE 1 Using the `libdevinfo` Interfaces To Print All Device Tree Node Names

The following is an example using the `libdevinfo` interfaces to print all device tree node names:

```
/*
 * Code to print all device tree node names
 */

#include <stdio.h>
#include <libdevinfo.h>

int
prt_nodename(di_node_t node, void *arg)
{
     printf("%s\n", di_node_name(node));
     return (DI_WALK_CONTINUE);
}

main()
{
     di_node_t root_node;
     if((root_node = di_init("/", DINFOSUBTREE)) == DI_NODE_NIL) {
           fprintf(stderr, "di_init() failed\n");
           exit(1);
     }
     di_walk_node(root_node, DI_WALK_CLDFIRST, NULL, prt_nodename);
     di_fini(root_node);
}
```

EXAMPLE 2 Using the `libdevinfo` Interfaces To Print The Physical Path Of SCSI Disks

The following example uses the `libdevinfo` interfaces to print the physical path of SCSI disks:

```
/*
 * Code to print physical path of scsi disks
 */

#include <stdio.h>
#include <libdevinfo.h>
#define    DISK_DRIVER    "sd"    /* driver name */

void
prt_diskinfo(di_node_t node)
{
    int instance;
        char *phys_path;

    /*
```

EXAMPLE 2 Using the `libdevinfo` Interfaces To Print The Physical Path Of SCSI Disks *(Continued)*

```
     * If the device node exports no minor nodes,
     * there is no physical disk.
     */
     if (di_minor_next(node, DI_MINOR_NIL) == DI_MINOR_NIL) {
              return;
         }

         instance = di_instance(node);
         phys_path = di_devfs_path(node);
         printf("%s%d: %s\n", DISK_DRIVER, instance, phys_path);
         di_devfs_path_free(phys_path);
}

void
walk_disknodes(di_node_t node)
{
        node = di_drv_first_node(DISK_DRIVER, node);
        while (node != DI_NODE_NIL) {
             prt_diskinfo(node);
             node = di_drv_next_node(node);
        }
}

main()
{
    di_node_t root_node;
    if ((root_node = di_init("/", DINFOCPYALL)) == DI_NODE_NIL) {
        fprintf(stderr, "di_init() failed\n");
        exit(1);
    }
        walk_disknodes(root_node);
        di_fini(root_node);
}
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also

`open(2)`, `ioctl(2)`, `mmap(2)`, `libdevinfo(3LIB)`, `attributes(5)`

Writing Device Drivers

Name di_link_next_by_node, di_link_next_by_lnode – libdevinfo link traversal functions

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

di_link_t di_link_next_by_node(di_lnode_t node, di_link_t link,
     uint_t endpoint);

di_link_t di_link_next_by_lnode(di_node_t lnode, di_link_t link,
     uint_t endpoint);
```

Parameters

link The handle to the current the link or `DI_LINK_NIL`.

endpoint Specify which endpoint of the link the node or lnode should correspond to, either `DI_LINK_TGT` or `DI_LINK_SRC`.

node The device node with which the link is associated.

lnode The lnode with which the link is associated.

Description The `di_link_next_by_node()` function returns a handle to the next link that has the same endpoint node as *link*. If *link* is `DI_LINK_NIL`, a handle is returned to the first link whose endpoint specified by *endpoint* matches the node specified by *node*.

The `di_link_next_by_lnode()` function returns a handle to the next link that has the same endpoint lnode as *link*. If *link* is `DI_LINK_NIL`, a handle is returned to the first link whose endpoint specified by *endpoint* matches the lnode specified by *lnode*.

Return Values Upon successful completion, a handle to the next link is returned. Otherwise, `DI_LINK_NIL` is returned and `errno` is set to indicate the error.

Errors The `di_link_next_by_node()` and `di_link_next_by_lnode()` functions will fail if:

`EINVAL` An argument is invalid.

`ENXIO` The end of the link list has been reached.

The `di_link_next_by_node()` function will fail if:

`ENOTSUP` Device usage information is not available in snapshot.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Safe

See Also di_init(3DEVINFO), libdevinfo(3LIB), attributes(5)

Name di_link_spectype, di_link_to_lnode – return libdevinfo link information

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

int di_link_spectype(di_link_t link);

di_lnode_t di_link_to_lnode(di_link_t link, uint_t endpoint);
```

Parameters

link A handle to a link.

endpoint specifies the endpoint of the link, which should correspond to either `DI_LINK_TGT` or `DI_LINK_SRC`

Description The `di_link_spectype()` function returns libdevinfo link information.

The `di_link_to_lnode()` function takes a link specified by *link* and returns the lnode corresponding to the link endpoint specified by *endpoint*.

Return Values The `di_link_spectype()` function returns the spectype parameter flag that was used to open the target device of a link, either `S_IFCHR` or `S_IFBLK`.

Upon successful completion, `di_link_to_lnode()` returns a handle to an lnode. Otherwise, `DI_LINK_NIL` is returned and `errno` is set to indicate the error.

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

`EINVAL` An argument is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Safe

See Also `di_init(3DEVINFO)`, `libdevinfo(3LIB)`, `attributes(5)`

Name di_lnode_name, di_lnode_devinfo, di_lnode_devt – return libdevinfo lnode information

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

char *di_lnode_name(di_lnode_t lnode);

di_node_t di_lnode_devinfo(di_lnode_t lnode);

int di_lnode_devt(di_lnode_t lnode, dev_t *devt);
```

Parameters

lnode A handle to an lnode.

devt A pointer to a `dev_t` that can be returned.

Description These functions return `libdevinfo` lnode information.

The `di_lnode_name()` function returns a pointer to the name associated with *lnode*.

The `di_lnode_devinfo()` function returns a handle to the device node associated with *lnode*.

The `di_lnode_devt()` function sets the `dev_t` pointed to by the *devt* parameter to the `dev_t` associated with *lnode*.

Return Values The `di_lnode_name()` function returns a pointer to the name associated with *lnode*.

The `di_lnode_devinfo()` function returns a handle to the device node associated with *lnode*.

The `di_lnode_devt()` function returns 0 if the requested attribute exists in *lnode* and was returned. It returns −1 if the requested attribute does not exist and sets `errno` to indicate the error.

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

`EINVAL` An argument was invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Safe

See Also `di_init(3DEVINFO)`, `libdevinfo(3LIB)`, `attributes(5)`

Name di_lnode_next – libdevinfo lnode traversal function

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

di_lnode_t di_lnode_next(di_node_t node, di_lnode_t lnode);
```

Parameters

node A handle to a di_node.

lnode A handle to an lnode.

Description The `di_lnode_next()` function returns a handle to the next lnode for the device node specified by *node*. If *lnode* is `DI_LNODE_NIL`, a handle to the first lnode is returned.

Return Values Upon successful completion, a handle to an lnode is returned. Otherwise, `DI_LNODE_NIL` is returned and `errno` is set to indicate the error.

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

`EINVAL` An argument is invalid.

`ENOTSUP` Device usage information is not available in snapshot.

`ENXIO` The end of the lnode list has been reached.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Safe

See Also `di_init(3DEVINFO)`, `libdevinfo(3LIB)`, `attributes(5)`

Name di_lnode_private_set, di_lnode_private_get, di_minor_private_set, di_minor_private_get, di_node_private_set, di_node_private_get, di_link_private_set, di_link_private_get – manipulate libdevinfo user traversal pointers

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

void di_lnode_private_set(di_lnode_t lnode, void *data);

void *di_lnode_private_get(di_lnode_t lnode);

void di_minor_private_set(di_minor_t minor, void *data);

void *di_minor_private_get(di_minor_t minor);

void di_node_private_set(di_node_t node, void *data);

void *di_node_private_get(di_node_t node);

void di_link_private_set(di_link_t link, void *data);

void *di_link_private_get(di_link_t link);
```

Parameters

lnode A handle to an lnode.

minor A handle to a minor node.

node A handle to a devinfo node.

link A handle to a link.

data A pointer to caller-specific data.

Description The `di_lnode_private_set()` function allows a caller to associate caller-specific data pointed to by *data* with an lnode specified by *lnode*, thereby facilitating traversal of lnodes in the snapshot.

The `di_lnode_private_get()` function allows a caller to retrieve a data pointer that was associated with an lnode by a call to `di_lnode_private_set()`.

The `di_minor_private_set()` function allows a caller to associate caller-specific data pointed to by *data* with a minor node specified by *minor*, thereby facilitating traversal of minor nodes in the snapshot.

The `di_minor_private_get()` function allows a caller to retrieve a data pointer that was associated with a minor node obtained by a call to `di_minor_private_set()`.

The `di_node_private_set()` function allows a caller to associate caller-specific data pointed to by *data* with a devinfo node, thereby facilitating traversal of devinfo nodes in the snapshot.

The `di_node_private_get()` function allows a caller to retrieve a data pointer that was associated with a devinfo node obtained by a call to `di_node_private_set()`.

The `di_link_private_set()` function allows a caller to associate caller-specific data pointed to by *data* with a link, thereby facilitating traversal of links in the snapshot.

The `di_link_private_get()` function allows a caller to retrieve a data pointer that was associated with a link obtained by a call to `di_link_private_set()`.

These functions do not perform any type of locking. It is up to the caller to satisfy any locking needs.

Return Values The `di_lnode_private_set()`, `di_minor_private_set()`, `di_node_private_set()`, and `di_link_private_set()` functions do not return values.

The `di_lnode_private_get()`, `di_minor_private_get()`, `di_node_private_get()`, and `di_node_private_get()` functions return a pointer to caller-specific data that was initialized with their corresponding set function. If no caller-specific data was assigned with a set function, the results are undefined.

Errors No errors are defined.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Safe

See Also `di_init(3DEVINFO)`, `libdevinfo(3LIB)`, `attributes(5)`

Name di_minor_devt, di_minor_name, di_minor_nodetype, di_minor_spectype – return libdevinfo minor node information

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

dev_t di_minor_devt(di_minor_t minor);

char *di_minor_name(di_minor_t minor);

char *di_minor_nodetype(di_minor_t minor);

int di_minor_spectype(di_minor_t minor);
```

Parameters *minor* A handle to minor data node.

Description These functions return `libdevinfo` minor node information.

Return Values The `di_minor_name()` function returns the minor *name*. See `ddi_create_minor_node(9F)` for a description of the *name* parameter.

The `di_minor_devt()` function returns the `dev_t` value of the minor node that is specified by SYS V ABI. See `getmajor(9F)`, `getminor(9F)`, and `ddi_create_minor_node(9F)` for more information.

The `di_minor_spectype()` function returns the *spec_type* of the file, either `S_IFCHR` or `S_IFBLK`. See `ddi_create_minor_node(9F)` for a description of the *spec_type* parameter.

The `di_minor_nodetype()` function returns the minor *node_type* of the minor node. See `ddi_create_minor_node(9F)` for a description of the *node_type* parameter.

Errors No errors are defined.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Safe

See Also `attributes(5)`, `ddi_create_minor_node(9F)`, `getmajor(9F)`, `getminor(9F)`

Writing Device Drivers

Name di_minor_next – libdevinfo minor node traversal functions

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

di_minor_t di_minor_next(di_node_t node, di_minor_t minor);
```

Parameters

minor Handle to the current minor node or `DI_MINOR_NIL`.

node Device node with which the minor node is associated.

Description The `di_minor_next()` function returns a handle to the next minor node for the device node *node*. If *minor* is `DI_MINOR_NIL`, a handle to the first minor node is returned.

Return Values Upon successful completion, a handle to the next minor node is returned. Otherwise, `DI_MINOR_NIL` is returned and `errno` is set to indicate the error.

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

`EINVAL` Invalid argument.

`ENOTSUP` Minor node information is not available in snapshot.

`ENXIO` End of minor node list.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Safe

See Also `libdevinfo(3LIB)`, `attributes(5)`

Writing Device Drivers

Name di_prom_init, di_prom_fini – create and destroy a handle to the PROM device information

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

di_prom_handle_t di_prom_init(void);

void di_prom_fini(di_prom_handle_t  ph);
```

Parameters *ph* Handle to `prom` returned by `di_prom_init()`.

Description For device nodes whose `nodeid` value is `DI_PROM_NODEID` (see `di_nodeid(3DEVINFO)`), additional properties can be retrieved from the PROM. The `di_prom_init()` function returns a handle that is used to retrieve such properties. This handle is passed to `di_prom_prop_lookup_bytes(3DEVINFO)` and `di_prom_prop_next(3DEVINFO)`.

The `di_prom_fini()` function destroys the handle and all handles to the PROM device information obtained from that handle.

Return Values Upon successful completion, `di_prom_init()` returns a handle. Otherwise, `DI_PROM_HANDLE_NIL` is returned and `errno` is set to indicate the error.

Errors The `di_prom_init()` sets `errno` function to any error code that can also be set by `openprom(7D)` or `malloc(3C)`.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Safe

See Also `di_nodeid(3DEVINFO)`, `di_prom_prop_next(3DEVINFO)`, `di_prom_prop_lookup_bytes(3DEVINFO)`, `libdevinfo(3LIB)`, `malloc(3C)`, `attributes(5)`, `openprom(7D)`

Name di_prom_prop_data, di_prom_prop_next, di_prom_prop_name – access PROM device information

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

di_prom_prop_t di_prom_prop_next(di_prom_handle_t ph, di_node_t node,
     di_prom_prop_t prom_prop);

char *di_prom_prop_name(di_prom_prop_t prom_prop);

int di_prom_prop_data(di_prom_prop_t prom_prop, uchar_t **prop_data);
```

Parameters

node	Handle to a device node in the snapshot of kernel device tree.
ph	PROM handle
prom_prop	Handle to a PROM property.
prop_data	Address of a pointer.

Description The `di_prom_prop_next()` function obtains a handle to the next property on the PROM property list associated with *node*. If *prom_prop* is `DI_PROM_PROP_NIL`, the first property associated with *node* is returned.

The `di_prom_prop_name()` function returns the name of the *prom_prop* property.

The `di_prom_prop_data()` function returns the value of the *prom_prop* property. The return value is a non-negative integer specifying the size in number of bytes in *prop_data*.

All memory allocated by these functions is managed by the library and must not be freed by the caller.

Return Values The `di_prom_prop_data()` function returns the number of bytes in *prop_data* and *prop_data* is updated to point to a byte array containing the property value. If 0 is returned, the property is a boolean property and the existence of this property indicates the value is true.

The `di_prom_prop_name()` function returns a pointer to a string that contains the name of *prom_prop*.

The `di_prom_prop_next()` function returns a handle to the next PROM property. `DI_PROM_PROP_NIL` is returned if no additional properties exist.

Errors See `openprom(7D)` for a description of possible errors.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Safe

See Also attributes(5), openprom(7D)

Writing Device Drivers

Name di_prom_prop_lookup_bytes, di_prom_prop_lookup_ints, di_prom_prop_lookup_strings – search for a PROM property

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

int di_prom_prop_lookup_bytes(di_prom_handle_t ph, di_node_t node,
     const char *prop_name, uchar_t **prop_data);

int di_prom_prop_lookup_ints(di_prom_handle_t ph, di_node_t node,
     const char *prop_name, int **prop_data);

int di_prom_prop_lookup_strings(di_prom_handle_t ph, di_node_t node,
     const char *prop_name, char **prop_data);
```

Parameters

node Handle to device node in snapshot created by `di_init`(3DEVINFO).

ph Handle returned by `di_prom_init`(3DEVINFO).

prop_data For `di_prom_prop_lookup_bytes()`, the address of a pointer to an array of unsigned characters.

For `di_prom_prop_lookup_ints()`, the address of a pointer to an integer.

For `di_prom_prop_lookup_strings()`, the address of pointer to a buffer.

prop_name The name of the property being searched.

Description These functions return the value of a known PROM property name and value type and update the *prop_data* pointer to reference memory that contains the property value. All memory allocated by these functions is managed by the library and must not be freed by the caller.

Return Values If the property is found, the number of entries in *prop_data* is returned. If the property is a boolean type, 0 is returned and the existence of this property indicates the value is true. Otherwise, -1 is returned and `errno` is set to indicate the error.

For `di_prom_prop_lookup_bytes()`, the number of entries is the number of unsigned characters contained in the buffer pointed to by *prop_data*.

For `di_prom_prop_lookup_ints()`, the number of entries is the number of integers contained in the buffer pointed to by *prop_data*.

For `di_prom_prop_lookup_strings()`, the number of entries is the number of null-terminated strings contained in the buffer. The strings are stored in a concatenated format in the buffer.

Errors These functions will fail if::

`EINVAL` Invalid argument.

`ENXIO` The property does not exist.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Safe

See Also di_init(3DEVINFO), di_prom_prop_next(3DEVINFO), libdevinfo(3LIB), attributes(5), openprom(7D)

Writing Device Drivers

Name di_prop_bytes, di_prop_devt, di_prop_ints, di_prop_name, di_prop_strings, di_prop_type, di_prop_int64 – access property values and attributes

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

int di_prop_bytes(di_prop_t prop, uchar_t **prop_data);

dev_t di_prop_devt(di_prop_t prop);

int di_prop_ints(di_prop_t prop, int **prop_data);

int di_prop_int64(di_prop_t prop, int64_t **prop_data);

char *di_prop_name(di_prop_t prop);

int di_prop_strings(di_prop_t prop, char **prop_data);

int di_prop_type(di_prop_t prop);
```

Parameters

prop Handle to a property returned by `di_prop_next(3DEVINFO)`.

prop_data For `di_prop_bytes()`, the address of a pointer to an unsigned character.

For `di_prop_ints()`, the address of a pointer to an integer.

For `di_prop_int64()`, the address of a pointer to a 64–bit integer.

For `di_prop_strings()`, the address of pointer to a character.

Description These functions access information associated with property values and attributes. All memory allocated by these functions is managed by the library and must not be freed by the caller.

The `di_prop_bytes()` function returns the property data as a series of unsigned characters.

The `di_prop_devt()` function returns the `dev_t` with which this property is associated. If the value is `DDI_DEV_T_NONE`, the property is not associated with any specific minor node.

The `di_prop_ints()` function returns the property data as a series of integers.

The `di_prop_int64()` function returns the property data as a series of 64–bit integers.

The `di_prop_name()` function returns the name of the property.

The `di_prop_strings()` function returns the property data as a concatenation of null-terminated strings.

The `di_prop_type()` function returns the type of the property. The type determines the appropriate interface to access property values. The following is a list of possible types:

`DI_PROP_TYPE_BOOLEAN`	There is no interface to call since there is no property data associated with boolean properties. The existence of the property defines a `TRUE` value.
`DI_PROP_TYPE_INT`	Use `di_prop_ints()` to access property data.
`DI_PROP_TYPE_INT64`	Use `di_prop_int64()` to access property data.
`DI_PROP_TYPE_STRING`	Use `di_prop_strings()` to access property data.
`DI_PROP_TYPE_BYTE`	Use `di_prop_bytes()` to access property data.
`DI_PROP_TYPE_UNKNOWN`	Use `di_prop_bytes()` to access property data. Since the type of property is unknown, the caller is responsible for interpreting the contents of the data.
`DI_PROP_TYPE_UNDEF_IT`	The property has been undefined by the driver. No property data is available.

Return Values Upon successful completion, `di_prop_bytes()`, `di_prop_ints()`, `di_prop_int64()`, and `di_prop_strings()` return a non-negative value, indicating the number of entries in the property value buffer. See `di_prom_prop_lookup_bytes(3DEVINFO)` for a description of the return values. Otherwise, -1 is returned and `errno` is set to indicate the error.

The `di_prop_devt()` function returns the `dev_t` value associated with the property.

The `di_prop_name()` function returns a pointer to a string containing the name of the property.

The `di_prop_type()` function can return one of types described in the DESCRIPTION section.

Errors These functions will fail if:

`EINVAL` Invalid argument. For example, the property type does not match the interface.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Safe

See Also `di_prom_prop_lookup_bytes(3DEVINFO)`, `di_prop_next(3DEVINFO)`, `libdevinfo(3LIB)`, `attributes(5)`

Writing Device Drivers

Name di_prop_lookup_bytes, di_prop_lookup_ints, di_prop_lookup_int64, di_prop_lookup_strings – search for a property

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

int di_prop_lookup_bytes(dev_t dev, di_node_t node,
     const char *prop_name, uchar_t **prop_data);

int di_prop_lookup_ints(dev_t dev, di_node_t node,
     const char *prop_name, int **prop_data);

int di_prop_lookup_int64(dev_t dev, di_node_t node,
     const char *prop_name, int64_t **prop_data);

int di_prop_lookup_strings(dev_t dev, di_node_t node,
     const char *prop_name, char **prop_data);
```

Parameters

dev `dev_t` of minor node with which the property is associated. `DDI_DEV_T_ANY` is a wild card that matches all `dev_t`'s, including `DDI_DEV_T_NONE`.

node Handle to the device node with which the property is associated.

prop_data For `di_prop_lookup_bytes()`, the address to a pointer to an array of unsigned characters containing the property data.

For `di_prop_lookup_ints()`, the address to a pointer to an array of integers containing the property data.

For `di_prop_lookup_int64()`, the address to a pointer to an array of 64–bit integers containing the property data.

For `di_prop_lookup_strings()`, the address to a pointer to a buffer containing a concatenation of null-terminated strings containing the property data.

prop_name Name of the property for which to search.

Description These functions return the value of a known property name type and `dev_t` value. All memory allocated by these functions is managed by the library and must not be freed by the caller.

Return Values If the property is found, the number of entries in *prop_data* is returned. If the property is a boolean type, 0 is returned and the existence of this property indicates the value is true. Otherwise, -1 is returned and `errno` is set to indicate the error.

Errors These functions will fail if:

`EINVAL` Invalid argument.

`ENOTSUP` The snapshot contains no property information.

`ENXIO` The property does not exist; try `di_prom_prop_lookup_*()`.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Safe

See Also di_init(3DEVINFO), di_prom_prop_lookup_bytes(3DEVINFO), libdevinfo(3LIB), attributes(5)

Writing Device Drivers

Name di_prop_next – libdevinfo property traversal function

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

di_prop_t di_prop_next(di_node_t node, di_prop_t prop);
```

Parameters

node Handle to a device node.

prop Handle to a property.

Description The `di_prop_next()` function returns a handle to the next property on the property list. If *prop* is `DI_PROP_NIL`, the handle to the first property is returned.

Return Values Upon successful completion, `di_prop_next()` returns a handle. Otherwise `DI_PROP_NIL` is returned and `errno` is set to indicate the error.

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

`EINVAL` Invalid argument.

`ENOTSUP` The snapshot does not contain property information.

`ENXIO` There are no more properties.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Safe

See Also `di_init(3DEVINFO)`, `libdevinfo(3LIB)`, `attributes(5)`

Writing Device Drivers

Name DisconnectToServer – disconnect from a DMI service provider

Synopsis

```
cc [ flag ... ] file ... -ldmici  -ldmimi    [ library ... ]
#include <dmi/api.hh>

bool_t DisconnectToServer(DmiRpcHandle *dmi_rpc_handle);
```

Description The `DisconnectToServer()` function disconnects a management application or a component instrumentation from a DMI service provider.

Return Values The `ConnectToServer()` function returns TRUE if successful, otherwise FALSE.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-level	Safe

See Also `ConnectToServer(3DMI)`,`attributes(5)`

Name di_walk_link – traverse libdevinfo links

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

int di_walk_link(di_node_t root, uint_t flag, uint_t endpoint, void *arg,
     int (*link_callback)(di_link_t link, void *arg));
```

Parameters

root	The handle to the root node of the subtree to visit.
flag	Specify 0. Reserved for future use.
endpoint	Specify if the current node being visited should be the target or source of an link, either `DI_LINK_TGT` or `DI_LINK_SRC`
arg	A pointer to caller-specific data.
link_callback	The caller-supplied callback function.

Description The `di_walk_link()` function visits all nodes in the subtree rooted at *root*. For each node found, the caller-supplied function *link_callback*() is invoked for each link associated with that node where that node is the specified *endpoint* of the link. The return value of *link_callback*() specifies subsequent walking behavior. See RETURN VALUES.

Return Values Upon successful completion, `di_walk_link()` returns 0. Otherwise, -1 is returned and `errno` is set to indicate the error.

The callback function, *link_callback*(), can return one of the following:

`DI_WALK_CONTINUE`	Continue walking.
`DI_WALK_TERMINATE`	Terminate the walk immediately.

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

`EINVAL`	An argument is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Safe

See Also `di_init(3DEVINFO)`, `libdevinfo(3LIB)`, `attributes(5)`

Name di_walk_lnode – traverse libdevinfo lnodes

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

int di_walk_lnode(di_node_t root, uint_t flag, void *arg,
     int (*lnode_callback)(di_lnode_t link, void *arg));
```

Parameters

root	The handle to the root node of the subtree to visit.
flag	Specify 0. Reserved for future use.
arg	A pointer to caller-specific data.
lnode_callback	The caller-supplied callback function.

Description The `di_walk_lnode()` function visits all nodes in the subtree rooted at *root*. For each node found, the caller-supplied function *lnode_callback*() is invoked for each lnode associated with that node. The return value of *lnode_callback*() specifies subsequent walking behavior where that node is the specified *endpoint* of the link.

Return Values Upon successful completion, `di_walk_lnode()` returns 0. Otherwise, -1 is returned and `errno` is set to indicate the error.

The callback function *lnode_callback*() can return one of the following:

`DI_WALK_CONTINUE`	Continue walking.
`DI_WALK_TERMINATE`	Terminate the walk immediately.

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

`EINVAL`	An argument is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Safe

See Also `di_init(3DEVINFO)`, `libdevinfo(3LIB)`, `attributes(5)`

Name di_walk_minor – traverse libdevinfo minor nodes

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

int di_walk_minor(di_node_t root, const char *minor_nodetype,
     uint_t flag, void *arg, int (*minor_callback)di_node_t node,
     di_minor_t minor, void *arg);
```

Parameters

arg	Pointer to caller– specific user data.
flag	Specify 0. Reserved for future use.
minor	The minor node visited.
minor_nodetype	A character string specifying the minor data type, which may be one of the types defined by the Solaris DDI framework, for example, `DDI_NT_BLOCK`. `NULL` matches all *minor_node* types. See `ddi_create_minor_node(9F)`.
node	The device node with which to the minor node is associated.
root	Root of subtree to visit.

Description The `di_walk_minor()` function visits all minor nodes attached to device nodes in a subtree rooted at *root*. For each minor node that matches *minor_nodetype*, the caller-supplied function *minor_callback*() is invoked. The walk terminates immediately when *minor_callback*() returns `DI_WALK_TERMINATE`.

Return Values Upon successful completion, `di_walk_minor()` returns 0. Otherwise, -1 is returned and errno is set to indicate the error.

The `minor_callback()` function returns one of the following:

`DI_WALK_CONTINUE`	Continue to visit subsequent minor data nodes.
`DI_WALK_TERMINATE`	Terminate the walk immediately.

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

`EINVAL`	Invalid argument.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Safe

See Also `di_minor_nodetype`(3DEVINFO), `libdevinfo`(3LIB), `attributes`(5), `ddi_create_minor_node`(9F)

Writing Device Drivers

Name di_walk_node – traverse libdevinfo device nodes

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

int di_walk_node(di_node_t root, uint_t flag, void *arg,
     int (*node_callback)di_node_t node, void *arg);
```

Description The `di_walk_node()` function visits all nodes in the subtree rooted at *root*. For each node found, the caller-supplied function *node_callback*() is invoked. The return value of *node_callback*() specifies subsequent walking behavior.

Parameters

arg Pointer to caller–specific data.

flag Specifies walking order, either `DI_WALK_CLDFIRST` (depth first) or `DI_WALK_SIBFIRST` (breadth first). `DI_WALK_CLDFIRST` is the default.

node The node being visited.

root The handle to the root node of the subtree to visit.

Return Values Upon successful completion, `di_walk_node()` returns 0. Otherwise, `-1` is returned and `errno` is set to indicate the error.

The *node_callback*() function can return one of the following:

`DI_WALK_CONTINUE` Continue walking.

`DI_WALK_PRUNESIB` Continue walking, but skip siblings and their child nodes.

`DI_WALK_PRUNECHILD` Continue walking, but skip subtree rooted at current node.

`DI_WALK_TERMINATE` Terminate the walk immediately.

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

`EINVAL` Invalid argument.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Safe

See Also `di_init(3DEVINFO)`, `libdevinfo(3LIB)`, `attributes(5)`

Writing Device Drivers

Name DmiAddComponent, DmiAddGroup, DmiAddLanguage, DmiDeleteComponent, DmiDeleteGroup, DmiDeleteLanguage – Management Interface database administration functions

Synopsis

```
cc [ flag ... ] file ... -ldmimi  -ldmi   -lnsl   -lrwtool  [ library ... ]
#include <dmi/server.h>
#include <dmi/miapi.h>

bool_t DmiAddComponent(DmiAddComponentIN argin, DmiAddComponentOUT *result,
     DmiRpcHandle *dmi_rpc_handle);

bool_t DmiAddGroup(DmiAddGroupIN argin, DmiAddGroupOUT *result, DmiRpcHandle *dmi_rpc_handle);

bool_t DmiAddLanguage(DmiAddLanguageIN argin, DmiAddLanguageOUT*result,
     DmiRpcHandle *dmi_rpc_handle);

bool_t DmiDeleteComponent(DmiDeleteComponentIN argin, DmiDeleteComponentOUT *result,
     DmiRpcHandle *dmi_rpc_handle);

bool_t DmiDeleteGroup(DmiDeleteGroupIN argin, DmiDeleteGroupOUT *result,
     DmiRpcHandle *dmi_rpc_handle);

bool_t DmiDeleteLanguage(DmiDeleteLanguageIN argin, DmiDeleteLanguageOUT *result,
     DmiRpcHandle *dmi_rpc_handle);
```

Description The database administration functions add a new component to the database or add a new language mapping for an existing component. You may also remove an existing component, remove a specific language mapping, or remove a group from a component.

The `DmiAddComponent()` function adds a new component to the DMI database. It takes the name of a file, or the address of memory block containing MIF data, checks the data for adherence to the DMI MIF grammar, and installs the MIF in the database. The procedure returns a unique component ID for the newly installed component. The *argin* parameter is an instance of a `DmiAddComponentIN` structure containing the following members:

```
DmiHandle_t        handle;         /* an open session handle */
DmiFileDataList_t  *fileData;      /* MIF data for component */
```

The *result* parameter is a pointer to a `DmiAddComponentOUT` structure containing the following members:

```
DmiErrorStatus_t   error_status;
DmiId_t            compId;         /* SP-allocated component ID */
DmiStringList_t    *errors;        /* installation error messages */
```

The `DmiAddLanguage()` function adds a new language mapping for an existing component in the database. It takes the name of a file, or the address of memory block containing translated MIF data, checks the data for adherence to the DMI MIF grammar, and installs the language MIF in the database. The *argin* parameter is an instance of a `DmiAddLanguageIN` structure containing the following members:

```
DmiHandle_t        handle;         /* an open session handle */
DmiFileDataList_t  *fileData;      /* language mapping file */
DmiId_t            compId;         /* component to access */
```

The *result* parameter is a pointer to a `DmiAddLanguageOUT` structure containing the following members:

```
DmiErrorStatus_t   error_status;
DmiStringList_t    *errors;        /* installation error messages */
```

The `DmiAddGroup()` function adds a new group to an existing component in the database. It takes the name of a file, or the address of memory block containing the group's MIF data, checks the data for adherence to the DMI MIF grammar, and installs the group MIF in the database. The *argin* parameter is an instance of a `DmiAddGroupIN` structure containing the following members:

```
DmiHandle_t        handle;         /* an open session handle */
DmiFileDataList_t  *fileData;      /* MIF file data for group */
DmiId_t            compId;         /* component to access */
```

The *result* parameter is a pointer to a `DmiAddGroupOUT` structure containing the following members:

```
DmiErrorStatus_t   error_status;
DmiId_t            groupId;        /* SP-allocated group ID */
DmiStringList_t    *errors;        /* installation error messages */
```

The `DmiDeleteComponent()` function removes an existing component from the database. The *argin* parameter is an instance of a `DmiDeleteComponentIN` structure containing the following members:

```
DmiHandle_t        handle;         /* an open session handle */
DmiId_t            compId;         /* component to delete */
```

The *result* parameter is a pointer to a `DmiDeleteComponentOUT` structure containing the following members:

```
DmiErrorStatus_t   error_status;
```

The `DmiDeleteLanguage()` function removes a specific language mapping for a component. You specify the language string and component ID. The *argin* parameter is an instance of a `DmiDeleteLanguageIN` structure containing the following members:

```
DmiHandle_t         handle;        /* an open session handle */
DmiString_t         *language;     /* language to delete */
DmiId_t             compId;        /* component to access */
```

The *result* parameter is a pointer to a `DmiDeleteLanguageOUT` structure containing the following members:

```
DmiErrorStatus_t    error_status;
```

The `DmiDeleteGroup()` function removes a group from a component. The caller specifies the component and group IDs. The *argin* parameter is an instance of a `DmiDeleteGroupIN` structure containing the following members:

```
DmiHandle_t         handle;        /* an open session handle */
DmiId_t             compId;        /* component containing group */
DmiId_t             groupId;       /* group to delete */
```

The *result* parameter is a pointer to a `DmiDeleteGroupOUT` structure containing the following members:

```
DmiErrorStatus_t    error_status;
```

Return Values The `DmiAddComponent()` function returns the following possible values:

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_RPC_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_ILLEGAL_PARAMETER
DMIERR_SP_INACTIVE
DMIERR_FILE_ERROR
DMIERR_BAD_SCHEMA_DESCRIPTION_FILE
```

The `DmiAddGroup()` function returns the following possible values:

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_RPC_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_ILLEGAL_PARAMETER
DMIERR_SP_INACTIVE
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_COMPONENT_NOT_FOUND
DMIERR_FILE_ERROR
DMIERR_BAD_SCHEMA_DESCRIPTION_FILE
```

The `DmiAddLanguage()` function returns the following possible values:

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_RPC_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_ILLEGAL_PARAMETER
DMIERR_SP_INACTIVE
DMIERR_COMPONENT_NOT_FOUND
DMIERR_FILE_ERROR
DMIERR_BAD_SCHEMA_DESCRIPTION_FILE
```

The `DmiDeleteComponent()` function returns the following possible values:

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_RPC_HANDLE
DMIERR_OUT_OF_MEMORY
```

```
DMIERR_ILLEGAL_PARAMETER
DMIERR_SP_INACTIVE
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_COMPONENT_NOT_FOUND
DMIERR_FILE_ERROR
```

THe `DmiDeleteGroup()` function returns the following possible values:

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_RPC_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_ILLEGAL_PARAMETER
DMIERR_SP_INACTIVE
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_COMPONENT_NOT_FOUND
DMIERR_FILE_ERROR
```

The `DmiDeleteLanguage()` function returns the following possible values:

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_RPC_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_ILLEGAL_PARAMETER
DMIERR_SP_INACTIVE
DMIERR_COMPONENT_NOT_FOUND
DMIERR_FILE_ERROR
```

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWsasdk
MT-level	Unsafe

See Also `attributes(5)`

Name DmiAddRow, DmiDeleteRow, DmiGetAttribute, DmiGetMultiple, DmiSetAttribute, DmiSetMultiple – Management Interface operation functions

Synopsis

```
cc [ flag ... ] file ... -ldmimi  -ldmi   -lnsl   -lrwtool  [ library ... ]
#include <server.h>
#include <miapi.h>

bool_t DmiAddRow(DmiAddRowIN argin, DmiAddRowOUT *result, DmiRpcHandle *dmi_rpc_handle);

bool_t DmiDeleteRow(DmiDeleteRowIN argin, DmiDeleteRowOUT *result,
     DmiRpcHandle *dmi_rpc_handle);

bool_t DmiGetAttribute(DmiGetAttributeIN argin, DmiGetAttributeOUT *result,
     DmiRpcHandle *dmi_rpc_handle);

bool_t DmiGetMultiple(DmiGetMultipleIN argin, DmiGetMultipleOUT *result,
     DmiRpcHandle *dmi_rpc_handle);

bool_t DmiSetAttribute(DmiSetAttributeIN argin, DmiSetAttributeOUT *result,
     DmiRpcHandle *dmi_rpc_handle);

bool_t DmiSetMultiple(DmiSetMultipleIN argin, DmiSetMultipleOUT *result,
     DmiRpcHandle *dmi_rpc_handle);
```

Description The operation functions provide a method for retrieving a single value from the Service Provider and for setting a single attribute value. In addition, you may also retrieve attribute values from the Service Provider. You may perform a set operation on an attribute or a list of attributes and add or delete a row from an existing table.

The `DmiAddRow()` function adds a row to an existing table. The `rowData` parameter contains the full data, including key attribute values, for a row. It is an error for the key list to specify an existing table row. The *argin* parameter is an instance of a `DmiAddRowIN` structure containing the following members:

```
DmiHandle_t          handle;       /* An open session handle */
DmiRowData_t         *rowData;     /* Attribute values to set */
```

The *result* parameter is a pointer to a `DmiAddRowOUT` structure containing the following members:

```
DmiErrorStatus_t     error_status;
```

`DmiDeleteRow()` function removes a row from an existing table. The key list must specify valid keys for a table row. The *argin* parameter is an instance of a `DmiDeleteRowIN` structure containing the following members:

```
DmiHandle_t          handle;       /* An open session handle */
DmiRowData_t         *rowData;     /* Row to delete */
```

The *result* parameter is a pointer to a `DmiDeleteRowOUT` structure containing the following members:

```
DmiErrorStatus_t      error_status;
```

The `DmiGetAttribute()` function provides a simple method for retrieving a single attribute value from the Service Provider. The `compId`, `groupId`, `attribId`, and `keyList` identify the desired attribute. The resulting attribute value is returned in a newly allocated `DmiDataUnion` structure. The address of this structure is returned through the `value` parameter. The *argin* parameter is an instance of a `DmiListComponentsIN` structure containing the following members:

```
DmiHandle_t           handle;       /* an open session handle */
DmiId_t               compId;       /* Component to access */
DmiId_t               groupId;      /* Group within component */
DmiId_t               attribId;     /* Attribute within a group */
DmiAttributeValues_t  *keyList;     /* Keylist to specify a table row */
```

The *result* parameter is a pointer to a `DmiGetAttributeOUT` structure containing the following members:

```
DmiErrorStatus_t      error_status;
DmiDataUnion_t        *value;       /* Attribute value returned */
```

The `DmiGetMultiple()` function retrieves attribute values from the Service Provider. This procedure may get the value for an individual attribute, or for multiple attributes across groups, components, or rows of a table.

The `DmiSetAttribute()` function provides a simple method for setting a single attribute value. The `compId`, `groupId`, `attribId`, and `keyList` identify the desired attribute. The `setMode` parameter defines the procedure call as a Set, Reserve, or Release operation. The new attribute value is contained in the `DmiDataUnion` structure whose address is passed in the `value` parameter. The *argin* parameter is an instance of a `DmiSetAttributeIN` structure containing the following members:

```
DmiHandle_t           handle;
DmiId_t               compId;
DmiId_t               groupId;
DmiId_t               attribId;
DmiAttributeValues_t  *keyList;
DmiSetMode_t          setMode;
DmiDataUnion_t        *value;
```

The *result* parameter is a pointer to a `DmiSetAttributeOUT` structure containing the following members:

```
DmiErrorStatus_t      error_status;
```

The `DmiSetMultiple()` function performs a set operation on an attribute or list of attributes. Set operations include actually setting the value, testing and reserving the attribute for future setting, or releasing the set reserve. These variations on the set operation are specified by the parameter `setMode`. The *argin* parameter is an instance of a `DmiSetMultipleIN` structure containing the following members:

```
DmiHandle_t           handle;       /* An open session handle */
DmiSetMode_t          setMode;      /* set, reserve, or release */
DmiMultiRowData_t     *rowData;     /* Attribute values to set */
```

The *result* parameter is a pointer to a `DmiSetMultipleOUT` structure containing the following members:

```
DmiErrorStatus_t      error_status;
```

The `rowData` array describes the attributes to set, and contains the new attribute values. Each element of rowData specifies a component, group, key list (for table accesses), and attribute list to set. No data is returned from this function.

Return Values The `DmiAddRow()` function returns the following possible values:

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_RPC_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_ILLEGAL_PARAMETER
DMIERR_SP_INACTIVE
DMIERR_VALUE_UNKNOWN
DMIERR_COMPONENT_NOT_FOUND
DMIERR_GROUP_NOT_FOUND
DMIERR_ILLEGAL_KEYS
DMIERR_DIRECT_INTERFACE_NOT_REGISTERED
DMIERR_UNKNOWN_CI_REGISTRY
DMIERR_VALUE_UNKNOWN
DMIERR_UNABLE_TO_ADD_ROW
```

The `DmiDeleteRow()` function returns the following possible values:

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_RPC_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_ILLEGAL_PARAMETER
DMIERR_SP_INACTIVE
DMIERR_ATTRIBUTE_NOT_FOUND
DMIERR_COMPONENT_NOT_FOUND
DMIERR_GROUP_NOT_FOUND
DMIERR_ILLEGAL_KEYS
DMIERR_ILLEGAL_TO_GET
DMIERR_DIRECT_INTERFACE_NOT_REGISTERED
DMIERR_ROW_NOT_FOUND
DMIERR_UNKNOWN_CI_REGISTRY
DMIERR_VALUE_UNKNOWN
DMIERR_UNABLE_TO_DELETE_ROW
```

The `DmiGetAttribute()` function returns the following possible values:

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_RPC_HANDLE
```

```
DMIERR_OUT_OF_MEMORY
DMIERR_ILLEGAL_PARAMETER
DMIERR_SP_INACTIVE
DMIERR_ATTRIBUTE_NOT_FOUND
DMIERR_COMPONENT_NOT_FOUND
DMIERR_GROUP_NOT_FOUND
DMIERR_ILLEGAL_KEYS
DMIERR_ILLEGAL_TO_GET
DMIERR_DIRECT_INTERFACE_NOT_REGISTERED
DMIERR_ROW_NOT_FOUND
DMIERR_UNKNOWN_CI_REGISTRY
DMIERR_FILE_ERROR
DMIERR_VALUE_UNKNOWN
```

The `DmiGetMultiple()` function returns the following possible values:

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_RPC_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_ILLEGAL_RPC_PARAMETER
DMIERR_SP_INACTIVE
DMIERR_ATTRIBUTE_NOT_FOUND
DMIERR_COMPONENT_NOT_FOUND
DMIERR_GROUP_NOT_FOUND
DMIERR_ILLEGAL_KEYS
DMIERR_ILLEGAL_TO_GET
DMIERR_DIRECT_INTERFACE_NOT_REGISTERED
DMIERR_ROW_NOT_FOUND
DMIERR_UNKNOWN_CI_REGISTRY
DMIERR_FILE_ERROR
DMIERR_VALUE_UNKNOWN
```

The `DmiSetAttribute()` function returns the following possible values:

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_RPC_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_ILLEGAL_PARAMETER
DMIERR_SP_INACTIVE
DMIERR_ATTRIBUTE_NOT_FOUND
DMIERR_COMPONENT_NOT_FOUND
DMIERR_GROUP_NOT_FOUND
DMIERR_ILLEGAL_KEYS
DMIERR_ILLEGAL_TO_GET
DMIERR_DIRECT_INTERFACE_NOT_REGISTERED
DMIERR_ROW_NOT_FOUND
DMIERR_UNKNOWN_CI_REGISTRY
DMIERR_FILE_ERROR
DMIERR_VALUE_UNKNOWN
```

The `DmiSetMultiple()` function returns the following possible values:

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_RPC_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_ILLEGAL_PARAMETER
DMIERR_SP_INACTIVE
DMIERR_ATTRIBUTE_NOT_FOUND
DMIERR_COMPONENT_NOT_FOUND
DMIERR_GROUP_NOT_FOUND
DMIERR_ILLEGAL_KEYS
DMIERR_ILLEGAL_TO_SET
DMIERR_DIRECT_INTERFACE_NOT_REGISTERED
DMIERR_ROW_NOT_FOUND
DMIERR_UNKNOWN_CI_REGISTRY
DMIERR_FILE_ERROR
DMIERR_VALUE_UNKNOWN
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-level	Unsafe

See Also

`attributes(5)`

Name dmi_error – print error in string form

Synopsis

```
cc [ flag ... ] file ... -ldmi -lnsl -lrwtool [ library ... ]
#include <dmi/dmi_error.hh>

void dmi_error(DmiErrorStatus_t error_status);
```

Description For the given *error_status*, the `dmi_error()` function prints the corresponding error in string form. The function prints "unknown dmi errors" if *error_status* is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-level	MT-Safe

See Also `libdmi(3LIB)`, `attributes(5)`

Name DmiGetConfig, DmiGetVersion, DmiRegister, DmiSetConfig, DmiUnregister – Management Interface initialization functions

Synopsis

```
cc [ flag ... ] file ... -ldmimi  -ldmi   -lnsl   -lrwtool  [ library ... ]
#include <server.h>
#include <miapi.h>

bool_t DmiGetConfig(DmiGetConfigIN argin, DmiGetConfigOUT *result,
     DmiRpcHandle *dmi_rpc_handle);

bool_t DmiGetVersion(DmiGetVersionIN argin, DmiGetVersionOUT *result,
     DmiRpcHandle *dmi_rpc_handle);

bool_t DmiRegister(DmiRegisterIN argin, DmiRegisterOUT *result, DmiRpcHandle *dmi_rpc_handle);

bool_t DmiSetConfig(DmiSetConfigIN argin, DmiSetConfigOUT *result,
     DmiRpcHandle *dmi_rpc_handle);

bool_t DmiUnregister(DmiUnregisterIN argin, DmiUnregisterOUT *result,
     DmiRpcHandle  *dmi_rpc_handle);
```

Description The Management Interface initialization functions enable you to register management applications to the Service Provider. You may also retrieve information about the Service Provider, get and set session configuration information for your session.

The `DmiGetConfig()` function retrieves the per-session configuration information. The configuration information consists of a string describing the current language being used for the session. The *argin* parameter is an instance of a `DmiGetConfigIN` structure containing the following member:

```
DmiHandle_t        handle;          /* an open session handle */
```

The *result* parameter is a pointer to a `DmiGetConfigOUT` structure containing the following members:

```
DmiErrorStatus_t   error_status;
DmiString_t        *language;       /* current session language */
```

The `DmiGetVersion()` function retrieves information about the Service Provider. The management application uses the `DmiGetVersion()` procedure to determine the DMI specification level supported by the Service Provider. This procedure also returns the service provided description string, and may contain version information about the Service Provider implementation. The *argin* parameter is an instance of a `DmiGetVersionIN` structure containing the following member:

```
DmiHandle_t        handle;          /* an open session handle */
```

The *result* parameter is a pointer to a `DmiGetVersionOUT` structure containing the following members:

```
DmiErrorStatus_t   error_status;
DmiString_t        *dmiSpecLevel;   /* DMI specification version */
```

```
DmiString_t        *description;    /* OS specific DMI SP version */
DmiFileTypeList_t  *fileTypes;      /* file types for MIF installation */
```

The `DmiRegister()` function provides the management application with a unique per-session handle. The Service Provider uses this procedure to initialize to an internal state for subsequent procedure calls made by the application. This procedure must be the first command executed by the management application. *argin* is an instance of a `DmiRegisterIN` structure containing the following member:

```
DmiHandle_t        handle;          /* an open session handle */
```

The *result* parameter is a pointer to a `DmiRegisterOUT` structure containing the following members:

```
DmiErrorStatus_t   error_status;
DmiHandle_t        *handle;         /* an open session handle */
```

The `DmiSetConfig()` function sets the per-session configuration information. The configuration information consists of a string describing the language required by the management application. The *argin* parameter is an instance of a `DmiSetConfigIN` structure containing the following member:

```
DmiHandle_t        handle;          /* an open session handle */
DmiString_t        *language;       /* current language required */
```

The *result* parameter is a pointer to a `DmiSetConfigOUT` structure containing the following member:

```
DmiErrorStatus_t   error_status;
```

The `DmiUnregister()` function is used by the Service Provider to perform end-of-session cleanup actions. On return from this function, the session handle is no longer valid. This function must be the last DMI command executed by the management application. The *argin* parameter is an instance of a `DmiUnregisterIN` structure containing the following member:

```
DmiHandle_t        handle;          /* an open session handle */
```

The *result* parameter is a pointer to a `DmiUnregisterOUT` structure containing the following members:

```
DmiErrorStatus_t   error_status;
```

Return Values The `DmiGetConfig()` function returns the following possible values:

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_RPC_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_ILLEGAL_PARAMETER
DMIERR_SP_INACTIVE
```

The `DmiGetVersion()` function returns the following possible values:

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_RPC_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_SP_INACTIVE
```

The `DmiRegister()` function returns the following possible values:

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_RPC_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_SP_INACTIVE
```

The `DmiSetConfig()` function returns the following possible values:

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_RPC_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_ILLEGAL_PARAMETER
DMIERR_SP_INACTIVE
DMIERR_ILLEGAL_TO_SET
```

The `DmiUnRegister()` function returns the following possible values:

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_RPC_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_ILLEGAL_PARAMETER
DMIERR_SP_INACTIVE
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-level	Unsafe

See Also

`attributes(5)`

Name DmiListAttributes, DmiListClassNames, DmiListComponents, DmiListComponentsByClass, DmiListGroups, DmiListLanguages – Management Interface listing functions

Synopsis

```
cc [ flag ... ] file ... -ldmimi  -ldmi   -lnsl   -lrwtool  [ library ... ]
#include <server.h>
#include <miapi.h>

bool_t DmiListAttributes(DmiListAttributesIN argin, DmiListAttributesOUT *result,
     DmiRpcHandle *dmi_rpc_handle);

bool_t DmiListClassNames(DmiListClassNamesIN argin, DmiListClassNamesOUT *result,
     DmiRpcHandle *dmi_rpc_handle);

bool_t DmiListComponents(DmiListComponentsIN argin, DmiListComponentsOUT *result,
     DmiRpcHandle *dmi_rpc_handle);

bool_t DmiListComponentsByClass(DmiListComponentsByClassIN argin,
     DmiListComponentsByClassOUT *result, DmiRpcHandle *dmi_rpc_handle);

bool_t DmiListGroups(DmiListGroupsIN argin, DmiListGroupsOUT *result,
     DmiRpcHandle *dmi_rpc_handle);

bool_t DmiListLanguages(DmiListLanguagesIN argin, DmiListLanguagesOUT *result,
     DmiRpcHandle *dmi_rpc_handle);
```

Description The listing functions enables you to retrieve the names and the description of components in a system. You may also list components by class that match a specified criteria. The listing functions retrieve the set of language mappings installed for a specified component, retrieve class name strings for all groups in a component, retrieve a list of groups within a component, and retrieve the properties for one or more attributes in a group.

The `DmiListComponents()` function retrieves the name and (optionally) the description of components in a system. Use this to interrogate a system to determine what components are installed. The *argin* parameter is an instance of a `DmiListComponentsIN` structure containing the following members:

```
DmiHandle_t          handle;          /* an open session handle */
DmiRequestMode_t     requestMode;     /* Unique, first, or next */
DmiUnsigned_t        maxCount;        /* maximum number to return,
                                         0 for all */
DmiBoolean_t         getPragma;       /* get optional pragma string */
DmiBoolean_t         getDescription;  /* get optional component
                                         description */
DmiId_t              compId;          /* component ID to start with */
```

The *result* parameter is a pointer to a `DmiListComponentsOUT` structure containing the following members:

```
DmiErrorStatus_t     error_status;
DmiComponentList_t   *reply;          /* list of components */
```

An enumeration accesses a specific component or may be used to sequentially access all components in a system. The caller may choose not to retrieve the component description by setting the value `getDescription` to false. The caller may choose not to retrieve the pragma string by setting the value of gutta-percha to false. The `maxCount`, `requestMode`, and `compId` parameters allow the caller to control the information returned by the Service Provider. When the `requestMode` is `DMI_UNIQUE`, `compId` specifies the first component requested (or only component if `maxCount` is one). When the `requestMode` is `DMI_NEXT`, `compId` specifies the component just before the one requested. When `requestMode` is `DMI_FIRST`, `compId` is unused.

To control the amount of information returned, the caller sets `maxCount` to something other than zero. The service provider must honor this limit on the amount of information returned. When `maxCount` is `0` the service provider returns information for all components, subject to the constraints imposed by `requestMode` and `compId`.

The `DmiListComponentsByClass()` function lists components that match specified criteria. Use this function to determine if a component contains a certain group or a certain row in a table. A filter condition may be that a component contains a specified group class name or that it contains a specific row in a specific group. As with `DmiListComponents()`, the description and pragma strings are optional return values. *argin* is an instance of a `DmiListComponentsByClassIN` structure containing the following members:

```
DmiHandle_t          handle;          /* an open session handle */
DmiRequestMode_t     requestMode;     /* Unique, first or next */
DmiUnsigned_t        maxCount;        /* maximum number to return,
                                         or 0 for all */
DmiBoolean_t         getPragma;       /* get the optional pragma
                                         string */
DmiBoolean_t         getDescription;  /* get optional component
                                         description */
DmiId_t              compId;          /* component ID to start with */
DmiString_t          *className;      /* group class name string
                                         to match*/
DmiAttributeValues_t *keyList;        /* group row keys to match */
```

The *result* parameter is a pointer to a `DmiListComponentsbyClassOUT` structure containing the following members:

```
DmiErrorStatus_t     error_status;
DmiComponentList_t   *reply;          /* list of components */
```

The `DmiListLanguages()` function retrieves the set of language mappings installed for the specified component. The *argin* parameter is an instance of a `DmiListLanguagesIN` structure containing the following members:

```
DmiHandle_t          handle;          /* An open session handle */
DmiUnsigned_t        maxCount;        /* maximum number to return,
                                         or 0 for all */
DmiId_t              compId;          /* Component to access */
```

The *result* parameter is a pointer to a `DmiListLanguagesOUT` structure containing the following members:

```
DmiErrorStatus_t     error_status;
DmiStringList_t      *reply;          /* List of language strings */
```

The `DmiListClassNames()` function retrieves the class name strings for all groups in a component. This enables the management application to easily determine if a component contains a specific group, or groups. The *argin* parameter is an instance of a `DmiListClassNamesIN` structure containing the following members:

```
DmiHandle_t          handle;          /* An open session handle */
DmiUnsigned_t        maxCount;        /* maximum number to return,
                                         or 0 for all */
DmiId_t              compId;          /* Component to access */
```

The *result* parameter is a pointer to a `DmiListClassNamesOUT` structure containing the following members:

```
DmiErrorStatus_t     error_status;
DmiClassNameList_t   *reply;          /* List of class names and
                                         group IDs */
```

The `DmiListGroups()` function retrieves a list of groups within a component. With this function you can access a specific group or sequentially access all groups in a component. All enumerations of groups occur within the specified component and do not span components. The *argin* parameter is an instance of a `DmiListGroupsIN` structure containing the following members:

```
DmiHandle_t          handle;          /* An open session handle */
DmiRequestMode_t     requestMode;     /* Unique, first or next group */
DmiUnsigned_t        maxCount;        /* Maximum number to return,
                                         or 0 for all */
DmiBoolean_t         getPragma;       /* Get the optional pragma string */
DmiBoolean_t         getDescription;  /* Get optional group description */
DmiId_t              compId;          /* Component to access */
DmiId_t              groupId;         /* Group to start with, refer to
                                         requestMode */
```

The *result* parameter is a pointer to a `DmiListGroupsOUT` structure containing the following members:

```
DmiErrorStatus_t     error_status;
DmiGroupList_t       *reply;
```

The caller may choose not to retrieve the group description by setting the value `getDescription` to false. The caller may choose not to retrieve the pragma string by setting the value of `getPragma` to false. The `maxCount`, `requestMode`, and `groupId` parameters allow the caller to control the information returned by the Service Provider. When the `requestMode` is `DMI_UNIQUE`, `groupId` specifies the first group requested (or only group if `maxCount` is one).

When the `requestMode` is `DMI_NEXT`, `groupId` specifies the group just before the one requested. When `requestMode` is `DMI_FIRST`, `groupId` is unused. To control the amount of information returned, the caller sets `maxCount` to something other than zero. The service provider must honor this limit on the amount of information returned. When `maxCount` is zero the service provider returns information for all groups, subject to the constraints imposed by `requestMode` and `groupId`.

The `DmiListAttributes()` function retrieves the properties for one or more attributes in a group. All enumerations of attributes occur within the specified group, and do not span groups. The *argin* parameter is an instance of a `DmiListAttributesIN` structure containing the following members:

```
DmiHandle_t          handle;          /* An open session handle */
DmiRequestMode_t     requestMode;     /* Unique, first or next group */
DmiUnsigned_t        maxCount;        /* Maximum number to return,
                                         or 0 for all */
DmiBoolean_t         getPragma;       /* Get the optional pragma string */
DmiBoolean_t         getDescription;  /* Get optional group description */
DmiId_t              compId;          /* Component to access */
DmiId_t              groupId;         /* Group to access */
DmiId_t              attribId;        /* Attribute to start with, refer
                                         to requestMode */
```

The *result* parameter is a pointer to a `DmiListAttributesOUT` structure containing the following members:

```
DmiErrorStatus_t     error_status;
DmiAttributeList_t   *reply;          /* List of attrbutes */
```

You may choose not to retrieve the description string by setting the value of `getDescription` to false. Likewise, you may choose not to retrieve the pragma string by setting the value of `getPragma` to false. The `maxCount`, `requestMode`, and `attribId` parameters allow you to control the information returned by the Service Provider. When the `requestMode` is `DMI_UNIQUE`, `attribId` specifies the first attribute requested (or only attribute if `maxCount` is one). When the `requestMode` is `DMI_NEXT`, `attribId` specifies the attribute just before the one requested. When `requestMode` is `DMI_FIRST`, `attribId` is unused. To control the amount of information returned, the caller sets `maxCount` to something other than zero. The Service Provider must honor this limit on the amount of information returned. When `maxCount` is zero the service provider returns information for all attributes, subject to the constraints imposed by `requestMode` and `attribId`.

Return Values The `DmiListAttributes()` function returns the following possible values:

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_RPC_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_ILLEGAL_PARAMETER
DMIERR_SP_INACTIVE
```

```
DMIERR_ATTRIBUTE_NOT_FOUND
DMIERR_COMPONENT_NOT_FOUND
DMIERR_GROUP_NOT_FOUND
DMIERR_FILE_ERROR
```

The `DmiListClassNames()` function returns the following possible values:

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_RPC_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_ILLEGAL_PARAMETER
DMIERR_SP_INACTIVE
DMIERR_COMPONENT_NOT_FOUND
DMIERR_FILE_ERROR
```

The `DmiListComponents()` function returns the following possible values:

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_RPC_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_ILLEGAL_PARAMETER
DMIERR_SP_INACTIVE
DMIERR_COMPONENT_NOT_FOUND
DMIERR_FILE_ERROR
```

The `DmiListComponentsByClass()` function returns the following possible values:

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_RPC_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_ILLEGAL_PARAMETER
DMIERR_SP_INACTIVE
DMIERR_COMPONENT_NOT_FOUND
DMIERR_FILE_ERROR
```

The `DmiListGroups()` function returns the following possible values:

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_RPC_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_ILLEGAL_PARAMETER
DMIERR_SP_INACTIVE
DMIERR_COMPONENT_NOT_FOUND
DMIERR_GROUP_NOT_FOUND
DMIERR_FILE_ERROR
```

The `DmiListLanguages()` function returns the following possible values:

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_RPC_HANDLE
DMIERR_OUT_OF_MEMORY
```

```
DMIERR_ILLEGAL_PARAMETER
DMIERR_SP_INACTIVE
DMIERR_COMPONENT_NOT_FOUND
DMIERR_FILE_ERROR
```

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-level	Unsafe

See Also attributes(5)

Name DmiRegisterCi, DmiUnregisterCi, DmiOriginateEvent – Service Provider functions for components

Synopsis

```
cc [ flag... ] file... -lci  -ldmi   -lnsl   -lrwtool  [ library... ]
#include <server.h>
#include <ciapi.h>

extern bool_t DmiRegisterCi(DmiRegisterCiIN argin, DmiRegisterCiOUT *result,
     DmiRpcHandle *dmi_rpc_handle);

bool_t DmiUnregisterCi(DmiUnregisterCiIN argin, DmiUnregisterCiOUT *result,
     DmiRpcHandle *dmi_rpc_handle);

bool_t DmiOriginateEvent(DmiOriginateEventIN argin, DmiOriginateEventOUT *result,
     DmiRpcHandle *dmi_rpc_handle);
```

Description These functions provide component communication with the DMI through the Component Interface (CI).

Component instrumentation code may register with the Service Provider to override its current mechanism for the registered attributes. Instead of manipulating the data in the MIF database or invoking programs, the Service Provider calls the entry points provided in the registration call. Once the component unregisters, the Service Provider returns to a normal method of processing requests for the data as defined in the MIF. Component instrumentation can temporarily interrupt normal processing to perform special functions.

Registering attributes through the direct interface overrides atttributes that are already being served through the direct interface. RPC is used for communication from the Service Provider to the component instrumentation.

For all three functions, *argin* is the parameter passed to initiate an RPC call, *result* is the result of the RPC call, and *dmi_rpc_handle* is an open session RPC handle.

The `DmiRegisterCi()` function registers a callable interface for components that have resident instrumentation code and/or to get the version of the Service Provider.

The `DmiUnregisterCi()` function communicates to the Service Provider to remove a direct component instrumentation interface from the Service Provider table of registered interfaces.

The `DmiOriginateEvent()` function originates an event for filtering and delivery. Any necessary indication filtering is performed by this function (or by subsequent processing) before the event is forwarded to the management applications.

A component ID value of zero (0) specifies the event was generated by something that has not been installed as a component, and has no component ID.

Return Values The `DmiRegisterCi()` function returns the following possible values:

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
DMIERR_ATTRIBUTE_NOT_FOUND
DMIERR_COMPONENT_NOT_FOUND
DMIERR_GROUP_NOT_FOUND
DMIERR_DATABASE_CORRUPT
DMIERR_OUT_OF_MEMORY
DMIERR_ILLEGAL_DMI_LEVEL
```

The `DmiUnregisterCi()` function returns the following possible values:

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
DMIERR_UNKNOWN_CI_REGISTRY
```

The `DmiOriginateEvent()` function returns the following possible values:

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
DMIERR_UNKNOWN_CI_REGISTRY
```

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-level	Unsafe

See Also `attributes(5)`

Name ea_error – error interface to extended accounting library

Synopsis

```
cc [ flag... ] file... -lexacct [ library ... ]
#include <exacct.h>

int ea_error(void);
```

Description The `ea_error()` function returns the error value of the last failure recorded by the invocation of one of the functions of the extended accounting library, `libexacct`.

Return Values

`EXR_CORRUPT_FILE`	A function failed because the file was not a valid exacct file.
`EXR_EOF`	A function detected the end of the file, either when reading forwards or backwards through the file.
`EXR_INVALID_BUF`	When unpacking an object, an invalid unpack buffer was specified.
`EXR_INVALID_OBJ`	The object type passed to the function is not valid for the requested operation, for example passing a group object to `ea_set_item(3EXACCT)`.
`EXR_NO_CREATOR`	When creating a new file no creator was specified, or when opening a file for reading the creator value did not match the value in the file.
`EXR_NOTSUPP`	An unsupported type of access was attempted, for example attempting to write to a file that was opened read-only.
`EXR_OK`	The function completed successfully.
`EXR_SYSCALL_FAIL`	A system call invoked by the function failed. The `errno` variable contains the error value set by the underlying call.
`EXR_UNKN_VERSION`	The file referred to by name uses an `exacct` file version that cannot be processed by this library.

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

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

See Also `read(2)`, `libexacct(3LIB)`, `attributes(5)`

Name ea_open, ea_close – open or close exacct files

Synopsis

```
cc [ flag... ] file... -lexacct [ library... ]
#include <exacct.h>

int ea_open(ea_file_t *ef, char *name, char *creator, int aflags,
     int oflags, mode_t mode);

int ea_close(ea_file_t *ef);
```

Description The `ea_open()` function provides structured access to `exacct` files. The *aflags* argument contains the appropriate `exacct` flags necessary to describe the file. The *oflags* and *mode* arguments contain the appropriate flags and mode to open the file; see `<fcntl.h>`. If `ea_open()` is invoked with `EO_HEAD` specified in *aflags*, the resulting file is opened with the object cursor located at the first object of the file. If `ea_open()` is invoked with `EO_TAIL` specified in *aflags*, the resulting file is opened with the object cursor positioned beyond the last object in the file. If `EO_NO_VALID_HDR` is set in *aflags* along with `EO_HEAD`, the initial header record will be returned as the first item read from the file. When creating a file, the *creator* argument should be set (system generated files use the value "SunOS"); when reading a file, this argument should be set to `NULL` if no validation is required; otherwise it should be set to the expected value in the file.

The `ea_close()` function closes an open `exacct` file.

Return Values Upon successful completion, `ea_open()` and `ea_close()` return `0`. Otherwise they return `−1` and call `ea_error(3EXACCT)` to return the extended accounting error value describing the error.

Errors The `ea_open()` and `ea_close()` functions may fail if:

`EXR_SYSCALL_FAIL` A system call invoked by the function failed. The `errno` variable contains the error value set by the underlying call.

The `ea_open()` function may fail if:

`EXR_CORRUPT_FILE` The file referred to by *name* is not a valid `exacct` file.

`EXR_NO_CREATOR` In the case of file creation, the *creator* argument was `NULL`. In the case of opening an existing file, a *creator* argument was not NULL and does not match the *creator* item of the `exacct` file.

`EXR_UNKN_VERSION` The file referred to by *name* uses an `exacct` file version that cannot be processed by this library.

Usage The `exacct` file format can be used to represent data other than that in the extended accounting format. By using a unique creator type in the file header, application writers can develop their own format suited to the needs of their application.

Examples

EXAMPLE 1 Open and close `exacct` file.

The following example opens the extended accounting data file for processes. The `exacct` file is then closed.

```
#include <exacct.h>

ea_file_t ef;
if (ea_open(&ef, "/var/adm/exacct/proc", NULL, EO_HEAD,
    O_RDONLY, 0) == -1)
        exit(1);
(void) ea_close(&ef);
```

Attributes

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

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

See Also

`ea_error(3EXACCT)`, `ea_pack_object(3EXACCT)`, `ea_set_item(3EXACCT)`, `libexacct(3LIB)`, `attributes(5)`

Name ea_pack_object, ea_unpack_object, ea_get_creator, ea_get_hostname, ea_next_object, ea_previous_object, ea_get_object, ea_write_object, ea_copy_object, ea_copy_object_tree, ea_get_object_tree – construct, read, and write extended accounting records

Synopsis

```
cc [ flag... ] file... -lexacct [ library... ]
#include <exacct.h>

size_t ea_pack_object(ea_object_t *obj, void *buf,
     size_t bufsize);

ea_object_type_t ea_unpack_object(ea_object_t **objp, int flag,
     void *buf, size_t bufsize);

const char *ea_get_creator(ea_file_t *ef);

const char *ea_get_hostname(ea_file_t *ef);

ea_object_type_t ea_next_object(ea_file_t *ef, ea_object_t *obj);

ea_object_type_t ea_previous_object(ea_file_t *ef,
     ea_object_t *obj);

ea_object_type_t ea_get_object(ea_file_t *ef, ea_object_t *obj);

int ea_write_object(ea_file_t *ef, ea_object_t *obj);

ea_object_type_t *ea_copy_object(const ea_object_t *src);

ea_object_type_t *ea_copy_object_tree(const ea_object_t *src);

ea_object_type_t *ea_get_object_tree(ea_file_t *ef,
     uint32_tnobj);
```

Description The `ea_pack_object()` function converts `exacct` objects from their in-memory representation to their file representation. It is passed an object pointer that points to the top of an `exacct` object hierarchy representing one or more `exacct` records. It returns the size of the buffer required to contain the packed buffer representing the object hierarchy. To obtain the correct size of the required buffer, the *buf* and *bufsize* parameters can be set to `NULL` and 0 respectively, and the required buffer size will be returned. The resulting packed record can be passed to `putacct(2)` or to `ea_set_item(3EXACCT)` when constructing an object of type `EXT_EXACCT_OBJECT`.

The `ea_unpack_object()` function reverses the packing process performed by `ea_pack_object()`. A packed buffer passed to `ea_unpack_object()` is unpacked into the original hierarchy of objects. If the unpack operation fails (for example, due to a corrupted or incomplete buffer), it returns `EO_ERROR`; otherwise, the object type of the first object in the hierarchy is returned. If `ea_unpack_object()` is invoked with *flag* equal to `EUP_ALLOC`, it allocates memory for the variable-length data in the included objects. Otherwise, with *flag* equal to `EUP_NOALLOC`, it sets the variable length data pointers within the unpacked object structures to point within the buffer indicated by *buf*. In both cases, `ea_unpack_object()` allocates all the necessary `exacct` objects to represent the unpacked record. The resulting object hierarchy can be freed using `ea_free_object(3EXACCT)` with the same *flag* value.

The ea_get_creator() function returns a pointer to a string representing the recorded creator of the exacct file. The ea_get_hostname() function returns a pointer to a string representing the recorded hostname on which the exacct file was created. These functions will return NULL if their respective field was not recorded in the exacct file header.

The ea_next_object() function reads the basic fields (eo_catalog and eo_type) into the ea_object_t indicated by *obj* from the exacct file referred to by *ef* and rewinds to the head of the record. If the read object is corrupted, ea_next_object() returns EO_ERROR and records the extended accounting error code, accessible with ea_error(3EXACCT). If end-of-file is reached, EO_ERROR is returned and the extended accounting error code is set to EXR_EOF.

The ea_previous_object() function skips back one object in the file and reads its basic fields (eo_catalog and eo_type) into the indicated ea_object_t. If the read object is corrupted, ea_previous_object() returns EO_ERROR and records the extended accounting error code, accessible with ea_error(3EXACCT). If end-of-file is reached, EO_ERROR is returned and the extended accounting error code is set to EXR_EOF.

The ea_get_object() function reads the value fields into the ea_object_t indicated by *obj*, allocating memory as necessary, and advances to the head of the next record. Once a record group object is retrieved using ea_get_object(), subsequent calls to ea_get_object() and ea_next_object() will track through the objects within the record group, and on reaching the end of the group, will return the next object at the same level as the group from the file. If the read object is corrupted, ea_get_object() returns EO_ERROR and records the extended accounting error code, accessible with ea_error(3EXACCT). If end-of-file is reached, EO_ERROR is returned and the extended accounting error code is set to EXR_EOF.

The ea_write_object() function appends the given object to the open exacct file indicated by *ef* and returns 0. If the write fails, ea_write_object() returns −1 and sets the extended accounting error code to indicate the error, accessible with ea_error(3EXACCT).

The ea_copy_object() function copies an ea_object_t. If the source object is part of a chain, only the current object is copied. If the source object is a group, only the group object is copied without its list of members and the eg_nobjs and eg_objs fields are set to 0 and NULL, respectively. Use ea_copy_tree() to copy recursively a group or a list of items.

The ea_copy_object_tree() function recursively copies an ea_object_t. All elements in the eo_next list are copied, and any group objects are recursively copied. The returned object can be completely freed with ea_free_object(3EXACCT) by specifying the EUP_ALLOC flag.

The ea_get_object_tree() function reads in *nobj* top-level objects from the file, returning the same data structure that would have originally been passed to ea_write_object(). On encountering a group object, the ea_get_object() function reads only the group header part of the group, whereas ea_get_object_tree() reads the group and all its member items, recursing into sub-records if necessary. The returned object data structure can be completely freed with ea_free_object() by specifying the EUP_ALLOC flag.

Return Values The `ea_pack_object()` function returns the number of bytes required to hold the `exacct` object being operated upon. If the returned size exceeds *bufsize*, the pack operation does not complete and the function returns (`size_t`) –1 and sets the extended accounting error code to indicate the error.

The `ea_get_object()` function returns the `ea_object_type` of the object if the object was retrieved successfully. Otherwise, it returns `EO_ERROR` and sets the extended accounting error code to indicate the error.

The `ea_next_object()` function returns the `ea_object_type` of the next `exacct` object in the file. It returns `EO_ERROR` if the `exacct` file is corrupted sets the extended accounting error code to indicate the error.

The `ea_unpack_object()` function returns the `ea_object_type` of the first `exacct` object unpacked from the buffer. It returns `EO_ERROR` if the exacct file is corrupted, and sets the extended accounting error code to indicate the error.

The `ea_write_object()` function returns 0 on success. Otherwise it returns –1 and sets the extended accounting error code to indicate the error.

The `ea_copy_object()` and `ea_copy_object_tree()` functions return the copied object on success. Otherwise they return `NULL` and set the extended accounting error code to indicate the error.

The `ea_get_object_tree()` function returns the list of objects read from the file on success. Otherwise it returns `NULL` and sets the extended accounting error code to indicate the error.

The extended account error code can be retrieved using `ea_error`(3EXACCT).

Errors These functions may fail if:

`EXR_SYSCALL_FAIL`
: A system call invoked by the function failed. The `errno` variable contains the error value set by the underlying call. On memory allocation failure, `errno` will be set to `ENOMEM`.

`EXR_CORRUPT_FILE`
: The file referred to by *name* is not a valid `exacct` file, or is unparsable, and therefore appears corrupted. This error is also used by `ea_unpack_buffer()` to indicate a corrupted buffer.

`EXR_EOF`
: The end of the file has been reached. In the case of `ea_previous_record()`, the previous record could not be reached, either because the head of the file was encountered or because the previous record could not be skipped over.

Usage The `exacct` file format can be used to represent data other than that in the extended accounting format. By using a unique creator type in the file header, application writers can develop their own format suited to the needs of their application.

Examples

EXAMPLE 1 Open and close exacct file.

The following example opens the extended accounting data file for processes. The `exacct` file is then closed.

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

ea_file_t ef;
ea_object_t *obj;

...

ea_open(&ef, "foo", O_RDONLY, ...);

while ((obj = ea_get_object_tree(&ef, 1)) != NULL) {
    if (obj->eo_type == EO_ITEM) {
        /* handle item */
    } else {
        /* handle group */
    }
    ea_free_object(obj, EUP_ALLOC);
}

if (ea_error() != EXR_EOF) {
    /* handle error */
}

ea_close(&ef);
```

EXAMPLE 2 Construct an exacct file consisting of a single object containing the current process ID.

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

...

ea_file_t ef;
ea_object_t obj;
pid_t my_pid;

ea_open(&ef, "foo", O_CREAT | O_WRONLY, ...);

my_pid = getpid();
ea_set_item(&obj, EXT_UINT32 | EXC_DEFAULT | EXT_PROC_PID, &my_pid, 0);
(void) ea_write_object(&ef, &obj);

ea_close(&ef);
```

EXAMPLE 2 Construct an `exacct` file consisting of a single object containing the current process ID. *(Continued)*

. . .

Attributes

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

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

See Also

`read(2)`, `ea_error(3EXACCT)`, `ea_open(3EXACCT)`, `ea_set_item(3EXACCT)`, `libexacct(3LIB)`, `attributes(5)`

Name ea_set_item, ea_alloc, ea_strdup, ea_set_group, ea_match_object_catalog, ea_attach_to_object, ea_attach_to_group, ea_free, ea_strfree, ea_free_item, ea_free_object – create, destroy and manipulate exacct objects

Synopsis

```
cc [ flag... ] file... -lexacct [ library... ]
#include <exacct.h>

int ea_set_item(ea_object_t *obj, ea_catalog_t tag, void *value,
     size_t valsize);

void *ea_alloc(size_t size);

char *ea_strdup(char *ptr);

int ea_set_group(ea_object_t *obj, ea_catalog_t tag);

int ea_match_object_catalog(ea_object_t *obj, ea_catalog_t catmask);

void ea_attach_to_object(ea_object_t *head_obj, ea_object_t *obj);

void ea_attach_to_group(ea_object_t *group_obj, ea_object_t *obj);

void ea_free(void *ptr, size_t size);

void ea_strfree(char *ptr);

int ea_free_item(ea_object_t *obj, int flag);

void ea_free_object(ea_object_t *obj, int flag);
```

Description The `ea_alloc()` function allocates a block of memory of the requested size. This block can be safely passed to `libexacct` functions, and can be safely freed by any of the `ea_free()` functions.

The `ea_strdup()` function can be used to duplicate a string that is to be stored inside an `ea_object_t` structure.

The `ea_set_item()` function assigns the given `exacct` object to be a data item with *value* set according to the remaining arguments. For buffer-based data values (`EXT_STRING`, `EXT_EXACCT_OBJECT`, and `EXT_RAW`), a copy of the passed buffer is taken. In the case of `EXT_EXACCT_OBJECT`, the passed buffer should be a packed `exacct` object as returned by `ea_pack_object(3EXACCT)`. Any item assigned with `ea_set_item()` should be freed with `ea_free_item()` specifying a flag value of `EUP_ALLOC` when the item is no longer needed.

The `ea_match_object_catalog()` function returns `TRUE` if the `exacct` object specified by *obj* has a catalog tag that matches the mask specified by *catmask*.

The `ea_attach_to_object()` function attaches an object to the given object. The `ea_attach_to_group()` function attaches a chain of objects as member items of the given group. Objects are inserted at the end of the list of any previously attached objects.

The `ea_free()` function frees a block of memory previously allocated by `ea_alloc()`.

The `ea_strfree()` function frees a string previously copied by `ea_strdup()`.

The `ea_free_item()` function frees the *value* fields in the `ea_object_t` indicated by *obj*, if `EUP_ALLOC` is specified. The object itself is not freed. The `ea_free_object()` function frees the specified object and any attached hierarchy of objects. If the *flag* argument is set to `EUP_ALLOC`, `ea_free_object()` will also free any variable-length data in the object hierarchy; if set to `EUP_NOALLOC`, `ea_free_object()` will not free variable-length data. In particular, these flags should correspond to those specified in calls to `ea_unpack_object(3EXACCT)`.

Return Values The `ea_match_object_catalog()` function returns 0 if the object's catalog tag does not match the given mask, and 1 if there is a match.

Other integer-valued functions return 0 if successful. Otherwise these functions return -1 and set the extended accounting error code appropriately. Pointer-valued functions return a valid pointer if successful and NULL otherwise, setting the extended accounting error code appropriately. The extended accounting error code can be examined with `ea_error(3EXACCT)`.

Errors The `ea_set_item()`, `ea_set_group()`, and `ea_match_object_catalog()` functions may fail if:

`EXR_SYSCALL_FAIL`	A system call invoked by the function failed. The `errno` variable contains the error value set by the underlying call.
`EXR_INVALID_OBJECT`	The passed object is of an incorrect type, for example passing a group object to `ea_set_item()`.

Usage The `exacct` file format can be used to represent data other than that in the extended accounting format. By using a unique creator type in the file header, application writers can develop their own format suited to the needs of their application.

Examples **EXAMPLE 1** Open and close `exacct` file.

Construct an exacct file consisting of a single object containing the current process ID.

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

...

ea_file_t ef;
ea_object_t obj;
pid_t my_pid;

my_pid = getpid();
ea_set_item(&obj, EXT_UINT32 | EXC_DEFAULT | EXT_PROC_PID,
       &my_pid, sizeof(my_pid));

...
```

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

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

See Also read(2), ea_error(3EXACCT), ea_open(3EXACCT), ea_pack_object(3EXACCT), libexacct(3LIB), 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 efi_alloc_and_init, efi_alloc_and_read, efi_free, efi_write – manipulate a disk's EFI Partition Table

Synopsis

```
cc [ flag ... ] file... -lefi [ library ... ]
#include <sys/vtoc.h>
#include <sys/efi_partition.h>

int efi_alloc_and_init(int fd, uint32_t nparts, dk_gpt_t **vtoc);

int efi_alloc_and_read(int fd, dk_gpt_t **vtoc);

void efi_free(dk_gpt_t *vtoc);

int efi_write(int fd, dk_gpt_t *vtoc);
```

Description The `efi_alloc_and_init()` function initializes the `dk_gpt_t` structure specified by *vtoc* in preparation for a call to `efi_write()`. It calculates and initializes the `efi_version`, `efi_lbasize`, `efi_nparts`, `efi_first_u_lba`, `efi_last_lba`, and `efi_last_u_lba` members of this sturcture. The caller can then set the `efi_nparts` member.

The `efi_alloc_and_read()` function allocates memory and returns the partition table.

The `efi_free()` function frees the memory allocated by `efi_alloc_and_init()` and `efi_alloc_and_read()`.

The `efi_write()` function writes the EFI partition table.

The *fd* argument refers to any slice on a raw disk, opened with `O_NDELAY`. See open(2).

The *nparts* argument specifies the number of desired partitions.

The *vtoc* argument is a `dk_gpt_t` structure that describes an EFI partition table and contains at least the following members:

```
uint_t          efi_version;     /* set to EFI_VERSION_CURRENT */
uint_t          efi_nparts;      /* number of partitions in efi_parts */
uint_t          efi_lbasize;     /* size of block in bytes */
diskaddr_t      efi_last_lba;    /* last block on the disk */
diskaddr_t      efi_first_u_lba; /* first block after labels */
diskaddr_t      efi_last_u_lba;  /* last block before backup labels */
struct dk_part  efi_parts[];     /* array of partitions */
```

Return Values Upon successful completion, `efi_alloc_and_init()` returns 0. Otherwise it returns `VT_EIO` if an I/O operation to the disk fails.

Upon successful completion, `efi_alloc_and_read()` returns a positive integer indicating the slice index associated with the open file descriptor. Otherwise, it returns a negative integer to indicate one of the following:

`VT_EIO` An I/O error occurred.

`VT_ERROR` An unknown error occurred.

`VT_EINVAL` An EFI label was not found.

Upon successful completion, `efi_write()` returns 0. Otherwise, it returns a negative integer to indicate one of the following:

`VT_EIO` An I/O error occurred.

`VT_ERROR` An unknown error occurred.

`VT_EINVAL` The label contains incorrect data.

Usage The EFI label is used on disks with more than 1^{32}-1 blocks. For compatibility reasons, the `read_vtoc(3EXT)` and `write_vtoc()` functions should be used on smaller disks. The application should attempt the `read_vtoc()` or `write_vtoc()` call, check for an error of `VT_ENOTSUP`, then call the analogous EFI function.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Unsafe

See Also `fmthard(1M)`, `format(1M)`, `prtvtoc(1M)`, `ioctl(2)`, `open(2)`, `libefi(3LIB)`, `read_vtoc(3EXT)`, `attributes(5)`, `dkio(7I)`

Name elf32_checksum, elf64_checksum – return checksum of elf image

Synopsis

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

long elf32_checksum(Elf *elf);

long elf64_checksum(Elf *elf);
```

Description The `elf32_checksum()` function returns a simple checksum of selected sections of the image identified by *elf*. The value is typically used as the `.dynamic` tag `DT_CHECKSUM`, recorded in dynamic executables and shared objects.

Selected sections of the image are used to calcluate the checksum in order that its value is not affected by utilities such as `strip(1)`.

For the 64–bit class, replace 32 with 64 as appropriate.

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

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

See Also `elf(3ELF)`, `elf_version(3ELF)`, `gelf(3ELF)`, `libelf(3LIB)`, `attributes(5)`

Name elf32_fsize, elf64_fsize – return the size of an object file type

Synopsis

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

size_t elf32_fsize(Elf_Type type, size_t count, unsigned ver);

size_t elf64_fsize(Elf_Type type, size_t count, unsigned ver);
```

Description `elf32_fsize()` gives the size in bytes of the 32-bit file representation of *count* data objects with the given `type`. The library uses version `ver` to calculate the size. See `elf`(3ELF) and `elf_version`(3ELF).

Constant values are available for the sizes of fundamental types:

```
Elf_Type    File Size    Memory Size
ELF_T_ADDR    ELF32_FSZ_ADDR    sizeof(Elf32_Addr)
ELF_T_BYTE    1    sizeof(unsigned char)
ELF_T_HALF    ELF32_FSZ_HALF    sizeof(Elf32_Half)
ELT_T_OFF    ELF32_FSZ_OFF    sizeof(Elf32_Off)
ELF_T_SWORD    ELF32_FSZ_SWORD    sizeof(Elf32_Sword)
ELF_T_WORD    ELF32_FSZ_WORD    sizeof(Elf32_Word)
```

`elf32_fsize()` returns `0` if the value of `type` or `ver` is unknown. See `elf32_xlatetof`(3ELF) for a list of the `type` values.

For the 64–bit class, replace 32 with 64 as appropriate.

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

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

See Also `elf`(3ELF), `elf32_xlatetof`(3ELF), `elf_version`(3ELF), `libelf`(3LIB), `attributes`(5)

Name elf32_getehdr, elf32_newehdr, elf64_getehdr, elf64_newehdr – retrieve class-dependent object file header

Synopsis

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

Elf32_Ehdr *elf32_getehdr(Elf *elf);

Elf32_Ehdr *elf32_newehdr(Elf *elf);

Elf64_Ehdr *elf64_getehdr(Elf *elf);

Elf64_Ehdr *elf64_newehdr(Elf *elf);
```

Description For a 32-bit class file, `elf32_getehdr()` returns a pointer to an ELF header, if one is available for the ELF descriptor *elf*. If no header exists for the descriptor, `elf32_newehdr()` allocates a clean one, but it otherwise behaves the same as `elf32_getehdr()`. It does not allocate a new header if one exists already. If no header exists for `elf32_getehdr()`, one cannot be created for `elf32_newehdr()`, a system error occurs, the file is not a 32-bit class file, or *elf* is `NULL`, both functions return a null pointer.

For the 64–bit class, replace 32 with 64 as appropriate.

The header includes the following members:

```
unsigned char    e_ident[EI_NIDENT];
Elf32_Half    e_type;
Elf32_Half    e_machine;
Elf32_Word    e_version;
Elf32_Addr    e_entry;
Elf32_Off     e_phoff;
Elf32_Off     e_shoff;
Elf32_Word    e_flags;
Elf32_Half    e_ehsize;
Elf32_Half    e_phentsize;
Elf32_Half    e_phnum;
Elf32_Half    e_shentsize;
Elf32_Half    e_shnum;
Elf32_Half    e_shstrndx;
```

The `elf32_newehdr()` function automatically sets the `ELF_F_DIRTY` bit. See `elf_flagdata(3ELF)`.

An application can use `elf_getident()` to inspect the identification bytes from a file.

An application can use `elf_getshnum()` and `elf_getshstrndx()` to obtain section header information. The location of this section header information differs between standard ELF files to those that require Extended Sections.

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

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

See Also elf(3ELF), elf_begin(3ELF), elf_flagdata(3ELF), elf_getident(3ELF), elf_getshnum(3ELF), elf_getshstrndx(3ELF), libelf(3LIB), attributes(5)

Name elf32_getphdr, elf32_newphdr, elf64_getphdr, elf64_newphdr – retrieve class-dependent program header table

Synopsis

```
cc [ flag ... ] file... -lelf [ library ... ]
#include <libelf.h>

Elf32_Phdr *elf32_getphdr(Elf *elf);

Elf32_Phdr *elf32_newphdr(Elf *elf, size_t count);

Elf64_Phdr *elf64_getphdr(Elf *elf);

Elf64_Phdr *elf64_newphdr(Elf *elf, size_t count);
```

Description For a 32-bit class file, `elf32_getphdr()` returns a pointer to the program execution header table, if one is available for the ELF descriptor *elf*.

`elf32_newphdr()` allocates a new table with *count* entries, regardless of whether one existed previously, and sets the `ELF_F_DIRTY` bit for the table. See `elf_flagdata(3ELF)`. Specifying a zero *count* deletes an existing table. Note this behavior differs from that of `elf32_newehdr()` allowing a program to replace or delete the program header table, changing its size if necessary. See `elf32_getehdr(3ELF)`.

If no program header table exists, the file is not a 32-bit class file, an error occurs, or *elf* is `NULL`, both functions return a null pointer. Additionally, `elf32_newphdr()` returns a null pointer if *count* is `0`.

The table is an array of `Elf32_Phdr` structures, each of which includes the following members:

```
Elf32_Word    p_type;
Elf32_Off    p_offset;
Elf32_Addr    p_vaddr;
Elf32_Addr    p_paddr;
Elf32_Word    p_filesz;
Elf32_Word    p_memsz;
Elf32_Word    p_flags;
Elf32_Word    p_align;
```

The `Elf64_Phdr` structures include the following members:

```
Elf64_Word    p_type;
Elf64_Word    p_flags;
Elf64_Off    p_offset;
Elf64_Addr    p_vaddr;
Elf64_Addr    p_paddr;
Elf64_Xword    p_filesz;
Elf64_Xword    p_memsz;
Elf64_Xword    p_align;
```

For the 64–bit class, replace 32 with 64 as appropriate.

The ELF header's `e_phnum` member tells how many entries the program header table has. See `elf32_getehdr(3ELF)`. A program may inspect this value to determine the size of an existing table; `elf32_newphdr()` automatically sets the member's value to *count*. If the program is building a new file, it is responsible for creating the file's ELF header before creating the program header table.

Attributes

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

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

See Also

`elf(3ELF)`, `elf32_getehdr(3ELF)`, `elf_begin(3ELF)`, `elf_flagdata(3ELF)`, `libelf(3LIB)`, `attributes(5)`

Name elf32_getshdr, elf64_getshdr – retrieve class-dependent section header

Synopsis

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

Elf32_Shdr *elf32_getshdr(Elf_Scn *scn);

Elf64_Shdr *elf64_getshdr(Elf_Scn *scn);
```

Description For a 32-bit class file, `elf32_getshdr()` returns a pointer to a section header for the section descriptor *scn*. Otherwise, the file is not a 32-bit class file, *scn* was `NULL`, or an error occurred; `elf32_getshdr()` then returns `NULL`.

The `elf32_getshdr` header includes the following members:

```
Elf32_Word    sh_name;
Elf32_Word    sh_type;
Elf32_Word    sh_flags;
Elf32_Addr    sh_addr;
Elf32_Off    sh_offset;
Elf32_Word    sh_size;
Elf32_Word    sh_link;
Elf32_Word    sh_info;
Elf32_Word    sh_addralign;
Elf32_Word    sh_entsize;
```

while the `elf64_getshdr` header includes the following members:

```
Elf64_Word    sh_name;
Elf64_Word    sh_type;
Elf64_Xword    sh_flags;
Elf64_Addr    sh_addr;
Elf64_Off    sh_offset;
Elf64_Xword    sh_size;
Elf64_Word    sh_link;
Elf64_Word    sh_info;
Elf64_Xword    sh_addralign;
Elf64_Xword    sh_entsize;
```

For the 64–bit class, replace 32 with 64 as appropriate.

If the program is building a new file, it is responsible for creating the file's ELF header before creating sections.

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 elf(3ELF), elf_flagdata(3ELF), elf_getscn(3ELF), elf_strptr(3ELF), libelf(3LIB), attributes(5)

Name elf32_xlatetof, elf32_xlatetom, elf64_xlatetof, elf64_xlatetom – class-dependent data translation

Synopsis

```
cc [ flag ... ] file... -lelf [ library ... ]
#include <libelf.h>

Elf_Data *elf32_xlatetof(Elf_Data *dst, const Elf_Data *src,
     unsigned encode);

Elf_Data *elf32_xlatetom(Elf_Data *dst, const Elf_Data *src,
     unsigned encode);

Elf_Data *elf64_xlatetof(Elf_Data *dst, const Elf_Data *src,
     unsigned encode);

Elf_Data *elf64_xlatetom(Elf_Data *dst, const Elf_Data *src,
     unsigned encode);
```

Description `elf32_xlatetom()` translates various data structures from their 32-bit class file representations to their memory representations; `elf32_xlatetof()` provides the inverse. This conversion is particularly important for cross development environments. *src* is a pointer to the source buffer that holds the original data; *dst* is a pointer to a destination buffer that will hold the translated copy. *encode* gives the byte encoding in which the file objects are to be represented and must have one of the encoding values defined for the ELF header's `e_ident[EI_DATA]` entry (see `elf_getident(3ELF)`). If the data can be translated, the functions return *dst*. Otherwise, they return `NULL` because an error occurred, such as incompatible types, destination buffer overflow, etc.

`elf_getdata(3ELF)` describes the `Elf_Data` descriptor, which the translation routines use as follows:

`d_buf` Both the source and destination must have valid buffer pointers.

`d_type` This member's value specifies the type of the data to which `d_buf` points and the type of data to be created in the destination. The program supplies a `d_type` value in the source; the library sets the destination's `d_type` to the same value. These values are summarized below.

`d_size` This member holds the total size, in bytes, of the memory occupied by the source data and the size allocated for the destination data. If the destination buffer is not large enough, the routines do not change its original contents. The translation routines reset the destination's `d_size` member to the actual size required, after the translation occurs. The source and destination sizes may differ.

`d_version` This member holds the version number of the objects (desired) in the buffer. The source and destination versions are independent.

Translation routines allow the source and destination buffers to coincide. That is, `dst→d_buf` may equal `src→d_buf`. Other cases where the source and destination buffers overlap give undefined behavior.

```
Elf_Type       32-Bit Memory Type
ELF_T_ADDR    Elf32_Addr
ELF_T_BYTE    unsigned char
ELF_T_DYN    Elf32_Dyn
ELF_T_EHDR    Elf32_Ehdr
ELF_T_HALF    Elf32_Half
ELT_T_OFF    Elf32_Off
ELF_T_PHDR    Elf32_Phdr
ELF_T_REL    Elf32_Rel
ELF_T_RELA    Elf32_Rela
ELF_T_SHDR    Elf32_Shdr
ELF_T_SWORD    Elf32_Sword
ELF_T_SYM    Elf32_Sym
ELF_T_WORD    Elf32_Word
```

Translating buffers of type `ELF_T_BYTE` does not change the byte order.

For the 64–bit class, replace 32 with 64 as appropriate.

Attributes

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

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

See Also

`elf(3ELF)`, `elf32_fsize(3ELF)`, `elf_getdata(3ELF)`, `elf_getident(3ELF)`, `libelf(3LIB)`, `attributes(5)`

Name elf – object file access library

Synopsis

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>
```

Description Functions in the ELF access library let a program manipulate ELF (Executable and Linking Format) object files, archive files, and archive members. The header provides type and function declarations for all library services.

Programs communicate with many of the higher-level routines using an *ELF descriptor*. That is, when the program starts working with a file, `elf_begin(3ELF)` creates an ELF descriptor through which the program manipulates the structures and information in the file. These ELF descriptors can be used both to read and to write files. After the program establishes an ELF descriptor for a file, it may then obtain *section descriptors* to manipulate the sections of the file (see `elf_getscn(3ELF)`). Sections hold the bulk of an object file's real information, such as text, data, the symbol table, and so on. A section descriptor "belongs" to a particular ELF descriptor, just as a section belongs to a file. Finally, *data descriptors* are available through section descriptors, allowing the program to manipulate the information associated with a section. A data descriptor "belongs" to a section descriptor.

Descriptors provide private handles to a file and its pieces. In other words, a data descriptor is associated with one section descriptor, which is associated with one ELF descriptor, which is associated with one file. Although descriptors are private, they give access to data that may be shared. Consider programs that combine input files, using incoming data to create or update another file. Such a program might get data descriptors for an input and an output section. It then could update the output descriptor to reuse the input descriptor's data. That is, the descriptors are distinct, but they could share the associated data bytes. This sharing avoids the space overhead for duplicate buffers and the performance overhead for copying data unnecessarily.

File Classes ELF provides a framework in which to define a family of object files, supporting multiple processors and architectures. An important distinction among object files is the *class*, or capacity, of the file. The 32-bit class supports architectures in which a 32-bit object can represent addresses, file sizes, and so on, as in the following:

Name	Purpose
`Elf32_Addr`	Unsigned address
`Elf32_Half`	Unsigned medium integer
`Elf32_Off`	Unsigned file offset
`Elf32_Sword`	Signed large integer
`Elf32_Word`	Unsigned large integer
`unsigned char`	Unsigned small integer

The 64–bit class works the same as the 32–bit class, substituting 64 for 32 as necessary. Other classes will be defined as necessary, to support larger (or smaller) machines. Some library services deal only with data objects for a specific class, while others are class-independent. To make this distinction clear, library function names reflect their status, as described below.

Data Representation

Conceptually, two parallel sets of objects support cross compilation environments. One set corresponds to file contents, while the other set corresponds to the native memory image of the program manipulating the file. Type definitions supplied by the headers work on the native machine, which may have different data encodings (size, byte order, and so on) than the target machine. Although native memory objects should be at least as big as the file objects (to avoid information loss), they may be bigger if that is more natural for the host machine.

Translation facilities exist to convert between file and memory representations. Some library routines convert data automatically, while others leave conversion as the program's responsibility. Either way, programs that create object files must write file-typed objects to those files; programs that read object files must take a similar view. See `elf32_xlatetof(3ELF)` and `elf32_fsize(3ELF)` for more information.

Programs may translate data explicitly, taking full control over the object file layout and semantics. If the program prefers not to have and exercise complete control, the library provides a higher-level interface that hides many object file details. `elf_begin()` and related functions let a program deal with the native memory types, converting between memory objects and their file equivalents automatically when reading or writing an object file.

ELF Versions

Object file versions allow ELF to adapt to new requirements. *Three independent versions* can be important to a program. First, an application program knows about a particular version by virtue of being compiled with certain headers. Second, the access library similarly is compiled with header files that control what versions it understands. Third, an ELF object file holds a value identifying its version, determined by the ELF version known by the file's creator. Ideally, all three versions would be the same, but they may differ.

If a program's version is newer than the access library, the program might use information unknown to the library. Translation routines might not work properly, leading to undefined behavior. This condition merits installing a new library.

The library's version might be newer than the program's and the file's. The library understands old versions, thus avoiding compatibility problems in this case.

Finally, a file's version might be newer than either the program or the library understands. The program might or might not be able to process the file properly, depending on whether the file has extra information and whether that information can be safely ignored. Again, the safe alternative is to install a new library that understands the file's version.

To accommodate these differences, a program must use `elf_version(3ELF)` to pass its version to the library, thus establishing the *working version* for the process. Using this, the library accepts data from and presents data to the program in the proper representations.

When the library reads object files, it uses each file's version to interpret the data. When writing files or converting memory types to the file equivalents, the library uses the program's working version for the file data.

System Services

As mentioned above, `elf_begin()` and related routines provide a higher-level interface to ELF files, performing input and output on behalf of the application program. These routines assume a program can hold entire files in memory, without explicitly using temporary files. When reading a file, the library routines bring the data into memory and perform subsequent operations on the memory copy. Programs that wish to read or write large object files with this model must execute on a machine with a large process virtual address space. If the underlying operating system limits the number of open files, a program can use `elf_cntl(3ELF)` to retrieve all necessary data from the file, allowing the program to close the file descriptor and reuse it.

Although the `elf_begin()` interfaces are convenient and efficient for many programs, they might be inappropriate for some. In those cases, an application may invoke the `elf32_xlatetom(3ELF)` or `elf32_xlatetof(3ELF)` data translation routines directly. These routines perform no input or output, leaving that as the application's responsibility. By assuming a larger share of the job, an application controls its input and output model.

Library Names

Names associated with the library take several forms.

`elf_`*name*	These class-independent names perform some service, *name*, for the program.
`elf32_`*name*	Service names with an embedded class, `32` here, indicate they work only for the designated class of files.
`Elf_`*Type*	Data types can be class-independent as well, distinguished by *Type*.
`Elf32_`*Type*	Class-dependent data types have an embedded class name, `32` here.
`ELF_C_`*CMD*	Several functions take commands that control their actions. These values are members of the `Elf_Cmd` enumeration; they range from zero through `ELF_C_NUM`–`1`.
`ELF_F_`*FLAG*	Several functions take flags that control library status and/or actions. Flags are bits that may be combined.
`ELF32_FSZ_`*TYPE*	These constants give the file sizes in bytes of the basic ELF types for the 32-bit class of files. See `elf32_fsize()` for more information.
`ELF_K_`*KIND*	The function `elf_kind()` identifies the *KIND* of file associated with an ELF descriptor. These values are members of the `Elf_Kind` enumeration; they range from zero through `ELF_K_NUM`–`1`.
`ELF_T_`*TYPE*	When a service function, such as `elf32_xlatetom()` or `elf32_xlatetof()`, deals with multiple types, names of this form specify the desired *TYPE*. Thus, for example, `ELF_T_EHDR` is directly

related to `Elf32_Ehdr`. These values are members of the `Elf_Type` enumeration; they range from zero through `ELF_T_NUM`–1.

Examples

EXAMPLE 1 An interpretation of elf file.

The basic interpretation of an ELF file consists of:

- opening an ELF object file
- obtaining an ELF descriptor
- analyzing the file using the descriptor.

The following example opens the file, obtains the ELF descriptor, and prints out the names of each section in the file.

```
#include    <fcntl.h>
#include    <stdio.h>
#include    <libelf.h>
#include    <stdlib.h>
#include    <string.h>
static void failure(void);
void
main(int argc, char ** argv)
{
    Elf32_Shdr *    shdr;
    Elf32_Ehdr *    ehdr;
    Elf *        elf;
    Elf_Scn *    scn;
    Elf_Data *    data;
    int        fd;
    unsigned int    cnt;

         /* Open the input file */
    if ((fd = open(argv[1], O_RDONLY)) == -1)
        exit(1);

        /* Obtain the ELF descriptor */
    (void) elf_version(EV_CURRENT);
    if ((elf = elf_begin(fd, ELF_C_READ, NULL)) == NULL)
        failure();

        /* Obtain the .shstrtab data buffer */
    if (((ehdr = elf32_getehdr(elf)) == NULL) ||
        ((scn = elf_getscn(elf, ehdr->e_shstrndx)) == NULL) ||
        ((data = elf_getdata(scn, NULL)) == NULL))
        failure();

        /* Traverse input filename, printing each section */
    for (cnt = 1, scn = NULL; scn = elf_nextscn(elf, scn); cnt++) {
```

EXAMPLE 1 An interpretation of elf file. *(Continued)*

```
        if ((shdr = elf32_getshdr(scn)) == NULL)
            failure();
    (void) printf("[%d]    %s\n", cnt,
        (char *)data->d_buf + shdr->sh_name);
    }
}        /* end main */

static void
failure()
{
    (void) fprintf(stderr, "%s\n", elf_errmsg(elf_errno()));
    exit(1);
}
```

Attributes

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

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

See Also

ar.h(3HEAD), elf32_checksum(3ELF), elf32_fsize(3ELF), elf32_getshdr(3ELF), elf32_xlatetof(3ELF), elf_begin(3ELF), elf_cntl(3ELF), elf_errmsg(3ELF), elf_fill(3ELF), elf_getarhdr(3ELF), elf_getarsym(3ELF), elf_getbase(3ELF), elf_getdata(3ELF), elf_getident(3ELF), elf_getscn(3ELF), elf_hash(3ELF), elf_kind(3ELF), elf_memory(3ELF), elf_rawfile(3ELF), elf_strptr(3ELF), elf_update(3ELF), elf_version(3ELF), gelf(3ELF), libelf(3LIB), attributes(5), lfcompile(5)

ANSI C Programmer's Guide

SPARC only a.out(4)

Notes

Information in the ELF headers is separated into common parts and processor-specific parts. A program can make a processor's information available by including the appropriate header: <sys/elf_*NAME*.h> where *NAME* matches the processor name as used in the ELF file header.

Name	Processor
M32	AT&T WE 32100
SPARC	SPARC

Name	Processor
386	Intel 80386, 80486, Pentium

Other processors will be added to the table as necessary.

To illustrate, a program could use the following code to "see" the processor-specific information for the SPARC based system.

```
#include <libelf.h>
#include <sys/elf_SPARC.h>
```

Without the <sys/elf_SPARC.h> definition, only the common ELF information would be visible.

A program could use the following code to "see" the processor-specific information for the Intel 80386:

```
#include <libelf.h>
#include <sys/elf_386.h>
```

Without the <sys/elf_386.h> definition, only the common ELF information would be visible.

Although reading the objects is rather straightforward, writing/updating them can corrupt the shared offsets among sections. Upon creation, relationships are established among the sections that must be maintained even if the object's size is changed.

Name elf_begin, elf_end, elf_memory, elf_next, elf_rand – process ELF object files

Synopsis

```
cc [ flag... ] file ... -lelf [ library ... ]
#include <libelf.h>

Elf *elf_begin(int fildes, Elf_Cmd cmd, Elf *ref);

int elf_end(Elf *elf);

Elf *elf_memory(char *image, size_t sz);

Elf_Cmd elf_next(Elf *elf);

size_t elf_rand(Elf *elf, size_t offset);
```

Description The `elf_begin()`, `elf_end()`, `elf_memory()`, `elf_next()`, and `elf_rand()` functions work together to process Executable and Linking Format (ELF) object files, either individually or as members of archives. After obtaining an ELF descriptor from `elf_begin()` or `elf_memory()`, the program can read an existing file, update an existing file, or create a new file. The *fildes* argument is an open file descriptor that `elf_begin()` uses for reading or writing. The *elf* argument is an ELF descriptor previously returned from `elf_begin()`. The initial file offset (see `lseek(2)`) is unconstrained, and the resulting file offset is undefined.

The *cmd* argument can take the following values:

`ELF_C_NULL` When a program sets *cmd* to this value, `elf_begin()` returns a null pointer, without opening a new descriptor. *ref* is ignored for this command. See the examples below for more information.

`ELF_C_READ` When a program wants to examine the contents of an existing file, it should set *cmd* to this value. Depending on the value of *ref*, this command examines archive members or entire files. Three cases can occur.

- If *ref* is a null pointer, `elf_begin()` allocates a new ELF descriptor and prepares to process the entire file. If the file being read is an archive, `elf_begin()` also prepares the resulting descriptor to examine the initial archive member on the next call to `elf_begin()`, as if the program had used `elf_next()` or `elf_rand()` to "move" to the initial member.
- If *ref* is a non-null descriptor associated with an archive file, `elf_begin()` lets a program obtain a separate ELF descriptor associated with an individual member. The program should have used `elf_next()` or `elf_rand()` to position *ref* appropriately (except for the initial member, which `elf_begin()` prepares; see the example below). In this case, *fildes* should be the same file descriptor used for the parent archive.
- If *ref* is a non-null ELF descriptor that is not an archive, `elf_begin()` increments the number of activations for the descriptor and returns *ref*, without allocating a new descriptor and without changing the descriptor's read/write permissions. To terminate the descriptor for *ref*,

the program must call `elf_end()` once for each activation. See the examples below for more information.

`ELF_C_RDWR` This command duplicates the actions of `ELF_C_READ` and additionally allows the program to update the file image (see `elf_update(3ELF)`). Using `ELF_C_READ` gives a read-only view of the file, while `ELF_C_RDWR` lets the program read *and* write the file. `ELF_C_RDWR` is not valid for archive members. If *ref* is non-null, it must have been created with the `ELF_C_RDWR` command.

`ELF_C_WRITE` If the program wants to ignore previous file contents, presumably to create a new file, it should set *cmd* to this value. *ref* is ignored for this command.

The `elf_begin()` function operates on all files (including files with zero bytes), providing it can allocate memory for its internal structures and read any necessary information from the file. Programs reading object files can call `elf_kind(3ELF)` or `elf32_getehdr(3ELF)` to determine the file type (only object files have an ELF header). If the file is an archive with no more members to process, or an error occurs, `elf_begin()` returns a null pointer. Otherwise, the return value is a non-null ELF descriptor.

Before the first call to `elf_begin()`, a program must call `elf_version()` to coordinate versions.

The `elf_end()` function is used to terminate an ELF descriptor, *elf*, and to deallocate data associated with the descriptor. Until the program terminates a descriptor, the data remain allocated. A null pointer is allowed as an argument, to simplify error handling. If the program wants to write data associated with the ELF descriptor to the file, it must use `elf_update()` before calling `elf_end()`.

Calling `elf_end()` removes one activation and returns the remaining activation count. The library does not terminate the descriptor until the activation count reaches `0`. Consequently, a `0` return value indicates the ELF descriptor is no longer valid.

The `elf_memory()` function returns a pointer to an ELF descriptor. The ELF image has read operations enabled (`ELF_C_READ`). The *image* argument is a pointer to an image of the Elf file mapped into memory. The *sz* argument is the size of the ELF image. An ELF image that is mapped in with `elf_memory()` can be read and modified, but the ELF image size cannot be changed.

The `elf_next()` function provides sequential access to the next archive member. Having an ELF descriptor, *elf*, associated with an archive member, `elf_next()` prepares the containing archive to access the following member when the program calls `elf_begin()`. After successfully positioning an archive for the next member, `elf_next()` returns the value `ELF_C_READ`. Otherwise, the open file was not an archive, *elf* was `NULL`, or an error occurred, and the return value is `ELF_C_NULL`. In either case, the return value can be passed as an argument to `elf_begin()`, specifying the appropriate action.

The `elf_rand()` function provides random archive processing, preparing *elf* to access an arbitrary archive member. The *elf* argument must be a descriptor for the archive itself, not a member within the archive. The *offset* argument specifies the byte offset from the beginning of the archive to the archive header of the desired member. See `elf_getarsym(3ELF)` for more information about archive member offsets. When `elf_rand()` works, it returns *offset*. Otherwise, it returns `0`, because an error occurred, *elf* was `NULL`, or the file was not an archive (no archive member can have a zero offset). A program can mix random and sequential archive processing.

System Services

When processing a file, the library decides when to read or write the file, depending on the program's requests. Normally, the library assumes the file descriptor remains usable for the life of the ELF descriptor. If, however, a program must process many files simultaneously and the underlying operating system limits the number of open files, the program can use `elf_cntl()` to let it reuse file descriptors. After calling `elf_cntl()` with appropriate arguments, the program can close the file descriptor without interfering with the library.

All data associated with an ELF descriptor remain allocated until `elf_end()` terminates the descriptor's last activation. After the descriptors have been terminated, the storage is released; attempting to reference such data gives undefined behavior. Consequently, a program that deals with multiple input (or output) files must keep the ELF descriptors active until it finishes with them.

Examples

EXAMPLE 1 A sample program of calling the `elf_begin()` function.

A prototype for reading a file appears on the next page. If the file is a simple object file, the program executes the loop one time, receiving a null descriptor in the second iteration. In this case, both `elf` and `arf` will have the same value, the activation count will be `2`, and the program calls `elf_end()` twice to terminate the descriptor. If the file is an archive, the loop processes each archive member in turn, ignoring those that are not object files.

```
if (elf_version(EV_CURRENT) == EV_NONE)
{
    /* library out of date */
    /* recover from error */
}
cmd = ELF_C_READ;
arf = elf_begin(fildes, cmd, (Elf *)0);
while ((elf = elf_begin(fildes, cmd, arf)) != 0)
{
    if ((ehdr = elf32_getehdr(elf)) != 0)
    {
        /* process the file  . . . */
    }
    cmd = elf_next(elf);
    elf_end(elf);
}
elf_end(arf);
```

EXAMPLE 1 A sample program of calling the `elf_begin()` function. *(Continued)*

Alternatively, the next example illustrates random archive processing. After identifying the file as an archive, the program repeatedly processes archive members of interest. For clarity, this example omits error checking and ignores simple object files. Additionally, this fragment preserves the ELF descriptors for all archive members, because it does not call `elf_end()` to terminate them.

```
elf_version(EV_CURRENT);
arf = elf_begin(fildes, ELF_C_READ, (Elf *)0);
if (elf_kind(arf) != ELF_K_AR)
{
    /* not an archive */
}
/* initial processing */
/* set offset =  . . . for desired member header */
while (elf_rand(arf, offset) == offset)
{
    if ((elf = elf_begin(fildes, ELF_C_READ, arf)) == 0)
        break;
    if ((ehdr = elf32_getehdr(elf)) != 0)
    {
        /* process archive member  . . . */
    }
    /* set offset =  . . . for desired member header */
}
```

An archive starts with a "magic string" that has `SARMAG` bytes; the initial archive member follows immediately. An application could thus provide the following function to rewind an archive (the function returns −1 for errors and `0` otherwise).

```
#include <ar.h>
#include <libelf.h>
int
rewindelf(Elf *elf)
{
     if (elf_rand(elf, (size_t)SARMAG) == SARMAG)
          return 0;
     return −1;
}
```

The following outline shows how one might create a new ELF file. This example is simplified to show the overall flow.

```
elf_version(EV_CURRENT);
fildes = open("path/name", O_RDWR|O_TRUNC|O_CREAT, 0666);
if ((elf = elf_begin(fildes, ELF_C_WRITE, (Elf *)0)) == 0)
    return;
```

EXAMPLE 1 A sample program of calling the `elf_begin()` function. *(Continued)*

```
ehdr = elf32_newehdr(elf);
phdr = elf32_newphdr(elf, count);
scn = elf_newscn(elf);
shdr = elf32_getshdr(scn);
data = elf_newdata(scn);
elf_update(elf, ELF_C_WRITE);
elf_end(elf);
```

Finally, the following outline shows how one might update an existing ELF file. Again, this example is simplified to show the overall flow.

```
elf_version(EV_CURRENT);
fildes = open("path/name", O_RDWR);
elf = elf_begin(fildes, ELF_C_RDWR, (Elf *)0);
/* add new or delete old information */
 . . .
/* ensure that the memory image of the file is complete */
elf_update(elf, ELF_C_NULL);
elf_update(elf, ELF_C_WRITE);   /* update file */
elf_end(elf);
```

Notice that both file creation examples open the file with write *and* read permissions. On systems that support `mmap(2)`, the library uses it to enhance performance, and `mmap(2)` requires a readable file descriptor. Although the library can use a write-only file descriptor, the application will not obtain the performance advantages of `mmap(2)`.

Attributes

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

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

See Also

`creat(2)`, `lseek(2)`, `mmap(2)`, `open(2)`, `ar.h(3HEAD)`, `elf(3ELF)`, `elf32_getehdr(3ELF)`, `elf_cntl(3ELF)`, `elf_getarhdr(3ELF)`, `elf_getarsym(3ELF)`, `elf_getbase(3ELF)`, `elf_getdata(3ELF)`, `elf_getscn(3ELF)`, `elf_kind(3ELF)`, `elf_rawfile(3ELF)`, `elf_update(3ELF)`, `elf_version(3ELF)`, `libelf(3LIB)`, `attributes(5)`

Name elf_cntl – control an elf file descriptor

Synopsis

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

int elf_cntl(Elf *elf, Elf_Cmd cmd);
```

Description `elf_cntl()` instructs the library to modify its behavior with respect to an ELF descriptor, *elf*. As `elf_begin(3ELF)` describes, an ELF descriptor can have multiple activations, and multiple ELF descriptors may share a single file descriptor. Generally, `elf_cntl()` commands apply to all activations of *elf*. Moreover, if the ELF descriptor is associated with an archive file, descriptors for members within the archive will also be affected as described below. Unless stated otherwise, operations on archive members do not affect the descriptor for the containing archive.

The *cmd* argument tells what actions to take and may have the following values:

`ELF_C_FDDONE` This value tells the library not to use the file descriptor associated with *elf*. A program should use this command when it has requested all the information it cares to use and wishes to avoid the overhead of reading the rest of the file. The memory for all completed operations remains valid, but later file operations, such as the initial `elf_getdata()` for a section, will fail if the data are not in memory already.

`ELF_C_FDREAD` This command is similar to `ELF_C_FDDONE`, except it forces the library to read the rest of the file. A program should use this command when it must close the file descriptor but has not yet read everything it needs from the file. After `elf_cntl()` completes the `ELF_C_FDREAD` command, future operations, such as `elf_getdata()`, will use the memory version of the file without needing to use the file descriptor.

If `elf_cntl()` succeeds, it returns `0`. Otherwise *elf* was `NULL` or an error occurred, and the function returns `−1`.

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

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

See Also `elf(3ELF)`, `elf_begin(3ELF)`, `elf_getdata(3ELF)`, `elf_rawfile(3ELF)`, `libelf(3LIB)`, `attributes(5)`

Notes If the program wishes to use the "raw" operations (see `elf_rawdata()`, which `elf_getdata(3ELF)` describes, and `elf_rawfile(3ELF)`) after disabling the file descriptor with `ELF_C_FDDONE` or `ELF_C_FDREAD`, it must execute the raw operations explicitly beforehand. Otherwise, the raw file operations will fail. Calling `elf_rawfile()` makes the entire image available, thus supporting subsequent `elf_rawdata()` calls.

Name

elf_errmsg, elf_errno – error handling

Synopsis

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

const char *elf_errmsg(int err);

int elf_errno(void);
```

Description

If an ELF library function fails, a program can call `elf_errno()` to retrieve the library's internal error number. As a side effect, this function resets the internal error number to `0`, which indicates no error.

The `elf_errmsg()` function takes an error number, *err*, and returns a null-terminated error message (with no trailing new-line) that describes the problem. A zero *err* retrieves a message for the most recent error. If no error has occurred, the return value is a null pointer (not a pointer to the null string). Using *err* of −1 also retrieves the most recent error, except it guarantees a non-null return value, even when no error has occurred. If no message is available for the given number, `elf_errmsg()` returns a pointer to an appropriate message. This function does not have the side effect of clearing the internal error number.

Examples

EXAMPLE 1 A sample program of calling the `elf_errmsg()` function.

The following fragment clears the internal error number and checks it later for errors. Unless an error occurs after the first call to `elf_errno()`, the next call will return `0`.

```
(void)elf_errno( );
/* processing  . . . */
while (more_to_do)
{
    if ((err = elf_errno( )) != 0)
    {
        /* print msg */
        msg = elf_errmsg(err);
    }
}
```

Attributes

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

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

See Also

elf(3ELF), libelf(3LIB), attributes(5)

Name elf_fill – set fill byte

Synopsis

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

void elf_fill(int fill);
```

Description Alignment constraints for ELF files sometimes require the presence of "holes." For example, if the data for one section are required to begin on an eight-byte boundary, but the preceding section is too "short," the library must fill the intervening bytes. These bytes are set to the *fill* character. The library uses zero bytes unless the application supplies a value. See `elf_getdata(3ELF)` for more information about these holes.

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

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

See Also `elf(3ELF)`, `elf_flagdata(3ELF)`, `elf_getdata(3ELF)`, `elf_update(3ELF)`, `libelf(3LIB)`, `attributes(5)`

Notes An application can assume control of the object file organization by setting the `ELF_F_LAYOUT` bit (see `elf_flagdata(3ELF)`). When this is done, the library does *not* fill holes.

Name elf_flagdata, elf_flagehdr, elf_flagelf, elf_flagphdr, elf_flagscn, elf_flagshdr – manipulate flags

Synopsis

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

unsigned elf_flagdata(Elf_Data *data, Elf_Cmd cmd, unsigned flags);

unsigned elf_flagehdr(Elf *elf, Elf_Cmd cmd, unsigned flags);

unsigned elf_flagelf(Elf *elf, Elf_Cmd cmd, unsigned flags);

unsigned elf_flagphdr(Elf *elf, Elf_Cmd cmd, unsigned flags);

unsigned elf_flagscn(Elf_Scn *scn, Elf_Cmd cmd, unsigned flags);

unsigned elf_flagshdr(Elf_Scn *scn, Elf_Cmd cmd, unsigned flags);
```

Description These functions manipulate the flags associated with various structures of an ELF file. Given an ELF descriptor (*elf*), a data descriptor (*data*), or a section descriptor (*scn*), the functions may set or clear the associated status bits, returning the updated bits. A null descriptor is allowed, to simplify error handling; all functions return `0` for this degenerate case.

cmd may have the following values:

`ELF_C_CLR` The functions clear the bits that are asserted in *flags*. Only the non-zero bits in *flags* are cleared; zero bits do not change the status of the descriptor.

`ELF_C_SET` The functions set the bits that are asserted in *flags*. Only the non-zero bits in *flags* are set; zero bits do not change the status of the descriptor.

Descriptions of the defined *flags* bits appear below:

`ELF_F_DIRTY` When the program intends to write an ELF file, this flag asserts the associated information needs to be written to the file. Thus, for example, a program that wished to update the ELF header of an existing file would call `elf_flagehdr()` with this bit set in *flags* and *cmd* equal to `ELF_C_SET`. A later call to `elf_update()` would write the marked header to the file.

`ELF_F_LAYOUT` Normally, the library decides how to arrange an output file. That is, it automatically decides where to place sections, how to align them in the file, etc. If this bit is set for an ELF descriptor, the program assumes responsibility for determining all file positions. This bit is meaningful only for `elf_flagelf()` and applies to the entire file associated with the descriptor.

When a flag bit is set for an item, it affects all the subitems as well. Thus, for example, if the program sets the `ELF_F_DIRTY` bit with `elf_flagelf()`, the entire logical file is “dirty."

Examples

EXAMPLE 1 A sample display of calling the `elf_flagdata()` function.

The following fragment shows how one might mark the ELF header to be written to the output file:

```
/* dirty ehdr  . . . */
ehdr = elf32_getehdr(elf);
elf_flagehdr(elf, ELF_C_SET, ELF_F_DIRTY);
```

Attributes

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

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

See Also

`elf(3ELF)`, `elf32_getehdr(3ELF)`, `elf_getdata(3ELF)`, `elf_update(3ELF)`, `attributes(5)`

Name elf_getarhdr – retrieve archive member header

Synopsis

```
cc [ flag ... ] file ... -lelf [ library... ]
#include <libelf.h>

Elf_Arhdr *elf_getarhdr(Elf *elf);
```

Description `elf_getarhdr()` returns a pointer to an archive member header, if one is available for the ELF descriptor *elf*. Otherwise, no archive member header exists, an error occurred, or *elf* was null; `elf_getarhdr()` then returns a null value. The header includes the following members.

```
char    *ar_name;
time_t    ar_date;
uid_t    ar_uid;
gid_t    ar_gid;
mode_t    ar_mode;
off_t    ar_size;
char    *ar_rawname;
```

An archive member name, available through `ar_name`, is a null-terminated string, with the `ar` format control characters removed. The `ar_rawname` member holds a null-terminated string that represents the original name bytes in the file, including the terminating slash and trailing blanks as specified in the archive format.

In addition to "regular" archive members, the archive format defines some special members. All special member names begin with a slash (`/`), distinguishing them from regular members (whose names may not contain a slash). These special members have the names (`ar_name`) defined below.

`/` This is the archive symbol table. If present, it will be the first archive member. A program may access the archive symbol table through `elf_getarsym()`. The information in the symbol table is useful for random archive processing (see `elf_rand()` on `elf_begin(3ELF)`).

`//` This member, if present, holds a string table for long archive member names. An archive member's header contains a 16-byte area for the name, which may be exceeded in some file systems. The library automatically retrieves long member names from the string table, setting `ar_name` to the appropriate value.

Under some error conditions, a member's name might not be available. Although this causes the library to set `ar_name` to a null pointer, the `ar_rawname` member will be set as usual.

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 ar.h(3HEAD), elf(3ELF), elf_begin(3ELF), elf_getarsym(3ELF), libelf(3LIB), attributes(5)

Name elf_getarsym – retrieve archive symbol table

Synopsis

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

Elf_Arsym *elf_getarsym(Elf *elf, size_t *ptr);
```

Description The `elf_getarsym()` function returns a pointer to the archive symbol table, if one is available for the ELF descriptor *elf*. Otherwise, the archive doesn't have a symbol table, an error occurred, or *elf* was null; `elf_getarsym()` then returns a null value. The symbol table is an array of structures that include the following members.

```
char    *as_name;
size_t    as_off;
unsigned long    as_hash;
```

These members have the following semantics:

`as_name` A pointer to a null-terminated symbol name resides here.

`as_off` This value is a byte offset from the beginning of the archive to the member's header. The archive member residing at the given offset defines the associated symbol. Values in `as_off` may be passed as arguments to `elf_rand()`. See `elf_begin(3ELF)` to access the desired archive member.

`as_hash` This is a hash value for the name, as computed by `elf_hash()`.

If *ptr* is non-null, the library stores the number of table entries in the location to which *ptr* points. This value is set to `0` when the return value is `NULL`. The table's last entry, which is included in the count, has a null `as_name`, a zero value for `as_off`, and `~0UL` for `as_hash`.

The hash value returned is guaranteed not to be the bit pattern of all ones (`~0UL`).

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

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

See Also `ar.h(3HEAD)`, `elf(3ELF)`, `elf_begin(3ELF)`, `elf_getarhdr(3ELF)`, `elf_hash(3ELF)`, `libelf(3LIB)`, `attributes(5)`

Name elf_getbase – get the base offset for an object file

Synopsis

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

off_t elf_getbase(Elf *elf);
```

Description The `elf_getbase()` function returns the file offset of the first byte of the file or archive member associated with *elf*, if it is known or obtainable, and −1 otherwise. A null *elf* is allowed, to simplify error handling; the return value in this case is −1. The base offset of an archive member is the beginning of the member's information, *not* the beginning of the archive member header.

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

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

See Also `ar.h(3HEAD)`, `elf(3ELF)`, `elf_begin(3ELF)`, `libelf(3LIB)`, `attributes(5)`

Name elf_getdata, elf_newdata, elf_rawdata – get section data

Synopsis

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

Elf_Data *elf_getdata(Elf_Scn *scn, Elf_Data *data);

Elf_Data *elf_newdata(Elf_Scn *scn);

Elf_Data *elf_rawdata(Elf_Scn *scn, Elf_Data *data);
```

Description These functions access and manipulate the data associated with a section descriptor, *scn*. When reading an existing file, a section will have a single data buffer associated with it. A program may build a new section in pieces, however, composing the new data from multiple data buffers. For this reason, the data for a section should be viewed as a list of buffers, each of which is available through a data descriptor.

The `elf_getdata()` function lets a program step through a section's data list. If the incoming data descriptor, *data*, is null, the function returns the first buffer associated with the section. Otherwise, *data* should be a data descriptor associated with *scn*, and the function gives the program access to the next data element for the section. If *scn* is null or an error occurs, `elf_getdata()` returns a null pointer.

The `elf_getdata()` function translates the data from file representations into memory representations (see `elf32_xlatetof(3ELF)`) and presents objects with memory data types to the program, based on the file's *class* (see `elf(3ELF)`). The working library version (see `elf_version(3ELF)`) specifies what version of the memory structures the program wishes `elf_getdata()` to present.

The `elf_newdata()` function creates a new data descriptor for a section, appending it to any data elements already associated with the section. As described below, the new data descriptor appears empty, indicating the element holds no data. For convenience, the descriptor's type (`d_type` below) is set to `ELF_T_BYTE`, and the version (`d_version` below) is set to the working version. The program is responsible for setting (or changing) the descriptor members as needed. This function implicitly sets the `ELF_F_DIRTY` bit for the section's data (see `elf_flagdata(3ELF)`). If *scn* is null or an error occurs, `elf_newdata()` returns a null pointer.

The `elf_rawdata()` function differs from `elf_getdata()` by returning only uninterpreted bytes, regardless of the section type. This function typically should be used only to retrieve a section image from a file being read, and then only when a program must avoid the automatic data translation described below. Moreover, a program may not close or disable (see `elf_cntl(3ELF)`) the file descriptor associated with *elf* before the initial raw operation, because `elf_rawdata()` might read the data from the file to ensure it doesn't interfere with `elf_getdata()`. See `elf_rawfile(3ELF)` for a related facility that applies to the entire file. When `elf_getdata()` provides the right translation, its use is recommended over `elf_rawdata()`. If *scn* is null or an error occurs, `elf_rawdata()` returns a null pointer.

The `Elf_Data` structure includes the following members:

```
void        *d_buf;
Elf_Type    d_type;
size_t      d_size;
off_t       d_off;
size_t      d_align;
unsigned    d_version;
```

These members are available for direct manipulation by the program. Descriptions appear below.

`d_buf` A pointer to the data buffer resides here. A data element with no data has a null pointer.

`d_type` This member's value specifies the type of the data to which `d_buf` points. A section's type determines how to interpret the section contents, as summarized below.

`d_size` This member holds the total size, in bytes, of the memory occupied by the data. This may differ from the size as represented in the file. The size will be zero if no data exist. (See the discussion of `SHT_NOBITS` below for more information.)

`d_off` This member gives the offset, within the section, at which the buffer resides. This offset is relative to the file's section, not the memory object's.

`d_align` This member holds the buffer's required alignment, from the beginning of the section. That is, `d_off` will be a multiple of this member's value. For example, if this member's value is `4`, the beginning of the buffer will be four-byte aligned within the section. Moreover, the entire section will be aligned to the maximum of its constituents, thus ensuring appropriate alignment for a buffer within the section and within the file.

`d_version` This member holds the version number of the objects in the buffer. When the library originally read the data from the object file, it used the working version to control the translation to memory objects.

Data Alignment

As mentioned above, data buffers within a section have explicit alignment constraints. Consequently, adjacent buffers sometimes will not abut, causing "holes" within a section. Programs that create output files have two ways of dealing with these holes.

First, the program can use `elf_fill()` to tell the library how to set the intervening bytes. When the library must generate gaps in the file, it uses the fill byte to initialize the data there. The library's initial fill value is `0`, and `elf_fill()` lets the application change that.

Second, the application can generate its own data buffers to occupy the gaps, filling the gaps with values appropriate for the section being created. A program might even use different fill values for different sections. For example, it could set text sections' bytes to no-operation instructions, while filling data section holes with zero. Using this technique, the library finds no holes to fill, because the application eliminated them.

Section and Memory Types

The `elf_getdata()` function interprets sections' data according to the section type, as noted in the section header available through `elf32_getshdr()`. The following table shows the section types and how the library represents them with memory data types for the 32-bit file class. Other classes would have similar tables. By implication, the memory data types control translation by `elf32_xlatetof(3ELF)`

Section Type	Elf_Type	32-bit Type
SHT_DYNAMIC	ELF_T_DYN	Elf32_Dyn
SHT_DYNSYM	ELF_T_SYM	Elf32_Sym
SHT_FINI_ARRAY	ELF_T_ADDR	Elf32_Addr
SHT_GROUP	ELF_T_WORD	Elf32_Word
SHT_HASH	ELF_T_WORD	Elf32_Word
SHT_INIT_ARRAY	ELF_T_ADDR	Elf32_Addr
SHT_NOBITS	ELF_T_BYTE	unsigned char
SHT_NOTE	ELF_T_NOTE	unsigned char
SHT_NULL	*none*	*none*
SHT_PREINIT_ARRAY	ELF_T_ADDR	Elf32_Addr
SHT_PROGBITS	ELF_T_BYTE	unsigned char
SHT_REL	ELF_T_REL	Elf32_Rel
SHT_RELA	ELF_T_RELA	Elf32_Rela
SHT_STRTAB	ELF_T_BYTE	unsigned char
SHT_SYMTAB	ELF_T_SYM	Elf32_Sym
SHT_SUNW_comdat	ELF_T_BYTE	unsigned char
SHT_SUNW_move	ELF_T_MOVE	Elf32_Move (sparc)
SHT_SUNW_move	ELF_T_MOVEP	Elf32_Move (ia32)
SHT_SUNW_syminfo	ELF_T_SYMINFO	Elf32_Syminfo
SHT_SUNW_verdef	ELF_T_VDEF	Elf32_Verdef
SHT_SUNW_verneed	ELF_T_VNEED	Elf32_Verneed
SHT_SUNW_versym	ELF_T_HALF	Elf32_Versym
other	ELF_T_BYTE	unsigned char

The `elf_rawdata()` function creates a buffer with type `ELF_T_BYTE`.

As mentioned above, the program's working version controls what structures the library creates for the application. The library similarly interprets section types according to the versions. If a section type belongs to a version newer than the application's working version, the library does not translate the section data. Because the application cannot know the data format in this case, the library presents an untranslated buffer of type `ELF_T_BYTE`, just as it would for an unrecognized section type.

A section with a special type, `SHT_NOBITS`, occupies no space in an object file, even when the section header indicates a non-zero size. `elf_getdata()` and `elf_rawdata()` work on such a section, setting the *data* structure to have a null buffer pointer and the type indicated above. Although no data are present, the `d_size` value is set to the size from the section header. When a program is creating a new section of type `SHT_NOBITS`, it should use `elf_newdata()` to add data buffers to the section. These empty data buffers should have the `d_size` members set to the desired size and the `d_buf` members set to `NULL`.

Examples

EXAMPLE 1 A sample program of calling `elf_getdata()`.

The following fragment obtains the string table that holds section names (ignoring error checking). See `elf_strptr(3ELF)` for a variation of string table handling.

```
ehdr = elf32_getehdr(elf);
scn = elf_getscn(elf, (size_t)ehdr->e_shstrndx);
shdr = elf32_getshdr(scn);
if (shdr->sh_type != SHT_STRTAB)
{
/* not a string table */
}
data = 0;
if ((data = elf_getdata(scn, data)) == 0 || data->d_size == 0)
{
/* error or no data */
}
```

The `e_shstrndx` member in an ELF header holds the section table index of the string table. The program gets a section descriptor for that section, verifies it is a string table, and then retrieves the data. When this fragment finishes, `data->d_buf` points at the first byte of the string table, and `data->d_size` holds the string table's size in bytes.

Attributes

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

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

See Also elf(3ELF), elf32_getehdr(3ELF), elf64_getehdr(3ELF), elf32_getshdr(3ELF), elf64_getshdr(3ELF), elf32_xlatetof(3ELF), elf64_xlatetof(3ELF), elf_cntl(3ELF), elf_fill(3ELF), elf_flagdata(3ELF), elf_getscn(3ELF), elf_rawfile(3ELF), elf_strptr(3ELF), elf_version(3ELF), libelf(3LIB), attributes(5)

Name elf_getident, elf_getshnum, elf_getshstrndx – retrieve ELF header data

Synopsis

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

char * elf_getident(Elf *elf, size_t *dst);

int elf_getshnum(Elf *elf, size_t *dst);

int elf_getshstrndx(Elf *elf, size_t *dst);
```

Description As elf(3ELF) explains, ELF provides a framework for various classes of files, where basic objects may have 32 bits, 64 bits, etc. To accommodate these differences, without forcing the larger sizes on smaller machines, the initial bytes in an ELF file hold identification information common to all file classes. Every ELF header's `e_ident` has `EI_NIDENT` bytes with the following interpretation:

e_ident Index	Value	Purpose
EI_MAG0	ELFMAG0	File identification
EI_MAG1	ELFMAG1	
EI_MAG2	ELFMAG2	
EI_MAG3	ELFMAG3	
EI_CLASS	ELFCLASSNONE	File class
	ELFCLASS32	
	ELFCLASS64	
EI_DATA	ELFDATANONE	Data encoding
	ELFDATA2LSB	
	ELFDATA2MSB	
EI_VERSION	EV_CURRENT	File version
7-15	0	Unused, set to zero

Other kinds of files (see elf_kind(3ELF)) also may have identification data, though they would not conform to e_ident.

elf_getident() returns a pointer to the file's "initial bytes." If the library recognizes the file, a conversion from the file image to the memory image may occur. In any case, the identification bytes are guaranteed not to have been modified, though the size of the unmodified area depends on the file type. If *dst* is non-null, the library stores the number of identification bytes in the location to which *dst* points. If no data are present, *elf* is null, or an error occurs, the return value is a null pointer, with 0 stored through *dst*, if *dst* is non-null.

The elf_getshnum() function obtains the number of sections recorded in the ELF file. The number of sections in a file is typically recorded in the e_shnum field of the ELF header, though a file that requires ELF Extended Sections records the value 0 in the e_shnum field and records the number of sections in the sh_size field of section header 0. See USAGE. *dst* points to the location where the number of sections will be stored. If a call to elf_newscn(3ELF) using the same *elf* descriptor has been performed, then the value obtained by elf_getshnum() is only valid after a successful call to elf_update(3ELF). If *elf* is NULL or an error occurs, elf_getshnum() returns -1.

The elf_getshstrndx() function obtains the section index of the string table associated with the section headers in the ELF file. The section header string table index is typically recorded in the e_shstrndx field of the ELF header, though a file that requires ELF Extended Sections records the value SHN_XINDEX in the e_shstrndx field and records the string table index in the sh_link field of section header 0. See USAGE. The *dst* argument points to the location where the section header string table index is stored. If *elf* is NULL or an error occurs, elf_getshstrndx() returns -1.

Usage ELF Extended Sections are employed to allow an ELF file to contain more than 0xff00 (SHN_LORESERVE) section. See the *Linker and Libraries Guide* for more information.

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

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

See Also elf(3ELF), elf32_getehdr(3ELF), elf_begin(3ELF), elf_kind(3ELF), elf_newscn(3ELF), elf_rawfile(3ELF), elf_update(3ELF), libelf(3LIB), attributes(5)

Linker and Libraries Guide

Name elf_getscn, elf_ndxscn, elf_newscn, elf_nextscn – get section information

Synopsis

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

Elf_Scn *elf_getscn(Elf *elf, size_t index);

size_t elf_ndxscn(Elf_Scn *scn);

Elf_Scn *elf_newscn(Elf *elf);

Elf_Scn *elf_nextscn(Elf *elf, Elf_Scn *scn);
```

Description These functions provide indexed and sequential access to the sections associated with the ELF descriptor *elf*. If the program is building a new file, it is responsible for creating the file's ELF header before creating sections; see `elf32_getehdr(3ELF)`.

The `elf_getscn()` function returns a section descriptor, given an *index* into the file's section header table. Note that the first "real" section has an index of `1`. Although a program can get a section descriptor for the section whose *index* is `0` (`SHN_UNDEF`, the undefined section), the section has no data and the section header is "empty" (though present). If the specified section does not exist, an error occurs, or *elf* is `NULL`, `elf_getscn()` returns a null pointer.

The `elf_newscn()` function creates a new section and appends it to the list for *elf*. Because the `SHN_UNDEF` section is required and not "interesting" to applications, the library creates it automatically. Thus the first call to `elf_newscn()` for an ELF descriptor with no existing sections returns a descriptor for section 1. If an error occurs or *elf* is `NULL`, `elf_newscn()` returns a null pointer.

After creating a new section descriptor, the program can use `elf32_getshdr()` to retrieve the newly created, "clean" section header. The new section descriptor will have no associated data (see `elf_getdata(3ELF)`). When creating a new section in this way, the library updates the `e_shnum` member of the ELF header and sets the `ELF_F_DIRTY` bit for the section (see `elf_flagdata(3ELF)`). If the program is building a new file, it is responsible for creating the file's ELF header (see `elf32_getehdr(3ELF)`) before creating new sections.

The `elf_nextscn()` function takes an existing section descriptor, *scn*, and returns a section descriptor for the next higher section. One may use a null *scn* to obtain a section descriptor for the section whose index is `1` (skipping the section whose index is `SHN_UNDEF`). If no further sections are present or an error occurs, `elf_nextscn()` returns a null pointer.

The `elf_ndxscn()` function takes an existing section descriptor, *scn*, and returns its section table index. If *scn* is null or an error occurs, `elf_ndxscn()` returns `SHN_UNDEF`.

Examples **EXAMPLE 1** A sample of calling `elf_getscn()` function.

An example of sequential access appears below. Each pass through the loop processes the next section in the file; the loop terminates when all sections have been processed.

EXAMPLE 1 A sample of calling `elf_getscn()` function. *(Continued)*

```
scn = 0;
while ((scn = elf_nextscn(elf, scn)) != 0)
{
    /* process section */
}
```

Attributes

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

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

See Also

`elf(3ELF)`, `elf32_getehdr(3ELF)`, `elf32_getshdr(3ELF)`, `elf_begin(3ELF)`, `elf_flagdata(3ELF)`, `elf_getdata(3ELF)`, `libelf(3LIB)`, `attributes(5)`

Name elf_hash – compute hash value

Synopsis

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

unsigned long elf_hash(const char *name);
```

Description The `elf_hash()` function computes a hash value, given a null terminated string, *name*. The returned hash value, *h*, can be used as a bucket index, typically after computing *h* mod x to ensure appropriate bounds.

Hash tables may be built on one machine and used on another because `elf_hash()` uses unsigned arithmetic to avoid possible differences in various machines' signed arithmetic. Although *name* is shown as `char*` above, `elf_hash()` treats it as `unsigned char*` to avoid sign extension differences. Using `char*` eliminates type conflicts with expressions such as `elf_hash(`*name*`)`.

ELF files' symbol hash tables are computed using this function (see `elf_getdata(3ELF)` and `elf32_xlatetof(3ELF)`). The hash value returned is guaranteed not to be the bit pattern of all ones (`~0UL`).

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

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

See Also `elf(3ELF)`, `elf32_xlatetof(3ELF)`, `elf_getdata(3ELF)`, `libelf(3LIB)`, `attributes(5)`

Name elf_kind – determine file type

Synopsis

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

Elf_Kind elf_kind(Elf *elf);
```

Description This function returns a value identifying the kind of file associated with an ELF descriptor (*elf*). Defined values are below:

`ELF_K_AR` The file is an archive [see `ar.h`(3HEAD)]. An ELF descriptor may also be associated with an archive *member*, not the archive itself, and then `elf_kind()` identifies the member's type.

`ELF_K_COFF` The file is a COFF object file. `elf_begin`(3ELF) describes the library's handling for COFF files.

`ELF_K_ELF` The file is an ELF file. The program may use `elf_getident()` to determine the class. Other functions, such as `elf32_getehdr()`, are available to retrieve other file information.

`ELF_K_NONE` This indicates a kind of file unknown to the library.

Other values are reserved, to be assigned as needed to new kinds of files. *elf* should be a value previously returned by `elf_begin()`. A null pointer is allowed, to simplify error handling, and causes `elf_kind()` to return `ELF_K_NONE`.

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

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

See Also `ar.h`(3HEAD), `elf`(3ELF), `elf32_getehdr`(3ELF), `elf_begin`(3ELF), `elf_getident`(3ELF), `libelf`(3LIB), `attributes`(5)

Name elf_rawfile – retrieve uninterpreted file contents

Synopsis

```
cc [ flag... ] file ... -lelf [ library ... ]
#include <libelf.h>

char *elf_rawfile(Elf *elf, size_t *ptr);
```

Description The `elf_rawfile()` function returns a pointer to an uninterpreted byte image of the file. This function should be used only to retrieve a file being read. For example, a program might use `elf_rawfile()` to retrieve the bytes for an archive member.

A program may not close or disable (see `elf_cntl(3ELF)`) the file descriptor associated with *elf* before the initial call to `elf_rawfile()`, because `elf_rawfile()` might have to read the data from the file if it does not already have the original bytes in memory. Generally, this function is more efficient for unknown file types than for object files. The library implicitly translates object files in memory, while it leaves unknown files unmodified. Thus, asking for the uninterpreted image of an object file may create a duplicate copy in memory.

`elf_rawdata()` is a related function, providing access to sections within a file. See `elf_getdata(3ELF)`.

If *ptr* is non-null, the library also stores the file's size, in bytes, in the location to which *ptr* points. If no data are present, *elf* is null, or an error occurs, the return value is a null pointer, with `0` stored through *ptr*, if *ptr* is non-null.

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

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

See Also `elf(3ELF)`, `elf32_getehdr(3ELF)`, `elf_begin(3ELF)`, `elf_cntl(3ELF)`, `elf_getdata(3ELF)`, `elf_getident(3ELF)`, `elf_kind(3ELF)`, `libelf(3LIB)`, `attributes(5)`

Notes A program that uses `elf_rawfile()` and that also interprets the same file as an object file potentially has two copies of the bytes in memory. If such a program requests the raw image first, before it asks for translated information (through such functions as `elf32_getehdr()`, `elf_getdata()`, and so on), the library "freezes" its original memory copy for the raw image. It then uses this frozen copy as the source for creating translated objects, without reading the file again. Consequently, the application should view the raw file image returned by `elf_rawfile()` as a read-only buffer, unless it wants to alter its own view of data subsequently translated. In any case, the application may alter the translated objects without changing bytes visible in the raw image.

Multiple calls to `elf_rawfile()` with the same ELF descriptor return the same value; the library does not create duplicate copies of the file.

Name elf_strptr – make a string pointer

Synopsis

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

char *elf_strptr(Elf *elf, size_t section, size_t offset);
```

Description The `elf_strptr()` function converts a string section *offset* to a string pointer. *elf* identifies the file in which the string section resides, and *section* identifies the section table index for the strings. `elf_strptr()` normally returns a pointer to a string, but it returns a null pointer when *elf* is null, *section* is invalid or is not a section of type `SHT_STRTAB`, the section data cannot be obtained, *offset* is invalid, or an error occurs.

Examples **EXAMPLE 1** A sample program of calling `elf_strptr()` function.

A prototype for retrieving section names appears below. The file header specifies the section name string table in the `e_shstrndx` member. The following code loops through the sections, printing their names.

```
/* handle the error */
if ((ehdr = elf32_getehdr(elf)) == 0) {
    return;
}
ndx = ehdr->e_shstrndx;
scn = 0;
while ((scn = elf_nextscn(elf, scn)) != 0) {
    char    *name = 0;
    if ((shdr = elf32_getshdr(scn)) != 0)
                name = elf_strptr(elf, ndx, (size_t)shdr->sh_name);
    printf("'%s'\n", name? name: "(null)");
}
```

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

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

See Also `elf(3ELF)`, `elf32_getshdr(3ELF)`, `elf32_xlatetof(3ELF)`, `elf_getdata(3ELF)`, `libelf(3LIB)`, `attributes(5)`

Notes A program may call `elf_getdata()` to retrieve an entire string table section. For some applications, that would be both more efficient and more convenient than using `elf_strptr()`.

Name elf_update – update an ELF descriptor

Synopsis

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

off_t elf_update(Elf *elf, Elf_Cmd cmd);
```

Description The `elf_update()` function causes the library to examine the information associated with an ELF descriptor, *elf*, and to recalculate the structural data needed to generate the file's image.

The *cmd* argument can have the following values:

`ELF_C_NULL` This value tells `elf_update()` to recalculate various values, updating only the ELF descriptor's memory structures. Any modified structures are flagged with the `ELF_F_DIRTY` bit. A program thus can update the structural information and then reexamine them without changing the file associated with the ELF descriptor. Because this does not change the file, the ELF descriptor may allow reading, writing, or both reading and writing (see `elf_begin(3ELF)`).

`ELF_C_WRITE` If *cmd* has this value, `elf_update()` duplicates its `ELF_C_NULL` actions and also writes any "dirty" information associated with the ELF descriptor to the file. That is, when a program has used `elf_getdata(3ELF)` or the `elf_flagdata(3ELF)` facilities to supply new (or update existing) information for an ELF descriptor, those data will be examined, coordinated, translated if necessary (see `elf32_xlatetof(3ELF)`), and written to the file. When portions of the file are written, any `ELF_F_DIRTY` bits are reset, indicating those items no longer need to be written to the file (see `elf_flagdata(3ELF)`). The sections' data are written in the order of their section header entries, and the section header table is written to the end of the file. When the ELF descriptor was created with `elf_begin()`, it must have allowed writing the file. That is, the `elf_begin()` command must have been either `ELF_C_RDWR` or `ELF_C_WRITE`.

If `elf_update()` succeeds, it returns the total size of the file image (not the memory image), in bytes. Otherwise an error occurred, and the function returns −1.

When updating the internal structures, `elf_update()` sets some members itself. Members listed below are the application's responsibility and retain the values given by the program.

The following table shows ELF Header members:

Member	Notes
e_ident[EI_DATA]	Library controls other `e_ident` values

e_type	
e_machine	
e_version	
e_entry	
e_phoff	Only when `ELF_F_LAYOUT` asserted
e_shoff	Only when `ELF_F_LAYOUT` asserted
e_flags	
e_shstrndx	

The following table shows the Program Header members:

Member	Notes
p_type	The application controls all
p_offset	program header entries
p_vaddr	
p_paddr	
p_filesz	
p_memsz	
p_flags	
p_align	

The following table shows the Section Header members:

Member	Notes
sh_name	
sh_type	
sh_flags	
sh_addr	

sh_offset	Only when ELF_F_LAYOUT asserted
sh_size	Only when ELF_F_LAYOUT asserted
sh_link	
sh_info	
sh_addralign	Only when ELF_F_LAYOUT asserted
sh_entsize	

The following table shows the Data Descriptor members:

Member	Notes
d_buf	
d_type	
d_size	
d_off	Only when ELF_F_LAYOUT asserted
d_align	
d_version	

Note that the program is responsible for two particularly important members (among others) in the ELF header. The `e_version` member controls the version of data structures written to the file. If the version is `EV_NONE`, the library uses its own internal version. The `e_ident[EI_DATA]` entry controls the data encoding used in the file. As a special case, the value may be `ELFDATANONE` to request the native data encoding for the host machine. An error occurs in this case if the native encoding doesn't match a file encoding known by the library.

Further note that the program is responsible for the `sh_entsize` section header member. Although the library sets it for sections with known types, it cannot reliably know the correct value for all sections. Consequently, the library relies on the program to provide the values for unknown section types. If the entry size is unknown or not applicable, the value should be set to `0`.

When deciding how to build the output file, `elf_update()` obeys the alignments of individual data buffers to create output sections. A section's most strictly aligned data buffer controls the section's alignment. The library also inserts padding between buffers, as necessary, to ensure the proper alignment of each buffer.

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

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

See Also elf(3ELF), elf32_fsize(3ELF), elf32_getehdr(3ELF), elf32_getshdr(3ELF), elf32_xlatetof(3ELF), elf_begin(3ELF), elf_flagdata(3ELF), elf_getdata(3ELF), libelf(3LIB), attributes(5)

Notes As mentioned above, the ELF_C_WRITE command translates data as necessary, before writing them to the file. This translation is *not* always transparent to the application program. If a program has obtained pointers to data associated with a file (for example, see elf32_getehdr(3ELF) and elf_getdata(3ELF)), the program should reestablish the pointers after calling elf_update().

Name elf_version – coordinate ELF library and application versions

Synopsis

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

unsigned elf_version(unsigned ver);
```

Description As elf(3ELF) explains, the program, the library, and an object file have independent notions of the latest ELF version. `elf_version()` lets a program query the ELF library's *internal version*. It further lets the program specify what memory types it uses by giving its own *working version*, `ver`, to the library. Every program that uses the ELF library must coordinate versions as described below.

The header <`libelf.h`> supplies the version to the program with the macro `EV_CURRENT`. If the library's internal version (the highest version known to the library) is lower than that known by the program itself, the library may lack semantic knowledge assumed by the program. Accordingly, `elf_version()` will not accept a working version unknown to the library.

Passing *ver* equal to `EV_NONE` causes `elf_version()` to return the library's internal version, without altering the working version. If `ver` is a version known to the library, `elf_version()` returns the previous (or initial) working version number. Otherwise, the working version remains unchanged and `elf_version()` returns `EV_NONE`.

Examples **EXAMPLE 1** A sample display of using the `elf_version()` function.

The following excerpt from an application program protects itself from using an older library:

```
if (elf_version(EV_CURRENT) == EV_NONE) {
    /* library out of date */
    /* recover from error */
}
```

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

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

See Also elf(3ELF), elf32_xlatetof(3ELF), elf_begin(3ELF), libelf(3LIB), attributes(5)

Notes The working version should be the same for all operations on a particular ELF descriptor. Changing the version between operations on a descriptor will probably not give the expected results.

Name erf, erff, erfl – error function

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double erf(double x);

float erff(float x);

long double erfl(long double x);
```

Description These functions compute the error function of their argument *x*, defined as:

$$\frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt$$

Return Values Upon successful completion, these functions return the value of the error function.

If *x* is NaN, a NaN is returned.

If *x* is ±0, ±0 is returned.

If *x* is ±Inf, ±1 is returned.

If *x* is subnormal, 2/sqrt(π) * 2 is returned.

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

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

See Also `erfc(3M)`, `feclearexcept(3M)`, `fetestexcept(3M)`, `isnan(3M)`, `math.h(3HEAD)`, `attributes(5)`, `standards(5)`

Name erfc, erfcf, erfcl – complementary error function

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double erfc(double x);

float erfcf(float x);

long double erfcl(long double x);
```

Description These function compute the complementary error function 1.0 – `erf(`*x*`)`.

Return Values Upon successful completion, these functions return the value of the complementary error function.

If *x* is NaN, a NaN is returned.

If *x* is ±0, +1 is returned.

If *x* is –Inf, +2 is returned.

If *x* is +Inf, 0 is returned.

Errors No errors are defined.

Usage The `erfc()` function is provided because of the extreme loss of relative accuracy if `erf(`*x*`)` is called for large *x* and the result subtracted from 1.0.

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

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

See Also `erf(3M)`, `isnan(3M)`, `math.h(3HEAD)`, `attributes(5)`, `standards(5)`

Name Exacct – exacct system calls and error handling

Synopsis

```
use Sun::Solaris::Exacct qw(:EXACCT_ALL);
my $ea_rec = getacct(P_PID, $$);
```

Description This module provides access to the ea_error(3EXACCT) function and for all the extended accounting system calls. Constants from the various libexacct(3LIB) header files are also provided.

Constants The P_PID, P_TASKID, P_PROJID and all the EW_*, EP_*, EXR_* macros are provided as Perl constants.

Functions

`getacct($idtype, $id)`
The $idtype parameter must be either P_TASKID or P_PID and $id must be a corresponding task or process ID. This function returns an object of type Sun::Solaris::Exacct::Object, representing the unpacked accounting buffer returned by the underlying getacct(2) system call. In the event of error, undef is returned.

`putacct($idtype, $id, $record)`
The $idtype parameter must be either P_TASKID or P_PID and $id must be a corresponding task or process ID. If $record is of type Sun::Solaris::Exacct::Object, it is converted to the corresponding packed libexacct object and passed to the putacct(2) system call. If $record is not of type Sun::Solaris::Exacct::Object it is converted to a string using the normal Perl conversion rules and stored as a raw buffer. For predictable and endian-independent results, any raw buffers should be constructed using the Perl pack() function. This function returns true on success and false on failure.

`wracct($idtype, $id, $flags)`
The $idtype parameter must be either P_TASKID or P_PID and $id must be a corresponding task or process ID. The $flags parameter must be either EW_INTERVAL or EW_PARTIAL. The parameters are passed directly to the underlying wracct(2) system call. This function returns true on success and false on failure.

`ea_error()`
This function provides access to the ea_error(3EXACCT) function. It returns a double-typed scalar that becomes one of the EXR_* constants. In a string context it becomes a descriptive error message. This is the exacct equivalent to the $!(errno) Perl variable.

`ea_error_str()`
This function returns a double-typed scalar that in a numeric context will be one of the EXR_* constants as returned by ea_error. In a string context it describes the value returned by ea_error. If ea_error returns EXR_SYSCALL_FAIL, the string value returned is the value returned by strerror(3C). This function is provided as a convenience so that repeated blocks of code like the following can be avoided:

```
if (ea_error() == EXR_SYSCALL_FAIL) {
        print("error: $!\n");
} else {
```

```
        print("error: ", ea_error(), "\n");
    }
```

`ea_register_catalog($cat_pfx, $catalog_id, $export, @idlist)`
This convenience function is a wrapper around the `Sun::Solaris::Exacct::Catalog->register()` method.

`ea_new_catalog($integer)`
`ea_new_catalog($cat_obj)`
`ea_new_catalog($type, $catalog, $id)`
These convenience functions are wrappers around the `Sun::Solaris::Exacct::Catalog->new()` method. See `Exacct::Catalog(3PERL)`.

`ea_new_file($name, $oflags, creator => $creator, aflags => $aflags, mode => $mode)`
This convenience function is a wrapper around the `Sun::Solaris::Exacct::File->new()` method. See `Exacct::File(3PERL)`.

`ea_new_item($catalog, $value)`
This convenience function is a wrapper around the `Sun::Solaris::Exacct::Object::Item->new()` method. See `Exacct::Object::Item(3PERL)`.

`ea_new_group($catalog, @objects)`
This convenience function is a wrapper around the `Sun::Solaris::Exacct::Object::Group->new()` method. See `Exacct::Object::Group(3PERL)`.

`ea_dump_object($object, $filehandle)`
This convenience function is a wrapper around the `Sun::Solaris::Exacct::Object->dump()` method. See `Exacct::Object(3PERL)`.

Class methods None.

Object methods None.

Exports By default nothing is exported from this module. The following tags can be used to selectively import constants and functions defined in this module:

`:SYSCALLS`	`getacct()`, `putacct()`, and `wracct()`
`:LIBCALLS`	`ea_error()` and `ea_error_str()`
`:CONSTANTS`	`P_PID`, `P_TASKID`, `P_PROJID`, `EW_*`, `EP_*`, and `EXR_*`
`:SHORTHAND`	`ea_register_catalog()`, `ea_new_catalog()`, `ea_new_file()`, `ea_new_item()`, and `ea_new_group()`
`:ALL`	`:SYSCALLS`, `:LIBCALLS`, `:CONSTANTS`, and `:SHORTHAND`

:EXACCT_CONSTANTS	:CONSTANTS, plus the :CONSTANTS tags for Sun::Solaris::Catalog, Sun::Solaris::File, and Sun::Solaris::Object
:EXACCT_ALL	:ALL, plus the :ALL tags for Sun::Solaris::Catalog, Sun::Solaris::File, and Sun::Solaris::Object

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWpl5u
Interface Stability	Evolving

See Also getacct(2), putacct(2), wracct(2), ea_error(3EXACCT), Exacct::Catalog(3PERL), Exacct::File(3PERL), Exacct::Object(3PERL), Exacct::Object::Group(3PERL), Exacct::Object::Item(3PERL), libexacct(3LIB), attributes(5)

Notes The modules described in the section 3PERL manual pages make extensive use of the Perl "double-typed scalar" facility. This facility allows a scalar value to behave either as an integer or as a string, depending upon context. It is the same behavior as exhibited by the $! Perl variable (errno). It is useful because it avoids the need to map from an integer value to the corresponding string to display a value. Some examples are provided below:

```
# Assume $obj is a Sun::Solaris::Item
my $type = $obj->type();

# Print "2 EO_ITEM"
printf("%d %s\n", $type, $type);

# Behave as an integer, $i == 2
my $i = 0 + $type;

# Behave as a string, $s = "abc EO_ITEM xyx"
my $s = "abc $type xyz";
```

Wherever a function or method is documented as returning a double-typed scalar, the returned value exhibits this type of behavior.

Name Exacct::Catalog – exacct catalog tag manipulation

Synopsis

```
use Sun::Solaris::Exacct::Catalog qw(:ALL);
my $ea_cat = Sun::Solaris::Exacct::Catalog->new(
    &EXT_UINT64 | &EXC_DEFAULT | &EXD_PROC_PID);
```

Description This class provides a wrapper around the 32-bit integer used as a catalog tag. The catalog tag is represented as a Perl object blessed into the `Sun::Solaris::Exacct::Catalog` class so that methods can be used to manipulate fields in a catalog tag.

Constants All the `EXT_*`, `EXC_*`, and `EXD_*` macros are provided as constants. Constants passed to the methods below can either be the integer value such as `EXT_UINT8` or the string representation such as "`EXT_UINT8`".

Functions None.

Class methods `register($cat_pfx, $catalog_id, $export, @idlist)`

This method is used to register application-defined `libexacct(3LIB)` catalogs with the exacct Perl library. See </usr/include/sys/exacct_catalog.h> for details of the catalog tag format. This method allows symbolic names and strings to be used for manipulating application-defined catalogs. The first two parameters define the catalog prefix and associated numeric catalog ID. If the `$export` parameter is true, the constants are exported into the caller's package. The final parameter is a list of (`id`, `name`) pairs that identify the required constants. The constants created by this method are formed by appending `$cat_pfx` and "_" to each name in the list, replacing any spaces with underscore characters and converting the resulting string to uppercase characters. The `$catalog_name` value is also created as a constant by prefixing it with `EXC_` and converting it to uppercase characters. Its value becomes that of `$catalog_id` shifted left by 24 bits. For example, the following call:

```
Sun::Solaris::Exacct::Catalog->ea_register("MYCAT", 0x01, 1,
   FIRST => 0x00000001, SECOND => 0x00000010);
```

results in the definition of the following constants:

```
EXC_MYCAT    0x01 << 24
MYCAT_FIRST  0x00000001
MYCAT_SECOND 0x00000010
```

Only the catalog ID value of 0x01 is available for application use (`EXC_LOCAL`). All other values are reserved. While it is possible to use values other than 0x01, they might conflict with future extensions to the `libexacct` file format.

If any errors are detected during this method, a string is returned containing the appropriate error message. If the call is sucessful, `undef` is returned.

```
new($integer)
new($cat_obj)
new($type, $catalog, $id)
```

This method creates and returns a new Catalog object, which is a wrapper around a 32-bit integer catalog tag. Three possible argument lists can be given. The first variant is to pass an integer formed by bitwise-inclusive OR of the appropriate `EX[TCD]_*` constants. The second variant is to pass an existing Catalog object that will be copied. The final variant is to pass in the type, catalog and ID fields as separate values. Each of these values can be either an appropriate integer constant or the string representation of the constant.

Object methods

`value()` This method allows the value of the catalog tag to be queried. In a scalar context it returns the 32-bit integer representing the tag. In a list context it returns a (`type`, `catalog`, `id`) triplet, where each member of the triplet is a dual-typed scalar.

`type()` This method returns the type field of the catalog tag as a dual-typed scalar.

`catalog()` This method returns the catalog field of the catalog tag as a dual-typed scalar.

`id()` This method returns the id field of the catalog tag as a dual-typed scalar.

`type_str()`
`catalog_str()`
`id_str()` These methods return string representations of the appropriate value. These methods can be used for textual output of the various catalog fields. The string representations of the constants are formed by removing the `EXT_`, `EXC_`, or `EXD_` prefix, replacing any underscore characters with spaces, and converting the remaining string to lowercase characters.

Exports

By default nothing is exported from this module. The following tags can be used to selectively import constants and functions defined in this module:

`:CONSTANTS` `EXT_*`, `EXC_*`, and `EXD_*`

`:ALL` `:CONSTANTS`

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWpl5u
Interface Stability	Evolving

See Also

`Exacct(3PERL)`, `Exacct::File(3PERL)`, `Exacct::Object(3PERL)`, `Exacct::Object::Group(3PERL)`, `Exacct::Object::Item(3PERL)`, `libexacct(3LIB)`, `attributes(5)`

Name Exacct::File – exacct file manipulation

Synopsis

```
use Sun::Solaris::Exacct::File qw(:ALL);
my $ea_file = Sun::Solaris::Exacct::File->new($myfile, &O_RDONLY);
my $ea_obj = $ea_file->get();
```

Description This module provides access to the libexacct(3LIB) functions that manipulate accounting files. The interface is object-oriented and allows the creation and reading of libexacct files. The C library calls wrapped by this module are ea_open(3EXACCT), ea_close(3EXACCT), ea_next_object(3EXACCT), ea_previous_object(3EXACCT), ea_write_object(3EXACCT), ea_get_object(3EXACCT), ea_get_creator(3EXACCT), and ea_get_hostname(3EXACCT). The file read and write methods all operate on Sun::Solaris::Exacct::Object objects and perform all the necessary memory management, packing, unpacking, and structure conversions that are required.

Constants EO_HEAD, EO_TAIL, EO_NO_VALID_HDR, EO_POSN_MSK, and EO_VALIDATE_MSK. Other constants needed by the new() method below are in the standard Perl Fcntl module.

Functions None.

Class methods `new($name, $oflags, creator => $creator,`

This method opens a libexacct file as specified by the mandatory parameters $name and $oflags, and returns a Sun::Solaris::Exacct::File object, or undef if an error occurs. The parameters $creator, $aflags, and $mode are optional and are passed as (name => value) pairs. The only valid values for $oflags are the combinations of O_RDONLY, O_WRONLY, O_RDWR, and O_CREAT described below.

The $creator parameter is a string describing the creator of the file. If it is required (for instance, when writing to a file) but absent, it is set to the string representation of the caller's UID. The $aflags parameter describes the required positioning in the file for O_RDONLY access: either EO_HEAD or EO_TAIL are allowed. If absent, EO_HEAD is assumed. The $mode parameter is the file creation mode and is ignored unless O_CREAT is specified in $oflags. If $mode is unspecified, the file creation mode is set to 0666 (octal). If an error occurs, it can be retrieved with the Sun::Solaris::Exacct::ea_error() function. See Exacct(3PERL).

$oflags	$aflags	Action
O_RDONLY	Absent or EO_HEAD	Open for reading at the start of the file.
O_RDONLY	EO_TAIL	Open for reading at the end of the file.
O_WRONLY	Ignored	File must exist, open for writing at the end of the file.

$oflags	$aflags	Action
O_WRONLY | O_CREAT	Ignored	Create file if it does not exist, otherwise truncate and open for writing.
O_RDWR	Ignored	File must exist, open for reading/writing, positioned at the end of the file.
O_RDWR | O_CREAT	Ignored	Create file if it does not exist, otherwise truncate and open for reading/writing.

Object methods

There is no explicit `close()` method for a `Sun::Solaris::Exacct::File`. The file is closed when the file handle object is undefined or reassigned.

`creator()`
: This method returns a string containing the creator of the file or `undef` if the file does not contain the information.

`hostname()`
: This method returns a string containing the hostname on which the file was created, or `undef` if the file does not contain the information.

`next()`
: This method reads the header information of the next record in the file. In a scalar context the value of the type field is returned as a dual-typed scalar that will be one of `EO_ITEM`, `EO_GROUP`, or `EO_NONE`. In a list context it returns a two-element list containing the values of the type and catalog fields. The type element is a dual-typed scalar. The catalog element is blessed into the `Sun::Solaris::Exacct::Catalog` class. If an error occurs, `undef` or (`undef`, `undef`) is returned depending upon context. The status can be accessed with the `Sun::Solaris::Exacct::ea_error()` function.See Exacct(3PERL).

`previous()`
: This method reads the header information of the previous record in the file. In a scalar context it returns the type field. In a list context it returns the two-element list containing the values of the type and catalog fields, in the same manner as the `next()` method. Error are also returned in the same manner as the `next()` method.

`get()`
: This method reads in the `libexacct` record at the current position in the file and returns a `Sun::Solaris::Exacct::Object` containing the unpacked data from the file. This object can then be further manipulated using its methods. In case of error undef is returned and the error status is made available with the `Sun::Solaris::Exacct::ea_error()` function. After this operation, the position in the file is set to the start of the next record in the file.

`write(@ea_obj)`
: This method converts the passed list of Sun::Solaris::Exacct::Objects into `libexacct` file format and appends them to the `libexacct` file, which must be open for writing. This

method returns true if successful and false otherwise. On failure the error can be examined with the `Sun::Solaris::Exacct::ea_error()` function.

Exports By default nothing is exported from this module. The following tags can be used to selectively import constants defined in this module:

`:CONSTANTS`
`EO_HEAD`, `EO_TAIL`, `EO_NO_VALID_HDR`, `EO_POSN_MSK`, and `EO_VALIDATE_MSK`

`:ALL`
`:CONSTANTS`, Fcntl(`:DEFAULT`).

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWpl5u
Interface Stability	Evolving

See Also

`ea_close(3EXACCT)`, `ea_get_creator(3EXACCT)`, `ea_get_hostname(3EXACCT)`, `ea_get_object(3EXACCT)`, `ea_next_object(3EXACCT)`, `ea_open(3EXACCT)`, `ea_previous_object(3EXACCT)`, `ea_write_object(3EXACCT)`, `Exacct(3PERL)`, `Exacct::Catalog(3PERL)`, `Exacct::Object(3PERL)`, `Exacct::Object::Group(3PERL)`, `Exacct::Object::Item(3PERL)`, `libexacct(3LIB)`, `attributes(5)`

Name Exacct::Object – exacct object manipulation

Synopsis

```
use Sun::Solaris::Exacct::Object qw(:ALL);
print($ea_obj->value(), "\n");
```

Description This module is used as a parent of the two possible types of Perl exacct objects: Items and Groups. An Item is either a single data value such as the number of seconds of user CPU time consumed by a process, an embedded Perl exacct object, or a block of raw data. A Group is an ordered collection of Perl exacct Items such as all of the resource usage values for a particular process or task. If Groups need to be nested within each other, the inner Groups can be stored as embedded Perl exacct objects inside the enclosing Group.

This module contains methods that are common to both Perl exacct Items and Groups. The attributes of `Sun::Solaris::Exacct::Object` and all classes derived from it are read-only after initial creation with `new()`. This behavior prevents the inadvertent modification of the attributes that could produce inconsistent catalog tags and data values. The only exception is the array used to store the Items inside a Group object, which can be modified using the normal Perl array operators. See the `value()` method below.

Constants `EO_ERROR`, `EO_NONE`, `EO_ITEM`, and `EO_GROUP`.

Functions None.

Class methods

`dump($object, $filehandle)` This method dumps formatted text representation of a Perl exacct object to the supplied file handle. If no file handle is specified, the text representation is dumped to `STDOUT`. See EXAMPLES below for sample output.

Object methods

`type()` This method returns the type field of the Perl exacct object. The value of the type field is returned as a dual-typed scalar and is either `EO_ITEM`, `EO_GROUP`, or `EO_NONE`.

`catalog()` This method returns the catalog field of the Perl exacct object. The value is returned as a `Sun::Solaris::Exacct::Catalog` object.

`match_catalog($catalog)` This method matches the passed catalog tag against the object. True is returned of a match occurs. Otherwise false is returned. This method has the same behavior as the underlying `ea_match_object_catalog(3EXACCT)` function.

`value()` This method returns the value of the Perl exacct object. In the case of an Item, this object will normally be a Perl scalar, either a number or string. For raw Items, the buffer contained inside the object is returned as a Perl string that can be manipulated with the Perl `unpack()` function. If the Item contains either a nested Item or a nested Group, the enclosed

Item is returned as a reference to an object of the appropriate subtype of the `Sun::Solaris::Exacct::Object` class.

For Group objects, if `value()` is called in a scalar context, the return value is a reference to the underlying array used to store the component Items of the Group. Since this array can be manipulated with the normal Perl array indexing syntax and array operators, the objects inside the Group can be manipulated. All objects in the array must be derived from the `Sun::Solaris::Exacct::Object` class. Any attempt to insert something else into the array will generate a fatal runtime error that can be caught with an `eval { }` block.

If `value()` is called in a list context for a Group object, it returns a list of all the objects in the Group. Unlike the array reference returned in a scalar context, this list cannot be manipulated to add or delete Items from a Group. This mechanism is considerably faster than the array mechanism described above and is the preferred mechanism if a Group is being examined in a read-only manner.

Exports By default nothing is exported from this module. The following tags can be used to selectively import constants and functions defined in this module:

`:CONSTANTS` `EO_ERROR`, `EO_NONE`, `EO_ITEM`, and `EO_GROUP`

`:ALL` `:CONSTANTS`

Examples

EXAMPLE 1 Output of the `dump()` method for a Perl exacct Group object.

The following is an example of output of the `dump()` method for a Perl exacct Group object.

```
GROUP
  Catalog = EXT_GROUP|EXC_DEFAULT|EXD_GROUP_PROC_PARTIAL
  ITEM
    Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_PID
    Value = 3
  ITEM
    Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_UID
    Value = 0
  ITEM
    Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_GID
    Value = 0
  ITEM
    Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_PROJID
    Value = 0
  ITEM
    Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_TASKID
```

EXAMPLE 1 Output of the `dump()` method for a Perl exacct Group object. *(Continued)*

```
    Value = 0
  ITEM
    Catalog = EXT_STRING|EXC_DEFAULT|EXD_PROC_COMMAND
    Value = fsflush
ENDGROUP
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWpl5u
Interface Stability	Evolving

See Also

`ea_match_object_catalog(3EXACCT)`, `Exacct(3PERL)`, `Exacct::Catalog(3PERL)`, `Exacct::File(3PERL)`, `Exacct::Object::Group(3PERL)`, `Exacct::Object::Item(3PERL)`, `libexacct(3LIB)`, `attributes(5)`

Name Exacct::Object::Group – exacct group manipulation

Synopsis

```
use Sun::Solaris::Exacct::Object;
my $ea_grp = Sun::Solaris::Exacct::Object::Group->new(
   & EXT_GROUP | &EXC_DEFAULT | &EXD_GROUP_PROC);
```

Description This module is used for manipulating libexacct(3LIB) Group objects. A `libexacct` Group object is represented as an opaque reference blessed into the `Sun::Solaris::Exacct::Object::Group` class, which is a subclass of the `Sun::Solaris::Exacct::Object` class. The Items within a Group are stored inside a Perl array. A reference to the array can be accessed with the inherited `value()` method. The individual Items within a Group can be manipulated with the normal Perl array syntax and operators. All data elements of the array must be derived from the `Sun::Solaris::Exacct::Object` class. Group objects can also be nested inside each other simply by adding an existing Group as a data Item.

Constants None.

Functions None.

Class methods Class methods include those inherited from the `Sun::Solaris::Exacct::Object` base class, plus the following:

`new($catalog, @objects)` This method creates and returns a new `Sun::Solaris::Exacct::Object::Group`. The catalog tag can be either an integer or a `Sun::Solaris::Exacct::Catalog`. The catalog tag should be a valid catalog tag for a Perl exacct Group object. The `@objects` parameter is a list of `Sun::Solaris::Exacct::Object` to be stored inside the Group. A copy of all the passed Items is taken and any Group objects are recursively copied. The contents of the returned Group object can be accessed with the array returned by the value method.

Object methods

`as_hash()` This method returns the contents of the group as a hash reference. It uses the string value of each item's catalog ID as the hash entry key and the scalar value returned by `value()` as the hash entry value. This form should be used if there are no duplicate catalog tags in the group.

This method and its companion `as_hashlist()` are the fastest ways to access the contents of a Group.

`as_hashlist()` This method returns the contents of the group as a hash reference. It uses the string value of each item's catalog id as the hash entry key and an array of the scalar values returned by `value()` as the hash entry value for all the items that share a common key. This form should be used if there might be duplicate catalog tags in the group.

This method and its companion `as_hash()` are the fastest ways to access the contents of a Group.

Exports None.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWpl5u
Interface Stability	Evolving

See Also `Exacct(3PERL)`, `Exacct::Catalog(3PERL)`, `Exacct::File(3PERL)`, `Exacct::Object(3PERL)`, `Exacct::Object::Item(3PERL)`, `libexacct(3LIB)`, `attributes(5)`

Name Exacct::Object::Item – exacct item manipulation

Synopsis

```
use Sun::Solaris::Exacct::Object;
my $ea_item = Sun::Solaris::Exacct::Object::Item->new(
    &EXT_UINT64 | &EXC_DEFAULT | &EXD_PROC_PID, $$);
```

Description This module is used for manipulating `libexacct(3LIB)` data Items. A `libexacct` Item is represented as an opaque reference blessed into the `Sun::Solaris::Exacct::Object::Item` class, which is a subclass of the `Sun::Solaris::Exacct::Object` class. The underlying `libexacct` data types are mapped onto Perl types as follows:

libexacct type	Perl internal type
`EXT_UINT8`	IV (integer)
`EXT_UINT16`	IV (integer)
`EXT_UINT32`	IV (integer)
`EXT_UINT64`	IV (integer)
`EXT_DOUBLE`	NV (double)
`EXT_STRING`	PV (string)
`EXT_RAW`	PV (string)
`EXT_EXACCT_OBJECT`	`Sun::Solaris::Exacct::Object subclass`

Constants None.

Functions None.

Class methods Class methods include those inherited from the `Sun::Solaris::Exacct::Object` base class, plus the following:

`new($catalog, $value)` This method creates and returns a new `Sun::Solaris::Exacct::Object::Item`. The catalog tag can be either an integer or a `Sun::Solaris::Exacct::Catalog`. This catalog tag controls the conversion of the Perl value to the corresponding Perl exacct data type as described in the table above. If the catalog tag has a type field of `EXT_EXACCT_OBJECT`, the value must be a reference to either an Item or a Group object and the passed object is recursively copied and stored inside the new Item. Because the returned Item is constant, it is impossible, for example, to create an Item representing CPU seconds and subsequently modify its value or change its catalog value. This behavior is intended to prevent mismatches between the catalog tag and the data value.

Object methods Object methods are those inherited from the `Sun::Solaris::Exacct::Object`.

Exports None.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWpl5u
Interface Stability	Evolving

See Also `Exacct(3PERL)`, `Exacct::Catalog(3PERL)`, `Exacct::File(3PERL)`, `Exacct::Object(3PERL)`, `Exacct::Object::Group(3PERL)`, `libexacct(3LIB)`, `attributes(5)`

Name exp2, exp2f, exp2l – exponential base 2 functions

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double exp2(double x);

float exp2f(float x);

long double exp2l(long double x);
```

Description These functions compute the base-2 exponential of *x*.

Return Values Upon successful completion, these functions return 2^x.

If the correct value would cause overflow, a range error occurs and `exp2()`, `exp2f()`, and `exp2l()` return the value of the macro `HUGE_VAL`, `HUGE_VALF`, and `HUGE_VALL`, respectively.

If *x* is NaN, a NaN is returned.

If *x* is ±0, 1 is returned.

If *x* is −Inf, +0 is returned.

If *x* is +Inf, *x* is returned.

Errors These functions will fail if:

Range Error — The result overflows.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, the overflow floating-point exception will be raised.

Usage An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. An application should either examine the return value or check the floating point exception flags to detect exceptions.

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

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

See Also `exp(3M)`, `feclearexcept(3M)`, `fetestexcept(3M)`, `isnan(3M)`, `log(3M)`, `math.h(3HEAD)`, `attributes(5)`, `standards(5)`

Name exp, expf, expl – exponential function

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double exp(double x);

float expf(float x);

long double expl(long double x);
```

Description These functions compute the base-*e* exponential of *x*.

Return Values Upon successful completion, these functions return the exponential value of *x*.

If the correct value would cause overflow, a range error occurs and `exp()`, `expf()`, and `expl()` return `HUGE_VAL`, `HUGE_VALF`, and `HUGE_VALL`, respectively.

If *x* is NaN, a NaN is returned.

If *x* is ±0, 1 is returned.

If *x* is +Inf, *x* is returned.

For exceptional cases, `matherr(3M)` tabulates the values to be returned by `exp()` as specified by SVID3 and XPG3. See `standards(5)`.

Errors These functions will fail if:

Range Error — The result overflows.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, the overflow floating-point exception is raised.

The `exp()` function sets `errno` to `ERANGE` if the result overflows.

Usage An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. An application should either examine the return value or check the floating point exception flags to detect exceptions.

An application can also set `errno` to 0 before calling `exp()`. On return, if `errno` is non-zero, an error has occurred. The `expf()` and `expl()` functions do not set `errno`.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also feclearexcept(3M), fetestexcept(3M), isnan(3M), log(3M), math.h(3HEAD), matherr(3M), mp(3MP), attributes(5), standards(5)

Name expm1, expm1f, expm1l – compute exponential function

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double expm1(double x);

float expm1f(float x);

long double expm1l(long double x);
```

Description These functions compute $e^x-1.0$.

Return Values Upon successful completion, these functions return $e^x-1.0$.

If x is NaN, a NaN is returned.

If x is ±0, ±0 is returned.

If x is −Inf, −1 is returned.

If x is +Inf, x is returned.

Errors These functions will fail if:

Range Error — The result overflows.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, the overflow floating-point exception is raised.

Usage The value of `expm1(`*x*`)` can be more accurate than `exp(`*x*`)`−1.0 for small values of *x*.

The `expm1()` and `log1p(3M)` functions are useful for financial calculations of $((1+x)^n-1)/x$, namely:

```
expm1(n * log1p(x)) / x
```

when *x* is very small (for example, when performing calculations with a small daily interest rate). These functions also simplify writing accurate inverse hyperbolic functions.

An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. An application should either examine the return value or check the floating point exception flags to detect exceptions.

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

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

See Also exp(3M), feclearexcept(3M), fetestexcept(3M), ilogb(3M), log1p(3M), math.h(3HEAD), attributes(5), standards(5)

Name fabs, fabsf, fabsl – absolute value function

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double fabs(double x);

float fabsf(float x);

long double fabsl(long double x);
```

Description These functions compute the absolute value of x, $|x|$.

Return Values Upon successful completion, these functions return the absolute value of x.

If x is NaN, a NaN is returned.

If x is ±0, +0 is returned.

If x is ±Inf, +Inf is returned.

Errors No errors are defined.

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

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

See Also isnan(3M), math.h(3HEAD), attributes(5), standards(5)

Name fdim, fdimf, fdiml – compute positive difference between two floating-point numbers

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double fdim(double x, double y);

float fdimf(float x, float y);

long double fdiml(long double x, long double y);
```

Description These functions determine the positive difference between their arguments. If *x* is greater than *y*, *x*–*y* is returned. If *x* is less than or equal to *y*, +0 is returned.

Return Values Upon successful completion, these functions return the positive difference value.

If *x*–*y* is positive and overflows, a range error occurs and `fdim()`, `fdimf()`, and `fdiml()` returns the value of the macro `HUGE_VAL`, `HUGE_VALF`, and `HUGE_VALL`, respectively.

If *x* or *y* is NaN, a NaN is returned.

Errors These functions will fail if:

Range Error — The result overflows.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, the overflow floating-point exception will be raised.

Usage An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. An application should either examine the return value or check the floating point exception flags to detect exceptions.

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

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

See Also `feclearexcept(3M)`, `fetestexcept(3M)`, `fmax(3M)`, `fmin(3M)`, `math.h(3HEAD)`, `attributes(5)`, `standards(5)`

Name feclearexcept – clear floating-point exception

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <fenv.h>

int feclearexcept(int excepts);
```

Description The `feclearexcept()` function attempts to clear the supported floating-point exceptions represented by *excepts*.

Return Values If *excepts* is 0 or if all the specified exceptions were successfully cleared, `feclearexcept()` returns 0. Otherwise, it returns a non-zero value.

Errors No errors are defined.

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

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

See Also `fenv.h(3HEAD)`, `fegetexceptflag(3M)`, `feraiseexcept(3M)`, `fesetexceptflag(3M)`, `fetestexcept(3M)`, `attributes(5)`, `standards(5)`

Name fegetenv, fesetenv – get and set current floating-point environment

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <fenv.h>

int fegetenv(fenv_t *envp);

int fesetenv(const fenv_t *envp);
```

Description The `fegetenv()` function attempts to store the current floating-point environment in the object pointed to by *envp*.

The `fesetenv()` function attempts to establish the floating-point environment represented by the object pointed to by *envp*. The *envp* argument points to an object set by a call to `fegetenv()` or `feholdexcept(3M)`, or equals a floating-point environment macro. The `fesetenv()` function does not raise floating-point exceptions, but only installs the state of the floating-point status flags represented through its argument.

Return Values If the representation was successfully stored, fegetenv returns 0. Otherwise, it returns a non-zero value.

If the environment was successfully established, fesetenv returns 0. Otherwise, it returns a non-zero value.

Errors No errors are defined.

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

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

See Also `feholdexcept(3M)`, `fenv.h(3HEAD)`, `feupdateenv(3M)`, `attributes(5)`, `standards(5)`

Notes In a multithreaded program, the `fegetenv()` and `fegetenv()` functions affect the floating point environment only for the calling thread.

These functions automatically install and deinstall `SIGFPE` handlers and set and clear the trap enable mode bits in the floating point status register as needed. If a program uses these functions and attempts to install a `SIGFPE` handler or control the trap enable mode bits independently, the resulting behavior is not defined.

As described in `fex_set_handling(3M)`, when a handling function installed in `FEX_CUSTOM` mode is invoked, all exception traps are disabled (and will not be reenabled while `SIGFPE` is blocked). Thus, attempting to change the environment from within a handler by calling `fesetenv` or `feupdateenv(3M)` might not produce the expected results.

Name fegetexceptflag, fesetexceptflag – get and set floating-point status flags

Synopsis

```
cc [ flag... ] file... -lm [ library... ]
#include <fenv.h>

int fegetexceptflag(fexcept_t *flagp, int excepts);

int fesetexceptflag(const fexcept_t *flagp, int excepts);
```

Description The `fegetexceptflag()` function attempts to store an implementation-defined representation of the states of the floating-point status flags indicated by the *excepts* argument in the object pointed to by the *flagp* argument.

The `fesetexceptflag()` function attempts to set the floating-point status flags indicated by the *excepts* argument to the states stored in the object pointed to by *flagp*. The value pointed to by *flagp* will have been set by a previous call to `fegetexceptflag()` whose second argument represented at least those floating-point exceptions represented by the *excepts* argument. This function does not raise floating-point exceptions but only sets the state of the flags.

Return Values If the representation was successfully stored, `fegetexceptflag()` returns 0. Otherwise, it returns a non-zero value.

If the excepts argument is 0 or if all the specified exceptions were successfully set, `fesetexceptflag()` returns 0. Otherwise, it returns a non-zero value.

Errors No errors are defined.

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

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

See Also `fenv.h`(3HEAD), `feclearexcept`(3M), `feraiseexcept`(3M), `fesetexceptflag`(3M), `attributes`(5), `standards`(5)

Name fegetround, fesetround – get and set current rounding direction

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <fenv.h>

int fegetround(void);

int fesetround(int round);
```

Description The fegetround function gets the current rounding direction.

The fesetround function establishes the rounding direction represented by its argument round. If the argument is not equal to the value of a rounding direction macro, the rounding direction is not changed.

Return Values The fegetround function returns the value of the rounding direction macro representing the current rounding direction, or a negative value if there is no such rounding direction macro or the current rounding direction is not determinable.

The fesetround function returns a 0 value if and only if the requested rounding direction was established.

Errors No errors are defined.

Examples The following example saves, sets, and restores the rounding direction, reporting an error and aborting if setting the rounding direction fails:

EXAMPLE 1 Save, set, and restore the rounding direction.

```
#include <fenv.h>
#include <assert.h>
void f(int round_dir)
{
      #pragma STDC FENV_ACCESS ON
      int save_round;
      int setround_ok;
      save_round = fegetround();
      setround_ok = fesetround(round_dir);
      assert(setround_ok == 0);
      /* ... */
      fesetround(save_round);
      /* ... */
}
```

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also fenv.h(3HEAD), attributes(5), standards(5)

Name feholdexcept – save current floating-point environment

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <fenv.h>

int feholdexcept(fenv_t *envp);
```

Description The `feholdexcept()` function saves the current floating-point environment in the object pointed to by *envp*, clears the floating-point status flags, and then installs a non-stop (continue on floating-point exceptions) mode, if available, for all floating-point exceptions.

Return Values The `feholdexcept()` function returns 0 if and only if non-stop floating-point exception handling was successfully installed.

Errors No errors are defined.

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

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

See Also `fegetenv(3M)`, `fenv.h(3HEAD)`, `feupdateenv(3M)`, `attributes(5)`, `standards(5)`

Notes In a multithreaded program, the `feholdexcept()` function affects the floating point environment only for the calling thread.

The `feholdexcept()` function automatically installs and deinstalls `SIGFPE` handlers and sets and clears the trap enable mode bits in the floating point status register as needed. If a program uses these functions and attempts to install a `SIGFPE` handler or control the trap enable mode bits independently, the resulting behavior is not defined.

Name feraiseexcept – raise floating-point exception

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <fenv.h>

int feraiseexcept(int excepts);
```

Description The `feraiseexcept()` function attempts to raise the supported floating-point exceptions represented by the *excepts* argument. The order in which these floating-point exceptions are raised is unspecified.

Return Values If *excepts* is 0 or if all the specified exceptions were successfully raised, `feraiseexcept()` returns 0. Otherwise, it returns a non-zero value.

Errors No errors are defined.

Usage The effect is intended to be similar to that of floating-point exceptions raised by arithmetic operations. Hence, enabled traps for floating-point exceptions raised by this function are taken.

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

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

See Also `feclearexcept(3M)`, `fegetexceptflag(3M)`, `fenv.h(3HEAD)`, `fetestexcept(3M)`, `attributes(5)`, `standards(5)`

Name fesetprec, fegetprec – control floating point rounding precision modes

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <fenv.h>

int fesetprec(int prec);

int fegetprec(void);
```

Description The IEEE 754 standard defines rounding precision modes for systems that always deliver intermediate results to destinations in extended double precision format. These modes allow such systems to deliver correctly rounded single and double precision results (in the absence of underflow and overflow) with only one rounding.

The `fesetprec()` function sets the current rounding precision to the precision specified by *prec*, which must be one of the following values defined in `<fenv.h>`:

`FE_FLTPREC` round to single precision

`FE_DBLPREC` round to double precision

`FE_LDBLPREC` round to extended double precision

The default rounding precision when a program starts is `FE_LDBLPREC`.

The `fegetprec()` function returns the current rounding precision.

Return Values The `fesetprec()` function returns a non-zero value if the requested rounding precision is established and 0 otherwise.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	Intel (see below)
Availability	SUNWlibms
Interface Stability	Stable
MT-Level	MT-Safe

These functions are not available on SPARC systems because SPARC processors deliver intermediate results to destinations in single or double format as determined by each floating point instruction.

See Also `fegetenv(3M)`, `fesetround(3M)`, `attributes(5)`

Numerical Computation Guide

Name fetestexcept – test floating-point exception flags

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <fenv.h>

int fetestexcept(int excepts);
```

Description The `fetestexcept()` function determines which of a specified subset of the floating-point exception flags are currently set. The *excepts* argument specifies the floating-point status flags to be queried.

Return Values The `fetestexcept()` function returns the value of the bitwise-inclusive OR of the floating-point exception macros corresponding to the currently set floating-point exceptions included in *excepts*.

Errors No errors are defined.

Examples **EXAMPLE 1** Example using `fetestexcept()`

The following example calls function *f*( ) if an invalid exception is set, and then function *g*( ) if an overflow exception is set:

```
#include <fenv.h>
/* ... */
{
#     pragma STDC FENV_ACCESS ON
      int set_excepts;
      feclearexcept(FE_INVALID | FE_OVERFLOW);
      // maybe raise exceptions
      set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
      if (set_excepts & FE_INVALID) f();
      if (set_excepts & FE_OVERFLOW) g();
      /* ... */
}
```

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

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

See Also `feclearexcept(3M)`, `fegetexceptflag(3M)`, `fenv.h(3HEAD)`, `attributes(5)`, `standards(5)`

Name feupdateenv – update floating-point environment

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <fenv.h>

int feupdateenv(const fenv_t *envp);
```

Description The `feupdateenv()` function attempts to save the currently raised floating-point exceptions in its automatic storage, attempts to install the floating-point environment represented by the object pointed to by *envp*, and then attempts to raise the saved floating-point exceptions. The *envp* argument points to an object set by a call to `fegetenv(3M)` or `feholdexcept(3M)`, or equals a floating-point environment macro.

Return Values The `feupdateenv()` function returns 0 if and only if all the required actions were successfully carried out.

Errors No errors are defined.

Examples The following example demonstrates sample code to hide spurious underflow floating-point exceptions:

EXAMPLE 1 Hide spurious underflow floating-point exceptions.

```
#include <fenv.h>
double f(double x)
{
#   pragma STDC FENV_ACCESS ON
    double result;
    fenv_t save_env;
    feholdexcept(&save_env);
    // compute result
    if (/* test spurious underflow */)
    feclearexcept(FE_UNDERFLOW);
    feupdateenv(&save_env);
    return result;
}
```

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

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

See Also `fegetenv(3M)`, `feholdexcept(3M)`, `fenv.h(3HEAD)`, `attributes(5)`, `standards(5)`

Notes In a multithreaded program, the `feupdateenv()` function affects the floating point environment only for the calling thread.

When the `FEX_CUSTOM` handling mode is in effect for an exception, raising that exception using `feupdateenv()` causes the handling function to be invoked. The handling function can then modify the exception flags to be set as described in `fex_set_handling(3M)`. Any result value the handler supplies will be ignored.

The `feupdateenv()` function automatically installs and deinstalls `SIGFPE` handlers and sets and clears the trap enable mode bits in the floating point status register as needed. If a program uses these functions and attempts to install a `SIGFPE` handler or control the trap enable mode bits independently, the resulting behavior is not defined.

As described in `fex_set_handling(3M)`, when a handling function installed in `FEX_CUSTOM` mode is invoked, all exception traps are disabled (and will not be reenabled while `SIGFPE` is blocked). Thus, attempting to change the environment from within a handler by calling `fesetenv(3M)` or `feupdateenv` might not produce the expected results.

Name fex_merge_flags – manage the floating point environment

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <fenv.h>

void fex_merge_flags(const fenv_t *envp);
```

Description The `fex_merge_flags()` function copies into the current environment those exception flags that are set in the environment represented by the object pointed to by *envp*. The argument *envp* must point to an object set by a call to `feholdexcept(3M)` or `fegetenv(3M)` or equal to the macro `FE_DFL_ENV`. The `fex_merge_flags()` function does not raise any exceptions, but only sets its flags.

Return Values The `fex_merge_flags` function does not return a value.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWlibms, SUNWlmsx
Interface Stability	Stable
MT-Level	MT-Safe

See Also `feclearexcept(3M)`, `fegetenv(3M)`, `fesetround(3M)`, `fesetprec(3M)`, `fex_set_handling(3M)`, `fex_set_log(3M)`, `attributes(5)`

Numerical Computation Guide

Notes In a multithreaded program, the `fex_merge_flags()` function affects the floating point environment only for the calling thread.

The `fex_merge_flags()` function automatically installs and deinstalls `SIGFPE` handlers and sets and clears the trap enable mode bits in the floating point status register as needed. If a program uses these functions and attempts to install a `SIGFPE` handler or control the trap enable mode bits independently, the resulting behavior is not defined.

Name fex_set_handling, fex_get_handling, fex_getexcepthandler, fex_setexcepthandler – control floating point exception handling modes

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <fenv.h>

int fex_set_handling(int ex, int mode, void(*handler);

int fex_get_handling(int ex);

void fex_getexcepthandler(fex_handler_t *buf, int ex);

void fex_setexcepthandler(const fex_handler_t *buf, int ex);
```

Description These functions provide control of floating point exception handling modes. For each function, the *ex* argument specifies one or more exceptions indicated by a bitwise-OR of any of the following values defined in <fenv.h>:

FEX_INEXACT	
FEX_UNDERFLOW	
FEX_OVERFLOW	
FEX_DIVBYZERO	division by zero
FEX_INV_ZDZ	0/0 invalid operation
FEX_INV_IDI	infinity/infinity invalid operation
FEX_INV_ISI	infinity–infinity invalid operation
FEX_INV_ZMI	0*infinity invalid operation
FEX_INV_SQRT	square root of negative operand
FEX_INV_SNAN	signaling NaN
FEX_INV_INT	invalid integer conversion
FEX_INV_CMP	invalid comparison

For convenience, the following combinations of values are also defined:

FEX_NONE	no exceptions
FEX_INVALID	all invalid operation exceptions
FEX_COMMON	overflow, division by zero, and invalid operation
FEX_ALL	all exceptions

The fex_set_handling() function establishes the specified *mode* for handling the floating point exceptions identified by *ex*. The selected *mode* determines the action to be taken when one of the indicated exceptions occurs. It must be one of the following values:

FEX_NOHANDLER	Trap but do not otherwise handle the exception, evoking instead whatever ambient behavior would normally be in effect. This is the default behavior when the exception's trap is enabled. The *handler* parameter is ignored.
FEX_NONSTOP	Provide the IEEE 754 default result for the operation that caused the exception, set the exception's flag, and continue execution. This is the default behavior when the exception's trap is disabled. The *handler* parameter is ignored.
FEX_ABORT	Call abort(3C). The *handler* parameter is ignored.
FEX_SIGNAL	Invoke the function **handler* with the parameters normally supplied to a signal handler installed with sigfpe(3C).
FEX_CUSTOM	Invoke the function **handler* as described in the next paragraph.

In FEX_CUSTOM mode, when a floating point exception occurs, the handler function is invoked as though its prototype were:

```
#include <fenv.h>
void handler(int ex, fex_info_t *info);
```

On entry, *ex* is the value (of the first twelve listed above) corresponding to the exception that occurred, info->op indicates the operation that caused the exception, info->op1 and info->op2 contain the values of the operands, info->res contains the default untrapped result value, and info->flags reflects the exception flags that the operation would have set had it not been trapped. If the handler returns, the value contained in info->res on exit is substituted for the result of the operation, the flags indicated by info->flags are set, and execution resumes at the point where the exception occurred. The handler might modify info->res and info->flags to supply any desired result value and flags. Alternatively, if the exception is underflow or overflow, the hander might set

info->res.type = fex_nodata;

which causes the exponent-adjusted result specified by IEEE 754 to be substituted. If the handler does not modify info->res or info->flags, the effect is the same as if the exception had not been trapped.

Although the default untrapped result of an exceptional operation is always available to a FEX_CUSTOM handler, in some cases, one or both operands may not be. In these cases, the handler may be invoked with info->op1.type == fex_nodata or info->op2.type == fex_nodata to indicate that the respective data structures do not contain valid data. (For example, info->op2.type == fex_nodata if the exceptional operation is a unary operation.) Before accessing the operand values, a custom handler should always examine the type field of the operand data structures to ensure that they contain valid data in the appropriate format.

The `fex_get_handling()` function returns the current handling mode for the exception specified by *ex*, which must be one of the first twelve exceptions listed above.

The `fex_getexcepthandler()` function saves the current handling modes and associated data for the exceptions specified by *ex* in the data structure pointed to by *buf*. The type `fex_handler_t` is defined in <fenv.h>.

The `fex_setexcepthandler()` function restores the handling modes and associated data for the exceptions specified by *ex* from the data structure pointed to by *buf*. This data structure must have been set by a previous call to `fex_getexcepthandler()`. Otherwise the effect on the indicated modes is undefined.

Return Values The `fex_set_handling()` function returns a non-zero value if the requested exception handling mode is established. Otherwise, it returns 0.

Examples The following example demonstrates how to substitute a predetermined value for the result of a 0/0 invalid operation.

```
#include <math.h>
#include <fenv.h>

double k;

void presub(int ex, fex_info_t *info) {
    info->res.type = fex_double;
    info->res.val.d = k;
}

int main() {
    double x, w;
    int i;
    fex_handler_t buf;
/*
 * save current 0/0 handler
 */
    (void) fex_getexcepthandler(&buf, FEX_INV_ZDZ);
/*
 * set up presubstitution handler for 0/0
 */
    (void) fex_set_handling(FEX_INV_ZDZ, FEX_CUSTOM, presub);
/*
 *  compute (k*x)/sin(x) for k=2.0, x=0.5, 0.4, ..., 0.1, 0.0
 */
    k = 2.0;
    (void) printf("Evaluating f(x) = (k*x)/sin(x)\n\n");
    for (i = 5; i >= 0; i--) {
            x = (double) i * 0.1;
```

```
            w = (k * x) / sin(x);
            (void) printf("\tx=%3.3f\t f(x) = % 1.20e\n", x, w);
    }
/*
 * restore old 0/0 handler
 */
    (void) fex_setexcepthandler(&buf, FEX_INV_ZDZ);
    return 0;
}
```

The output from the preceding program reads:

```
Evaluating f(x) = (k*x)/sin(x)

    x=0.500  f(x) =  2.085829642933488160000e+00
    x=0.400  f(x) =  2.054345964438226260000e+00
    x=0.300  f(x) =  2.030318017094473680000e+00
    x=0.200  f(x) =  2.013395819068937610000e+00
    x=0.100  f(x) =  2.003337226326955540000e+00
    x=0.000  f(x) =  2.000000000000000000000e+00
```

When $x = 0$, $f(x)$ is computed as 0/0 and an invalid operation exception occurs. In this example, the value 2.0 is substituted for the result.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWlibms, SUNWlmxs
Interface Stability	Stable
MT-Level	MT-Safe (see Notes)

See Also sigfpe(3C), feclearexcept(3M), fegetenv(3M), fex_set_log(3M), attributes(5)

Numerical Computation Guide

Notes In a multithreaded application, the preceding functions affect exception handling modes only for the calling thread.

The functions described on this page automatically install and deinstall SIGFPE handlers and set and clear the trap enable mode bits in the floating point status register as needed. If a program uses these functions and attempts to install a SIGFPE handler or control the trap enable mode bits independently, the resulting behavior is not defined.

All traps are disabled before a handler installed in FEX_CUSTOM mode is invoked. When the SIGFPE signal is blocked, as it is when such a handler is invoked, the floating point environment, exception flags, and retrospective diagnostic functions described in

feclearexcept(3M), fegetenv(3M), and fex_set_log(3M) do not re-enable traps. Thus, the handler itself always runs in FEX_NONSTOP mode with logging of retrospective diagnostics disabled. Attempting to change these modes within the handler may not produce the expected results.

Name fex_set_log, fex_get_log, fex_set_log_depth, fex_get_log_depth, fex_log_entry – log retrospective diagnostics for floating point exceptions

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <fenv.h>

int fex_set_log(FILE *fp);

FILE *fex_get_log(void);

int fex_set_log_depth(int depth);

int fex_get_log_depth(void);

void fex_log_entry(const char *msg);
```

Description The `fex_set_log()` function enables logging of retrospective diagnostic messages regarding floating point exceptions to the file specified by *fp*. If *fp* is `NULL`, logging is disabled. When a program starts, logging is initially disabled.

The occurrence of any of the twelve exceptions listed in `fex_set_handling(3M)` constitutes an event that can be logged. To prevent the log from becoming exhorbitantly long, the logging mechanism eliminates redundant entries by two methods. First, each exception is associated with a *site* in the program. The site is identified by the address of the instruction that caused the exception together with a stack trace. Only the first exception of a given type to occur at a given site will be logged. Second, when `FEX_NONSTOP` handling mode is in effect for some exception, only those occurrences of that exception that set its previously clear flag are logged. Clearing a flag using `feclearexcept()` allows the next occurrence of the exception to be logged provided it does not occur at a site at which it was previously logged.

Each of the different types of invalid operation exceptions can be logged at the same site. Because all invalid operation exceptions share the same flag, however, of those types for which `FEX_NONSTOP` mode is in effect, only the first exception to set the flag will be logged. When the invalid operation exception is raised by a call to `feraiseexcept(3M)` or `feupdateenv(3M)`, which type of invalid operation is logged depends on the implementation.

If an exception results in the creation of a log entry, the entry is created at the time the exception occurs and before any exception handling actions selected with `fex_set_handling()` are taken. In particular, the log entry is available even if the program terminates as a result of the exception. The log entry shows the type of exception, the address of the instruction that caused it, how it will be handled, and the stack trace. If symbols are available, the address of the excepting instruction and the addresses in the stack trace are followed by the names of the corresponding symbols.

The `fex_get_log()` function returns the current log file.

The `fex_set_log_depth()` sets the maximum depth of the stack trace recorded with each exception to *depth* stack frames. The default depth is 100.

The`fex_get_log_depth()` function returns the current maximum stack trace depth.

The `fex_log_entry()` function adds a user-supplied entry to the log. The entry includes the string pointed to by *msg* and the stack trace. Like entries for floating point exceptions, redundant user-supplied entries are eliminated: only the first user-supplied entry with a given *msg* to be requested from a given site will be logged. For the purpose of a user-supplied entry, the site is defined only by the stack trace, which begins with the function that called `fex_log_entry()`.

Return Values The `fex_set_log()` function returns a non-zero value if logging is enabled or disabled accordingly and returns 0 otherwise. The `fex_set_log_depth()` returns a non-zero value if the requested stack trace depth is established (regardless of whether logging is enabled) and returns 0 otherwise.

Examples The following example demonstrates the output generated when a floating point overflow occurs in `sscanf(3C)`.

```
#include <fenv.h>

int
main() {
    double x;
/*
 * enable logging of retrospective diagnostics
 */
    (void) fex_set_log(stdout);
/*
 * establish default handling for overflows
 */
    (void) fex_set_handling(FEX_OVERFLOW, FEX_NONSTOP, NULL);
/*
 * trigger an overflow in sscanf
 */
    (void) sscanf("1.0e+400", "%lf", &x);
    return 0;
}
```

The output from the preceding program reads:

```
Floating point overflow at 0xef71cac4 __base_conversion_set_exceptio
n, nonstop mode
   0xef71cacc  __base_conversion_set_exception
   0xef721820  _decimal_to_double
   0xef75aba8  number
   0xef75a94c  __doscan_u
   0xef75ecf8  sscanf
   0x00010f20  main
```

Recompiling the program or running it on another system can produce different text addresses from those shown above.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWlibms, SUNWlmxs
Interface Stability	Stable
MT-Level	MT-Safe (see NOTES)

See Also feclearexcept(3M), fegetenv(3M), feraiseexcept(3M), feupdateenv(3M), fex_set_handling(3M), attributes(5)

Numerical Computation Guide

Notes All threads in a process share the same log file. Each call to fex_set_log() preempts the previous one.

In addition to the log file itself, two additional file descriptors are used during the creation of a log entry in order to obtain symbol names from the executable and any shared objects it uses. These file descriptors are relinquished once the log entry is written. If the file descriptors cannot be allocated, symbols names are omitted from the stack trace.

The functions described on this page automatically install and deinstall SIGFPE handlers and set and clear the trap enable mode bits in the floating point status register as needed. If a program uses these functions and attempts to install a SIGFPE handler or control the trap enable mode bits independently, the resulting behavior is not defined.

As described in fex_set_handling(), when a handling function installed in FEX_CUSTOM mode is invoked, all exception traps are disabled (and will not be reenabled while SIGFPE is blocked). Thus, retrospective diagnostic messages are not logged for exceptions that occur within such a handler.

Name floor, floorf, floorl – floor function

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double floor(double x);

float floorf(float x);

long double floorl(long double x);
```

Description These functions compute the largest integral value not greater than *x*.

Return Values Upon successful completion, these functions return the largest integral value not greater than *x*, expressed as a `double`, `float`, or `long double`, as appropriate for the return type of the function.

If *x* is NaN, a NaN is returned.

If *x* is ±Inf or ±0, *x* is returned.

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

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

See Also `ceil(3M)`, `feclearexcept(3M)`, `fetestexcept(3M)`, `isnan(3M)`, `math.h(3HEAD)`, `attributes(5)`, `standards(5)`

Name fma, fmaf, fmal – floating-point multiply-add

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double fma(double x, double y, double z);

float fmaf(float x, float y, float z);

long double fmal(long double x, long double y, long double z);
```

Description These functions compute $(x * y) + z$, rounded as one ternary operation. They compute the value (as if) to infinite precision and round once to the result format, according to the rounding mode characterized by the value of `FLT_ROUNDS`.

Return Values Upon successful completion, these functions return $(x * y) + z$, rounded as one ternary operation.

If *x* or *y* are NaN, a NaN is returned.

If *x* multiplied by *y* is an exact infinity and *z* is also an infinity but with the opposite sign, a domain error occurs and a NaN is returned.

If one of *x* and *y* is infinite, the other is 0, and *z* is not a NaN, a domain error occurs and a NaN is returned.

If *x***y* is not 0*Inf nor Inf*0 and *z* is a NaN, a NaN is returned.

Errors These functions will fail if:

Domain Error — The value of *x***y*+*z* is invalid or the value *x***y* is invalid.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, the invalid floating-point exception will be raised.

Range Error — The result overflows.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, the overflow floating-point exception will be raised.

Usage An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. An application should either examine the return value or check the floating point exception flags to detect exceptions.

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

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

See Also feclearexcept(3M), fetestexcept(3M), math.h(3HEAD), attributes(5), standards(5)

Name fmax, fmaxf, fmaxl – determine maximum numeric value of two floating-point numbers

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double fmax(double x, double y);

float fmaxf(float x, float y);

long double fmaxl(long double x, long double y);
```

Description These functions determine the maximum numeric value of their arguments. NaN arguments are treated as missing data: if one argument is a NaN and the other numeric, these functions choose the numeric value.

Return Values Upon successful completion, these functions return the maximum numeric value of their arguments.

If just one argument is a NaN, the other argument is returned.

If *x* and *y* are NaN, a NaN is returned.

Errors No errors are defined.

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

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

See Also fdim(3M), fmin(3M), math.h(3HEAD), attributes(5), standards(5)

Name fmev_shdl_init, fmev_shdl_fini, fmev_shdl_subscribe, fmev_shdl_unsubscribe, fmev_errno, fmev_strerror, fmev_attr_list, fmev_class, fmev_timespec, fmev_time_sec, fmev_time_nsec, fmev_localtime, fmev_hold, fmev_rele, fmev_dup, fmev_shdl_alloc, fmev_shdl_zalloc, fmev_shdl_free, fmev_shdlctl_serialize, fmev_shdlctl_thrattr, fmev_shdlctl_sigmask, fmev_shdlctl_thrsetup, fmev_shdlctl_thrcreate – subscription to fault management events from an external process

Synopsis

```
cc [ flag... ] file... −L/usr/lib/fm −lfmevent −lnvpair [ library... ]
#include <fm/libfmevent.h>
#include <libnvpair.h>

typedef enum fmev_err_t;
extern fmev_err_t fmev_errno;
const char *fmev_strerror(fmev_err_t err);

typedef struct fmev_shdl *fmev_shdl_t;

typedef void fmev_cbfunc_t(fmev_t, const char *, nvlist_t *, void *);

fmev_shdl_t fmev_shdl_init(uint32_t api_version,
     void *(*alloc)(size_t), void *(*zalloc)(size_t),
     void (*free)(void *, size_t));

fmev_err_t fmev_shdl_fini(fmev_shdl_t hdl);

fmev_err_t fmev_shdl_subscribe(fmev_shdl_t hdl, const char *classpat,
     fmev_cbfunc_t callback, void *cookie);

fmev_err_t fmev_shdl_unsubscribe(fmev_shdl_t hdl,
     const char *classpat);

fmev_err_t fmev_shdlctl_serialize(fmev_shdl_t hdl);

fmev_err_t fmev_shdlctl_thrattr(fmev_shdl_t hdl, pthread_attr_t *attr);

fmev_err_t fmev_shdlctl_sigmask(fmev_shdl_t hdl, sigset_t *set);

fmev_err_t fmev_shdlctl_thrsetup(fmev_shdl_t hdl,
     door_xcreate_thrsetup_func_t *setupfunc, void *cookie);

fmev_err_t fmev_shdlctl_thrcreate(fmev_shdl_t hdl,
     door_xcreate_server_func_t *createfunc, void *cookie);

typedef struct fmev *fmev_t;

nvlist_t *fmev_attr_list(fmev_t ev)

const char *fmev_class(fmev_t ev);

fmev_err_t fmev_timespec(fmev_t ev, struct timespec *res);

uint64_t fmev_time_sec(fmev_t ev);

uint64_t fmev_time_nsec(fmev_t ev);

struct tm *fmev_localtime(fmev_t ev, struct tm *res);
```

```
void fmev_hold(fmev_t ev);

void fmev_rele(fmev_t ev);

fmev_t fmev_dup(fmev_t ev);

void *fmev_shdl_alloc(fmev_shdl_t hdl, size_t sz);

void *fmev_shdl_zalloc(fmev_shdl_t hdl, size_t sz);

void fmev_shdl_free(fmev_shdl_t hdl, void *buf, size_t sz);
```

Description

The Solaris fault management daemon (fmd) is the central point in Solaris for fault management. It receives fault management protocol events from various sources and publishes additional protocol events such as to describe a diagnosis it has arrived at or a subsequent repair event. The event protocol is specified in the Sun Fault Management Event Protocol Specification. The interfaces described here allow an external process to subscribe to protocol events. See the Fault Management Daemon Programmer's Reference Guide for additional information on fmd.

The fmd module API (not a Committed interface) allows plugin modules to load within the fmd process, subscribe to events of interest, and participate in various diagnosis and response activities. Of those modules, some are notification agents and will subscribe to events describing diagnoses and their subsequent lifecycle and render these to console/syslog (for the `syslog-msgs` agent) and via SNMP trap and browsable MIB (for the `snmp-trapgen` module and the corresponding dlmod for the SNMP daemon). It has not been possible to subscribe to protocol events outside of the context of an fmd plugin. The `libfmevent` interface provides this external subscription mechanism. External subscribers may receive protocol events as fmd modules do, but they cannot participate in other aspects of the fmd module API such as diagnosis. External subscribers are therefore suitable as notification agents and for transporting fault management events.

Fault Management Protocol Events

This protocol is defined in the Sun Fault Management Event Protocol Specification. Note that while the API described on this manual page are Committed, the protocol events themselves (in class names and all event payload) are not Committed along with this API. The protocol specification document describes the commitment level of individual event classes and their payload content. In broad terms, the list.* events are Committed in most of their content and semantics while events of other classes are generally Uncommitted with a few exceptions.

All protocol events include an identifying class string, with the hierarchies defined in the protocol document and individual events registered in the Events Registry. The `libfmevent` mechanism will permit subscription to events with Category 1 class of “list” and “swevent”, that is, to classes matching patterns “list.*” and “swevent.*”.

All protocol events consist of a number of (name, datatype, value) tuples (“nvpairs”). Depending on the event class various nvpairs are required and have well-defined meanings. In Solaris fmd protocol events are represented as name-value lists using the `libnvpair(3LIB)` interfaces.

API Overview

The API is simple to use in the common case (see Examples), but provides substantial control to cater for more-complex scenarios.

We obtain an opaque subscription handle using `fmev_shdl_init()`, quoting the ABI version and optionally nominating `alloc()`, `zalloc()` and `free()` functions (the defaults use the `umem` family). More than one handle may be opened if desired. Each handle opened establishes a communication channel with fmd, the implementation of which is opaque to the `libfmevent` mechanism.

On a handle we may establish one or more subscriptions using `fmev_shdl_subscribe()`. Events of interest are specified using a simple wildcarded pattern which is matched against the event class of incoming events. For each match that is made a callback is performed to a function we associate with the subscription, passing a nominated cookie to that function. Subscriptions may be dropped using `fmev_shdl_unsubscribe()` quoting exactly the same class or class pattern as was used to establish the subscription.

Each call to `fmev_shdl_subscribe()` creates a single thread dedicated to serving callback requests arising from this subscription.

An event callback handler has as arguments an opaque event handle, the event class, the event nvlist, and the cookie it was registered with in `fmev_shdl_subscribe()`. The timestamp for when the event was generated (not when it was received) is available as a `struct timespec` with `fmev_timespec()`, or more directly with `fmev_time_sec()` and `fmev_time_nsec()`; an event handle and `struct tm` can also be passed to `fmev_localtime()` to fill the `struct tm`.

The event handle, class string pointer, and `nvlist_t` pointer passed as arguments to a callback are valid for the duration of the callback. If the application wants to continue to process the event beyond the duration of the callback then it can hold the event with `fmev_hold()`, and later release it with `fmev_rele()`. When the reference count drops to zero the event is freed.

Error Handling

In `<libfmevent.h>` an enumeration `fmev_err_t` of error types is defined. To render an error message string from an `fmev_err_t` use `fmev_strerror()`. An `fmev_errno` is defined which returns the error number for the last failed `libfmevent` API call made by the current thread. You may not assign to `fmev_errno`.

If a function returns type `fmev_err_t`, then success is indicated by `FMEV_SUCCESS` (or `FMEV_OK` as an alias); on failure a `FMEVERR_*` value is returned (see `<fm/libfmevent.h>`).

If a function returns a pointer type then failure is indicated by a `NULL` return, and `fmev_errno` will record the error type.

Subscription Handles

A subscription handle is required in order to establish and manage subscriptions. This handle represents the abstract communication mechanism between the application and the fault management daemon running in the current zone.

A subscription handle is represented by the opaque `fmev_shdl_t` datatype. A handle is initialized with `fmev_shdl_init()` and quoted to subsequent API members.

To simplify usage of the API, subscription attributes for all subscriptions established on a handle are a property of the handle itself ; they cannot be varied per-subscription. In such use cases multiple handles will need to be used.

`libfmevent` ABI version

The first argument to `fmev_shdl_init()` indicates the `libfmevent` ABI version with which the handle is being opened. Specify either `LIBFMEVENT_VERSION_LATEST` to indicate the most recent version available at compile time or `LIBFMEVENT_VERSION_1` (_2, etc. as the interface evolves) for an explicit choice.

Interfaces present in an earlier version of the interface will continue to be present with the same or compatible semantics in all subsequent versions. When additional interfaces and functionality are introduced the ABI version will be incremented. When an ABI version is chosen in `fmev_shdl_init()`, only interfaces introduced in or before that version will be available to the application via that handle. Attempts to use later API members will fail with `FMEVERR_VERSION_MISMATCH`.

This manual page describes `LIBFMEVENT_VERSION_1`.

Privileges

The `libfmevent` API is not least-privilege aware; you need to have all privileges to call `fmev_shdl_init()`. Once a handle has been initialized with `fmev_shdl_init()` a process can drop privileges down to the basic set and continue to use `fmev_shdl_subscribe()` and other `libfmevent` interfaces on that handle.

Underlying Event Transport

The implementation of the event transport by which events are published from the fault manager and multiplexed out to `libfmevent` consumers is strictly private. It is subject to change at any time, and you should not encode any dependency on the underlying mechanism into your application. Use only the API described on this manual page and in `<libfmevent.h>`.

The underlying transport mechanism is guaranteed to have the property that a subscriber may attach to it even before the fault manager is running. If the fault manager starts first then any events published before the first consumer subscribes will wait in the transport until a consumer appears.

The underlying transport will also have some maximum depth to the queue of events pending delivery. This may be hit if there are no consumers, or if consumers are not processing events quickly enough. In practice the rate of events is small. When this maximum depth is reached additional events will be dropped.

The underlying transport has no concept of priority delivery; all events are treated equally.

Subscription Handle Initialization

Obtain a new subscription handle with `fmev_shdl_init()`. The first argument is the `libfmevent` ABI version to be used (see above). The remaining three arguments should be all `NULL` to leave the library to use its default allocator functions (the `libumem` family), or all non-`NULL` to appoint wrappers to custom allocation functions if required.

`FMEVERR_VERSION_MISMATCH`
The library does not support the version requested.

FMEVERR_ALLOC
An error occurred in trying to allocate data structures.

FMEVERR_API
The alloc(), zalloc(), or free() arguments must either be all NULL or all non-NULL.

FMEVERR_NOPRIV
Insufficient privilege to perform operation. In version 1 root privilege is required.

FMEVERR_INTERNAL
Internal library error.

Subscription Handle Finalization

Close a subscription handle with fmev_shdl_fini(). This call must not be performed from within the context of an event callback handler, else it will fail with FMEVERR_API.

The fmev_shdl_fini() call will remove all active subscriptions on the handle and free resources used in managing the handle.

FMEVERR_API
May not be called from event delivery context for a subscription on the same handle.

Subscribing To Events

To establish a new subscription on a handle, use fmev_shdl_subscribe(). Besides the handle argument you provide the class or class pattern to subscribe to (the latter permitting simple wildcarding using '*'), a callback function pointer for a function to be called for all matching events, and a cookie to pass to that callback function.

The class pattern must match events per the fault management protocol specification, such as "list.suspect" or "list.*". Patterns that do not map onto existing events will not be rejected - they just won't result in any callbacks.

A callback function has type fmev_cbfunc_t. The first argument is an opaque event handle for use in event access functions described below. The second argument is the event class string, and the third argument is the event nvlist; these could be retrieved using fmev_class() and fmev_attr_list() on the event handle, but they are supplied as arguments for convenience. The final argument is the cookie requested when the subscription was established in fmev_shdl_subscribe().

Each call to fmev_shdl_subscribe() opens a new door into the process that the kernel uses for event delivery. Each subscription therefore uses one file descriptor in the process.

See below for more detail on event callback context.

FMEVERR_API
Class pattern is NULL or callback function is NULL.

FMEVERR_BADCLASS
Class pattern is the empty string, or exceeds the maximum length of FMEV_MAX_CLASS.

FMEVERR_ALLOC
An attempt to fmev_shdl_zalloc() additional memory failed.

FMEVERR_DUPLICATE
Duplicate subscription request. Only one subscription for a given class pattern may exist on a handle.

FMEVERR_MAX_SUBSCRIBERS
A system-imposed limit on the maximum number of subscribers to the underlying transport mechanism has been reached.

FMEVERR_INTERNAL
An unknown error occurred in trying to establish the subscription.

Unsubscribing

An unsubscribe request using `fmev_shdl_unsubscribe()` must exactly match a previous subscription request or it will fail with `FMEVERR_NOMATCH`. The request stops further callbacks for this subscription, waits for any existing active callbacks to complete, and drops the subscription.

Do not call `fmev_shdl_unsubscribe` from event callback context, else it will fail with `FMEVERR_API`.

FMEVERR_API
A `NULL` pattern was specified, or the call was attempted from callback context.

FMEVERR_NOMATCH
The pattern provided does not match any open subscription. The pattern must be an exact match.

FMEVERR_BADCLASS
The class pattern is the empty string or exceeds `FMEV_MAX_CLASS`.

Event Callback Context

Event callback context is defined as the duration of a callback event, from the moment we enter the registered callback function to the moment it returns. There are a few restrictions on actions that may be performed from callback context:

- You can perform long-running actions, but this thread will not be available to service other event deliveries until you return.
- You must not cause the current thread to exit.
- You must not call either `fmev_shdl_unsubscribe()` or `fmev_shdl_fini()` for the subscription handle on which this callback has been made.
- You can invoke `fork()`, `popen()`, etc.

Event Handles

A callback receives an `fmev_t` as a handle on the associated event. The callback may use the access functions described below to retrieve various event attributes.

By default, an event handle `fmev_t` is valid for the duration of the callback context. You cannot access the event outside of callback context.

If you need to continue to work with an event beyond the initial callback context in which it is received, you may place a “hold” on the event with `fmev_hold()`. When finished with the

event, release it with `fmev_rele()`. These calls increment and decrement a reference count on the event; when it drops to zero the event is freed. On initial entry to a callback the reference count is 1, and this is always decremented when the callback returns.

An alternative to `fmev_hold()` is `fmev_dup()`, which duplicates the event and returns a new event handle with a reference count of 1. When `fmev_rele()` is applied to the new handle and reduces the reference count to 0, the event is freed. The advantage of `fmev_dup()` is that it allocates new memory to hold the event rather than continuing to hold a buffer provided by the underlying delivery mechanism. If your operation is going to be long-running, you may want to use `fmev_dup()` to avoid starving the underlying mechanism of event buffers.

The `fmev_hold()` and `fmev_rele()` functions always succeed.

The `fmev_dup()` function may fail and return `NULL` with `fmev_errno` of:

`FMEVERR_API` A `NULL` event handle was passed.

`FMEVERR_ALLOC` The `fmev_shdl_alloc()` call failed.

Event Class

A delivery callback already receives the event class as an argument, so `fmev_class()` will only be of use outside of callback context (that is, for an event that was held or duped in callback context and is now being processed in an asynchronous handler). This is a convenience function that returns the same result as accessing the event attributes with `fmev_attr_list()` and using nvlist_lookup_string(3NVPAIR) to lookup a string member of name "class".

The string returned by `fmev_class()` is valid for as long as the event handle itself.

The `fmev_class()` function may fail and return `NULL` with `fmev_errno` of:

`FMEVERR_API` A `NULL` event handle was passed.

`FMEVERR_MALFORMED_EVENT` The event appears corrupted.

Event Attribute List

All events are defined as a series of (name, type) pairs. An instance of an event is therefore a series of tuples (name, type, value). Allowed types are defined in the protocol specification. In Solaris, and in `libfmevent`, an event is represented as an `nvlist_t` using the `libnvpair(3LIB)` library.

The nvlist of event attributes can be accessed using `fmev_attr_list()`. The resulting `nvlist_t` pointer is valid for the same duration as the underlying event handle. Do not use `nvlist_free()` to free the nvlist. You may then lookup members, iterate over members, and so on using the `libnvpair` interfaces.

The `fmev_attr_list()` function may fail and return `NULL` with `fmev_errno` of:

`FMEVERR_API` A `NULL` event handle was passed.

`FMEVERR_MALFORMED_EVENT` The event appears corrupted.

Event Timestamp

These functions refer to the time at which the event was originally produced, not the time at which it was forwarded to `libfmevent` or delivered to the callback.

Use `fmev_timespec()` to fill a `struct timespec` with the event time in seconds since the Epoch (`tv_sec`, signed integer) and nanoseconds past that second (`tv_nsec`, a signed long). This call can fail and return `FMEVERR_OVERFLOW` if the seconds value will not fit in a signed 32-bit integer (as used in `struct timespec tv_sec`).

You can use `fmev_time_sec()` and `fmev_time_nsec()` to retrieve the same second and nanosecond values as `uint64_t` quantities.

The `fmev_localtime` function takes an event handle and a `struct tm` pointer and fills that structure according to the timestamp. The result is suitable for use with `strftime(3C)`. This call will return `NULL` and `fmev_errno` of `FMEVERR_OVERFLOW` under the same conditions as above.

`FMEVERR_OVERFLOW` The `fmev_timespec()` function cannot fit the seconds value into the signed long integer `tv_sec` member of a `struct timespec`.

Memory Allocation

The `fmev_shdl_alloc()`, `fmev_shdl_zalloc()`, and `fmev_shdl_free()` functions allocate and free memory using the choices made for the given handle when it was initialized, typically the `libumem(3LIB)` family if all were specified `NULL`.

Subscription Handle Control

The `fmev_shdlctl_*()` interfaces offer control over various properties of the subscription handle, allowing fine-tuning for particular applications. In the common case the default handle properties will suffice.

These properties apply to the handle and uniformly to all subscriptions made on that handle. The properties may only be changed when there are no subscriptions in place on the handle, otherwise `FMEVERR_BUSY` is returned.

Event delivery is performed through invocations of a private door. A new door is opened for each `fmev_shdl_subscribe()` call. These invocations occur in the context of a single private thread associated with the door for a subscription. Many of the `fmev_shdlctl_*()` interfaces are concerned with controlling various aspects of this delivery thread.

If you have applied `fmev_shdlctl_thrcreate()`, "custom thread creation semantics" apply on the handle; otherwise "default thread creation semantics" are in force. Some `fmev_shdlctl_*()` interfaces apply only to default thread creation semantics.

The `fmev_shdlctl_serialize()` control requests that all deliveries on a handle, regardless of which subscription request they are for, be serialized - no concurrent deliveries on this handle. Without this control applied deliveries arising from each subscription established with `fmev_shdl_subscribe()` are individually single-threaded, but if multiple subscriptions have been established then deliveries arising from separate subscriptions may be concurrent. This control applies to both custom and default thread creation semantics.

The `fmev_shdlctl_thrattr()` control applies only to default thread creation semantics. Threads that are created to service subscriptions will be created with `pthread_create(3C)` using the `pthread_attr_t` provided by this interface. The attribute structure is not copied and so must persist for as long as it is in force on the handle.

The default thread attributes are also the minimum requirement: threads must be created `PTHREAD_CREATE_DETACHED` and `PTHREAD_SCOPE_SYSTEM`. A `NULL` pointer for the `pthread_attr_t` will reinstate these default attributes.

The `fmev_shdlctl_sigmask()` control applies only to default thread creation semantics. Threads that are created to service subscriptions will be created with the requested signal set masked - a `pthread_sigmask(3C)` request to `SIG_SETMASK` to this mask prior to `pthread_create()`. The default is to mask all signals except `SIGABRT`.

See `door_xcreate(3DOOR)` for a detailed description of thread setup and creation functions for door server threads.

The `fmev_shdlctl_thrsetup()` function runs in the context of the newly-created thread before it binds to the door created to service the subscription. It is therefore a suitable place to perform any thread-specific operations the application may require. This control applies to both custom and default thread creation semantics.

Using `fmev_shdlctl_thrcreate()` forfeits the default thread creation semantics described above. The function appointed is responsible for all of the tasks required of a `door_xcreate_server_func_t` in `door_xcreate()`.

The `fmev_shdlctl_*()` functions may fail and return `NULL` with `fmev_errno` of:

`FMEVERR_BUSY` Subscriptions are in place on this handle.

Examples

EXAMPLE 1 Subscription example

The following example subscribes to `list.suspect` events and prints out a simple message for each one that is received. It foregoes most error checking for the sake of clarity.

```
#include <fm/libfmevent.h>
#include <libnvpair.h>

/*
 * Callback to receive list.suspect events
 */
void
mycb(fmev_t ev, const char *class, nvlist_t *attr, void *cookie)
{
        struct tm tm;
        char buf[64];
        char *evcode;
```

EXAMPLE 1 Subscription example *(Continued)*

```
        if (strcmp(class, "list.suspect") != 0)
                return; /* only happens if this code has a bug! */

        (void) strftime(buf, sizeof (buf), NULL,
            fmev_localtime(ev, &tm));

        (void) nvlist_lookup_string(attr, "code", &evcode);

        (void) fprintf(stderr, "Event class %s published at %s, "
            "event code %s\
", class, buf, evcode);
}

int
main(int argc, char *argv[])
{
        fmev_shdl_t hdl;
        sigset_t set;

        hdl = fmev_shdl_init(LIBFMEVENT_VERSION_LATEST,
            NULL, NULL, NULL);

        (void) fmev_shdl_subscribe(hdl, "list.suspect", mycb, NULL);

        /* Wait here until signalled with SIGTERM to finish */
        (void) sigemptyset(&set);
        (void) sigaddset(&set, SIGTERM);
        (void) sigwait(&set);

        /* fmev_shdl_fini would do this for us if we skipped it */
        (void) fmev_shdl_unsubscribe(hdl, "list.suspect");

        (void) fmev_shdl_fini(hdl);

        return (0);
}
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	all
Availability	SUNWfmd

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also door_xcreate(3DOOR), libnvpair(3LIB), libumem(3LIB), nvlist_lookup_string(3NVPAIR), pthread_create(3C), pthread_sigmask(3C), strftime(3C), attributes(5), privileges(5)

Name fmin, fminf, fminl – determine minimum numeric value of two floating-point numbers

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double fmin(double x, double y);

float fminf(float float x, float y);

long double fminl(long double x, long double y);
```

Description These functions determine the minimum numeric value of their arguments. NaN arguments are treated as missing data: if one argument is a NaN and the other numeric, these functions choose the numeric value.

Return Values Upon successful completion, these functions return the minimum numeric value of their arguments.

If just one argument is a NaN, the other argument is returned.

If *x* and *y* are NaN, a NaN is returned.

Errors No errors are defined.

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

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

See Also fdim(3M), fmax(3M), math.h(3HEAD), attributes(5), standards(5)

Name fmod, fmodf, fmodl – floating-point remainder value function

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double fmod(double x, double y);

float fmodf(float x, float y);

long double fmodl(long double x, long double y);
```

Description These functions return the floating-point remainder of the division of *x* by *y*.

Return Values These functions return the value $x - i * y$, for some integer *i* such that, if *y* is non-zero, the result has the same sign as *x* and magnitude less than the magnitude of *y*.

If *x* or *y* is NaN, a NaN is returned.

If *y* is 0, a domain error occurs and a NaN is returned.

If *x* is infinite, a domain error occurs and a NaN is returned.

If *x* is ±0 and *y* is not 0, ±0 is returned.

If *x* is not infinite and *y* is ±Inf, *x* is returned.

Errors These functions will fail if:

Domain Error — The *x* argument is infinite or *y* is 0.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, the invalid floating-point exception is raised.

Usage An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. An application should either examine the return value or check the floating point exception flags to detect exceptions.

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

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

See Also `feclearexcept(3M)`, `fetestexcept(3M)`, `isnan(3M)`, `math.h(3HEAD)`, `attributes(5)`, `standards(5)`

Name fpclassify – classify real floating type

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

int fpclassify(real-floating x);
```

Description The `fpclassify()` macro classifies its argument value as NaN, infinite, normal, subnormal, or zero. First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then classification is based on the type of the argument.

Return Values The `fpclassify()` macro returns the value of the number classification macro appropriate to the value of its argument.

Errors No errors are defined.

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

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

See Also `isfinite(3M)`, `isinf(3M)`, `isnan(3M)`, `isnormal(3M)`, `math.h(3HEAD)`, `signbit(3M)`, `attributes(5)`, `standards(5)`

Name freeDmiString – free dynamic memory allocated for input DmiString structure

Synopsis

```
cc [ flag ... ] file ... -ldmi -lnsl -lrwtool [ library ... ]
#include <dmi/util.hh>

void freeDmiString(DmiString_t *dstr);
```

Description The `freeDmiString()` function frees dynamic memory allocated for the input `DmiString` structure.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-level	MT-Safe

See Also `newDmiString(3DMI)`, `libdmi(3LIB)`, `attributes(5)`

Name frexp, frexpf, frexpl – extract mantissa and exponent from a floating-point number

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double frexp(double num, int *exp);

float frexpf(float num, int *exp);

long double frexpl(long double num, int *exp);
```

Description These functions break a floating-point number into a normalized fraction and an integral power of 2. They store the integer exponent in the `int` object pointed to by *exp*.

Return Values For finite arguments, these functions return the value *x*, such that *x* is a `double` with magnitude in the interval [½, 1) or 0, and *num* equals *x* times 2 raised to the power **exp*.

If *num* is NaN, NaN is returned and the value of **exp* is unspecified.

If *num* is ± 0, ± 0 is returned and the value of **exp* is 0.

If *num* is ±Inf, *num* is returned and the value of **exp* is unspecified.

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

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

See Also `isnan(3M)`, `ldexp(3M)`, `modf(3M)`, `attributes(5)`, `standards(5)`

Name gelf, gelf_checksum, gelf_fsize, gelf_getcap, gelf_getclass, gelf_getdyn, gelf_getehdr, gelf_getmove, gelf_getphdr, gelf_getrel, gelf_getrela, gelf_getshdr, gelf_getsym, gelf_getsyminfo, gelf_getsymshndx, gelf_newehdr, gelf_newphdr, gelf_update_cap, gelf_update_dyn, gelf_update_ehdr, gelf_update_getmove, gelf_update_move, gelf_update_phdr, gelf_update_rel, gelf_update_rela, gelf_update_shdr, gelf_update_sym, gelf_update_symshndx, gelf_update_syminfo, gelf_xlatetof, gelf_xlatetom – generic class-independent ELF interface

Synopsis

```
cc [ flag... ] file...  -lelf [ library... ]
#include <gelf.h>

long gelf_checksum(Elf *elf);

size_t gelf_fsize(Elf *elf, Elf_Type type, size_t cnt, unsigned ver);

int gelf_getcap(Elf_Data *src, int ndx, GElf_Cap *dst);

int gelf_getclass(Elf *elf);

GElf_Dyn *gelf_getdyn(Elf_Data *src, int ndx, GElf_Dyn *dst);

GElf_Ehdr *gelf_getehdr(Elf *elf, GElf_Ehdr *dst);

GElf_Move *gelf_getmove(Elf_Data *src, int ndx, GElf_Move *dst);

GElf_Phdr *gelf_getphdr(Elf *elf, int ndx, GElf_Phdr *dst);

GElf_Rel *gelf_getrel(Elf_Data *src, int ndx, GElf_Rel *dst);

GElf_Rela *gelf_getrela(Elf_Data *src, int ndx, GElf_Rela *dst);

GElf_Shdr *gelf_getshdr(Elf_Scn *scn, GElf_Shdr *dst);

GElf_Sym *gelf_getsym(Elf_Data *src, int ndx, GElf_Sym *dst);

GElf_Syminfo *gelf_getsyminfo(Elf_Data *src, int ndx, GElf_Syminfo *dst);

GElf_Sym *gelf_getsymshndx(Elf_Data *symsrc, Elf_Data *shndxsrc,
     int ndx, GElf_Sym *symdst, Elf32_Word *shndxdst);

unsigned long gelf_newehdr(Elf *elf, int class);

unsigned long gelf_newphdr(Elf *elf, size_t phnum);

int gelf_update_cap(Elf_Data *dst, int ndx, GElf_Cap *src);

int gelf_update_dyn(Elf_Data *dst, int ndx, GElf_Dyn *src);

int gelf_update_ehdr(Elf *elf, GElf_Ehdr *src);

int gelf_update_move(Elf_Data *dst, int ndx, GElf_Move *src);

int gelf_update_phdr(Elf *elf, int ndx, GElf_Phdr *src);

int gelf_update_rel(Elf_Data *dst, int ndx, GElf_Rel *src);

int gelf_update_rela(Elf_Data *dst, int ndx, GElf_Rela *src);

int gelf_update_shdr(Elf_Scn *dst, GElf_Shdr *src);
```

```
int gelf_update_sym(Elf_Data *dst, int ndx, GElf_Sym *src);

int gelf_update_syminfo(Elf_Data *dst, int ndx, GElf_Syminfo *src);

int gelf_update_symshndx(Elf_Data *symdst, Elf_Data *shndxdst, int ndx,
     GElf_Sym *symsrc, Elf32_Word shndxsrc);

Elf_Data *gelf_xlatetof(Elf *elf, Elf_Data *dst, const Elf_Data *src,
     unsigned encode);

Elf_Data *gelf_xlatetom(Elf *elf, Elf_Data *dst, const Elf_Data *src,
     unsigned encode);
```

Description

`GElf` is a generic, `ELF` class-independent API for manipulating `ELF` object files. `GElf` provides a single, common interface for handling 32–bit and 64–bit `ELF` format object files. `GElf` is a translation layer between the application and the class-dependent parts of the `ELF` library. Thus, the application can use `GElf`, which in turn, will call the corresponding `elf32_` or `elf64_` functions on behalf of the application. The data structures returned are all large enough to hold 32–bit and 64–bit data.

`GElf` provides a simple, class-independent layer of indirection over the class-dependent `ELF32` and `ELF64` `API`'s. `GElf` is stateless, and may be used along side the `ELF32` and `ELF64` API's.

`GElf` always returns a copy of the underlying `ELF32` or `ELF64` structure, and therefore the programming practice of using the address of an `ELF` header as the base offset for the `ELF`'s mapping into memory should be avoided. Also, data accessed by type-casting the `Elf_Data` buffer to a class-dependent type and treating it like an array, for example, a symbol table, will not work under `GElf`, and the `gelf_get` functions must be used instead. See the EXAMPLE section.

Programs that create or modify `ELF` files using `libelf(3LIB)` need to perform an extra step when using `GElf`. Modifications to `GElf` values must be explicitly flushed to the underlying `ELF32` or `ELF64` structures by way of the `gelf_update_` interfaces. Use of `elf_update` or `elf_flagelf` and the like remains the same.

The sizes of versioning structures remain the same between `ELF32` and `ELF64`. The `GElf` API only defines types for versioning, rather than a functional API. The processing of versioning information will stay the same in the `GElf` environment as it was in the class-dependent `ELF` environment.

List of Functions

`gelf_checksum()`	An analog to `elf32_checksum(3ELF)` and `elf64_checksum(3ELF)`.
`gelf_fsize()`	An analog to `elf32_fsize(3ELF)` and `elf64_fsize(3ELF)`.
`gelf_getcap()`	Retrieves the `Elf32_Cap` or `Elf64_Cap` information from the capability table at the given index. `dst` points to the location where the `GElf_Cap` capability entry is stored.
`gelf_getclass()`	Returns one of the constants `ELFCLASS32`, `ELFCLASS64` or `ELFCLASSNONE`.

`gelf_getdyn()`	Retrieves the `Elf32_Dyn` or `Elf64_Dyn` information from the dynamic table at the given index. `dst` points to the location where the `GElf_Dyn` dynamic entry is stored.
`gelf_getehdr()`	An analog to `elf32_getehdr`(3ELF) and `elf64_getehdr`(3ELF). `dst` points to the location where the `GElf_Ehdr` header is stored.
`gelf_getmove()`	Retrieves the `Elf32_Move` or `Elf64_Move` information from the move table at the given index. `dst` points to the location where the `GElf_Move` move entry is stored.
`gelf_getphdr()`	An analog to`elf32_getphdr`(3ELF) and `elf64_getphdr`(3ELF). `dst` points to the location where the `GElf_Phdr` program header is stored.
`gelf_getrel()`	Retrieves the `Elf32_Rel` or `Elf64_Rel` information from the relocation table at the given index. `dst` points to the location where the `GElf_Rel` relocation entry is stored.
`gelf_getrela()`	Retrieves the `Elf32_Rela` or `Elf64_Rela` information from the relocation table at the given index. `dst` points to the location where the `GElf_Rela` relocation entry is stored.
`gelf_getshdr()`	An analog to `elf32_getshdr`(3ELF) and `elf64_getshdr`(3ELF). `dst` points to the location where the `GElf_Shdr` section header is stored.
`gelf_getsym()`	Retrieves the `Elf32_Sym` or `Elf64_Sym` information from the symbol table at the given index. `dst` points to the location where the `GElf_Sym` symbol entry is stored.
`gelf_getsyminfo()`	Retrieves the `Elf32_Syminfo` or `Elf64_Syminfo` information from the relocation table at the given index. `dst` points to the location where the `GElf_Syminfo` symbol information entry is stored.
`gelf_getsymshndx()`	Provides an extension to `gelf_getsym()` that retrieves the `Elf32_Sym` or `Elf64_Sym` information, and the section index from the symbol table at the given index *ndx*. The symbols section index is typically recorded in the `st_shndx` field of the symbols structure. However, a file that requires ELF Extended Sections may record an `st_shndx` of `SHN_XINDEX` indicating that the section index must be obtained from an associated `SHT_SYMTAB_SHNDX` section entry. If *xshndx* and *shndxdata* are non-null, the value recorded at index *ndx* of

	the SHT_SYMTAB_SHNDX table pointed to by *shndxdata* is returned in *xshndx*. See USAGE.
gelf_newehdr()	An analog to elf32_newehdr(3ELF) and elf64_newehdr(3ELF).
gelf_newphdr()	An analog to elf32_newphdr(3ELF) and elf64_newphdr(3ELF).
gelf_update_cap()	Copies the GElf_Cap information back into the underlying Elf32_Cap or Elf64_Cap structure at the given index.
gelf_update_dyn()	Copies the GElf_Dyn information back into the underlying Elf32_Dyn or Elf64_Dyn structure at the given index.
gelf_update_ehdr()	Copies the contents of the GElf_Ehdr ELF header to the underlying Elf32_Ehdr or Elf64_Ehdr structure.
gelf_update_move()	Copies the GElf_Move information back into the underlying Elf32_Move or Elf64_Move structure at the given index.
gelf_update_phdr()	Copies of the contents of GElf_Phdr program header to underlying the Elf32_Phdr or Elf64_Phdr structure.
gelf_update_rel()	Copies the GElf_Rel information back into the underlying Elf32_Rel or Elf64_Rel structure at the given index.
gelf_update_rela()	Copies the GElf_Rela information back into the underlying Elf32_Rela or Elf64_Rela structure at the given index.
gelf_update_shdr()	Copies of the contents of GElf_Shdr section header to underlying the Elf32_Shdr or Elf64_Shdr structure.
gelf_update_sym()	Copies the GElf_Sym information back into the underlying Elf32_Sym or Elf64_Sym structure at the given index.
gelf_update_syminfo()	Copies the GElf_Syminfo information back into the underlying Elf32_Syminfo or Elf64_Syminfo structure at the given index.
gelf_update_symshndx()	Provides an extension to gelf_update_sym() that copies the GElf_Sym information back into the Elf32_Sym or Elf64_Sym structure at the given index *ndx*, and copies the extended *xshndx* section index into the Elf32_Word at the given index *ndx* in the buffer described by shndxdata. See USAGE.
gelf_xlatetof()	An analog to elf32_xlatetof(3ELF) and elf64_xlatetof(3ELF)
gelf_xlatetom()	An analog to elf32_xlatetom(3ELF) and elf64_xlatetom(3ELF)

Return Values Upon failure, all `GElf` functions return `0` and set `elf_errno`. See `elf_errno(3ELF)`

Examples **EXAMPLE 1** Printing the ELF Symbol Table

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <libelf.h>
#include <gelf.h>

void
main(int argc, char **argv)
{
    Elf         *elf;
    Elf_Scn     *scn = NULL;
    GElf_Shdr   shdr;
    Elf_Data    *data;
    int         fd, ii, count;

    elf_version(EV_CURRENT);

    fd = open(argv[1], O_RDONLY);
    elf = elf_begin(fd, ELF_C_READ, NULL);

    while ((scn = elf_nextscn(elf, scn)) != NULL) {
        gelf_getshdr(scn, &shdr);
        if (shdr.sh_type == SHT_SYMTAB) {
            /* found a symbol table, go print it. */
            break;
        }
    }

    data = elf_getdata(scn, NULL);
    count = shdr.sh_size / shdr.sh_entsize;

    /* print the symbol names */
    for (ii = 0; ii < count; ++ii) {
        GElf_Sym sym;
        gelf_getsym(data, ii, &sym);
        printf("%s\n", elf_strptr(elf, shdr.sh_link, sym.st_name));
    }
    elf_end(elf);
    close(fd);
}
```

Usage ELF Extended Sections are employed to allow an ELF file to contain more than `0xff00` (`SHN_LORESERVE`) section. See the *Linker and Libraries Guide* for more information.

Files

`/lib/libelf.so.1`	shared object
`/lib/64/libelf.so.1`	64–bit shared object

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

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

See Also `elf(3ELF)`, `elf32_checksum(3ELF)`, `elf32_fsize(3ELF)`, `elf32_getehdr(3ELF)`, `elf32_newehdr(3ELF)`, `elf32_getphdr(3ELF)`, `elf32_newphdr(3ELF)`, `elf32_getshdr(3ELF)`, `elf32_xlatetof(3ELF)`, `elf32_xlatetom(3ELF)`, `elf_errno(3ELF)`, `libelf(3LIB)`, `attributes(5)`

Linker and Libraries Guide

REFERENCE

Extended Library Functions - Part 3

Name getacinfo, getacdir, getacflg, getacmin, getacna, setac, endac – get audit control file information

Synopsis

```
cc [ flag... ] file... -lbsm -lsocket -lnsl [ library... ]
#include <bsm/libbsm.h>

int getacdir( char *dir, int len);

int getacmin( int *min_val);

int getacflg( char *auditstring, int len);

int getacna( char *auditstring, int len);

void setac(void);

void endac(void);
```

Description When first called, `getacdir()` provides information about the first audit directory in the `audit_control` file. Thereafter, it returns the next directory in the file. Successive calls list all the directories listed in `audit_control(4)` The *len* argument specifies the length of the buffer *dir*. On return, *dir* points to the directory entry.

The `getacmin()` function reads the minimum value from the `audit_control` file and returns the value in *min_val*. The minimum value specifies how full the file system to which the audit files are being written can get before the script `audit_warn(1M)` is invoked.

The `getacflg()` function reads the system audit value from the `audit_control` file and returns the value in *auditstring*. The *len* argument specifies the length of the buffer *auditstring*.

The `getacna()` function reads the system audit value for non-attributable audit events from the `audit_control` file and returns the value in *auditstring*. The *len* argument specifies the length of the buffer *auditstring*. Non-attributable events are events that cannot be attributed to an individual user. The `inetd(1M)` utility and several other daemons record non-attributable events.

The `setac()` function rewinds the `audit_control` file to allow repeated searches.

The `endac()` function closes the `audit_control` file when processing is complete.

Files

`/etc/security/audit_control`	file containing default parameters read by the audit daemon, `auditd(1M)`

Return Values The `getacdir()`, `getacflg()`, `getacna()`, and `getacmin()` functions return:

`0` on success.

−2 on failure and set `errno` to indicate the error.

The `getacmin()` and `getacflg()` functions return:

1 on EOF.

The `getacdir()` function returns:

−1 on EOF.

2 if the directory search had to start from the beginning because one of the other functions was called between calls to `getacdir()`.

These functions return:

−3 if the directory entry format in the `audit_control` file is incorrect.

The `getacdir()`, `getacflg()`, and `getacna()` functions return:

−3 if the input buffer is too short to accommodate the record.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed (Obsolete)
MT-Level	Safe

See Also `audit_warn(1M)`, `bsmconv(1M)`, `inetd(1M)`, `audit_control(4)`, `attributes(5)`

Notes The functionality described on this manual page is available only if the Basic Security Module (BSM) has been enabled. See `bsmconv(1M)` for more information.

These functions are Obsolete and may be replaced with equivalent functionality in a future release.

Name getauclassent, getauclassnam, setauclass, endauclass, getauclassnam_r, getauclassent_r – get audit_class entry

Synopsis

```
cc [ flag... ] file... -lbsm  -lsocket   -lnsl   [ library... ]
#include <sys/param.h>
#include <bsm/libbsm.h>

struct au_class_ent *getauclassnam( const char *name);

struct au_class_ent *getauclassnam_r( au_class_ent_t *class_int,
     const char *name);

struct au_class_ent *getauclassent(void);

struct au_class_ent *getauclassent_r( au_class_ent_t *class_int);

void setauclass(void);

void endauclass(void);
```

Description The `getauclassent()` function and `getauclassnam()` each return an `audit_class` entry.

The `getauclassnam()` function searches for an `audit_class` entry with a given class name *name*.

The `getauclassent()` function enumerates audit_class entries. Successive calls to `getauclassent()` return either successive audit_class entries or `NULL`.

The `setauclass()` function "rewinds" to the beginning of the enumeration of audit_class entries. Calls to `getauclassnam()` may leave the enumeration in an indeterminate state, so `setauclass()` should be called before the first `getauclassent()`.

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

The `getauclassent_r()` and `getauclassnam_r()` functions both return a pointer to an audit_class entry as do their similarly named counterparts. They each take an additional argument, a pointer to pre-allocated space for an `au_class_ent_t`, which is returned if the call is successful. To assure there is enough space for the information returned, the applications programmer should be sure to allocate `AU_CLASS_NAME_MAX` and `AU_CLASS_DESC_MAX` bytes for the `ac_name` and `ac_desc` members of the `au_class_ent_t` data structure.

The internal representation of an `audit_user` entry is an `au_class_ent` structure defined in `<bsm/libbsm.h>` with the following members:

```
char        *ac_name;
au_class_t    ac_class;
char       *ac_desc;
```

Return Values The `getauclassnam()` and `getauclassnam_r()` functions return a pointer to a `au_class_ent` structure if they successfully locate the requested entry. Otherwise they return `NULL`.

The `getauclassent()` and `getauclassent_r()` functions return a pointer to a `au_class_ent` structure if they successfully enumerate an entry. Otherwise they return `NULL`, indicating the end of the enumeration.

Files `/etc/security/audit_class` file that aps audit class numbers to audit class names

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

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

All of the functions described on this man-page are MT-Safe except `getauclassent()` and `getauclassnam`, which are Unsafe. The `getauclassent_r()` and `getauclassnam_r()` functions have the same functionality as the Unsafe functions, but have a slightly different function call interface to make them MT-Safe.

See Also `bsmconv(1M)`, `audit_class(4)`, `audit_event(4)`, `attributes(5)`

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

The functionality described on this manual page is available only if the Basic Security Module (BSM) has been enabled. See `bsmconv(1M)` for more information.

Name getauditflags, getauditflagsbin, getauditflagschar – convert audit flag specifications

Synopsis

```
cc [ flag... ] file... -lbsm  -lsocket    -lnsl   [ library... ]
#include <sys/param.h>
#include <bsm/libbsm.h>

int getauditflagsbin(char *auditstring, au_mask_t *masks);

int getauditflagschar(char *auditstring, au_mask_t *masks, int verbose);
```

Description The `getauditflagsbin()` function converts the character representation of audit values pointed to by *auditstring* into `au_mask_t` fields pointed to by *masks*. These fields indicate which events are to be audited when they succeed and which are to be audited when they fail. The character string syntax is described in `audit_control(4)`.

The `getauditflagschar()` function converts the `au_mask_t` fields pointed to by *masks* into a string pointed to by *auditstring*. If *verbose* is 0, the short (2-character) flag names are used. If *verbose* is non-zero, the long flag names are used. The *auditstring* argument should be large enough to contain the ASCII representation of the events.

The *auditstring* argument contains a series of event names, each one identifying a single audit class, separated by commas. The `au_mask_t` fields pointed to by *masks* correspond to binary values defined in `<bsm/audit.h>`, which is read by `<bsm/libbsm.h>`.

Return Values Upon successful completion, `getauditflagsbin()` and `getauditflagschar()` return 0. Otherwise they return −1.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `bsmconv(1M)`, `audit.log(4)`, `audit_control(4)`, `attributes(5)`

Bugs This is not a very extensible interface.

Notes The functionality described on this manual page is available only if the Basic Security Module (BSM) has been enabled. See `bsmconv(1M)` for more information.

Name getauevent, getauevnam, getauevnum, getauevnonam, setauevent, endauevent, getauevent_r, getauevnam_r, getauevnum_r – get audit_event entry

Synopsis

```
cc [ flag... ] file... -lbsm  -lsocket   -lnsl   [ library... ]
#include <sys/param.h>
#include <bsm/libbsm.h>

struct au_event_ent *getauevent(void);

struct au_event_ent *getauevnam(char *name);

struct au_event_ent *getauevnum(au_event_t event_number);

au_event_t getauevnonam(char *event_name);

void setauevent(void);

void endauevent(void);

struct au_event_ent *getauevent_r(au_event_ent_t *e);

struct au_event_ent *getauevnam_r(au_event_ent_t *e, char *name);

struct au_event_ent *getauevnum_r(au_event_ent_t *e,
     au_event_t event_number);
```

Description These functions document the programming interface for obtaining entries from the `audit_event(4)` file. The `getauevent()`, `getauevnam()`, `getauevnum()`, `getauevent()`, `getauevnam()`, and `getauevnum()` functions each return a pointer to an `audit_event` structure.

The `getauevent()` and `getauevent_r()` functions enumerate `audit_event` entries. Successive calls to these functions return either successive `audit_event` entries or `NULL`.

The `getauevnam()` and `getauevnam_r()` functions search for an `audit_event` entry with *event_name*.

The `getauevnum()` and `getauevnum_r()` functions search for an `audit_event` entry with *event_number*.

The `getauevnonam()` function searches for an `audit_event` entry with *event_name* and returns the corresponding event number.

The `setauevent()` function "rewinds" to the beginning of the enumeration of `audit_event` entries. Calls to `getauevnam()`, `getauevnum()`, `getauevnonum()`, `getauevnam_r()`, or `getauevnum_r()` can leave the enumeration in an indeterminate state. The `setauevent()` function should be called before the first call to `getauevent()` or `getauevent_r()`.

The `endauevent()` function can be called to indicate that `audit_event` processing is complete. The system can then close any open `audit_event` file, deallocate storage, and so forth.

The `getauevent_r()`, `getauevnam_r()`, and `getauevnum_r()` functions each take an argument *e*, which is a pointer to an `au_event_ent_t`. This pointer is returned on a successful function call. To assure there is enough space for the information returned, the applications programmer should be sure to allocate `AU_EVENT_NAME_MAX` and `AU_EVENT_DESC_MAX` bytes for the `ae_name` and `ac_desc` elements of the `au_event_ent_t` data structure.

The internal representation of an `audit_event` entry is an `au_event_ent` structure defined in <`bsm/libbsm.h`> with the following members:

```
au_event_t      ae_number
char            *ae_name;
char            *ae_desc*;
au_class_t      ae_class;
```

Return Values The `getauevent()`, `getauevnam()`, `getauevnum()`, `getauevent_r()`, `getauevnam_r()`, and `getauevnum_r()` functions return a pointer to a `au_event_ent` structure if the requested entry is successfully located. Otherwise they return `NULL`.

The `getauevnonam()` function returns an event number of type `au_event_t` if it successfully enumerates an entry. Otherwise it returns `NULL`, indicating it could not find the requested event name.

Files

`/etc/security/audit_event`	file that maps audit event numbers to audit event names
`/etc/passwd`	file that stores user-ID to username mappings

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

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

The `getauevent()`, `getauevnam()`, and `getauevnum()` functions are Unsafe. The equivalent functions `getauevent_r()`, `getauevnam_r()`, and `getauevnum_r()` provide the same functionality and an MT-Safe function call interface.

See Also `bsmconv(1M)`, `getauclassent(3BSM)`, `getpwnam(3C)`, `audit_class(4)`, `audit_event(4)`, `passwd(4)`, `attributes(5)`

Notes All information for the `getauevent()`, `getauevnam()`, and `getauevnum()` functions is contained in a static area, so it must be copied if it is to be saved.

The functionality described on this manual page is available only if the Solaris Auditing has been enabled. See `bsmconv(1M)` for more information.

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, it 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.

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.

	/etc/security/policy.conf or	**Is user**
Authorization name	**user_attr or prof_attr entry**	**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` otherwise.

Usage The `getauthattr()` and `getauthnam()` functions both allocate memory for the pointers they return. This memory should be de-allocated 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 getauusernam, getauuserent, setauuser, endauuser, getauusernam_r, getauuserent_r – get audit_user entry

Synopsis

```
cc [ flag... ] file... -lbsm   -lsocket    -lnsl   [ library... ]
#include <sys/param.h>
#include <bsm/libbsm.h>

struct au_user_ent *getauusernam(const char *name);

struct au_user_ent *getauuserent(void);

void setauuser(void);

void endauuser(void);

struct au_user_ent *getauusernam_r(au_user_ent_t  *u, const char *name);

struct au_user_ent *getauuserent_r(au_user_ent_t *u);
```

Description The `getauuserent()`, `getauusernam()`, `getauuserent_r()`, and `getauusernam_r()` functions each return an `audit_user` entry. Entries can come from any of the sources specified in the `/etc/nsswitch.conf` file (see `nsswitch.conf(4)`).

The `getauusernam()` and `getauusernam_r()` functions search for an `audit_user` entry with a given login name *name*.

The `getauuserent()` and `getauuserent_r()` functions enumerate `audit_user` entries; successive calls to these functions will return either successive `audit_user` entries or `NULL`.

The `setauuser()` function "rewinds" to the beginning of the enumeration of `audit_user` entries. Calls to `getauusernam()` and `getauusernam_r()` may leave the enumeration in an indeterminate state, so `setauuser()` should be called before the first call to `getauuserent()` or `getauuserent_r()`.

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

The `getauuserent_r()` and `getauusernam_r()` functions both take as an argument a pointer to an `au_user_ent` that is returned on successful function calls.

The internal representation of an `audit_user` entry is an `au_user_ent` structure defined in `<bsm/libbsm.h>` with the following members:

```
char        *au_name;
au_mask_t   au_always;
au_mask_t   au_never;
```

Return Values The `getauusernam()` function returns a pointer to a `au_user_ent` structure if it successfully locates the requested entry. Otherwise it returns `NULL`.

The `getauuserent()` function returns a pointer to a `au_user_ent` structure if it successfully enumerates an entry. Otherwise it returns `NULL`, indicating the end of the enumeration.

Usage The functionality described on this manual page is available only if the Basic Security Module (BSM) has been enabled. See `bsmconv(1M)` for more information.

Files

`/etc/security/audit_user`	file that stores per-user audit event mask
`/etc/passwd`	file that stores user ID to username mappings

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

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

See Also `bsmconv(1M)`, `getpwnam(3C)`, `audit_user(4)`, `nsswitch.conf(4)`, `passwd(4)`, `attributes(5)`

Notes All information for the `getauuserent()` and `getauusernam()` functions is contained in a static area, so it must be copied if it is to be saved.

The `getauusernam()` and `getauuserent()` functions are Unsafe in multithreaded applications. The `getauusernam_r()` and `getauuserent_r()` functions provide the same functionality with interfaces that are MT-Safe.

These interfaces are Obsolete and may be removed from a future release.

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 getfauditflags – generate process audit state

Synopsis

```
cc [ flag... ] file... -lbsm   -lsocket   -lnsl  [ library... ]
#include <sys/param.h>
#include <bsm/libbsm.h>

int getfauditflags(au_mask_t *usremasks, au_mask_t *usrdmasks,
     au_mask_t *lastmasks);
```

Description The `getfauditflags()` function generates a process audit state by combining the audit masks passed as parameters with the system audit masks specified in the `audit_control(4)` file. The `getfauditflags()` function obtains the system audit value by calling `getacflg()` (see `getacinfo(3BSM)`).

The *usremasks* argument points to `au_mask_t` fields that contains two values. The first value defines which events are always to be audited when they succeed. The second value defines which events are always to be audited when they fail.

The *usrdmasks* argument points to `au_mask_t` fields that contains two values. The first value defines which events are never to be audited when they succeed. The second value defines which events are never to be audited when they fail.

The structures pointed to by *usremasks* and *usrdmasks* can be obtained from the `audit_user(4)` file by calling `getauusernam(3BSM)`, which returns a pointer to a strucure containing all `audit_user(4)` fields for a user.

The output of this function is stored in *lastmasks*, a pointer of type `au_mask_t` as well. The first value defines which events are to be audited when they succeed and the second defines which events are to be audited when they fail.

Both *usremasks* and *usrdmasks* override the values in the system audit values.

Return Values Upon successful completion, `getfauditflags()` returns 0. Otherwise it returns −1.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `bsmconv(1M)`, `getacinfo(3BSM)`, `getauditflags(3BSM)`, `getauusernam(3BSM)`, `audit.log(4)`, `audit_control(4)`, `audit_user(4)`, `attributes(5)`

Notes The functionality described on this manual page is available only if the Basic Security Module (BSM) has been enabled. See `bsmconv(1M)` for more information.

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
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 *Oracle 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 getprojent, getprojbyname, getprojbyid, getdefaultproj, inproj, getprojidbyname, setprojent, endprojent, fgetprojent – project database entry operations

Synopsis

```
cc [ flag... ] file... –lproject [ library... ]
#include <project.h>

struct project *getprojent(struct project *proj, void *buffer,
     size_t bufsize);

struct project *getprojbyname(const char *name,
     struct project *proj, void *buffer, size_t bufsize);

struct project *getprojbyid(projid_t projid,
     struct project *proj, void *buffer, size_t bufsize);

struct project *getdefaultproj(const char *username,
     struct project *proj, void *buffer, size_t bufsize);

int inproj(const char *username, const char *projname,
     void *buffer, size_t bufsize);

projid_t getprojidbyname(const char *name);

void setprojent(void);

void endprojent(void);

struct project *fgetprojent(FILE *f, struct project *proj,
     void *buffer, size_t bufsize);
```

Description These functions are used to obtain entries describing user projects. Entries can come from any of the sources for a project specified in the `/etc/nsswitch.conf` file (see `nsswitch.conf(4)`).

The `setprojent()`, `getprojent()`, and `endprojent()` functions are used to enumerate project entries from the database.

The `setprojent()` function effectively rewinds the project database to allow repeated searches. It sets (or resets) the enumeration to the beginning of the set of project entries. This function should be called before the first call to `getprojent()`.

The `getprojent()` function returns a pointer to a structure containing the broken-out fields of an entry in the project database. When first called, `getprojent()` returns a pointer to a project structure containing the first project structure in the project database. Successive calls can be used to read the entire database.

The `endprojent()` function closes the project database and deallocates resources when processing is complete. It is permissible, though possibly less efficient, for the process to call more project functions after calling `endprojent()`.

The `getprojbyname()` function searches the project database for an entry with the project name specified by the character string *name*.

The `getprojbyid()` function searches the project database for an entry with the (numeric) project ID specified by *projid*.

The `getdefaultproj()` function first looks up the project key word in the `user_attr` database used to define user attributes in restricted Solaris environments. If the database is available and the keyword is present, the function looks up the named project, returning `NULL` if it cannot be found or if the user is not a member of the named project. If absent, the function looks for a match in the project database for the special project `user`.*username*. If no match is found, or if the user is excluded from project `user`.*username*, the function looks at the default group entry of the `passwd` database for the user, and looks for a match in the project database for the special name `group`.*groupname*, where *groupname* is the default group associated with the password entry corresponding to the given *username*. If no match is found, or if the user is excluded from project `group`.*groupname*, the function returns `NULL`. A special project entry called 'default' can be looked up and used as a last resort, unless the user is excluded from project 'default'. On successful lookup, this function returns a pointer to the valid `project` structure. By convention, the user must have a default project defined on a system to be able to log on to that system.

The `inproj()` function checks if the user specified by *username* is able to use the project specified by *projname*. This function returns 1 if the user belongs to the list of project's users, if there is a project's group that contains the specified user, if project is a user's default project, or if project's user or group list contains "*" wildcard. In all other cases it returns 0.

The `getprojidbyname()` function searches the project database for an entry with the project name specified by the character string name. This function returns the project ID if the requested entry is found; otherwise it returns −1.

The `fgetprojent()` function, unlike the other functions described above, does not use `nsswitch.conf`; it reads and parses the next line from the stream *f*, which is assumed to have the format of the `project(4)` file. This function returns the same values as `getprojent()`.

The `getprojent()`, `getprojbyname()`, `getprojbyid()`, `getdefaultproj()`, and `inproj()` functions are reentrant interfaces for operations with the `project` database. These functions use buffers supplied by the caller to store returned results and are safe for use in both single-threaded and multithreaded applications.

Reentrant interfaces require the additional arguments *proj*, *buffer*, and *bufsize*. The *proj* argument must be a pointer to a `struct project` structure allocated by the caller. On successful completion, the function returns the project entry in this structure. Storage referenced by the `project` structure is allocated from the memory provided with the *buffer* argument, which is *bufsize* bytes in size. The content of the memory buffer could be lost in cases when these functions return errors.

For enumeration in multithreaded applications, the position within the enumeration is a process-wide property shared by all threads. The `setprojent()` function can be used in a multithreaded application but resets the enumeration position for all threads. If multiple

threads interleave calls to `getprojent()`, the threads will enumerate disjoint subsets of the project database. The `inproj()`, `getprojbyname()`, `getprojbyid()`, and `getdefaultproj()` functions leave the enumeration position in an indeterminate state.

Return Values Project entries are represented by the `struct project` structure defined in <`project.h`>.

```
struct project {
    char      *pj_name;     /* name of the project */
    projid_t  pj_projid;    /* numerical project id */
    char      *pj_comment;  /* project comment */
    char      **pj_users;   /* vector of pointers to
                               project user names */
    char      **pj_groups;  /* vector of pointers to
                               project group names */
    char      *pj_attr;     /* project attributes */
};
```

The `getprojbyname()` and `getprojbyid()` functions each return a pointer to a `struct project` if they successfully locate the requested entry; otherwise they return `NULL`.

The `getprojent()` function returns a pointer to a `struct project` if it successfully enumerates an entry; otherwise it returns `NULL`, indicating the end of the enumeration.

The `getprojidbyname()` function returns the project ID if the requsted entry is found; otherwise it returns –1 and sets errno to indicate the error.

When the pointer returned by the reentrant functions `getprojbyname()`, `getprojbyid()`, and `getprojent()` is non-null, it is always equal to the *proj* pointer that was supplied by the caller.

Upon failure, `NULL` is returned and errno is set to indicate the error.

Errors The `getprojent()`, `getprojbyname()`, `getprojbyid()`, `inproj()`, `getprojidbyname()`, `fgetprojent()`, and `getdefaultproj()` functions will fail if:

`EINTR` A signal was caught during the operation.

`EIO` An I/O error has occurred.

`EMFILE` There are `OPEN_MAX` file descriptors currently open in the calling process.

`ENFILE` The maximum allowable number of files is currently open in the system.

`ERANGE` Insufficient storage was supplied by *buffer* and *bufsize* to contain the data to be referenced by the resulting `project` structure.

These functions can also fail if the name service switch does not specify valid `project(4)` name service sources. In the case of an incompletely configurated name service switch configuration, `getprojbyid()` and other functions can return error values other than those documented above. These conditions usually occur when the `nsswitch.conf` file indicates that one or more name services is providing entries for the project database when that name service does not actually make a project table available.

Usage When compiling multithreaded applications, see Intro(3), Notes On Multithreaded Applications.

Use of the enumeration interface `getprojent()` is discouraged. Enumeration is supported for the project file, NIS, and LDAP but in general is not efficient. The semantics of enumeration are discussed further in nsswitch.conf(4).

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	See Description.

See Also Intro(3), libproject(3LIB), project_walk(3PROJECT), sysconf(3C), nsswitch.conf(4), project(4), attributes(5)

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 gmatch – shell global pattern matching

Synopsis

```
cc [ flag ... ] file ... -lgen [ library ... ]
#include <libgen.h>

int gmatch(const char *str, const char *pattern);
```

Description `gmatch()` checks whether the null-terminated string *str* matches the null-terminated pattern string *pattern*. See the sh(1), section `File Name Generation`, for a discussion of pattern matching. A backslash (\) is used as an escape character in pattern strings.

Return Values `gmatch()` returns non-zero if the pattern matches the string, zero if the pattern does not.

Examples **EXAMPLE 1** Examples of `gmatch()` function.

In the following example, `gmatch()` returns non-zero (true) for all strings with "`a`" or "`-`" as their last character.

```
char *s;
gmatch (s, "*[a\-]" )
```

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also sh(1), attributes(5)

Notes When compiling multithreaded applications, the `_REENTRANT` flag must be defined on the compile line. This flag should only be used in multithreaded applications.

Name HBA_GetAdapterAttributes – retrieve attributes about a specific HBA

Synopsis

```
cc [ flag... ] file... -lHBAAPI [ library... ]
#include <hbaapi.h>

HBA_STATUS HBA_GetAdapterAttributes(HBA_HANDLE handle,
     HBA_ADAPTERATTRIBUTES *hbaattributes);
```

Parameters

handle an open handle returned from HBA_OpenAdapter(3HBAAPI)

hbaattributes a pointer to an HBA_ADAPTERATTRIBUTES structure. Upon successful completion, this structure contains the specified adapter attributes.

Description The HBA_GetAdapterAttributes() function retrieves the adapter attributes structure for a given HBA. The caller is responsible for allocating *hbaattributes*.

Return Values Upon successful completion, HBA_STATUS_OK is returned. Otherwise, an error value is returned and the values in *hbaattributes* are undefined.

Errors See libhbaapi(3LIB) for general error status values.

Examples **EXAMPLE 1** Return adapter attributes.

The following example returns the adapter attributes into hbaAttrs for the given handle.

```
if ((status = HBA_GetAdapterAttributes(handle, &hbaAttrs)) !=
                  HBA_STATUS_OK) {
        fprintf(stderr, "Unable to get adapter attributes for "
               "HBA %d with name \"%s\".\n", hbaCount, adaptername);
        HBA_CloseAdapter(handle);
        continue;
}
```

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: FC-MI 1.92 (API version 1) Standard: FC-HBA Version 4 (API version 2)
MT-Level	Safe

See Also HBA_OpenAdapter(3HBAAPI), libhbaapi(3LIB), attributes(5)

T11 FC-MI Specification

Name HBA_GetAdapterName – retrieve the name of a specific HBA

Synopsis

```
cc [ flag... ] file... -lHBAAPI [ library... ]
#include <hbaapi.h>

HBA_STATUS HBA_GetAdapterName(HBA_UINT32 adapterindex,
     char *adaptername);
```

Parameters

adapterindex the index of the adapter, between 0 and one less than the value returned by HBA_GetNumberOfAdapters(3HBAAPI).

adaptername the buffer where the name of the adapter will be stored. The recommended size is 256 bytes.

Description The HBA_GetAdapterName() function stores the name of the adapter specified by *adapterindex* in the buffer pointed to by *adaptername*. The caller is responsible for allocating space for the name.

Return Values Upon successful completion, HBA_STATUS_OK is returned. Otherwise, an error value is returned and the content of *adaptername* is undefined.

Errors See libhbaapi(3LIB) for general error status values.

Examples **EXAMPLE 1** Return adapter name.

Given an *hbacount* >= 0 and < total number of adapters on the system, the following example returns the *adaptername* for that adapter.

```
if ((status = HBA_GetAdapterName(hbaCount, adaptername)) !=
            HBA_STATUS_OK) {
        fprintf(stderr, "HBA %d name not available for "
            "reason %d\n", hbaCount, status);
        continue;
}
```

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: FC-MI 1.92 (API version 1) Standard: FC-HBA Version 4 (API version 2)
MT-Level	Safe

See Also HBA_GetNumberOfAdapters(3HBAAPI), libhbaapi(3LIB), attributes(5)

T11 FC-MI Specification

Bugs The `HBA_GetAdapterName()` function does not take a name length argument to define how large the buffer is, yet the specification does not indicate a maximum name length. Failure to pass in a large enough buffer will result in a buffer over-run, which may lead to segmentation faults or other failures. Callers should be sure to allocate a large buffer to ensure the Vendor library will not overrun during the copy.

Name HBA_GetAdapterPortAttributes, HBA_GetDiscoveredPortAttributes, HBA_GetPortAttributesByWWN – retrieve Fibre Channel port attributes for a specific device

Synopsis

```
cc [ flag... ] file... -lHBAAPI [ library... ]
#include <hbaapi.h>

HBA_STATUS HBA_GetAdapterPortAttributes(HBA_HANDLE handle,
     HBA_UINT32 portindex, HBA_PORTATTRIBUTES *portattributes);

HBA_STATUS HBA_GetDiscoveredPortAttributes(HBA_HANDLE handle,
     HBA_UINT32 portindex, HBA_UINT32 discoveredportindex,
     HBA_PORTATTRIBUTES *portattributes);

HBA_STATUS HBA_GetPortAttributesByWWN(HBA_HANDLE handle,
     HBA_WWN PortWWN, HBA_PORTATTRIBUTES *portattributes);
```

Parameters

handle	an open handle returned from HBA_OpenAdapter(3HBAAPI)
portindex	the index of a specific port on the HBA as returned by a call to HBA_GetAdapterAttributes(3HBAAPI). The maximum value specified should be (HBA_ADAPTERATTRIBUTES.NumberOfPorts - 1).
portattributes	a pointer to an HBA_PORTATTRIBUTES structure. Upon successful completion, this structure contains the specified port attributes.
discoveredportindex	the index of a specific discovered port on the HBA as returned by HBA_GetAdapterPortAttributes(3HBAAPI). The maximum value specified should be (HBA_PORTATTRIBUTES.NumberOfDiscoveredPorts - 1).
PortWWN	the port WWN of the device for which port attributes are retrieved.

Description The HBA_GetAdapterPortAttributes() function retrieves Port Attributes for a specific port on the HBA.

The HBA_GetDiscoveredPortAttributes() function retrieves Port Attributes for a specific discovered device connected to the HBA.

The HBA_GetPortAttributesByWWN() function retrieves Port Attributes for a specific device based on the *PortWWN* argument.

Return Values Upon successful completion, HBA_STATUS_OK is returned. Otherwise, an error value is returned from the underlying VSL and the values in *hbaattributes* are undefined.

Errors See libhbaapi(3LIB) for general error status values.

Examples **EXAMPLE 1** Retrieve the port attributes for each port on the HBA.

The following example retrieves the port attributes for each port on the HBA.

EXAMPLE 1 Retrieve the port attributes for each port on the HBA. *(Continued)*

```
for (hbaPort = 0; hbaPort < hbaAttrs.NumberOfPorts; hbaPort++) {
    if ((status = HBA_GetAdapterPortAttributes(handle,
            hbaPort, &hbaPortAttrs)) != HBA_STATUS_OK) {
        fprintf(stderr, "Unable to get adapter port %d "
            "attributes for HBA %d with name \"%s\".\n",
            hbaPort, hbaCount, adaptername);
        HBA_CloseAdapter(handle);
        continue;
    }
    memcpy(&wwn, hbaPortAttrs.PortWWN.wwn, sizeof (wwn));
    printf(" Port %d: WWN=%016llx\n", hbaPort, wwn);

    /* ... */
}
```

EXAMPLE 2 Retrieve the discovered port target attributes for each discovered target port on the HBA.

The following example retrieves the discovered port target attributes for each discovered target port on the HBA.

```
for (discPort = 0;
        discPort < hbaPortAttrs.NumberofDiscoveredPorts;
        discPort++) {
    if ((status = HBA_GetDiscoveredPortAttributes(
            handle, hbaPort, discPort,
            &discPortAttrs)) != HBA_STATUS_OK) {
        fprintf(stderr, "Unable to get "
            "discovered port %d attributes for "
            "HBA %d with name \"%s\".\n",
            discPort, hbaCount, adaptername);
        continue;
    }
    memcpy(&wwn, discPortAttrs.PortWWN.wwn,
        sizeof (wwn));
    printf("  Discovered Port %d: WWN=%016llx\n",
        discPort, wwn);

    /* ... */
}
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: FC-MI 1.92 (API version 1)

ATTRIBUTE TYPE	ATTRIBUTE VALUE
	Standard: FC-HBA Version 4 (API version 2)
MT-Level	Safe

See Also HBA_GetAdapterPortAttributes(3HBAAPI), HBA_OpenAdapter(3HBAAPI), libhbaapi(3LIB), attributes(5)

T11 FC-MI Specification

Name HBA_GetBindingCapability, HBA_GetBindingSupport, HBA_SetBindingSupport – return and sets binding capabilities on an HBA port

Synopsis

```
cc [ flag... ] file... -lHBAAPI [ library... ]
#include <hbaapi.h>

HBA_HANDLE HBA_GetBindingCapability(HBA_HANDLE handle,
     HBA_WWN hbaPortWWN, HBA_BIND_CAPABILITY *pFlags);

HBA_STATUS HBA_GetBindingSupport(HBA_HANDLE handle, HBA_WWN
     hbaPortWWN, HBA_BIND_CAPABILITY *pFlags);

void HBA_SetBindingSupport(HBA_HANDLE handle, HBA_WWN hbaPortWWN,
     HBA_BIND_CAPABILITY Flags);
```

Parameters

handle	an open handle returned from HBA_OpenAdapter(3HBAAPI)
hbaPortWWN	the Port WWN of the local HBA through which the binding capabilities implemented by the HBA is returned
pFlags	a pointer to an HBA_BIND_CAPABILITY structure that returns the persistent binding capabilites implemented by the HBA
Flags	an HBA_BIND_CAPABILITY structure containing the persistent binding capabilites to enable for the HBA

Description The HBA_GetBindingCapability() function returns the binding capabilities implemented by the HBA.

The HBA_GetBindingSupport() function returns the currently enabled binding capabilities for the HBA.

The HBA_SetBindingSupport() function sets the currently enabled binding capabilites for the HBA to a subset of the binding capabilities implemented by the HBA.

Return Values The HBA_GetBindingCapability() and HBA_GetBindingSupport() functions return the following values:

HBA_STATUS_OK	Persistent binding capabilites have been returned.
HBA_STATUS_ERROR_ILLEGAL_WWN	Port WWN hbaPortWWN is not a WWN contained by the HBA referenced by *handle*.
HBA_STATUS_ERROR_NOT_SUPPORTED	The HBA handle specified by *handle* does not support persistent binding.
HBA_STATUS_ERROR	An error occurred. The value of *pFlags* remains unchanged and points to the persistent binding capabilites.

The HBA_SetBindingSupport() function returns:

`HBA_STATUS_OK`	Persistent binding capabilites have been enabled.
`HBA_STATUS_ERROR_ILLEGAL_WWN`	Port WWN hbaPortWWN is not a WWN contained by the HBA referenced by *handle*.
`HBA_STATUS_ERROR_NOT_SUPPORTED`	The HBA handle specified by *handle* does not support persistent binding.
`HBA_STATUS_ERROR_INCAPABLE`	The *flags* argument contains a capability not implemented by the HBA.
`HBA_STATUS_ERROR`	An error occurred.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: FC-MI 1.92 (API version 1) Standard: FC-HBA Version 4 (API version 2)
MT-Level	Safe

See Also `libhbaapi(3LIB)`, `attributes(5)`

T11 FC-MI Specification

Name HBA_GetEventBuffer – remove and return the next event from the HBA's event queue

Synopsis

```
cc [ flag... ] file... -lHBAAPI [ library... ]
#include <hbaapi.h>

HBA_STATUS HBA_GetEventBuffer(HBA_HANDLE handle,
     HBA_EVENTINFO *EventBuffer, HBA_UINT32 *EventBufferCount);
```

Parameters

handle	an open handle returned from `HBA_OpenAdapter(3HBAAPI)`
EventBuffer	a pointer to an `HBA_EVENTINFO` buffer
EventBufferCount	a pointer to the maximum number of events that can be stored in the `HBA_EVENTINFO` buffer. The value will be changed to the actual number of events placed in the buffer on completion.

Description The `HBA_GetEventBuffer()` function retrieves events from the HBA's event queue. The number of events returned is the lesser of *EventBufferCount* and the number of events on the queue. The returned events are removed from the queue.

Return Values Upon successful completion, `HBA_STATUS_OK` is returned. Otherwise, an error value is returned and the value of *EventBufferCount* is undefined.

Errors See `libhbaapi(3LIB)` for general error status values.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: FC-MI 1.92 (API version 1) Standard: FC-HBA Version 4 (API version 2)
MT-Level	Safe

See Also `HBA_OpenAdapter(3HBAAPI)`, `libhbaapi(3LIB)`, `attributes(5)`

T11 FC-MI Specification

Name HBA_GetFcpPersistentBinding, HBA_GetPersistentBindingV2, HBA_SetPersistentBindingV2, HBA_RemovePersistentBinding, HBA_RemoveAllPersistentBindings – handle persistent bindings between FCP-2 discovered devices and operating system SCSI information

Synopsis

```
cc [ flag... ] file... -lHBAAPI [ library... ]
#include <hbaapi.h>

HBA_STATUS HBA_GetFcpPersistentBinding(HBA_HANDLE handle,
     HBA_FCPBINDING *binding);

HBA_STATUS HBA_GetPersistentBindingV2(HBA_HANDLE handle,
     HBA_WWN hbaPortWWN, HBA_FCPBINDING2 *binding);

HBA_STATUS HBA_SetPersistentBindingV2(HBA_HANDLE handle,
     HBA_WWN hbaPortWWN, HBA_FCPBINDING2 *binding);

HBA_STATUS HBA_RemovePersistentBinding(HBA_HANDLE handle,
     HBA_WWN hbaPortWWN, HBA_FCPBINDING2 *binding);

HBA_STATUS HBA_RemoveAllPersistentBindings(HBA_HANDLE handle,
     HBA_WWN hbaPortWWN);
```

Parameters

handle an open handle returned from HBA_OpenAdapter(3HBAAPI)

binding

HBA_GetFcpPersistentBinding()
: a buffer to store the binding entries in. The *binding*->NumberOfEntries member must indicate the maximum number of entries that fit within the buffer. On completion, the *binding*->NumberOfEntries member will indicate the actual number of binding entries for the HBA. This value can be greater than the number of entries the buffer can store.

HBA_GetPersistentBindingV2()
: a pointer to a HBA_FCPBINDING2 structure. The NumberOfEntries member will be the maximum number of entries returned.

HBA_SetPersistentBindingV2()
: a pointer to a HBA_FCPBINDING2 structure. The NumberOfEntries member will be the number of bindings requested in the structure.

HBA_RemovePersistentBinding()
: a pointer to a HBA_FCPBINDING2 structure. The structure will contain all the bindings to be removed. The NumberOfEntries member will be the number of bindings being requested to be removed in the structure.

hbaPortWWN

`HBA_GetPersistentBindingV2()`	The Port WWN of the local HBA through which persistent bindings will be retrieved.
`HBA_SetPersistentBindingV2()`	The Port WWN of the local HBA through which persistent bindings will be set.
`HBA_RemovePersistentBinding()` `HBA_RemoveAllPersistentBindings()`	The Port WWN of the local HBA through which persistent bindings will be removed.

Description The `HBA_GetFcpPersistentBinding()` function retrieves the set of mappings between FCP LUNs and SCSI LUNs that are reestablished upon initialization or reboot. The means of establishing the persistent bindings is vendor-specific and accomplished outside the scope of the HBA API.

The `HBA_GetPersistentBindingV2()` function retrieves the set of persistent bindings between FCP LUNs and SCSI LUNs for the specified HBA Port that are reestablished upon initialization or reboot. The means of establishing the persistent bindings is vendor-specific and accomplished outside the scope of the HBA API. The binding information can contain bindings to Logical Unit Unique Device Identifiers.

The `HBA_SetPersistentBindingV2()` function sets additional persistent bindings between FCP LUNs and SCSI LUNs for the specified HBA Port. It can also accept bindings to Logical Unit Unique Device Identifiers. Bindings already set will remain set. An error occurs if a request is made to bind to an OS SCSI ID which has already been bound. Persistent bindings will not affect Target Mappings until the OS, HBA, and/or Fabric has been reinitialized. Before then, the effects are not specified.

The `HBA_RemovePersistentBinding()` function removes one or more persistent bindings. The persistent binding will only be removed if both the OS SCSI LUN and the SCSI Lun match a binding specifed in the arguments. Persistent bindings removed will not affect Target Mappings until the OS, HBA, and/or Fabric has been reinitialized. Before then, the effects are not specified.

The `HBA_RemoveAllPersistentBindings()` function removes all persistent bindings. Persistent bindings removed will not affect Target Mappings until the OS, HBA, and/or Fabric has been reinitialized. Before then, the effects are not specified.

Return Values The `HBA_GetFcpPersistentBinding()` function returns the following values:

`HBA_STATUS_OK`
The HBA was able to retrieve information.

`HBA_STATUS_ERROR_MORE_DATA`
A a larger buffer is required. The value of *binding*->`NumberOfEntries` after the call indicates the total number of entries available. The caller should reallocate a larger buffer to accomodate the indicated number of entries and reissue the routine.

`HBA_STATUS_ERROR_NOT_SUPPORTED`
The HBA handle specified by *handle* does not support persistent binding.

In the event that other error codes are returned, the value of *binding*->`NumberOfEntries` after the call should be checked, and if greater than the value before the call, a larger buffer should be allocated for a retry of the routine.

The `HBA_GetPersistentBindingV2()` function returns the following values:

`HBA_STATUS_OK`
The HBA was able to retrieve information.

`HBA_STATUS_ERROR_MORE_DATA`
A a larger buffer is required. The value of *binding*->`NumberOfEntries` after the call indicates the total number of entries available. The caller should reallocate a larger buffer to accomodate the indicated number of entries and reissue the routine.

`HBA_STATUS_ERROR_ILLEGAL_WWN`
The Port WWN *hbaPortWWN* is not a WWN contained by the HBA referenced by *handle*.

`HBA_STATUS_ERROR_NOT_SUPPORTED`
The HBA handle specified by *handle* does not support persistent binding.

The value of *binding* remains unchanged. The structure it points to contains binding information. The number of entries returned is the minimum between the number of entries specifed in the binding argument and the total number of bindings.

The `HBA_SetPersistentBindingV2()` function returns the following values.

`HBA_STATUS_OK`
The HBA was able to set bindings.

`HBA_STATUS_ERROR_ILLEGAL_WWN`
The Port WWN *hbaPortWWN* is not a WWN contained by the HBA referenced by *handle*.

`HBA_STATUS_ERROR_NOT_SUPPORTED`
The HBA handle specified by *handle* does not support persistent binding.

The value of *binding* remains unchanged. The success or failure of each Persistent binding set is indicated in the `Status` member of the `HBA_FCPBINDINGENTRY2` structure.

The `HBA_RemovePersistentBinding()` function returns the following values:

`HBA_STATUS_OK`
The HBA was able to retrieve information.

`HBA_STATUS_ERROR_ILLEGAL_WWN`
The Port WWN *hbaPortWWN* is not a WWN contained by the HBA referenced by *handle*.

`HBA_STATUS_ERROR_NOT_SUPPORTED`
The HBA handle specified by *handle* does not support persistent binding.

The value of *binding* remains unchanged. The success or failure of each Persistent binding set is indicated in the `Status` member of the `HBA_FCPBINDINGENTRY2` structure.

The `HBA_RemoveAllPersistentBindings()` function returns the following values:

`HBA_STATUS_OK`
The HBA was able to retrieve information.

`HBA_STATUS_ERROR_ILLEGAL_WWN`
The Port WWN *hbaPortWWN* is not a WWN contained by the HBA referenced by *handle*.

`HBA_STATUS_ERROR_NOT_SUPPORTED`
The HBA handle specified by *handle* does not support persistent binding.

Errors See `libhbaapi(3LIB)` for general error status values.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: FC-MI 1.92 (API version 1) Standard: FC-HBA Version 4 (API version 2)
MT-Level	Safe

See Also `HBA_GetFcpTargetMapping(3HBAAPI)`, `HBA_OpenAdapter(3HBAAPI)`, `libhbaapi(3LIB)`, `attributes(5)`

T11 FC-MI Specification

Bugs The `HBA_GetFcpTargetMapping(3HBAAPI)` and `HBA_GetFcpPersistentBinding()` functions do not take a *portindex* to define to which port of a multi-ported HBA the command should apply. The behavior on multi-ported HBAs is vendor-specific and could result in mappings or bindings for all ports being intermixed in the response buffer. SNIA version 2 defines a `HBA_GetFcpTargetMappingV2()` that takes a Port WWN as an argument. This fixes the bug with multi-ported HBAs in `HBA_GetFcpTargetMapping()`.

Name HBA_GetFcpTargetMapping, HBA_GetFcpTargetMappingV2 – retrieve mapping between FCP-2 discovered devices and operating system SCSI information

Synopsis

```
cc [ flag... ] file... -lHBAAPI [ library... ]
#include <hbaapi.h>

HBA_STATUS HBA_GetFcpTargetMapping(HBA_HANDLE handle,
     HBA_FCPTARGETMAPPING *mapping);

HBA_STATUS HBA_GetFcpTargetMappingV2(HBA_HANDLE handle,
     HBA_WWN hbaPortWWN, HBA_FCPTARGETMAPPINGV2 *mapping);
```

Parameters

handle an open handle returned from HBA_OpenAdapter(3HBAAPI)

mapping a buffer in which to store the mapping entries. The *mapping*->NumberOfEntries member must indicate the maximum number of entries that will fit within the buffer. On completion, the *mapping*->NumberOfEntries member indicates the actual number of mapping entries for the HBA. This value can be greater than the number of entries the buffer can store.

hbaPortWWN the Port Name of the local HBA Port for which the caller is requesting target mappings.

Description The HBA_GetFcpTargetMapping() function retrieves the current set of mappings between FCP LUNs and SCSI LUNs for a given HBA port.

The HBA_GetFcpTargetMappingV2() function retrieves the current set of mappings between FCP LUNs and SCSI LUNs for a given HBA. The mapping also includes a Logical Unit Unique Identifier for each logical unit.

Return Values The HBA_GetFcpTargetMappingV2() function returns the following values:

HBA_STATUS_ERROR_ILLEGAL_WWN
The port WWN specified by *hbaPortWWN* is not a valid port WWN on the specified HBA

HBA_STATUS_ERROR_NOT_SUPPORTED
Target mappings are not supported on the HBA.

HBA_STATUS_ERROR
An error occurred.

The HBA_GetFcpTargetMapping() and HBA_GetFcpTargetMappingV2() functions return the following values:

HBA_STATUS_OK
The HBA was able to retrieve information.

HBA_STATUS_ERROR_MORE_DATA
A larger buffer is required. The value of *mapping*->NumberOfEntries after the call indicates the total number of entries available. The caller should reallocate the buffer large

enough to accomodate the indicated number of entries and reissue the routine.

In the event that other error values are returned, the value of *mapping*->`NumberOfEntries` after the call should be checked, and if greater than the value before the call, a larger buffer should be allocated for a retry of the routine.

Errors See `libhbaapi(3LIB)` for general error status values.

Examples **EXAMPLE 1** Return target mapping data.

The following example returns target mapping data. It initially allocates space for one target mapping. If the number of entries returned is greater than the allocated space, a new buffer with sufficient space is allocated and `HBA_GetFcpTargetMapping()` is called again.

```
map = (HBA_FCPTARGETMAPPING *)calloc(1,
    sizeof (HBA_FCPTARGETMAPPING));
status = HBA_GetFcpTargetMapping(handle, map);
if (map->NumberOfEntries > 0) {
    HBA_UINT32  noe = map->NumberOfEntries;
    free(map);
    map = (HBA_FCPTARGETMAPPING *)calloc (1,
        sizeof (HBA_FCPTARGETMAPPING) +
        (sizeof (HBA_FCPSCSIENTRY)*(noe - 1)));
    map->NumberOfEntries = noe;
    if ((status = HBA_GetFcpTargetMapping(handle, map)) !=
            HBA_STATUS_OK) {
        fprintf(stderr, " Failed to get target "
            "mappings %d", status);
        free(map);
    } else {
        printf(" FCP Mapping entries: \n");
        for (cntr = 0;
            cntr < map->NumberOfEntries;
            cntr ++) {
        printf("  Path(%d): \"%s\"\n", cntr,
        map->entry[cntr].ScsiId.OSDeviceName);
        }
    }
}
```

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: FC-MI 1.92 (API version 1) Standard: FC-HBA Version 4 (API version 2)
MT-Level	Safe

See Also `HBA_OpenAdapter`(3HBAAPI), `libhbaapi`(3LIB), `attributes`(5)

T11 FC-MI Specification

Bugs The `HBA_GetFcpTargetMapping()` routine does not take a *portindex* to define which port of a multi-ported HBA the command should apply to. The behavior on multi-ported HBAs is vendor specific, and may result in mappings or bindings for all ports being intermixed in the response buffer. SNIA version 2 defines a `HBA_GetFcpTargetMappingV2()` which takes a Port WWN as an argument. This fixes the bug with multi-ported HBAs in `HBA_GetFcpTargetMapping()`.

Name HBA_GetNumberOfAdapters – report the number of HBAs known to the Common Library

Synopsis

```
cc [ flag... ] file... -lHBAAPI [ library... ]
#include <hbaapi.h>

HBA_UINT32 HBA_GetNumberOfAdapters(void);
```

Description The `HBA_GetNumberOfAdapters()` function report the number of HBAs known to the Common Library. This number is the sum of the number of HBAs reported by each VSL loaded by the Common Library.

Return Values The `HBA_GetNumberOfAdapters()` function returns the number of adapters known to the Common Library will be returned.

Examples **EXAMPLE 1** Using `HBA_GetNumberOfAdapters()`

```
numberOfAdapters = HBA_GetNumberOfAdapters();
for (hbaCount = 0; hbaCount < numberOfAdapters; hbaCount++) {
    /* ... */
}
```

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: FC-MI 1.92 (API version 1) Standard: FC-HBA Version 4 (API version 2)
MT-Level	Safe

See Also `libhbaapi(3LIB)`, `attributes(5)`

T11 FC-MI Specification

Name HBA_GetPortStatistics, HBA_GetFC4Statistics, HBA_GetFCPStatistics, HBA_ResetStatistics – Access Port statistics for a specific HBA port.

Synopsis

```
cc [ flag... ] file... -lHBAAPI [ library... ]
#include <hbaapi.h>

HBA_STATUS HBA_GetPortStatistics(HBA_HANDLE handle,
     HBA_UINT32 portindex, HBA_PORTSTATISTICS *portstatistics);

HBA_STATUS HBA_GetFC4Statistics(HBA_HANDLE handle, HBA_WWN portWWN,
     HBA_UINT8 FC4type, HBA_FC4STATISTICS * statistics);

HBA_STATUS HBA_GetFCPStatistics(HBA_HANDLE handle,
     const HBA_SCSIID * lunid, HBA_FC4STATISTICS * statistics);

void HBA_ResetStatistics(HBA_HANDLE handle, HBA_UINT32 portindex);
```

Parameters

handle	an open handle returned from HBA_OpenAdapter(3HBAAPI)
portindex	the index of a specific port on the HBA as returned by a call to HBA_GetAdapterAttributes(3HBAAPI). The maximum value specified should be (HBA_ADAPTERATTRIBUTES.NumberOfPorts - 1).
portstatistics	a pointer to an HBA_PORTSTATISTICS structure. Upon successful completion, this structure contains the specified port attributes.
portWWN	the Port WWN of the local HBA for which FC-4 statistics is being returned
FC4type	FC-4 protocol Data Structure Type as defined in FC-FS for which statistics are being requested
statistics	a pointer to an HBA_FC4STATISTICS structure where the specified statistics is being returned
lunid	a pointer to an HBA_SCSIID structure specifying the OS SCSI logical unit where statistics are being requested

Description The HBA_GetPortStatistics() function retrieves the statistical information from a given HBA port.

The HBA_GetFC4Statistics() function retrieves the traffic statistics for a specific FC-4 protocol.

The HBA_GetFCPStatistics() function retrieves the traffic statistics for a specific FC-4 protocol on the specified OS SCSI logical unit through that port.

The HBA_ResetStatistics() function sesets the statistical counters to zero for a given HBA port.

Return Values Upon successful completion, `HBA_GetPortStatistics()` returns `HBA_STATUS_OK`. Otherwise, an error value is returned from the underlying VSL and the values in *portstatistics* are undefined. If the VSL does not support a specific statistic, that statistic will have every bit set to 1.

Upon successful completion, `HBA_GetFC4Statistics()` and `HBA_GetFCPStatistics()` return `HBA_STATUS_OK`. Otherwise, an error value is returned from the underlying VSL and the values in *statistics* are undefined. If the VSL does not support a specific statistic, that statistic will have every bit set to 1.

Errors See `libhbaapi(3LIB)` for general error status values.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: FC-MI 1.92 (API version 1) Standard: FC-HBA Version 4 (API version 2)
MT-Level	Safe

See Also `HBA_GetAdapterAttributes(3HBAAPI)`, `HBA_OpenAdapter(3HBAAPI)`, `libhbaapi(3LIB)`, `attributes(5)`

T11 FC-MI Specification

Name HBA_GetVersion – determine the version of the API supported by the Common Library

Synopsis

```
cc [ flag... ] file... -lHBAAPI [ library... ]
#include <hbaapi.h>

HBA_UINT32 HBA_GetVersion(void);
```

Description The `HBA_GetVersion()` function returns the version of the API that the Common Library supports.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: FC-MI 1.92 (API version 1) Standard: FC-HBA Version 4 (API version 2)
MT-Level	Safe

See Also `libhbaapi(3LIB)`, `attributes(5)`

T11 FC-MI Specification

Name HBA_GetWrapperLibraryAttributes, HBA_GetVendorLibraryAttributes – return details about the implementation of the wrapper library and the vendor specific library

Synopsis

```
cc [ flag... ] file... -lHBAAPI [ library... ]
#include <hbaapi.h>

HBA_UINT32 HBA_GetWrapperLibraryAttributes(
     HBA_LIBRARYATTRIBUTES *attributes);

HBA_UINT32 HBA_GetVendorLibraryAttributes(HBA_UINT32 adapter_index,
     HBA_LIBRARYATTRIBUTES *attributes);
```

Parameters *attributes*

`HBA_GetWrapperLibraryAttributes()`	a pointer to a `HBA_LIBRARYATTRIBUTES` structure where the wrapper library information is returned
`HBA_GetVendorLibraryAttributes()`	a pointer to a `HBA_LIBRARYATTRIBUTES` structure where the vendor-specific library information is returned

adapter_index index of the HBA. The value ust be withing the range of 1 and the value returned by `HBA_GetNumberOfAdapters(3HBAAPI)`.

Description The `HBA_GetWrapperLibraryAttributes()` function returns details about the wrapper library.

The `HBA_GetVendorLibraryAttributes()` function returns details about the vendor specific library. The vendor-specific library selected is based on the *adapter_index*.

Return Values The `HBA_GetWrapperLibraryAttributes()` and `HBA_GetVendorLibraryAttributes()` functions return the version of the HBA API specification.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: FC-MI 1.92 (API version 1) Standard: FC-HBA Version 4 (API version 2)
MT-Level	Safe

See Also `HBA_GetNumberOfAdapters(3HBAAPI)`, `libhbaapi(3LIB)`, `attributes(5)`

T11 FC-MI Specification

Name HBA_LoadLibrary, HBA_FreeLibrary – load and free the resources used by the HBA Common Library

Synopsis

```
cc [ flag... ] file... -lHBAAPI [ library... ]
#include <hbaapi.h>

HBA_STATUS HBA_LoadLibrary(void);

HBA_STATUS HBA_FreeLibrary(void);
```

Description The `HBA_LoadLibrary()` function loads the Common Library, which in turn loads each VSL specified in the `hba.conf(4)` file.

The `HBA_FreeLibrary()` function releases resources held by the Common Library and each loaded VSL.

Return Values Upon successful completion, `HBA_LoadLibrary()` and `HBA_FreeLibrary()` return `HBA_STATUS_OK`. Otherwise, an error value is returned.

Errors See `libhbaapi(3LIB)` for general error status values.

Examples **EXAMPLE 1** Load the common library and each VSL.

The following example loads the common library and each VSL.

```
if ((status = HBA_LoadLibrary()) != HBA_STATUS_OK) {
    fprintf(stderr, "HBA_LoadLibrary failed: %d\\\
", status);
    return;
}
```

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: FC-MI 1.92 (API version 1) Standard: FC-HBA Version 4 (API version 2)
MT-Level	Safe

See Also `libhbaapi(3LIB)`, `hba.conf(4)`, `attributes(5)`

T11 FC-MI Specification

Name HBA_OpenAdapter, HBA_OpenAdapterByWWN, HBA_CloseAdapter – open and close a specific adapter

Synopsis

```
cc [ flag... ] file... -lHBAAPI [ library... ]
#include <hbaapi.h>

HBA_HANDLE HBA_OpenAdapter(char *adaptername);

HBA_STATUS HBA_OpenAdapterByWWN(HBA_HANDLE *handle, HBA_WWN wwn);

void HBA_CloseAdapter(HBA_HANDLE handle);
```

Parameters

adaptername the name of the adapter to open, as returned by HBA_GetAdapterName(3HBAAPI)

handle

HBA_OpenAdapterByWWN()	a pointer to an HBA_HANDLE
HBA_CloseAdapter()	the open handle of the adapter to close, as returned by HBA_OpenAdapter(3HBAAPI)

wwn the WWN to match the Node WWN or Port WWN of the HBA to open

Description The HBA_OpenAdapter() function opens the adapter specified by *adaptername* and returns a handle used for subsequent operations on the HBA.

The HBA_OpenAdapterByWWN() function opens a handle to the HBA whose Node or Port WWN matches the *wwn* argument.

The HBA_CloseAdapter() function closes the open handle.

Return Values Upon successful completion, HBA_OpenAdapter() returns a valid HBA_HANDLE with a numeric value greater than 0. Otherwise, 0 is returned.

The HBA_OpenAdapterByWWN() function returns the following values:

HBA_STATUS_OK	The *handle* argument contains a valid HBA handle.
HBA_STATUS_ERROR_ILLEGAL_WWN	The *wwn* argument is not a valid port WWN on the specified HBA.
HBA_STATUS_ERROR_AMBIGUOUS_WWN	The WWN is matched to multiple adapters.
HBA_STATUS_ERROR	An error occurred while opening the adapter.

Examples **EXAMPLE 1** Open an adapter.

The following example opens the specified adapter.

```
handle = HBA_OpenAdapter(adaptername);
if (handle == 0) {
    fprintf(stderr, "Unable to open HBA %d with name "
```

EXAMPLE 1 Open an adapter. *(Continued)*

```
        "\"%s\".\n", hbaCount, adaptername);
    continue;
}
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: FC-MI 1.92 (API version 1) Standard: FC-HBA Version 4 (API version 2)
MT-Level	Safe

See Also

`HBA_GetAdapterName(3HBAAPI)`, `HBA_OpenAdapter(3HBAAPI)`, `libhbaapi(3LIB)`, `attributes(5)`

T11 FC-MI Specification

Name HBA_RefreshInformation, HBA_RefreshAdapterConfiguration – refresh information for a specific HBA

Synopsis

```
cc [ flag... ] file... -lHBAAPI [ library... ]
#include <hbaapi.h>

void HBA_RefreshInformation(HBA_HANDLE handle);

void HBA_RefreshAdapterConfiguration(void);
```

Parameters *handle* an open handle returned from HBA_OpenAdapter(3HBAAPI)

Description The HBA_RefreshInformation() function requests that the underlying VSL reload all information about the given HBA. This function should be called whenever any function returns HBA_STATUS_ERROR_STALE_DATA, or if an index that was previously valid returns HBA_STATUS_ERROR_ILLEGAL_INDEX. Because the underlying VSL can reset all indexes relating to the HBA, all old index values must be discarded by the caller.

The HBA_RefreshAdapterConfiguration() function updates information about the HBAs present on the system. This function does not change any of the relationships between the HBA API and adapters that have not been reconfigured. HBA handles continue to refer to the same HBA even if it is no longer installed. The HBA name or index assigned by the library remains assigned to the same HBA even if it has been removed and reinstalled, as long as the bus position, WWN, and OS device have not changed. Adapter that have been removed and not replaced cannot have their HBA handles, HBA names, and HBA indexes reassigned. Calls to these adapters will generate HBA_STATUS_ERROR_UNAVAILABLE.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: FC-MI 1.92 (API version 1) Standard: FC-HBA Version 4 (API version 2)
MT-Level	Safe

See Also HBA_OpenAdapter(3HBAAPI), libhbaapi(3LIB), attributes(5)

T11 FC-MI Specification

Name HBA_RegisterForAdapterEvents, HBA_RegisterForAdapterAddEvents, HBA_RegisterForAdapterPortEvents, HBA_RegisterForAdapterPortStatEvents, HBA_RegisterForTargetEvents, HBA_RegisterForLinkEvents, HBA_RemoveCallback – SNIA event handling functions

Synopsis

```
cc [ flag... ] file... -lHBAAPI [ library... ]
#include <hbaapi.h>

HBA_STATUS HBA_RegisterForAdapterEvents(void (*pCallback)
     (void *pData, HBA_WWN PortWWN, HBA_UINT32 eventType),
     void *pUserData, HBA_HANDLE handle,
     HBA_CALLBACKHANDLE *pCallbackHandle);

HBA_STATUS HBA_RegisterForAdapterAddEvents(void (*pCallback)
     (void *pData, HBA_WWN PortWWN, HBA_UINT32 eventType),
     void *pUserData, HBA_CALLBACKHANDLE *pCallbackHandle);

HBA_STATUS HBA_RegisterForAdapterPortEvents(void (*pCallback)
     (void *pData, HBA_WWN PortWWN, HBA_UINT32 eventType,
     HBA_UINT32 fabricPortID), void *pUserData, HBA_HANDLE handle,
     HBA_WWN PortWWN, HBA_CALLBACKHANDLE *pCallbackHandle);

HBA_STATUS HBA_RegisterForAdapterPortStatEvents(void (*pCallback)
     (void *pData, HBA_WWN PortWWN, HBA_UINT32 eventType),
     void *pUserData, HBA_HANDLE handle, HBA_WWN PortWWN,
     HBA_PortStatistics stats, HBA_UINT32 statType,
     HBA_CALLBACKHANDLE *pCallbackHandle);

HBA_STATUS HBA_RegisterForTargetEvents(void (*pCallback)
     (void *pData, HBA_WWN hbaPortWWN, HBA_WWN discoveredPortWWN,
     HBA_UINT32 eventType), void * pUserData, HBA_HANDLE handle,
     HBA_WWN hbaPortWWN, HBA_WWN discoveredPortWWN,
     HBA_CALLBACKHANDLE *pCallbackHandle, HBA_UINT32 allTargets);

HBA_STATUS HBA_RegisterForLinkEvents(void (*pCallback)
     (void *pData, HBA_WWN adapterWWN, HBA_UINT32 eventType,
     void * pRLIRBuffer, HBA_UINT32 RLIRBufferSize),
     void *pUserData, void *PLIRBuffer, HBA_UINT32 RLIRBufferSize,
     HBA_HANDLE handle, HBA_CALLBACKHANDLE *pCallbackHandle);

HBA_STATUS HBA_RemoveCallback(HBA_CALLBACKHANDLE *pCallbackHandle);
```

Parameters *pCallback*

A pointer to the entry of the callback routine.

pData

the *pUserData* that is passed in from registration. This parameter can be used to correlate the event with the source of its event registration.

PortWWN

The Port WWN of the HBA for which the event is being reported.

hbaPortWWN
The Port WWN of the HBA for which the target event is being reported.

discoveredPortWWN
The Port WWN of the target for which the target event is being reported.

adapterWWN
The Port WWN of the of the HBA for which the link event is being reported.

eventType
a value indicating the type of event that has occured.

`HBA_RegisterForAdapterEvents()`	Possible values are `HBA_EVENT_ADAPTER_REMOVE` and `HBA_EVENT_ADAPTER_CHANGE`.
`HBA_RegisterForAdaterAddEvents()`	The only possible value is `HBA_EVENT_ADAPTER_ADD`.
`HBA_RegisterForAdaterPortEvents()`	Possible values are `HBA_EVENT_PORT_OFFLINE`, `HBA_EVENT_PORT_ONLINE`, `HBA_EVENT_PORT_NEW_TARGETS`, `HBA_EVENT_PORT_FABRIC`, and `HBA_EVENT_PORT_UNKNOWN`.
`HBA_RegisterForAdapterPortStatEvents()`	Possible values are `HBA_EVENT_PORT_STAT_THRESHOLD` and `HBA_EVENT_PORT_STAT_GROWTH`.
`HBA_RegisterForTargetEvents()`	If the value is `HBA_EVENT_LINK_INCIDENT`, RLIR has occured and information is in the RLIRBuffer. If the value is `HBA_EVENT_LINK_UNKNOWN`, a fabric link or topology change has occured and was not detected by RLIR. The RLIRBuffer is ignored
`HBA_RegisterForLinkEvents()`	Possible values are `HBA_EVENT_TARGET_OFFLINE`, `HBA_EVENT_TARGET_ONLINE`, `HBA_EVENT_TARGET_REMOVED`, and `HBA_EVENT_TARGET_UNKNOWN`.

fabricPortID
If the event is of type `HBA_EVENT_PORT_FABRIC`, this parameter will be the RSCN-affected Port ID page as defined in FC-FS. It is ignored for all other event types.

pRLIRBuffer
A pointer to a buffer where RLIR data may be passed to the callback function. The buffer will be overwritten for each fabric link callback function, but will not be overwritten within a single call to the callback function.

RLIRBufferSize
Size in bytes of the RLIRBuffer.

pUserData
a pointer passed with each event to the callback routine that can be used to correlate the event with the source of its event registration

pRLIRBuffer
A pointer to a buffer where RLIR data may be passed to the callback function. The buffer will be overwritten for each fabric link callback function, but will not be overwritten within a single call to the callback function.

RLIRBufferSize
Size in bytes of the RLIRBuffer.

handle
a handle to the HBA that event callbacks are being requested

PortWWN
The Port WWN of the HBA for which the event is being reported.

hbaPortWWN
The Port WWN of the HBA of which the event callbacks are being requested.

stats
an HBA_PortStatistics structure which indicates the counters to be monitored. If *statType* is `HBA_EVENT_PORT_STAT_THRESHOLD`, any non-null values are thresholds for which to watch. If *statType* is `HBA_EVET_PORT_STAT_GROWTH`, any non-null values are growth rate numbers over 1 minute.

statType
A value either `HBA_EVENT_PORT_STAT_TRHESHOLD` or `HBA_EVENT_PORT_STAT_GROWTH` used to determine whether counters registered are for threshold crossing or growth rate.

discoveredPortWWN
The Port WWN of the target that the event callbacks are being requested of.

pCallbackHandle
A pointer to structure in which an opaque identifier is returned that is used to deregister the callback. To deregister this event, call `HBA_RemoveCallback()` with this *pCallbackHandle* as an argument.

allTargets
If value is non-zero, *discoveredPortWWN* is ignored. Events for all discovered targets will be registered by this call. If value is zero, only events for *discoveredPortWWN* will be registered.

pcallbackHandle
A handle returned by the event registration function of the routine that is to be removed.

Description The `HBA_RegisterForAdapterEvents()` function registers an application-defined function that is called when an HBA category asynchronous event occurs. An HBA catetory event can have one of the following event types: `HBA_EVENT_ADAPTER_REMOVE` or `HBA_EVENT_ADAPTER_CHANGE`. If either of these events occur, the callback function is called, regardless of whether the HBA handle specified at registration is open. The `HBA_RemoveCallback()` function must be called to end event delivery.

The `HBA_RegisterForAdapterAddEvents()` function registers an application-defined function that is called whenever an HBA add category asynchronous event occurs. The callback function is called when a new HBA is added to the local system. The `HBA_RemoveCallback()` function must be called to end event delivery.

The `HBA_RegisterForAdapterPortEvents()` function registers an application-defined function that is called on the specified HBA whenever a port category asynchronous event occurs. A port catetory event can be one of the following event types: `HBA_EVENT_PORT_OFFLINE`, `HBA_EVENT_PORT_ONLINE`, `HBA_EVENT_PORT_NEW_TARGETS`, `HBA_EVENT_PORT_FABRIC`, or `HBA_EVENT_PORT_UNKNOWN`. The handle need not be open for callbacks to occur. The `HBA_RemoveCallback()` function must be called to end event delivery.

The `HBA_RegisterForAdapterPortStatEvents()` function defines conditions that would cause an HBA port statistics asynchronous event and registers an application-defined function that is called whenever one of these events occur. An HBA port statistics asynchronous event can be one of the following event types: `HBA_EVENT_PORT_STAT_THRESHOLD` or `HBA_EVENT_PORT_STAT_GROWTH`. More than one statistic can be registered with one call by setting multiple statistics in the *stats* argument. For threshold events, once a specific threshold has been crossed, the callback is automatically deregistered for that statistic. The handle need not be open for callbacks to occur. The `HBA_RemoveCallback()` function must be called to end event delivery.

The `HBA_RegisterForTargetEvents()` function registers an application-defined function that is called on the specified HBA whenever a target category asynchronous event occurs. A Target category event can be one of the following event types: `HBA_EVENT_TARGET_OFFLINE`, `HBA_EVENT_TARGET_ONLINE`, `HBA_EVENT_TARGET_REMOVED`, `HBA_EVENT_TARGET_UNKNOWN`. The handle need not be open for callbacks to occur. The `HBA_RemoveCallback()` function must be called to end event delivery.

The `HBA_RegisterForLinkEvents()` function registers an application defined function that is called on the specified HBA whenever a link category asynchronous event occurs. A link category event can be one of the following event types: `HBA_EVENT_LINK_INCIDENT` or `HBA_EVENT_LINK_UNKNOWN`. RLIR ELS is the only fabric link event type and the callback function is called whenever is it detected by the HBA. The handle need not be open for callbacks to occur. The `HBA_RemoveCallback()` function must be called to end event delivery.

The `HBA_RemoveCallback()` function removes the `HBA_CALLBACKHANDLE` instance of the callback routine.

Return Values Upon successful completion, `HBA_RegisterForAdapterEvents()`, `HBA_RegisterForAdapterAddEvents()`, `HBA_RegisterForAdapterPortEvents()`, `HBA_RegisterForAdapterPortStatEvents()`, `HBA_RegisterForTargetEvents()`, and `HBA_RegisterForLinkEvents()` return `HBA_STATUS_OK` and *pCallbackHandle* may be used to deregister the callback. Otherwise, an error value is returned and *pCallbackHandle* is not valid.

Upon successful completion, `HBA_RemoveCallback()` returns `HBA_STATUS_OK`. Otherwise, an error value is returned.

Errors See `libhbaapi(3LIB)` for general error status values.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: FC-MI 1.92 (API version 1) Standard: FC-HBA Version 4 (API version 2)
MT-Level	Safe

See Also `libhbaapi(3LIB)`, `attributes(5)`

T11 FC-MI Specification

Name HBA_SendCTPassThru, HBA_SendCTPassThruV2 – end a Fibre Channel Common Transport request to a Fabric

Synopsis

```
cc [ flag... ] file... -lHBAAPI [ library... ]
#include <hbaapi.h>

HBA_STATUS HBA_SendCTPassThru(HBA_HANDLE handle,
     void *pReqBuffer, HBA_UINT32 ReqBufferSize,
     void *pRspBuffer, HBA_UINT32 RspBufferSize);

HBA_STATUS HBA_SendCTPassThruV2(HBA_HANDLE handle,
     HBA_WWN hbaPortWWN, void *pReqBuffer,
     HBA_UINT32 ReqBufferSize, void *pRspBuffer,
     HBA_UINT32 *RspBufferSize);
```

Parameters

handle an open handle returned from HBA_OpenAdapter(3HBAAPI)

hbaPortWWN the Port Name of the local HBA Port through which the caller is issuing the CT request

pReqBuffer a pointer to a CT_IU request. The contents of the buffer must be in big-endian byte order

ReqBufferSize the length of the CT_IU request buffer *pReqBuffer*

pRspBuffer a pointer to a CT_IU response buffer. The response received from the fabric is copied into this buffer in big-endian byte order. Success of the function need not imply success of the command. The CT_IU Command/Response field should be checked for the Accept Response code.

RspBufferSize

HBA_SendCTPassThru()	the length of the CT_IU accept response buffer *pRspBuffer*.
HBA_SendCTPassThruV2()	a Pointer to the length of the CT_IU accept response buffer *pRspBuffer*.

Description The HBA_SendCTPassThru() and HBA_SendCTPassThruV2() functions provide access to the standard in-band fabric management interface. The *pReqBuffer* argument is interpreted as a CT_IU request, as defined by the T11 specification FC-GS-3, and is routed in the fabric based on the GS_TYPE field.

Return Values Upon successful transport and receipt of a CT_IU response, HBA_SendCTPassThru() returns HBA_STATUS_OK. The CT_IU payload indicates whether the command was accepted by the fabric based on the Command/Response code returned. Otherwise, an error value is returned from the underlying VSL and the values in *pRspBuffer* are undefined.

Upon successful transport and receipt of a CT_IU response, HBA_SendCTPassThruV2() returns HBA_STATUS_OK. The CT_IU payload indicates whether the command was accepted by

the fabric based on the Command/Response code returned. Otherwise, an error code is returned from the underlying VSL, and the values in *pRspBuffer* are undefined. The `HBA_SendCTPassThruV2()` function returns the following values:

`HBA_STATUS_ERROR_ILLEGAL_WWN`	The value of *hbaPortWWN* is not a valid port WWN on the specified HBA.
`HBA_STATUS_ERROR`	An error occurred.

Errors See `libhbaapi(3LIB)` for general error status values.

Examples **EXAMPLE 1** Data structures for the GIEL command.

```
struct ct_iu_preamble {
    uint32_t  ct_rev        : 8,
              ct_inid       : 24;
    uint32_t  ct_fcstype    : 8,
              ct_fcssubtype : 8,
              ct_options    : 8,
              ct_reserved1  : 8;
    uint32_t  ct_cmdrsp     : 16,
              ct_aiusize    : 16;
    uint32_t  ct_reserved2  : 8,
              ct_reason     : 8,
              ct_expln      : 8,
              ct_vendor     : 8;
};
struct gs_ms_ic_elem {
    uchar_t   elem_name[8];
    uint32_t  reserved1   : 24,
              elem_type   : 8;
};
struct gs_ms_giel_rsp {
    struct ct_iu_preamble  ct_header;
    uint32_t               num_elems;
struct gs_ms_ic_elem   elem_list[1];
};
#define MAX_PAYLOAD_LEN 65536 /* 64K */
```

EXAMPLE 2 Send an GIEL Management Service command through the given HBA handle.

The following example sends an GIEL Management Service command through the given HBA handle.

```
req.ct_rev        = 0x01;
req.ct_fcstype    = 0xFA;   /* Management Service */
req.ct_fcssubtype = 0x01;   /* Config server */
req.ct_cmdrsp     = 0x0101; /* GIEL command */
req.ct_aiusize    = MAX_PAYLOAD_LEN / 4 -
```

EXAMPLE 2 Send an GIEL Management Service command through the given HBA handle. *(Continued)*

```
         sizeof (struct ct_iu_preamble) / 4;
if ((status = HBA_SendCTPassThru(handle, &req, sizeof (req),
        rsp, MAX_PAYLOAD_LEN)) != HBA_STATUS_OK) {
    fprintf(stderr, "Unable to issue CT command on \\"%s\\""
        " for reason %d\
", adaptername, status);
} else {
    giel = (struct gs_ms_giel_rsp *)rsp;
    if (giel->ct_header.ct_cmdrsp != 0x8002) {
        fprintf(stderr, "CT command rejected on HBA "
            "\\"%s\\"\
", adaptername);
    } else {
        for (cntr = 0; cntr < giel->num_elems; cntr++) {
        memcpy(&wwn, giel->elem_list[cntr].elem_name, 8);
        printf(" Fabric element name: %016llx\
", wwn);
        }
    }
}
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: FC-MI 1.92 (API version 1) Standard: FC-HBA Version 4 (API version 2)
MT-Level	Safe

See Also

`HBA_OpenAdapter(3HBAAPI)`, `libhbaapi(3LIB)`, `attributes(5)`

T11 FC-MI Specification

Bugs

The `HBA_SendCTPassThru()` function does not take a *portindex* to define through which port of a multi-ported HBA to send the command. The behavior on multi-ported HBAs is vendor specific, and can result in the command always being sent on port 0 of the HBA. SNIA version 2 defines `HBA_SendCTPassThruV2()` which takes a Port WWN as an argument. This fixes the bug with multi-ported HBAs in `HBA_SendCTPassThru()`.

Name HBA_SendRLS, HBA_SendRPL, HBA_SendRPS, HBA_SendSRL, HBA_SendLIRR – issue an Extended Link Service through the local HBA Port

Synopsis

```
cc [ flag... ] file... -lHBAAPI [ library... ]
#include <hbaapi.h>

HBA_STATUS  HBA_SendRLS(HBA_HANDLE handle, HBA_WWN hbaPortWWN,
     HBA_WWN destWWN, void * pRspBuffer,
     HBA_UINT32 *pRspBufferSize);

HBA_STATUS HBA_SendRPL(HBA_HANDLE handle, HBA_WWN hbaPortWWN,
     HBA_WWN agentWWN, HBA_UINT32 agent_domain,
     HBA_UINT32 portIndex, void * pRspBuffer,
     HBA_UINT32 *pRspBufferSize);

HBA_STATUS HBA_SendRPS(HBA_HANDLE handle, HBA_WWN hbaPortWWN,
     HBA_WWN agentWWN, HBA_UINT32 agent_domain,
     HBA_WWN object_wwn, HBA_UINT32 object_port_number,
     void * pRspBuffer, HBA_UINT32 *pRspBufferSize);

HBA_STATUS HBA_SendSRL(HBA_HANDLE handle, HBA_WWN hbaPortWWN,
     HBA_WWN wwn, HBA_UINT32 domain,
     void * pRspBuffer, HBA_UINT32 *pRspBufferSize);

HBA_STATUS HBA_SendLIRR(HBA_HANDLE handle, HBA_WWN hbaPortWWN,
     HBA_WWN destWWN, HBA_UINT8 function, HBA_UINT8 type,
     void * pRspBuffer, HBA_UINT32 *pRspBufferSize);
```

Parameters *handle*

an open handle returned from HBA_OpenAdapter(3HBAAPI)

hbaPortWWN

HBA_SendRLS()	the Port WWN of the local HBA through which to send the RLS
HBA_SendRPL()	the Port WWN of the local HBA through which to send the RPL
HBA_SendRPS()	the Port WWN of the local HBA through which to send the RPS
HBA_SendSRL()	the Port WWN of the local HBA through which to send the SRL
HBA_SendLIRR()	the Port WWN of the local HBA through which to send the LIRR

destWWN

HBA_SendRLS()	the Port WWN of the remote Target to which the RLS is sent
HBA_SendLIRR()	he Port WWN of the remote Target to which the LIRR is sent

wwn

If non-zero, *wwn* is the port WWN to be scanned. If *wwn* is zero, it is ignored.

domain
If *wwn* is zero, *domain* is the domain number for which loops will be scanned. If *wwn* is non-zero, domain is ignored.

agent_wwn
If non-zero, *agent_wwn* is the port WWN for which the port list is requested. If *agent_wwn* is zero, it is ignored.

agent_domain
If *agent_wwn* is non-zero, *agent_domain* is the domain number and the domain controller for which the port list is requested. If *agent_wwn* is zero, it is ignored.

port_index
index of the first FC_Port returned in the response list

object_wwn
If non-zero, *object_wwn* is the port WWN for which the Port Status is requested. If *object_wwn* is zero, it is ignored.

object_port_number
If *object_wwn* is zero, *object_port_number* is the relative port number of the FC_Port for which the Port Status is requested. If *object_wwn* is non-zero, *object_port_number* is ignored.

function
the registration function to be performed

type
If *type* is non-zero, the type is the FC-4 device TYPE for which specific link incident information requested is requested. If *type* is zero, only common link incident information is requested.

pRspBuffer

`HBA_SendRLS()`	a pointer to a buffer into which the RLS response is copied
`HBA_SendRPL()`	a pointer to a buffer into which the RPL response is copied
`HBA_SendRPS()`	a pointer to a buffer into which the RPS response is copied
`HBA_SendSRL()`	a pointer to a buffer into which the SRL response is copied
`HBA_SendLIRR()`	A pointer to a buffer into which the LIRRresponse is copied.

RspBufferSize
a pointer to the size of the buffer

`HBA_SendRLS()` `HBA_SendLIRR()`	A size of 28 is sufficient for the largest response.
`HBA_SendRPS()`	A size of 58 is sufficient for the largest response.
`HBA_SendSRL()`	A size of 8 is sufficient for the largest response.

Description The `HBA_SendRLS()` function returns the Link Error Status Block associated with the agent WWN or agent-domain. For more information see "Read Link Status Block (RLS)" in FC-FS.

The `HBA_SendRPL()` function returns the Read Port List associated with the agent WWN or agent-domain. For more information see "Read Port List (RPL)" in FC-FS.

The `HBA_SendRPS()` function returns the Read Port Status Block associated with the agent WWN or agent-domain. For more information see "Read Port Status Block(RPS)" in FC-FS.

The `HBA_SendSRL()` function returns the Scan Remote Loop associated with the agent WWN or agent-domain. For more information see "Scan Remote Loop(SRL)" in FC-FS.

The `HBA_SendLIRR()` function returns the Link Incident Record Registration associated with the destportWWN. For more information see "Link Incident Record Registration (LIRR) in FC-FS.

Return Values These functions return the following values:

`HBA_STATUS_OK`	The LS_ACC for the ELS has been returned.
`HBA_STATUS_ERROR_ELS_REJECT`	The ELS has been rejeced by the local HBA Port.
`HBA_STATUS_ERROR_ILLEGAL_WWN`	The value of *hbaPortWWN* is not a valid port WWN on the specified HBA.
`HBA_STATUS_ERROR`	An error occurred.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: FC-MI 1.92 (API version 1) Standard: FC-HBA Version 4 (API version 2)
MT-Level	Safe

See Also `HBA_OpenAdapter(3HBAAPI)`, `libhbaapi(3LIB)`, `attributes(5)`

T11 FC-MI Specification

Name HBA_SendScsiInquiry, HBA_ScsiInquiryV2, HBA_SendReportLUNs, HBA_ScsiReportLUNsV2, HBA_SendReadCapacity, HBA_ScsiReadCapacityV2 – gather SCSI information from discovered ports

Synopsis

```
cc [ flag... ] file... -lHBAAPI [ library... ]
#include <hbaapi.h>

HBA_STATUS HBA_SendScsiInquiry(HBA_HANDLE handle, HBA_WWN PortWWN,
     HBA_UINT64 fcLUN, HBA_UINT8 EVPD, HBA_UINT32 PageCode,
     void *pRspBuffer, HBA_UINT32 RspBufferSize,
     void *pSenseBuffer, HBA_UINT32 SenseBufferSize);

HBA_STATUS HBA_ScsiInquiryV2(HBA_HANDLE handle, HBA_WWN hbaPortWWN,
     HBA_WWN discoveredPortWWN, HBA_UINT64 fcLUN, HBA_UINT8 CDB_BYTE1,
     HBA_UINT8 CDB_BYTE2, void *pRspBuffer, HBA_UINT32 *pRspBufferSize,
     HBA_UINT8 *pScsiStatus, void *pSenseBuffer,
     HBA_UINT32 *pSenseBufferSize);

HBA_STATUS HBA_SendReportLUNs(HBA_HANDLE handle, HBA_WWN PortWWN,
     void *pRspBuffer, HBA_UINT32 RspBufferSize,
     void *pSenseBuffer, HBA_UINT32 SenseBufferSize);

HBA_STATUS HBA_ScsiReportLUNsV2(HBA_HANDLE handle, HBA_WWN hbaPortWWN,
     HBA_WWN discoveredPortWWN, void *pRspBuffer,
     HBA_UINT32 *pRspBufferSize, HBA_UINT8 *pScsiStatus,
     void *pSenseBuffer, HBA_UINT32 *pSenseBufferSize);

HBA_STATUS HBA_SendReadCapacity(HBA_HANDLE handle, HBA_WWN PortWWN,
     HBA_UINT64 fcLUN, void *pRspBuffer, HBA_UINT32 RspBufferSize,
     void *pSenseBuffer, HBA_UINT32 SenseBufferSize);

HBA_STATUS HBA_ScsiReadCapacityV2(HBA_HANDLE handle
     HBA_WWN hbaPortWWN, HBA_WWN discoveredPortWWN,
     HBA_UINT64 fcLUN, void *pRspBuffer, HBA_UINT32 *pRspBufferSize,
     HBA_UINT8 *pScsiStatus, void *pSenseBuffer,
     HBA_UINT32 *pSenseBufferSize);
```

Parameters

handle
an open handle returned from HBA_OpenAdapter(3HBAAPI)

PortWWN
the port WWN of the discovered remote device to which the command is sent

hbaPortWWN

HBA_ScsiInquiryV2()
the Port WWN of the local HBA through which the SCSI INQUIRY command is issued

HBA_ScsiReportLUNsV2()
the Port WWN of the local HBA through which the SCSI REPORT LUNS command is issued

`HBA_ScsiReadCapacityV2()`
the Port WWN of a local HBA through which the `SCSI READ CAPACITY` command is issued

discoveredPortWWN

`HBA_ScsiInquiryV2()`	the Remote Port WWN to which the `SCSI INQUIRY` command is being sent
`HBA_ScsiReportLUNsV2()`	the Remote Port WWN to which the `SCSI REPORT LUNS` command is sent
`HBA_ScsiReadCapacityV2()`	the Remote Port WWN to which the `SCSI READ CAPACITY` command is sent

fcLUN
the FCP LUN as defined in the T10 specification SAM-2 to which the command is sent

EVPD
If set to 0, indicates a Standard Inquiry should be returned. If set to 1, indicates Vital Product Data should be returned.

PageCode
If *EVPD* is set to 1, *PageCode* indicates which Vital Product Data page should be returned.

CDB_Byte1
the second byte of the CDB for the `SCSI INQUIRY` command

CDB_Byte2
the third byte of the CDB for the `SCSI INQUIRY` command

pRspBuffer
a buffer in which to store the response payload

RspBufferSize
the size of the response buffer

pRspBufferSize
a pointer to the size of the response buffer

pScsiStatus
a buffer to receive SCSI sense data

pSenseBuffer
a buffer in which to store any SCSI sense data

SenseBufferSize
the size of the sense buffer

pSenseBufferSize
a pointer to the size of the sense buffer

Description The `HBA_SendScsiInquiry()` and `HBA_SendScsiInquiryV2()` functions send a SCSI Inquiry command as defined in the T10 specification SPC-2 to a remote FCP port.

The `HBA_SendReportLUNs()` and `HBA_SendReportLUNsV2()` functions send a SCSI Report LUNs command as defined in the T10 specification SPC-2 to a remote FCP port.

The `HBA_SendReadCapacity()` and `HBA_SendReadCapacityV2()` functions end a SCSI Read Capacity command as defined in the T10 specification SBC-2 to a remote FCP port.

Return Values The `HBA_SendScsiInquiry()` function returns the following value:

`HBA_STATUS_OK`
The command has completed. Success or failure should be determined by verifying that the sense data does not contain a check-condition. If a check-condition is present, the content of *pRspBuffer* is undefined.

The `HBA_ScsiInquiryV2()` function returns the following values:

`HBA_STATUS_OK`
The command has completed. The complete payload of the `SCSI INQUIRY` command is returned in *pRspBuffer*.

`HBA_STATUS_ERROR_ILLEGAL_WWN`
The port WWN *hbaPortWWN* is not a WWN contained by the HBA specified by *handle*.

`HBA_STATUS_ERROR_NOT_A_TARGET`
The identified remote Port does not have SCSI Target functionality.

`HBA_STATUS_ERROR_TARGET_BUSY`
The command cannot be sent due to a SCSI overlapped command condition.

`HBA_STATUS_ERROR`
An error occurred.

The `HBA_SendReportLUNs()` function returns the following values:

`HBA_STATUS_OK`
The command has completed. Success or failure should be determined by verifying the sense data does not contain a check-condition. If a check-condition is present, the content of *pRspBuffer* is undefined.

`HBA_STATUS_SCSI_CHECK_CONDITION`
The HBA detected a check-condition state. Details are present in the *pSenseBuffer* payload. The content of *pRspBuffer* is undefined. Not all VSLs support this error condition.

Other error values indicate the content of *pRspBuffer* is undefined. In some cases, the *pSenseBuffer* can contain sense data.

The `HBA_SendReportLUNsV2()` function returns the following values:

`HBA_STATUS_OK`
The command has completed. Sense data must be verified to ensure that it does not contain a check-condition to determine success. If a check-condition is present, the content of *pRspBuffer* is undefined.

`HBA_STATUS_ERROR_ILLEGAL_WWN`
The port WWN *hbaPortWWN* is not a WWN contained by the HBA specified by *handle*.

`HBA_STATUS_ERROR_NOT_A_TARGET`
The identified remote Port does not have SCSI Target functionality.

`HBA_STATUS_ERROR_TARGET_BUSY`
The command cannot be sent due to a SCSI overlapped command condition.

`HBA_STATUS_ERROR`
An error occurred.

The `HBA_SendReadCapacity()` function returns the following values:

`HBA_STATUS_OK`
The command has completed. Success or failure should be determined by verifying that the sense data does not contain a check-condition. If a check-condition is present, the content of *pRspBuffer* is undefined.

`HBA_STATUS_SCSI_CHECK_CONDITION`
The HBA detected a check-condition state. Details are present in the *pSenseBuffer* payload. The content of *pRspBuffer* is undefined. Not all VSLs support this error condition.

Other error values indicate the content of *pRspBuffer* is undefined. In some cases, the *pSenseBuffer* can contain sense data.

The `HBA_ScsiReadCapacityV2()` function returns the following values:

`HBA_STATUS_OK`
The command has completed. Sense data must be verified to ensure that it does not contain a check-condition to determine success. If a check-condition is present, the content of *pRspBuffer* is undefined.

`HBA_STATUS_ERROR_ILLEGAL_WWN`
The port WWN *hbaPortWWN* is not a WWN contained by the HBA specified by *handle*.

`HBA_STATUS_ERROR_NOT_A_TARGET`
The identified remote Port does not have SCSI Target functionality.

`HBA_STATUS_ERROR_TARGET_BUSY`
The command cannot be sent due to a SCSI overlapped command condition.

`HBA_STATUS_ERROR`
An error occurred.

Other error values indicate the content of *pRspBuffer* is undefined. In some cases, the *pSenseBuffer* can contain sense data.

Errors See `libhbaapi(3LIB)` for general error status values.

Examples **EXAMPLE 1** Send a SCSI inquiry to the given discovered Target port WWN.

The following example sends a SCSI inquiry to the given discovered Target port WWN.

```
memset(&inq, 0, sizeof (inq));
memset(&sense, 0, sizeof (sense));
if ((status = HBA_SendScsiInquiry(handle,
        discPortAttrs.PortWWN, 0, 0, 0, &inq,
        sizeof (inq), &sense, sizeof (sense))) !=
        HBA_STATUS_OK) {
    fprintf(stderr, "Unable to send SCSI "
        "inquiry, reason %d\n", status);
    continue;
}
printf("    Vendor: %.*s\n", 8, inq.inq_vid);
printf("    Model: %.*s\n", 16, inq.inq_pid);
```

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: FC-MI 1.92 (API version 1) Standard: FC-HBA Version 4 (API version 2)
MT-Level	Safe

See Also `HBA_OpenAdapter(3HBAAPI)`, `libhbaapi(3LIB)`, `attributes(5)`

T11 FC-MI Specification

Bugs The `HBA_SendScsiInquiry()`, `HBA_SendReportLUNs()`, and `HBA_SendReadCapacity()` functions do not take a *portindex* to define through which port of a multi-ported HBA the command should be sent. The behavior on multi-ported HBAs is vendor-specific and can result in the command being sent through the first HBA port, the first HBA port the given *PortWWN* is connected to, or other vendor-specific behavior. SNIA version 2 defines `HBA_ScsiInquiryV2()`, `HBA_ScsiReportLUNs()`, and `HBA_ScsiReadCapacity()` to take a Port WWN as an argument. This fixes the bug with multi-ported HBAs in `HBA_ScsiInquiry()`, `HBA_SendReportLUNs()`, and `HBA_SendReadCapacity()`.

Name HBA_SetRNIDMgmtInfo, HBA_GetRNIDMgmtInfo, HBA_SendRNID, HBA_SendRNIDV2 – access Fibre Channel Request Node Identification Data (RNID)

Synopsis

```
cc [ flag... ] file... -lHBAAPI [ library... ]
#include <hbaapi.h>

HBA_STATUS HBA_SetRNIDMgmtInfo(HBA_HANDLE handle,
     HBA_MGMTINFO *pInfo);

HBA_STATUS HBA_GetRNIDMgmtInfo(HBA_HANDLE handle,
     HBA_MGMTINFO *pInfo);

HBA_STATUS HBA_SendRNID(HBA_HANDLE handle, HBA_WWN wwn,
     HBA_WWNTYPE wwntype, void *pRspBuffer,
     HBA_UINT32 *RspBufferSize);

HBA_STATUS HBA_SendRNIDV2(HBA_HANDLE handle, HBA_WWN hbaPortWWN,
     HBA_WWN destWWN, HBA_UINT32 destFCID,
     HBA_UINT32 NodeIdDataFormat, void *pRspBuffer,
     HBA_UINT32 *RspBufferSize);
```

Parameters

handle
an open handle returned from HBA_OpenAdapter(3HBAAPI)

pInfo

HBA_SetRNIDMgmtInfo()
a pointer to a HBA_MGMTINFO structure containing the new RNID

HBA_GetRNIDMgmtInfo()
a pointer to a HBA_MGMTINFO structure into which the RNID is copied

wwn
the discovered port WWN to which the request is sent

wwntype
deprecated

hbaPortWWN
the Port WWN of the local HBA through which to send the ELS

destWWN
the Port WWN of the remote Target to which the ELS is sent

destFCID
If *destFCID* is non-zero, *destFCID* is the address identifier of the remote target to which the ELS is sent. If *destFCID* is 0, *destFCID* is ignored.

NodeIdDataFormat
the Node Identification Data Fromat value as defined in FC-FS

pRspBuffer
A pointer to a buffer into which the RNID response is copied . The data will be in Big Endian format.

RspBufferSize
A pointer to the size of the buffer. On completion it will contain the size of the actual response payload copied into the buffer.

Description These functions access Fibre Channel Request Node Identification Data (RNID) as defined in the T11 specification FC-FS.

The `HBA_SetRNIDMgmtInfo()` function sets the RNID returned from by HBA.

The `HBA_GetRNIDMgmtInfo()` function retrieves the stored RNID from the HBA.

The `HBA_SendRNID()` function sends an RNID request to a discovered port. The Node Identification Data format is always set to 0xDF for General Topology Discovery Format as defined in the T11 specification FC-FS.

The `HBA_SendRNIDV2()` function sends an RNID request to a discovered port requesting a specified Node Identification Data format.

Return Values Upon successful completion, `HBA_SetRNIDMgmtInfo()` returns `HBA_STATUS_OK` and sets the RNID.

Upon successful completion, `HBA_GetRNIDMgmtInfo()` returns `HBA_STATUS_OK`. Otherwise, an error value is returned and the content of *pInfo* is undefined.

Upon successful completion, `HBA_SendRNID()` returns `HBA_STATUS_OK`. Otherwise, an error value is returned and the content of *pRspBuffer* is undefined.

The `HBA_SendRNIDV2()` returns the following values:

`HBA_STATUS_OK`
The RNID ELS has been successfully returned.

`HBA_STATUS_ERROR_ELS_REJECT`
The RNID ELS was rejected by the HBA Port.

`HBA_STATUS_ERROR_ILLEGAL_WWN`
The value of *hbaPortWWN* is not a valid port WWN on the specified HBA.

`HBA_STATUS_ERROR_ILLEGAL_FCID`
The *destWWN*/*destFCID* pair conflicts with a discovered Port Name/address identifier pair known by the HBA.

`HBA_STATUS_ERROR_ILLEGAL_FCID`
The N_Port WWN in the RNID response does not match *destWWN*.

`HBA_STATUS_ERROR`
An error occurred.

Errors See `attributes(5)` for general error status values.

Attributes See `libhbaapi(3LIB)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: FC-MI 1.92 (API version 1) Standard: FC-HBA Version 4 (API version 2)
MT-Level	Safe

See Also `HBA_OpenAdapter(3HBAAPI)`, `libhbaapi(3LIB)`, `attributes(5)`

T11 FC-MI Specification

Bugs The `HBA_SetRNIDMgmtInfo()` and `HBA_GetRNIDMgmtInfo()` functions do not take a *portindex* to define to which port of a multi-ported HBA the command should apply. The behavior on multi-ported HBAs is vendor-specific and can result in all ports being set to the same value.

The `HBA_SetRNIDMgmtInfo()` and `HBA_GetRNIDMgmtInfo()` functions allow only 0xDF (General Topology Discovery Format).

The `HBA_SendRNID()` function does not take a *portindex* to define through which port of a multi-ported HBA to send the command. The behavior on multi-ported HBAs is vendor-specific and can result in the command being sent through the first port.

The `HBA_SendRNID()` function does not take an argument to specify the Node Identification Data Format. It always assumes that 0xDF (General Topology Discovery Format) is desired. SNIA version 2 defines `HBA_SendRNIDV2()` to take a Port WWN and a Node Data Format. This fixes the bugs with multi-ported HBAs of allowing only0xDF (General Topology Discovery Format) in `HBA_SendRNID()`.

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 hypot, hypotf, hypotl – Euclidean distance function

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double hypot(double x, double y);

float hypotf(float x, float y);

long double hypotl(long double x, long double y);
```

Description These functions compute the length of the square root of $x^2 + y^2$ without undue overflow or underflow.

Return Values Upon successful completion, these functions return the length of the hypotenuse of a right angled triangle with sides of length x^2 and y^2.

If the correct value would cause overflow, a range error occurs and `hypot()`, `hypotf()`, and `hypotl()` return the value of the macro `HUGE_VAL`, `HUGE_VALF`, and `HUGE_VALL`, respectively.

If *x* or *y* is ±Inf, +Inf is returned even if one of *x* or *y* is NaN.

If *x* or *y* is NaN and the other is not ±Inf, a NaN is returned.

Errors These functions will fail if:

Range Error — The result overflows.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, the overflow floating-point exception is raised.

Usage `hypot`(*x*,*y*), `hypot`(*y*,*x*), and `hypot`(*x*, −*y*) are equivalent.

`hypot`(*x*, ±0) is equivalent to `fabs`(*x*).

These functions takes precautions against underflow and overflow during intermediate steps of the computation.

An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. An application should either examine the return value or check the floating point exception flags to detect exceptions.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also fabs(3M), feclearexcept(3M), fetestexcept(3M), isnan(3M), math.h(3HEAD), sqrt(3M), attributes(5), standards(5)

Name idn_decodename, idn_decodename2, idn_enable, idn_encodename, idn_nameinit – IDN (Internationalized Domain Name) conversion functions

Synopsis

```
cc [ flag... ] file... -lidnkit [ library... ]
#include <idn/api.h>

idn_result_t idn_decodename(int actions, const char *from, char *to,
     size_t tolen);

idn_result_t idn_decodename2(int actions, const char *from, char *to,
     size_t tolen, const char *auxencoding);

idn_result_t idn_encodename(int actions, const char *from, char *to,
     size_t tolen);

void idn_enable(int on_off);

idn_result_t idn_nameinit(int load_file);
```

Description The `idn_nameinit()` function initializes the library. It also sets default configuration if load_file is 0, otherwise it tries to read a configuration file. If `idn_nameinit()` is called more than once, the library initialization will take place only at the first call while the actual configuration procedure will occur at every call.

If there are no errors, `idn_nameinit()` returns `idn_success`. Otherwise, the returned value indicates the cause of the error. See the section RETURN VALUES below for the error codes.

It is usually not necessary to call this function explicitly because it is implicitly called when `idn_encodename()`, `idn_decodename()`, or `idn_decodename2()` is first called without prior calling of `idn_nameinit()`.

The `idn_encodename()` function performs name preparation and encoding conversion on the internationalized domain name specified by *from*, and stores the result to *to*, whose length is specified by *tolen*. The *actions* argument is a bitwise-OR of the following macros, specifying which subprocesses in the encoding process are to be employed.

`IDN_LOCALCONV`	Local encoding to UTF-8 conversion
`IDN_DELIMMAP`	Delimiter mapping
`IDN_LOCALMAP`	Local mapping
`IDN_NAMEPREP`	`NAMEPREP` mapping, normalization, prohibited character check, and bidirectional string check
`IDN_UNASCHECK`	`NAMEPREP` unassigned codepoint check
`IDN_ASCCHECK`	ASCII range character check
`IDN_IDNCONV`	UTF-8 to IDN encoding conversion
`IDN_LENCHECK`	Label length check

Details of this encoding process can be found in the section Name Encoding

For convenience, also IDN_ENCODE_QUERY, IDN_ENCODE_APP, and IDN_ENCODE_STORED macros are provided. IDN_ENCODE_QUERY is used to encode a "query string" (see the IDNA specification). It is equal to:

```
(IDN_LOCALCONV | IDN_DELIMMAP | IDN_LOCALMAP | IDN_NAMEPREP |
        IDN_IDNCONV | IDN_LENCHECK)
```

IDN_ENCODE_APP is used for ordinary application to encode a domain name. It performs IDN_ASCCHECK in addition with IDN_ENCODE_QUERY. IDN_ENCODE_STORED is used to encode a "stored string" (see the IDNA specification). It performs IDN_ENCODE_APP plus IDN_UNASCHECK.

The idn_decodename() function performs the reverse of idn_encodename(). It converts the internationalized domain name given by *from*, which is represented in a special encoding called ACE (ASCII Compatible Encoding), to the application's local codeset and stores in *to*, whose length is specified by *tolen*. As in idn_encodename(), *actions* is a bitwise-OR of the following macros.

IDN_DELIMMAP	Delimiter mapping
IDN_NAMEPREP	NAMEPREP mapping, normalization, prohibited character check and bidirectional string check
IDN_UNASCHECK	NAMEPREP unassigned codepoint check
IDN_IDNCONV	UTF-8 to IDN encoding conversion
IDN_RTCHECK	Round trip check
IDN_ASCCHECK	ASCII range character check
IDN_LOCALCONV	Local encoding to UTF-8 conversion

Details of this decoding process can be found in the section Name Decoding.

For convenience, IDN_DECODE_QUERY, IDN_DECODE_APP, and IDN_DECODE_STORED macros are also provided. IDN_DECODE_QUERY is used to decode a "query string" (see the IDNA specification). It is equal to

```
(IDN_DELIMMAP | IDN_NAMEPREP | IDN_IDNCONV | IDN_RTCHECK | IDN_LOCALCONV)
```

IDN_DECODE_APP is used for ordinary application to decode a domain name. It performs IDN_ASCCHECK in addition to IDN_DECODE_QUERY. IDN_DECODE_STORED is used to decode a "stored string" (see the IDNA specification). It performs IDN_DECODE_APP plus IDN_UNASCHECK.

The `idn_decodename2()` function provides the same functionality as `idn_decodename()` except that character encoding of *from* is supposed to be auxencoding. If IDN encoding is Punycode and auxencoding is ISO8859-2, for example, it is assumed that the Punycode string stored in *from* is written in ISO8859-2.

In the IDN decode procedure, `IDN_NAMEPREP` is done before `IDN_IDNCONV`, and some non-ASCII characters are converted to ASCII characters as the result of `IDN_NAMEPREP`. Therefore, ACE string specified by *from* might contains those non-ASCII characters. That is the reason `docode_name2()` exists.

All of thsee functions return an error value of type `idn_result_t`. All values other than `idn_success` indicates some kind of failure.

Name Encoding

Name encoding is a process that transforms the specified internationalized domain name to a certain string suitable for name resolution. For each label in a given domain name, the encoding processor performs:

1. Convert to UTF-8 (`IDN_LOCALCONV`)

   Convert the encoding of the given domain name from application's local encoding (for example, ISO8859-1) to UTF-8.

2. Delimiter mapping (`IDN_DELIMMAP`)

   Map domain name delimiters to '.' (U+002E). The reco- ginzed delimiters are: U+3002 (ideographic full stop), U+FF0E (fullwidth full stop), U+FF61 (halfwidth ideographic full stop).

3. Local mapping (`IDN_LOCALMAP`)

   Apply character mapping whose rule is determined by the top-level domain name.

4. `NAMEPREP` (`IDN_NAMEPREP`, `IDN_UNASCHECK`)

   Perform name preparation (`NAMEPREP`), which is a standard process for name canonicalizaion of internationalized domain names.

   `NAMEPREP` consists of 5 steps: mapping, normalization, prohibited character check, bidirectional text check, and unassigned codepoint check. The first four steps are done by `IDN_NAMEPREP`, and the last step is done by `IDN_UNASCHECK`.

5. ASCII range character check (`IDN_ASCCHECK`)

   Checks if the domain name contains non-LDH ASCII characters (not letter, digit, or hyphen characters), or it begins or end with hyphen.

6. Convert to ACE (`IDN_IDNCONV`)

   Convert the `NAMEPREP`ed name to a special encoding designed for representing internationalized domain names.

The encoding is known as ACE (ASCII Compatible Encoding) since a string in the encoding is just like a traditional ASCII domain name consisting of only letters, digits and hyphens.

7. Label length check (`IDN_LENCHECK`)

   For each label, check the number of characters in it. It must be in the range of 1 to 63.

Name Decoding

Name decoding is a reverse process of the name encoding. It transforms the specified internationalized domain name in a special encoding suitable for name resolution to the normal name string in the application's current codeset. However, name encoding and name decoding are not symmetric.

For each label in a given domain name, the decoding processor performs:

1. Delimiter mapping (`IDN_DELIMMAP`)

   Map domain name delimiters to '.' (U+002E). The recoginzed delimiters are: U+3002 (ideographic full stop), U+FF0E (fullwidth full stop), U+FF61 (halfwidth ideographic full stop).

2. `NAMEPREP` (`IDN_NAMEPREP`, `IDN_UNASCHECK`)

   Perform name preparation (`NAMEPREP`), which is a standard process for name canonicalizaion of internationalized domain names.

3. Convert to UTF-8 (`IDN_IDNCONV`)

   Convert the encoding of the given domain name from ACE to UTF-8.

4. Round trip check (`IDN_RTCHECK`)

   Encode the result of (3) using the Name Encoding scheme, and then compare it with the result of the step (2). If they are different, the check is failed. If `IDN_UNASCHECK`, `IDN_ASCCHECK` or both are specified, they are also done in the encoding processes.

5. Convert to local encoding

   Convert the result of (3) from UTF-8 to the application's local encoding (for example, ISO8859-1).

If prohibited character check, unassigned codepoint check or bidirectional text check at step (2) failed, or if round trip check at step (4) failed, the original input label is returned.

Disabling IDN

If your application should always disable internationalized domain name support for some reason, call

```
(void) idn_enable(0);
```

before performing encoding/decoding. Afterward, you can enable the support by calling

```
(void) idn_enable(1);
```

Return Values These functions return values of type `idn_result_t` to indicate the status of the call. The following is a complete list of the status codes.

`idn_success`	Not an error. The call succeeded.
`idn_notfound`	Specified information does not exist.
`idn_invalid_encoding`	The encoding of the specified string is invalid.
`idn_invalid_syntax`	There is a syntax error in internal configuration file(s).
`idn_invalid_name`	The specified name is not valid.
`idn_invalid_message`	The specified message is not valid.
`idn_invalid_action`	The specified action contains invalid flags.
`idn_invalid_codepoint`	The specified Unicode code point value is not valid.
`idn_invalid_length`	The number of characters in an ACE label is not in the range of 1 to 63.
`idn_buffer_overflow`	The specified buffer is too small to hold the result.
`idn_noentry`	The specified key does not exist in the hash table.
`idn_nomemory`	Memory allocation using malloc failed.
`idn_nofile`	The specified file could not be opened.
`idn_nomapping`	Some characters do not have the mapping to the target character set.
`idn_context_required`	Context information is required.
`idn_prohibited`	The specified string contains some prohibited characters.
`idn_failure`	Generic error which is not covered by the above codes.

Examples **EXAMPLE 1** Get the address of an internationalized domain name.

To get the address of an internationalized domain name in the application's local codeset, use `idn_encodename()` to convert the name to the format suitable for passing to resolver functions.

```
#include <idn/api.h>
#include <sys/socket.h>
#include <netdb.h>

...

idn_result_t r;
char ace_name[256];
struct hostent *hp;
```

EXAMPLE 1 Get the address of an internationalized domain name. *(Continued)*

```
int error_num;

...

r = idn_encodename(IDN_ENCODE_APP, name, ace_name,
                   sizeof(ace_name));
if (r != idn_success) {
    fprintf(stderr, gettext("idn_encodename failed.\n"));
    exit(1);
}

hp = getipnodebyname(ace_name, AF_INET6, AI_DEFAULT, &error_num);

...
```

EXAMPLE 2 Decode the internationalized domain name.

To decode the internationalized domain name returned from a resolver function, use `idn_decodename()`.

```
#include <idn/api.h>
#include <sys/socket.h>
#include <netdb.h>

...

idn_result_t r;
char local_name[256];
struct hostent *hp;
int error_num;

...

hp = getipnodebyname(name, AF_INET, AI_DEFAULT, &error_num);
if (hp != (struct hostent *)NULL) {
     r = idn_decodename(IDN_DECODE_APP, hp->h_name, local_name,
                   sizeof(local_name));
     if (r != idn_success) {
          fprintf(stderr, gettext("idn_decodename failed.\n"));
          exit(1);
     }
     printf(gettext("name: %s\n"), local_name);
}

...
```

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWidnl, SUNWidnd
CSI	Enabled
Interface Stability	Evolving
MT-Level	Unsafe

See Also Intro(3), libidnkit(3LIB), setlocale(3C), hosts(4), attributes(5), environ(5)

RFC 3490 Internationalizing Domain Names in Applications (IDNA)

RFC 3491 Nameprep: A Stringprep Profile for Internationalized Domain Names (IDN)

RFC 3492 Punycode: A Bootstring encoding of Unicode for Internationalized Domain Names in Applications (IDNA)

RFC 3454 Preparation of Internationalized Strings ("stringprep")

RFC 952 DoD Internet Host Table Specification

RFC 921 Domain Name System Implementation Schedule - Revised

STD 3, RFC 1122 Requirements for Internet Hosts -- Communication Layers

STD 3, RFC 1123 Requirements for Internet Hosts -- Applications and Support

Unicode Standard Annex #15: Unicode Normalization Forms, Version 3.2.0.
http://www.unicode.org

International Language Environments Guide (for this version of Solaris)

Copyright And License

2. Redistribution of source code must retain the copyright notices as they appear in each source code file, this License Terms and Conditions.
3. Redistribution in binary form must reproduce the Copyright Notice, this License Terms and Conditions, in the documentation and/or other materials provided with the distribution. For the purposes of binary distribution the "Copyright Notice" refers to the following language: "Copyright (c) 2000-2002 Japan Network Information Center. All rights reserved."
4. The name of JPNIC may not be used to endorse or promote products derived from this Software without specific prior written approval of JPNIC.
5. Disclaimer/Limitation of Liability: THIS SOFTWARE IS PROVIDED BY JPNIC "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL JPNIC BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Notes The `idn_nameinit()` function checks internal system configuration files such as `/etc/idn/idn.conf` and `/etc/idn/idnalias.conf` if they are in the proper access mode and owership. If they are not in the proper access mode or ownership, the function will not read and use the configurations defined in the files but use default values. In this case the function will also issue a warning message such as:

```
idn_nameinit: warning: config file (/etc/idn/idn.conf) not in proper
                       access mode or ownership - the file ignored.
```

The proper access mode and the ownership are described in the package prototype file of SUNWidnl. It is also recommended not to change the system configuration files.

Name IFDHCloseChannel – close the communication channel with an IFD

Synopsis

```
#include <smartcard/ifdhandler.h>

RESPONSECODE IFDHCloseChannel(DWORD Lun);
```

Parameters The `IFDHCloseChannel()` function takes the following parameters:

Input *Lun* Logical Unit Number

Description The `IFDHCloseChannel()` function closes the communication channel for the Interface Device (IFD) specified by *Lun*. If a smart card is present in the slot, it must be powered down. All internal resources (such as file descriptors) associated with this IFD can be freed with this function.

Return Values The following values are returned:

`IFD_SUCCESS`	Successful completion.
`IFD_COMMUNICATION_ERROR`	An error has occurred.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

See Also `IFDHCreateChannelByName(3SMARTCARD)`, `attributes(5)`

Notes This manual page is derived from the MUSCLE PC/SC IFD Driver Developer Kit documentation. License terms and attribution and copyright information for this interface are available at the default location `/var/sadm/pkg/SUNWocfh/install/copyright`. If the Solaris Operating Environment has been installed anywhere other than the default location, modify the path to access the file at the installed location.

Name IFDHControl – send control information to an IFD

Synopsis

```
#include <smartcard/ifdhandler.h>

RESPONSECODE IFDHControl(DWORD Lun, PUCHAR TxBuffer,
     DWORD TxLength, PUCHAR RxBuffer,
     PDWORD RxLength);
```

Parameters The `IFDHControl()` function takes the following parameters:

Input	*Lun*	Logical Unit Number
	TxBuffer	Control bytes to send
	TxLength	Length of bytes to send
	RxLength	Expected length of response
Output	*RxBuffer*	Buffer to receive response
	RxLength	Length of response received

Description The `IFDHControl()` performs control information exchange with some types of readers such as PIN pads, biometrics, and LCD panels according to the MCT and CTBCS specification. This function does not exchange data with the card.

Return Values The following values are returned:

`IFD_SUCCESS`	Successful completion.
`IFD_RESPONSE_TIMEOUT`	The response has timed out.
`IFD_COMMUNICATION_ERROR`	An error has occurred.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

See Also `IFDHCreateChannelByName(3SMARTCARD)`, `attributes(5)`

Notes This manual page is derived from the MUSCLE PC/SC IFD Driver Developer Kit documentation. License terms and attribution and copyright information for this interface are available at the default location `/var/sadm/pkg/SUNWocfh/install/copyright`. If the Solaris Operating Environment has been installed anywhere other than the default location, modify the path to access the file at the installed location.

Name IFDHCreateChannel – create a communication channel with an IFD

Synopsis

```
#include <smartcard/ifdhandler.h>

RESPONSECODE IFDHCreateChannel(DWORD Lun, DWORD Channel);
```

Parameters The `IFDHCreateChannel()` function takes the following parameters:

Input

Lun Logical Unit Number

Channel Channel ID

Description The `IFDHCreateChannel()` function is similar to `IFDHCreateChannelByName(3SMARTCARD)`. It takes *Channel* (a number) as an argument instead of the device name string. The *Channel* argument is typically passed from configuration information by the smart card framework server/daemon (the caller) to the IFD handler. The IFD handler can use this *Channel* appropriately to create a communication channel to the card terminal.

Return Values The following values are returned:

`IFD_SUCCESS` Successful completion.

`IFD_COMMUNICATION_ERROR` An error has occurred.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

See Also `IFDHCreateChannelByName(3SMARTCARD)`, `attributes(5)`

Notes This manual page is derived from the MUSCLE PC/SC IFD Driver Developer Kit documentation. License terms and attribution and copyright information for this interface are available at the default location `/var/sadm/pkg/SUNWocfh/install/copyright`. If the Solaris Operating Environment has been installed anywhere other than the default location, modify the path to access the file at the installed location.

Name IFDHCreateChannelByName – create a communication channel with an IFD

Synopsis

```
#include <smartcard/ifdhandler.h>

RESPONSECODE IFDHCreateChannelByName(DWORD Lun, LPSTR devicename);
```

Parameters The `IFDHCreateChannelByName()` function takes the following parameters:

Input

Lun Logical Unit Number

devicename Device name path

Description The `IFDHCreateChannelByName()` function opens a communication channel with a card terminal specified by *devicename*. This function can use `open(2)` or other system call to open the device and establish a communication path. The caller of this function (smart card framework) assigns a logical unit number *Lun* per card terminal and slot and passes this value to `IFDHCreateChannelByName()`. If the IFD handler supports multiple terminals with one instance of the handler (as indicated by capability `TAG_IFD_SIMULTANEOUS_ACCESS`), it communicates with the card terminal corresponding to this *Lun*.

If the IFD handler supports only one terminal with one slot per instance, it can choose to ignore the *Lun*.

The Logical Unit Number, *Lun* is encoded as `0xXXXXYYYY`, where

`YYYY` represents the lower 16 bits that correspond to the slot number for terminals with multiple slots. Most of the readers have only one slot, in which case `YYYY` is 0.

`XXXX` represents the next 16 bits that correspond to the card terminal and can range between 0 and a number returned by `TAG_IFD_SUMULTANEOUS_ACCESS`.

Return Values The following values are returned:

`IFD_SUCCESS` Successful completion.

`IFD_COMMUNICATION_ERROR` An error occurred.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

See Also `attributes(5)`

Notes This manual page is derived from the MUSCLE PC/SC IFD Driver Developer Kit documentation. License terms and attribution and copyright information for this interface are available at the default location `/var/sadm/pkg/SUNWocfh/install/copyright`. If the Solaris Operating Environment has been installed anywhere other than the default location, modify the path to access the file at the installed location.

Name IFDHGetCapabilities – get IFD capabilities

Synopsis

```
#include <smartcard/ifdhandler.h>

RESPONSECODE IFDHGetCapabilities(DWORD Lun, DWORD Tag,
     PDWORD Length, PUCHAR Value);
```

Parameters The `IFDHGetCapabilities()` function takes the following parameters:

Input *Lun* Logical Unit Number

Tag Tag of the desired data value

Length Maximum length of the desired data value

Output *Length* Length of the data returned

Value Value of the desired data

Description The `IFDHGetCapabilities()` function retrieves the terminal or card capabilities for the terminal or card specified by *Lun*.

The *Tag* parameter can have one of the following values:

`TAG_IFD_ATR`	Return the ATR (Answer To Reset). This is the default value.
`TAG_IFD_SIMULTANEOUS_ACCESS`	Return the number of sessions the driver can handle. This value is used for multiple terminals sharing the same IFD handler.
`TAG_IFD_SLOTS_NUMBER`	Return the number of slots in this terminal.

If the `TAG_IFD_SIMULTANEOUS_ACCESS` and `TAG_IFD_SLOTS_NUMBER` tags are not supported, the error value `IFD_ERROR_TAG` must be returned.

Return Values The following values are returned:

`IFD_SUCCESS`	Successful completion.
`IFD_ERROR_TAG`	An error occurred.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

See Also `IFDHCreateChannelByName(3SMARTCARD)`, `attributes(5)`

Notes This manual page is derived from the MUSCLE PC/SC IFD Driver Developer Kit documentation. License terms and attribution and copyright information for this interface are available at the default location `/var/sadm/pkg/SUNWocfh/install/copyright`. If the Solaris Operating Environment has been installed anywhere other than the default location, modify the path to access the file at the installed location.

Name IFDHICCPresence – check for the presence of a smart card

Synopsis

```
#include <smartcard/ifdhandler.h>

RESPONSECODE IFDHICCPresence(DWORD Lun);
```

Parameters The `IFDHICCPresence()` function takes the following parameter:

Input *Lun* Logical Unit Number

Description The `IFDHICCPresence()` function checks for the presence of an ICC (smart card) in the reader or slot specified by *Lun*.

Return Values The following values are returned:

`IFD_ICC_PRESENT`	ICC is present.
`IFD_ICC_NOT_PRESENT`	ICC is not present.
`IFD_COMMUNICATION_ERROR`	An error occurred.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

See Also `IFDHCreateChannelByName(3SMARTCARD)`, `attributes(5)`

Notes This manual page is derived from the MUSCLE PC/SC IFD Driver Developer Kit documentation. License terms and attribution and copyright information for this interface are available at the default location `/var/sadm/pkg/SUNWocfh/install/copyright`. If the Solaris Operating Environment has been installed anywhere other than the default location, modify the path to access the file at the installed location.

Name IFDHPowerICC – power up or power down the smart card

Synopsis

```
#include <smartcard/ifdhandler.h>

RESPONSECODE IFDHPowerICC(DWORD Lun, DWORD Action, PUCHAR Atr, PDWORD AtrLength);
```

Parameters The `IFDHPowerICC()` takes the following parameters:

Input	*Lun*	Logical Unit Number
	Action	Action to be taken
	AtrLength	Maximum length of the ATR
Output	*Atr*	Answer to Reset (ATR) value of the inserted card
	AtrLength	Actual length of the ATR

Description The `IFDHPowerICC()` function controls the power and reset signals of the ICC (smart card) at the reader or slot specified by *Lun*. The *Action* parameter can take one of the following values:

`IFD_POWER_UP`	Power and reset the card. Return the ATR and its length.
`IFD_POWER_DOWN`	Power down the card. The *Atr* and *AtrLength* parameters are set to 0.
`IFD_RESET`	Perform a quick reset on the card. Return the ATR and its length.

The IFD handler caches the ATR during a power up or reset and returns the ATR and its length in a call to `IFDHGetCapabilities(3SMARTCARD)`.

Memory cards without an ATR return `IFD_SUCCESS` on power up or reset but *Atr* and *AtrLength* are set to 0.

Return Values The following values are returned:

`IFD_SUCCESS`	Successful completion.
`IFD_ERROR_POWER_ACTION`	An error occurred while powering up or resetting the card.
`IFD_NOT_SUPPORTED`	The action specified by *Action* is not supported.
`IFD_COMMUNICATION_ERROR`	An error occurred.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

See Also `IFDHCreateChannelByName(3SMARTCARD)`, `IFDHGetCapabilities(3SMARTCARD)`, `attributes(5)`

Notes This manual page is derived from the MUSCLE PC/SC IFD Driver Developer Kit documentation. License terms and attribution and copyright information for this interface are available at the default location `/var/sadm/pkg/SUNWocfh/install/copyright`. If the Solaris Operating Environment has been installed anywhere other than the default location, modify the path to access the file at the installed location.

Name IFDHSetCapabilities – set slot or card capabilities

Synopsis

```
#include <smartcard/ifdhandler.h>

RESPONSECODE IFDHSetCapabilities(DWORD Lun, DWORD Tag,
     DWORD Length, PUCHAR Value);
```

Parameters The `IFDHSetCapabilities()` function takes the following parameters:

Input

Lun	Logical Unit Number
Tag	Tag of the desired data value
Length	Maximum length of the desired data value
Value	Value of the desired data

Description The `IFDHSetCapabilities()` function sets the slot or card capabilities for the slot or card specified by *Lun*.

Return Values The following values are returned:

`IFD_SUCCESS`	Successful completion.
`IFD_ERROR_TAG`	The tag is invalid.
`IFD_ERROR_SET_FAILURE`	The value of the data could not be set.
`IFD_COMMUNICATION_ERROR`	An error occurred.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

See Also `IFDHCreateChannelByName(3SMARTCARD)`, `IFDHGetCapabilities(3SMARTCARD)`, `attributes(5)`

Notes This manual page is derived from the MUSCLE PC/SC IFD Driver Developer Kit documentation. License terms and attribution and copyright information for this interface are available at the default location `/var/sadm/pkg/SUNWocfh/install/copyright`. If the Solaris Operating Environment has been installed anywhere other than the default location, modify the path to access the file at the installed location.

Name IFDHSetProtocolParameters – set protocol parameters

Synopsis

```
#include <smartcard/ifdhandler.h>

RESPONSECODE IFDHSetProtocolParameters(DWORD Lun, DWORD Protocol,
     UCHAR Flags,UCHAR PTS1, UCHAR PTS2,
     UCHAR PTS3);
```

Parameters The `IFDHSetProtocolParameters()` function takes the following parameters:

Input

Lun	Logical Unit Number
Protocol	Desired protocol
Flags	The bitwise-inclusive OR of the flags
PTS1	1st PTS Value
PTS2	2nd PTS Value
PTS3	3rd PTS Value

Description The `IFDHSetProtocolParameters()` function sets the Protocol Type Selection (PTS) of the slot or card using the three PTS values as defined in ISO 7816.

The *Protocol* parameter can take an integer value between 0 and 14, inclusive, corresponding to the protocol T=0, T=1, ..., T=14.

The *Flags* parameter can have the value of one of the following or the bitwise-inclusive OR of two or more of the following:

`IFD_NEGOTIATE_PTS1`	Use the *PTS1* value.
`IFD_NEGOTIATE_PTS2`	Use the *PTS2* value.
`IFD_NEGOTIATE_PTS3`	Use the *PTS3* value.

Return Values The following values are returned:

`IFD_SUCCESS`	Successful completion.
`IFD_COMMUNICATION_ERROR`	An error occurred.
`IFD_ERROR_PTS_FAILURE`	The PTS value could not be set.
`IFD_PROTOCOL_NOT_SUPPORTED`	The protocol is not supported.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

See Also `IFDHCreateChannelByName(3SMARTCARD)`, `attributes(5)`

Notes This manual page is derived from the MUSCLE PC/SC IFD Driver Developer Kit documentation. License terms and attribution and copyright information for this interface is available at the default location `/var/sadm/pkg/SUNWocfh/install/copyright`. If the Solaris Operating Environment has been installed anywhere other than the default location, modify the path to access the file at the installed location.

Name IFDHTransmitToICC – transmit APDU to a smart card

Synopsis

```
#include <smartcard/ifdhandler.h>

RESPONSECODE IFDHTransmitToICC(DWORD Lun, SCARD_IO_HEADER SendPci,
     PUCHAR TxBuffer, DWORD TxLength, PUCHAR RxBuffer,
     PDWORD RxLength, PSCARD_IO_HEADER RecvPci);
```

Parameters The `IFDHTransmitToICC()` function takes the following parameters:

Input	*Lun*	Logical Unit Number
	SendPci	Send-Protocol structure
	TxBuffer	Buffer containing the APDU to be sent
	TxLength	Length of sent APDU
Output	*RxBuffer*	Received buffer for response APDU
	RxLength	Length of APDU-response
	RecvPci	Receive-Protocol structure

Description The `IFDHTransmitToICC()` function performs an Application Protocol Data Unit (APDU) exchange with the card or slot specified by *Lun*. The IFD handler is responsible for performing any protocol-specific (such as T0/T1) APDU exchanges with the card.

The Protocol structure `SCARD_IO_HEADER` contains the following members:

`Protocol`	Values range from 0 through 14, inclusive, indicating protocol T=0, T=1, ..., T=14.
`Length`	Not used.

Return Values The following values are returned:

`IFD_SUCCESS`	Successful completion.
`IFD_RESPONSE_TIMEOUT`	The response timed out.
`IFD_ICC_NOT_PRESENT`	The card is not present.
`IFD_PROTOCOL_NOT_SUPPORTED`	The protocol is not supported
`IFD_COMMUNICATION_ERROR`	An error occurred.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

See Also `IFDHCreateChannelByName(3SMARTCARD)`, `attributes(5)`

Notes This manual page is derived from the MUSCLE PC/SC IFD Driver Developer Kit documentation. License terms and attribution and copyright information for this interface are available at the default location `/var/sadm/pkg/SUNWocfh/install/copyright`. If the Solaris Operating Environment has been installed anywhere other than the default location, modify the path to access the file at the installed location.

Name ilogb, ilogbf, ilogbl – return an unbiased exponent

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

int ilogb(double x);

int ilogbf(float x);

int ilogbl(long double x);

cc [ flag... ] file... -lm [ library... ]
#include <math.h>

int ilogb(double x);

int ilogbf(float x);

int ilogbl(long double x);
```

Description These functions return the exponent part of their argument *x*. Formally, the return value is the integral part of $\log_r |x|$ as a signed integral value, for non-zero *x*, where *r* is the radix of the machine's floating point arithmetic, which is the value of `FLT_RADIX` defined in `<float.h>`.

Return Values Upon successful completion, these functions return the exponent part of *x* as a signed integer value. They are equivalent to calling the corresponding `logb(3M)` function and casting the returned value to type `int`.

If *x* is 0, the value `FP_ILOGB0` is returned. For SUSv3–conforming applications compiled with the `c99` compiler driver (see `standards(5)`), a domain error occurs.

If *x* is ±Inf, the value `INT_MAX` is returned. For SUSv3–conforming applications compiled with the `c99` compiler driver, a domain error occurs.

If *x* is NaN, the value `FP_ILOGBNAN` is returned. For SUSv3–conforming applications compiled with the `c99` compiler driver, a domain error occurs.

Errors These functions will fail if:

Domain Error — The *x* argument is zero, NaN, or ±Inf.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, then the invalid floating-point exception is raised.

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

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

See Also feclearexcept(3M), fetestexcept(3M), limits.h(3HEAD), logb(3M), math.h(3HEAD), scalb(3M), attributes(5), standards(5)

Name isencrypt – determine whether a buffer of characters is encrypted

Synopsis

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

#include<libgen.h>

int isencrypt(const char *fbuf, size_t ninbuf);
```

Description `isencrypt()` uses heuristics to determine whether a buffer of characters is encrypted. It requires two arguments: a pointer to an array of characters and the number of characters in the buffer.

`isencrypt()` assumes that the file is not encrypted if all the characters in the first block are ASCII characters. If there are non-ASCII characters in the first *ninbuf* characters, and if the `setlocale()` `LC_CTYPE` category is set to `C` or `ascii`, `isencrypt()` assumes that the buffer is encrypted

If the `LC_CTYPE` category is set to a value other than `C` or `ascii`, then `isencrypt()` uses a combination of heuristics to determine if the buffer is encrypted. If *ninbuf* has at least 64 characters, a chi-square test is used to determine if the bytes in the buffer have a uniform distribution; if it does, then `isencrypt()` assumes the buffer is encrypted. If the buffer has less than 64 characters, a check is made for null characters and a terminating new-line to determine whether the buffer is encrypted.

Return Values If the buffer is encrypted, `1` is returned; otherwise, zero is returned.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `setlocale(3C)`, `attributes(5)`

Notes When compiling multithreaded applications, the `_REENTRANT` flag must be defined on the compile line. This flag should only be used in multithreaded applications.

Name isfinite – test for finite value

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

int isfinite(real-floating x);
```

Description The `isfinite()` macro determines whether its argument has a finite value (zero, subnormal, or normal, and not infinite or NaN). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

Return Values The `isfinite()` macro returns a non-zero value if and only if its argument has a finite value.

Errors No errors are defined.

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

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

See Also `fpclassify(3M)`, `isinf(3M)`, `isnan(3M)`, `isnormal(3M)`, `math.h(3HEAD)`, `signbit(3M)`, `attributes(5)`, `standards(5)`

Name isgreater – test if x greater than y

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

int isgreater(real-floating x, real-floating y);
```

Description The `isgreater()` macro determines whether its first argument is greater than its second argument. The value of `isgreater`(*x*, *y*) is equal to (*x*) > (*y*); however, unlike (*x*) > (*y*), `isgreater`(*x*, *y*) does not raise the invalid floating-point exception when *x* and *y* are unordered.

Return Values Upon successful completion, the `isgreater()` macro returns the value of (*x*) > (*y*).

If x or y is NaN, 0 is returned.

Errors No errors are defined.

Usage The relational and equality operators support the usual mathematical relationships between numeric values. For any ordered pair of numeric values, exactly one of the relationships (less, greater, and equal) is true. Relational operators can raise the invalid floating-point exception when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the unordered relationship is true. This macro is a quiet (non-floating-point exception raising) version of a relational operator. It facilitates writing efficient code that accounts for quiet NaNs without suffering the invalid floating-point exception. In the `SYNOPSIS` section, `real-floating` indicates that the argument is an expression of `real-floating` type.

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

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

See Also `isgreaterequal(3M)`, `isless(3M)`, `islessequal(3M)`, `islessgreater(3M)`, `isunordered(3M)`, `math.h(3HEAD)`, `attributes(5)`, `standards(5)`

Name isgreaterequal – test if x greater than or equal to y

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

int isgreaterequal(real-floating x, real-floating y);
```

Description The `isgreaterequal()` macro determines whether its first argument is greater than or equal to its second argument. The value of `isgreaterequal`(x, y) is equal to $(x) \geq (y)$; however, unlike $(x) \geq (y)$, `isgreaterequal`(x, y) does not raise the invalid floating-point exception when x and y are unordered.

Return Values Upon successful completion, the `isgreaterequal()` macro returns the value of $(x) \geq (y)$.

If x or y is NaN, 0 is returned.

Errors No errors are defined.

Usage The relational and equality operators support the usual mathematical relationships between numeric values. For any ordered pair of numeric values, exactly one of the relationships (less, greater, and equal) is true. Relational operators can raise the invalid floating-point exception when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the unordered relationship is true. This macro is a quiet (non-floating-point exception raising) version of a relational operator. It facilitates writing efficient code that accounts for quiet NaNs without suffering the invalid floating-point exception. In the `SYNOPSIS` section, `real-floating` indicates that the argument is an expression of `real-floating` type.

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

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

See Also `isgreater(3M)`, `isless(3M)`, `islessequal(3M)`, `islessgreater(3M)`, `isunordered(3M)`, `math.h(3HEAD)`, `attributes(5)`, `standards(5)`

Name isinf – test for infinity

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

int isinf(real-floating x);
```

Description The `isinf()` macro determines whether its argument value is an infinity (positive or negative). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

Return Values The `isinf()` macro returns a non-zero value if and only if its argument has an infinite value.

Errors No errors are defined.

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

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

See Also `fpclassify(3M)`, `isfinite(3M)`, `isnan(3M)`, `isnormal(3M)`, `math.h(3HEAD)`, `signbit(3M)`, `attributes(5)`, `standards(5)`

Name isless – test if x is less than y

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

int isless(real-floating x, real-floating y);
```

Description The `isless()` macro determines whether its first argument is less than its second argument. The value of `isless`(*x*, *y*) is equal to (*x*) < (*y*); however, unlike (*x*) < (*y*), `isless`(*x*, *y*) does not raise the invalid floating-point exception when *x* and *y* are unordered.

Return Values Upon successful completion, the `isless()` macro returns the value of (*x*) < (*y*).

If *x* or *y* is NaN, 0 is returned.

Errors No errors are defined.

Usage The relational and equality operators support the usual mathematical relationships between numeric values. For any ordered pair of numeric values, exactly one of the relationships (less, greater, and equal) is true. Relational operators can raise the invalid floating-point exception when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the unordered relationship is true. This macro is a quiet (non-floating-point exception raising) version of a relational operator. It facilitates writing efficient code that accounts for quiet NaNs without suffering the invalid floating-point exception. In the `SYNOPSIS` section, `real-floating` indicates that the argument is an expression of `real-floating` type.

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

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

See Also `isgreater(3M)`, `isgreaterequal(3M)`, `islessequal(3M)`, `islessgreater(3M)`, `isunordered(3M)`, `math.h(3HEAD)`, `attributes(5)`, `standards(5)`

Name islessequal – test if x is less than or equal to y

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

int islessequal(real-floating x, real-floating y);
```

Description The `islessequal()` macro determines whether its first argument is less than or equal to its second argument. The value of `islessequal`(x, y) is equal to (x) ≤ (y); however, unlike (x) ≤ (y), `islessequal`(x, y) does not raise the invalid floating-point exception when x and y are unordered.

Return Values Upon successful completion, the `islessequal()` macro returns the value of (x) ≤ (y).

If x or y is NaN, 0 is returned.

Errors No errors are defined.

Usage The relational and equality operators support the usual mathematical relationships between numeric values. For any ordered pair of numeric values, exactly one of the relationships (less, greater, and equal) is true. Relational operators can raise the invalid floating-point exception when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the unordered relationship is true. This macro is a quiet (non-floating-point exception raising) version of a relational operator. It facilitates writing efficient code that accounts for quiet NaNs without suffering the invalid floating-point exception. In the `SYNOPSIS` section, `real-floating` indicates that the argument is an expression of `real-floating` type.

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

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

See Also `isgreater(3M)`, `isgreaterequal(3M)`, `isless(3M)`, `islessgreater(3M)`, `isunordered(3M)`, `math.h(3HEAD)`, `attributes(5)`, `standards(5)`

Name islessgreater – test if x is less than or greater than y

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

int islessgreater(real-floating x, real-floating y);
```

Description The `islessgreater()` macro determines whether its first argument is less than or greater than its second argument. The `islessgreater`(*x*, *y*) macro is similar to (*x*) < (*y*) || (*x*) > (*y*); however, `islessgreater`(*x*, *y*) does not raise the invalid floating-point exception when x and y are unordered (nor does it evaluate *x* and *y* twice).

Return Values Upon successful completion, the `islessgreater()` macro returns the value of (*x*) < (*y*) || (*x*) > (*y*).

If *x* or *y* is NaN, 0 is returned.

Errors No errors are defined.

Usage The relational and equality operators support the usual mathematical relationships between numeric values. For any ordered pair of numeric values, exactly one of the relationships (less, greater, and equal) is true. Relational operators can raise the invalid floating-point exception when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the unordered relationship is true. This macro is a quiet (non-floating-point exception raising) version of a relational operator. It facilitates writing efficient code that accounts for quiet NaNs without suffering the invalid floating-point exception. In the `SYNOPSIS` section, `real-floating` indicates that the argument is an expression of `real-floating` type.

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

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

See Also `isgreater(3M)`, `isgreaterequal(3M)`, `isless(3M)`, `islessequal(3M)`, `isunordered(3M)`, `math.h(3HEAD)`, `attributes(5)`, `standards(5)`

Name isnan – test for NaN

Synopsis

```
cc [ flag... ] file... -lm [ library... ]
#include <math.h>

int isnan(double x);
```

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

int isnan(real–floating x);
```

Description In C90 mode, the `isnan()` function tests whether *x* is NaN.

In C99 mode, the `isnan()` macro determines whether its argument value is NaN. First, an argument represented in a format wider than its semantic type is converted to its semantic type. The determination is then based on the type of the argument.

Return Values Both the `isnan()` function and macro return non-zero if and only if *x* is NaN.

Errors No errors are defined.

Warnings In C99 mode, the practice of explicitly supplying a prototype for `isnan()` after the line

```
#include <math.h>
```

is obsolete and will no longer work.

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

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

See Also `fpclassify(3M)`, `isfinite(3M)`, `isinf(3M)`, `isnormal(3M)`, `math.h(3HEAD)`, `signbit(3M)`, `attributes(5)`, `standards(5)`

Name isnormal – test for a normal value

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

int isnormal(real-floating x);
```

Description The `isnormal()` macro determines whether its argument value is normal (neither zero, subnormal, infinite, nor NaN). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

Return Values The `isnormal()` macro returns a non-zero value if and only if its argument has a normal value.

Errors No errors are defined.

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

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

See Also `fpclassify(3M)`, `isfinite(3M)`, `isinf(3M)`, `isnan(3M)`, `math.h(3HEAD)`, `signbit(3M)`, `attributes(5)`, `standards(5)`

Name isunordered – test if arguments are unordered

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

int isunordered(real-floating x, real-floating y);
```

Description The `isunordered()` macro determines whether its arguments are unordered.

Return Values Upon successful completion, the `isunordered()` macro returns 1 if its arguments are unordered and 0 otherwise.

Errors No errors are defined.

Usage The relational and equality operators support the usual mathematical relationships between numeric values. For any ordered pair of numeric values, exactly one of the relationships (less, greater, and equal) is true. Relational operators can raise the invalid floating-point exception when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the unordered relationship is true. This macro is a quiet (non-floating-point exception raising) version of a relational operator. It facilitates writing efficient code that accounts for quiet NaNs without suffering the invalid floating-point exception. In the `SYNOPSIS` section, `real-floating` indicates that the argument shall be an expression of `real-floating` type.

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

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

See Also `isgreater(3M)`, `isgreaterequal(3M)`, `isless(3M)`, `islessequal(3M)`, `islessgreater(3M)`, `math.h(3HEAD)`, `attributes(5)`, `standards(5)`

Name j0, j0f, j0l, j1, j1f, j1l, jn, jnf, jnl – Bessel functions of the first kind

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double j0(double x);

float j0f(float x);

long double j0l(long double x);

double j1(double x);

float j1f(float x);

long double j1l(long double x);

double jn(int n, double x);

float jnf(int n, float x);

long double jnl(int n, long double x);
```

Description These functions compute Bessel functions of *x* of the first kind of orders 0, 1 and *n* respectively.

Return Values Upon successful completion, these functions return the relevant Bessel value of *x* of the first kind.

If *x* is NaN, a NaN is returned.

Errors No errors are defined.

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

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

The j0(), j1(), and jn() functions are Standard. The j0f(), j0l(), j1f(), j1l(), jnf(), and jnl() functions are Stable.

See Also isnan(3M), y0(3M), math.h(3HEAD), attributes(5), standards(5)

Name kstat – kernel statistics facility

Description The `kstat` facility is a general-purpose mechanism for providing kernel statistics to users.

The kstat model The kernel maintains a linked list of statistics structures, or kstats. Each kstat has a common header section and a type-specific data section. The header section is defined by the `kstat_t` structure:

kstat header

```
typedef   int kid_t;    /* unique kstat id */

typedef struct kstat {
   /*
    * Fields relevant to both kernel and user
    */
   hrtime_t      ks_crtime;              /* creation time */
   struct kstat  *ks_next;               /* kstat chain linkage */
   kid_t         ks_kid;                 /* unique kstat ID */
   char          ks_module[KSTAT_STRLEN]; /* module name */
   uchar_t       ks_resv;                /* reserved */
   int           ks_instance;            /* module's instance */
   char          ks_name[KSTAT_STRLEN];  /* kstat name */
   uchar_t       ks_type;                /* kstat data type */
   char          ks_class[KSTAT_STRLEN]; /* kstat class */
   uchar_t       ks_flags;               /* kstat flags */
   void          *ks_data;               /* kstat type-specific
                                            data */
   uint_t        ks_ndata;               /* # of data records */
   size_t        ks_data_size;           /* size of kstat data
                                            section */
   hrtime_t      ks_snaptime;            /* time of last data
                                            snapshot */

   /*
    * Fields relevant to kernel only
    */
   int(*ks_update)(struct kstat *, int);
   void  *ks_private;
   int(*ks_snapshot)(struct kstat *, void *, int);
   void  *ks_lock;
} kstat_t;
```

The fields that are of significance to the user are:

`ks_crtime` The time the kstat was created. This allows you to compute the rates of various counters since the kstat was created; "rate since boot" is replaced by the more general concept of "rate since kstat creation". All times associated with kstats (such as creation time, last snapshot time, `kstat_timer_t` and `kstat_io_t` timestamps, and the like) are 64-bit nanosecond values. The accuracy of kstat timestamps is machine

dependent, but the precision (units) is the same across all platforms. See `gethrtime(3C)` for general information about high-resolution timestamps.

`ks_next` kstats are stored as a linked list, or chain. `ks_next` points to the next kstat in the chain.

`ks_kid` A unique identifier for the kstat.

`ks_module`, `ks_instance` contain the name and instance of the module that created the kstat. In cases where there can only be one instance, `ks_instance` is 0.

`ks_name` gives a meaningful name to a kstat. The full kstat namespace is <`ks_module`,`ks_instance`,`ks_name`>, so the name only need be unique within a module.

`ks_type` The type of data in this kstat. kstat data types are discussed below.

`ks_class` Each kstat can be characterized as belonging to some broad class of statistics, such as disk, tape, net, vm, and streams. This field can be used as a filter to extract related kstats. The following values are currently in use: `disk`, `tape`, `controller`, `net`, `rpc`, `vm`, `kvm`, `hat`, `streams`, `kmem`, `kmem_cache`, `kstat`, and `misc`. (The kstat class encompasses things like *kstat_types*.)

`ks_data`, `ks_ndata`, `ks_data_size` `ks_data` is a pointer to the kstat's data section. The type of data stored there depends on `ks_type`. `ks_ndata` indicates the number of data records. Only some kstat types support multiple data records. Currently, `KSTAT_TYPE_RAW`, `KSTAT_TYPE_NAMED` and `KSTAT_TYPE_TIMER` kstats support multiple data records. `KSTAT_TYPE_INTR` and `KSTAT_TYPE_IO` kstats support only one data record. `ks_data_size` is the total size of the data section, in bytes.

`ks_snaptime` The timestamp for the last data snapshot. This allows you to compute activity rates:

```
rate = (new_count - old_count) / (new_snaptime - old_snaptime);
```

kstat data types

The following types of kstats are currently available:

```
#define KSTAT_TYPE_RAW    0   /* can be anything */
#define KSTAT_TYPE_NAMED  1   /* name/value pairs */
#define KSTAT_TYPE_INTR   2   /* interrupt statistics */
#define KSTAT_TYPE_IO     3   /* I/O statistics */
#define KSTAT_TYPE_TIMER  4   /* event timers */
```

To get a list of all kstat types currently supported in the system, tools can read out the standard system kstat *kstat_types* (full name spec is <*"unix", 0, "kstat_types"*>). This is a KSTAT_TYPE_NAMED kstat in which the *name* field describes the type of kstat, and the *value* field is the kstat type number (for example, KSTAT_TYPE_IO is type 3 -- see above).

Raw kstat KSTAT_TYPE_RAW raw data

The "raw" kstat type is just treated as an array of bytes. This is generally used to export well-known structures, like *sysinfo*.

Name=value kstat KSTAT_TYPE_NAMED A list of arbitrary *name=value* statistics.

```
typedef struct kstat_named {
   char    name[KSTAT_STRLEN];    /* name of counter */
   uchar_t data_type;             /* data type */
   union {
            charc[16];            /* enough for 128-bit ints */
            struct {
               union {
                   char *ptr;    /* NULL-terminated string */
               } addr;
               uint32_t len;     /* length of string */
            } str;
            int32_t   i32;
            uint32_t  ui32;
            int64_t   i64;
            uint64_t  ui64;

  /* These structure members are obsolete */

            int32_t   l;
            uint32_t  ul;
            int64_t   ll;
            uint64_t  ull;
         } value;                 /* value of counter */
} kstat_named_t;

/* The following types are Stable

KSTAT_DATA_CHAR
KSTAT_DATA_INT32
KSTAT_DATA_LONG
KSTAT_DATA_UINT32
KSTAT_DATA_ULONG
KSTAT_DATA_INT64
KSTAT_DATA_UINT64

/* The following type is Evolving */
```

```
KSTAT_DATA_STRING

/* The following types are Obsolete */

KSTAT_DATA_LONGLONG
KSTAT_DATA_ULONGLONG
KSTAT_DATA_FLOAT
KSTAT_DATA_DOUBLE
```

Some devices need to publish strings that exceed the maximum value for `KSTAT_DATA_CHAR` in length; `KSTAT_DATA_STRING` is a data type that allows arbitrary-length strings to be associated with a named kstat. The macros below are the supported means to read the pointer to the string and its length.

```
#define KSTAT_NAMED_STR_PTR(knptr) ((knptr)->value.str.addr.ptr)
#define KSTAT_NAMED_STR_BUFLEN(knptr) ((knptr)->value.str.len)
```

`KSTAT_NAMED_STR_BUFLEN()` returns the number of bytes required to store the string pointed to by `KSTAT_NAMED_STR_PTR()`; that is, `strlen(KSTAT_NAMED_STR_PTR()) + 1`.

Interrupt kstat `KSTAT_TYPE_INTR` Interrupt statistics.

An interrupt is a hard interrupt (sourced from the hardware device itself), a soft interrupt (induced by the system via the use of some system interrupt source), a watchdog interrupt (induced by a periodic timer call), spurious (an interrupt entry point was entered but there was no interrupt to service), or multiple service (an interrupt was detected and serviced just prior to returning from any of the other types).

```
#define KSTAT_INTR_HARD      0
#define KSTAT_INTR_SOFT      1
#define KSTAT_INTR_WATCHDOG  2
#define KSTAT_INTR_SPURIOUS  3
#define KSTAT_INTR_MULTSVC   4
#define KSTAT_NUM_INTRS      5

typedef struct kstat_intr {
   uint_t intrs[KSTAT_NUM_INTRS]; /* interrupt counters */
} kstat_intr_t;
```

Event timer kstat `KSTAT_TYPE_TIMER` Event timer statistics.

These provide basic counting and timing information for any type of event.

```
typedef struct kstat_timer {
   char         name[KSTAT_STRLEN]; /* event name */
   uchar_t      resv;               /* reserved */
   u_longlong_t num_events;         /* number of events */
   hrtime_t     elapsed_time;       /* cumulative elapsed time */
```

```
   hrtime_t     min_time;           /* shortest event duration */
   hrtime_t     max_time;           /* longest event duration */
   hrtime_t     start_time;         /* previous event start time */
   hrtime_t     stop_time;          /* previous event stop time */
} kstat_timer_t;
```

I/O kstat `KSTAT_TYPE_IO` I/O statistics.

```
typedef struct kstat_io {
/*
 * Basic counters.
 */
u_longlong_t     nread;      /* number of bytes read */
u_longlong_t     nwritten;   /* number of bytes written */
uint_t           reads;      /* number of read operations */
uint_t           writes;     /* number of write operations */
/*
* Accumulated time and queue length statistics.
*
* Time statistics are kept as a running sum of "active" time.
* Queue length statistics are kept as a running sum of the
* product of queue length and elapsed time at that length --
* that is, a Riemann sum for queue length integrated against time.
*
*              ^
*              |                         _________
*              8                         | i4    |
*              |                         |       |
*       Queue  6                         |       |
*       Length |       _________         |       |
*              4       | i2    |_______|       |
*              |       |       i3              |
*              2_______|                       |
*              |    i1                         |
*              |_______________________________|
*              Time->  t1      t2      t3      t4
*
* At each change of state (entry or exit from the queue),
* we add the elapsed time (since the previous state change)
* to the active time if the queue length was non-zero during
* that interval; and we add the product of the elapsed time
* times the queue length to the running length*time sum.
*
* This method is generalizable to measuring residency
* in any defined system: instead of queue lengths, think
* of "outstanding RPC calls to server X".
*
* A large number of I/O subsystems have at least two basic
```

```
* "lists" of transactions they manage: one for transactions
* that have been accepted for processing but for which processing
* has yet to begin, and one for transactions which are actively
* being processed (but not done). For this reason, two cumulative
* time statistics are defined here: pre-service (wait) time,
* and service (run) time.
*
* The units of cumulative busy time are accumulated nanoseconds.
* The units of cumulative length*time products are elapsed time
* times queue length.
*/
hrtime_t   wtime;            /* cumulative wait (pre-service) time */
hrtime_t   wlentime;         /* cumulative wait length*time product*/
hrtime_t   wlastupdate;      /* last time wait queue changed */
hrtime_t   rtime;            /* cumulative run (service) time */
hrtime_t   rlentime;         /* cumulative run length*time product */
hrtime_t   rlastupdate;      /* last time run queue changed */
uint_t     wcnt;             /* count of elements in wait state */
uint_t     rcnt;             /* count of elements in run state */
} kstat_io_t;
```

Using libkstat

The kstat library, `libkstat`, defines the user interface (API) to the system's kstat facility.

You begin by opening libkstat with `kstat_open(3KSTAT)`, which returns a pointer to a fully initialized kstat control structure. This is your ticket to subsequent libkstat operations:

```
typedef struct kstat_ctl {
   kid_t     kc_chain_id;    /* current kstat chain ID */
   kstat_t   *kc_chain;      /* pointer to kstat chain */
   int       kc_kd;          /* /dev/kstat descriptor */
} kstat_ctl_t;
```

Only the first two fields, `kc_chain_id` and `kc_chain`, are of interest to `libkstat` clients. (*kc_kd* is the descriptor for `/dev/kstat`, the kernel statistics driver. libkstat functions are built on top of `/dev/kstat` `ioctl(2)` primitives. Direct interaction with `/dev/kstat` is strongly discouraged, since it is *not* a public interface.)

`kc_chain` points to your copy of the kstat chain. You typically walk the chain to find and process a certain kind of kstat. For example, to display all I/O kstats:

```
kstat_ctl_t    *kc;
kstat_t        *ksp;
kstat_io_t     kio;

kc = kstat_open();
for (ksp = kc->kc_chain; ksp != NULL; ksp = ksp->ks_next) {
        if (ksp->ks_type == KSTAT_TYPE_IO) {
              kstat_read(kc, ksp, &kio);
```

```
                my_io_display(kio);
        }
}
```

`kc_chain_id` is the kstat chain ID, or KCID, of your copy of the kstat chain. See `kstat_chain_update(3KSTAT)` for an explanation of KCIDs.

Files

`/dev/kstat`	kernel statistics driver
`/usr/include/kstat.h`	header
`/usr/include/sys/kstat.h`	header

See Also `ioctl(2)`, `gethrtime(3C)`, `getloadavg(3C)`, `kstat_chain_update(3KSTAT)`, `kstat_close(3KSTAT)`, `kstat_data_lookup(3KSTAT)`, `kstat_lookup(3KSTAT)`, `kstat_open(3KSTAT)`, `kstat_read(3KSTAT)`, `kstat_write(3KSTAT)`, `attributes(5)`

Name Kstat – Perl tied hash interface to the kstat facility

Synopsis

```
use Sun::Solaris::Kstat;

Sun::Solaris::Kstat->new();
Sun::Solaris::Kstat->update();
Sun::Solaris::Kstat->{module}{instance}{name}{statistic}
```

Description Kernel statistics are categorized using a 3-part key consisting of the module, the instance, and the statistic name. For example, CPU information can be found under `cpu_stat:0:cpu_stat0`, as in the above example. The method `Sun::Solaris::Kstat→new()` creates a new 3-layer tree of Perl hashes with the same structure; that is, the statistic for CPU 0 can be accessed as `$ks→{cpu_stat}{0}{cpu_stat0}`. The fourth and lowest layer is a tied hash used to hold the individual statistics values for a particular system resource.

For performance reasons, the creation of a `Sun::Solaris::Kstat` object is not accompanied by a following read of all possible statistics. Instead, the 3-layer structure described above is created, but reads of a statistic's values are done only when referenced. For example, accessing `$ks→{cpu_stat}{0}{cpu_stat0}{syscall}` will read in all the statistics for CPU 0, including user, system, and wait times, and the other CPU statistics, as well as the number of system call entries. Once you have accessed a lowest level statistics value, calling `$ks→update()` will automatically update all the individual values of any statistics you have accessed.

There are two values of the lowest-level hash that can be read without causing the full set of statistics to be read from the kernel. These are "class", which is the `kstat` class of the statistics, and "crtime"n, which is the time that the kstat was created. See `kstat(3KSTAT)` for full details of these fields.

Methods

`new()` Create a new kstat statistics hierarchy and return a reference to the top-level hash. Use it like any normal hash to access the statistics.

`update()` Update all the statistics that have been accessed so far. In scalar context, `update()` returns 1 if the `kstat` structure has changed, and 0 otherwise. In list context, `update()` returns references to two arrays: the first holds the keys of any kstats that have been added, and the second holds the keys of any kstats that have been deleted. Each key will be returned in the form "`module:instance:name`".

Examples **EXAMPLE 1** Sun::Solaris::Kstat example

```
use Sun::Solaris::Kstat;

my $kstat = Sun::Solaris::Kstat->new();
my ($usr1, $sys1, $wio1, $idle1) =
   @{$kstat->{cpu_stat}{0}{cpu_stat0}}{qw(user kernel
```

EXAMPLE 1 Sun::Solaris::Kstat example *(Continued)*

```
        wait idle)};
print("usr sys wio idle\n");
while (1) {
   sleep 5;
   if ($kstat->update()) {
           print("Configuration changed\n");
   }
   my ($usr2, $sys2, $wio2, $idle2) =
       @{$kstat->{cpu_stat}{0}{cpu_stat0}}{qw(user kernel
            wait idle)};
   printf(" %.2d  %.2d  %.2d  %.2d\n",
        ($usr2 - $usr1) / 5, ($sys2 - $sys1) / 5,
        ($wio2 - $wio1) / 5, ($idle2 - $idle1) / 5);
   $usr1 = $usr2;
   $sys1 = $sys2;
   $wio1 = $wio2;
   $idle1 = $idle2;
}
```

See Also perl(1), kstat(1M), kstat(3KSTAT), kstat_chain_update(3KSTAT), kstat_close(3KSTAT), kstat_open(3KSTAT), kstat_read(3KSTAT)

Notes As the statistics are stored in a tied hash, taking additional references of members of the hash, such as

```
my $ref = \ks->{cpu_stat}{0}{cpu_stat0}{syscall};
print("$$ref\n");
```

will be recorded as a hold on that statistic's value, preventing it from being updated by `refresh()`. Copy the values explicitly if persistence is necessary.

Several of the statistics provided by the `kstat` facility are stored as 64-bit integer values. Perl 5 does not yet internally support 64-bit integers, so these values are approximated in this module. There are two classes of 64-bit value to be dealt with:

64-bit intervals and times — These are the `crtime` and `snaptime` fields of all the statistics hashes, and the `wtime`, `wlentime`, `wlastupdate`, `rtime`, `rlentime` and `rlastupdate` fields of the `kstat` I/O statistics structures. These are measured by the `kstat` facility in nanoseconds, meaning that a 32-bit value would represent approximately 4 seconds. The alternative is to store the values as floating-point numbers, which offer approximately 53 bits of precision on present hardware. 64-bit intervals and timers as

	floating point values expressed in seconds, meaning that time-related kstats are being rounded to approximately microsecond resolution.
64-bit counters	It is not useful to store these values as 32-bit values. As noted above, floating-point values offer 53 bits of precision. Accordingly, all 64-bit counters are stored as floating-point values.

Name kstat_chain_update – update the kstat header chain

Synopsis

```
cc [ flag... ] file... -lkstat [ library...]
#include <kstat.h>

kid_t kstat_chain_update(kstat_ctl_t *kc);
```

Description The `kstat_chain_update()` function brings the user's kstat header chain in sync with that of the kernel. The kstat chain is a linked list of kstat headers (`kstat_t`'s) pointed to by *kc*->`kc_chain`, which is initialized by `kstat_open(3KSTAT)`. This chain constitutes a list of all kstats currently in the system.

During normal operation, the kernel creates new kstats and delete old ones as various device instances are added and removed, thereby causing the user's copy of the kstat chain to become out of date. The `kstat_chain_update()` function detects this condition by comparing the kernel's current kstat chain ID (KCID), which is incremented every time the kstat chain changes, to the user's KCID, *kc*->`kc_chain_id`. If the KCIDs match, `kstat_chain_update()` does nothing. Otherwise, it deletes any invalid kstat headers from the user's kstat chain, adds any new ones, and sets *kc*->`kc_chain_id` to the new KCID. All other kstat headers in the user's kstat chain are unmodified.

Return Values Upon successful completion, `kstat_chain_update()` returns the new KCID if the kstat chain has changed and 0 if it has not changed. Otherwise, it returns −1 and sets `errno` to indicate the error.

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

`EAGAIN` The kstat was temporarily unavailable for reading or writing.

`ENOMEM` Insufficient storage space is available.

`ENXIO` The given kstat could not be located for reading.

`EOVERFLOW` The data for the given kstat was too large to be stored in the structure.

Files `/dev/kstat` kernel statistics driver

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Stable
MT-Level	Unsafe

See Also kstat(3KSTAT), kstat_lookup(3KSTAT), kstat_open(3KSTAT), kstat_read(3KSTAT), attributes(5)

Name kstat_lookup, kstat_data_lookup – find a kstat by name

Synopsis

```
cc [ flag... ] file... -lkstat [ library...]
#include <kstat.h>

kstat_t *kstat_lookup(kstat_ctl_t *kc, char *ks_module, int ks_instance,
     char *ks_name);

void *kstat_data_lookup(kstat_t *ksp, char *name);
```

Description The `kstat_lookup()` function traverses the kstat chain, *kc*->`kc_chain`, searching for a kstat with the same *ks_module*, *ks_instance*, and *ks_name* fields; this triplet uniquely identifies a kstat. If *ks_module* is `NULL`, *ks_instance* is -1, or *ks_name* is `NULL`, those fields will be ignored in the search. For example, `kstat_lookup(kc, NULL, -1, "foo")` will find the first kstat with name "foo".

The `kstat_data_lookup()` function searches the kstat's data section for the record with the specified *name*. This operation is valid only for those kstat types that have named data records: `KSTAT_TYPE_NAMED` and `KSTAT_TYPE_TIMER`.

Return Values The `kstat_lookup()` function returns a pointer to the requested kstat if it is found. Otherwise it returns `NULL` and sets `errno` to indicate the error.

The `kstat_data_lookup()` function returns a pointer to the requested data record if it is found. Otherwise it returns `NULL` and sets `errno` to indicate the error .

Errors The `kstat_lookup()` and `kstat_data_lookup()` functions will fail if:

`EINVAL` An attempt was made to look up data for a kstat that was not of type `KSTAT_TYPE_NAMED` or `KSTAT_TYPE_TIMER`.

`ENOENT` The requested kstat could not be found.

Files `/dev/kstat` kernel statistics driver

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Stable
MT-Level	Unsafe

See Also `kstat(3KSTAT)`, `kstat_chain_update(3KSTAT)`, `kstat_open(3KSTAT)`, `kstat_read(3KSTAT)`, `attributes(5)`

Name kstat_open, kstat_close – initialize kernel statistics facility

Synopsis

```
cc[ flag... ] file... -lkstat [ library...]
#include <kstat.h>

kstat_ctl_t *kstat_open(void);

int kstat_close(kstat_ctl_t *kc);
```

Description The `kstat_open()` function initializes a kstat control structure that provides access to the kernel statistics library. It returns a pointer to this structure, which must be supplied as the *kc* argument in subsequent `libkstat` function calls.

The `kstat_close()` function frees all resources that were associated with *kc*. This is performed automatically on `exit(2)` and `execve(2)`.

Return Values Upon successful completion, `kstat_open()` returns a pointer to a kstat control structure. Otherwise, it returns `NULL`, no resources are allocated, and `errno` is set to indicate the error.

Upon successful completion, `kstat_close()` returns 0. Otherwise, −1 is returned and `errno` is set to indicate the error.

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

`ENOMEM` Insufficient storage space is available.

`EAGAIN` The kstat was temporarily unavailable for reading or writing.

`ENXIO` The given kstat could not be located for reading.

`EOVERFLOW` The data for the given kstat was too large to be stored in the structure.

The `kstat_open()` function can also return the error values for `open(2)`.

The `kstat_close()` function can also return the error values for `close(2)`.

Files `/dev/kstat` kernel statistics driver

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Stable
MT-Level	Unsafe

See Also `close(2)`, `execve(2)`, `open(2)`, `exit(2)`, `kstat(3KSTAT)`, `kstat_chain_update(3KSTAT)`, `kstat_lookup(3KSTAT)`, `kstat_read(3KSTAT)`, `attributes(5)`

Name kstat_read, kstat_write – read or write kstat data

Synopsis

```
cc [ flag... ] file... -lkstat [ library... ]
#include <kstat.h>

kid_t kstat_read(kstat_ctl_t *kc, kstat_t *ksp, void *buf);

kid_t kstat_write(kstat_ctl_t *kc, kstat_t *ksp, void *buf);
```

Description The `kstat_read()` function gets data from the kernel for the kstat pointed to by *ksp*. The *ksp*->`ks_data` field is automatically allocated (or reallocated) to be large enough to hold all of the data. The *ksp*->`ks_ndata` field is set to the number of data fields, *ksp*->`ks_data_size` is set to the total size of the data, and *ksp*->`ks_snaptime` is set to the high-resolution time at which the data snapshot was taken. If *buf* is non-null, the data is copied from *ksp*->`ks_data` to *buf*.

The `kstat_write()` function writes data from *buf*, or from *ksp*->`ks_data` if *buf* is `NULL`, to the corresponding kstat in the kernel. Only the superuser can use `kstat_write()`.

Return Values Upon successful completion, `kstat_read()` and `kstat_write()` return the current kstat chain ID (KCID). Otherwise, they return −1 and set `errno` to indicate the error.

Errors The `kstat_read()` and `kstat_write()` functions will fail if:

`EACCES` An attempt was made to write to a non-writable kstat.

`EAGAIN` The kstat was temporarily unavailable for reading or writing.

`EINVAL` An attempt was made to write data to a kstat, but the number of elements or the data size does not match.

`ENOMEM` Insufficient storage space is available.

`ENXIO` The given kstat could not be located for reading or writing.

`EOVERFLOW` The data for the given kstat was too large to be stored in the structure.

`EPERM` An attempt was made to write to a kstat, but {`PRIV_SYS_CONFIG`} was not asserted in the effective privilege set.

Files `/dev/kstat` kernel statistics driver

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Stable
MT-Level	Unsafe

See Also kstat(3KSTAT), kstat_chain_update(3KSTAT), kstat_lookup(3KSTAT), kstat_open(3KSTAT), attributes(5), privileges(5)

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 kvm_getu, kvm_getcmd – get the u-area or invocation arguments for a process

Synopsis

```
cc [ flag... ] file... -lkvm [ library...]
#include <kvm.h>
#include <sys/param.h>
#include <sys/user.h>
#include <sys/proc.h>

struct user *kvm_getu(kvm_t *kd, struct proc *proc);

int kvm_getcmd(kvm_t *kd, struct proc *proc, struct user *u, char ***arg,
     char ***env);
```

Description The `kvm_getu()` function reads the u-area of the process specified by *proc* to an area of static storage associated with *kd* and returns a pointer to it. Subsequent calls to `kvm_getu()` will overwrite this static area.

The *kd* argument is a pointer to a kernel descriptor returned by `kvm_open(3KVM)`. The *proc* argument is a pointer to a copy in the current process's address space of a *proc* structure, obtained, for instance, by a prior `kvm_nextproc(3KVM)` call.

The `kvm_getcmd()` function constructs a list of string pointers that represent the command arguments and environment that were used to initiate the process specified by *proc*.

The *kd* argument is a pointer to a kernel descriptor returned by `kvm_open(3KVM)`. The *u* argument is a pointer to a copy in the current process's address space of a `user` structure, obtained, for instance, by a prior `kvm_getu()` call. If *arg* is not `NULL`, the command line arguments are formed into a null-terminated array of string pointers. The address of the first such pointer is returned in *arg*. If *env* is not `NULL`, the environment is formed into a null-terminated array of string pointers. The address of the first of these is returned in *env*.

The pointers returned in *arg* and *env* refer to data allocated by `malloc()` and should be freed by a call to `free()` when no longer needed. See `malloc(3C)`. Both the string pointers and the strings themselves are deallocated when freed.

Since the environment and command line arguments might have been modified by the user process, there is no guarantee that it will be possible to reconstruct the original command at all. The `kvm_getcmd()` function will make the best attempt possible, returning −1 if the user process data is unrecognizable.

Return Values On success, `kvm_getu()` returns a pointer to a copy of the u-area of the process specified by *proc*. On failure, it returns `NULL`.

The `kvm_getcmd()` function returns 0 on success and −1 on failure. If −1 is returned, the caller still has the option of using the command line fragment that is stored in the u-area.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Stable
MT-Level	Unsafe

See Also kvm_nextproc(3KVM), kvm_open(3KVM), kvm_kread(3KVM), malloc(3C), libkvm(3LIB), attributes(5)

Notes On systems that support both 32-bit and 64-bit processes, the 64-bit implementation of libkvm ensures that the *arg* and env pointer arrays for kvm_getcmd() are translated to the same form as if they were 64-bit processes. Applications that wish to access the raw 32-bit stack directly can use kvm_uread(). See kvm_read(3KVM).

Name kvm_kread, kvm_kwrite, kvm_uread, kvm_uwrite – copy data to or from a kernel image or running system

Synopsis

```
cc [ flag... ] file... -lkvm [ library...]
#include <kvm.h>

ssize_t kvm_kread(kvm_t *kd, uintptr_t addr, void *buf, size_t nbytes);

ssize_t kvm_kwrite(kvm_t *kd, uintptr_t addr, void *buf, size_t nbytes);

ssize_t kvm_uread(kvm_t *kd, uintptr_t addr, void *buf, size_t nbytes);

ssize_t kvm_uwrite(kvm_t *kd, uintptr_t addr, void *buf, size_t nbytes);
```

Description The `kvm_kread()` function transfers data from the kernel address space to the address space of the process. *nbytes* bytes of data are copied from the kernel virtual address given by *addr* to the buffer pointed to by *buf*.

The `kvm_kwrite()` function is like `kvm_kread()`, except that the direction of the transfer is reversed. To use this function, the `kvm_open(3KVM)` call that returned *kd* must have specified write access.

The `kvm_uread()` function transfers data from the address space of the processes specified in the most recent `kvm_getu(3KVM)` call. *nbytes* bytes of data are copied from the user virtual address given by *addr* to the buffer pointed to by *buf*.

The `kvm_uwrite()` function is like `kvm_uread()`, except that the direction of the transfer is reversed. To use this function, the `kvm_open(3KVM)` call that returned *kd* must have specified write access. The address is resolved in the address space of the process specified in the most recent `kvm_getu(3KVM)` call.

Return Values On success, these functions return the number of bytes actually transferred. On failure, they return −1.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Stable
MT-Level	Unsafe

See Also `kvm_getu(3KVM)`, `kvm_nlist(3KVM)`, `kvm_open(3KVM)`, `attributes(5)`

Name kvm_nextproc, kvm_getproc, kvm_setproc – read system process structures

Synopsis

```
cc [ flag... ] file... -lkvm [ library...]
#include <kvm.h>
#include <sys/param.h>
#include <sys/time.h>
#include <sys/proc.h>

struct proc *kvm_nextproc(kvm_t *kd);

int kvm_setproc(kvm_t *kd);

struct proc *kvm_getproc(kvm_t *kd, pid_t pid);
```

Description The `kvm_nextproc()` function reads sequentially all of the system process structures from the kernel identified by *kd* (see `kvm_open(3KVM)`). Each call to `kvm_nextproc()` returns a pointer to the static memory area that contains a copy of the next valid process table entry. There is no guarantee that the data will remain valid across calls to `kvm_nextproc()`, `kvm_setproc()`, or `kvm_getproc()`. If the process structure must be saved, it should be copied to non-volatile storage.

For performance reasons, many implementations will cache a set of system process structures. Since the system state is liable to change between calls to `kvm_nextproc()`, and since the cache may contain obsolete information, there is no guarantee that every process structure returned refers to an active process, nor is it certain that all processes will be reported.

The `kvm_setproc()` function rewinds the process list, enabling `kvm_nextproc()` to rescan from the beginning of the system process table. This function will always flush the process structure cache, allowing an application to re-scan the process table of a running system.

The `kvm_getproc()` function locates the `proc` structure of the process specified by *pid* and returns a pointer to it. Although this function does not interact with the process table pointer manipulated by `kvm_nextproc()`, the restrictions regarding the validity of the data still apply.

Return Values On success, `kvm_nextproc()` returns a pointer to a copy of the next valid process table entry. On failure, it returns `NULL`.

On success, `kvm_getproc()` returns a pointer to the *proc* structure of the process specified by *pid*. On failure, it returns `NULL`.

The `kvm_setproc()` function returns 0 on success and –1 on failure.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Stable
MT-Level	Unsafe

See Also kvm_getu(3KVM), kvm_open(3KVM), kvm_kread(3KVM), attributes(5)

Name kvm_nlist – get entries from kernel symbol table

Synopsis

```
cc [ flag... ] file... -lkvm [ library...]
#include <kvm.h>
#include <nlist.h>

int kvm_nlist(kvm_t *kd, struct nlist *nl);
```

Description The `kvm_nlist()` function examines the symbol table from the kernel image identified by *kd* (see `kvm_open(3KVM)`) and selectively extracts a list of values and puts them in the array of `nlist` structures pointed to by *nl*. The name list pointed to by *nl* consists of an array of structures containing names, types and values. The `n_name` field of each such structure is taken to be a pointer to a character string representing a symbol name. The list is terminated by an entry with a null pointer (or a pointer to a null string) in the `n_name` field. For each entry in *nl*, if the named symbol is present in the kernel symbol table, its value and type are placed in the `n_value` and `n_type` fields. If a symbol cannot be located, the corresponding `n_type` field of *nl* is set to 0.

Return Values The `kvm_nlist()` functions returns the value of `nlist(3UCB)` or `nlist(3ELF)`, depending on the library used.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Stable
MT-Level	Unsafe

See Also `kvm_open(3KVM)`, `kvm_kread(3KVM)`, `nlist(3ELF)`, `nlist(3UCB)`, `attributes(5)`

Notes Although the `libkvm` API is Stable, the symbol names and data values that can be accessed through this set of interfaces are Private and are subject to ongoing change.

Name kvm_open, kvm_close – specify a kernel to examine

Synopsis

```
cc [ flag... ] file... -lkvm [ library...]
#include <kvm.h>
#include <fcntl.h>

kvm_t *kvm_open(char *namelist, char *corefile, char *swapfile, int flag,
     char *errstr);

int kvm_close(kvm_t *kd);
```

Description The `kvm_open()` function initializes a set of file descriptors to be used in subsequent calls to kernel virtual memory (VM) routines. It returns a pointer to a kernel identifier that must be used as the *kd* argument in subsequent kernel VM function calls.

The *namelist* argument specifies an unstripped executable file whose symbol table will be used to locate various offsets in *corefile*. If *namelist* is `NULL`, the symbol table of the currently running kernel is used to determine offsets in the core image. In this case, it is up to the implementation to select an appropriate way to resolve symbolic references, for instance, using `/dev/ksyms` as a default *namelist* file.

The *corefile* argument specifies a file that contains an image of physical memory, for instance, a kernel crash dump file (see `savecore(1M)`) or the special device `/dev/mem`. If *corefile* is `NULL`, the currently running kernel is accessed, using `/dev/mem` and `/dev/kmem`.

The *swapfile* argument specifies a file that represents the swap device. If both *corefile* and *swapfile* are `NULL`, the swap device of the currently running kernel is accessed. Otherwise, if *swapfile* is `NULL`, `kvm_open()` may succeed but subsequent `kvm_getu(3KVM)` function calls may fail if the desired information is swapped out.

The *flag* function is used to specify read or write access for *corefile* and may have one of the following values:

`O_RDONLY` open for reading

`O_RDWR` open for reading and writing

The *errstr* argument is used to control error reporting. If it is a null pointer, no error messages will be printed. If it is non-null, it is assumed to be the address of a string that will be used to prefix error messages generated by `kvm_open`. Errors are printed to `stderr`. A useful value to supply for *errstr* would be *argv*[0]. This has the effect of printing the process name in front of any error messages.

Applications using `libkvm` are dependent on the underlying data model of the kernel image, that is, whether it is a 32–bit or 64–bit kernel.

The data model of these applications must match the data model of the kernel in order to correctly interpret the size and offsets of kernel data structures. For example, a 32–bit application that uses the 32–bit version of the `libkvm` interfaces will fail to open a 64–bit

kernel image. Similarly, a 64–bit application that uses the 64–bit version of the `libkvm` interfaces will fail to open a 32–bit kernel image.

The `kvm_close()` function closes all file descriptors that were associated with *kd*. These files are also closed on `exit(2)` and `execve( )` (see `exec(2)`). `kvm_close()` also resets the `proc` pointer associated with `kvm_nextproc(3KVM)` and flushes any cached kernel data.

Return Values The `kvm_open()` function returns a non-null value suitable for use with subsequent kernel VM function calls. On failure, it returns `NULL` and no files are opened.

The `kvm_close()` function returns 0 on success and –1 on failure.

Files `/dev/kmem`

`/dev/ksyms`

`/dev/mem`

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Stable
MT-Level	Unsafe

See Also `savecore(1M)`, `exec(2)`, `exit(2)`, `pathconf(2)`, `getloadavg(3C)`, `kstat(3KSTAT)`, `kvm_getu(3KVM)`, `kvm_nextproc(3KVM)`, `kvm_nlist(3KVM)`, `kvm_kread(3KVM)`, `libkvm(3LIB)`,`sysconf(3C)`, `proc(4)`, `attributes(5)`, `lfcompile(5)`

Notes Kernel core dumps should be examined on the platform on which they were created. While a 32-bit application running on a 64-bit kernel can examine a 32-bit core dump, a 64-bit application running on a 64-bit kernel cannot examine a kernel core dump from the 32-bit system.

On 32-bit systems, applications that use `libkvm` to access the running kernel must be 32-bit applications. On systems that support both 32-bit and 64-bit applications, applications that use the `libkvm` interfaces to access the running kernel must themselves be 64-bit applications.

Although the `libkvm` API is Stable, the symbol names and data values that can be accessed through this set of interfaces are Private and are subject to ongoing change.

Applications using `libkvm` are likely to be platform- and release-dependent.

Most of the traditional uses of `libkvm` have been superseded by more stable interfaces that allow the same information to be extracted more efficiently, yet independent of the kernel data model. For examples, see `sysconf(3C)`, `proc(4)`, `kstat(3KSTAT)`, `getloadavg(3C)`, and `pathconf(2)`.

Name kvm_read, kvm_write – copy data to or from a kernel image or running system

Synopsis

```
cc [ flag... ] file... -lkvm [ library...]
#include <kvm.h>

ssize_t kvm_read(kvm_t *kd, uintptr_t addr, void *buf, size_t nbytes);

ssize_t kvm_write(kvm_t *kd, uintptr_t addr, void *buf, size_t nbytes);
```

Description The `kvm_read()` function transfers data from the kernel image specified by *kd* (see `kvm_open(3KVM)`) to the address space of the process. *nbytes* bytes of data are copied from the kernel virtual address given by *addr* to the buffer pointed to by *buf*.

The `kvm_write()` function is like `kvm_read()`, except that the direction of data transfer is reversed. To use this function, the `kvm_open(3KVM)` call that returned *kd* must have specified write access. If a user virtual address is given, it is resolved in the address space of the process specified in the most recent `kvm_getu(3KVM)` call.

Usage The `kvm_read()` and `kvm_write()` functions are obsolete and might be removed in a future release. The functions described on the `kvm_kread(3KVM)` manual page should be used instead.

Return Values On success, these functions return the number of bytes actually transferred. On failure, they return −1.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete
MT-Level	Unsafe

See Also `kvm_getu(3KVM)`, `kvm_kread(3KVM)kvm_nlist(3KVM)`, `kvm_open(3KVM)`, `attributes(5)`

Name labelbuilder, tsol_lbuild_create, tsol_lbuild_get, tsol_lbuild_set, tsol_lbuild_destroy – create a Motif-based user interface for interactively building a valid label or clearance

Synopsis

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

#include <Dt/ModLabel.h>

ModLabelData *tsol_lbuild_create(Widget widget,
     void (*event_handler)() ok_callback,
     lbuild_attributes extended_operation, ..., NULL);

void *tsol_lbuild_get(ModLabelData *data,
     lbuild_attributes extended_operation);

void tsol_lbuild_set(ModLabelData *data,
     lbuild_attributes extended_operation, ..., NULL);

void tsol_lbuild_destroy(ModLabelData *data);
```

Description The label builder user interface prompts the end user for information and generates a valid sensitivity label or clearance from the user input based on specifications in the `label_encodings(4)` file on the system where the application runs. The end user can build the label or clearance by typing a text value or by interactively choosing options.

Application-specific functionality is implemented in the callback for the OK pushbutton. This callback is passed to the `tsol_lbuild_create()` call where it is mapped to the OK pushbutton widget.

When choosing options, the label builder shows the user only those classifications (and related compartments and markings) dominated by the workspace sensitivity label unless the executable has the `PRIV_SYS_TRANS_LABEL` privilege in its effective set.

If the end user does not have the authorization to upgrade or downgrade labels, or if the user-built label is out of the user's accreditation range, the OK and Reset pushbuttons are grayed. There are no privileges to override these restrictions.

`tsol_lbuild_create()` creates the graphical user interface and returns a pointer variable of type `ModLabeldata*` that contains information on the user interface. This information is a combination of values passed in the `tsol_lbuild_create()` input parameter list, default values for information not provided, and information on the widgets used by the label builder to create the user interface. All information except the widget information should be accessed with the `tsol_lbuild_get()` and `tsol_lbuild_set()` routines.

The widget information is accessed directly by referencing the following fields of the `ModLabelData` structure.

`lbuild_dialog`	The label builder dialog box.
`ok`	The OK pushbutton.
`cancel`	The Cancel pushbutton.

`reset`	The Reset pushbutton.
`help`	The Help pushbutton.

The `tsol_lbuild_create()` parameter list takes the following values:

widget	The widget from which the dialog box is created. Any Motif widget can be passed.
ok_callback	A callback function that implements the behavior of the OK pushbutton on the dialog box.
`..., NULL`	A `NULL` terminated list of extended operations and value pairs that define the characteristics and behavior of the label builder dialog box.

`tsol_lbuild_destroy()` destroys the `ModLabelData` structure returned by `tsol_lbuild_create()`.

`tsol_lbuild_get()` and `tsol_lbuild_set()` access the information stored in the `ModLabelData` structure returned by `tsol_lbuild_create()`.

The following extended operations can be passed to `tsol_lbuild_create()` to build the user interface, to `tsol_lbuild_get()` to retrieve information on the user interface, and to `tsol_lbuild_set()` to change the user interface information. All extended operations are valid for `tsol_lbuild_get()`, but the *`WORK`* operations are not valid for `tsol_lbuild_set()` or `tsol_lbuild_create()` because these values are set from input supplied by the end user. These exceptions are noted in the descriptions.

`LBUILD_MODE`	Create a user interface to build a sensitivity label or a clearance. Value is `LBUILD_MODE_SL` by default.	
	`LBUILD_MODE_SL`	Build a sensitivity label.
	`LBUILD_MODE_CLR`	Build a clearance.
`LBUILD_VALUE_SL`	The starting sensitivity label. This value is `ADMIN_LOW` by default and is used when the mode is `LBUILD_MODE_SL`.	
`LBUILD_VALUE_CLR`	The starting clearance. This value is `ADMIN_LOW` by default and is used when the mode is `LBUILD_MODE_CLR`.	
`LBUILD_USERFIELD`	A character string prompt that displays at the top of the label builder dialog box. Value is `NULL` by default.	
`LBUILD_SHOW`	Show or hide the label builder dialog box. Value is `FALSE` by default.	
	`TRUE`	Show the label builder dialog box.
	`FALSE`	Hide the label builder dialog box.

LBUILD_TITLE	A character string title that appears at the top of the label builder dialog box. Value is NULL by default.
LBUILD_WORK_SL	Not valid for tsol_lbuild_set() or tsol_lbuild_create(). The sensitivity label the end user is building. Value is updated to the end user's input when the end user selects the Update pushbutton or interactively chooses an option.
LBUILD_WORK_CLR	Not valid for tsol_lbuild_set() or tsol_lbuild_create(). The clearance the end user is building. Value is updated to the end user's input when the end user selects the Update pushbutton or interactively chooses an option.
LBUILD_X	The X position in pixels of the top-left corner of the label builder dialog box in relation to the top-left corner of the screen. By default the label builder dialog box is positioned in the middle of the screen.
LBUILD_Y	The Y position in pixels of the top-left corner of the label builder dialog box in relation to the top-left corner of the screen. By default the label builder dialog box is positioned in the middle of the screen.
LBUILD_LOWER_BOUND	The lowest classification (and related compartments and markings) available to the user as radio buttons for interactively building a label or clearance. This value is the user's minimum label.
LBUILD_UPPER_BOUND	The highest classification (and related compartments and markings) available to the user as radio buttons for interactively building a label or clearance. A supplied value should be within the user's accreditation range. If no value is specified, the value is the user's workspace sensitivity label, or if the executable has the PRIV_SYS_TRANS_LABEL privilege, the value is the user's clearance.
LBUILD_CHECK_AR	Check that the user-built label entered in the Update With field is within the user's accreditation range. A value of 1 means check, and a value of 0 means do not check. If checking is on and the label is out of range, an error message is raised to the end user.
LBUILD_VIEW	Use the internal or external label representation. Value is LBUILD_VIEW_EXTERNAL by default. LBUILD_VIEW_INTERNAL Use the internal names for the highest and lowest labels in the system: ADMIN_HIGH and ADMIN_LOW. LBUILD_VIEW_EXTERNAL Promote an ADMIN_LOW label to the next highest label, and demote an ADMIN_HIGH label to the next lowest label.

Return Values The `tsol_lbuild_get()` function returns –1 if it is unable to get the value.

The `tsol_lbuild_create()` function returns a variable of type `ModLabelData` that contains the information provided in the `tsol_lbuild_create()` input parameter list, default values for information not provided, and information on the widgets used by the label builder to create the user interface.

Examples **EXAMPLE 1** Create a Label Builder.

```
(ModLabelData *)lbldata = tsol_lbuild_create(widget0, callback_function,
     LBUILD_MODE, LBUILD_MODE_SL,
     LBUILD_TITLE, "Setting Sensitivity Label",
     LBUILD_VIEW, LBUILD_VIEW_INTERNAL,
     LBUILD_X, 200,
     LBUILD_Y, 200,
     LBUILD_USERFIELD, "Pathname:",
     LBUILD_SHOW, FALSE,
  NULL);
```

EXAMPLE 2 Query the Mode and Display the Label Builder.

These examples call the `tsol_lbuild_get()` function to query the mode being used, and call the `tsol_lbuild_set()` function so the label builder dialog box displays.

```
mode = (int)tsol_lbuild_get(lbldata, LBUILD_MODE );

tsol_lbuild_set(lbldata, LBUILD_SHOW, TRUE, NULL);
```

EXAMPLE 3 Destroy the ModLabelData Variable.

This example destroys the `ModLabelData` variable returned in the call to `tsol_lbuild_create()`.

```
tsol_lbuild_destroy(lbldata);
```

Files `/usr/dt/include/Dt/ModLabel.h`
Header file for label builder 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.

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), `label_encodings`(4), `attributes`(5)

Chapter 7, "Label Builder APIs," in *Oracle 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 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 *Oracle 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 ldexp, ldexpf, ldexpl – load exponent of a floating point number

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double ldexp(double x, int exp);

float ldexpf(float x, int exp);

long double ldexpl(long double x, int exp);
```

Description These functions computes the quantity x * 2^{exp}.

Return Values Upon successful completion, these functions return *x* multiplied by 2 raised to the power *exp*.

If these functions would cause overflow, a range error occurs and `ldexp()`, `ldexpf()`, and `ldexpl()` return ±`HUGE_VAL`, ±`HUGE_VALF`, and ±`HUGE_VALL` (according to the sign of *x*), respectively.

If *x* is NaN, a NaN is returned.

If *x* is ±0 or ±Inf, *x* is returned.

If *exp* is 0, *x* is returned.

Errors These functions will fail if:

Range Error — The result overflows.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, the overflow floating-point exception is raised.

The `ldexp()` function sets `errno` to `ERANGE` if the result overflows.

Usage An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. An application should either examine the return value or check the floating point exception flags to detect exceptions.

An application can also set `errno` to 0 before calling `ldexp()`. On return, if `errno` is non-zero, an error has occurred. The `ldexpf()` and `ldexpl()` functions do not set `errno`.

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

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

See Also frexp(3M), isnan(3M), modf(3M), attributes(5), standards(5)

Name ld_support, ld_atexit, ld_atexit64, ld_file, ld_file64, ld_input_done, ld_input_section, ld_input_section64, ld_open, ld_open64, ld_section, ld_section64, ld_start, ld_start64, ld_version – link-editor support functions

Synopsis

```
void ld_atexit(int status);

void ld_atexit64(int status);

void ld_file(const char *name, const Elf_Kind kind, int flags,
     Elf *elf);

void ld_file64(const char *name, const Elf_Kind kind, int flags,
     Elf *elf);

void ld_input_done(uint_t *flags);

void ld_input_section(const char *name, Elf32_Shdr **shdr,
     Elf32_Word sndx, Elf_Data *data, Elf *elf, uint_t *flags);

void ld_input_section64(const char *name, Elf64_Shdr **shdr,
     Elf64_Word sndx, Elf_Data *data, Elf *elf, uint_t *flags);

void ld_open(const char **pname, const char **fname, int *fd,
     int flags, Elf **elf, Elf *ref, size_t off, Elf_kind kind);

void ld_open64(const char **pname, const char **fname, int *fd,
     int flags, Elf **elf, Elf *ref, size_t off, Elf_kind kind);

void ld_section(const char *name, Elf32_Shdr shdr, Elf32_Word sndx,
     Elf_Data *data, Elf *elf);

void ld_section64(const char *name, Elf64_Shdr shdr, Elf64_Word sndx,
     Elf_Data *data, Elf *elf);

void ld_start(const char *name, const Elf32_Half type,
     const char *caller);

void ld_start64(const char *name, const Elf64_Half type,
     const char *caller);

void ld_version(uint_t version);
```

Description A link-editor support library is a user-created shared object offering one or more of these interfaces. These interfaces are called by the link-editor `ld(1)` at various stages of the link-editing process. See the *Linker and Libraries Guide* for a full description of the link-editor support mechanism.

See Also `ld(1)`

Linker and Libraries Guide

Name lgamma, lgammaf, lgammal, lgamma_r, lgammaf_r, lgammal_r, gamma, gammaf, gammal, gamma_r, gammaf_r, gammal_r – log gamma function

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

extern int signgam;

double lgamma(double x);

float lgammaf(float x);

long double lgammal(long double x);

double gamma(double x);

float gammaf(float x);

long double gammal(long double x);

double lgamma_r(double x, int *signgamp);

float lgammaf_r(float x, int *signgamp);

long double lgammal_r(long double x, int *signgamp);

double gamma_r(double x, int *signgamp);

float gammaf_r(float x, int *signgamp);

long double gammal_r(long double x, int *signgamp);
```

Description These functions return

$$\ln |\Gamma(x)|$$

where

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t}\,dt$$

for $x > 0$ and

$$\Gamma(x) = \pi/(\Gamma(1-x)\sin(\pi x))$$

for $x < 1$.

These functions use the external integer `signgam` to return the sign of `|~(x)` while `lgamma_r()` and `gamma_r()` use the user-allocated space addressed by `signgamp`.

Return Values Upon successful completion, these functions return the logarithmic gamma of *x*.

If *x* is a non-positive integer, a pole error occurs and these functions return `+HUGE_VAL`, `+HUGE_VALF`, and `+HUGE_VALL`, respectively.

If *x* is NaN, a NaN is returned.

If *x* is 1 or 2, +0 shall be returned.

If *x* is ±Inf, +Inf is returned.

Errors These functions will fail if:

Pole Error — The *x* argument is a negative integer or 0.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, then the divide-by-zero floating-point exception is raised.

Usage An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. An application should either examine the return value or check the floating point exception flags to detect exceptions.

In the case of `lgamma()`, do not use the expression `signgam*exp(lgamma(x))` to compute

$$g := \Gamma(x)$$

Instead compute `lgamma()` first:

```
lg = lgamma(x); g = signgam*exp(lg);
```

only after `lgamma()` has returned can `signgam` be correct. Note that `|~(x)` must overflow when *x* is large enough, underflow when $-x$ is large enough, and generate a division by 0 exception at the singularities *x* a nonpositive integer.

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

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

The `lgamma()`, `lgammaf()`, `lgammal()`, and `gamma()` functions are Standard. The `lgamma_r()`, `lgammaf_r()`, `lgammal_r()`, `gamma_r()`, `gammaf_r()`, and `gammal_r()`, functions are Stable.

The `lgamma()`, `lgammaf()`, `lgammal()`, `gamma()`, `gammaf()`, and `gammal()` functions are Unsafe in multithreaded applications. The `lgamma_r()`, `lgammaf_r()`, `lgammal_r()`, `gamma_r()`, `gammaf_r()`, and `gammal_r()` functions are MT-Safe and should be used instead.

See Also `exp(3M)`, `feclearexcept(3M)`, `fetestexcept(3M)`, `isnan(3M)`, `math.h(3HEAD)`, `attributes(5)`, `standards(5)`

Notes When compiling multithreaded applications, the `_REENTRANT` flag must be defined on the compile line. This flag should only be used in multithreaded applications.

Name lgrp_affinity_get, lgrp_affinity_set – get of set lgroup affinity

Synopsis

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

lgrp_affinity_t lgrp_affinity_get(idtype_t idtype, id_t id,
     lgrp_id_t lgrp);

int lgrp_affinity_set(idtype_t idtype, id_t id, lgrp_id_t lgrp,
     lgrp_affinity_t affinity);
```

Description The `lgrp_affinity_get()` function returns the affinity that the LWP or set of LWPs specified by the *idtype* and *id* arguments have for the given lgroup.

The `lgrp_affinity_set()` function sets the affinity that the LWP or set of LWPs specified by *idtype* and *id* have for the given lgroup. The lgroup affinity can be set to `LGRP_AFF_STRONG`, `LGRP_AFF_WEAK`, or `LGRP_AFF_NONE`.

If the *idtype* is `P_PID`, the affinity is retrieved for one of the LWPs in the process or set for all the LWPs of the process with process ID (PID) *id*. The affinity is retrieved or set for the LWP of the current process with LWP ID *id* if *idtype* is `P_LWPID`. If *id* is `P_MYID`, then the current LWP or process is specified.

The operating system uses the lgroup affinities as advice on where to run a thread and allocate its memory and factors this advice in with other constraints. Processor binding and processor sets can restrict which lgroups a thread can run on, but do not change the lgroup affinities.

Each thread can have an affinity for an lgroup in the system such that the thread will tend to be scheduled to run on that lgroup and allocate memory from there whenever possible. If the thread has affinity for more than one lgroup, the operating system will try to run the thread and allocate its memory on the lgroup for which it has the strongest affinity, then the next strongest, and so on up through some small, system-dependent number of these lgroup affinities. When multiple lgroups have the same affinity, the order of preference among them is unspecified and up to the operating system to choose. The lgroup with the strongest affinity that the thread can run on is known as its "home lgroup" (see `lgrp_home(3LGRP)`) and is usually the operating system's first choice of where to run the thread and allocate its memory.

There are different levels of affinity that can be specified by a thread for a particuliar lgroup. The levels of affinity are the following from strongest to weakest:

```
LGRP_AFF_STRONG         /* strong affinity */
LGRP_AFF_WEAK           /* weak affinity */
LGRP_AFF_NONE           /* no affinity */
```

The `LGRP_AFF_STRONG` affinity serves as a hint to the operating system that the calling thread has a strong affinity for the given lgroup. If this is the thread's home lgroup, the operating system will avoid rehoming it to another lgroup if possible. However, dynamic reconfiguration, processor offlining, processor binding, and processor set binding and

manipulation are examples of events that can cause the operating system to change the thread's home lgroup for which it has a strong affinity.

The LGRP_AFF_WEAK affinity is a hint to the operating system that the calling thread has a weak affinity for the given lgroup. If a thread has a weak affinity for its home lgroup, the operating system interpets this to mean that thread does not mind whether it is rehomed, unlike LGRP_AFF_STRONG. Load balancing, dynamic reconfiguration, processor binding, or processor set binding and manipulation are examples of events that can cause the operating system to change a thread's home lgroup for which it has a weak affinity.

The LGRP_AFF_NONE affinity signifies no affinity and can be used to remove a thread's affinity for a particuliar lgroup. Initially, each thread has no affinity to any lgroup. If a thread has no lgroup affinities set, the operating system chooses a home lgroup for the thread with no affinity set.

Return Values Upon successful completion, `lgrp_affinity_get()` returns the affinity for the given lgroup.

Upon successful completion, `lgrp_affinity_set()` return 0.

Otherwise, both functions return −1 and set `errno` to indicate the error.

Errors The `lgrp_affinity_get()` and `lgrp_affinity_set()` functions will fail if:

`EINVAL` The specified lgroup, affinity, or ID type is not valid.

`EPERM` The effective user of the calling process does not have appropriate privileges, and its real or effective user ID does not match the real or effective user ID of one of the LWPs.

`ESRCH` The specified lgroup or LWP(s) was not found.

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

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

See Also `lgrp_home(3LGRP)`, `liblgrp(3LIB)`, `attributes(5)`

Name lgrp_children – get children of given lgroup

Synopsis

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

int lgrp_children(lgrp_cookie_t cookie, lgrp_id_t parent,
     lgrp_id_t *lgrp_array, uint_t lgrp_array_size);
```

Description The `lgrp_children()` function takes a *cookie* representing a snapshot of the lgroup hierarchy retrieved from `lgrp_init(3LGRP)` and returns the number of lgroups that are children of the specified lgroup. If the *lgrp_array* and *lgrp_array_size* arguments are non-null, the array is filled with as many of the children lgroup IDs as will fit, given the size of the array.

Return Values – returns the number of child lgroup IDs. Otherwise, it returns –1 and sets `errno` to indicate the error.

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

`EINVAL` The specified lgroup ID is not valid or the cookie is invalid.

`ESRCH` The specified lgroup ID was not found.

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

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

See Also `lgrp_init(3LGRP)`, `lgrp_nlgrps(3LGRP)`, `lgrp_parents(3LGRP)`, `liblgrp(3LIB)`, `attributes(5)`

Name lgrp_cookie_stale – determine whether snapshot of lgroup hierarchy is stale

Synopsis

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

int lgrp_cookie_stale(lgrp_cookie_t cookie);
```

Description The `lgrp_cookie_stale()` function takes a *cookie* representing the snapshot of the lgroup hierarchy obtained from `lgrp_init(3LGRP)` and returns whether it is stale. The snapshot can become out-of-date for a number of reasons depending on its view. If the snapshot was taken with `LGRP_VIEW_OS`, changes in the lgroup hierarchy from dynamic reconfiguration, CPU on/offline, or other conditions can cause the snapshot to become out-of-date. A snapshot taken with `LGRP_VIEW_CALLER` can be affected by the caller's processor set binding and changes in its processor set itself, as well as changes in the lgroup hierarchy.

If the snapshot needs to be updated, `lgrp_fini(3LGRP)` should be called with the old cookie and `lgrp_init()` should be called to obtain a new snapshot.

Return Values Upon successful completion, `lgrp_cookie_stale()` returns whether the cookie is stale. Otherwise, it returns −1 and sets `errno` to indicate the error.

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

`EINVAL` The cookie is not valid.

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

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

See Also `lgrp_init(3LGRP)`, `lgrp_fini(3LGRP)`, `lgrp_view(3LGRP)`, `liblgrp(3LIB)`, `attributes(5)`

Name lgrp_cpus – get CPU IDs contained in specified lgroup

Synopsis

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

int lgrp_cpus(lgrp_cookie_t cookie, lgrp_id_t lgrp,
     processorid_t *cpuids, uint_t count, int content);
```

Description The `lgrp_cpus()` function takes a *cookie* representing a snapshot of the lgroup hierarchy obtained from `lgrp_init(3LGRP)` and returns the number of CPUs in the lgroup specified by *lgrp*. If both the *cpuids*[] argument is non-null and the count is non-zero, `lgrp_cpus()` stores up to the specified count of CPU IDs into the *cpuids*[] array.

The *content* argument should be set to one of the following values to specify whether the direct contents or everything in this lgroup should be returned:

```
LGRP_CONTENT_ALL        /* everything in this lgroup */
LGRP_CONTENT_DIRECT     /* directly contained in lgroup */
LGRP_CONTENT_HIERARCHY  /* everything within this hierarchy (for
                           compatibility only, use LGRP_CONTENT_ALL) */
```

The `LGRP_CONTENT_HIERARCHY` value can still be used, but is being replaced by `LGRP_CONTENT_ALL`.

Return Values Upon successful completion, the number of CPUs in the given lgroup is returned. Otherwise, −1 is returned and `errno` is set to indicate the error.

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

`EINVAL` The specified cookie, lgroup ID, or one of the flags is not valid.

`ESRCH` The specified lgroup ID was not found.

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

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

See Also `lgrp_init(3LGRP)`, `lgrp_mem_size(3LGRP)`, `lgrp_resources(3LGRP)`, `liblgrp(3LIB)`, `attributes(5)`

Name lgrp_fini – finished using lgroup interface

Synopsis

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

int lgrp_fini(lgrp_cookie_t cookie);
```

Description The `lgrp_fini()` function takes a *cookie*, frees the snapshot of the lgroup hierarchy created by `lgrp_init(3LGRP)`, and cleans up anything else set up by `lgrp_init()`. After this function is called, any memory allocated and returned by the lgroup interface might no longer be valid and should not be used.

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

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

`EINVAL` The *cookie* is not valid.

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

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

See Also `lgrp_init(3LGRP)`, `lgrp_cookie_stale(3LGRP)`, `liblgrp(3LIB)`, `attributes(5)`

Name lgrp_home – get home lgroup

Synopsis

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

lgrp_id_t lgrp_home(idtype_t idtype, id_t id);
```

Description The `lgrp_home()` function returns the ID of the home lgroup for the given process or thread. A thread can have an affinity for an lgroup in the system such that the thread will tend to be scheduled to run on that lgroup and allocate memory from there whenever possible. The lgroup with the strongest affinity that the thread can run on is known as the "home lgroup" of the thread. If the thread has no affinity for any lgroup that it can run on, the operating system will choose a home for it.

The *idtype* argument should be `P_PID` to specify a process and the *id* argument should be its process ID. Otherwise, the *idtype* argument should be `P_LWPID` to specify a thread and the *id* argument should be its LWP ID. The value `P_MYID` can be used for the *id* argument to specify the current process or thread.

Return Values Upon successful completion, `lgrp_home()` returns the ID of the home lgroup of the specified process or thread. Otherwise, −1 is returned and `errno` is set to indicate the error.

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

`EINVAL` The ID type is not valid.

`EPERM` The effective user of the calling process does not have appropriate privileges, and its real or effective user ID does not match the real or effective user ID of one of the threads.

`ESRCH` The specified process or thread was not found.

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

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

See Also `lgrp_affinity_get(3LGRP)`, `lgrp_init(3LGRP)`, `attributes(5)`

Name lgrp_init – initialize lgroup interface

Synopsis

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

lgrp_cookie_t lgrp_init(lgrp_view_t view);
```

Description The `lgrp_init()` function initializes the lgroup interface and takes a snapshot of the lgroup hierarchy with the given *view*. If the given *view* is `LGRP_VIEW_CALLER`, the snapshot contains only the resources that are available to the caller (for example, with respect to processor sets). When the *view* is `LGRP_VIEW_OS`, the snapshot contains what is available to the operating system.

Given the *view*, `lgrp_init()` returns a cookie representing this snapshot of the lgroup hierarchy. This cookie should be used with other routines in the lgroup interface needing the lgroup hierarchy. The `lgrp_fini(3LGRP)` function should be called with the cookie when it is no longer needed.

The lgroup hiearchy represents the latency topology of the machine. The hierarchy is simplified to be a tree and can be used to find the nearest resources.

The lgroup hiearchy consists of a root lgroup, which is the maximum bounding locality group of the system, contains all the CPU and memory resources of the machine, and may contain other locality groups that contain CPUs and memory within a smaller locality. The leaf lgroups contain resources within the smallest latency.

The resources of a given lgroup come directly from the lgroup itself or from leaf lgroups contained within the lgroup. Leaf lgroups directly contain their own resources and do not encapsulate any other lgroups.

The lgroup hierarchy can be used to find the nearest resources. From a given lgroup, the closest resources can be found in the lgroup itself. After that, the next nearest resources can be found in its parent lgroup, and so on until the root lgroup is reached where all the resources of the machine are located.

Return Values Upon successful completion, `lgrp_init()` returns a cookie. Otherwise it returns `LGRP_COOKIE_NONE` and sets `errno` to indicate the error.

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

`EINVAL` The view is not valid.

`ENOMEM` There was not enough memory to allocate the snapshot of the lgroup hierarchy.

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

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

See Also lgrp_children(3LGRP), lgrp_cookie_stale(3LGRP), lgrp_cpus(3LGRP), lgrp_fini(3LGRP), lgrp_mem_size(3LGRP), lgrp_nlgrps(3LGRP), lgrp_parents(3LGRP), lgrp_resources(3LGRP), lgrp_root(3LGRP), lgrp_view(3LGRP), liblgrp(3LIB), attributes(5)

Name lgrp_latency, lgrp_latency_cookie – get latency between two lgroups

Synopsis

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

int lgrp_latency_cookie(lgrp_cookie_t cookie, lgrp_id_t from,
     lgrp_id_t to, lgrp_lat_between_t between);

int lgrp_latency(lgrp_id_t from, lgrp_id_t to);
```

Description The `lgrp_latency_cookie()` function takes a cookie representing a snapshot of the lgroup hierarchy obtained from `lgrp_init(3LGRP)` and returns the latency value between a hardware resource in the *from* lgroup to a hardware resource in the *to* lgroup. If *from* is the same lgroup as *to*, the latency value within that lgroup is returned.

The *between* argument should be set to the following value to specify between which hardware resources the latency should be measured:

```
LGRP_LAT_CPU_TO_MEM     /* latency from CPU to memory */
```

The latency value is defined by the operating system and is platform-specific. It can be used only for relative comparison of lgroups on the running system. It does not necessarily represent the actual latency between hardware devices, and it might not be applicable across platforms.

The `lgrp_latency()` function is similiar to the `lgrp_latency_cookie()` function, but returns the latency between the given lgroups at the given instant in time. Since lgroups can be freed and reallocated, this function might not be able to provide a consistent answer across calls. For that reason, the `lgrp_latency_cookie()` function should be used in its place.

Return Values Upon successful completion, the latency value is returned. Otherwise −1 is returned and `errno` is set to indicate the error.

Errors The `lgrp_latency_cookie()` and `lgrp_latency()` functions will fail if:

`EINVAL` The specified cookie, lgroup ID, or value given for the *between* argument is not valid.

`ESRCH` The specified lgroup ID was not found, the *from* lgroup does not contain any CPUs, or the *to* lgroup does not have any memory.

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

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

See Also lgrp_init(3LGRP), lgrp_parents(3LGRP), lgrp_children(3LGRP), liblgrp(3LIB), attributes(5)

Name lgrp_mem_size – return the memory size of the given lgroup

Synopsis

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

lgrp_mem_size_t lgrp_mem_size(lgrp_cookie_t cookie, lgrp_id_t lgrp,
     int type, int content);
```

Description The `lgrp_mem_size()` function takes a *cookie* representing a snapshot of the lgroup hierarchy. The *cookie* was obtained by calling `lgrp_init(3LGRP)`. The `lgrp_mem_size()` function returns the memory size of the given lgroup in bytes. The *type* argument should be set to one of the following values:

```
LGRP_MEM_SZ_FREE                /* free memory */
LGRP_MEM_SZ_INSTALLED           /* installed memory */
```

The *content* argument should be set to one of the following values to specify whether the direct contents or everything in this lgroup should be returned:

```
LGRP_CONTENT_ALL        /* everything in this lgroup */
LGRP_CONTENT_DIRECT     /* directly contained in lgroup */
LGRP_CONTENT_HIERARCHY  /* everything within this hierarchy (for */
                           compatibility only, use LGRP_CONTENT_ALL) */
```

The `LGRP_CONTENT_HIERARCHY` value can still be used, but is being replaced by `LGRP_CONTENT_ALL`.

The total sizes include all the memory in the lgroup including its children, while the others reflect only the memory contained directly in the given lgroup.

Return Values Upon successful completion, the size in bytes is returned. Otherwise, −1 is returned and `errno` is set to indicate the error.

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

`EINVAL` The specified cookie, lgroup ID, or one of the flags is not valid.

`ESRCH` The specified lgroup ID was not found.

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

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

See Also lgrp_init(3LGRP), lgrp_cpus(3LGRP), lgrp_resources(3LGRP), liblgrp(3LIB), attributes(5)

Name lgrp_nlgrps – get number of lgroups

Synopsis

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

int lgrp_nlgrps(lgrp_cookie_t cookie);
```

Description The `lgrp_nlgrps()` function takes a *cookie* representing a snapshot of the lgroup hierarchy obtained from `lgrp_init(3LGRP)`. It returns the number of lgroups in the hierarchy where the number is always at least one.

Return Values Upon successful completion, `lgrp_nlgrps()` returns the number of lgroups in the system. Otherwise, it returns −1 and sets `errno` to indicate the error.

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

`EINVAL` The *cookie* is not valid.

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

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

See Also `lgrp_children(3LGRP)`, `lgrp_init(3LGRP)`, `lgrp_parents(3LGRP)`, `liblgrp(3LIB)`, `attributes(5)`

Name lgrp_parents – get parents of given lgroup

Synopsis

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

int lgrp_parents(lgrp_cookie_t cookie, lgrp_id_t child,
     lgrp_id_t *lgrp_array, uint_t lgrp_array_size);
```

Description The `lgrp_parents()` function takes a *cookie* representing a snapshot of the lgroup hierarchy obtained from `lgrp_init(3LGRP)` and returns the number of parent lgroups of the specified lgroup. If *lgrp_array* is non-null and the *lgrp_array_size* is non-zero, the array is filled with as many of the parent lgroup IDs as will fit given the size of the array. For the root lgroup, the number of parents returned is 0 and the *lgrp_array* argument is not filled in.

Return Values Upon successful completion, `lgrp_parents()` returns the number of parent lgroup IDs. Otherwise, –1 is returned and `errno` is set to indicate the error.

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

`EINVAL` The specified cookie or lgroup ID is not valid.

`ESRCH` The specified lgroup ID was not found.

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

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

See Also `lgrp_children(3LGRP)`, `lgrp_init(3LGRP)`, `lgrp_nlgrps(3LGRP)`, `liblgrp(3LIB)`, `attributes(5)`

Name lgrp_resources – get lgroup resources of given lgroup

Synopsis

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

int lgrp_resources(lgrp_cookie_t cookie, lgrp_id_t lgrp,
     lgrp_id_t *lgrpids, uint_t count, lgrp_rsrc_t type);
```

Description The `lgrp_resources()` function takes a cookie representing a snapshot of the lgroup hierarchy obtained from `lgrp_init(3LGRP)` and returns the number of resources in the lgroup specified by *lgrp*. The resources are represented by a set of lgroups in which each lgroup directly contains CPU and/or memory resources.

The *type* argument should be set to one of the following values to specify whether the CPU or memory resources should be returned:

```
LGRP_RSRC_CPU      /* CPU resources */
LGRP_RSRC_MEM      /* Memory resources */
```

If the *lgrpids*[] argument is non-null and the *count* argument is non-zero, `lgrp_resources()` stores up to the specified count of lgroup IDs into the *lgrpids*[] array.

Return Values Upon successful completion, `lgrp_resources()` returns the number of lgroup resources. Otherwise, -1 is returned and `errno` is set to indicate the error.

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

`EINVAL` The specified cookie, lgroup ID, or type is not valid.

`ESRCH` The specified lgroup ID was not found.

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

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

See Also `lgrp_children(3LGRP)`, `lgrp_init(3LGRP)`, `lgrp_parents(3LGRP)`, `liblgrp(3LIB)`, `attributes(5)`

Name lgrp_root – return root lgroup ID

Synopsis

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

lgrp_id_t lgrp_root(lgrp_cookie_t cookie);
```

Description The `lgrp_root()` function returns the root lgroup ID.

Return Values Upon successful completion, `lgrp_root()` returns the lgroup ID of the root lgroup. Otherwise, it returns −1 and sets `errno` to indicate the error.

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

`EINVAL` The *cookie* is not valid.

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

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

See Also `lgrp_children(3LGRP)`, `lgrp_init(3LGRP)`, `lgrp_nlgrps(3LGRP)`, `lgrp_parents(3LGRP)`, `liblgrp(3LIB)`, `attributes(5)`

Name lgrp_version – coordinate library and application versions

Synopsis

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

int lgrp_version(const int version);
```

Description The `lgrp_version()` function takes an interface version number, *version*, as an argument and returns an lgroup interface version. The *version* argument should be the value of `LGRP_VER_CURRENT` bound to the application when it was compiled or `LGRP_VER_NONE` to find out the current lgroup interface version on the running system.

Return Values If *version* is still supported by the implementation, then `lgrp_version()` returns the requested version. If `LGRP_VER_NONE` is returned, the implementation cannot support the requested version. The application should be recompiled and might require further changes.

If *version* is `LGRP_VER_NONE`, `lgrp_version()` returns the current version of the library.

Examples **EXAMPLE 1** Test whether the version of the interface used by the caller is supported.

The following example tests whether the version of the interface used by the caller is supported:

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

if (lgrp_version(LGRP_VER_CURRENT) != LGRP_VER_CURRENT) {
    fprintf(stderr, "Built with unsupported lgroup interface %d\n",
        LGRP_VER_CURRENT);
    exit (1);
}
```

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

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

See Also `lgrp_init(3LGRP)`, `liblgrp(3LIB)`, `attributes(5)`

Name lgrp_view – get view of lgroup hierarchy

Synopsis

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

lgrp_view_t lgrp_view(lgrp_cookie_t cookie);
```

Description The `lgrp_view()` function takes a *cookie* representing the snapshot of the lgroup hierarchy obtained from `lgrp_init(3LGRP)` and returns the snapshot's view of the lgroup hierarchy.

If the given view is `LGRP_VIEW_CALLER`, the snapshot contains only the resources that are available to the caller (such as those with respect to processor sets). When the view is `LGRP_VIEW_OS`, the snapshot contains what is available to the operating system.

Return Values Upon succesful completion, `lgrp_view()` returns the view for the snapshot of the lgroup hierarchy represented by the given cookie. Otherwise, −1 is returned and `errno` is set to indicate the error.

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

`EINVAL` The *cookie* is not valid.

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

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

See Also `lgrp_cookie_stale(3LGRP)`, `lgrp_fini(3LGRP)`, `lgrp_init(3LGRP)`, `liblgrp(3LIB)`, `attributes(5)`

Name libpicl – PICL interface library

Synopsis

```
cc [ flag . . . ] file . . . -lpicl [ library . . . ]
#include <picl.h>
```

Description The PICL interface is the platform-independent interface for clients to access the platform information. The set of functions and data structures of this interface are defined in the <picl.h> header.

The information published through PICL is organized in a tree, where each node is an instance of a well-defined PICL class. The functions in the PICL interface allow the clients to access the properties of the nodes.

The name of the base PICL class is `picl`, which defines a basic set of properties that all nodes in the tree must possess. The following table shows the property set of a `picl` class node.

Property Name	Property Value
`name`	The name of the node
`_class`	The PICL class name of the node
`_parent`	Node handle of the parent node
`_child`	Node handle of the first child node
`_peer`	Node handle of the next peer node

Property names with a a leading underscore ('_') are reserved for use by the PICL framework. The property names `_class`, `_parent`, `_child`, and `_peer` are reserved names of the PICL framework, and are used to refer to a node's parent, child, and peer nodes, respectively. A client shall access a reserved property by their names only as they do not have an associated handle. The property `name` is not a reserved property, but a mandatory property for all nodes.

Properties are classified into different types. Properties of type integer, unsigned-integer, and float have integer, unsigned integer, and floating-point values, respectively. A `table` property type has the handle to a table as its value. A table is a matrix of properties. A `reference` property type has a handle to a node in the tree as its value. A `reference` property may be used to establish an association between any two nodes in the tree. A `timestamp` property type has the value of time in seconds since Epoch. A `bytearray` property type has an array of bytes as its value. A `charstring` property type has a nul ('\0') terminated sequence of ASCII characters. The size of a property specifies the size of its value in bytes. A `void` property type denotes a property that exists but has no value.

The following table lists the different PICL property types enumerated in `picl_prop_type_t`.

Property Type	Property Value
PICL_PTYPE_VOID	None
PICL_PTYPE_INT	Is an integer
PICL_PTYPE_UNSIGNED_INT	Is an unsigned integer
PICL_PTYPE_FLOAT	Is a floating-point number
PICL_PTYPE_REFERENCE	Is a PICL node handle

Reference Property Naming Convention

Reference properties may be used by plug-ins to publish properties in nodes of different classes. To make these property names unique, their names must be prefixed by *_picl_class_name_*, where *picl_class_name* is the class name of the node referenced by the property. Valid PICL class names are combinations of uppercase and lowercase letters 'a' through 'z', digits '0' through '9', and '-' (minus) characters. The string that follows the '*_picl_class_name_*' portion of a reference property name may be used to indicate a specific property in the referenced class, when applicable.

Property Information

The information about a node's property that can be accessed by PICL clients is defined by the `picl_propinfo_t` structure.

```
typedef struct {
    picl_prop_type_t  type;           /* property type */
    unsigned int      accessmode;     /* read, write */
    size_t            size;           /* item size or
                                         string size */
    char              name[PICL_PROPNAMELEN_MAX];
} picl_propinfo_t;
```

The `type` member specifies the property value type and the `accessmode` specifies the allowable access to the property. The plug-in module that adds the property to the PICL tree also sets the access mode of that property. The volatile nature of a property created by the plug-in is not visible to the PICL clients. The `size` member specifies the number of bytes occupied by the property's value. The maximum allowable size of property value is `PICL_PROPSIZE_MAX`, which is set to 512KB.

Property Access Modes

The plug-in module may publish a property granting a combination of the following access modes to the clients:

```
#define   PICL_READ   0x1   /* read permission */
#define   PICL_WRITE  0x2   /* write permission */
```

Property Names

The maximum length of the name of any property is specified by `PICL_PROPNAMELEN_MAX`.

Class Names

The maximum length of a PICL class name is specified by `PICL_CLASSNAMELEN_MAX`.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also libpicl(3LIB), attributes(5)

Name libpicltree – PTree and Plug-in Registration interface library

Synopsis

```
cc [flag ...] file ... -lpicltree [library ...]
#include <picltree.h>
```

Description The PTree interface is the set of functions and data structures to access and manipulate the PICL tree. The daemon and the plug-in modules use the PTree interface.

The Plug-in Registration interface is used by the plug-in modules to register themselves with the daemon.

The plug-in modules create the nodes and properties of the tree. At the time of creating a property, the plug-ins specify the property information in the `ptree_propinfo_t` structure defined as:

```
typedef struct {
    int             version;    /* version */
    picl_propinfo_t piclinfo;   /* info to clients */
    int             (*read)(ptree_rarg_t *arg, void *buf);
                                /* read access function for */
                                /* volatile prop */
    int             (*write)(ptree_warg_t *arg, const void *buf);
                                /* write access function for */
                                /* volatile prop */
} ptree_propinfo_t;
```

See `libpicl(3PICL)` for more information on PICL tree nodes and properties.

The maximum size of a property value cannot exceed `PICL_PROPSIZE_MAX`. It is currently set to 512KB.

Volatile Properties In addition to `PICL_READ` and `PICL_WRITE` property access modes, the plug-in modules specify whether a property is volatile or not by setting the bit `PICL_VOLATILE`.

```
#define   PICL_VOLATILE   0x4
```

For a volatile property, the plug-in module provides the access functions to read and/or write the property in the `ptree_propinfo_t` argument passed when creating the property.

The daemon invokes the access functions of volatile properties when clients access their values. Two arguments are passed to the read access functions. The first argument is a pointer to `ptree_rarg_t`, which contains the handle of the node, the handle of the accessed property and the credentials of the caller. The second argument is a pointer to the buffer where the value is to be copied.

```
typedef struct {
         picl_nodehdl_t nodeh;
         picl_prophdl_t proph;
         door_cred_t    cred;
} ptree_rarg_t;
```

The prototype of the read access function for volatile property is:

```
int read(ptree_rarg_t *rarg, void *buf);
```

The read function returns `PICL_SUCCESS` to indicate successful completion.

Similarly, when a write access is performed on a volatile property, the daemon invokes the write access function provided by the plug-in for that property and passes it two arguments. The first argument is a pointer to `ptree_warg_t`, which contains the handle to the node, the handle of the accessed property and the credentials of the caller. The second argument is a pointer to the buffer containing the value to be written.

```
typedef struct {
        picl_nodehdl_t  nodeh;
        picl_prophdl_t  proph;
        door_cred_t     cred;
} ptree_warg_t;
```

The prototype of the write access function for volatile property is:

```
int write(ptree_warg_t *warg, const void *buf);
```

The write function returns `PICL_SUCCESS` to indicate successful completion.

For all volatile properties, the 'size' of the property must be specified to be the maximum possible size of the value. The maximum size of the value cannot exceed `PICL_PROPSIZE_MAX`. This allows a client to allocate a sufficiently large buffer before retrieving a volatile property's value

Plug-in Modules

Plug-in modules are shared objects that are located in well-known directories for the daemon to locate and load them. Plug-in module's are located in the one of the following plug-in directories depending on the plaform-specific nature of the data they collect and publish.

```
/usr/platform/picl/plugins/‘uname -i‘/
/usr/platform/picl/plugins/‘uname -m‘/
/usr/lib/picl/plugins/
```

A plug-in module may specify its dependency on another plug-in module using the `-l` linker option. The plug-ins are loaded by the PICL daemon using `dlopen(3C)` according to the specified dependencies. Each plug-in module must define a `.init` section, which is executed when the plug-in module is loaded, to register themselves with the daemon. See `picld_plugin_register(3PICLTREE)` for more information on plug-in registration.

The plug-in modules may use the `picld_log(3PICLTREE)` function to log their messages to the system log file.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `libpicl`(3PICL), `libpicltree`(3LIB), `picld_log`(3PICLTREE), `picld_plugin_register`(3PICLTREE), `attributes`(5)

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 llrint, llrintf, llrintl – round to nearest integer value using current rounding direction

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

long long llrint(double x);

long long llrintf(float x);

long long llrintl(long double x);
```

Description These functions round their argument to the nearest integer value, rounding according to the current rounding direction.

Return Values Upon successful completion, these functions return the rounded integer value.

If *x* is NaN, a domain error occurs and an unspecified value is returned.

If *x* is +Inf, a domain error occurs and an unspecified value is returned.

If *x* is -Inf, a domain error occurs and an unspecified value is returned.

If the correct value is positive and too large to represent as a `long long`, a domain error occurs and an unspecified value is returned.

If the correct value is negative and too large to represent as a `long long`, a domain error occurs and an unspecified value is returned.

Errors These functions will fail if:

Domain Error — The *x* argument is NaN or ±Inf, or the correct value is not representable as an integer.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, then the invalid floating-point exception will be raised.

Usage An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. An application should either examine the return value or check the floating point exception flags to detect exceptions.

These functions provide floating-to-integer conversions. They round according to the current rounding direction. If the rounded value is outside the range of the return type, the numeric result is unspecified and the invalid floating-point exception is raised. When they raise no other floating-point exception and the result differs from the argument, they raise the inexact floating-point exception.

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

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

See Also feclearexcept(3M), fetestexcept(3M), lrint(3M), math.h(3HEAD), attributes(5), standards(5)

Name llround, llroundf, llroundl – round to nearest integer value

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

long long llround(double x);

long long llroundf(float x);

long long llroundl(long double x);
```

Description These functions rounds their argument to the nearest integer value, rounding halfway cases away from 0 regardless of the current rounding direction.

Return Values Upon successful completion, these functions return the rounded integer value.

If *x* is NaN, a domain error occurs and an unspecified value is returned.

If *x* is +Inf, a domain error occurs and an unspecified value is returned.

If *x* is -Inf, a domain error occurs and an unspecified value is returned.

If the correct value is positive and too large to represent as a `long long`, a domain error occurs and an unspecified value is returned.

If the correct value is negative and too large to represent as a `long long`, a domain error occurs and an unspecified value is returned.

Errors These functions will fail if:

Domain Error — The *x* argument is NaN or ±Inf, or the correct value is not representable as an integer.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, then the invalid floating-point exception will be raised.

Usage An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. An application should either examine the return value or check the floating point exception flags to detect exceptions.

These functions differ from the `llrint(3M)` functions in that the default rounding direction for the `llround()` functions round halfway cases away from 0 and need not raise the inexact floating-point exception for non-integer arguments that round to within the range of the return type.

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

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

See Also feclearexcept(3M), fetestexcept(3M), llrint(3M), lrint(3M), lround(3M), math.h(3HEAD), attributes(5), standards(5)

Name log10, log10f, log10l – base 10 logarithm function

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double log10(double x);

float log10f(float x);

long double log10l(long double x);
```

Description These functions compute the base 10 logarithm of x, $\log_{10}(x)$.

Return Values Upon successful completion, `log10()` returns the base 10 logarithm of x.

If x is ±0, a pole error occurs and `log10()`, `log10f()`, and `log10l()` return −`HUGE_VAL`, −`HUGE_VALF`, and −`HUGE_VALL`, respectively.

For finite values of x that are less than 0, or if x is −Inf, a domain error occurs and a NaN is returned.

If x is NaN, a NaN is returned.

If x is 1, +0 is returned.

If x is +Inf, x is returned.

For exceptional cases, matherr(3M) tabulates the values to be returned by `log10()` as specified by SVID3 and XPG3.

Errors These functions will fail if:

Domain Error — The finite value of x is negative, or x is -Inf.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, the invalid floating-point exception is raised.

The `log10()` function sets `errno` to `EDOM` if the value of x is negative.

Pole Error — The value of x is 0.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, the divide-by-zero floating-point exception is raised.

Usage An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. An application should either examine the return value or check the floating point exception flags to detect exceptions.

An application can also set `errno` to 0 before calling `log10()`. On return, if `errno` is non-zero, an error has occurred. The `log10f()` and `log10l()` functions do not set `errno`.

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

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

See Also feclearexcept(3M), fetestexcept(3M), isnan(3M), log(3M), math.h(3HEAD), matherr(3M), pow(3M), attributes(5), standards(5)

Name log1p, log1pf, log1pl – compute natural logarithm

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double log1p(double x);

float log1pf(float x);

long double log1pl(long double x);
```

Description These functions compute $\log_e(1.0 + x)$.

Return Values Upon successful completion, these functions return the natural logarithm of $1.0 + x$.

If x is −1, a pole error occurs and `log1p()`, `log1pf()`, and `log1pl()` return −`HUGE_VAL`, −`HUGE_VALF`, and −`HUGE_VALL`, respectively.

For finite values of x that are less than −1, or if x is −Inf, a domain error occurs and a NaN is returned.

If x is NaN, a NaN is returned.

If x is ±0 or +Inf, x is returned.

For exceptional cases, matherr(3M) tabulates the values to be returned by `log1p()` as specified by SVID3 and XPG3.

Errors These functions will fail if:

Domain Error — The finite value of x is less than −1, or x is -Inf.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, the invalid floating-point exception is raised.

The `log1p()` function sets `errno` to `EDOM` if the value of x is less than −1.

Pole Error — The value of x is −1.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, the divide-by-zero floating-point exception is raised.

Usage An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. An application should either examine the return value or check the floating point exception flags to detect exceptions.

An application can also set `errno` to 0 before calling `log1p()`. On return, if `errno` is non-zero, an error has occurred. The `log1pf()` and `log1pl()` functions do not set `errno`.

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

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

See Also feclearexcept(3M), fetestexcept(3M), log(3M), math.h(3HEAD), matherr(3M), attributes(5), standards(5)

Name log2, log2f, log2l – compute base 2 logarithm functions

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double log2(double x);

float log2f(float x);

long double log2l(long double x);
```

Description These functions compute the base 2 logarithm of their argument *x*, `log2`(*x*).

Return Values Upon successful completion, these functions return the base 2 logarithm of *x*.

If *x* is ±0, a pole error occurs and `log2()`, `log2f()`, and `log2l()` return −`HUGE_VAL`, −`HUGE_VALF`, and −`HUGE_VALL`, respectively.

For finite values of *x* that are less than 0, or if *x* is −Inf a domain error occurs and a NaN is returned.

If *x* is NaN, a NaN is returned.

If *x* is 1, +0 is returned.

If *x* is +Inf, *x* is returned.

Errors These functions will fail if:

Domain Error — The finite value of *x* is less than 0, or *x* is −Inf.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, then the invalid floating-point exception is raised.

Pole Error — The value of *x* is 0.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, then the divide-by-zero floating-point exception is raised.

Usage An application wanting to check for exceptions should call `feclearexcept`(`FE_ALL_EXCEPT`) before calling these functions. On return, if `fetestexcept`(`FE_INVALID` | `FE_DIVBYZERO` | `FE_OVERFLOW` | `FE_UNDERFLOW`) is non-zero, an exception has been raised. An application should either examine the return value or check the floating point exception flags to detect exceptions.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also feclearexcept(3M), fetestexcept(3M), log(3M), math.h(3HEAD), attributes(5), standards(5)

Name log, logf, logl – natural logarithm function

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double log(double x);

float logf(float x);

long double logl(long double x);
```

Description These functions compute the natural logarithm of their argument x, $\log_e(x)$.

Return Values Upon successful completion, `log()` returns the natural logarithm of x.

If x is ±0, a pole error occurs and `log()`, `logf()`, and `logl()` return −`HUGE_VAL`, −`HUGE_VALF`, and −`HUGE_VALL`, respectively.

For finite values of x that are less than 0, or if x is −Inf, a domain error occurs and a NaN is returned.

If x is NaN, a NaN is returned.

If x is 1, +0 is returned.

If x is +Inf, x is returned.

For exceptional cases, `matherr(3M)` tabulates the values to be returned by `log()` as specified by SVID3 and XPG3.

Errors These functions will fail if:

Domain Error — The finite value of x is negative, or x is -Inf.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, the invalid floating-point exception is raised.

The `log()` function sets `errno` to `EDOM` if the value of x is negative.

Pole Error — The value of x is 0.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, the divide-by-zero floating-point exception is raised.

Usage An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. An application should either examine the return value or check the floating point exception flags to detect exceptions.

An application can also set `errno` to 0 before calling `log()`. On return, if `errno` is non-zero, an error has occurred. The `logf()` and `logl()` functions do not set `errno`.

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

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

See Also exp(3M), feclearexcept(3M), fetestexcept(3M), isnan(3M), log10(3M), log1p(3M), math.h(3HEAD), matherr(3M), attributes(5), standards(5)

Name logb, logbf, logbl – radix-independent exponent

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double logb(double x);

float logbf(float x);

long double logbl(long double x);

cc [ flag... ] file... -lm [ library... ]
#include <math.h>

double logb(double x);

float logbf(float x);

long double logbl(long double x);
```

Description These functions compute the exponent of x, which is the integral part of $\log_r |x|$, as a signed floating point value, for non-zero x, where r is the radix of the machine's floating-point arithmetic, which is the value of `FLT_RADIX` defined in the <`float.h`> header.

Return Values Upon successful completion, these functions return the exponent of x.

If x is subnormal:

- For SUSv3–conforming applications compiled with the `c99` compiler driver (see `standards(5)`), the exponent of x as if x were normalized is returned.
- Otherwise, if compiled with the `cc` compiler driver, −1022, −126, and −16382 are returned for `logb()`, `logbf()`, and `logbl()`, respectively.

If x is ±0, a pole error occurs and `logb()`, `logbf()`, and `logbl()` return −`HUGE_VAL`, −`HUGE_VALF`, and −`HUGE_VALL`, respectively.

If x is NaN, a NaN is returned.

If x is ±Inf, +Inf is returned.

Errors These functions will fail if:

Pole Error — The value of x is ±0.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, the divide-by-zero floating-point exception is raised.

The `logb()` function sets `errno` to `EDOM` if the value of x is 0.

Usage An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. An application should either examine the return value or check the floating point exception flags to detect exceptions.

An application can also set `errno` to 0 before calling `logb()`. On return, if `errno` is non-zero, an error has occurred. The `logbf()` and `logbl()` functions do not set `errno`.

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

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

See Also `feclearexcept(3M)`, `fetestexcept(3M)`, `ilogb(3M)`, `math.h(3HEAD)`, `matherr(3M)`, `scalb(3M)`, `attributes(5)`, `standards(5)`

Name lrint, lrintf, lrintl – round to nearest integer value using current rounding direction

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

long lrint(double x);

long lrintf(float x);

long lrintl(long double x);
```

Description These functions round their argument to the nearest integer value, rounding according to the current rounding direction.

Return Values Upon successful completion, these functions return the rounded integer value.

If *x* is NaN, a domain error occurs and an unspecified value is returned.

If *x* is +Inf, a domain error occurs and an unspecified value is returned.

If *x* is –Inf, a domain error occurs and an unspecified value is returned.

If the correct value is positive and too large to represent as a `long`, a domain error occurs and an unspecified value is returned.

If the correct value is negative and too large to represent as a `long`, a domain error occurs and an unspecified value is returned.

Errors These functions will fail if:

Domain Error — The *x* argument is NaN or ±Inf, or the correct value is not representable as an integer.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, then the invalid floating-point exception is raised.

Usage An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. An application should either examine the return value or check the floating point exception flags to detect exceptions.

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

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

See Also feclearexcept(3M), fetestexcept(3M), llrint(3M), math.h(3HEAD), attributes(5), standards(5)

Name lround, lroundf, lroundl – round to nearest integer value

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

long lround(double x);

long lroundf(float x);

long lroundl(long double x);
```

Description These functions round their argument to the nearest integer value, rounding halfway cases away from zero, regardless of the current rounding direction.

Return Values Upon successful completion, these functions return the rounded integer value.

If *x* is NaN, a domain error occurs and an unspecified value is returned.

If *x* is +Inf, a domain error occurs and an unspecified value is returned.

If *x* is –Inf, a domain error occurs and an unspecified value is returned.

If the correct value is positive and too large to represent as a `long`, a domain error occurs and an unspecified value is returned.

If the correct value is negative and too large to represent as a `long`, a domain error occurs and an unspecified value is returned.

Errors These functions will fail if:

Domain Error — The *x* argument is NaN or ±Inf, or the correct value is not representable as an integer.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, then the invalid floating-point exception is raised.

Usage An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. An application should either examine the return value or check the floating point exception flags to detect exceptions.

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

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

See Also feclearexcept(3M), fetestexcept(3M), llround(3M), math.h(3HEAD), attributes(5), standards(5)

Name maillock, mailunlock, touchlock – functions to manage lockfile(s) for user's mailbox

Synopsis

```
cc [ flag ... ] file ... -lmail [ library ... ]
#include <maillock.h>

int maillock(const char *user, int retrycnt);

void mailunlock(void);

void touchlock(void);
```

Description The `maillock()` function attempts to create a lockfile for the user's mailfile. If a lockfile already exists, and it has not been modified in the last 5 minutes, `maillock()` will remove the lockfile and set its own lockfile.

It is crucial that programs locking mail files refresh their locks at least every three minutes to maintain the lock. Refresh the lockfile by calling the `touchlock()` function with no arguments.

The algorithm used to determine the age of the lockfile takes into account clock drift between machines using a network file system. A zero is written into the lockfile so that the lock will be respected by systems running the standard version of System V.

If the lockfile has been modified in the last 5 minutes the process will sleep until the lock is available. The sleep algorithm is to sleep for 5 seconds times the attempt number. That is, the first sleep will be for 5 seconds, the next sleep will be for 10 seconds, etc. until the number of attempts reaches *retrycnt*.

When the lockfile is no longer needed, it should be removed by calling `mailunlock()`.

The *user* argument is the login name of the user for whose mailbox the lockfile will be created. `maillock()` assumes that user's mailfiles are in the "standard" place as defined in `<maillock.h>`.

Return Values Upon successful completion, `.maillock()` returns `0`. Otherwise it returns `−1`.

Files

`/var/mail/*`	user mailbox files
`/var/mail/*.lock`	user mailbox lockfiles

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also libmail(3LIB),attributes(5)

Notes The `mailunlock()` function will only remove the lockfile created from the most previous call to `maillock()`. Calling `maillock()` for different users without intervening calls to `mailunlock()` will cause the initially created lockfile(s) to remain, potentially blocking subsequent message delivery until the current process finally terminates.

Name matherr – math library exception-handling function

Synopsis

```
#include <math.h>

int matherr(struct exception *exc);
```

Description The System V Interface Definition, Third Edition (SVID3) specifies that certain `libm` functions call `matherr()` when exceptions are detected. Users may define their own mechanisms for handling exceptions, by including a function named `matherr()` in their programs. The `matherr()` function is of the form described above. When an exception occurs, a pointer to the exception structure *exc* will be passed to the user-supplied `matherr()` function. This structure, which is defined in the `<math.h>` header file, is as follows:

```
struct exception {
    int type;
    char *name;
    double arg1, arg2, retval;
};
```

The `type` member is an integer describing the type of exception that has occurred, from the following list of constants (defined in the header file):

`DOMAIN`	argument domain exception
`SING`	argument singularity
`OVERFLOW`	overflow range exception
`UNDERFLOW`	underflow range exception
`TLOSS`	total loss of significance
`PLOSS`	partial loss of significance

Both `TLOSS` and `PLOSS` reflect limitations of particular algorithms for trigonometric functions that suffer abrupt declines in accuracy at definite boundaries. Since the implementation does not suffer such abrupt declines, `PLOSS` is never signaled. `TLOSS` is signaled for Bessel functions *only* to satisfy SVID3 requirements.

The `name` member points to a string containing the name of the function that incurred the exception. The `arg1` and `arg2` members are the arguments with which the function was invoked. `retval` is set to the default value that will be returned by the function unless the user's `matherr()` sets it to a different value.

If the user's `matherr()` function returns non-zero, no exception message will be printed and `errno` is not set.

Svid3 Standard Conformance When an application is built as a SVID3 conforming application (see `standards(5)`), if `matherr()` is not supplied by the user, the default matherr exception-handling mechanisms, summarized in the table below, are invoked upon exception:

`DOMAIN` 0.0 is usually returned, `errno` is set to `EDOM` and a message is usually printed on standard error.

`SING` The largest finite single-precision number, `HUGE` of appropriate sign, is returned, `errno` is set to `EDOM`, and a message is printed on standard error.

`OVERFLOW` The largest finite single-precision number, `HUGE` of appropriate sign, is usually returned and `errno` is set to `ERANGE`.

`UNDERFLOW` 0.0 is returned and `errno` is set to `ERANGE`.

`TLOSS` 0.0 is returned, `errno` is set to `ERANGE`, and a message is printed on standard error.

In general, `errno` is not a reliable error indicator because it can be unexpectedly set by a function in a handler for an asynchronous signal.

SVID3 ERROR HANDLING PROCEDURES (compile with cc \-Xt)

<math.h> type	DOMAIN	SING	OVERFLOW	UNDERFLOW	TLOSS
errno	EDOM	EDOM	ERANGE	ERANGE	ERANGE
IEEE Exception	Invalid Operation	Division by Zero	Overflow	Underflow	–
fp_exception_type	fp_invalid	fp_division	fp_overflow	fp_underflow	–
ACOS, ASIN (|x| > 1):	Md, 0.0	–	–	–	–
ACOSH (x < 1), ATANH (|x| > 1):	NaN	–	–	–	–
ATAN2 (0,0):	Md, 0.0	–	–	–	–
COSH, SINH:	–	–	±HUGE	–	–
EXP:	–	–	+HUGE	0.0	–
FMOD (x,0):	x	–	–	–	–
HYPOT:	–	–	+HUGE	–	–
J0, J1, JN (|x| > X_TLOSS):	–	–	–	–	Mt, 0.0
LGAMMA:					
usual cases	–	–	+HUGE	–	–
(x = 0 or –integer)	–	Ms, +HUGE	–	–	–
LOG, LOG10:					
(x < 0)	Md, –HUGE	–	–	–	–

<math.h> type	DOMAIN	SING	OVERFLOW	UNDERFLOW	TLOSS
(x = 0)	–	Ms, –HUGE	–	–	–
POW:					
usual cases	–	–	±HUGE	±0.0	–
(x < 0) ** (y not an integer)	Md, 0.0	–	–	–	–
0 ** 0	Md, 0.0	–	–	–	–
0 ** (y < 0)	Md, 0.0	–	–	–	
REMAINDER (x,0):	NaN	–	–	–	–
SCALB:	–	–	±HUGE_VAL	±0.0	–
SQRT (x < 0):	Md, 0.0	–	–	–	–
Y0, Y1, YN:					
(x < 0)	Md, –HUGE	–	–	–	–
(x = 0)	–	Md, –HUGE	–	–	–
(x > X_TLOSS)	–	–	–	–	Mt, 0.0

Abbreviations

Md	Message is printed (DOMAIN error).
Ms	Message is printed (SING error).
Mt	Message is printed (TLOSS error).
NaN	IEEE NaN result and invalid operation exception.
HUGE	Maximum finite single-precision floating-point number.
HUGE_VAL	IEEE ∞ result and division-by-zero exception.
X_TLOSS	The value X_TLOSS is defined in <values.h>.

The interaction of IEEE arithmetic and `matherr()` is not defined when executing under IEEE rounding modes other than the default round to nearest: `matherr()` is not always called on overflow or underflow and can return results that differ from those in this table.

X/OPEN Common Application Environment (CAE) Specifications Conformance

The X/Open System Interfaces and Headers (XSH) Issue 3 and later revisions of that specification no longer sanctions the use of the `matherr` interface. The following table summarizes the values returned in the exceptional cases. In general, XSH dictates that as long as one of the input argument(s) is a NaN, NaN is returned. In particular, `pow(NaN,0)` = NaN.

CAE SPECIFICATION ERROR HANDLING PROCEDURES (compile with cc -Xa)

<math.h> type	DOMAIN	SING	OVERFLOW	UNDERFLOW	TLOSS
errno	EDOM	EDOM	ERANGE	ERANGE	ERANGE
ACOS, ASIN (|x| > 1):	0.0	–	–	–	–
ATAN2 (0,0):	0.0	–	–	–	–
COSH, SINH:	–	–	{±HUGE_VAL}	–	–
EXP:	–	–	{+HUGE_VAL}	{0.0}	–
FMOD (x,0):	{NaN}	–	–	–	–
HYPOT:	–	–	{+HUGE_VAL}	–	–
J0, J1, JN (|x| > X_TLOSS):	–	–	–	–	{0.0}
LGAMMA:					
usual cases	–	–	{+HUGE_VAL}	–	–
(x = 0 or –integer)	–	+HUGE_VAL	–	–	–
LOG, LOG10:					
(x < 0)	-HUGE_VAL	–	–	–	–
(x = 0)	–	-HUGE_VAL	–	–	–
POW:					
usual cases	–	–	±HUGE_VAL	±0.0	–
(x < 0) ** (y not an integer)	0.0	–	–	–	–
0 ** 0	{1.0}	–	–	–	–
0 ** (y < 0)	{-HUGE_VAL}	–	–	–	–
SQRT (x < 0):	0.0	–	–	–	–
Y0, Y1, YN:					
(x < 0)	{-HUGE_VAL}	–	–	–	–
(x = 0)	–	{-HUGE_VAL}	–	–	–
(x > X_TLOSS)	–	–	–	–	0.0

Abbreviations

{...}	`errno` is not to be relied upon in all braced cases.
NaN	IEEE NaN result and invalid operation exception.
HUGE_VAL	IEEE ∞ result and division-by-zero exception.
X_TLOSS	The value X_TLOSS is defined in <`values.h`>.

Ansi/ISO-C Standard Conformance

The ANSI/ISO-C standard covers a small subset of the CAE specification.

The following table summarizes the values returned in the exceptional cases.

ANSI/ISO-C ERROR HANDLING PROCEDURES (compile with cc -Xc)

<math.h> type	DOMAIN	SING	OVERFLOW	UNDERFLOW
`errno`	EDOM	EDOM	ERANGE	ERANGE
ACOS, ASIN (|x| > 1):	0.0	–	–	–
ATAN2 (0,0):	0.0	–	–	–
EXP:	–	–	+HUGE_VAL	0.0
FMOD (x,0):	NaN	–	–	–
LOG, LOG10:				
(x < 0)	`-HUGE_VAL`	–	–	–
(x = 0)	–	`-HUGE_VAL`	–	–
POW:				
usual cases	–	–	±HUGE_VAL	±0.0
(x < 0) ** (y not an integer)	0.0	–	–	–
0 ** (y < 0)	`-HUGE_VAL`	–	–	–
SQRT (x < 0):	0.0	–	–	–

ABBREVIATIONS

NaN	IEEE NaN result and invalid operation exception.
HUGE_VAL	IEEE ∞ result and division-by-zero.

Examples

EXAMPLE 1 Example of `matherr()` function

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

int
matherr(struct exception *x) {
```

EXAMPLE 1 Example of `matherr()` function *(Continued)*

```
    switch (x->type) {
        case DOMAIN:
            /* change sqrt to return sqrt(-arg1), not NaN */
   if (!strcmp(x->name, "sqrt")) {
    x->retval = sqrt(-x->arg1);
    return (0); /* print message and set errno */
   } /* FALLTHRU */
  case SING:
   /* all other domain or sing exceptions, print message and */
   /* abort */
   fprintf(stderr, "domain exception in %s\n", x->name);
   abort( );
   break;
  }
  return (0); /* all other exceptions, execute default procedure */
 }
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also

`attributes(5)`, `standards(5)`

Name m_create_layout – initialize a layout object

Synopsis

```
cc [ flag... ] file... –llayout [ library... ]
#include <sys/layout.h>

LayoutObject m_create_layout(const AttrObject attrobj,
      const char*modifier);
```

Description The `m_create_layout()` function creates a `LayoutObject` associated with the locale identified by *attrobj*.

The `LayoutObject` is an opaque object containing all the data and methods necessary to perform the layout operations on context-dependent or directional characters of the locale identified by the *attrobj*. The memory for the `LayoutObject` is allocated by `m_create_layout()`. The `LayoutObject` created has default layout values. If the *modifier* argument is not `NULL`, the layout values specified by the *modifier* overwrite the default layout values associated with the locale. Internal states maintained by the layout transformation function across transformations are set to their initial values.

The *attrobj* argument is or may be an amalgam of many opaque objects. A locale object is just one example of the type of object that can be attached to an attribute object. The *attrobj* argument specifies a name that is usually associated with a locale category. If *attrobj* is `NULL`, the created `LayoutObject` is associated with the current locale as set by the `setlocale(3C)` function.

The *modifier* argument announces a set of layout values when the `LayoutObject` is created.

Return Values Upon successful completion, the `m_create_layout()` function returns a `LayoutObject` for use in subsequent calls to `m_*_layout()` functions. Otherwise the `m_create_layout()` function returns (`LayoutObject`) 0 and sets `errno` to indicate the error.

Errors The `m_create_layout()` function may fail if:

`EBADF` The attribute object is invalid or the locale asssociated with the attribute object is not available.

`EINVAL` The *modifier* string has a syntax error or it contains unknown layout values.

`EMFILE` There are `{OPEN_MAX}` file descriptors currently open in the calling process.

`ENOMEM` Insufficient storage space is available.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also setlocale(3C), attributes(5), standards(5)

Name md4, MD4Init, MD4Update, MD4Final – MD4 digest functions

Synopsis

```
cc [ flag ... ] file ... -lmd [ library ... ]
#include <md4.h>

void MD4Init(MD4_CTX *context);

void MD4Update(MD4_CTX *context, unsigned char *input,
     unsigned int inlen);

void MD4Final(unsigned char *output, MD4_CTX *context);
```

Description The MD4 functions implement the MD4 message-digest algorithm. The algorithm takes as input a message of arbitrary length and produces a "fingerprint" or "message digest" as output. The MD4 message-digest algorithm is intended for digital signature applications in which large files are "compressed" in a secure manner before being encrypted with a private (secret) key under a public-key cryptosystem such as RSA.

MD4Init(), MD4Update(), MD4Final()

The MD4Init(), MD4Update(), and MD4Final() functions allow an MD4 digest to be computed over multiple message blocks. Between blocks, the state of the MD4 computation is held in an MD4 context structure allocated by the caller. A complete digest computation consists of calls to MD4 functions in the following order: one call to MD4Init(), one or more calls to MD4Update(), and one call to MD4Final().

The MD4Init() function initializes the MD4 context structure pointed to by *context*.

The MD4Update() function computes a partial MD4 digest on the *inlen*-byte message block pointed to by *input*, and updates the MD4 context structure pointed to by *context* accordingly.

The MD4Final() function generates the final MD4 digest, using the MD4 context structure pointed to by *context*. The MD4 digest is written to output. After a call to MD4Final(), the state of the context structure is undefined. It must be reinitialized with MD4Init() before it can be used again.

Return Values These functions do not return a value.

Security The MD4 digest algorithm is not currently considered cryptographically secure. It is included in libmd(3LIB) for use by legacy protocols and systems only. It should not be used by new systems or protocols.

Examples **EXAMPLE 1** Authenticate a message found in multiple buffers

The following is a sample function that must authenticate a message that is found in multiple buffers. The calling function provides an authentication buffer that will contain the result of the MD4 digest.

```
#include <sys/types.h>
#include <sys/uio.h>
#include <md4.h>
```

EXAMPLE 1 Authenticate a message found in multiple buffers *(Continued)*

```
int
AuthenticateMsg(unsigned char *auth_buffer, struct iovec
                *messageIov, unsigned int num_buffers)
{
    MD4_CTX ctx;
    unsigned int i;

    MD4Init(&ctx);

    for(i=0; i<num_buffers; i++)
    {
         MD4Update(&ctx, messageIov->iov_base,
                   messageIov->iov_len);
         messageIov += sizeof(struct iovec);
    }

    MD4Final(auth_buffer, &ctx);

    return 0;
}
```

Attributes

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

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

See Also

libmd(3LIB)

RFC 1320

Name md5, md5_calc, MD5Init, MD5Update, MD5Final – MD5 digest functions

Synopsis

```
cc [ flag ... ] file ... -lmd5 [ library ... ]
#include <md5.h>

void md5_calc(unsigned char *output, unsigned char *input,
     unsigned int inlen);

void MD5Init(MD5_CTX *context);

void MD5Update(MD5_CTX *context, unsigned char *input,
     unsigned int inlen);

void MD5Final(unsigned char *output, MD5_CTX *context);
```

Description These functions implement the MD5 message-digest algorithm, which takes as input a message of arbitrary length and produces as output a 128-bit "fingerprint" or "message digest" of the input. It is intended for digital signature applications, where large file must be "compressed" in a secure manner before being encrypted with a private (secret) key under a public-key cryptosystem such as RSA.

`md5_calc()` The `md5_calc()` function computes an MD5 digest on a single message block. The *inlen*-byte block is pointed to by *input*, and the 16-byte MD5 digest is written to *output*.

`MD5Init()`, `MD5Update()`, `MD5Final()` The `MD5Init()`, `MD5Update()`, and `MD5Final()` functions allow an MD5 digest to be computed over multiple message blocks; between blocks, the state of the MD5 computation is held in an MD5 context structure, allocated by the caller. A complete digest computation consists of one call to `MD5Init()`, one or more calls to `MD5Update()`, and one call to `MD5Final()`, in that order.

The `MD5Init()` function initializes the MD5 context structure pointed to by *context*.

The `MD5Update()` function computes a partial MD5 digest on the *inlen*-byte message block pointed to by *input*, and updates the MD5 context structure pointed to by *context* accordingly.

The `MD5Final()` function generates the final MD5 digest, using the MD5 context structure pointed to by *context*; the 16-byte MD5 digest is written to *output*. After calling `MD5Final()`, the state of the context structure is undefined; it must be reinitialized with `MD5Init()` before being used again.

Return Values These functions do not return a value.

Examples **EXAMPLE 1** Authenticate a message found in multiple buffers

The following is a sample function that must authenticate a message that is found in multiple buffers. The calling function provides an authentication buffer that will contain the result of the MD5 digest.

```
#include <sys/types.h>
#include <sys/uio.h>
#include <md5.h>
```

EXAMPLE 1 Authenticate a message found in multiple buffers *(Continued)*

```
int
AuthenticateMsg(unsigned char *auth_buffer, struct iovec
                *messageIov, unsigned int num_buffers)
{
    MD5_CTX md5_context;
    unsigned int i;

    MD5Init(&md5_context);

    for(i=0; i<num_buffers; i++)
    {
         MD5Update(&md5_context, messageIov->iov_base,
                   messageIov->iov_len);
         messageIov += sizeof(struct iovec);
    }

    MD5Final(auth_buffer, &md5_context);

    return 0;
}
```

EXAMPLE 2 Use `md5_calc()` to generate the MD5 digest

Since the buffer to be computed is contiguous, the `md5_calc()` function can be used to generate the MD5 digest.

```
int AuthenticateMsg(unsigned char *auth_buffer, unsigned
                    char *buffer, unsigned int length)
{
    md5_calc(buffer, auth_buffer, length);

    return (0);
}
```

Attributes

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

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

See Also

`libmd5(3LIB)`

Rivest, R., The MD5 Message-Digest Algorithm, RFC 1321, April 1992.

Name m_destroy_layout – destroy a layout object

Synopsis

```
cc [ flag... ] file... –llayout [ library... ]
#include <sys/layout.h>

int m_destroy_layout(const LayoutObject layoutobject);
```

Description The `m_destroy_layout()` function destroys a `LayoutObject` by deallocating the layout object and all the associated resources previously allocated by the `m_create_layout(3LAYOUT)` function.

Return Values Upon successful completion, `0` is returned. Otherwise `–1` is returned and `errno` is set to indicate the error.

Errors The `m_destroy_layout()` function may fail if:

`EBADF` The attribute object is erroneous.

`EFAULT` Errors occurred while processing the request.

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

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

See Also `m_create_layout(3LAYOUT)`, `attributes(5)`, `standards(5)`

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 `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).
`/vol/...`	An absolute Volume Management pathname, such as `/vol/dev/aliases/floppy0` or `/vol/dsk/fred`. If this supplied pathname is not a symbolic link, then a copy of that pathname is returned. If the supplied pathname is a symbolic link then it is dereferenced and a copy of that dereferenced pathname is returned.
volume_name	The Volume Management volume name for a particular volume, such as `fred` (see `fdformat(1)` for a description of how to label floppies). In this case a pathname in the Volume Management namespace is returned.
volmgt_symname	The Volume Management symbolic name for a device, such as `floppy0` or `cdrom2` (see `volfs(7FS)` for more information on Volume Management symbolic names), in which case a pathname in the Volume Management namespace is returned.
media_type	The Volume Management 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 Upon successful completion `media_findname()` returns a pointer to the pathname found. In the case of an error a null pointer is returned.

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 Volume Management 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

See Also

`cc(1B)`, `fdformat(1)`, `vold(1M)`, `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)`, `volfs(7FS)`

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 `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 Upon successful completion `media_getattr()` returns a pointer to the value corresponding to the specified attribute. A null pointer is returned if the specified volume does not exist, if the specified attribute for that volume doesn't exist, if the specified attribute is boolean and its value is false, or if `malloc(3C)` fails to allocate space for the return value.

`media_setattr()` returns `1` upon success, and `0` upon failure.

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.

Additionally, either routine can fail returning the following error values:

`ENXIO` The Volume Management daemon, `vold`, is not running

`EINTR` The routine was interrupted by the user before finishing

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("/vol/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("/vol/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

See Also `cc(1B)`, `vold(1M)`, `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)`

Notes Upon success `media_getattr()` returns a pointer to a string which has been allocated, and should be freed when no longer in use (see `free(3C)`).

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 `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 `media_getid()` returns the *id* of the volume. This value is unique for each volume. If `media_getid()` returns `0`, the *path* provided is not valid, for example, it is a `block` or `char` device.

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
Commitment Level	Public

See Also volmgt_ownspath(3VOLMGT),volmgt_running(3VOLMGT),attributes(5)

Name m_getvalues_layout – query layout values of a LayoutObject

Synopsis

```
cc [ flag... ] file... –llayout [ library... ]
#include <sys/layout.h>

int m_getvalues_layout(const LayoutObject
     layout_object, LayoutValues values, int *index_returned);
```

Description The `m_getvalues_layout()` function queries the current setting of layout values within a `LayoutObject`.

The *layout_object* argument specifies a `LayoutObject` returned by the `m_create_layout(3LAYOUT)` function.

The *values* argument specifies the list of layout values that are to be queried. Each value element of a `LayoutValueRec` must point to a location where the layout value is stored. That is, if the layout value is of type `T`, the argument must be of type `T*`. The values are queried from the `LayoutObject` and represent its current state.

It is the user's responsibility to manage the space allocation for the layout values queried. If the layout value name has `QueryValueSize` OR-ed to it, instead of the value of the layout value, only its size is returned. The caller can use this option to determine the amount of memory needed to be allocated for the layout values queried.

Return Values Upon successful completion, the `m_getvalues_layout()` function returns `0`. If any value cannot be queried, the index of the value causing the error is returned in *index_returned*, `–1` is returned and `errno` is set to indicate the error.

Errors The `m_getvalues_layout()` function may fail if:

`EINVAL` The layout value specified by *index_returned* is unknown, its value is invalid, or the *layout_object* argument is invalid. In the case of an invalid *layout_object* argument, the value returned in *index_returned* is `–1`.

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

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

See Also `m_create_layout(3LAYOUT)`, `attributes(5)`, `standards(5)`

Name mkdirp, rmdirp – create or remove directories in a path

Synopsis

```
cc [ flag ... ] file ... -lgen [ library ... ]
#include <libgen.h>

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

int rmdirp(char *dir, char *dir1);
```

Description The `mkdirp()` function creates all the missing directories in *path* with *mode*. See `chmod(2)` for the values of *mode*.

The `rmdirp()` function removes directories in path *dir*. This removal begins at the end of the path and moves backward toward the root as far as possible. If an error occurs, the remaining path is stored in *dir1*.

Return Values If *path* already exists or if a needed directory cannot be created, `mkdirp()` returns `−1` and sets `errno` to one of the error values listed for `mkdir(2)`. It returns zero if all the directories are created.

The `rmdirp()` function returns `0` if it is able to remove every directory in the path. It returns `−2` if a "`.`" or "`..`" is in the path and `−3` if an attempt is made to remove the current directory. Otherwise it returns `−1`.

Examples **EXAMPLE 1** Example of creating scratch directories.

The following example creates scratch directories.

```
/* create scratch directories */
if(mkdirp("/tmp/sub1/sub2/sub3", 0755) == −1) {
    fprintf(stderr, "cannot create directory");
    exit(1);
}
chdir("/tmp/sub1/sub2/sub3");
.
.
.
/* cleanup */
chdir("/tmp");
rmdirp("sub1/sub2/sub3");
```

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also chmod(2), mkdir(2), rmdir(2), malloc(3C), attributes(5)

Notes The `mkdirp()` function uses malloc(3C) to allocate temporary space for the string.

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 *Oracle 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 modf, modff, modfl – decompose floating-point number

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double modf(double x, double *iptr);

float modff(float x, float *iptr);

long double modfl(long double x, long double *iptr);
```

Description These functions break the argument *x* into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part as a `double` for the `modf()` function, a float for the `modff()` function, or a long double for the`modfl()` function in the object pointed to by *iptr*.

Return Values Upon successful completion, these functions return the signed fractional part of *x*.

If *x* is NaN, a NaN is returned and **iptr* is set to NaN.

If *x* is ±Inf, ±0 is returned and **iptr* is set to ±Inf.

Errors No errors are defined.

Usage These functions compute the function result and **iptr* such that:

```
a = modf(x, &iptr) ;
x == a+*iptr ;
```

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

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

See Also `frexp(3M)`, `isnan(3M)`, `ldexp(3M)`, `attributes(5)`, `standards(5)`

Name mp, mp_madd, mp_msub, mp_mult, mp_mdiv, mp_mcmp, mp_min, mp_mout, mp_pow, mp_gcd, mp_rpow, mp_msqrt, mp_sdiv, mp_itom, mp_xtom, mp_mtox, mp_mfree – multiple precision integer arithmetic

Synopsis

```
cc [ flag... ] file... -lmp [ library... ]
#include <mp.h>

void mp_madd(MINT *a, MINT *b, MINT *c);

void mp_msub(MINT *a, MINT *b, MINT *c);

void mp_mult(MINT *a, MINT *b, MINT *c);

void mp_mdiv(MINT *a, MINT *b, MINT *q, MINT *r);

int mp_mcmp(MINT *a, MINT *b);

int mp_min(MINT *a);

void mp_mout(MINT *a);

void mp_pow(MINT *a, MINT *b, MINT *c, MINT *d);

void mp_gcd(MINT *a, MINT *b, MINT *c);

void mp_rpow(MINT *a, short n, MINT *b);

int mp_msqrt(MINT *a, MINT *b, MINT *r);

void mp_sdiv(MINT *a, short n, MINT *q, short *r);

MINT * mp_itom(short n);

MINT * mp_xtom(char *a);

char * mp_mtox(MINT *a);

void mp_mfree(MINT *a);
```

Description These functions perform arithmetic on integers of arbitrary length. The integers are stored using the defined type `MINT`. Pointers to a `MINT` should be initialized using the function `mp_itom(`*n*`)`, which sets the initial value to *n*. Alternatively, `mp_xtom(`*a*`)` may be used to initialize a `MINT` from a string of hexadecimal digits. `mp_mfree(`*a*`)` may be used to release the storage allocated by the `mp_itom(`*a*`)` and `mp_xtom(`*a*`)` routines.

The `mp_madd(`*a*,*b*,*c*`)`, `mp_msub(`*a*,*b*,*c*`)` and `mp_mult(`*a*,*b*,*c*`)` functions assign to their third arguments the sum, difference, and product, respectively, of their first two arguments. The `mp_mdiv(`*a*,*b*,*q*,*r*`)` function assigns the quotient and remainder, respectively, to its third and fourth arguments. The `mp_sdiv(`*a*,*n*,*q*,*r*`)` function is similar to `mp_mdiv(`*a*,*b*,*q*,*r*`)` except that the divisor is an ordinary integer. The `mp_msqrt(`*a*,*b*,*r*`)` function produces the square root and remainder of its first argument. The `mp_mcmp(`*a*,*b*`)` function compares the values of its arguments and returns `0` if the two values are equal, a value greater than `0` if the first argument is greater than the second, and a value less than `0` if the second argument is greater than the first. The `mp_rpow(`*a*,*n*,*b*`)` function raises *a* to the *n*th power and assigns this value to *b*. The `mp_pow(`*a*,*b* ,*c*,*d*`)` function raises *a* to the *b*th power, reduces the result `modulo` *c* and assigns this

value to *d*. The `mp_min(`*a*`)` and `mp_mout(`*a*`)` functions perform decimal input and output. The `mp_gcd(`*a*,*b*,*c*`)` function finds the greatest common divisor of the first two arguments, returning it in the third argument. The `mp_mtox(` *a*`)` function provides the inverse of `mp_xtom(a)`. To release the storage allocated by `mp_mtox(`*a*`)` use `free()` (see `malloc(3C)`).

Use the `-lmp` loader option to obtain access to these functions.

Files `/usr/lib/libmp.so` shared object

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also `exp(3M)`, `malloc(3C)`, `libmp(3LIB)`, `attributes(5)`

Diagnostics Illegal operations and running out of memory produce messages and core images.

Warnings The function `pow()` exists in both `libmp` and `libm` with widely differing semantics. This is the reason `libmp.so.2` exists. `libmp.so.1` exists solely for reasons of backward compatibility, and should not be used otherwise. Use the `mp_*( )` functions instead. See `libmp(3LIB)`.

Name MP_AssignLogicalUnitToTPG – assign a multipath logical unit to a target port group

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_AssignLogicalUnitToTPG(MP_OID tpgOid, MP_OID luOid);
```

Parameters

tpgOid An object ID that has type `MP_TARGET_PORT_GROUP`. The target port group currently in active access state that the administrator would like the LU assigned to.

luOid An object ID that has type `MP_MULTIPATH_LOGICAL_UNIT`.

Description The `MP_AssignLogicalUnitToTPG()` function assigns a multipath logical unit to a target port group.

Calling this function is valid only if the field `supportsLuAssignment` in the data structure `TARGET_PORT_GROUP_PROPERTIES` is true. This capability is not defined in SCSI standards. In some cases, devices support this capability through non-SCSI interfaces (such as SMI-S or SNMP). This method is only used when devices support this capability through vendor-specific means.

At any given time, each LU will typically be associated with two target port groups, one in active state and one in standby state. The result of this API will be that the LU associations change to a different pair of target port groups. The caller should specify the object ID of the desired target port group in active access state.

Return Values

`MP_STATUS_INVALID_OBJECT_TYPE`	The *tpgOid* or *luOid* parameter does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.
`MP_STATUS_INVALID_PARAMETER`	The *tpgOid* parameter has a type subfield other than `MP_OBJECT_TYPE_TARGET_PORT_GROUP` or *luOid* has a type subfield other than `MP_OBJECT_TYPE_MULTIPATH_LU`.
`MP_STATUS_OBJECT_NOT_FOUND`	The *tpgOid* or *luOid* owner ID or object sequence number is invalid.
`MP_STATUS_UNSUPPORTED`	The API is not supported.
`MP_STATUS_SUCCESS`	The operation is successful.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also libMPAPI(3LIB), MP_GetAssociatedTPGOidList(3MPAPI), MP_GetMPLuOidListFromTPG(3MPAPI), attributes(5)

Name MP_CancelOverridePath – cancel a path override

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_CancelOverridePath(MP_OID logicalUnitOid);
```

Parameters *logicalUnitOid* An object ID that has type `MP_MULTIPATH_LOGICAL_UNIT`.

Description The `MP_CancelOverridePath()` function cancels a path override and re-enables load balancing.

Calling this function is valid only if the field `canOverridePaths` in data structure `MP_PLUGIN_PROPERTIES` is true.

The previous load balance configuration and preferences in effect before the path was overridden are restored.

Return Values

`MP_STATUS_INVALID_OBJECT_TYPE`	The *logicalUnitOid* parameter does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.
`MP_STATUS_INVALID_PARAMETER`	The *logicalUnitOid* parameter has a type subfield other than `MP_MULTIPATH_LOGICAL_UNIT`.
`MP_STATUS_OBJECT_NOT_FOUND`	The *logicalUnitOid* owner ID or object sequence number is invalid.
`MP_STATUS_SUCCESS`	The operation is successful.
`MP_STATUS_UNSUPPORTED`	The API is not supported.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also `libMPAPI(3LIB)`, `MP_SetOverridePath(3MPAPI)`, `attributes(5)`

Multipath Management API Version 1.0

Name MP_CompareOIDs – compare two object IDs

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_CompareOIDs(MP_OID oid1, MP_OID oid2);
```

Parameters

oid1 An object ID that has type MP_OIDs for two objects to compare.

oid2 An object ID that has type MP_OIDs for two objects to compare.

Description The `MP_CompareOIDs()` function compares two object IDs (OIDs) for equality to see whether they refer to the same object. The fields in the two object IDs are compared field-by-field for equality.

Return Values

`MP_STATUS_FAILED` The object IDs do not compare.

`MP_STATUS_SUCCESS` The two object IDs refer to the same object.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also `libMPAPI(3LIB)`, `attributes(5)`

Multipath Management API Version 1.0

Name MP_DeregisterForObjectPropertyChanges – deregister a previously registered client function

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_DeregisterForObjectPropertyChanges(
     MP_OBJECT_PROPERTY_FN pClientFn, MP_OBJECT_TYPE objectType,
     MP_OID pluginOid);
```

Parameters

pClientFn A pointer to an object ID that has type `MP_OBJECT_PROPERTY_FN` function defined by the client that was previously registered using the `MP_RegisterForObjectPropertyChanges(3MPAPI)` API. With a successful return this function will no longer be called to inform the client of object property changes.

objectType The type of object the client wants to deregister for property change callbacks.

pluginOid If this is a valid plugin object ID, then registration will be removed from that plugin. If this is zero, then registration is removed for all plugins.

Description The `MP_DeregisterForObjectPropertyChanges()` function deregisters a previously registered client function that is to be invoked whenever an object's property changes.

The function specified by *pClientFn* takes a single parameter of type `MP_OBJECT_PROPERTY_FN`.

The function specified by *pClientFn* will no longer be called whenever an object's property changes.

Return Values

`MP_STATUS_INVALID_OBJECT_TYPE`	The *pluginOid* parameter does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.
`MP_STATUS_INVALID_PARAMETER`	The *pluginOid* parmeter is not zero and has a type subfield other than `MP_OBJECT_TYPE_PLUGIN`.
`MP_STATUS_OBJECT_NOT_FOUND`	The *pluginOid* owner ID or object sequence number is invalid.
`MP_STATUS_UNKNOWN_FN`	The *pClientFn* parmeter is not the same as the previously registered function.
`MP_STATUS_SUCCESS`	The *pClientFn* parmeter is deregistered successfully.
`MP_STATUS_FAILED`	The *pClientFn* parmeter deregistration is not possible.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also libMPAPI(3LIB), MP_RegisterForObjectPropertyChanges(3MPAPI), attributes(5)

Multipath Management API Version 1.0

Name MP_DeregisterForObjectVisibilityChanges – deregister a client function

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_DeregisterForObjectVisibilityChanges(
     MP_OBJECT_VISIBILITY_FN pClientFn, MP_OBJECT_TYPE objectType,
     MP_OID pluginOid);
```

Parameters

pClientFn	A pointer to an object ID that has type `MP_OBJECT_VISIBILITY_FN` function defined by the client that was previously registered using the `MP_RegisterForObjectVisibilityChanges(3MPAPI)` API. With a successful return this function will no longer be called to inform the client of object visibility changes.
objectType	The type of object the client wishes to deregister for visibility change callbacks.
pluginOid	If this is a valid plugin object ID, then registration will be removed from that plugin. If this is zero, then registration is removed for all plugins.

Description The `MP_DeregisterForObjectVisibilityChanges()` function deregistesr a client function to be called whenever a high level object appears or disappears.

The function specified by *pClientFn* takes a single parameter of type MP_OBJECT_VISIBILITY_FN.

The function specified by *pClientFn* will no longer be called whenever high level objects appear or disappear.

Return Values

`MP_STATUS_INVALID_OBJECT_TYPE`	The *pluginOid* parameter does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.
`MP_STATUS_INVALID_PARAMETER`	The *pluginOid* parameter is not zero or has a type subfield other than `MP_OBJECT_TYPE_PLUGIN`.
`MP_STATUS_OBJECT_NOT_FOUND`	The *pluginOid* owner ID or object sequence number is invalid.
`MP_STATUS_UNKNOWN_FN`	The *pluginOid* parameter is not zero or has a type subfield other than `MP_OBJECT_TYPE_PLUGIN`.
`MP_STATUS_SUCCESS`	The *pClientFn* parameter is deregistered successfully.
`MP_STATUS_FAILED`	The *pClientFn* parameter deregistration is not possible at this time.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also libMPAPI(3LIB), MP_RegisterForObjectVisibilityChanges(3MPAPI), attributes(5)

Multipath Management API Version 1.0

Name MP_DeregisterPlugin – deregister a plugin

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_DeregisterPlugin(MP_WCHAR *pPluginId);
```

Parameters

pPluginId A pointer to a Plugin ID previously registered using the `MP_RegisterPlugin(3MPAPI)` API.

Description The `MP_DeregisterPlugin()` function deregisters a plugin from the common library.

The plugin will no longer be invoked by the common library. This API does not dynamically remove the plugin from a running library instance. Instead, it prevents an application that is currently not using a plugin from accessing the plugin. This is generally the behavior expected from dynamically loaded modules.

Return Values

`MP_STATUS_INVALID_PARAMETER`	The *pPluginId* parameter is null or specifies a memory area that is not executable.
`MP_STATUS_UNKNOWN_FN`	The *pPluginId* parameter is not the same as a previously registered function.
`MP_STATUS_SUCCESS`	The *pPluginId* parameter is deregistered successfully.
`MP_STATUS_FAILED`	The *pPluginId*parameter deregistration is not possible at this time

Files `/etc/mpapi.conf` MPAPI library configuration file

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also `libMPAPI(3LIB)`, `MP_RegisterPlugin(3MPAPI)`, `mpapi.conf(4)`, `attributes(5)`

Multipath Management API Version 1.0

Name MP_DisableAutoFailback – disable auto-failback

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_DisableAutoFailback(MP_OID oid);
```

Parameters *oid* The object ID of the plugin or the multipath logical unit.

Description The `MP_DisableAutoFailback()` function disables auto-failback for the specified plugin or multipath logical unit.

Return Values

`MP_STATUS_INVALID_OBJECT_TYPE`	The *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the AP I.
`MP_STATUS_INVALID_PARAMETER`	The *oid* has a type subfield other than `MP_OBJECT_TYPE_PLUGIN` or `MP_OBJECT_TYPE_MULTIPATH_LU`.
`MP_STATUS_OBJECT_NOT_FOUND`	The *oid* owner ID or object sequence number is invalid.
`MP_STATUS_SUCCESS`	The operation is successful.
`MP_STATUS_UNSUPPORTED`	The API is not supported.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also `libMPAPI(3LIB)`, `MP_EnableAutoFailback(3MPAPI)`, `attributes(5)`

Multipath Management API Version 1.0

Name MP_DisableAutoProbing – disable auto-probing

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_DisableAutoProbing(MP_OID oid);
```

Parameters *oid* The object ID of the plugin or the multipath logical unit.

Description The `MP_DisableAutoProbing()` function disables auto-probing for the specified plugin or multipath logical unit.

Return Values

`MP_STATUS_INVALID_OBJECT_TYPE`	The *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.
`MP_STATUS_INVALID_PARAMETER`	The *oid* has a type subfield other than `MP_OBJECT_TYPE_PLUGIN` or `MP_OBJECT_TYPE_MULTIPATH_LU`.
`MP_STATUS_OBJECT_NOT_FOUND`	The *oid* owner ID or object sequence number is invalid.
`MP_STATUS_SUCCESS`	The operation is successful.
`MP_STATUS_UNSUPPORTED`	The API is not supported.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also `libMPAPI(3LIB)`, `MP_EnableAutoProbing(3MPAPI)`, `attributes(5)`

Multipath Management API Version 1.0

Name MP_DisablePath – disable a path

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_DisablePath(MP_OID oid);
```

Parameters *oid* The object ID of the path.

Description The `MP_DisablePath()` function disables a path. This API might cause failover in a logical unit with asymmetric access.

This API sets the disabled field of structure `MP_PATH_LOGICAL_UNIT_PROPERTIES` to true.

Return Values

`MP_STATUS_INVALID_OBJECT_TYPE`	The *oid* parameter does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.
`MP_STATUS_OBJECT_NOT_FOUND`	The *oid* parameter owner ID or object sequence number is invalid.
`MP_STATUS_INVALID_PARAMETER`	The *oid* parameter does not have a type subfield of `MP_OBJECT_TYPE_PATH_LU`.
`MP_STATUS_UNSUPPORTED`	The API is not supported.
`MP_STATUS_TRY_AGAIN`	The path cannot be disabled at this time.
`MP_STATUS_NOT_PERMITTED`	Disabling this path causes the logical unit to become unavailable. The plugin that administers the path might return this value or allow the last path to be disabled.
`MP_STATUS_SUCCESS`	The operation is successful.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also `libMPAPI(3LIB)`, `MP_EnablePath(3MPAPI)`, `attributes(5)`

Multipath Management API Version 1.0

Name MP_EnableAutoFailback – enable auto-failback

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_EnableAutoFailback(MP_OID oid);
```

Parameters *oid* The object ID of the plugin or multipath logical unit.

Description The `MP_EnableAutoFailback()` function enables auto-failback.

Return Values

`MP_STATUS_INVALID_OBJECT_TYPEkkk`	The *oid* parameter does not specify any valid object type . This is most likely to happen if an uninitialized object ID is passed to the AP I.
`MP_STATUS_INVALID_PARAMETER`	The *oid* parameter has a type subfield other than `MP_OBJE CT_TYPE_PLUGIN` or `MP_OBJECT_TYPE_MULTIPATH_LU`.
`MP_STATUS_OBJECT_NOT_FOUND`	The *oid* parameter owner ID or object sequence number is invalid.
`MP_STATUS_SUCCESS`	The operation is successful
`MP_STATUS_UNSUPPORTED`	The API is not supported.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also `libMPAPI(3LIB)`, `MP_DisableAutoFailback(3MPAPI)`, `attributes(5)`

Multipath Management API Version 1.0

Name MP_EnableAutoProbing – enable auto-probing

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_EnableAutoProbing(MP_OID oid);
```

Parameters *oid* The object ID of the plugin or multipath logical unit.

Description The `MP_EnableAutoProbing()` function enables auto-probing.

Return Values

`MP_STATUS_INVALID_OBJECT_TYPE`	The *oid* parameter does not specify any valid object type . This is most likely to happen if an uninitialized object ID is passed to the API.
`MP_STATUS_INVALID_PARAMETER`	The *oid* parameter has a type subfield other than `MP_OBJE CT_TYPE_PLUGIN` or `MP_OBJECT_TYPE_MULTIPATH_LU`.
`MP_STATUS_OBJECT_NOT_FOUND`	The *oid* parameter owner ID or object sequence number is invalid.
`MP_STATUS_SUCCESS`	The operation is successful.
`MP_STATUS_UNSUPPORTED`	The API is not supported.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also `libMPAPI(3LIB)`, `MP_DisableAutoProbing(3MPAPI)`, `attributes(5)`

Multipath Management API Version 1.0

REFERENCE

Extended Library Functions - Part 4

Name MP_EnablePath – enable a path

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_EnablePath(MP_OID oid);
```

Parameters *oid* The object ID of the path.

Description The `MP_EnablePath()` function enables a path. This API might cause failover in a logical unit with asymmetric access.

This API sets the field disabled of structure `MP_PATH_LOGICAL_UNIT_PROPERTIES` to false.

Return Values

`MP_STATUS_INVALID_OBJECT_TYPE`	The *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.
`MP_STATUS_INVALID_PARAMETER`	The *oid* has a type subfield other than `MP_OBJECT_TYPE_PATH_LU`.
`MP_STATUS_OBJECT_NOT_FOUND`	The *oid* owner ID or object sequence number is invalid.
`MP_STATUS_UNSUPPORTED`	The API is not supported.
`MP_STATUS_TRY_AGAIN`	The path cannot be enabled at this time.
`MP_STATUS_SUCCESS`	The operation is successful.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also `libMPAPI(3LIB)`, `MP_DisablePath(3MPAPI)`, `attributes(5)`

Multipath Management API Version 1.0

Name MP_FreeOidList – free up memory

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_FreeOidList(MP_OID_LIST *pOidList);
```

Parameters

pOidList A pointer to an object ID list returned by an MP API. With a successful return, the allocated memory is freed.

The client will free all `MP_OID_LIST` structures returned by any API by using this function.

Description The `MP_FreeOidList()` function frees memory returned by an MP API.

Return Values

`MP_STATUS_INVALID_PARAMETER` The *pOidList* is null or specifies a memory area to which data cannot be written.

`MP_STATUS_SUCCESS` The operation is successful.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also `libMPAPI(3LIB)`, `attributes(5)`

Multipath Management API Version 1.0

Name MP_GetAssociatedPathOidList – get a list of object IDs

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_GetAssociatedPathOidList(
     MP_OID oid, MP_OID MP_OID_LIST **ppList);
```

Parameters

oid The object ID of the multipath logical unit, initiator port, or target port.

ppList A pointer to a pointer to an object ID that has type `MP_OID_LIST` structure. With a successful return, this will contain a pointer to an object ID that has type `MP_OID_LIST` that contains the object IDs of all the paths associated with the specified (multipath) logical unit, initiator port, or target port *oid*.

Description The `MP_GetAssociatedPathOidList()` function gets a list of oid object IDs for all the path logica l units associated with the specified multipath logical unit , initiator port, or target port.

Returns a list of object IDs for all the path logical units associated with the specified mult ipath logical unit, initiator port, or target port.

When the caller is finished using the list it must free the memory used by the list by calling `MP_FreeOidList`.

Return Values

`MP_STATUS_INVALID_PARAMETER` The *ppList* is null or specifies a memory area to that the data cannot be written or when the *oid* has a type subfield other than `MP_OBJECT_TYPE_MULTI PATH_LU`, `MP_OBJECT_TYPE_INITIATOR_PORT`, or `MP_OBJECT_TYPE_TARGET_PORT`.

`MP_STATUS_INVALID_OBJECT_TYPE` The *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.

`MP_STATUS_OBJECT_NOT_FOUND` The *oid* owner ID or object sequence number is invalid.

`MP_STATUS_SUCCESS` The operation is successful.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also libMPAPI(3LIB), MP_GetPathLogicalUnitProperties(3MPAPI), attributes(5)

Multipath Management API Version 1.0

Name MP_GetAssociatedPluginOid – get the object ID for the plugin

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_GetAssociatedPluginOid(MP_OID oid,
     MP_OID  *pPluginOID);
```

Parameters

oid The object ID of an object that has been received from a previous API call.

pPluginOID A pointer to an object ID that has type `MP_OID` structure allocated by the caller. With a successful return this will contain the object ID of the plugin associated with the object specified by the *oid*.

Description The `MP_GetAssociatedPluginOid()` function gets the object ID for the plugin associated with the specified object ID. The sequence number subfield of the *oid* is not validate since this API is implemented in the common library.

Return Values

`MP_STATUS_INVALID_OBJECT_TYPE` The *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.

`MP_STATUS_INVALID_PARAMETER` The *pluginOid* is null or specifies a memory area to which data cannot be written.

`MP_STATUS_OBJECT_NOT_FOUND` The *oid* owner ID is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also `libMPAPI(3LIB)`, `attributes(5)`

Multipath Management API Version 1.0

Name MP_GetAssociatedTPGOidList – get a list of the object IDs

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_GetAssociatedTPGOidList(MP_OID oid,
     MP_OID_LIST **ppList);
```

Parameters

oid The object ID of the multipath logical unit.

ppList A pointer to a pointer to an object ID that has type `MP_OID_LIST` structure. With a successful return, this will contain a pointer to an object ID that has type `MP_OID_LIST` that contains the object IDs of target port groups associated with the specified logical unit.

Description The `MP_GetAssociatedTPGOidList()` function gets a list of the object IDs containing the target port group associated with the specified multipath logical unit.

When the caller is finished using the list, it must free the memory used by the list by calling `MP_FreeOidList`.

Return Values

`MP_STATUS_INVALID_OBJECT_TYPE`	The *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.
`MP_STATUS_INVALID_PARAMETER`	The *ppList* is null or specifies a memory area to which data cannot be written, or the *oid* has a type subfield other than `MP_OBJECT_TYPE_MULTIPATH_LU`.
`MP_STATUS_OBJECT_NOT_FOUND`	The *oid* owner ID or object sequence number is invalid.
`MP_STATUS_SUCCESS`	The operation is successful.
`MP_STATUS_FAILED`	The target port group list for the specified object ID is not found.
`MP_STATUS_INSUFFICIENT_MEMORY`	A memory allocation failure occurred.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also libMPAPI(3LIB), MP_GetTargetPortGroupProperties(3MPAPI), attributes(5)

Multipath Management API Version 1.0

Name MP_GetDeviceProductOidList – get a list of the object IDs

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_GetDeviceProductOidList(MP_OID oid,
     MP_OID_LIST **ppList);
```

Parameters

oid The object ID of the plugin.

ppList A pointer to a pointer to an object ID that has type `MP_OID_LIST` structure. With a successful return, this will contain a pointer to an object ID that has type `MP_OID_LIST` that contains the object IDs of all the device product descriptors associated with the specified plugin.

Description The `MP_GetDeviceProductOidList()` function gets a list of the object IDs of all the device product properties associated with this plugin. When the caller is finished using the list, it must free the memory used by the list by calling `MP_FreeOidList`.

Return Values

`MP_STATUS_INVALID_OBJECT_TYPE`	The *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.
`MP_STATUS_INVALID_PARAMETER`	The *ppList* is null or specifies a memory area to which data cannot be written because the *oid* has a type subfield other than`MP_OBJECT_TYPE_PLUGIN`.
`MP_STATUS_OBJECT_NOT_FOUND`	The *oid* owner ID or object sequence number is invalid.
`MP_STATUS_SUCCESS`	The operation is successful
`MP_STATUS_FAILED`	The plugin for the specified object ID is not found.
`MP_STATUS_INSUFFICIENT_MEMORY`	A memory allocation failure occurred.
`MP_STATUS_UNSUPPORTED`	The API is not supported.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also `libMPAPI(3LIB)`, `MP_GetDeviceProductProperties(3MPAPI)`, `attributes(5)`

Multipath Management API Version 1.0

Name MP_GetDeviceProductProperties – get the properties of a specified device product

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_GetDeviceProductProperties(MP_OID oid,
     MP_DEVICE_PRODUCT_PROPERTIES *pProps);
```

Parameters

oid The object ID of the device product.

pProps A pointer to an object ID that has type `MP_DEVICE_PR ODUCT_PROPERTIES` structure allocated by the caller. With a successful return, this structure contains the properties of the device product specified by the *oid*.

Description The `MP_GetDeviceProductProperties()` function gets the properties of the specified device product.

Return Values

`MP_STATUS_INVALID_OBJECT_TYPE`	The *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.
`MP_STATUS_OBJECT_NOT_FOUND`	The *oid* owner ID or object sequence number is invalid.
`MP_STATUS_INVALID_PARAMETER`	Returned when *pProps* is null or specifies a memory area to which data cannot be written, or the *oid* has a type subfield other than `MP_OBJECT_TYPE_DEVICE_PRODUCT`.
`MP_STATUS_SUCCESS`	The operation is successful.
`MP_STATUS_FAILED`	The plugin for the specified *oid* is not found.
`MP_STATUS_UNSUPPORTED`	The implementation does not support the API.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also `libMPAPI(3LIB)`, `MP_GetDeviceProductOidList(3MPAPI)`, `attributes(5)`

Multipath Management API Version 1.0

Name MP_GetInitiatorPortOidList – gets a list of the object IDs

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

 MP_STATUS MP_GetInitiatorPortOidList(MP_OID oid,
     MP_OID_LIST **ppList);
```

Parameters

oid The object ID of the plugin.

ppList A pointer to a pointer to an object ID that has type `MP_OID_LIST` structure. With a successful return, this contains a pointer to an`MP_OID_LIST` that contains the object IDs of all the initiator ports associated with the specified plugin.

Description The `MP_GetInitiatorPortOidList()` function gets a list of the object IDs of all the initiator ports associated with this plugin. When the caller is finished using the list it must free the memory used by the list by calling `MP_FreeOidList`.

Return Values

`MP_STATUS_INVALID_OBJECT_TYPE`	The *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.
`MP_STATUS_INVALID_PARAMETER`	The *pplist* is null or specifies a memory area to which data cannot be written, or when the *oid* has a type subfield other than `MP_OBJECT_TYPE_PLUGIN`.
`MP_STATUS_OBJECT_NOT_FOUND`	The *oid* owner ID or object sequence number is invalid.
`MP_STATUS_SUCCESS`	The operation is successful.
`MP_STATUS_INSUFFICIENT_MEMORY`	A memory allocation failure occurred.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also `libMPAPI(3LIB)`, `MP_GetInitiatorPortProperties(3MPAPI)`, `attributes(5)`

Multipath Management API Version 1.0

Name MP_GetInitiatorPortProperties – get initiator port properties

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_GetInitiatorPortProperties(MP_OID oid,
     MP_INITIATOR_PORT_PROPERTIES *pProps);
```

Parameters

oid The object ID of the Port.

pProps A pointer to an object ID that has type `MP_INITIATOR_PORT_PROPERTIES` structure allocated by the caller. With a successful return, this structure contains the properties of the port specified by the *oid* parameter.

Description The `MP_GetInitiatorPortProperties()` function gets the properties of the specified initiator port.

Return Values

`MP_STATUS_INVALID_PARAMETER`	The *pProps* is null or specifies a memory area to which data cannot be written, or when the *oid* has a type subfield other than `MP_OBJECT_TYPE_INITIATOR_PORT`.
`MP_STATUS_INVALID_OBJECT_TYPE`	The *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.
`MP_STATUS_OBJECT_NOT_FOUND`	The *oid* owner ID or object sequence number is invalid.
`MP_STATUS_SUCCESS`	The operation is successful.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also `libMPAPI(3LIB)`, `MP_GetInitiatorPortOidList(3MPAPI)`, `attributes(5)`

Multipath Management API Version 1.0

Name MP_GetLibraryProperties – get MP library properties

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_GetLibraryProperties(MP_LIBRARY_PROPERTIES *pProps);
```

Parameters

pProps A pointer to an object ID that has type `MP_LIBRARY_PROPERTIES` structure allocated by the caller. With a successful return, this structure contains the properties of the MP library currently in use.

Description The `MP_GetLibraryProperties()` function gets the properties of the MP library currently in use.

Return Values

`MP_STATUS_INVALID_PARAMETER` The *pProps* is null or specifies a memory area that cannot be written to.

`MP_STATUS_SUCCESS` The operation is successful.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also `libMPAPI(3LIB)`, `attributes(5)`

Multipath Management API Version 1.0

Name MP_GetMPLogicalUnitProperties – get logical unit properties

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_GetMPLogicalUnitProperties(MP_OID oid,
     MP_MULTIPATH_LOGICAL_UNIT_PROPERTIES *pProps);
```

Parameters

oid The object ID of the multipath logical unit.

pProps A pointer to an object ID that has type `MP_MULTIPATH_LOGICAL_UNIT_PROPERTIES` structure allocated by the caller. With a successful return, this structure contains the properties of the multipath logical unit specified by the object ID.

Description The `MP_GetMPLogicalUnitProperties()` function gets the properties of the specified logical unit.

Return Values

`MP_STATUS_INVALID_PARAMETER`	The *pProps* is null or specifies a memory area to which data cannot be written, or when the *oid* has a type subfield other than `MP_OBJECT_TYPE_MULTIPATH_LU`.
`MP_STATUS_INVALID_OBJECT_TYPE`	The *oid* does not specify any valid object type. This is most likely to happen if an uninitialized *oid* is passed to the API.
`MP_STATUS_OBJECT_NOT_FOUND`	The *oid* owner ID or object sequence number is invalid.
`MP_STATUS_SUCCESS`	The operation is successful.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also `libMPAPI(3LIB)`, `MP_GetMPLuOidListFromTPG(3MPAPI)`, `MP_GetMultipathLus(3MPAPI)`, `attributes(5)`

Multipath Management API Version 1.0

Name MP_GetMPLuOidListFromTPG – return a list of object IDs

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_GetMPLuOidListFromTPG(MP_OID oid,
     MP_OID_LIST **ppList);
```

Parameters

oid The object ID of the target port group.

ppList A pointer to a pointer to an object ID that has type `MP_OID_LIST` structure. With a successful return, this contains a pointer to an object ID that has type `MP_OID_LIST` that contains the object IDs of all the (multipath) logical units associated with the specified target port group.

Description The `MP_GetMPLuOidListFromTPG()` function returns the list of object IDs for the multipath logical units associated with the specific target port group.

When the caller is finished using the list, it must free the memory used by the list by calling `MP_FreeOidList`.

Return Values

`MP_STATUS_INVALID_OBJECT_TYPE`	The *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.
`MP_STATUS_INVALID_PARAMETER`	The *pplist* is null or specifies a memory area to which data cannot be written, or when the *oid* has a type subfield other than `MP_OBJECT_TYPE_TARGET_PORT`.
`MP_STATUS_OBJECT_NOT_FOUND`	The *oid* owner ID or object sequence number is invalid.
`MP_STATUS_SUCCESS`	The operation is successful.
`MP_STATUS_FAILED`	The multipath logical unit list for the specified target port group object ID is not found.
`MP_STATUS_INSUFFICIENT_MEMORY`	A memory allocation failure occurred.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also libMPAPI(3LIB), MP_GetMPLogicalUnitProperties(3MPAPI), attributes(5)

Multipath Management API Version 1.0

Name MP_GetMultipathLus – return a list of multipath logical units

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_GetMultipathLus(MP_OID oid, MP_OID_LIST **ppList);
```

Parameters

oid The object ID of the plugin or device product object.

ppList A pointer to a pointer to an object ID that has type `MP_OID_LIST` structure. With a successful return, this contains a pointer to an `MP_OID_LIST` that contains the object IDs of all the (multipath) logical units associated with the specified plugin object ID.

Description The `MP_GetMultipathLus()` function returns a list of multipath logical units associated to a plugin.

When the caller is finished using the list it must free the memory used by the list by calling `MP_FreeOidList`.

Return Values

`MP_STATUS_INVALID_OBJECT_TYPE`	The *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.
`MP_STATUS_INVALID_PARAMETER`	The *ppList* is null or specifies a memory area that cannot be written, or when *oid* has a type subfield other than `MP_OBJECT_TYPE_DEVICE_PRODUCT` or `MP_OBJECT_TYPE_PLUG IN`.
`MP_STATUS_OBJECT_NOT_FOUND`	The *oid* owner ID or object sequence number is invalid.
`MP_STATUS_SUCCESS`	The operation is successful.
`MP_STATUS_FAILED`	The plugin for the specified object ID is not found.
`MP_STATUS_INSUFFICIENT_MEMORY`	A memory allocation failure occurred.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also `libMPAPI(3LIB)`, `MP_GetMPLogicalUnitProperties(3MPAPI)`, `attributes(5)`

Multipath Management API Version 1.0

Name MP_GetObjectType – get an object type

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_GetObjectType(MP_OID oid, MP_OBJECT_TYPE *pObjectType);
```

Parameters

oid	The initialized object ID to have the type determined.
pObjectType	A pointer to an object ID that has type `MP_OBJECT_TYPE` variable allocated by the caller. With a successful return it contains the object type of *oid*.

Description The `MP_GetObjectType()` function gets the object type of an initialized object ID.

This API is provided so that clients can determine the type of object an object ID represents. This can be very useful for a client function that receives notifications.

Return Values

`MP_STATUS_INVALID_OBJECT_TYPE`	The *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to th e API.
`MP_STATUS_INVALID_PARAMETER`	The *pObjectType* is null or specifies a memory area to which data cannot be written.
`MP_STATUS_SUCCESS`	The operation is successful.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also `libMPAPI(3LIB)`, `MP_RegisterForObjectVisibilityChanges(3MPAPI)`, `attributes(5)`

Multipath Management API Version 1.0

Name MP_GetPathLogicalUnitProperties – get the specified path properties

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_GetPathLogicalUnitProperties(MP_OID oid,
    MP_PATH_LOGICAL_UNIT_PROPERTIES *pProps);
```

Parameters

oid The object ID of the logical unit path.

pProps A pointer to an object ID that has type `MP_PATH_LOGICAL_UNIT_PROPERTIES` structure allocated by the caller. With a successful return, this structure contains the properties of the path specified by *oid*.

Description The `MP_GetPathLogicalUnitProperties()` function gets the properties of the specified path.

Return Values

`MP_STATUS_INVALID_PARAMETER` The *pProps* is null or specifies a memory area to which data cannot be written, or when the *oid* has a type subfield other than `MP_OBJECT_TYPE_PATH_LU`.

`MP_STATUS_INVALID_OBJECT_TYPE` The *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.

`MP_STATUS_OBJECT_NOT_FOUND` The *oid* owner ID or object sequence number is invalid.

`MP_STATUS_SUCCESS` The operation is successful.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also `libMPAPI(3LIB)`, `MP_GetAssociatedPathOidList(3MPAPI)`, `attributes(5)`

Multipath Management API Version 1.0

Name MP_GetPluginOidList – get a list of the object IDs

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_GetPluginOidList(MP_OID_LIST **ppList);
```

Parameters

ppList A pointer to a pointer to an object ID that has type `MP_OID_LIST`. With a successful return, this contains a pointer to an object ID that has type `MP_OID_LIST` that contains the object IDs of all of the plugins currently loaded by the library.

Description The `MP_GetPluginOidList()` function returns a list of the object IDs of all currently loaded plugins. The returned list is guaranteed to not contain any duplicate entries.

When the caller is finished using the list it must free the memory used by the list by calling `MP_FreeOidList(3MPAPI)`.

Return Values

`MP_STATUS_INVALID_PARAMETER`	The *ppList* is null or specifies a memory area to which data cannot be written.
`MP_STATUS_SUCCESS`	The operation is successful.
`MP_STATUS_FAILED`	The plugin for the specified object ID is not found.
`MP_STATUS_INSUFFICIENT_MEMORY`	A memory allocation failure occurred.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also `libMPAPI(3LIB)`, `MP_FreeOidList(3MPAPI)`, `MP_GetPluginProperties(3MPAPI)`, `attributes(5)`

Multipath Management API Version 1.0

Name MP_GetPluginProperties – get specified plugin properties

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_GetPluginProperties(MP_OID oid,
     MP_PLUGIN_PROPERTIES *pProps);
```

Parameters

oid The object ID of the plugin.

pProps A pointer to an object ID that has type `MP_PLUGIN_PR OPERTIES` structure allocated by the caller. With a successful return, this structure contains the properties of the plugin specified by *oid*.

Description The `MP_GetPluginProperties()` function gets the properties of the specified plugin.

Return Values

`MP_STATUS_INVALID_PARAMETER` The *pProps* is null or specifies a memory area to which data cannot be written, or the *oid* has a type subfield other than `MP_OBJECT_TYPE_PLUGIN`.

`MP_STATUS_INVALID_OBJECT_TYPE` The *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.

`MP_STATUS_OBJECT_NOT_FOUND` The *oid* owner ID or object sequence number is invalid.

`MP_STATUS_SUCCESS` The operation is successful.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also `libMPAPI(3LIB)`, `MP_GetProprietaryLoadBalanceProperties(3MPAPI)`, `MP_GetPluginOidList(3MPAPI)`, `attributes(5)`

Multipath Management API Version 1.0

Name MP_GetProprietaryLoadBalanceOidList – get a list of object IDs

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_GetProprietaryLoadBalanceOidList(MP_OID oid
     MP_OID_LIST **ppList);
```

Parameters

oid The object ID of the plugin.

ppList A pointer to a pointer to an object ID that has type `MP_OID_LIST` structure. With a successful return, this contains a pointer to an object ID that has type `MP_OID_LIST` that contains the object IDs of all the proprietary load balance types associated with the specified plugin.

Description The `MP_GetProprietaryLoadBalanceOidList()` function returns a list of the object IDs of all the proprietary load balance algorithms associated with this plugin.

When the caller is finished using the list, it must free the memory used by the list by calling `MP_FreeOidList`.

Return Values

`MP_STATUS_INVALID_OBJECT_TYPE`	The *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.
`MP_STATUS_INVALID_PARAMETER`	The *ppList* in null or specifies a memory area to which data cannot be written, or if the *oid* has a type subfield other than `MP_OBJECT_TYPE_PLUGIN`.
`MP_STATUS_OBJECT_NOT_FOUND`	The *oid* owner ID or object sequence number is invalid.
`MP_STATUS_SUCCESS`	The operation is successful.
`MP_STATUS_FAILED`	The plugin for the specified object ID is not found.
`MP_STATUS_INSUFFICIENT_MEMORY`	A memory allocation failure occurred.
`MP_STATUS_UNSUPPORTED`	The implementation does not support the API.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also libMPAPI(3LIB), MP_GetProprietaryLoadBalanceProperties(3MPAPI), attributes(5)

Multipath Management API Version 1.0

Name MP_GetProprietaryLoadBalanceProperties – get load balance properties

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_GetProprietaryLoadBalanceProperties(MP_OID oid,
     MP_PROPRIETARY_LOAD_BALANCE_PROPERTIES *pProps);
```

Parameters

oid The object ID of the proprietary load balance.

pProps A pointer to an object ID that has type `MP_PROPRIETARY_LOAD_BALANCE_PROPERTIES` structure allocated by the caller. With a successful return, this structure contains the properties of the proprietary load balance algorithm specified by the *oid*.

Description The `MP_GetProprietaryLoadBalanceProperties()` function returns the properties of the specified load balance.

Return Values

`MP_STATUS_INVALID_PARAMETER`
The *pObjectType* is null or specifies a memory area to which data cannot be written, or when the *oid* has a type subfield other than `MP_OBJECT_TYPE_PROPRIETARY_LOAD_BALANCE`.

`MP_STATUS_INVALID_OBJECT_TYPE`
The *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.

`MP_STATUS_OBJECT_NOT_FOUND`
The *oid* owner ID or object sequence number is invalid.

`MP_STATUS_SUCCESS`
The operation is successful.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also `libMPAPI(3LIB)`, `MP_GetProprietaryLoadBalanceOidList(3MPAPI)`, `attributes(5)`

Multipath Management API Version 1.0

Name MP_GetTargetPortGroupProperties – return properties of the target port group

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_GetTargetPortGroupProperties(MP_OID oid,
     MP_TARGET_PORT_GROUP_PROPERTIES *pProps);
```

Parameters

oid The object ID of the target port group.

pProps A pointer to an object ID that has type `MP_TARGET_PORT_GROUP_PROPERTIES` structure allocated by the caller. With a successful return, this structure contains the properties of the target port group specified by the *oid*.

Description The `MP_GetTargetPortGroupProperties()` function returns the properties of the specified target port group.

Return Values

`MP_STATUS_INVALID_PARAMETER`
The *pProps* is null or specifies a memory area to which data cannot be written, or when the *oid* has a type subfield other than `MP_OBJECT_TYPE_TARGET_PORT_GROUP`.

`MP_STATUS_INVALID_OBJECT_TYPE`
The *oid* does not specify a valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.

`MP_STATUS_OBJECT_NOT_FOUND`
The *oid* owner ID or object sequence number is invalid.

`MP_STATUS_SUCCESS`
The operation is successful.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also `libMPAPI(3LIB)`, `MP_GetAssociatedTPGOidList(3MPAPI)`, `attributes(5)`

Multipath Management API Version 1.0

Name MP_GetTargetPortOidList – get a list of target port object IDs

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_GetTargetPortOidList(MP_OID oid,
     MP_OID_LIST **ppList);
```

Parameters

oid The object ID of the target port group.

ppList A pointer to a pointer to an object ID that has type `MP_OID_LIST` structure. With a successful return, this contains a pointer to an object ID that has type `MP_OID_LIST` that contains the object IDs of all the target ports associated with the specified target port group *oid*.

Description The `MP_GetTargetPortOidList()` function returns a list of the object IDs of the target ports in the specified target port group.

When the caller is finished using the list it must free the memory used by the list by calling `MP_FreeOidList`.

Return Values

`MP_STATUS_INVALID_OBJECT_TYPE`	The *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.
`MP_STATUS_INVALID_PARAMETER`	The *ppList* is null or specifies a memory area to which data cannot be written, or when the *oid* has a type subfield other than `MP_OBJECT_TYPE_TARGET_PORT`.
`MP_STATUS_OBJECT_NOT_FOUND`	The *oid* owner ID or object sequence number is invalid.
`MP_STATUS_SUCCESS`	The operation is successful.
`MP_STATUS_FAILED`	The target port group for the specified object ID is not found.
`MP_STATUS_INSUFFICIENT_MEMORY`	A memory allocation failure occurred.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also libMPAPI(3LIB), MP_GetTargetPortProperties(3MPAPI), attributes(5)

Multipath Management API Version 1.0

Name MP_GetTargetPortProperties – get targrt port properties

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_GetTargetPortProperties(MP_OID oid,
     MP_TARGET_PORT_GROUP_PROPERTIES *pProps);
```

Parameters

oid The object ID of the target port group.

pProps A pointer to an object ID that has type `MP_TARGET_PORT_GROUP_PROPERTIES` structure allocated by the caller. With a successful return, this structure contains the properties of the target port group specified by the *oid*.

Description The `MP_GetTargetPortProperties()` function returns the properties of the specified target port.

Return Values

`MP_STATUS_INVALID_PARAMETER`	The *pProps* is null or specifies a memory area to which data cannot be written or when the *oid* has a type subfield other than `MP_OBJECT_TYPE_TARGET_PORT`.
`MP_STATUS_INVALID_OBJECT_TYPE`	The *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.
`MP_STATUS_OBJECT_NOT_FOUND`	The *oid* owner ID or object sequence number is invalid.
`MP_STATUS_SUCCESS`	The operation is successful.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also `libMPAPI(3LIB)`, `MP_GetTargetPortOidList(3MPAPI)`, `attributes(5)`

Multipath Management API Version 1.0

Name MP_RegisterForObjectPropertyChanges – register a client function to be called

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_RegisterForObjectPropertyChanges(
     MP_OBJECT_PROPERTY_FN pClientFn, MP_OBJECT_TYPE objectType,
     void *pCallerData, MP_OID pluginOid);
```

Parameters

pClientFn	A pointer to an object ID that has type `MP_OBJECT_PROPERTY_FN` function defined by the client. With a successful return, this function is called to inform the client of objects that have had one or more properties changed.
objectType	The type of object the client wishes to register for property change callbacks.
pCallerData	A pointer that is passed to the callback routine with each event. This might be used by the caller to correlate the event to the source of the registration.
pluginOid	If this is a valid plugin object ID, then registration is limited to that plugin. If this is zero, then the registration is for all plugins.

Description The `MP_RegisterForObjectPropertyChanges()` function registers a client function to be called whenever the property of an object changes.

The function specified by *pClientFn* is called whenever the property of an object changes. For the purposes of this function, a property is defined to be a field in an object's property structure and the object's status. Therefore, the client function is not called if a statistic of the associated object changes. But, it is called when the status changes (e.g., from working to failed) or when a name or other field in a property structure changes.

It is not an error to re-register a client function. However, a client function has only one registration. The first call to deregister a client function will deregister it no matter how many calls to register the function have been made.

If multiple properties of an object change simultaneously, a client function can be called only once to be notified that all the changes have occurred.

Return Values

`MP_STATUS_INVALID_OBJECT_TYPE`	The *pluginOid* or *objectType* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.
`MP_STATUS_OBJECT_NOT_FOUND`	The *pluginOid* owner ID or object sequence number is invalid.
`MP_STATUS_INVALID_PARAMETER`	The *pCallerData* is null or if the *pluginOid* has a type subfield other than `MP_OBJECT_TYPE_PLUGIN`, or when *objectType* is invalid.
`MP_STATUS_SUCCESS`	The operation is successful.

`MP_STATUS_FN_REPLACED`	An existing client function is replaced with the one specified in *pClientFn*.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also `libMPAPI(3LIB)`, `MP_DeregisterForObjectPropertyChanges(3MPAPI)`, `attributes(5)`

Multipath Management API Version 1.0

Name MP_RegisterForObjectVisibilityChanges – register a client function to be called

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_RegisterForObjectVisibilityChanges(
     MP_OBJECT_PROPERTY_FN pClientFn, MP_OBJECT_TYPE objectType,
     void *pCallerData, MP_OID pluginOid);
```

Parameters

pClientFn	A pointer to an object ID that has type `MP_OBJECT_VISIBILITY_FN` function defined by the client. With a successful return, this function is called to inform the client of objects that have had one or more properties changed.
objectType	The type of object the client wishes to register for property change callbacks.
pCallerData	A pointer that is passed to the callback routine with each event. This might be used by the caller to correlate the event to the source of the registration.
pluginOid	If this is a valid plugin object ID, then registration is limited to that plugin. If this is zero, then the registration is for all plugins.

Description The `MP_RegisterForObjectVisibilityChanges()` function registers a client function to be called whenever the property of an object changes. The function specified by *pClientFn* is called whenever objects appear or disappear.

It is not an error to re-register a client function. However, a client function has only one registration. The first call to deregister a client function will deregister it no matter how many calls to register the function have been made.

Return Values

`MP_STATUS_INVALID_OBJECT_TYPE`	The *pluginOid* or *objectType* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.
`MP_STATUS_OBJECT_NOT_FOUND`	The *pluginOid* owner ID or object sequence number is invalid.
`MP_STATUS_INVALID_PARAMETER`	The *pCallerData* is null or if the *pluginOid* has a type subfield other than `MP_OBJECT_TYPE_PLUGIN`, or when *objectType* is invalid.
`MP_STATUS_SUCCESS`	The operation is successful.
`MP_STATUS_FN_REPLACED`	An existing client function is replaced with the one specified in *pClientFn*.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also libMPAPI(3LIB), MP_DeregisterForObjectVisibilityChanges(3MPAPI), attributes(5)

Multipath Management API Version 1.0

Name MP_RegisterPlugin – register a plugin with the common library

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_RegisterPlugin(MP_WCHAR *pPluginId,
     MP_CHAR *pFileName);
```

Parameters

pPluginId A pointer to the key name shall be the reversed domain name of the vendor followed by a ".", followed by the vendor-specific name for the plugin that uniquely identifies it.

pFileName The full path name of the plugin library.

Description The `MP_RegisterPlugin()` function registers a plugin with the common library. The current implementation adds an entry to the `/etc/mpapi.conf` file.

Unlike some other APIs, this API is implemented entirely in the common library. It must be called before a plugin is invoked by the common library.

This API does not impact dynamically add or change plugins bound to a running library instance. Instead, it causes an application that is currently not using a plugin to access the specified plugin on future calls to the common library. This is generally the behavior expected from dynamically loaded modules.

This API is typically called by a plugin's installation software to inform the common library of the path for the plugin library.

It is not an error to re-register a plugin. However, a plugin has only one registration. The first call to deregister a plugin will deregister it no matter how many calls to register the plugin have been made.

A vendor may register multiple plugins by using separate plugin IDs and filenames.

Return Values

`MP_STATUS_INVALID_PARAMETER` The *pFileName* does not exist.

`MP_STATUS_SUCCESS` The operation is successful.

Files `/etc/mpapi.conf` MPAPI library configuration file

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also libMPAPI(3LIB), MP_DeregisterPlugin(3MPAPI), mpapi.conf(4), attributes(5)

Multipath Management API Version 1.0

Name MP_SetFailbackPollingRate – set the polling rates

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_SetFailbackPollingRate(MP_OID oid,
     MP_UINT32 pollingRate);
```

Parameters

oid An object ID of either the plugin or a multipath logical unit.

pollingRate The value to be set in `MP_PLUGIN_PROPERTIES` *currentFailbackPollingRate* or `MP_MULTIPATH_LOGICAL_UNIT_PROPERTIES` *failbackPollingRate*.

Description The `MP_SetFailbackPollingRate()` function sets the polling rates. Setting the *pollingRate* to zero disables polling.

If the object ID refers to a plugin, this sets the *currentFailbackPollingRate* property in the plugin properties. If the object ID refers to a multipath logical unit, this sets the *failbackPollingRate* property.

Return Values

`MP_STATUS_INVALID_OBJECT_TYPE` The *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.

`MP_STATUS_INVALID_PARAMETER` One of the polling values is outside the range supported by the driver, or when the *oid* has a type subfield other than `MP_OBJECT_TYPE_PLUGIN` or `MP_OBJECT_TYPE_MULTIPATH_LU`.

`MP_STATUS_OBJECT_NOT_FOUND` The *oid* ownerID or object sequence number is invalid.

`MP_STATUS_SUCCESS` The operation is successful.

`MP_STATUS_UNSUPPORTED` The implementation does not support the API.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also `libMPAPI(3LIB)`, `attributes(5)`

Multipath Management API Version 1.0

Name MP_SetLogicalUnitLoadBalanceType – set a load balancing policy

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_SetLogicalUnitLoadBalanceType(MP_OID logicalUnitoid,
     MP_LOAD_BALANCE_TYPE  loadBalance);
```

Parameters

logicalUnitOid The object ID of the multipath logical unit.

loadBalance The desired load balance policy for the specified logical unit.

Description The `MP_SetLogicalUnitLoadBalanceType()` function sets the multipath logical unit's load balancing policy. The value must correspond to one of the supported values in `MP_PLUGIN_PROPERTIES.SupportedLogicalUnitLoadBalanceTypes`.

Return Values

`MP_STATUS_INVALID_OBJECT_TYPE`	The *logicalUnitOid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.
`MP_STATUS_INVALID_PARAMETER`	The *loadBalance* is invalid or *logicalUnitOid* has a type subfield other than `MP_OBJECT_TYPE_MULTIPATH_LU`.
`MP_STATUS_OBJECT_NOT_FOUND`	The *logicalUnitOid* owner ID or object sequence number is invalid.
`MP_STATUS_SUCCESS`	The operation is successful.
`MP_STATUS_FAILED`	The specified *loadBalance* type cannot be handled by the plugin. One possible reason for this is a request to set `MP_LOAD_BALANCE_TYPE_PRODUCT` when the specified logical unit has no corresponding `MP_DEVICE_PRODUCT_PROPERTIES` instance (i.e., the plugin does not have a product-specific load balance algorithm for the LU product).
`MP_STATUS_UNSUPPORTED`	The implementation does not support the API.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also libMPAPI(3LIB), attributes(5)

Multipath Management API Version 1.0

Name MP_SetOverridePath – manually override a logical unit path

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_SetOverridePath(MP_OID logicalUnitOid,
     MP_OID pathOid);
```

Parameters

logicalUnitOid The object ID of the multipath logical unit.

pathOid The object ID of the path logical unit.

Description The `MP_SetOverridePath()` function is used to manually override the path for a logical unit. The path is exclusively used to access the logical unit until cleared. Use `MP_CancelOverride` to clear the override.

This API allows the administrator to disable the dri ver's load balance algorithm and force all I/O operations to a specific path. The existing path weight configuration is maintained. If the administrator undoes the override (by calling `MP_CancelOverridePath`), the driver starts load balancing based on the weights of available paths (and target port group access state for asymmetric devices).

If the multipath logical unit is part of a target with asymmetrical access, executing this command could cause failover.

Return Values

`MP_STATUS_INVALID_OBJECT_TYPE` The *logicalUnitOid* or *pathOid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.

`MP_STATUS_INVALID_PARAMETER` The *logicalUnitOid* has a type subfield other than `MP_OBJECT_TYPE_MULTIPATH_LU`, or if *pathOid* has an object type other than `MP_OBJECT_TYPE_PATH_LU`.

`MP_STATUS_OBJECT_NOT_FOUND` The *logicalUnitOid*, *pathOid* owner ID, or object sequence number is invalid.

`MP_STATUS_SUCCESS` The operation is successful.

`MP_STATUS_UNSUPPORTED` The implementation does not support the API.

`MP_STATUS_PATH_NONOPERATIONAL` The driver cannot communicate through selected path

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also libMPAPI(3LIB), attributes(5)

Multipath Management API Version 1.0

Name MP_SetPathWeight – set the weight of a path

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_SetPathWeight(MP_OID pathOid, MP_UINT32 weight);
```

Parameters

pathOid The object ID of the path logical unit.

weight A weight that will be assigned to the path logical unit.

Description The `MP_SetPathWeight()` function sets the weight to be assigned to a particular path.

Return Values

`MP_STATUS_INVALID_OBJECT_TYPE`	The *pathOid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.
`MP_STATUS_OBJECT_NOT_FOUND`	The *pathOid* ownerID or object sequence number is invalid.
`MP_STATUS_INVALID_PARAMETER`	The *pathOid* has a type subfield other than `MP_OBJECT_TYPE_PATH_LU`, or when the weight parameter is greater than the plugin's maximum weight property.
`MP_STATUS_SUCCESS`	The operation is successful.
`MP_STATUS_FAILED`	The operation failed.
`MP_STATUS_UNSUPPORTED`	The driver does not support setting path weight.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also `libMPAPI(3LIB)`, `attributes(5)`

Multipath Management API Version 1.0

Name MP_SetPluginLoadBalanceType – set the plugin default load balance policy

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_SetPluginLoadBalanceType(MP_OID oid,
     MP_LOAD_BALANCE_TYPE loadBalance);
```

Parameters

oid The object ID of the plugin.

loadBalance The desired default load balance policy for the specified plugin.

Description The `MP_SetPluginLoadBalanceType()` function sets the default load balance policy for the plugin. The value must correspond to one of the supported values in `MP_PLUGIN_PROPERTIES.SupportedPluginLoadBalanceTypes`.

Return Values

`MP_STATUS_INVALID_OBJECT_TYPE`	The *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.
`MP_STATUS_INVALID_PARAMETER`	The *loadBalance* is invalid or when the *oid* has a type subfield other than `MP_OBJECT_TYPE_PLUGIN`.
`MP_STATUS_OBJECT_NOT_FOUND`	The *oid* ownerID or sequence number is invalid.
`MP_STATUS_SUCCESS`	The operation is successful.
`MP_STATUS_FAILED`	The specified *loadBalance* type cannot be handled by the plugin.
`MP_STATUS_UNSUPPORTED`	The implementation does not support the API.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also `libMPAPI(3LIB)`, `attributes(5)`

Multipath Management API Version 1.0

Name MP_SetProbingPollingRate – set the polling rate

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_SetProbingPollingRate(MP_OID oid,
     MP_UINT32 pollingRate);
```

Parameters

oid An object ID of either the plugin or a multipath logical unit.

pollingRate The value to be set in `MP_PLUGIN_PROPERTIES` *currentProbingPollingRate* or `MP_MULTIPATH_LOGICAL_UNIT_PROPERTIES` *ProbingPollingRate*.

Description The `MP_SetProbingPollingRate()` function sets the polling rates. Setting the *pollingRate* to zero disables polling.

If the object ID refers to a plugin, this sets the *currentProbingPollingRate* property in the plugin properties. If the object ID refers to a multipath logical unit, this sets the *ProbingPollingRate* property.

Return Values

`MP_STATUS_INVALID_OBJECT_TYPE` The *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.

`MP_STATUS_INVALID_PARAMETER` One of the polling values is outside the range supported by the driver or when the *oid* has a type subfield other than `MP_OBJECT_TYPE_PLUGIN` or `MP_OBJECT_TYPE_MULTIPATH_LU`.

`MP_STATUS_OBJECT_NOT_FOUND` The *oid* ownerID or sequence number is invalid.

`MP_STATUS_SUCCESS` The operation is successful.

`MP_STATUS_UNSUPPORTED` The implementation does not support the API.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also `libMPAPI(3LIB)`, `attributes(5)`

Multipath Management API Version 1.0

Name MP_SetProprietaryProperties – set proprietary properties

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_SetProprietaryProperties(MP_OID oid,
     MP_UINT32 count, MP_PROPRIETARY_PROPERTY *pPropertyList);
```

Parameters

oid The object ID representing an object ID that has type `MP_LOAD_BALANCE_PROPIETARY_TYPE`, or `MP_PLUGIN_PROPERTIES`, or `MP_MULTIPATH_LOGICAL_UNIT_PROPERTIES` instance.

count The number of valid items in *pPropertyList*.

pPropertyList A pointer to an array of property name/value pairs. This array must contain the same number of elements as does *count*.

Description The `MP_SetProprietaryProperties()` function sets proprietary properties in supported object instances.

This API allows an application with a priori knowledge of proprietary plugin capabilities to set proprietary properties. The *pPropertyList* is a list of property name/value pairs. The property names shall be a subset of the proprietary property names listed in the referenced object ID.

Return Values

`MP_STATUS_INVALID_OBJECT_TYPE`
The *oid* does not specify a valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.

`MP_STATUS_OBJECT_NOT_FOUND`
The *oid* owner ID or object sequence number is invalid.

`MP_STATUS_INVALID_PARAMETER`
The *pPropertyList* is null, or when one of the properties referenced in the list is not associated with the specified object ID, or the *oid* has a type subfield other than`MP_OBJECT_TYPE_PROPRIETARY_LOAD_BALANCE`, or `MP_OBJECT_TYPE_PLUGIN`, or `MP_OBJECT_TYPE_MULTIPATH_LU`.

`MP_STATUS_SUCCESS`
The operation is successful.

`MP_STATUS_UNSUPPORTED`
The API is not supported.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also `libMPAPI(3LIB)`, `attributes(5)`

Multipath Management API Version 1.0

Name MP_SetTPGAccess – set a target port group access state

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>

MP_STATUS MP_SetTPGAccess(MP_OID luOid, MP_UINT32 count,
     MP_TPG_STATE_PAIR *pTpgStateList);
```

Parameters

luOid	An object ID that has type `MP_MULTIPATH_LOGICAL_UNIT`.
count	The number of valid items in the *pTpgStateList*.
pTpgStateList	A pointer to an array of data structure `MP_TPG_STATE_PAIR`. This array must contain the same number of elements as *count*.

Description The `MP_SetTPGAccess()` function sets the access state for a list of target port groups. This allows a client to force a failover or failback to a desired set of target port groups. This is only valid for devices that support explicit access state manipulation (i.e., the field *explicitFailover* of data structure `MP_TARGET_PORT_GROUP_PROPERTIES` must be true).

This API provides the information needed to set up a `SCSI SET TARGET PORT GROUPS` command.

The plugin should not implement this API by directly calling the `SCSI SET TARGET PORT GROUPS` command. The plugin should use the MP drivers API (for example, `ioctl`) if available.

There are two reasons why this API is restricted to devices supporting explicit failover commands. Without an explicit command, the behavior of failback tends to be device-specific.

When the caller is finished using the list it must free the memory used by the list by calling `MP_FreeOidList`.

Return Values

`MP_STATUS_ACCESS_STATE_INVALID`
: The target device returns a status indicating the caller is attempting to establish an illegal combination of access states.

`MP_STATUS_FAILED`
: The underlying interface failed the command for some reason other than `MP_STATUS_ACCESS_STATE_INVALID`.

`MP_STATUS_INVALID_OBJECT_TYPE`
: The *luOid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.

`MP_STATUS_OBJECT_NOT_FOUND`
: The *luOid* owner ID or object sequence number is invalid.

`MP_STATUS_INVALID_PARAMETER`
The *pTpgStateList* is null, or when one of the TPGs referenced in the list is not associated with the specified MP logical unit, or the *luOid* has a type subfield other than `MP_OBJECT_TYPE_MULTIPATH_LU`.

`MP_STATUS_SUCCESS`
The operation is successful.

`MP_STATUS_UNSUPPORTED`
The API is not supported.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard: ANSI INCITS 412 Multipath Management API
MT-Level	Safe

See Also `libMPAPI(3LIB)`, `attributes(5)`

Multipath Management API Version 1.0

Name m_setvalues_layout – set layout values of a `LayoutObject`

Synopsis

```
cc [ flag... ] file... –llayout [ library... ]
#include <sys/layout.h>

int m_setvalues_layout(LayoutObject layout_object,
     const LayoutValues values, int *index_returned);
```

Description The `m_setvalues_layout()` function changes the layout values of a LayoutObject.

The *layout_object* argument specifies a `LayoutObject` returned by the `m_create_layout(3LAYOUT)` function.

The *values* argument specifies the list of layout values that are to be changed. The values are written into the `LayoutObject` and may affect the behavior of subsequent layout functions. Some layout values do alter internal states maintained by a `LayoutObject`.

The `m_setvalues_layout()` function can be implemented as a macro that evaluates the first argument twice.

Return Values Upon successful completion, the requested layout values are set and `0` is returned. Otherwise `–1` is returned and `errno` is set to indicate the error. If any value cannot be set, none of the layout values are changed and the (zero-based) index of the first value causing the error is returned in *index_returned*.

Errors The `m_setvalues_layout()` function may fail if:

`EINVAL` The layout value specified by *index_returned* is unknown, its value is invalid, or the *layout_object* argument is invalid.

`EMFILE` There are `{OPEN_MAX}` file descriptors currently open in the calling process.

Usage Do not use expressions with side effects such as auto-increment or auto-decrement within the first argument to the `m_setvalues_layout()` function.

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

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

See Also `m_create_layout(3LAYOUT)`, `attributes(5)`, `standards(5)`

Name m_transform_layout – layout transformation

Synopsis

```
cc [ flag... ] file... –llayout [ library... ]
#include <sys/layout.h>


int m_transform_layout(LayoutObject layout_object,
     const char *InpBuf, const size_t ImpSize, const void *OutBuf,
     size_t *Outsize, size_t *InpToOut, size_t *OutToInp,
     unsigned char *Property, size_t *InpBufIndex);
```

Description The `m_transform_layout()` function performs layout transformations (reordering, shaping, cell determination) or provides additional information needed for layout transformation (such as the expected size of the transformed layout, the nesting level of different segments in the text and cross-references between the locations of the corresponding elements before and after the layout transformation). Both the input text and output text are character strings.

The `m_transform_layout()` function transforms the input text in *InpBuf* according to the current layout values in *layout_object*. Any layout value whose value type is `LayoutTextDescriptor` describes the attributes of the *InpBuf* and *OutBuf* arguments. If the attributes are the same for both *InpBuf* and *OutBuf*, a null transformation is performed with respect to that specific layout value.

The *InpBuf* argument specifies the source text to be processed. The *InpBuf* may not be `NULL`, unless there is a need to reset the internal state.

The *InpSize* argument is the number of bytes within *InpBuf* to be processed by the transformation. Its value will not change after return from the transformation. *InpSize* set to −1 indicates that the text in *InpBuf* is delimited by a null code element. If *InpSize* is not set to −1, it is possible to have some null elements in the input buffer. This might be used, for example, for a "one shot" transformation of several strings, separated by nulls.

Output of this function may be one or more of the following depending on the setting of the arguments:

OutBuf Any transformed data is stored in *OutBuf*, converted to `ShapeCharset`.

Outsize The number of bytes in *OutBuf*.

InpToOut A cross-reference from each *InpBuf* code element to the transformed data. The cross-reference relates to the data in *InpBuf* starting with the first element that *InpBufIndex* points to (and not necessarily starting from the beginning of the *InpBuf*).

OutToInp A cross-reference to each *InpBuf* code element from the transformed data. The cross-reference relates to the data in *InpBuf* starting with the first element that *InpBufIndex* points to (and not necessarily starting from the beginning of the *InpBuf*).

Property A weighted value that represents peculiar input string transformation properties with different connotations as explained below. If this argument is not a null pointer, it represents an array of values with the same number of elements as the source substring text before the transformation. Each byte will contain relevant "property" information of the corresponding element in *InpBuf* starting from the element pointed by *InpBufIndex*. The four rightmost bits of each "property" byte will contain information for bidirectional environments (when `ActiveDirectional` is True) and they will mean "`NestingLevels`." The possible value from 0 to 15 represents the nesting level of the corresponding element in the *InpBuf* starting from the element pointed by *InpBufIndex*. If `ActiveDirectional` is false the content of `NestingLevel` bits will be ignored. The leftmost bit of each "property" byte will contain a "new cell indicator" for composed character environments, and will have a value of either 1 (for an element in *InpBuf* that is transformed to the beginning of a new cell) or 0 (for the "zero-length" composing character elements, when these are grouped into the same presentation cell with a non-composing character). Here again, each element of "property" pertains to the elements in the *InpBuf* starting from the element pointed by *InpBufIndex*. (Remember that this is not necessarily the beginning of *InpBuf*). If none of the transformation properties is required, the argument *Property* can be `NULL`. The use of "property" can be enhanced in the future to pertain to other possible usage in other environments.

The *InpBufIndex* argument is an offset value to the location of the transformed text. When `m_transform_layout()` is called, *InpBufIndex* contains the offset to the element in *InpBuf* that will be transformed first. (Note that this is not necessarily the first element in *InpBuf*). At the return from the transformation, *InpBufIndex* contains the offset to the first element in the *InpBuf* that has not been transformed. If the entire substring has been transformed successfully, *InpBufIndex* will be incremented by the amount defined by *InpSize*.

Each of these output arguments may be `NULL` to specify that no output is desired for the specific argument, but at least one of them should be set to a non-null value to perform any significant work.

The layout object maintains a directional state that keeps track of directional changes, based on the last segment transformed. The directional state is maintained across calls to the layout transformation functions and allows stream data to be processed with the layout functions. The directional state is reset to its initial state whenever any of the layout values `TypeOfText`, `Orientation`, or `ImplicitAlg` is modified by means of a call to `m_setvalues_layout()`.

The *layout_object* argument specifies a `LayoutObject` returned by the `m_create_layout()` function.

The *OutBuf*argument contains the transformed data. This argument can be specified as a null pointer to indicate that no transformed data is required.

The encoding of the *OutBuf* argument depends on the `ShapeCharset` layout value defined in *layout_object*. If the `ActiveShapeEditing` layout value is not set (False), the encoding of *OutBuf* is guaranteed to be the same as the codeset of the locale associated with the `LayoutObject` defined by *layout_object*.

On input, the *OutSize* argument specifies the size of the output buffer in number of bytes. The output buffer should be large enough to contain the transformed result; otherwise, only a partial transformation is performed. If the `ActiveShapeEditing` layout value is set (True) the *OutBuf* should be allocated to contain at least the *InpSize* multiplied by `ShapeCharsetSize`.

On return, the *OutSize* argument is modified to the actual number of bytes placed in *OutBuf*.

When the *OutSize* argument is specified as zero, the function calculates the size of an output buffer large enough to contain the transformed text, and the result is returned in this field. The content of the buffers specified by *InpBuf* and *OutBuf*, and the value of *InpBufIndex*, remain unchanged. If *OutSize* = NULL, the `EINVAL` error condition should be returned.

If the *InpToOut* argument is not a null pointer, it points to an array of values with the same number of bytes in *InpBuf* starting with the one pointed by *InpBufIndex* and up to the end of the substring in the buffer. On output, the nth value in *InpToOut* corresponds to the nth byte in *InpBuf*. This value is the index (in units of bytes) in *OutBuf*that identifies the transformed `ShapeCharset` element of the nth byte in *InpBuf*. In the case of multibyte encoding, the index points (for each of the bytes of a code element in the *InpBuf*) to the first byte of the transformed code element in the *OutBuf*.

InpToOut may be specified as `NULL` if no index array from *InpBuf* to *OutBuf* is desired.

If the *OutToInp* argument is not a null pointer, it points to an array of values with the same number of bytes as contained in *OutBuf*. On output, the nth value in *OutToInp* corresponds to the nth byte in *OutBuf* This value is the index in *InpBuf*, starting with the byte pointed to by *InpBufIndex*, that identifies the logical code element of the nth byte in *OutBuf*. In the case of multibyte encoding, the index will point for each of the bytes of a transformed code element in the *OutBuf* to the first byte of the code element in the *InpBuf*.

OutToInp may be specified as `NULL` if no index array from *OutBuf* to *InpBuf* is desired.

To perform shaping of a text string without reordering of code elements, the *layout_object* should be set with input and output layout value *TypeOfText* set to `TEXT_VISUAL` and both in and out of `Orientation` set to the same value.

Return Values If successful, the `m_transform_layout()` function returns `0`. If unsuccessful, the returned value is `−1` and the `errno` is set to indicate the source of error. When the size of *OutBuf* is not large enough to contain the entire transformed text, the input text state at the end of the uncompleted transformation is saved internally and the error condition `E2BIG` is returned in `errno`.

Errors The `m_transform_layout()` function may fail if:

`E2BIG` The output buffer is full and the source text is not entirely processed.

`EBADF` The layout values are set to a meaningless combination or the layout object is not valid.

`EILSEQ` Transformation stopped due to an input code element that cannot be shaped or is invalid. The *InpBufIndex* argument is set to indicate the code element causing the error. The suspect code element is either a valid code element but cannot be shaped into the `ShapeCharset` layout value, or is an invalid code element not defined by the codeset of the locale of *layout_object*. The `mbtowc()` and `wctomb()` functions, when used in the same locale as the `LayoutObject`, can be used to determine if the code element is valid.

`EINVAL` Transformation stopped due to an incomplete composite sequence at the end of the input buffer, or *OutSize* contains `NULL`.

`ERANGE` More than 15 embedding levels are in source text or *InpBuf* contain unbalanced directional layout information (push/pop) or an incomplete composite sequence has been detected in the input buffer at the beginning of the string pointed to by *InpBufIndex*.

An incomplete composite sequence at the end of the input buffer is not always detectable. Sometimes, the fact that the sequence is incomplete will only be detected when additional character elements belonging to the composite sequence are found at the beginning of the next input buffer.

Usage A `LayoutObject` will have a meaningful combination of default layout values. Whoever chooses to change the default layout values is responsible for making sure that the combination of layout values is meaningful. Otherwise, the result of `m_transform_layout()` might be unpredictable or implementation-specific with `errno` set to `EBADF`.

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

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

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

Name m_wtransform_layout – layout transformation for wide character strings

Synopsis

```
cc [ flag... ] file... –llayout [ library... ]
#include <sys/layout.h>

int m_wtransform_layout(LayoutObject layout_object,
     const wchar_t *InpBuf, const size_t ImpSize, const void *OutBuf,
     size_t *Outsize, size_t *InpToOut, size_t *OutToInp,
     unsignedchar *Property, size_t *InpBufIndex);
```

Description The `m_wtransform_layout()` function performs layout transformations (reordering, shaping, cell determination) or provides additional information needed for layout transformation (such as the expected size of the transformed layout, the nesting level of different segments in the text and cross-references between the locations of the corresponding elements before and after the layout transformation). Both the input text and output text are wide character strings.

The `m_wtransform_layout()` function transforms the input text in *InpBuf* according to the current layout values in *layout_object*. Any layout value whose value type is `LayoutTextDescriptor` describes the attributes of the *InpBuf* and *OutBuf* arguments. If the attributes are the same for both *InpBuf* and *OutBuf*, a null transformation is performed with respect to that specific layout value.

The *InpBuf* argument specifies the source text to be processed. The *InpBuf* may not be `NULL`, unless there is a need to reset the internal state.

The *InpSize* argument is the number of bytes within *InpBuf* to be processed by the transformation. Its value will not change after return from the transformation. *InpSize* set to –1 indicates that the text in *InpBuf* is delimited by a null code element. If *InpSize* is not set to –1, it is possible to have some null elements in the input buffer. This might be used, for example, for a "one shot" transformation of several strings, separated by nulls.

Output of this function may be one or more of the following depending on the setting of the arguments:

OutBuf Any transformed data is stored in *OutBuf*, converted to `ShapeCharset`.

Outsize The number of wide characters in *OutBuf*.

InpToOut A cross-reference from each *InpBuf* code element to the transformed data. The cross-reference relates to the data in *InpBuf* starting with the first element that *InpBufIndex* points to (and not necessarily starting from the beginning of the *InpBuf*).

OutToInp A cross-reference to each *InpBuf* code element from the transformed data. The cross-reference relates to the data in *InpBuf* starting with the first element that *InpBufIndex* points to (and not necessarily starting from the beginning of the *InpBuf*).

Property A weighted value that represents peculiar input string transformation properties with different connotations as explained below. If this argument is not a nullpointer, it represents an array of values with the same number of elements as the source substring text before the transformation. Each byte will contain relevant "property" information of the corresponding element in *InpBuf* starting from the element pointed by *InpBufIndex*. The four rightmost bits of each "property" byte will contain information for bidirectional environments (when `ActiveDirectional` is True) and they will mean "`NestingLevels`." The possible value from 0 to 15 represents the nesting level of the corresponding element in the *InpBuf* starting from the element pointed by *InpBufIndex*. If `ActiveDirectional` is false the content of `NestingLevel` bits will be ignored. The leftmost bit of each "property" byte will contain a "new cell indicator" for composed character environments, and will have a value of either 1 (for an element in *InpBuf* that is transformed to the beginning of a new cell) or 0 (for the "zero-length" composing character elements, when these are grouped into the same presentation cell with a non- composing character). Here again, each element of "property" pertains to the elements in the *InpBuf* starting from the element pointed by *InpBufIndex*. (Remember that this is not necessarily the beginning of *InpBuf*). If none of the transformation properties is required, the argument *Property* can be `NULL`. The use of "property" can be enhanced in the future to pertain to other possible usage in other environments.

The *InpBufIndex* argument is an offset value to the location of the transformed text. When `m_wtransform_layout()` is called, *InpBufIndex* contains the offset to the element in *InpBuf* that will be transformed first. (Note that this is not necessarily the first element in *InpBuf*). At the return from the transformation, *InpBufIndex* contains the offset to the first element in the *InpBuf* that has not been transformed. If the entire substring has been transformed successfully, *InpBufIndex* will be incremented by the amount defined by *InpSize*.

Each of these output arguments may be null to specify that no output is desired for the specific argument, but at least one of them should be set to a non-null value to perform any significant work.

In addition to the possible outputs above, *layout_object* maintains a directional state across calls to the transform functions. The directional state is reset to its initial state whenever any of the layout values `TypeOfText`, `Orientation`, or `ImplicitAlg` is modified by means of a call to `m_setvalues_layout()`.

The *layout_object* argument specifies a `LayoutObject` returned by the `m_create_layout()` function.

The *OutBuf*argument contains the transformed data. This argument can be specified as a null pointer to indicate that no transformed data is required.

The encoding of the *OutBuf* argument depends on the `ShapeCharset` layout value defined in *layout_object*. If the `ActiveShapeEditing` layout value is not set (False), the encoding of *OutBuf* is guaranteed to be the same as the codeset of the locale associated with the `LayoutObject` defined by *layout_object*.

On input, the *OutSize* argument specifies the size of the output buffer in number of wide characters. The output buffer should be large enough to contain the transformed result; otherwise, only a partial transformation is performed. If the `ActiveShapeEditing` layout value is set (True) the *OutBuf* should be allocated to contain at least the *InpSize* multiplied by `ShapeCharsetSize`.

On return, the *OutSize* argument is modified to the actual number of code elements in *OutBuf*.

When the *OutSize* argument is specified as zero, the function calculates the size of an output buffer large enough to contain the transformed text, and the result is returned in this field. The content of the buffers specified by *InpBuf* and *OutBuf*, and the value of *InpBufIndex*, remain unchanged. If *OutSize* = `NULL`, the `EINVAL` error condition should be returned.

If the *InpToOut* argument is not a null pointer, it points to an array of values with the same number of wide characters in *InpBuf* starting with the one pointed by *InpBufIndex* and up to the end of the substring in the buffer. On output, the nth value in *InpToOut* corresponds to the nth byte in *InpBuf*. This value is the index (in units of wide characters) in *OutBuf*that identifies the transformed `ShapeCharset` element of the nth byte in *InpBuf*.

InpToOut may be specified as `NULL` if no index array from *InpBuf* to *OutBuf* is desired.

If the *OutToInp* argument is not a null pointer, it points to an array of values with the same number of wide characters as contained in *OutBuf*. On output, the nth value in *OutToInp* corresponds to the nth byte in *OutBuf*. This value is the index in *InpBuf*, starting with wide character byte pointed to by *InpBufIndex*, that identifies the logical code element of the nth wide character in *OutBuf*.

OutToInp may be specified as `NULL` if no index array from *OutBuf* to *InpBuf* is desired.

To perform shaping of a text string without reordering of code elements, the *layout_object* should be set with input and output layout value *TypeOfText* set to `TEXT_VISUAL` and both in and out of `Orientation` set to the same value.

Return Values If successful, the `m_wtransform_layout()` function returns `0`. If unsuccessful, the returned value is `−1` and the `errno` is set to indicate the source of error. When the size of *OutBuf* is not

large enough to contain the entire transformed text, the input text state at the end of the uncompleted transformation is saved internally and the error condition `E2BIG` is returned in `errno`.

Errors The `m_wtransform_layout()` function may fail if:

`E2BIG` The output buffer is full and the source text is not entirely processed.

`EBADF` The layout values are set to a meaningless combination or the layout object is not valid.

`EILSEQ` Transformation stopped due to an input code element that cannot be shaped or is invalid. The *InpBufIndex* argument is set to indicate the code element causing the error. The suspect code element is either a valid code element but cannot be shaped into the `ShapeCharset` layout value, or is an invalid code element not defined by the codeset of the locale of *layout_object*. The `mbtowc()` and `wctomb()` functions, when used in the same locale as the `LayoutObject`, can be used to determine if the code element is valid.

`EINVAL` Transformation stopped due to an incomplete composite sequence at the end of the input buffer, or *OutSize* contains `NULL`.

`ERANGE` More than 15 embedding levels are in source text or *InpBuf* contain unbalanced directional layout information (push/pop) or an incomplete composite sequence has been detected in the input buffer at the beginning of the string pointed to by *InpBufIndex*.

An incomplete composite sequence at the end of the input buffer is not always detectable. Sometimes the fact that the sequence is incomplete will only be detected when additional character elements belonging to the composite sequence are found at the beginning of the next input buffer.

Usage A `LayoutObject` will have a meaningful combination of default layout values. Whoever chooses to change the default layout values is responsible for making sure that the combination of layout values is meaningful. Otherwise, the result of `m_wtransform_layout()` might be unpredictable or implementation-specific with `errno` set to `EBADF`.

Examples **EXAMPLE 1** Shaping and reordering input string into output buffer

The following example illustrated what the different arguments of `m_wtransform_layout()` look like when a string in *InpBuf* is shaped and reordered into *OutBuf*. Upper-case letters in the example represent left-to-right letters while lower-case letters represent right-to-left letters. `xyz` represents the shapes of cde.

```
Position:                   0123456789
InpBuf:                     AB cde 12z

Position:                   0123456789
```

EXAMPLE 1 Shaping and reordering input string into output buffer *(Continued)*

```
OutBuf:                     AB 12 zyxZ

Position:                   0123456789
OutToInp:                   0127865439

Position:                   0123456789
Property.NestLevel:         0001111220
Property.CelBdry:           1111111111
```

The values (encoded in bianry) returned in the *Property* argument define the directionality of each code element in the source text as defined by the type of algorithm used within the *layout_object*. Whlie the algorithm may be implementation dependent, the resulting values and levels are defined such as to allow a single method to be used in determining the directionality of the sourece text. The base rules are:

- Odd levels are always RTL.
- Even levels are always LTR.
- The `Orientation` layout value setting determines the initial level (0 or 1) used.

Within a *Property* array each increment in the level indicates the corresponding code elements should be presented in the opposite direction. Callers of this function should realize that the *Property* values for certain code elements is dependent on the context of the given character and the layout values: `Orientation` and `ImplicitAlg`. Callers should not assume that a given code element always has the same *Property* value in all cases.

EXAMPLE 2 Algorithm to handle nesting

The following is an example of a standard presentation algorithm that handles nesting correctly. The goal of this algorithm is ultimately to return to a zero nest level. Note that more efficient algorithms do exist; the following is provided for clarity rather than for efficiency.

1. Search for the highest next level in the string.
2. Reverse all surrounding code elements of the same level. Reduce the nest level of these code elements by 1.
3. Repeat 1 and 2 until all code elements are of level 0.

The following shows the progression of the example from above:

```
Position:           0123456789    0123456789    0123456789
InpBuf:             AB cde 12Z    AB cde 21Z    AB 12 edcZ
Property.NestLevel: 0001111220    0001111110    0000000000
Property.CellBdry:  1111111111    1111111111    1111111111
```

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

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

See Also attributes(5), standards(5)

Name nan, nanf, nanl – return quiet NaN

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double nan(const char *tagp);

float nanf(const char *tagp);

long double nanl(const char *tagp);
```

Description The function call nan("*n*-char-sequence") is equivalent to:

```
strtod("NAN(n-char-sequence)", (char **) NULL);
```

The function call nan("") is equivalent to:

```
strtod("NAN()", (char **) NULL)
```

If *tagp* does not point to an *n*-char sequence or an empty string, the function call is equivalent to:

```
strtod("NAN", (char **) NULL)
```

Function calls to `nanf()` and `nanl()` are equivalent to the corresponding function calls to `strtof()` and `strtold()`. See `strtod(3C)`.

Return Values These functions return a quiet NaN.

Errors No errors are defined.

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

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

See Also `math.h(3HEAD)`, `strtod(3C)`, `attributes(5)`, `standards(5)`

Name nearbyint, nearbyintf, nearbyintl – floating-point rounding functions

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double nearbyint(double x);

float nearbyintf(float x);

long double nearbyintl(long double x);
```

Description These functions round their argument to an integer value in floating-point format, using the current rounding direction and without raising the inexact floating-point exception.

Return Values Upon successful completion, these functions return the rounded integer value.

If *x* is NaN, a NaN is returned.

If *x* is ±0, ±0 is returned.

If *x* is ±Inf, *x* is returned.

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

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

See Also feclearexcept(3M), fetestexcept(3M), math.h(3HEAD), attributes(5), standards(5)

Name newDmiOctetString – create DmiOctetString in dynamic memory

Synopsis

```
cc [ flag ... ] file ... -ldmi -lnsl -lrwtool [ library ... ]
#include <dmi/util.hh>

DmiOctetString_t *newDmiOctetString(DmiOctetString_t *str);
```

Description The `newDmiOctetString()` function creates a `DmiOctetString` in dynamic memory and returns a pointer to the newly created `DmiOctetString`. The function returns `NULL` if no memory is available.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-level	MT-Safe

See Also `libdmi(3LIB)`, `attributes(5)`

Name newDmiString – create DmiString in dynamic memory

Synopsis

```
cc [ flag ... ] file ... -ldmi -lnsl -lrwtool [ library ... ]
#include <dmi/util.hh>

DmiString_t *newDmiString(char *str);
```

Description The `newDmiString()` function creates a `DmiString` in dynamic memory and returns a pointer to the newly created `DmiString`. The function returns `NULL` if no memory is available.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-level	MT-Safe

See Also `freeDmiString(3DMI)`, `libdmi(3LIB)`, `attributes(5)`

Name nextafter, nextafterf, nextafterl, nexttoward, nexttowardf, nexttowardl – next representable double-precision floating-point number

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double nextafter(double x, double y);

float nextafterf(float x, float y);

long double nextafterl(long double x, long double y);

double nexttoward(double x, long double y);

float nexttowardf(float x, long double y);

long double nexttowardl(long double x, long double y);
```

Description The `nextafter()`, `nextafterf()`, and `nextafterl()` functions compute the next representable floating-point value following *x* in the direction of *y*. Thus, if *y* is less than *x*, `nextafter()` returns the largest representable floating-point number less than *x*. The `nextafter()`, `nextafterf()`, and `nextafterl()` functions return *y* if *x* equals *y*.

The `nexttoward()`, `nexttowardf()`, and `nexttowardl()` functions are equivalent to the corresponding `nextafter()` functions, except that the second parameter has type `long double` and the functions return *y* converted to the type of the function if *x* equals *y*.

Return Values Upon successful completion, these functions return the next representable floating-point value following *x* in the direction of *y*.

If *x* == *y*, *y* (of the type *x*) is returned.

If *x* is finite and the correct function value would overflow, a range error occurs and ±`HUGE_VAL`, ±`HUGE_VALF`, and ±`HUGE_VALL` (with the same sign as *x*) is returned as appropriate for the return type of the function.

If *x* or *y* is NaN, a NaN is returned.

If *x* != *y* and the correct function value is subnormal, zero, or underflows, a range error occurs and either the correct function value (if representable) or 0.0 is returned.

Errors These functions will fail if:

Range Error — The correct value overflows.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, the overflow floating-point exception is raised.

The `nextafter()` function sets `errno` to `ERANGE` if the correct value would overflow.

Range Error — The correct value underflows.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, the underflow floating-point exception is raised.

Usage An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. An application should either examine the return value or check the floating point exception flags to detect exceptions.

An application can also set `errno` to 0 before calling `nextafter()`. On return, if `errno` is non-zero, an error has occurred. The `nextafterf()`, `nextafterl()`. `nexttoward()`, `nexttowardf()`, and `nexttowardl()` functions do not set `errno`.

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

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

See Also `feclearexcept(3M)`, `fetestexcept(3M)`, `math.h(3HEAD)`, `attributes(5)`, `standards(5)`

Name nlist – get entries from name list

Synopsis

```
cc [ flag... ] file ... -lelf [ library ... ]
#include <nlist.h>

int nlist(const char *filename, struct nlist *nl);
```

Description `nlist()` examines the name list in the executable file whose name is pointed to by *filename*, and selectively extracts a list of values and puts them in the array of `nlist()` structures pointed to by `nl`. The name list `nl` consists of an array of structures containing names of variables, types, and values. The list is terminated with a null name, that is, a null string is in the name position of the structure. Each variable name is looked up in the name list of the file. If the name is found, the type, value, storage class, and section number of the name are inserted in the other fields. The type field may be set to 0 if the file was not compiled with the `-g` option to cc(1B).

`nlist()` will always return the information for an external symbol of a given name if the name exists in the file. If an external symbol does not exist, and there is more than one symbol with the specified name in the file (such as static symbols defined in separate files), the values returned will be for the last occurrence of that name in the file. If the name is not found, all fields in the structure except `n_name` are set to 0.

This function is useful for examining the system name list kept in the file `/dev/ksyms`. In this way programs can obtain system addresses that are up to date.

Return Values All value entries are set to 0 if the file cannot be read or if it does not contain a valid name list.

`nlist()` returns 0 on success, −1 on error.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Stable
MT-Level	Safe

See Also cc(1B), elf(3ELF), kvm_nlist(3KVM), kvm_open(3KVM), libelf(3LIB), a.out(4), attributes(5), ksyms(7D), mem(7D)

Name NOTE, _NOTE – annotate source code with info for tools

Synopsis

```
#include <note.h>

NOTE(NoteInfo)

#include<sys/note.h>

_NOTE(NoteInfo)
```

Description These macros are used to embed information for tools in program source. A use of one of these macros is called an "annotation". A tool may define a set of such annotations which can then be used to provide the tool with information that would otherwise be unavailable from the source code.

Annotations should, in general, provide documentation useful to the human reader. If information is of no use to a human trying to understand the code but is necessary for proper operation of a tool, use another mechanism for conveying that information to the tool (one which does not involve adding to the source code), so as not to detract from the readability of the source. The following is an example of an annotation which provides information of use to a tool and to the human reader (in this case, which data are protected by a particular lock, an annotation defined by the static lock analysis tool `lock_lint`).

```
NOTE(MUTEX_PROTECTS_DATA(foo_lock, foo_list Foo))
```

Such annotations do not represent executable code; they are neither statements nor declarations. They should not be followed by a semicolon. If a compiler or tool that analyzes C source does not understand this annotation scheme, then the tool will ignore the annotations. (For such tools, `NOTE(`*x*`)` expands to nothing.)

Annotations may only be placed at particular places in the source.

These places are where the following C constructs would be allowed:

- a top-level declaration (that is, a declaration not within a function or other construct)
- a declaration or statement within a block (including the block which defines a function)
- a member of a `struct` or `union`.

Annotations are not allowed in any other place. For example, the following are illegal:

```
x = y + NOTE(...) z ;
typedef NOTE(...) unsigned int uint ;
```

While `NOTE` and `_NOTE` may be used in the places described above, a particular type of annotation may only be allowed in a subset of those places. For example, a particular annotation may not be allowed inside a `struct` or `union` definition.

NOTE vs _NOTE

Ordinarily, NOTE should be used rather than _NOTE, since use of _NOTE technically makes a program non-portable. However, it may be inconvenient to use NOTE for this purpose in existing code if NOTE is already heavily used for another purpose. In this case one should use a different macro and write a header file similar to /usr/include/note.h which maps that macro to _NOTE in the same manner. For example, the following makes FOO such a macro:

```
#ifndef _FOO_H
#define _FOO_H
#define FOO _NOTE
#include <sys/note.h>
#endif
```

Public header files which span projects should use _NOTE rather than NOTE, since NOTE may already be used by a program which needs to include such a header file.

NoteInfo Argument

The actual *NoteInfo* used in an annotation should be specified by a tool that deals with program source (see the documentation for the tool to determine which annotations, if any, it understands).

NoteInfo must have one of the following forms:

NoteName
NoteName(*Args*)

where *NoteName* is simply an identifier which indicates the type of annotation, and *Args* is something defined by the tool that specifies the particular *NoteName*. The general restrictions on *Args* are that it be compatible with an ANSI C tokenizer and that unquoted parentheses be balanced (so that the end of the annotation can be determined without intimate knowledge of any particular annotation).

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also

note(4), attributes(5)

Name nvlist_add_boolean, nvlist_add_boolean_value, nvlist_add_byte, nvlist_add_int8, nvlist_add_uint8, nvlist_add_int16, nvlist_add_uint16, nvlist_add_int32, nvlist_add_uint32, nvlist_add_int64, nvlist_add_uint64, nvlist_add_string, nvlist_add_nvlist, nvlist_add_nvpair, nvlist_add_boolean_array, nvlist_add_byte_array, nvlist_add_int8_array, nvlist_add_uint8_array, nvlist_add_int16_array, nvlist_add_uint16_array, nvlist_add_int32_array, nvlist_add_uint32_array, nvlist_add_int64_array, nvlist_add_uint64_array, nvlist_add_string_array, nvlist_add_nvlist_array – add new name-value pair to nvlist_t

Synopsis

```
cc [ flag... ] file... -lnvpair [ library... ]
#include <libnvpair.h>

int nvlist_add_boolean(nvlist_t *nvl, const char *name);

int nvlist_add_boolean_value(nvlist_t *nvl,
     const char *name, boolean_t val);

int nvlist_add_byte(nvlist_t *nvl, const char *name,
     uchar_t val);

int nvlist_add_int8(nvlist_t *nvl, const char *name,
     int8_t val);

int nvlist_add_uint8(nvlist_t *nvl, const char *name,
     uint8_t val);

int nvlist_add_int16(nvlist_t *nvl, const char *name,
     int16_t val);

int nvlist_add_uint16(nvlist_t *nvl, const char *name,
     uint16_t val);

int nvlist_add_int32(nvlist_t *nvl, const char *name,
     int32_t val);

int nvlist_add_uint32(nvlist_t *nvl, const char *name,
     uint32_t val);

int nvlist_add_int64(nvlist_t *nvl, const char *name,
     int64_t val);

int nvlist_add_uint64(nvlist_t *nvl, const char *name,
     uint64_t val);

int nvlist_add_string(nvlist_t *nvl, const char *name,
     const char *val);

int nvlist_add_nvlist(nvlist_t *nvl, const char *name,
     nvlist_t *val);

int nvlist_add_nvpair(nvlist_t *nvl, nvpair_t *nvp);

int nvlist_add_boolean_array(nvlist_t *nvl, const char *name,
     boolean_t *val, uint_t nelem);
```

```
int nvlist_add_byte_array(nvlist_t *nvl, const char *name,
     uchar_t *val, uint_t nelem);

int nvlist_add_int8_array(nvlist_t *nvl, const char *name,
     int8_t *val, uint_t nelem);

int nvlist_add_uint8_array(nvlist_t *nvl, const char *name,
     uint8_t *val, uint_t nelem);

int nvlist_add_int16_array(nvlist_t *nvl, const char *name,
     int16_t *val, uint_t nelem);

int nvlist_add_uint16_array(nvlist_t *nvl, const char *name,
     uint16_t *val, uint_t nelem);

int nvlist_add_int32_array(nvlist_t *nvl, const char *name,
     int32_t *val, uint_t nelem);

int nvlist_add_uint32_array(nvlist_t *nvl, const char *name,
     uint32_t *val, uint_t nelem);

int nvlist_add_int64_array(nvlist_t *nvl, const char *name,
     int64_t *val, uint_t nelem);

int nvlist_add_uint64_array(nvlist_t *nvl, const char *name,
     uint64_t *val, uint_t nelem);

int nvlist_add_string_array(nvlist_t *nvl, const char *name,
     char *const *val, uint_t nelem);

int nvlist_add_nvlist_array(nvlist_t *nvl, const char *name,
     nvlist_t **val, uint_t nelem);
```

Parameters

nvl	The `nvlist_t` (name-value pair list) to be processed.
nvp	The `nvpair_t` (name-value pair) to be processed.
name	Name of the `nvpair` (name-value pair).
nelem	Number of elements in value (that is, array size).
val	Value or starting address of the array value.

Description

These functions add a new name-value pair to an `nvlist_t`. The uniqueness of `nvpair` name and data types follows the *nvflag* argument specified for `nvlist_alloc()`. See `nvlist_alloc(3NVPAIR)`.

If `NV_UNIQUE_NAME` was specified for *nvflag*, existing `nvpairs` with matching names are removed before the new `nvpair` is added.

If `NV_UNIQUE_NAME_TYPE` was specified for *nvflag*, existing `nvpairs` with matching names and data types are removed before the new `nvpair` is added.

If neither was specified for *nvflag*, the new `nvpair` is unconditionally added at the end of the list. The library preserves the order of the name-value pairs across packing, unpacking, and duplication.

Multiple threads can simultaneously read the same `nvlist_t`, but only one thread can actively change a given `nvlist_t` at a time. The caller is responsible for the synchronization.

The `nvlist_add_boolean()` function is deprecated. The `nvlist_add_boolean_value()` function should be used instead.

Return Values These functions return 0 on success and an error value on failure.

Errors These functions will fail if:

`EINVAL` There is an invalid argument.

`ENOMEM` There is insufficient memory.

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

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

See Also `libnvpair(3LIB)`, `attributes(5)`

Name nvlist_alloc, nvlist_free, nvlist_size, nvlist_pack, nvlist_unpack, nvlist_dup, nvlist_merge, nvlist_xalloc, nvlist_xpack, nvlist_xunpack, nvlist_xdup, nvlist_lookup_nv_alloc, nv_alloc_init, nv_alloc_reset, nv_alloc_fini – manage a name-value pair list

Synopsis

```
cc [ flag... ] file... -lnvpair [ library... ]
#include <libnvpair.h>

int nvlist_alloc(nvlist_t **nvlp, uint_t nvflag, int flag);

int nvlist_xalloc(nvlist_t **nvlp, uint_t nvflag,
     nv_alloc_t * nva);

void nvlist_free(nvlist_t *nvl);

int nvlist_size(nvlist_t *nvl, size_t *size, int encoding);

int nvlist_pack(nvlist_t *nvl, char **bufp, size_t *buflen,
     int encoding, int flag);

int nvlist_xpack(nvlist_t *nvl, char **bufp, size_t *buflen,
     int encoding, nv_alloc_t * nva);

int nvlist_unpack(char *buf, size_t buflen, nvlist_t **nvlp,
     int flag);

int nvlist_xunpack(char *buf, size_t buflen, nvlist_t **nvlp,
     nv_alloc_t * nva);

int nvlist_dup(nvlist_t *nvl, nvlist_t **nvlp, int flag);

int nvlist_xdup(nvlist_t *nvl, nvlist_t **nvlp,
     nv_alloc_t * nva);

int nvlist_merge(nvlist_t *dst, nvlist_t *nvl, int flag);

nv_alloc_t * nvlist_lookup_nv_alloc(nvlist_t *nvl);

int nv_alloc_init(nv_alloc_t *nva, const nv_alloc_ops_t *nvo,
     /* args */ ...);

void nv_alloc_reset(nv_alloc_t *nva);

void nv_alloc_fini(nv_alloc_t *nva);
```

Parameters

nvlp Address of a pointer to `nvlist_t`.

nvflag Specify bit fields defining `nvlist` properties:

`NV_UNIQUE_NAME`	The `nvpair` names are unique.
`NV_UNIQUE_NAME_TYPE`	Name-data type combination is unique.

flag Specify 0. Reserved for future use.

nvl The `nvlist_t` to be processed.

dst The destination `nvlist_t`.

size	Pointer to buffer to contain the encoded size.
bufp	Address of buffer to pack `nvlist` into. Must be 8-byte aligned. If `NULL`, library will allocate memory.
buf	Buffer containing packed `nvlist`.
buflen	Size of buffer *bufp* or *buf* points to.
encoding	Encoding method for packing.
nvo	Pluggable allocator operations pointer (`nv_alloc_ops_t`).
nva	A pointer to an `nv_alloc_t` structure to be used for the specified `nvlist_t`.

Description

List Manipulation

The `nvlist_alloc()` function allocates a new name-value pair list and updates *nvlp* to point to the handle. The *nvflag* argument specifies `nvlist` properties to remain persistent across packing, unpacking, and duplication. If `NV_UNIQUE_NAME` was specified for *nvflag*, existing nvpairs with matching names are removed before the new nvpair is added. If `NV_UNIQUE_NAME_TYPE` was specified for *nvflag*, existing nvpairs with matching names and data types are removed before the new nvpair is added. See `nvlist_add_byte(3NVPAIR)` for more information.

The `nvlist_xalloc()` function is identical to `nvlist_alloc()` except that `nvlist_xalloc()` can use a different allocator, as described in the Pluggable Allocators section.

The `nvlist_free()` function frees a name-value pair list.

The `nvlist_size()` function returns the minimum size of a contiguous buffer large enough to pack *nvl*. The *encoding* parameter specifies the method of encoding when packing *nvl*. Supported encoding methods are:

`NV_ENCODE_NATIVE`	Straight `bcopy()` as described in `bcopy(3C)`.
`NV_ENCODE_XDR`	Use XDR encoding, suitable for sending to another host.

The `nvlist_pack()` function packs *nvl* into contiguous memory starting at **bufp*. The *encoding* parameter specifies the method of encoding (see above).

- If **bufp* is not `NULL`, **bufp* is expected to be a caller-allocated buffer of size **buflen*.
- If **bufp* is `NULL`, the library will allocate memory and update **bufp* to point to the memory and update **buflen* to contain the size of the allocated memory.

The `nvlist_xpack()` function is identical to `nvlist_pack()` except that `nvlist_xpack()` can use a different allocator.

The nvlist_unpack() function takes a buffer with a packed nvlist_t and unpacks it into a searchable nvlist_t. The library allocates memory for nvlist_t. The caller is responsible for freeing the memory by calling nvlist_free().

The nvlist_xunpack() function is identical to nvlist_unpack() except that nvlist_xunpack() can use a different allocator.

The nvlist_dup() function makes a copy of *nvl* and updates *nvlp* to point to the copy.

The nvlist_xdup() function is identical to nvlist_dup() except that nvlist_xdup() can use a different allocator.

The nvlist_merge() function adds copies of all name-value pairs from *nvl* to *dst*. Name-value pairs in *dst* are replaced with name-value pairs from *nvl* that have identical names (if *dst* has the type NV_UNIQUE_NAME) or identical names and types (if *dst* has the type NV_UNIQUE_NAME_TYPE).

The nvlist_lookup_nv_alloc() function retrieves the pointer to the allocator that was used when manipulating a name-value pair list.

Pluggable Allocators

Using Pluggable Allocators

The nv_alloc_init(), nv_alloc_reset() and nv_alloc_fini() functions provide an interface to specify the allocator to be used when manipulating a name-value pair list.

The nv_alloc_init() function determines the allocator properties and puts them into the *nva* argument. The application must specify the *nv_arg* and *nvo* arguments and an optional variable argument list. The optional arguments are passed to the (*nv_ao_init()) function.

The *nva* argument must be passed to nvlist_xalloc(), nvlist_xpack(), nvlist_xunpack() and nvlist_xdup().

The nv_alloc_reset() function is responsible for resetting the allocator properties to the data specified by nv_alloc_init(). When no (*nv_ao_reset()) function is specified, nv_alloc_reset() has no effect.

The nv_alloc_fini() function destroys the allocator properties determined by nv_alloc_init(). When a (*nv_ao_fini()) function is specified, it is called from nv_alloc_fini().

The disposition of the allocated objects and the memory used to store them is left to the allocator implementation.

The nv_alloc_nosleep nv_alloc_t can be used with nvlist_xalloc() to mimic the behavior of nvlist_alloc().

The nvpair allocator framework provides a pointer to the operation structure of a fixed buffer allocator. This allocator, `nv_fixed_ops`, uses a pre-allocated buffer for memory allocations. It is intended primarily for kernel use and is described on `nvlist_alloc(9F)`.

An example program that uses the pluggable allocator functionality is provided on `nvlist_alloc(9F)`.

Creating Pluggable Allocators

Any producer of name-value pairs can specify its own allocator functions. The application must provide the following pluggable allocator operations:

```
int (*nv_ao_init)(nv_alloc_t *nva, va_list nv_valist);
void (*nv_ao_fini)(nv_alloc_t *nva);
void *(*nv_ao_alloc)(nv_alloc_t *nva, size_t sz);
void (*nv_ao_reset)(nv_alloc_t *nva);
void (*nv_ao_free)(nv_alloc_t *nva, void *buf, size_t sz);
```

The *nva* argument of the allocator implementation is always the first argument.

The optional (`*nv_ao_init()`) function is responsible for filling the data specified by `nv_alloc_init()` into the *nva_arg* argument. The (`*nv_ao_init()`) function is only called when `nv_alloc_init()` is executed.

The optional (`*nv_ao_fini()`) function is responsible for the cleanup of the allocator implementation. It is called by `nv_alloc_fini()`.

The required (`*nv_ao_alloc()`) function is used in the nvpair allocation framework for memory allocation. The *sz* argument specifies the size of the requested buffer.

The optional (`*nv_ao_reset()`) function is responsible for resetting the *nva_arg* argument to the data specified by `nv_alloc_init()`.

The required (`*nv_ao_free()`) function is used in the nvpair allocator framework for memory deallocation. The *buf* argument is a pointer to a block previously allocated by the (`*nv_ao_alloc()`) function. The size argument *sz* must exactly match the original allocation.

The disposition of the allocated objects and the memory used to store them is left to the allocator implementation.

Return Values

These functions return 0 on success and an error value on failure.

The `nvlist_lookup_nv_alloc()` function returns a pointer to an allocator.

Errors

These functions will fail if:

`EINVAL` There is an invalid argument.

The `nvlist_alloc()`, `nvlist_dup()`, `nvlist_pack()`, `nvlist_unpack()`, `nvlist_merge()`, `nvlist_xalloc()`, `nvlist_xdup()`, `nvlist_xpack()`, and `nvlist_xunpack()` functions will fail if:

`ENOMEM` There is insufficient memory.

The `nvlist_pack()`, `nvlist_unpack()`, `nvlist_xpack()`, and `nvlist_xunpack()` functions will fail if:

`EFAULT` An encode/decode error occurs.

`ENOTSUP` An encode/decode method is not supported.

Examples

```
/*
 * Program to create an nvlist.
 */
#include <stdio.h>
#include <sys/types.h>
#include <string.h>
#include <libnvpair.h>

/* generate a packed nvlist */
static int
create_packed_nvlist(char **buf, uint_t *buflen, int encode)
{
    uchar_t bytes[] = {0xaa, 0xbb, 0xcc, 0xdd};
    int32_t int32[] = {3, 4, 5};
    char *strs[] = {"child0", "child1", "child2"};
    int err;
    nvlist_t *nvl;

    err = nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0);    /* allocate list */
    if (err) {
        (void) printf("nvlist_alloc() failed\n");
        return (err);
    }

    /* add a value of some types */
    if ((nvlist_add_byte(nvl, "byte", bytes[0]) != 0) ||
        (nvlist_add_int32(nvl, "int32", int32[0]) != 0) ||
        (nvlist_add_int32_array(nvl, "int32_array", int32, 3) != 0) ||
        (nvlist_add_string_array(nvl, "string_array", strs, 3) != 0)) {
        nvlist_free(nvl);
        return (-1);
    }

    err = nvlist_size(nvl, buflen, encode);
    if (err) {
        (void) printf("nvlist_size: %s\n", strerror(err));
```

```
        nvlist_free(nvl);
        return (err);
    }

    /* pack into contig. memory */
    err = nvlist_pack(nvl, buf, buflen, encode, 0);
    if (err)
        (void) printf("nvlist_pack: %s\n", strerror(err));

    /* free the original list */
    nvlist_free(nvl);
    return (err);
}

/* selectively print nvpairs */
static void
nvlist_lookup_and_print(nvlist_t *nvl)
{
    char **str_val;
    int i, int_val;
    uint_t nval;

    if (nvlist_lookup_int32(nvl, "int32", &int_val) == 0)
        (void) printf("int32 = %d\n", int_val);
    if (nvlist_lookup_string_array(nvl, "string_array", &str_val, &nval)
        == 0) {
            (void) printf("string_array =");
            for (i = 0; i < nval; i++)
                    (void) printf(" %s", str_val[i]);
            (void) printf("\n");
    }
}

/*ARGSUSED*/
int
main(int argc, char *argv[])
{
    int err;
    char *buf = NULL;
    size_t buflen;
    nvlist_t *nvl = NULL;

    if (create_packed_nvlist(&buf, &buflen, NV_ENCODE_XDR) != 0) {
        (void) printf("cannot create packed nvlist buffer\n");
        return(-1);
        }
```

```
    /* unpack into an nvlist_t */
    err = nvlist_unpack(buf, buflen, &nvl, 0);
    if (err) {
        (void) printf("nvlist_unpack(): %s\n", strerror(err));
        return(-1);
    }

    /* selectively print out attributes */
    nvlist_lookup_and_print(nvl);
    return(0);
}
```

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

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

See Also libnvpair(3LIB), attributes(5), nvlist_alloc(9F)

Name nvlist_lookup_boolean, nvlist_lookup_boolean_value, nvlist_lookup_byte, nvlist_lookup_int8, nvlist_lookup_uint8, nvlist_lookup_int16, nvlist_lookup_uint16, nvlist_lookup_int32, nvlist_lookup_uint32, nvlist_lookup_int64, nvlist_lookup_uint64, nvlist_lookup_string, nvlist_lookup_nvlist, nvlist_lookup_boolean_array, nvlist_lookup_byte_array, nvlist_lookup_int8_array, nvlist_lookup_uint8_array, nvlist_lookup_int16_array, nvlist_lookup_uint16_array, nvlist_lookup_int32_array, nvlist_lookup_uint32_array, nvlist_lookup_int64_array, nvlist_lookup_uint64_array, nvlist_lookup_nvlist_array, nvlist_lookup_string_array, nvlist_lookup_pairs – match name and type indicated by the interface name and retrieve data value

Synopsis

```
cc [ flag... ] file... -lnvpair [ library... ]
#include <libnvpair.h>

int nvlist_lookup_boolean(nvlist_t *nvl, const char *name);

int nvlist_lookup_boolean_value(nvlist_t *nvl,
     const char *name, boolean_t *val);

int nvlist_lookup_byte(nvlist_t *nvl, const char *name,
     uchar_t *val);

int nvlist_lookup_int8(nvlist_t *nvl, const char *name,
     int8_t *val);

int nvlist_lookup_uint8(nvlist_t *nvl, const char *name,
     uint8_t *val);

int nvlist_lookup_int16(nvlist_t *nvl, const char *name,
     int16_t *val);

int nvlist_lookup_uint16(nvlist_t *nvl, const char *name,
     uint16_t *val);

int nvlist_lookup_int32(nvlist_t *nvl, const char *name,
     int32_t *val);

int nvlist_lookup_uint32(nvlist_t *nvl, const char *name,
     uint32_t *val);

int nvlist_lookup_int64(nvlist_t *nvl, const char *name,
     int64_t *val);

int nvlist_lookup_uint64(nvlist_t *nvl, const char *name,
     uint64_t *val);

int nvlist_lookup_string(nvlist_t *nvl, const char *name,
     char **val);

int nvlist_lookup_nvlist(nvlist_t *nvl, const char *name,
     nvlist_t **val);

int nvlist_lookup_boolean_array(nvlist_t *nvl, const char *name,
     boolean_t **val, uint_t *nelem);
```

```
int nvlist_lookup_byte_array(nvlist_t *nvl, const char *name,
     uchar_t **val, uint_t *nelem);

int nvlist_lookup_int8_array(nvlist_t *nvl, const char *name,
     int8_t **val, uint_t *nelem);

int nvlist_lookup_uint8_array(nvlist_t *nvl, const char *name,
     uint8_t **val, uint_t *nelem);

int nvlist_lookup_int16_array(nvlist_t *nvl, const char *name,
     int16_t **val, uint_t *nelem);

int nvlist_lookup_uint16_array(nvlist_t *nvl, const char *name,
     uint16_t **val, uint_t *nelem);

int nvlist_lookup_int32_array(nvlist_t *nvl, const char *name,
     int32_t **val, uint_t *nelem);

int nvlist_lookup_uint32_array(nvlist_t *nvl, const char *name,
     uint32_t **val, uint_t *nelem);

int nvlist_lookup_int64_array(nvlist_t *nvl, const char *name,
     int64_t **val, uint_t *nelem);

int nvlist_lookup_uint64_array(nvlist_t *nvl, const char *name,
     uint64_t **val, uint_t *nelem);

int nvlist_lookup_string_array(nvlist_t *nvl, const char *name,
     char ***val, uint_t *nelem);

int nvlist_lookup_nvlist_array(nvlist_t *nvl, const char *name,
     nvlist_t ***val, uint_t *nelem);

int nvlist_lookup_pairs(nvlist_t *nvl, int flag...);
```

Parameters

nvl The `nvlist_t` to be processed.

name Name of the name-value pair to search.

nelem Address to store the number of elements in value.

val Address to store the starting address of the value.

flag Specify bit fields defining lookup behavior:

`NV_FLAG_NOENTOK` The retrival function will not fail if no matching name-value pair is found.

Description

These functions find the `nvpair` (name-value pair) that matches the name and type as indicated by the interface name. If one is found, *nelem* and *val* are modified to contain the number of elements in value and the starting address of data, respectively.

These functions work for nvlists (lists of name-value pairs) allocated with `NV_UNIQUE_NAME` or `NV_UNIQUE_NAME_TYPE` specified in `nvlist_alloc()`. (See `nvlist_alloc(3NVPAIR)`.) If this is not the case, the function returns `ENOTSUP` because the list potentially contains multiple `nvpairs` with the same name and type.

Multiple threads can simultaneously read the same `nvlist_t` but only one thread can actively change a given `nvlist_t` at a time. The caller is responsible for the synchronization.

All memory required for storing the array elements, including string value, are managed by the library. References to such data remain valid until `nvlist_free()` is called on *nvl*.

The `nvlist_lookup_pairs()` function retrieves a set of nvpairs. The arguments are a null-terminated list of pairs (data type `DATA_TYPE_BOOLEAN`), triples (non-array data types) or quads (array data types). The interpretation of the arguments depends on the value of *type* (see `nvpair_type(3NVPAIR)`) as follows:

name Name of the name-value pair to search.

type Data type (see `nvpair_type(3NVPAIR)`).

val Address to store the starting address of the value. When using data type `DATA_TYPE_BOOLEAN`, the *val* argument is omitted.

nelem Address to store the number of elements in value. Non-array data types have only one argument and *nelem* is omitted.

The order of the arguments is *name*, *type*, [*val*], [*nelem*].

When using `NV_FLAG_NOENTOK` and no matching name-value pair is found, the memory pointed to by *val* and *nelem* is left untouched.

Return Values These functions return 0 on success and an error value on failure.

Errors These functions will fail if:

`EINVAL` There is an invalid argument.

`ENOENT` No matching name-value pair is found

`ENOTSUP` An encode/decode method is not supported.

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

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

See Also libnvpair(3LIB), nvpair_type(3NVPAIR), attributes(5)

Name nvlist_next_nvpair, nvpair_name, nvpair_type – return data regarding name-value pairs

Synopsis

```
cc [ flag... ] file... -lnvpair [ library... ]
#include <libnvpair.h>

nvpair_t *nvlist_next_nvpair(nvlist_t *nvl, nvpair_t *nvpair);

char *nvpair_name(nvpair_t *nvpair);

data_type_t nvpair_type(nvpair_t *nvpair);
```

Parameters

nvl The `nvlist_t` to be processed.

nvpair Handle to a name-value pair.

Description The `nvlist_next_nvpair()` function returns a handle to the next `nvpair` in the list following `nvpair`. If `nvpair` is `NULL`, the first pair is returned. If `nvpair` is the last pair in the `nvlist`, `NULL` is returned.

The `nvpair_name()` function returns a string containing the name of `nvpair`.

The `nvpair_type()` function retrieves the value of the `nvpair` in the form of enumerated type `data_type_t`. This is used to determine the appropriate `nvpair_*()` function to call for retrieving the value.

Return Values Upon successful completion, `nvpair_name()` returns a string containing the name of the name-value pair.

Upon successful completion, `nvpair_type()` returns an enumerated data type `data_type_t`. Possible values for `data_type_t` are as follows:

- `DATA_TYPE_BOOLEAN`
- `DATA_TYPE_BOOLEAN_VALUE`
- `DATA_TYPE_BYTE`
- `DATA_TYPE_INT8`
- `DATA_TYPE_UINT8`
- `DATA_TYPE_INT16`
- `DATA_TYPE_UINT16`
- `DATA_TYPE_INT32`
- `DATA_TYPE_UINT32`
- `DATA_TYPE_INT64`
- `DATA_TYPE_UINT64`
- `DATA_TYPE_STRING`
- `DATA_TYPE_NVLIST`
- `DATA_TYPE_BOOLEAN_ARRAY`
- `DATA_TYPE_BYTE_ARRAY`
- `DATA_TYPE_INT8_ARRAY`
- `DATA_TYPE_UINT8_ARRAY`

- DATA_TYPE_INT16_ARRAY
- DATA_TYPE_UINT16_ARRAY
- DATA_TYPE_INT32_ARRAY
- DATA_TYPE_UINT32_ARRAY
- DATA_TYPE_INT64_ARRAY
- DATA_TYPE_UINT64_ARRAY
- DATA_TYPE_STRING_ARRAY
- DATA_TYPE_NVLIST_ARRAY

Upon reaching the end of a list, `nvlist_next_pair()` returns `NULL`. Otherwise, the function returns a handle to next `nvpair` in the list.

These and other `libnvpair(3LIB)` functions cannot manipulate nvpairs after they have been removed from or replaced in an nvlist. Replacement can occur during pair additions to nvlists created with `NV_UNIQUE_NAME_TYPE` and `NV_UNIQUE_NAME`. See `nvlist_alloc(3NVPAIR)`.

Errors No errors are defined.

Examples **EXAMPLE 1** Example of usage of `nvlist_next_nvpair()`.

```
/*
 * usage of nvlist_next_nvpair()
 */
 static int
 edit_nvl(nvlist_t *nvl)
 {
     nvpair_t *curr = nvlist_next_nvpair(nvl, NULL);

     while (curr != NULL) {
         int err;
        nvpair_t *next = nvlist_next_nvpair(nvl, curr);

        if (!nvl_check(curr))
            if ((err = nvlist_remove(nvl, nvpair_name(curr),
                nvpair_type(curr))) != 0)
                    return (err);

         curr = next;
     }
    return (0);
 }
```

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

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

See Also `libnvpair(3LIB)`, `nvlist_alloc(3NVPAIR)`, `attributes(5)`

Notes The enumerated nvpair data types might not be an exhaustive list and new data types can be added. An application using the data type enumeration, `data_type_t`, should be written to expect or ignore new data types.

Name nvlist_remove, nvlist_remove_all – remove name-value pairs

Synopsis

```
cc [ flag... ] file... -lnvpair [ library... ]
#include <libnvpair.h>

int nvlist_remove(nvlist_t *nvl, const char *name,
     data_type_t type);

int nvlist_remove_all(nvlist_t *nvl, const char *name);
```

Parameters

nvl The `nvlist_t` to be processed.

name Name of the name-value pair to be removed.

type Data type of the `nvpair` to be removed.

Description The `nvlist_remove()` function removes the first occurrence of `nvpair` that matches the name and the type.

The `nvlist_remove_all()` function removes all occurrences of `nvpair` that match the name, regardless of type.

Multiple threads can simultaneously read the same `nvlist_t` but only one thread can actively change a given `nvlist_t` at a time. The caller is responsible for the synchronization.

Return Values These functions return 0 on success and an error value on failure.

Errors These functions will fail if:

`EINVAL` There is an invalid argument.

`ENOENT` No name-value pairs were found to match the criteria specified by *name* and *type*.

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

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

See Also `libnvpair(3LIB)`, `attributes(5)`

Name nvpair_value_byte, nvpair_value_boolean_value, nvpair_value_int8, nvpair_value_uint8, nvpair_value_int16, nvpair_value_uint16, nvpair_value_int32, nvpair_value_uint32, nvpair_value_int64, nvpair_value_uint64, nvpair_value_string, nvpair_value_nvlist, nvpair_value_boolean_array, nvpair_value_byte_array, nvpair_value_int8_array, nvpair_value_uint8_array, nvpair_value_int16_array, nvpair_value_uint16_array, nvpair_value_int32_array, nvpair_value_uint32_array, nvpair_value_int64_array, nvpair_value_uint64_array, nvpair_value_string_array, nvpair_value_nvlist_array – retrieve value from a name-value pair

Synopsis

```
cc [ flag... ] file... -lnvpair [ library... ]
#include <libnvpair.h>

int nvpair_value_byte(nvpair_t *nvpair, uchar_t *val);

int nvpair_value_boolean_value(nvpair_t *nvpair,
     boolean_t *val);

int nvpair_value_int8(nvpair_t *nvpair, int8_t *val);

int nvpair_value_uint8(nvpair_t *nvpair, uint8_t *val);

int nvpair_value_int16(nvpair_t *nvpair, int16_t *val);

int nvpair_value_uint16(nvpair_t *nvpair, uint16_t *val);

int nvpair_value_int32(nvpair_t *nvpair, int32_t *val);

int nvpair_value_uint32(nvpair_t *nvpair, uint32_t *val);

int nvpair_value_int64(nvpair_t *nvpair, int64_t *val);

int nvpair_value_uint64(nvpair_t *nvpair, uint64_t *val);

int nvpair_value_string(nvpair_t *nvpair, char **val);

int nvpair_value_nvlist(nvpair_t *nvpair, nvlist_t **val);

int nvpair_value_boolean_array(nvpair_t *nvpair,
     boolean_t **val, uint_t *nelem);

int nvpair_value_byte_array(nvpair_t *nvpair, uchar_t **val,
     uint_t *nelem);

int nvpair_value_int8_array(nvpair_t *nvpair, int8_t **val,
     uint_t *nelem);

int nvpair_value_uint8_array(nvpair_t *nvpair, uint8_t **val,
     uint_t *nelem);

int nvpair_value_int16_array(nvpair_t *nvpair, int16_t **val,
     uint_t *nelem);

int nvpair_value_uint16_array(nvpair_t *nvpair,
     uint16_t **val, uint_t *nelem);

int nvpair_value_int32_array(nvpair_t *nvpair,
     int32_t **val, uint_t *nelem);
```

```
int nvpair_value_uint32_array(nvpair_t *nvpair,
     uint32_t **val, uint_t *nelem);

int nvpair_value_int64_array(nvpair_t *nvpair,
     int64_t **val, uint_t *nelem);

int nvpair_value_uint64_array(nvpair_t *nvpair,
     uint64_t **val, uint_t *nelem);

int nvpair_value_string_array(nvpair_t *nvpair,
     char ***val, uint_t *nelem);

int nvpair_value_nvlist_array(nvpair_t *nvpair,
     nvlist_t ***val, uint_t *nelem);
```

Parameters

nvpair Name-value pair to be processed.

nelem Address to store the number of elements in value.

val Address to store the value or the starting address of the array value.

Description

These functions retrieve the value of *nvpair*. The data type of *nvpair* must match the interface name for the call to be successful.

There is no `nvpair_value_boolean()`; the existence of the name implies the value is true.

For array data types, including string, the memory containing the data is managed by the library and references to the value remains valid until `nvlist_free()` is called on the `nvlist_t` from which *nvpair* is obtained. See `nvlist_free(3NVPAIR)`.

The value of an nvpair may not be retrieved after the nvpair has been removed from or replaced in an nvlist. Replacement can occur during pair additions to nvlists created with `NV_UNIQUE_NAME_TYPE` and `NV_UNIQUE_NAME`. See `nvlist_alloc(3NVPAIR)`.

Return Values

These functions return 0 on success and an error value on failure.

Errors

These functions will fail if:

`EINVAL` Either one of the arguments is `NULL` or the type of *nvpair* does not match the function name.

Attributes

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

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

See Also libnvpair(3LIB), nvlist_alloc(3NVPAIR), attributes(5)

Name p2open, p2close – open, close pipes to and from a command

Synopsis

```
cc [ flag ... ] file ... -lgen [ library ... ]
#include <libgen.h>

int p2open(const char *cmd, FILE *fp[2]);

int p2close(FILE *fp[2]);
```

Description The `p2open()`gfunction forks and execs a shell running the command line pointed to by *cmd*. On return, `fp[0]` points to a `FILE` pointer to write the command's standard input and `fp[1]` points to a `FILE` pointer to read from the command's standard output. In this way the program has control over the input and output of the command.

The function returns `0` if successful; otherwise, it returns `−1`.

The `p2close()` function is used to close the file pointers that `p2open()` opened. It waits for the process to terminate and returns the process status. It returns `0` if successful; otherwise, it returns `−1`.

Return Values A common problem is having too few file descriptors. The `p2close()` function returns `−1` if the two file pointers are not from the same `p2open()`.

Examples **EXAMPLE 1** Example of file descriptors.

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

main(argc,argv)
int argc;
char **argv;
{
    FILE *fp[2];
    pid_t pid;
    char buf[16];

        pid=p2open("/usr/bin/cat", fp);
        if ( pid == −1 ) {
        fprintf(stderr, "p2open failed\n");
        exit(1);
    }
    write(fileno(fp[0]),"This is a test\n", 16);
    if(read(fileno(fp[1]), buf, 16) <=0)
        fprintf(stderr, "p2open failed\n");
    else
        write(1, buf, 16);
    (void)p2close(fp);
}
```

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also fclose(3C), popen(3C), setbuf(3C), attributes(5)

Notes Buffered writes on `fp[0]` can make it appear that the command is not listening. Judiciously placed `fflush()` calls or unbuffering `fp[0]` can be a big help; see fclose(3C).

Many commands use buffered output when connected to a pipe. That, too, can make it appear as if things are not working.

Usage is not the same as for `popen()`, although it is closely related.

Name pam – PAM (Pluggable Authentication Module)

Synopsis

```
#include <security/pam_appl.h>
cc [ flag... ] file ... -lpam [ library ... ]
```

Description The PAM framework, `libpam`, consists of an interface library and multiple authentication service modules. The PAM interface library is the layer implementing the Application Programming Interface (API). The authentication service modules are a set of dynamically loadable objects invoked by the PAM API to provide a particular type of user authentication. PAM gives system administrators the flexibility of choosing any authentication service available on the system to perform authentication. This framework also allows new authentication service modules to be plugged in and made available without modifying the applications.

Refer to *Oracle Solaris Security for Developers Guide* for information about providing authentication, account management, session management, and password management through PAM modules.

Interface Overview The PAM library interface consists of six categories of functions, the names for which all start with the prefix `pam_`.

The first category contains functions for establishing and terminating an authentication activity, which are `pam_start`(3PAM) and `pam_end`(3PAM). The functions `pam_set_data`(3PAM) and `pam_get_data`(3PAM) maintain module specific data. The functions `pam_set_item`(3PAM) and `pam_get_item`(3PAM) maintain state information. `pam_strerror`(3PAM) is the function that returns error status information.

The second category contains the functions that authenticate an individual user and set the credentials of the user, `pam_authenticate`(3PAM) and `pam_setcred`(3PAM).

The third category of PAM interfaces is account management. The function `pam_acct_mgmt`(3PAM) checks for password aging and access-hour restrictions.

Category four contains the functions that perform session management after access to the system has been granted. See `pam_open_session`(3PAM) and `pam_close_session`(3PAM)

The fifth category consists of the function that changes authentication tokens, `pam_chauthtok`(3PAM). An authentication token is the object used to verify the identity of the user. In UNIX, an authentication token is a user's password.

The sixth category of functions can be used to set values for PAM environment variables. See `pam_putenv`(3PAM), `pam_getenv`(3PAM), and `pam_getenvlist`(3PAM).

The `pam_*( )` interfaces are implemented through the library `libpam`. For each of the categories listed above, excluding categories one and six, dynamically loadable shared modules exist that provides the appropriate service layer functionality upon demand. The functional entry points in the service layer start with the `pam_sm_` prefix. The only difference between the `pam_sm_*( )` interfaces and their corresponding `pam_` interfaces is that all the

pam_sm_*( ) interfaces require extra parameters to pass service–specific options to the shared modules. Refer to pam_sm(3PAM) for an overview of the PAM service module APIs.

Stateful Interface

A sequence of calls sharing a common set of state information is referred to as an authentication transaction. An authentication transaction begins with a call to pam_start(). pam_start() allocates space, performs various initialization activities, and assigns a PAM authentication handle to be used for subsequent calls to the library.

After initiating an authentication transaction, applications can invoke pam_authenticate() to authenticate a particular user, and pam_acct_mgmt() to perform system entry management. For example, the application may want to determine if the user's password has expired.

If the user has been successfully authenticated, the application calls pam_setcred() to set any user credentials associated with the authentication service. Within one authentication transaction (between pam_start() and pam_end()), all calls to the PAM interface should be made with the same authentication handle returned by pam_start(). This is necessary because certain service modules may store module-specific data in a handle that is intended for use by other modules. For example, during the call to pam_authenticate(), service modules may store data in the handle that is intended for use by pam_setcred().

To perform session management, applications call pam_open_session(). Specifically, the system may want to store the total time for the session. The function pam_close_session() closes the current session.

When necessary, applications can call pam_get_item() and pam_set_item() to access and to update specific authentication information. Such information may include the current username.

To terminate an authentication transaction, the application simply calls pam_end(), which frees previously allocated space used to store authentication information.

Application–Authentication Service Interactive Interface

The authentication service in PAM does not communicate directly with the user; instead it relies on the application to perform all such interactions. The application passes a pointer to the function, conv(), along with any associated application data pointers, through a *pam_conv* structure to the authentication service when it initiates an authentication transaction, via a call to pam_start(). The service will then use the function, conv(), to prompt the user for data, output error messages, and display text information. Refer to pam_start(3PAM) for more information.

Stacking Multiple Schemes

The PAM architecture enables authentication by multiple authentication services through *stacking*. System entry applications, such as login(1), stack multiple service modules to authenticate users with multiple authentication services. The order in which authentication service modules are stacked is specified in the configuration file, pam.conf(4). A system administrator determines this ordering, and also determines whether the same password can be used for all authentication services.

Administrative Interface

The authentication library, `/usr/lib/libpam.so.1`, implements the framework interface. Various authentication services are implemented by their own loadable modules whose paths are specified through the `pam.conf(4)` file.

Return Values

The PAM functions may return one of the following generic values, or one of the values defined in the specific man pages:

`PAM_SUCCESS`	The function returned successfully.
`PAM_OPEN_ERR`	`dlopen()` failed when dynamically loading a service module.
`PAM_SYMBOL_ERR`	Symbol not found.
`PAM_SERVICE_ERR`	Error in service module.
`PAM_SYSTEM_ERR`	System error.
`PAM_BUF_ERR`	Memory buffer error.
`PAM_CONV_ERR`	Conversation failure.
`PAM_PERM_DENIED`	Permission denied.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	MT-Safe with exceptions

See Also

`login(1)`, `pam_authenticate(3PAM)`, `pam_chauthtok(3PAM)`, `pam_open_session(3PAM)`, `pam_set_item(3PAM)`, `pam_setcred(3PAM)`, `pam_sm(3PAM)`, `pam_start(3PAM)`, `pam_strerror(3PAM)`, `pam.conf(4)`, `attributes(5)`

Oracle Solaris Security for Developers Guide

Notes

The interfaces in `libpam()` are MT-Safe only if each thread within the multithreaded application uses its own PAM handle.

Name pam_acct_mgmt – perform PAM account validation procedures

Synopsis

```
cc [ flag ... ] file ... -lpam [ library ... ]
#include <security/pam_appl.h>

int pam_acct_mgmt(pam_handle_t *pamh, int flags);
```

Description The `pam_acct_mgmt()` function is called to determine if the current user's account is valid. It checks for password and account expiration, and verifies access hour restrictions. This function is typically called after the user has been authenticated with `pam_authenticate(3PAM)`.

The *pamh* argument is an authentication handle obtained by a prior call to `pam_start()`. The following flags may be set in the *flags* field:

`PAM_SILENT`	The account management service should not generate any messages.
`PAM_DISALLOW_NULL_AUTHTOK`	The account management service should return PAM_NEW_AUTHTOK_REQD if the user has a null authentication token.

Return Values Upon successful completion, `PAM_SUCCESS` is returned. In addition to the error return values described in `pam(3PAM)`, the following values may be returned:

`PAM_USER_UNKNOWN`	User not known to underlying account management module.
`PAM_AUTH_ERR`	Authentication failure.
`PAM_NEW_AUTHTOK_REQD`	New authentication token required. This is normally returned if the machine security policies require that the password should be changed because the password is NULL or has aged.
`PAM_ACCT_EXPIRED`	User account has expired.

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

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

See Also `pam(3PAM)`, `pam_authenticate(3PAM)`, `pam_start(3PAM)`, `libpam(3LIB)`, `attributes(5)`

Notes The interfaces in `libpam` are MT-Safe only if each thread within the multithreaded application uses its own PAM handle.

Name pam_authenticate – perform authentication within the PAM framework

Synopsis

```
cc [ flag ... ] file ... -lpam [ library ... ]
#include <security/pam_appl.h>

int pam_authenticate(pam_handle_t *pamh, int flags);
```

Description The `pam_authenticate()` function is called to authenticate the current user. The user is usually required to enter a password or similar authentication token depending upon the authentication service configured within the system. The user in question should have been specified by a prior call to `pam_start()` or `pam_set_item()`.

The following flags may be set in the *flags* field:

`PAM_SILENT`	Authentication service should not generate any messages.
`PAM_DISALLOW_NULL_AUTHTOK`	The authentication service should return `PAM_AUTH_ERR` if the user has a null authentication token.

Return Values Upon successful completion, `PAM_SUCCESS` is returned. In addition to the error return values described in pam(3PAM), the following values may be returned:

`PAM_AUTH_ERR`	Authentication failure.
`PAM_CRED_INSUFFICIENT`	Cannot access authentication data due to insufficient credentials.
`PAM_AUTHINFO_UNAVAIL`	Underlying authentication service cannot retrieve authentication information.
`PAM_USER_UNKNOWN`	User not known to the underlying authentication module.
`PAM_MAXTRIES`	An authentication service has maintained a retry count which has been reached. No further retries should be attempted.

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

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

See Also pam(3PAM), pam_open_session(3PAM), pam_set_item(3PAM), pam_setcred(3PAM), pam_start(3PAM), libpam(3LIB), attributes(5)

Notes In the case of authentication failures due to an incorrect username or password, it is the responsibility of the application to retry `pam_authenticate()` and to maintain the retry count. An authentication service module may implement an internal retry count and return an error `PAM_MAXTRIES` if the module does not want the application to retry.

If the PAM framework cannot load the authentication module, then it will return `PAM_ABORT`. This indicates a serious failure, and the application should not attempt to retry the authentication.

For security reasons, the location of authentication failures is hidden from the user. Thus, if several authentication services are stacked and a single service fails, `pam_authenticate()` requires that the user re-authenticate each of the services.

A null authentication token in the authentication database will result in successful authentication unless `PAM_DISALLOW_NULL_AUTHTOK` was specified. In such cases, there will be no prompt to the user to enter an authentication token.

The interfaces in `libpam` are MT-Safe only if each thread within the multithreaded application uses its own PAM handle.

Name pam_chauthtok – perform password related functions within the PAM framework

Synopsis

```
cc [ flag ... ] file ... -lpam [ library ... ]
#include <security/pam_appl.h>

int pam_chauthtok(pam_handle_t *pamh, const int flags);
```

Description The `pam_chauthtok()` function is called to change the authentication token associated with a particular user referenced by the authentication handle *pamh*.

The following flag may be passed in to `pam_chauthtok()`:

`PAM_SILENT`	The password service should not generate any messages.
`PAM_CHANGE_EXPIRED_AUTHTOK`	The password service should only update those passwords that have aged. If this flag is not passed, all password services should update their passwords.

Upon successful completion of the call, the authentication token of the user will be changed in accordance with the password service configured in the system through `pam.conf(4)`.

Return Values Upon successful completion, `PAM_SUCCESS` is returned. In addition to the error return values described in `pam(3PAM)`, the following values may be returned:

`PAM_PERM_DENIED`	No permission.
`PAM_AUTHTOK_ERR`	Authentication token manipulation error.
`PAM_AUTHTOK_RECOVERY_ERR`	Authentication information cannot be recovered.
`PAM_AUTHTOK_LOCK_BUSY`	Authentication token lock busy.
`PAM_AUTHTOK_DISABLE_AGING`	Authentication token aging disabled.
`PAM_USER_UNKNOWN`	User unknown to password service.
`PAM_TRY_AGAIN`	Preliminary check by password service failed.

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

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

See Also login(1), passwd(1), pam(3PAM), pam_authenticate(3PAM), pam_start(3PAM), attributes

Notes The flag PAM_CHANGE_EXPIRED_AUTHTOK is typically used by a login application which has determined that the user's password has aged or expired. Before allowing the user to login, the login application may invoke pam_chauthtok() with this flag to allow the user to update the password. Typically, applications such as passwd(1) should not use this flag.

The pam_chauthtok() functions performs a preliminary check before attempting to update passwords. This check is performed for each password module in the stack as listed in pam.conf(4). The check may include pinging remote name services to determine if they are available. If pam_chauthtok() returns PAM_TRY_AGAIN, then the check has failed, and passwords are not updated.

The interfaces in libpam are MT-Safe only if each thread within the multithreaded application uses its own PAM handle.

Name pam_getenv – returns the value for a PAM environment name

Synopsis

```
cc [ flag ... ] file ... -lpam [ library ... ]
#include <security/pam_appl.h>

char *pam_getenv(pam_handle_t *pamh, const char *name);
```

Description The `pam_getenv()` function searches the PAM handle *pamh* for a value associated with *name*. If a value is present, `pam_getenv()` makes a copy of the value and returns a pointer to the copy back to the calling application. If no such entry exists, `pam_getenv()` returns `NULL`. It is the responsibility of the calling application to free the memory returned by `pam_getenv()`.

Return Values If successful, `pam_getenv()` returns a copy of the *value* associated with *name* in the PAM handle; otherwise, it returns a `NULL` pointer.

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

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

See Also `pam(3PAM)`, `pam_getenvlist(3PAM)`, `pam_putenv(3PAM)`, `libpam(3LIB)`, `attributes(5)`

Notes The interfaces in `libpam` are MT-Safe only if each thread within the multithreaded application uses its own PAM handle.

Name pam_getenvlist – returns a list of all the PAM environment variables

Synopsis

```
cc [ flag ... ] file ... -lpam [ library ... ]
#include <security/pam_appl.h>

char **pam_getenvlist(pam_handle_t *pamh);
```

Description The `pam_getenvlist()` function returns a list of all the PAM environment variables stored in the PAM handle *pamh*. The list is returned as a null-terminated array of pointers to strings. Each string contains a single PAM environment variable of the form *name*=*value*. The list returned is a duplicate copy of all the environment variables stored in *pamh*. It is the responsibility of the calling application to free the memory returned by `pam_getenvlist()`.

Return Values If successful, `pam_getenvlist()` returns in a null-terminated array a copy of all the PAM environment variables stored in *pamh*. Otherwise, `pam_getenvlist()` returns a null pointer.

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

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

See Also `pam(3PAM)`, `pam_getenv(3PAM)`, `pam_putenv(3PAM)`, `libpam(3LIB)`, `attributes(5)`

Notes The interfaces in `libpam` are MT-Safe only if each thread within the multithreaded application uses its own PAM handle.

Name pam_get_user – PAM routine to retrieve user name

Synopsis

```
cc [ flag ... ] file ...  -lpam [ library ... ]
#include <security/pam_appl.h>


int pam_get_user(pam_handle_t *pamh, char **user,
     const char *prompt);
```

Description The `pam_get_user()` function is used by PAM service modules to retrieve the current user name from the PAM handle. If the user name has not been set with `pam_start()` or `pam_set_item()`, the PAM conversation function will be used to prompt the user for the user name with the string "prompt". If *prompt* is `NULL`, then `pam_get_item()` is called and the value of `PAM_USER_PROMPT` is used for prompting. If the value of `PAM_USER_PROMPT` is `NULL`, the following default prompt is used:

```
Please enter user name:
```

After the user name is gathered by the conversation function, `pam_set_item()` is called to set the value of `PAM_USER`. By convention, applications that need to prompt for a user name should call `pam_set_item()` and set the value of `PAM_USER_PROMPT` before calling `pam_authenticate()`. The service module's `pam_sm_authenticate()` function will then call `pam_get_user()` to prompt for the user name.

Note that certain PAM service modules, such as a smart card module, may override the value of `PAM_USER_PROMPT` and pass in their own prompt. Applications that call `pam_authenticate()` multiple times should set the value of `PAM_USER` to `NULL` with `pam_set_item()` before calling `pam_authenticate()`, if they want the user to be prompted for a new user name each time. The value of *user* retrieved by `pam_get_user()` should not be modified or freed. The item will be released by `pam_end()`.

Return Values Upon success, `pam_get_user()` returns `PAM_SUCCESS`; otherwise it returns an error code. Refer to pam(3PAM) for information on error related return values.

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

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

See Also pam(3PAM), pam_authenticate(3PAM), pam_end(3PAM), pam_get_item(3PAM), pam_set_item(3PAM), pam_sm(3PAM), pam_sm_authenticate(3PAM), pam_start(3PAM), attributes(5)

Notes The interfaces in `libpam` are MT-Safe only if each thread within the multithreaded application uses its own PAM handle.

Name pam_open_session, pam_close_session – perform PAM session creation and termination operations

Synopsis

```
cc [ flag ... ] file ... -lpam [ library ... ]
#include <security/pam_appl.h>

int pam_open_session(pam_handle_t *pamh, int flags);

int pam_close_session(pam_handle_t *pamh, int flags);
```

Description The `pam_open_session()` function is called after a user has been successfully authenticated. See `pam_authenticate(3PAM)` and `pam_acct_mgmt(3PAM)`. It is used to notify the session modules that a new session has been initiated. All programs that use the `pam(3PAM)` library should invoke `pam_open_session()` when beginning a new session. Upon termination of this activity, `pam_close_session()` should be invoked to inform `pam(3PAM)` that the session has terminated.

The *pamh* argument is an authentication handle obtained by a prior call to `pam_start()`. The following flag may be set in the *flags* field for `pam_open_session()` and `pam_close_session()`:

`PAM_SILENT` The session service should not generate any messages.

Return Values Upon successful completion, `PAM_SUCCESS` is returned. In addition to the return values defined in `pam(3PAM)`, the following value may be returned on error:

`PAM_SESSION_ERR` Cannot make or remove an entry for the specified session.

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

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

See Also `getutxent(3C)`, `pam(3PAM)`, `pam_acct_mgmt(3PAM)`, `pam_authenticate(3PAM)`, `pam_start(3PAM)`, `attributes(5)`

Notes In many instances, the `pam_open_session()` and `pam_close_session()` calls may be made by different processes. For example, in UNIX the `login` process opens a session, while the `init` process closes the session. In this case, UTMP/WTMP entries may be used to link the call to `pam_close_session()` with an earlier call to `pam_open_session()`. This is possible because UTMP/WTMP entries are uniquely identified by a combination of attributes, including the user login name and device name, which are accessible through the PAM handle, *pamh*. The call to `pam_open_session()` should precede UTMP/WTMP entry management, and the call to `pam_close_session()` should follow UTMP/WTMP exit management.

The interfaces in `libpam` are MT-Safe only if each thread within the multithreaded application uses its own PAM handle.

Name pam_putenv – change or add a value to the PAM environment

Synopsis

```
cc [ flag ... ] file ... -lpam [ library ... ]
#include <security/pam_appl.h>

int pam_putenv(pam_handle_t *pamh, const char *name_value);
```

Description The `pam_putenv()` function sets the value of the PAM environment variable *name* equal to *value* either by altering an existing PAM variable or by creating a new one.

The *name_value* argument points to a string of the form *name*=*value*. A call to `pam_putenv()` does not immediately change the environment. All *name_value* pairs are stored in the PAM handle *pamh*. An application such as `login(1)` may make a call to `pam_getenv(3PAM)` or `pam_getenvlist(3PAM)` to retrieve the PAM environment variables saved in the PAM handle and set them in the environment if appropriate. `login` will not set PAM environment values which overwrite the values for `SHELL`, `HOME`, `LOGNAME`, `MAIL`, `CDPATH`, `IFS`, and `PATH`. Nor will `login` set PAM environment values which overwrite any value that begins with `LD_`.

If *name_value* equals `NAME=`, then the value associated with `NAME` in the PAM handle will be set to an empty value. If *name_value* equals `NAME`, then the environment variable `NAME` will be removed from the PAM handle.

Return Values The `pam_putenv()` function may return one of the following values:

`PAM_SUCCESS`	The function returned successfully.
`PAM_OPEN_ERR`	`dlopen()` failed when dynamically loading a service module.
`PAM_SYMBOL_ERR`	Symbol not found.
`PAM_SERVICE_ERR`	Error in service module.
`PAM_SYSTEM_ERR`	System error.
`PAM_BUF_ERR`	Memory buffer error.
`PAM_CONV_ERR`	Conversation failure.
`PAM_PERM_DENIED`	Permission denied.

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

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

See Also dlopen(3C), pam(3PAM), pam_getenv(3PAM), pam_getenvlist(3PAM), libpam(3LIB), attributes(5)

Notes The interfaces in libpam are MT-Safe only if each thread within the multithreaded application uses its own PAM handle.

Name pam_setcred – modify or delete user credentials for an authentication service

Synopsis

```
cc [ flag ... ] file ... -lpam [ library ... ]
#include <security/pam_appl.h>

int pam_setcred(pam_handle_t *pamh, int flags);
```

Description The `pam_setcred()` function is used to establish, modify, or delete user credentials. It is typically called after the user has been authenticated and after a session has been validated. See `pam_authenticate(3PAM)` and `pam_acct_mgmt(3PAM)`.

The user is specified by a prior call to `pam_start()` or `pam_set_item()`, and is referenced by the authentication handle, *pamh*. The following flags may be set in the *flags* field. Note that the first four flags are mutually exclusive:

`PAM_ESTABLISH_CRED`	Set user credentials for an authentication service.
`PAM_DELETE_CRED`	Delete user credentials associated with an authentication service.
`PAM_REINITIALIZE_CRED`	Reinitialize user credentials.
`PAM_REFRESH_CRED`	Extend lifetime of user credentials.
`PAM_SILENT`	Authentication service should not generate any messages.

If no flag is set, `PAM_ESTABLISH_CRED` is used as the default.

Return Values Upon success, `pam_setcred()` returns `PAM_SUCCESS`. In addition to the error return values described in `pam(3PAM)` the following values may be returned upon error:

`PAM_CRED_UNAVAIL`	Underlying authentication service can not retrieve user credentials unavailable.
`PAM_CRED_EXPIRED`	User credentials expired.
`PAM_USER_UNKNOWN`	User unknown to underlying authentication service.
`PAM_CRED_ERR`	Failure setting user credentials.

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

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

See Also pam(3PAM), pam_acct_mgmt(3PAM), pam_authenticate(3PAM), pam_set_item(3PAM), pam_start(3PAM), libpam(3LIB), attributes(5)

Notes The interfaces in `libpam` are MT-Safe only if each thread within the multithreaded application uses its own PAM handle.

Name pam_set_data, pam_get_data – PAM routines to maintain module specific state

Synopsis

```
cc [ flag ... ] file ... -lpam [ library ... ]
#include <security/pam_appl.h>

int pam_set_data(pam_handle_t *pamh,
     const char *module_data_name, void *data,
     void  (*cleanup) (pam_handle_t *pamh, void *data,
     int pam_end_status));

int pam_get_data(const pam_handle_t *pamh,
     const char *module_data_name, const void **data);
```

Description The `pam_set_data()` and `pam_get_data()` functions allow PAM service modules to access and update module specific information as needed. These functions should not be used by applications.

The `pam_set_data()` function stores module specific data within the PAM handle *pamh*. The *module_data_name* argument uniquely identifies the data, and the *data* argument represents the actual data. The *module_data_name* argument should be unique across all services.

The *cleanup* function frees up any memory used by the *data* after it is no longer needed, and is invoked by `pam_end()`. The *cleanup* function takes as its arguments a pointer to the PAM handle, *pamh*, a pointer to the actual data, *data*, and a status code, *pam_end_status*. The status code determines exactly what state information needs to be purged.

If `pam_set_data()` is called and module data already exists from a prior call to `pam_set_data()` under the same *module_data_name*, then the existing *data* is replaced by the new *data*, and the existing *cleanup* function is replaced by the new *cleanup* function.

The `pam_get_data()` function retrieves module-specific data stored in the PAM handle, *pamh*, identified by the unique name, *module_data_name*. The *data* argument is assigned the address of the requested data. The *data* retrieved by `pam_get_data()` should not be modified or freed. The *data* will be released by `pam_end()`.

Return Values In addition to the return values listed in pam(3PAM), the following value may also be returned:

`PAM_NO_MODULE_DATA` No module specific data is present.

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

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

See Also pam(3PAM), pam_end(3PAM), libpam(3LIB), attributes(5)

Notes The interfaces in libpam are MT-Safe only if each thread within the multithreaded application uses its own PAM handle.

Name pam_set_item, pam_get_item – authentication information routines for PAM

Synopsis

```
cc [ flag ... ] file ... -lpam [ library ... ]
#include <security/pam_appl.h>

int pam_set_item(pam_handle_t *pamh, int item_type,
     const void *item);

int pam_get_item(const pam_handle_t *pamh, int item_type,
     void **item);
```

Description The `pam_get_item()` and `pam_set_item()` functions allow applications and PAM service modules to access and to update PAM information as needed. The information is specified by *item_type*, and can be one of the following:

`PAM_AUSER`	The authenticated user name. Applications that are trusted to correctly identify the authenticated user should set this item to the authenticated user name. See NOTES and `pam_unix_cred(5)`.
`PAM_AUTHTOK`	The user authentication token.
`PAM_CONV`	The `pam_conv` structure.
`PAM_OLDAUTHTOK`	The old user authentication token.
`PAM_RESOURCE`	A semicolon-separated list of *key=value* pairs that represent the set of resource controls for application by `pam_setcred(3PAM)` or `pam_open_session(3PAM)`. See the individual service module definitions, such as `pam_unix_cred(5)`, for interpretations of the keys and values.
`PAM_RHOST`	The remote host name.
`PAM_RUSER`	The `rlogin`/`rsh` untrusted remote user name.
`PAM_SERVICE`	The service name.
`PAM_TTY`	The tty name.
`PAM_USER`	The user name.
`PAM_USER_PROMPT`	The default prompt used by `pam_get_user()`.
PAM_REPOSITORY	The repository that contains the authentication token information.

The pam_repository structure is defined as:

```
struct pam_repository {
    char   *type;       /* Repository type, e.g., files, */
                        /* nis, ldap */
    void   *scope;      /* Optional scope information */
    size_t  scope_len;  /* length of scope information */
};
```

The *item_type* PAM_SERVICE can be set only by pam_start() and is read-only to both applications and service modules.

For security reasons, the *item_type* PAM_AUTHTOK and PAM_OLDAUTHTOK are available only to the module providers. The authentication module, account module, and session management module should treat PAM_AUTHTOK as the current authentication token and ignore PAM_OLDAUTHTOK. The password management module should treat PAM_OLDAUTHTOK as the current authentication token and PAM_AUTHTOK as the new authentication token.

The pam_set_item() function is passed the authentication handle, *pamh*, returned by pam_start(), a pointer to the object, *item*, and its type, *item_type*. If successful, pam_set_item() copies the item to an internal storage area allocated by the authentication module and returns PAM_SUCCESS. An item that had been previously set will be overwritten by the new value.

The pam_get_item() function is passed the authentication handle, *pamh*, returned by pam_start(), an *item_type*, and the address of the pointer, *item*, which is assigned the address of the requested object. The object data is valid until modified by a subsequent call to pam_set_item() for the same *item_type*, or unless it is modified by any of the underlying service modules. If the item has not been previously set, pam_get_item() returns a null pointer. An *item* retrieved by pam_get_item() should not be modified or freed. The item will be released by pam_end().

Return Values Upon success, pam_get_item() returns PAM_SUCCESS; otherwise it returns an error code. Refer to pam(3PAM) for information on error related return values.

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

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

The functions in libpam(3LIB) are MT-Safe only if each thread within the multithreaded application uses its own PAM handle.

See Also libpam(3LIB), pam(3PAM), pam_acct_mgmt(3PAM), pam_authenticate(3PAM), pam_chauthtok(3PAM), pam_get_user(3PAM), pam_open_session(3PAM), pam_setcred(3PAM), pam_start(3PAM), attributes(5), pam_unix_cred(5)

Notes If the PAM_REPOSITORY *item_type* is set and a service module does not recognize the type, the service module does not process any information, and returns PAM_IGNORE. If the PAM_REPOSITORY *item_type* is not set, a service module performs its default action.

PAM_AUSER is not intended as a replacement for PAM_USER. It is expected to be used to supplement PAM_USER when there is an authenticated user from a source other than `pam_authenticate(3PAM)`. Such sources could be `sshd` host-based authentication, kerberized `rlogin`, and `su(1M)`.

Name pam_sm – PAM Service Module APIs

Synopsis

```
#include <security/pam_appl.h>
#include <security/pam_modules.h>
cc [ flag ...] file ... -lpam [ library ...]
```

Description PAM gives system administrators the flexibility of choosing any authentication service available on the system to perform authentication. The framework also allows new authentication service modules to be plugged in and made available without modifying the applications.

The PAM framework, `libpam`, consists of an interface library and multiple authentication service modules. The PAM interface library is the layer that implements the Application Programming Interface (API). The authentication service modules are a set of dynamically loadable objects invoked by the PAM API to provide a particular type of user authentication.

This manual page gives an overview of the PAM APIs for the service modules, also called the Service Provider Interface (PAM-SPI).

Interface Overview The PAM service module interface consists of functions which can be grouped into four categories. The names for all the authentication library functions start with `pam_sm`. The only difference between the `pam_*()` interfaces and their corresponding `pam_sm_*()` interfaces is that all the `pam_sm_*()` interfaces require extra parameters to pass service-specific options to the shared modules. They are otherwise identical.

The first category contains functions to authenticate an individual user, `pam_sm_authenticate(3PAM)`, and to set the credentials of the user, `pam_sm_setcred(3PAM)`. These back-end functions implement the functionality of `pam_authenticate(3PAM)` and `pam_setcred(3PAM)` respectively.

The second category contains the function to do account management: `pam_sm_acct_mgmt(3PAM)`. This includes checking for password aging and access-hour restrictions. This back-end function implements the functionality of `pam_acct_mgmt(3PAM)`.

The third category contains the functions `pam_sm_open_session(3PAM)` and `pam_sm_close_session(3PAM)` to perform session management after access to the system has been granted. These back-end functions implement the functionality of `pam_open_session(3PAM)` and `pam_close_session(3PAM)`, respectively.

The fourth category consists a function to change authentication tokens `pam_sm_chauthtok(3PAM)`. This back-end function implements the functionality of `pam_chauthtok(3PAM)`.

Stateful Interface A sequence of calls sharing a common set of state information is referred to as an authentication transaction. An authentication transaction begins with a call to `pam_start()`. `pam_start()` allocates space, performs various initialization activities, and assigns an authentication handle to be used for subsequent calls to the library. Note that the service modules do not get called or initialized when `pam_start()` is called. The modules are loaded and the symbols resolved upon first use of that function.

The PAM handle keeps certain information about the transaction that can be accessed through the `pam_get_item()` API. Though the modules can also use `pam_set_item()` to change any of the item information, it is recommended that nothing be changed except `PAM_AUTHTOK` and `PAM_OLDAUTHTOK`.

If the modules want to store any module specific state information then they can use the `pam_set_data(3PAM)` function to store that information with the PAM handle. The data should be stored with a name which is unique across all modules and module types. For example, `SUNW_PAM_UNIX_AUTH_userid` can be used as a name by the UNIX module to store information about the state of user's authentication. Some modules use this technique to share data across two different module types.

Also, during the call to `pam_authenticate()`, the UNIX module may store the authentication status (success or reason for failure) in the handle, using a unique name such as `SUNW_SECURE_RPC_DATA`. This information is intended for use by `pam_setcred()`.

During the call to `pam_acct_mgmt()`, the account modules may store data in the handle to indicate which passwords have aged. This information is intended for use by `pam_chauthtok()`.

The module can also store a cleanup function associated with the data. The PAM framework calls this cleanup function, when the application calls `pam_end()` to close the transaction.

Interaction with the User

The PAM service modules do not communicate directly with the user; instead they rely on the application to perform all such interactions. The application passes a pointer to the function, `conv()`, along with any associated application data pointers, through the `pam_conv` structure when it initiates an authentication transaction (by means of a call to `pam_start()`. The service module will then use the function, `conv()`, to prompt the user for data, output error messages, and display text information. Refer to `pam_start(3PAM)` for more information. The modules are responsible for the localization of all messages to the user.

Conventions

By convention, applications that need to prompt for a user name should call `pam_set_item()` and set the value of `PAM_USER_PROMPT` before calling `pam_authenticate()`. The service module's `pam_sm_authenticate()` function will then call `pam_get_user()` to prompt for the user name. Note that certain PAM service modules (such as a smart card module) may override the value of `PAM_USER_PROMPT` and pass in their own prompt.

Though the PAM framework enforces no rules about the module's names, location, options and such, there are certain conventions that all module providers are expected to follow.

By convention, the modules should be located in the `/usr/lib/security` directory. Additional modules may be located in `/opt/<pkg>/lib`. Architecture specific libraries (for example, sparcv9 or amd64) are located in their respective subdirectories.

For every such module, there should be a corresponding manual page in section 5 which should describe the *module_type* it supports, the functionality of the module, along with the options it supports. The dependencies should be clearly identified to the system

administrator. For example, it should be made clear whether this module is a stand-alone module or depends upon the presence of some other module. One should also specify whether this module should come before or after some other module in the stack.

By convention, the modules should support the following options:

`debug`	Syslog debugging information at `LOG_DEBUG` level. Be careful as to not log any sensitive information such as passwords.
`nowarn`	Turn off warning messages such as "password is about to expire."

If an unsupported option is passed to the modules, it should syslog the error at `LOG_ERR` level.

The permission bits on the service module should be set such that it is not writable by either "group" or "other." The service module should also be owned by root. The PAM framework will not load the module if the above permission rules are not followed.

Errors If there are any errors, the modules should log them using `syslog(3C)` at the `LOG_ERR` level.

Return Values The PAM service module functions may return any of the PAM error numbers specified in the specific man pages. It can also return a `PAM_IGNORE` error number to mean that the PAM framework should ignore this module regardless of whether it is required, optional or sufficient. This error number is normally returned when the module does not contribute to the decision being made by the PAM framework.

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

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

See Also `pam(3PAM)`, `pam_authenticate(3PAM)`, `pam_chauthtok(3PAM)`, `pam_get_user(3PAM)`, `pam_open_session(3PAM)`, `pam_setcred(3PAM)`, `pam_set_item(3PAM)`, `pam_sm_authenticate(3PAM)`, `pam_sm_chauthtok(3PAM)`, `pam_sm_open_session(3PAM)`, `pam_sm_setcred(3PAM)`, `pam_start(3PAM)`, `pam_strerror(3PAM)`, `syslog(3C)`, `pam.conf(4)`, `attributes(5)`, `pam_authtok_check(5)`, `pam_authtok_get(5)`, `pam_authtok_store(5)`, `pam_dhkeys(5)`, `pam_passwd_auth(5)`, `pam_unix_account(5)`, `pam_unix_auth(5)`, `pam_unix_session(5)`

Notes The interfaces in `libpam` are MT-Safe only if each thread within the multithreaded application uses its own PAM handle.

The `pam_unix(5)` module is no longer supported. Similar functionality is provided by `pam_authtok_check(5)`, `pam_authtok_get(5)`, `pam_authtok_store(5)`, `pam_dhkeys(5)`, `pam_passwd_auth(5)`, `pam_unix_account(5)`, `pam_unix_auth(5)`, and `pam_unix_session(5)`.

Name pam_sm_acct_mgmt – service provider implementation for pam_acct_mgmt

Synopsis

```
cc [ flag ... ] file ... -lpam [ library ... ]
#include <security/pam_appl.h>
#include <security/pam_modules.h>


int pam_sm_acct_mgmt(pam_handle_t *pamh, int flags, int argc,
     const char **argv);
```

Description In response to a call to pam_acct_mgmt(3PAM), the PAM framework calls pam_sm_acct_mgmt() from the modules listed in the pam.conf(4) file. The account management provider supplies the back-end functionality for this interface function. Applications should not call this API directly.

The pam_sm_acct_mgmt() function determines whether or not the current user's account and password are valid. This includes checking for password and account expiration, and valid login times. The user in question is specified by a prior call to pam_start(), and is referenced by the authentication handle, *pamh*, which is passed as the first argument to pam_sm_acct_mgmt(). The following flags may be set in the *flags* field:

PAM_SILENT	The account management service should not generate any messages.
PAM_DISALLOW_NULL_AUTHTOK	The account management service should return PAM_NEW_AUTHTOK_REQD if the user has a null authentication token.

The *argc* argument represents the number of module options passed in from the configuration file pam.conf(4). *argv* specifies the module options, which are interpreted and processed by the account management service. Please refer to the specific module man pages for the various available *options*. If an unknown option is passed to the module, an error should be logged through syslog(3C) and the option ignored.

If an account management module determines that the user password has aged or expired, it should save this information as state in the authentication handle, *pamh*, using pam_set_data(). pam_chauthok() uses this information to determine which passwords have expired.

Return Values If there are no restrictions to logging in, PAM_SUCCESS is returned. The following error values may also be returned upon error:

PAM_USER_UNKNOWN	User not known to underlying authentication module.
PAM_NEW_AUTHTOK_REQD	New authentication token required.
PAM_ACCT_EXPIRED	User account has expired.
PAM_PERM_DENIED	User denied access to account at this time.

PAM_IGNORE	Ignore underlying account module regardless of whether the control flag is *required, optional* or *sufficient*.

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

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

See Also pam(3PAM), pam_acct_mgmt(3PAM), pam_set_data(3PAM), pam_start(3PAM), syslog(3C), libpam(3LIB), pam.conf(4), attributes(5)

Notes The interfaces in libpam are MT-Safe only if each thread within the multithreaded application uses its own PAM handle.

If the PAM_REPOSITORY *item_type* is set and a service module does not recognize the type, the service module does not process any information, and returns PAM_IGNORE. If the PAM_REPOSITORY *item_type* is not set, a service module performs its default action.

Name pam_sm_authenticate – service provider implementation for pam_authenticate

Synopsis

```
cc [ flag... ] file... -lpam [ library... ]
#include <security/pam_appl.h>
#include <security/pam_modules.h>


int pam_sm_authenticate(pam_handle_t *pamh, int flags,
     int argc, const char **argv);
```

Description In response to a call to pam_authenticate(3PAM), the PAM framework calls pam_sm_authenticate() from the modules listed in the pam.conf(4) file. The authentication provider supplies the back-end functionality for this interface function.

The pam_sm_authenticate() function is called to verify the identity of the current user. The user is usually required to enter a password or similar authentication token depending upon the authentication scheme configured within the system. The user in question is specified by a prior call to pam_start(), and is referenced by the authentication handle *pamh*.

If the user is unknown to the authentication service, the service module should mask this error and continue to prompt the user for a password. It should then return the error, PAM_USER_UNKNOWN.

The following flag may be passed in to pam_sm_authenticate():

PAM_SILENT	The authentication service should not generate any messages.
PAM_DISALLOW_NULL_AUTHTOK	The authentication service should return
PAM_AUTH_ERR	The user has a null authentication token.

The *argc* argument represents the number of module options passed in from the configuration file pam.conf(4). *argv* specifies the module options, which are interpreted and processed by the authentication service. Please refer to the specific module man pages for the various available *options*. If any unknown option is passed in, the module should log the error and ignore the option.

Before returning, pam_sm_authenticate() should call pam_get_item() and retrieve PAM_AUTHTOK. If it has not been set before and the value is NULL, pam_sm_authenticate() should set it to the password entered by the user using pam_set_item().

An authentication module may save the authentication status (success or reason for failure) as state in the authentication handle using pam_set_data(3PAM). This information is intended for use by pam_setcred().

Return Values Upon successful completion, PAM_SUCCESS must be returned. In addition, the following values may be returned:

PAM_MAXTRIES	Maximum number of authentication attempts exceeded.
PAM_AUTH_ERR	Authentication failure.
PAM_CRED_INSUFFICIENT	Cannot access authentication data due to insufficient credentials.
PAM_AUTHINFO_UNAVAIL	Underlying authentication service can not retrieve authentication information.
PAM_USER_UNKNOWN	User not known to underlying authentication module.
PAM_IGNORE	Ignore underlying authentication module regardless of whether the control flag is *required*,*optional,* or *sufficient*1.

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

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

See Also pam(3PAM), pam_authenticate(3PAM), pam_get_item(3PAM), pam_set_data(3PAM), pam_set_item(3PAM), pam_setcred(3PAM), pam_start(3PAM), libpam(3LIB), pam.conf(4), attributes(5)

Notes Modules should not retry the authentication in the event of a failure. Applications handle authentication retries and maintain the retry count. To limit the number of retries, the module can return a PAM_MAXTRIES error.

The interfaces in libpam are MT-Safe only if each thread within the multithreaded application uses its own PAM handle.

If the PAM_REPOSITORY *item_type* is set and a service module does not recognize the type, the service module does not process any information, and returns PAM_IGNORE. If the PAM_REPOSITORY *item_type* is not set, a service module performs its default action.

Name pam_sm_chauthtok – service provider implementation for pam_chauthtok

Synopsis

```
cc [ flag ...] file ... -lpam [ library ... ]
#include <security/pam_appl.h>
#include <security/pam_modules.h>

int pam_sm_chauthtok(pam_handle_t *pamh, int flags, int argc, const char **argv);
```

Description In response to a call to `pam_chauthtok()` the PAM framework calls `pam_sm_chauthtok(3PAM)` from the modules listed in the `pam.conf(4)` file. The password management provider supplies the back-end functionality for this interface function.

The `pam_sm_chauthtok()` function changes the authentication token associated with a particular user referenced by the authentication handle *pamh*.

The following flag may be passed to `pam_chauthtok()`:

`PAM_SILENT`	The password service should not generate any messages.
`PAM_CHANGE_EXPIRED_AUTHTOK`	The password service should only update those passwords that have aged. If this flag is not passed, the password service should update all passwords.
`PAM_PRELIM_CHECK`	The password service should only perform preliminary checks. No passwords should be updated.
`PAM_UPDATE_AUTHTOK`	The password service should update passwords.

Note that `PAM_PRELIM_CHECK` and `PAM_UPDATE_AUTHTOK` cannot be set at the same time.

Upon successful completion of the call, the authentication token of the user will be ready for change or will be changed, depending upon the flag, in accordance with the authentication scheme configured within the system.

The *argc* argument represents the number of module options passed in from the configuration file `pam.conf(4)`. The *argv* argument specifies the module options, which are interpreted and processed by the password management service. Please refer to the specific module man pages for the various available *options*.

It is the responsibility of `pam_sm_chauthtok()` to determine if the new password meets certain strength requirements. `pam_sm_chauthtok()` may continue to re-prompt the user (for a limited number of times) for a new password until the password entered meets the strength requirements.

Before returning, `pam_sm_chauthtok()` should call `pam_get_item()` and retrieve both `PAM_AUTHTOK` and `PAM_OLDAUTHTOK`. If both are `NULL`, `pam_sm_chauthtok()` should set them to the new and old passwords as entered by the user.

Return Values Upon successful completion, `PAM_SUCCESS` must be returned. The following values may also be returned:

`PAM_PERM_DENIED`	No permission.
`PAM_AUTHTOK_ERR`	Authentication token manipulation error.
`PAM_AUTHTOK_RECOVERY_ERR`	Old authentication token cannot be recovered.
`PAM_AUTHTOK_LOCK_BUSY`	Authentication token lock busy.
`PAM_AUTHTOK_DISABLE_AGING`	Authentication token aging disabled.
`PAM_USER_UNKNOWN`	User unknown to password service.
`PAM_TRY_AGAIN`	Preliminary check by password service failed.

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

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

See Also `ping(1M)`, `pam(3PAM)`, `pam_chauthtok(3PAM)`, `pam_get_data(3PAM)`, `pam_get_item(3PAM)`, `pam_set_data(3PAM)`, `libpam(3LIB)`, `pam.conf(4)`, `attributes(5)`

Notes The PAM framework invokes the password services twice. The first time the modules are invoked with the flag, `PAM_PRELIM_CHECK`. During this stage, the password modules should only perform preliminary checks. For example, they may `ping` remote name services to see if they are ready for updates. If a password module detects a transient error such as a remote name service temporarily down, it should return `PAM_TRY_AGAIN` to the PAM framework, which will immediately return the error back to the application. If all password modules pass the preliminary check, the PAM framework invokes the password services again with the flag, `PAM_UPDATE_AUTHTOK`. During this stage, each password module should proceed to update the appropriate password. Any error will again be reported back to application.

If a service module receives the flag `PAM_CHANGE_EXPIRED_AUTHTOK`, it should check whether the password has aged or expired. If the password has aged or expired, then the service module should proceed to update the password. If the status indicates that the password has not yet aged or expired, then the password module should return `PAM_IGNORE`.

If a user's password has aged or expired, a PAM account module could save this information as state in the authentication handle, *pamh*, using `pam_set_data()`. The related password

management module could retrieve this information using `pam_get_data()` to determine whether or not it should prompt the user to update the password for this particular module.

The interfaces in `libpam` are MT-Safe only if each thread within the multithreaded application uses its own PAM handle.

If the `PAM_REPOSITORY` *item_type* is set and a service module does not recognize the type, the service module does not process any information, and returns `PAM_IGNORE`. If the `PAM_REPOSITORY` *item_type* is not set, a service module performs its default action.

Name pam_sm_open_session, pam_sm_close_session – service provider implementation for pam_open_session and pam_close_session

Synopsis

```
cc [ flag ... ]  file ... -lpam [ library ... ]
#include <security/pam_appl.h>
#include <security/pam_modules.h>


int pam_sm_open_session(pam_handle_t *pamh, int flags,
     int argc, const char **argv);

int pam_sm_close_session(pam_handle_t *pamh, int flags,
     int argc, const char **argv);
```

Description In response to a call to pam_open_session(3PAM) and pam_close_session(3PAM), the PAM framework calls pam_sm_open_session() and pam_sm_close_session(), respectively from the modules listed in the pam.conf(4) file. The session management provider supplies the back-end functionality for this interface function.

The pam_sm_open_session() function is called to initiate session management. Thepam_sm_close_session() function is invoked when a session has terminated. The argument *pamh* is an authentication handle. The following flag may be set in the *flags* field:

PAM_SILENT Session service should not generate any messages.

The *argc* argument represents the number of module options passed in from the configuration file pam.conf(4). *argv* specifies the module options, which are interpreted and processed by the session management service. If an unknown option is passed in, an error should be logged through syslog(3C) and the option ignored.

Return Values Upon successful completion, PAM_SUCCESS should be returned. The following values may also be returned upon error:

PAM_SESSION_ERR Cannot make or remove an entry for the specified session.

PAM_IGNORE Ignore underlying session module regardless of whether the control flag is *required*, *optional* or *sufficient*.

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

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

See Also pam(3PAM), pam_open_session(3PAM), syslog(3C), libpam(3LIB), pam.conf(4), attributes(5)

Notes The interfaces in libpam are MT-Safe only if each thread within the multithreaded application uses its own PAM handle.

Name pam_sm_setcred – service provider implementation for pam_setcred

Synopsis

```
cc [ flag ... ] file ... -lpam [ library ... ]
#include <security/pam_appl.h>
#include <security/pam_modules.h>


int pam_sm_setcred(pam_handle_t *pamh, int flags, int argc,
     const char **argv);
```

Description In response to a call to pam_setcred(3PAM), the PAM framework calls pam_sm_setcred() from the modules listed in the pam.conf(4) file. The authentication provider supplies the back-end functionality for this interface function.

The pam_sm_setcred() function is called to set the credentials of the current user associated with the authentication handle, *pamh*. The following flags may be set in the *flags* field. Note that the first four flags are mutually exclusive:

PAM_ESTABLISH_CRED	Set user credentials for the authentication service.
PAM_DELETE_CRED	Delete user credentials associated with the authentication service.
PAM_REINITIALIZE_CRED	Reinitialize user credentials.
PAM_REFRESH_CRED	Extend lifetime of user credentials.
PAM_SILENT	Authentication service should not generate messages

If no flag is set, PAM_ESTABLISH_CRED is used as the default.

The *argc* argument represents the number of module options passed in from the configuration file pam.conf(4). *argv* specifies the module options, which are interpreted and processed by the authentication service. If an unknown option is passed to the module, an error should be logged and the option ignored.

If the PAM_SILENT flag is not set, then pam_sm_setcred() should print any failure status from the corresponding pam_sm_authenticate() function using the conversation function.

The authentication status (success or reason for failure) is saved as module-specific state in the authentication handle by the authentication module. The status should be retrieved using pam_get_data(), and used to determine if user credentials should be set.

Return Values Upon successful completion, PAM_SUCCESS should be returned. The following values may also be returned upon error:

PAM_CRED_UNAVAIL	Underlying authentication service can not retrieve user credentials.
PAM_CRED_EXPIRED	User credentials have expired.

PAM_USER_UNKNOWN	User unknown to the authentication service.
PAM_CRED_ERR	Failure in setting user credentials.
PAM_IGNORE	Ignore underlying authentication module regardless of whether the control flag is *required*,*optional*, or *sufficient*.

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

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

See Also pam(3PAM), pam_authenticate(3PAM), pam_get_data(3PAM) pam_setcred(3PAM), pam_sm_authenticate(3PAM), libpam(3LIB), pam.conf(4), attributes(5)

Notes The pam_sm_setcred() function is passed the same module options that are used by pam_sm_authenticate().

The interfaces in libpam are MT-Safe only if each thread within the multithreaded application uses its own PAM handle.

If the PAM_REPOSITORY *item_type* is set and a service module does not recognize the type, the service module does not process any information, and returns PAM_IGNORE. If the PAM_REPOSITORY *item_type* is not set, a service module performs its default action.

Name pam_start, pam_end – PAM authentication transaction functions

Synopsis

```
cc [ flag ... ] file ... -lpam [ library ... ]
#include <security/pam_appl.h>

int pam_start(const char *service, const char *user,
     const struct pam_conv *pam_conv, pam_handle_t **pamh);

int pam_end(pam_handle_t *pamh, int status);
```

Description The `pam_start()` function is called to initiate an authentication transaction. It takes as arguments the name of the current service, *service*, the name of the user to be authenticated, *user*, the address of the conversation structure, *pam_conv*, and the address of a variable to be assigned the authentication handle *pamh*. Upon successful completion, *pamh* refers to a PAM handle for use with subsequent calls to the authentication library.

The *pam_conv* structure contains the address of the conversation function provided by the application. The underlying PAM service module invokes this function to output information to and retrieve input from the user. The `pam_conv` structure has the following entries:

```
struct pam_conv {
    int   (*conv)();     /* Conversation function */
    void  *appdata_ptr;  /* Application data */
};

int conv(int num_msg, const struct pam_message **msg,
         struct pam_response **resp, void *appdata_ptr);
```

The `conv()` function is called by a service module to hold a PAM conversation with the application or user. For window applications, the application can create a new pop-up window to be used by the interaction.

The *num_msg* parameter is the number of messages associated with the call. The parameter *msg* is a pointer to an array of length *num_msg* of the *pam_message* structure.

The `pam_message` structure is used to pass prompt, error message, or any text information from the authentication service to the application or user. It is the responsibility of the PAM service modules to localize the messages. The memory used by `pam_message` has to be allocated and freed by the PAM modules. The `pam_message` structure has the following entries:

```
struct pam_message{
    int     msg_style;
    char    *msg;
};
```

The message style, `msg_style`, can be set to one of the following values:

`PAM_PROMPT_ECHO_OFF` Prompt user, disabling echoing of response.

`PAM_PROMPT_ECHO_ON` Prompt user, enabling echoing of response.

PAM_ERROR_MSG	Print error message.
PAM_TEXT_INFO	Print general text information.

The maximum size of the message and the response string is PAM_MAX_MSG_SIZE as defined in <security/pam.appl.h>.

The structure *pam_response* is used by the authentication service to get the user's response back from the application or user. The storage used by *pam_response* has to be allocated by the application and freed by the PAM modules. The *pam_response* structure has the following entries:

```
struct pam_response{
    char *resp;
    int  resp_retcode;  /* currently not used, */
                        /* should be set to 0 */
 };
```

It is the responsibility of the conversation function to strip off NEWLINE characters for PAM_PROMPT_ECHO_OFF and PAM_PROMPT_ECHO_ON message styles, and to add NEWLINE characters (if appropriate) for PAM_ERROR_MSG and PAM_TEXT_INFO message styles.

The *appdata_ptr* argument is an application data pointer which is passed by the application to the PAM service modules. Since the PAM modules pass it back through the conversation function, the applications can use this pointer to point to any application-specific data.

The pam_end() function is called to terminate the authentication transaction identified by *pamh* and to free any storage area allocated by the authentication module. The argument, *status*, is passed to the cleanup(|) function stored within the pam handle, and is used to determine what module-specific state must be purged. A cleanup function is attached to the handle by the underlying PAM modules through a call to pam_set_data(3PAM) to free module-specific data.

Refer to *Oracle Solaris Security for Developers Guide* for information about providing authentication, account management, session management, and password management through PAM modules.

Return Values Refer to the RETURN VALUES section on pam(3PAM).

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

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

See Also `libpam`(3LIB), `pam`(3PAM), `pam_acct_mgmt`(3PAM), `pam_authenticate`(3PAM), `pam_chauthtok`(3PAM), `pam_open_session`(3PAM), `pam_setcred`(3PAM), `pam_set_data`(3PAM), `pam_strerror`(3PAM), `attributes`(5)

Oracle Solaris Security for Developers Guide

Notes The interfaces in `libpam` are MT-Safe only if each thread within the multithreaded application uses its own PAM handle.

Name pam_strerror – get PAM error message string

Synopsis

```
cc [ flag... ] file... -lpam [ library... ]
#include <security/pam_appl.h>

const char *pam_strerror(pam_handle_t*pamh, int errnum);
```

Description The `pam_strerror()` function maps the PAM error number in *errnum* to a PAM error message string, and returns a pointer to that string. The application should not free or modify the string returned.

The *pamh* argument is the PAM handle obtained by a prior call to `pam_start()`. If `pam_start()` returns an error, a null PAM handle should be passed.

Errors The `pam_strerror()` function returns the string "Unknown error" if *errnum* is out-of-range.

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

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

See Also `pam(3PAM)`, `pam_start(3PAM)`, `attributes(5)`

Notes The interfaces in `libpam` are MT-Safe only if each thread within the multithreaded application uses its own PAM handle.

Name papiAttributeListAddValue, papiAttributeListAddBoolean, papiAttributeListAddCollection, papiAttributeListAddDatetime, papiAttributeListAddInteger, papiAttributeListAddMetadata, papiAttributeListAddRange, papiAttributeListAddResolution, papiAttributeListAddString, papiAttributeListDelete, papiAttributeListGetValue, papiAttributeListGetNext, papiAttributeListFind, papiAttributeListGetBoolean, papiAttributeListGetCollection, papiAttributeListGetDatetime, papiAttributeListGetInteger, papiAttributeListGetMetadata, papiAttributeListGetRange, papiAttributeListGetResolution, papiAttributeListGetString, papiAttributeListFromString, papiAttributeListToString, papiAttributeListFree – manage PAPI attribute lists

Synopsis

```
cc [ flag... ] file... -lpapi [ library... ]
#include <papi.h>

papi_status_t papiAttributeListAddValue(papi_attribute_t ***attrs,
     int flags, char *name, papi_attribute_value_type_t type,
     papi_attribute_value_t *value);

papi_status_t papiAttributeListAddString(papi_attribute_t ***attrs,
     int flags, char *name, char *string);

papi_status_t papiAttributeListAddInteger(papi_attribute_t ***attrs,
     int flags, char *name, int integer);

papi_status_t papiAttributeListAddBoolean(papi_attribute_t ***attrs,
     int flags, char *name, char boolean);

papi_status_t papiAttributeListAddRange(papi_attribute_t ***attrs,
     int flags, char *name, int lower, int upper);

papi_status_t papiAttributeListAddResolution(papi_attribute_t ***attrs,
     int flags, char *name, int xres, int yres,
     papi_resolution_unit_t units);

papi_status_t papiAttributeListAddDatetime(papi_attribute_t ***attrs,
     int flags, char *name, time_t datetime);

papi_status_t papiAttributeListAddCollection(papi_attribute_t ***attrs,
     int flags, char *name, papi_attribute_t **collection);

papi_status_t papiAttributeListAddMetadata(papi_attribute_t ***attrs,
     int flags, char *name, papi_metadata_t metadata);

papi_status_t papiAttributeListDelete(papi_attribute_t ***attributes,
     char *name);

papi_status_t papiAttributeListGetValue(papi_attribute_t **list,
     void **iterator, char *name, papi_attribute_value_type_t type,
     papi_attribute_value_t **value);

papi_status_t papiAttributeListGetString(papi_attribute_t **list,
     void **iterator, char *name, char **vptr);
```

```
papi_status_t papiAttributeListGetInteger(papi_attribute_t **list,
     void **iterator, char *name, int *vptr);

papi_status_t papiAttributeListGetBoolean(papi_attribute_t **list,
     void **iterator, char *name, char *vptr);

papi_status_t papiAttributeListGetRange(papi_attribute_t **list,
     void **iterator, char *name, int *min, int *max);

papi_status_t papiAttributeListGetResolution(papi_attribute_t **list,
     void **iterator, char *name, int *x, int *y,
     papi_resolution_unit_t *units);

papi_status_t papiAttributeListGetDatetime(papi_attribute_t **list,
     void **iterator, char *name, time_t *dt);

papi_status_t papiAttributeListGetCollection(papi_attribute_t **list,
     void **iterator, char *name, papi_attribute_t ***collection);

papi_status_t papiAttributeListGetMetadata(papi_attribute_t **list,
     void **iterator, char *name, papi_metadata_t *vptr);

papi_attribute_t *papiAttributeListFind(papi_attribute_t **list,
     char *name);

papi_attribute_t *papiAttributeListGetNext(papi_attribute_t **list,
     void **iterator);

void papiAttributeListFree(papi_attribute_t **attributes);

papi_status_t papiAttributeListFromString(papi_attribute_t ***attrs,
     int flags, char *string);

papi_status_t papiAttributeListToString(papi_attribute_t **attrs,
     char *delim, char *buffer, size_t buflen);
```

Parameters

attrs	address of array of pointers to attributes
attributes	a list of attributes (of type `papi_attribute_t **`) contained in a collection. Lists can be hierarchical.
boolean	boolean value (`PAPI_TRUE` or `PAPI_FALSE`)
buffer	buffer to fill
buflen	length of supplied buffer
collection	list of attributes
datetime	attribute time value specified in `time_t` representation
delim	delimiter used in construction of a string representation of an attribute list
dt	date and time represented as a `time_t`
flags	Specify bit fields defining how actions will be performed:

`PAPI_ATTR_REPLACE`
Free any existing value(s) and replace it with the supplied value(s).

`PAPI_ATTR_APPEND`
Add the supplied value to the attribute.

`PAPI_ATTR_EXCL`
Add the supplied value to the attribute, if the attribute was not previously defined.

integer	integer value
iterator	iterator for enumerating multiple values of multi-value attributes
list	array of pointers to attributes (attribute list)
lower	lower bound for a range of integer
max	maximum value in a range
metadata	pseudo-values for specialized attributes `PAPI_UNSUPPORTED`, `PAPI_DEFAULT`, `PAPI_UNKNOWN`, `PAPI_NO_VALUE`, `PAPI_NOT_SETTABLE`, `PAPI_DELETE`
min	minimum value in a range
name	attribute name
string	string value
type	attribute type (`PAPI_STRING`, `PAPI_INTEGER`, `PAPI_BOOLEAN`, `PAPI_RANGE`, `PAPI_RESOLUTION`, `PAPI_DATETIME`, `PAPI_COLLECTION`, `PAPI_METADATA`)
units	resolution unit of measure (`PAPI_RES_PER_INCH` or `PAPI_RES_PER_CM`)
upper	upper bound for a range of integer
value	attribute value
vptr	pointer to arbitrary data
x	horizontal (x) resolution
xres	horizontal (x) component of a resolution
y	vertical (y) resolution
yres	vertical (y) component of a resolution

Description The `papiAttributeListAdd*()` functions add new attributes and/or values to the attribute list passed in. If necessary, the attribute list passed in is expanded to contain a new attribute pointer for any new attributes added to the list. The list is null-terminated. Space for the new attributes and values is allocated and the name and value are copied into this allocated space.

If PAPI_ATTR_REPLACE is specified in flags, any existing attribute values are freed and replaced with the supplied value.

If PAPI_ATTR_APPEND is specified, the supplied value is appended to the attribute's list of values.

If PAPI_ATTR_EXCL is specified, the operation succeeds only if the attribute was not previously defined.

The papiAttributeListGet*() functions retrieve an attribute value from an attribute list. If the attribute is a multi-valued attribute, the first call to retrieve a value should pass in an iterator and attribute name. Subsequent calls to retrieve additional values should pass in the iterator and a null value for the attribute name. If a single-valued attribute is to be retrieved, NULL can be used in place of the iterator.

Upon successful completion of a get operation, the value passed in (string, integer, boolean, ...) is changed to the value from the attribute list. If the operation fails for any reason (type mismatch, not found, ...), the value passed in remains untouched.

The resulting value returned from a get operation is returned from the attribute list's allocated memory. It is not guaranteed to be available after the attribute list has been freed.

The papiAttributeListDelete() function removes an attribute from a supplied list.

The papiAttributeListFind() function allows an application to retrieve an entire attribute structure from the passed-in attribute list.

The papiAttributeListGetNext() function allows an application to walk through an attribute list returning subsequent attributes from the list. With the first call, the iterator should be initialized to NULL and subsequent calls should use NULL for the list argument.

The papiAttributeListFree() function deallocates all memory associated with an attribute list, including values that might have been retrieved previously using papiAttributeListGet*() calls.

The papiAttributeListFromString() function takes in a string representation of a set of attributes, parses the string and adds the attributes to the passed in attribute list using the flags to determine how to add them. String values are specified with “key=value”. Integer values are specified with “key=number”. Boolean values are specified with either “key=(true|false)” or “[no]key”. Multiple attributes can be specified in the string by separating them with a whitespace character.

The papiAttributeListToString() function converts an attribute list to a string representation that can be displayed to a user. The delimiter value is placed between attributes in the string.

Return Values These functions return `PAPI_OK` upon successful completion and one of the following on failure:

`PAPI_BAD_ARGUMENT`	The supplied arguments were not valid.
`PAPI_CONFLICT`	There was an attribute type mismatch.
`PAPI_NOT_FOUND`	The requested attribute could not be found.
`PAPI_NOT_POSSIBLE`	The requested operation could not be performed due to buffer overflow.
`PAPI_TEMPORARY_ERROR`	Memory could not be allocated to add to the attribute list.

Examples **EXAMPLE 1** The following example manipulates a PAPI attribute list.

```
/*
 * program to manipulate a PAPI attribute list
 */
#include <stdio.h>
#include <papi.h>

/*ARGSUSED*/
int
main(int ac, char *av[])
{
    char buf[BUFSIZ];
    papi_status_t status;
    papi_attribute_t **list = NULL;
    void *iter = NULL;
    char *string = NULL;
    int32_t integer = 0;

    /* build an attribute list */
    (void) papiAttributeListAddString(&list, PAPI_ATTR_EXCL,
                                "job-title", "example");
    (void) papiAttributeListAddInteger(&list, PAPI_ATTR_EXCL,
                                "copies", 1);
    (void) papiAttributeListFromString(&list, PAPI_ATTR_REPLACE, av[1]);
    status = papiAttributeListAddString(&list, PAPI_ATTR_EXCL,
                                "document-format", "text/plain");
    if (status != PAPI_OK)
        printf("failed to set document-format to text/plain: %s\n",
               papiStatusString(status));

    /* dump the list */
    status = papiAttributeListToString(list, "\n\t", buf, sizeof (buf));
    if (status == PAPI_OK)
        printf("Attributes: %s\n", buf);
```

EXAMPLE 1 The following example manipulates a PAPI attribute list. *(Continued)*

```
    else
        printf("Attribute list to big to dump\n");


    /* retreive various elements */
    integer = 12;
    (void) papiAttributeListGetInteger(list, NULL, "copies", &integer);
    printf("copies: %d\n", integer);

    string = NULL;
    for (status = papiAttributeListGetString(list, &oter,
                                             "job-files", &string);
         status == PAPI_OK;
         status = papiAttributeListGetString(list, &iter, NULL, &string))
        printf("file: %s\n", string);

    papiAttributeListFree(list);
}
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Volatile
MT-Level	Safe

See Also

`libpapi(3LIB)`, `attributes(5)`

Name papiJobSubmit, papiJobSubmitByReference, papiJobValidate, papiJobStreamOpen, papiJobStreamWrite, papiJobStreamClose, papiJobQuery, papiJobModify, papiJobMove, papiJobCancel, papiJobHold, papiJobRelease, papiJobRestart, papiJobPromote, papiJobGetAttributeList, papiJobGetPrinterName, papiJobGetId, papiJobGetJobTicket, papiJobFree, papiJobListFree – job object manipulation

Synopsis

```
cc [ flag... ] file... -lpapi [ library... ]
#include <papi.h>

papi_status_t papiJobSubmit(papi_service_t handle,
     char *printer, papi_attribute_t **job_attributes,
     papi_job_ticket_t *job_ticket, char **files,
     papi_job_t *job);

papi_status_t papiJobSubmitByReference(papi_service_t handle,
     char *printer, papi_attribute_t **job_attributes,
     papi_job_ticket_t *job_ticket, char **files,
     papi_job_t *job);

papi_status_t papiJobValidate(papi_service_t handle,
     char *printer, papi_attribute_t **job_attributes,
     papi_job_ticket_t *job_ticket, char **files,
     papi_job_t *job);

papi_status_t papiJobStreamOpen(papi_service_t handle,
     char *printer, papi_attribute_t **job_attributes,
     papi_job_ticket_t *job_ticket, papi_stream_t *stream);

papi_status_t papiJobStreamWrite(papi_service_t handle,
     papi_stream_t stream, void *buffer, size_t buflen);

papi_status_t papiJobStreamClose(papi_service_t handle,
     papi_stream_t stream, papi_job_t *job);

papi_status_t papiJobQuery(papi_service_t handle,
     char *printer, int32_t job_id, char **requested_attrs,
     papi_job_t *job);

papi_status_t papiJobModify(papi_service_t handle,
     char *printer, int32_t job_id,
     papi_attribute_t **attributes, papi_job_t *job);

papi_status_t papiJobMove(papi_service_t handle,
     char *printer, int32_t job_id, char *destination);

papi_status_t papiJobCancel(papi_service_t handle,
     char *printer, int32_t job_id);

papi_status_t papiJobHold(papi_service_t handle,
     char *printer, int32_t job_id);

papi_status_t papiJobRelease(papi_service_t handle,
     char *printer, int32_t job_id);
```

```
papi_status_t papiJobRestart(papi_service_t handle,
     char *printer, int32_t job_id);

papi_status_t papiJobPromote(papi_service_t handle,
     char *printer, int32_t job_id);

papi_attribute_t **papiJobGetAttributeList(papi_job_t job);

char *papiJobGetPrinterName(papi_job_t job);

int32_t papiJobGetId(papi_job_t job);

papi_job_ticket_t *papiJobGetJobTicket(papi_job_t job);

void papiJobFree(papi_job_t job);

void papiJobListFree(papi_job_t *jobs);
```

Parameters

attributes	a set of attributes to be applied to a printer object
buffer	a buffer of data to be written to the job stream
bufflen	the size of the supplied buffer
destination	the name of the printer where a print job should be relocated, which must reside within the same print services as the job is currently queued
files	files to use during job submission
handle	a pointer to a handle to be used for all PAPI operations that is created by calling `papiServiceCreate()`
job	a pointer to a printer object (initialized to NULL) to be filled in by `papiJobQuery()`, `papiJobSubmit()`, `papiJobSubmitByReference()`, `papiJobValidate()`, `papiJobStreamClose()`, and `papiJobModify()`
job_attributes	attributes to apply during job creation or modification
job_id	ID number of the job reported on or manipulated
job_ticket	unused
jobs	a list of job objects returned by `papiPrinterListJobs()` or `papiPrinterPurgeJobs()`
printer	name of the printer where the job is or should reside
requested_attrs	a null-terminated array of pointers to attribute names requested during job enumeration (`papiPrinterListJobs()`) or job query (`papiJobQuery()`)
stream	a communication endpoint for sending print job data

Description The `papiJobSubmit()` function creates a print job containing the passed in files with the supplied attributes. When the function returns, the data in the passed files will have been copied by the print service. A job object is returned that reflects the state of the job.

The `papiJobSubmitByReference()` function creates a print job containing the passed in files with the supplied attributes. When the function returns, the data in the passed files might have been copied by the print service. A job object is returned that reflects the state of the job.

The `papiJobStreamOpen()`, `papiJobStreamWrite()`, `papiJobStreamClose()` functions create a print job by opening a stream, writing to the stream, and closing it.

The `papiJobValidate()` function validates that the supplied attributes and files will result in a valid print job.

The `papiJobQuery()` function retrieves job information from the print service.

The `papiJobModify()` function modifies a queued job according to the attribute list passed into the call. A job object is returned that reflects the state of the job after the modification has been applied.

The `papiJobMove()` function moves a job from its current queue to the named destination within the same print service.

The `papiJobCancel()` function removes a job from the queue.

The `papiJobHold()` and `papiJobRelease()` functions set the job state to "held" or "idle" to indicate whether the job is eligible for processing.

The `papiJobRestart()` function restarts processing of a currently queued print job.

The `papiJobGetAttributeList()` function returns a list of attributes describing the job. This list can be searched and/or enumerated using `papiAttributeList*()` calls. See `papiAttributeListAddValue(3PAPI)`.

The `papiJobGetPrinterName()` function returns the name of the queue where the job is currently queued.

The `papiJobGetId()` function returns a job identifier number from the job object passed in.

The `papiJobPromote()` function moves a job to the head of the print queue.

The `papiJobGetJobTicket()` function retrieves a pointer to a job ticket associated with the job object.

The `papiJobFree()` and `papiJobListFree()` functions deallocate memory allocated for the return of printer object(s) from functions that return printer objects.

Return Values Upon successful completion, all `papiJob*()` functions that return a value return `PAPI_OK`. Otherwise, they return an appropriate `papi_status_t` indicating the type of failure.

Upon successful completion, `papiJobGetAttributeList()` returns a pointer to the requested data. Otherwise, it returns `NULL`.

Examples **EXAMPLE 1** Enumerate all jobs in a queue

```
/*
 * program to enumerate queued jobs using PAPI interfaces.
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <libintl.h>
#include <pwd.h>
#include <papi.h>

static int
authCB(papi_service_t svc, void *app_data)
{
    char prompt[BUFSIZ];
    char *user, *svc_name, *passphrase;

    /* get the name of the service we are contacting */
    if ((svc_name = papiServiceGetServiceName(svc)) == NULL)
            return (-1);

    /* find out who we are supposed to be */
    if ((user = papiServiceGetUserName(svc)) == NULL) {
            struct passwd *pw;

            if ((pw = getpwuid(getuid())) != NULL)
                    user = pw->pw_name;
            else
                    user = "nobody";
    }

    /* build the prompt string */
    snprintf(prompt, sizeof (prompt),
            gettext("passphrase for %s to access %s: "), user,
                  svc_name);

    /* ask for the passphrase */
    if ((passphrase = getpassphrase(prompt)) != NULL)
            papiServiceSetPassword(svc, passphrase);

    return (0);
```

EXAMPLE 1 Enumerate all jobs in a queue *(Continued)*

```
}

/*ARGSUSED*/
int
main(int ac, char *av[])
{
    papi_status_t status;
    papi_service_t svc = NULL;
    papi_job_t *jobs = NULL;
    char *svc_name = NULL;
    char *pname = "unknown";
    int c;

    while ((c = getopt(ac, av, "s:p:")) != EOF)
        switch (c) {
        case 's':
            svc_name = optarg;
            break;
        case 'p':
            pname = optarg;
            break;
        }

    status = papiServiceCreate(&svc, svc_name, NULL, NULL, authCB,
                           PAPI_ENCRYPT_NEVER, NULL);

    if (status != PAPI_OK) {
        printf("papiServiceCreate(%s): %s\
", svc_name ? svc_name :
                      "NULL", papiStatusString(status));
        papiServiceDestroy(svc);
        exit(1);
    }

    status = papiPrinterListJobs(svc, pname, NULL, 0, 0, &jobs);
    if (status != PAPI_OK) {
        printf("papiPrinterListJobs(%s): %s\
", pname,
                    papiStatusString(status));
        papiServiceDestroy(svc);
        exit(1);
    }

    if (jobs != NULL) {
        int i;
```

EXAMPLE 1 Enumerate all jobs in a queue *(Continued)*

```
        for (i = 0; jobs[i] != NULL; i++) {
            papi_attribute_t **list = papiJobGetAttributeList(jobs[i]);

            if (list != NULL) {
                char *name = "unknown";
                            int32_t id = 0;
                char *buffer = NULL;
                size_t size = 0;

               (void) papiAttributeListGetString(list, NULL,
                                   "printer-name", &name);
               (void) papiAttributeListGetInteger(list, NULL,
                                   "job-id", &id);
                while (papiAttributeListToString(list, "\
\\t", buffer,
                            size) != PAPI_OK)
                    buffer = realloc(buffer, size += BUFSIZ);

                printf("%s-%d:\
\\t%s\
", name, id, buffer);
                free(buffer);
            }
        }
        papiJobListFree(jobs);
    }

    papiServiceDestroy(svc);

    exit(0);
}
```

EXAMPLE 2 Dump all job attributes.

```
/*
 * program to dump a queued job's attributes using PAPI interfaces.
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <libintl.h>
#include <pwd.h>
#include <papi.h>

static int
```

EXAMPLE 2 Dump all job attributes. *(Continued)*

```
authCB(papi_service_t svc, void *app_data)
{
    char prompt[BUFSIZ];
    char *user, *svc_name, *passphrase;

    /* get the name of the service we are contacting */
    if ((svc_name = papiServiceGetServiceName(svc)) == NULL)
            return (-1);

    /* find out who we are supposed to be */
    if ((user = papiServiceGetUserName(svc)) == NULL) {
            struct passwd *pw;

            if ((pw = getpwuid(getuid())) != NULL)
                    user = pw->pw_name;
            else
                    user = "nobody";
    }

    /* build the prompt string */
    snprintf(prompt, sizeof (prompt),
            gettext("passphrase for %s to access %s: "), user,
                     svc_name);

    /* ask for the passphrase */
    if ((passphrase = getpassphrase(prompt)) != NULL)
            papiServiceSetPassword(svc, passphrase);

    return (0);
}

/*ARGSUSED*/
int
main(int ac, char *av[])
{
    papi_status_t status;
    papi_service_t svc = NULL;
    papi_job_t job = NULL;
    char *svc_name = NULL;
    char *pname = "unknown";
    int id = 0;
    int c;

    while ((c = getopt(ac, av, "s:p:j:")) != EOF)
        switch (c) {
```

EXAMPLE 2 Dump all job attributes. *(Continued)*

```
        case 's':
            svc_name = optarg;
            break;
        case 'p':
            pname = optarg;
            break;
        case 'j':
            id = atoi(optarg);
            break;
        }

    status = papiServiceCreate(&svc, svc_name, NULL, NULL, authCB,
                           PAPI_ENCRYPT_NEVER, NULL);

    if (status != PAPI_OK) {
        printf("papiServiceCreate(%s): %s\
", svc_name ? svc_name :
                       "NULL", papiStatusString(status));
        papiServiceDestroy(svc);
        exit(1);
    }

    status = papiJobQuery(svc, pname, id, NULL, &job);
    if ((status == PAPI_OK) && (job != NULL)) {
        papi_attribute_t **list = papiJobGetAttributeList(job);

        if (list != NULL) {
            char *name = "unknown";
            int32_t id = 0;
            char *buffer = NULL;
            size_t size = 0;

            (void) papiAttributeListGetString(list, NULL,
                                "printer-name", &name);
            (void) papiAttributeListGetInteger(list, NULL,
                                "job-id", &id);
            while (papiAttributeListToString(list, "\
\\t", buffer, size)
                           != PAPI_OK)
                buffer = realloc(buffer, size += BUFSIZ);

            printf("%s-%d:\
\\t%s\
", name, id, buffer);
            free(buffer);
```

EXAMPLE 2 Dump all job attributes. *(Continued)*

```
        }
    } else
        printf("papiJobQuery(%s-%d): %s\
", pname, id,
                    papiStatusString(status));

    papiJobFree(job);
    papiServiceDestroy(svc);

    exit(0);
}
```

EXAMPLE 3 Submit a job (stream).

```
/*
 * program to submit a job from standard input.
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <libintl.h>
#include <pwd.h>
#include <papi.h>

static int
authCB(papi_service_t svc, void *app_data)
{
    char prompt[BUFSIZ];
    char *user, *svc_name, *passphrase;

    /* get the name of the service we are contacting */
    if ((svc_name = papiServiceGetServiceName(svc)) == NULL)
            return (-1);

    /* find out who we are supposed to be */
    if ((user = papiServiceGetUserName(svc)) == NULL) {
            struct passwd *pw;

            if ((pw = getpwuid(getuid())) != NULL)
                    user = pw->pw_name;
            else
                    user = "nobody";
    }

    /* build the prompt string */
    snprintf(prompt, sizeof (prompt),
```

EXAMPLE 3 Submit a job (stream). *(Continued)*

```
            gettext("passphrase for %s to access %s: "), user,
                    svc_name);

    /* ask for the passphrase */
    if ((passphrase = getpassphrase(prompt)) != NULL)
            papiServiceSetPassword(svc, passphrase);

    return (0);
}

/*ARGSUSED*/
int
main(int ac, char *av[])
{
    papi_status_t status;
    papi_service_t svc = NULL;
    papi_stream_t stream = NULL;
    papi_job_t job = NULL;
    papi_attribute_t **attrs = NULL;
    char *svc_name = NULL;
    char *pname = "unknown";
    int id = 0;
    int c;
    int rc;
    char buf[BUFSIZ];

    while ((c = getopt(ac, av, "s:p:")) != EOF)
        switch (c) {
        case 's':
            svc_name = optarg;
            break;
        case 'p':
            pname = optarg;
            break;
        }

    status = papiServiceCreate(&svc, svc_name, NULL, NULL, authCB,
                            PAPI_ENCRYPT_NEVER, NULL);

    if (status != PAPI_OK) {
        printf("papiServiceCreate(%s): %s\
", svc_name ? svc_name :
                    "NULL", papiStatusString(status));
        papiServiceDestroy(svc);
        exit(1);
```

EXAMPLE 3 Submit a job (stream). *(Continued)*

```
    }

    papiAttributeListAddInteger(&attrs, PAPI_ATTR_EXCL, "copies", 1);
    papiAttributeListAddString(&attrs, PAPI_ATTR_EXCL,
                "document-format", "application/octet-stream");
    papiAttributeListAddString(&attrs, PAPI_ATTR_EXCL,
                "job-title", "Standard Input");

    status = papiJobStreamOpen(svc, pname, attrs, NULL, &stream);
    while ((status == PAPI_OK) && ((rc = read(0, buf,
                    sizeof (buf))) > 0))
            status = papiJobStreamWrite(svc, stream, buf, rc);

    if (status == PAPI_OK)
            status = papiJobStreamClose(svc, stream, &job);


    if ((status == PAPI_OK) && (job != NULL)) {
        papi_attribute_t **list = papiJobGetAttributeList(job);

        if (list != NULL) {
            char *name = "unknown";
            int32_t id = 0;
            char *buffer = NULL;
            size_t size = 0;

            (void) papiAttributeListGetString(list, NULL,
                                    "printer-name", &name);
            (void) papiAttributeListGetInteger(list, NULL,
                                    "job-id", &id);
            while (papiAttributeListToString(list, "\
\\t", buffer, size)
                                != PAPI_OK)
                buffer = realloc(buffer, size += BUFSIZ);

            printf("%s-%d:\
\\t%s\
", name, id, buffer);
            free(buffer);
        }
    } else
        printf("papiJobStream*(%s-%d): %s\
", pname, id,
                    papiStatusString(status));
```

EXAMPLE 3 Submit a job (stream). *(Continued)*

```
    papiJobFree(job);
    papiServiceDestroy(svc);

    exit(0);
}
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Volatile
MT-Level	Safe

See Also

libpapi(3LIB), papiAttributeListAddValue(3PAPI), attributes(5)

Name papiLibrarySupportedCall, papiLibrarySupportedCalls – determine if a PAPI function returns valid data

Synopsis

```
cc [ flag... ] file... -lpapi [ library... ]
#include <papi.h>

char papiLibrarySupportedCall(const char *name);

char **papiLibrarySupportedCalls(void);
```

Parameters *name* the name of a PAPI function

Description The `papiLibrarySupportedCall()` function queries to determine if a particular PAPI function returns valid data other than `PAPI_OPERATION_NOT_SUPPORTED`.

The `papiLibrarySupportedCalls()` function enumerates all PAPI functions that return valid data other than `PAPI_OPERATION_NOT_SUPPORTED`.

Return Values The `papiLibrarySupportedCall()` function returns `PAPI_TRUE` if the specified PAPI function returns valid data other than `PAPI_OPERATION_NOT_SUPPORTED`. Otherwise, `PAPI_FALSE` is returned.

The `papiLibrarySupportedCalls()` function returns a null-terminated array of strings listing all of the PAPI functions that return valid data other than `PAPI_OPERATION_NOT_SUPPORTED`. Otherwise, `NULL` is returned.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Volatile
MT-Level	Safe

See Also `libpapi(3LIB)`, `attributes(5)`

Name papiPrintersList, papiPrinterQuery, papiPrinterAdd, papiPrinterModify, papiPrinterRemove, papiPrinterDisable, papiPrinterEnable, papiPrinterPause, papiPrinterResume, papiPrinterPurgeJobs, papiPrinterListJobs, papiPrinterGetAttributeList, papiPrinterFree, papiPrinterListFree – print object manipulation

Synopsis

```
cc [ flag... ] file... -lpapi [ library... ]
#include <papi.h>

papi_status_t papiPrintersList(papi_service_t handle,
     char **requested_attrs, papi_filter_t *filter,
     papi_printer_t **printers);

papi_status_t papiPrinterQuery(papi_service_t handle, char *name,
     char **requested_attrs, papi_attribute_t **job_attributes,
     papi_printer_t *printer);

papi_status_t papiPrinterAdd(papi_service_t handle, char *name,
     papi_attribute_t **attributes, papi_printer_t *printer);

papi_status_t papiPrinterModify(papi_service_t handle, char *name,
     papi_attribute_t **attributes, papi_printer_t *printer);

papi_status_t papiPrinterRemove(papi_service_t handle, char *name);

papi_status_t papiPrinterDisable(papi_service_t handle, char *name,
     char *message);

papi_status_t papiPrinterEnable(papi_service_t handle, char *name);

papi_status_t papiPrinterPause(papi_service_t handle, char *name,
     char *message);

papi_status_t papiPrinterResume(papi_service_t handle, char *name);

papi_status_t papiPrinterPurgeJobs(papi_service_t handle, char *name,
     papi_job_t **jobs);

papi_status_t papiPrinterListJobs(papi_service_t handle, char *name,
     char **requested_attrs, int type_mask, int max_num_jobs,
     papi_job_t **jobs);

papi_attribute_t **papiPrinterGetAttributeList(papi_printer_t printer);

void papiPrinterFree(papi_printer_t printer);

void papiPrinterListFree(papi_printer_t *printers);
```

Parameters

attributes	a set of attributes to be applied to a printer object
filter	a filter to be applied during printer enumeration
handle	a pointer to a handle to be used for all PAPI operations, created by calling `papiServiceCreate()`
job_attributes	unused

jobs	a pointer to a list to return job objects (initialized to `NULL`) enumerated by `papiPrinterGetJobs()`
max_num_jobs	the maximum number of jobs to return from a `papiPrinterGetJobs()` request
message	a message to be associated with a printer while disabled or paused
name	the name of the printer object being operated on
printer	a pointer to a printer object (initialized to `NULL`) to be filled in by `papiPrinterQuery()`, `papiPrinterAdd()`, and `papiPrinterModify()`
printers	a pointer to a list to return printer objects (initialized to `NULL`) enumerated by `papiPrintersList()`
requested_attrs	a null-terminated array of pointers to attribute names requested during printer enumeration (`papiPrintersList()`), printer query (`papiPrinterQuery()`), or job enumeration (`papiPrinterListJobs()`)
type_mask	a bit field indicating which type of jobs to return `PAPI_LIST_JOBS_OTHERS` include jobs submitted by others. The default is to report only on your own jobs
	`PAPI_LIST_JOBS_COMPLETED` include completed jobs
	`PAPI_LIST_JOBS_NOT_COMPLETED` include jobs not complete
	`PAPI_LIST_JOBS_ALL` report on all jobs

Description The `papiPrintersList()` function retrieves the requested attributes from the print service(s) for all available printers. Because the Solaris implementation is name service-enabled, applications should retrieve only the `printer-name` and `printer-uri-supported` attributes using this function, thereby reducing the overhead involved in generating a printer list. Further integration of printer state and capabilities can be performed with `papiPrinterQuery()`.

The `papiPrinterAdd()`, `papiPrinterModify()`, and `papiPrinterRemove()` functions allow for creation, modification, and removal of print queues. Print queues are added or modified according to the attribute list passed into the call. A printer object is returned that reflects the configuration of the printer after the addition or modification has been applied. At this time, they provide only minimal functionality and only for the LP print service.

The `papiPrinterDisable()` and `papiPrinterEnable()` functions allow applications to turn off and on queueing (accepting print requests) for a print queue. The `papiPrinterEnable()` and `papiPrinterDisable()` functions allow applications to turn on and off print job processing for a print queue.

The `papiPrinterPause()` function stops queueing of print jobs on the named print queue.

The `papiPrinterResume()` function resumes queueing of print jobs on the named print queue.

The `papiPrinterPurgeJobs()` function allows applications to delete all print jobs that it has privilege to remove. A list of cancelled jobs is returned in the jobs argument.

The `papiPrinterListJobs()` function enumerates print jobs on a particular queue. `papiPrinterGetAttributeList()` retrieves an attribute list from a printer object.

The `papiPrinterGetAttributeList()` function retrieves an attribute list from a printer object returned from `papiPrinterQuery()`, `papiPrintersList()`, `papiPrinterModify()`, and `papiPrinterAdd()`. This attribute list can be searched for various information about the printer object.

The `papiPrinterFree()` and `papiPrinterListFree()` functions deallocate memory allocated for the return of printer object(s) from functions that return printer objects.

Return Values Upon successful completion, all functions that return a value return `PAPI_OK`. Otherwise, they return an appropriate `papi_status_t()` indicating the type of failure.

Upon successful completion, `papiPrinterGetAttributeList()` returns a pointer to the requested data. Otherwise, it returns `NULL`.

Examples **EXAMPLE 1** Enumerate all available printers.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <libintl.h>
#include <pwd.h>
#include <papi.h>

static int
authCB(papi_service_t svc, void *app_data)
{
    char prompt[BUFSIZ];
    char *user, *svc_name, *passphrase;

    /* get the name of the service we are contacting */
    if ((svc_name = papiServiceGetServiceName(svc)) == NULL)
            return (-1);

    /* find out who we are supposed to be */
    if ((user = papiServiceGetUserName(svc)) == NULL) {
            struct passwd *pw;

            if ((pw = getpwuid(getuid())) != NULL)
```

EXAMPLE 1 Enumerate all available printers. *(Continued)*

```
                user = pw->pw_name;
        else
                user = "nobody";
    }

    /* build the prompt string */
    snprintf(prompt, sizeof (prompt),
            gettext("passphrase for %s to access %s: "), user,
                   svc_name);

    /* ask for the passphrase */
    if ((passphrase = getpassphrase(prompt)) != NULL)
            papiServiceSetPassword(svc, passphrase);

    return (0);
}

/*ARGSUSED*/
int
main(int ac, char *av[])
{
    papi_status_t status;
    papi_service_t svc = NULL;
    papi_printer_t *printers = NULL;
    char *attrs[] = { "printer-name", "printer-uri-supported", NULL };
    char *svc_name = NULL;
    int c;

    while ((c = getopt(ac, av, "s:")) != EOF)
        switch (c) {
        case 's':
            svc_name = optarg;
            break;
        }

    status = papiServiceCreate(&svc, svc_name, NULL, NULL, authCB,
                           PAPI_ENCRYPT_NEVER, NULL);

    if (status != PAPI_OK) {
        printf("papiServiceCreate(%s): %s\n", svc_name ? svc_name :
                    "NULL", papiStatusString(status));
        papiServiceDestroy(svc);
        exit(1);
    }

    status = papiPrintersList(svc, attrs, NULL, &printers);
```

EXAMPLE 1 Enumerate all available printers. *(Continued)*

```
    if (status != PAPI_OK) {
        printf("papiPrintersList(%s): %s\n", svc_name ? svc_name :
                    "NULL", papiStatusString(status));
        papiServiceDestroy(svc);
        exit(1);
    }

    if (printers != NULL) {
        int i;

        for (i = 0; printers[i] != NULL; i++) {
            papi_attribute_t **list =
                papiPrinterGetAttributeList(printers[i]);

            if (list != NULL) {
                char *name = "unknown";
                char *uri = "unknown";

                (void) papiAttributeListGetString(list, NULL,
                                    "printer-name", &name);

                (void) papiAttributeListGetString(list, NULL,
                                    "printer-uri-supported", &uri);
                printf("%s is %s\
", name, uri);
            }
        }
        papiPrinterListFree(printers);
    }

    papiServiceDestroy(svc);

    exit(0);
}
```

EXAMPLE 2 Dump all printer attributes.

```
/*
 * program to query a printer for it's attributes via PAPI
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <libintl.h>
#include <pwd.h>
#include <papi.h>
```

EXAMPLE 2 Dump all printer attributes. *(Continued)*

```
static int
authCB(papi_service_t svc, void *app_data)
{
    char prompt[BUFSIZ];
    char *user, *svc_name, *passphrase;

    /* get the name of the service we are contacting */
    if ((svc_name = papiServiceGetServiceName(svc)) == NULL)
            return (-1);

    /* find out who we are supposed to be */
    if ((user = papiServiceGetUserName(svc)) == NULL) {
            struct passwd *pw;

            if ((pw = getpwuid(getuid())) != NULL)
                    user = pw->pw_name;
            else
                    user = "nobody";
    }

    /* build the prompt string */
    snprintf(prompt, sizeof (prompt),
            gettext("passphrase for %s to access %s: "), user,
                    svc_name);

    /* ask for the passphrase */
    if ((passphrase = getpassphrase(prompt)) != NULL)
            papiServiceSetPassword(svc, passphrase);

    return (0);
}

/*ARGSUSED*/
int
main(int ac, char *av[])
{
    papi_status_t status;
    papi_service_t svc = NULL;
    papi_printer_t printer = NULL;
    char *svc_name = NULL;
    char *pname = "unknown";
    int c;

    while ((c = getopt(ac, av, "s:p:")) != EOF)
        switch (c) {
        case 's':
```

EXAMPLE 2 Dump all printer attributes. *(Continued)*

```
            svc_name = optarg;
            break;
        case 'p':
            pname = optarg;
            break;
        }

    status = papiServiceCreate(&svc, svc_name, NULL, NULL, authCB,
                            PAPI_ENCRYPT_NEVER, NULL);

    if (status != PAPI_OK) {
        printf("papiServiceCreate(%s): %s\n", svc_name ? svc_name :
                    "NULL", papiStatusString(status));
        papiServiceDestroy(svc);
        exit(1);
    }

    status = papiPrinterQuery(svc, pname, NULL, NULL, &printer);
    if ((status == PAPI_OK) && (printer != NULL)) {
        papi_attribute_t **list = papiPrinterGetAttributeList(printer);
        char *buffer = NULL;
        size_t size = 0;

        while (papiAttributeListToString(list, "\n\t", buffer, size)
                    != PAPI_OK)
            buffer = realloc(buffer, size += BUFSIZ);

        printf("%s:\n\t%s\n", pname, buffer);
    } else
        printf("papiPrinterQuery(%s): %s\n", pname,
                    papiStatusString(status));

    papiPrinterFree(printer);
    papiServiceDestroy(svc);

    exit(0);
}
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Volatile
MT-Level	Safe

See Also libpapi(3LIB), attributes(5)

Name papiServiceCreate, papiServiceDestroy, papiServiceSetUserName, papiServiceSetPassword, papiServiceSetEncryption, papiServiceSetAuthCB, papiServiceSetAppData, papiServiceGetServiceName, papiServiceGetUserName, papiServiceGetPassword, papiServiceGetEncryption, papiServiceGetAppData, papiServiceGetAttributeList, papiServiceGetStatusMessage – service context manipulation

Synopsis

```
cc [ flag... ] file... -lpapi [ library... ]
#include <papi.h>

papi_status_t papiServiceCreate(papi_service_t *handle,
     char *service_name, char *user_name, char *password,
     int (*authCB)(papi_service_t svc, void *app_data),
     papi_encryption_t encryption, void *app_data);

void papiServiceDestroy(papi_service_t handle);

papi_status_t papiServiceSetUserName(papi_service_t handle,
     char *user_name);

papi_status_t papiServiceSetPassword(papi_service_t handle,
     char *password);

papi_status_t papiServiceSetEncryption(papi_service_t handle,
     papi_encryption_t encryption);

papi_status_t papiServiceSetAuthCB(papi_service_t handle,
     int (*authCB)(papi_service_t s, void *app_data));

papi_status_t papiServiceSetAppData(papi_service_t handle,
     void *app_data);

char *papiServiceGetServiceName(papi_service_t handle);

char *papiServiceGetUserName(papi_service_t handle);

char *papiServiceGetPassword(papi_service_t handle);

papi_encryption_t papiServiceGetEncryption(papi_service_t handle);

void *papiServiceGetAppData(papi_service_t handle);

papi_attribute_t **papiServiceGetAttributeList(papi_service_t handle);

char *papiServiceGetStatusMessage(papi_service_t handle);
```

Parameters

app_data	a set of additional data to be passed to the *authCB* if/when it is called
authCB	a callback routine use to gather additional authentication information on behalf of the print service
encryption	whether or not encryption should be used for communication with the print service, where applicable. If `PAPI_ENCRYPT_IF_REQUESTED` is specified, encryption will be used if the print service requests it. If `PAPI_ENCRYPT_NEVER` is specified, encryption will not be used while

communicating with the print service. If `PAPI_ENCRYPT_REQUIRED` or `PAPI_ENCRYPT_ALWAYS` is specified, encryption will be required while communicating with the print service

handle — a pointer to a handle to be used for all `libpapi` operations. This handle should be initialized to `NULL` prior to creation

password — a plain text password to be used during any required user authentication with the print service function set with `papiServiceSetAuthCB()`. This provides the callback function a means of interrogating the service context for user information and setting a password.

s — the service context passed into the the authentication callback

service_name — the name of a print service to contact. This can be `NULL`, a print service name like "lpsched", a resolvable printer name, or a `printer-uri` like `ipp://server/printers/queue`.

svc — a handle (service context) used by subsequent PAPI calls to keep/pass information across PAPI calls. It generally contains connection, state, and authentication information.

user_name — the name of the user to act on behalf of while contacting the print service. If a value of `NULL` is used, the user name associated with the current processes UID will be used. Specifying a user name might require further authentication with the print service.

Description

The `papiServiceCreate()` function creates a service context for use in subsequent calls to `libpapi` functions. The context is returned in the handle argument. This context must be destroyed using `papiServiceDestroy()` even if the `papiServiceCreate()` call failed.

The `papiServiceSet*()` functions modifies the requested value in the service context passed in. It is recommended that these functions be avoided in favor of supplying the information when the context is created.

The `papiServiceGetStatusMessage()` function retrieves a detailed error message associated with the service context. This message will apply to the last failed operation.

The remaining `papiServiceGet*()` functions return the requested information associated with the service context. A value of `NULL` indicates that the requested value was not initialized or is unavailable.

Return Values

Upon successful completion, `papiServiceCreate()` and the `papiServiceSet*()` functions return `PAPI_OK`. Otherwise, they return an appropriate `papi_status_t` indicating the type of failure.

Upon successful completion, the `papiServiceGet*()` functions return a pointer to the requested data. Otherwise, they return `NULL`.

Examples

EXAMPLE 1 Create a PAPI service context.

```
/*
 * program to create a PAPI service context.
 */
#include <stdio.h>
#include <papi.h>

static int
authCB(papi_service_t svc, void *app_data)
{
    char prompt[BUFSIZ];
    char *user, *svc_name, *passphrase;

    /* get the name of the service we are contacting */
    if ((svc_name = papiServiceGetServiceName(svc)) == NULL)
            return (-1);

    /* find out who we are supposed to be */
    if ((user = papiServiceGetUserName(svc)) == NULL) {
            struct passwd *pw;

            if ((pw = getpwuid(getuid())) != NULL)
                    user = pw->pw_name;
            else
                    user = "nobody";
    }

    /* build the prompt string */
    snprintf(prompt, sizeof (prompt),
            gettext("passphrase for %s to access %s: "), user,
                  svc_name);

    /* ask for the passphrase */
    if ((passphrase = getpassphrase(prompt)) != NULL)
            papiServiceSetPassword(svc, passphrase);

    return (0);
}

/*ARGSUSED*/
int
main(int ac, char *av[])
{
    char buf[BUFSIZ];
    papi_status_t status;
    papi_service_t *svc = NULL;
```

EXAMPLE 1 Create a PAPI service context. *(Continued)*

```
    status = papiServiceCreate(&svc, av[1], NULL, NULL, authCB,
                            PAPI_ENCRYPT_NEVER, NULL);

    if (status != PAPI_OK) {
        /* do something */
    } else
        printf("Failed(%s): %s: %s\n", av[1], papiStatusString(status),
                papiStatusMessage(svc));

    papiServiceDestroy(svc);
}
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Volatile
MT-Level	Safe

See Also

libpapi(3LIB), attributes(5)

Name papiStatusString – return the string equivalent of a papi_status_t

Synopsis

```
cc [ flag... ] file... -lpapi [ library... ]
#include <papi.h>

char *papiStatusString(papi_status_t status);
```

Parameters *status* a `papi_status_t` returned from most `papi*()` functions

Description The `papiStatusString()` function takes a *status* value and returns a localized human-readable version of the supplied status.

Return Values The `papiStatusString()` function always returns a text string.

Errors None.

Examples **EXAMPLE 1** Print status.

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

/*ARGSUSED*/
int
main(int ac, char *av[])
{

    printf("status: %s\n", papiStatusString(PAPI_OK));
    printf("status: %s\n", papiStatusString(PAPI_DEVICE_ERROR));
    printf("status: %s\n", papiStatusString(PAPI_DOCUMENT_ACCESS_ERROR));

    exit(0);
}
```

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Volatile
MT-Level	Safe

See Also `libpapi(3LIB)`, `attributes(5)`

Name pathfind – search for named file in named directories

Synopsis

```
cc [ flag ... ] file ... -lgen [ library ... ]
#include <libgen.h>

char *pathfind(const char *path, const char *name, const char *mode);
```

Description The `pathfind()` function searches the directories named in *path* for the file *name*. The directories named in *path* are separated by colons (:). The *mode* argument is a string of option letters chosen from the set `[rwxfbcdpugks]:`

Letter	Meaning
r	readable
w	writable
x	executable
f	normal file
b	block special
c	character special
d	directory
p	FIFO (pipe)
u	set user ID bit
g	set group ID bit
k	sticky bit
s	size non-zero

Options read, write, and execute are checked relative to the real (not the effective) user ID and group ID of the current process.

If *name* begins with a slash, it is treated as an absolute path name, and *path* is ignored.

An empty *path* member is treated as the current directory. A slash (/) character is not prepended at the occurrence of the first match; rather, the unadorned *name* is returned.

Examples **EXAMPLE 1** Example of finding the `ls` command using the PATH environment variable.

To find the `ls` command using the `PATH` environment variable:

pathfind (getenv ("PATH"), "ls", "rx")

Return Values The `pathfind()` function returns a `(char *)` value containing static, thread-specific data that will be overwritten upon the next call from the same thread.

If the file *name* with all characteristics specified by *mode* is found in any of the directories specified by *path*, then `pathfind()` returns a pointer to a string containing the member of *path*, followed by a slash character (`/`), followed by *name*.

If no match is found, `pathname()` returns a null pointer, `((char *) 0)`.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `sh(1)`, `test(1)`, `access(2)`, `mknod(2)`, `stat(2)`, `getenv(3C)`, `attributes(5)`

Notes The string pointed to by the returned pointer is stored in an area that is reused on subsequent calls to `pathfind()`. The string should not be deallocated by the caller.

When compiling multithreaded applications, the `_REENTRANT` flag must be defined on the compile line. This flag should only be used in multithreadedapplications.

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 pctx_capture, pctx_create, pctx_run, pctx_release – process context library

Synopsis

```
cc [ flag... ] file... –lpctx [ library... ]
#include <libpctx.h>

typedef void (pctx_errfn_t)(const char *fn, const char *fmt, va_list ap);

pctx_t *pctx_create(const char *filename, char *const *argv, void *arg,
     int verbose, pctx_errfn_t *errfn);

pctx_t *pctx_capture(pid_t pid, void *arg, int verbose,
     pctx_errfn_t *errfn);

int pctx_run(pctx_t *pctx, uint_t sample, uint_t nsamples,
     int (*tick)(pctx *, pid_t, id_t, void *));

void pctx_release(pctx_t *pctx);
```

Description This family of functions allows a controlling process (the process that invokes them) to create or capture controlled processes. The functions allow the occurrence of various events of interest in the controlled process to cause the controlled process to be stopped, and to cause callback routines to be invoked in the controlling process.

`pctx_create()` and `pctx_capture()` There are two ways a process can be acquired by the process context functions. First, a named application can be invoked with the usual *argv*[] array using `pctx_create()`, which forks the caller and execs the application in the child. Alternatively, an existing process can be captured by its process ID using `pctx_capture()`.

Both functions accept a pointer to an opaque handle, *arg*; this is saved and treated as a caller-private handle that is passed to the other functions in the library. Both functions accept a pointer to a `printf(3C)`-like error routine *errfn*; a default version is provided if `NULL` is specified.

A freshly-created process is created stopped; similarly, a process that has been successfully captured is stopped by the act of capturing it, thereby allowing the caller to specify the handlers that should be called when various events occur in the controlled process. The set of handlers is listed on the `pctx_set_events(3CPC)` manual page.

`pctx_run()` Once the callback handlers have been set with `pctx_set_events()`, the application can be set running using `pctx_run()`. This function starts the event handling loop; it returns only when either the process has exited, the number of time samples has expired, or an error has occurred (for example, if the controlling process is not privileged, and the controlled process has exec-ed a setuid program).

Every *sample* milliseconds the process is stopped and the *tick*( ) routine is called so that, for example, the performance counters can be sampled by the caller. No periodic sampling is performed if *sample* is 0.

`pctx_release()` Once `pctx_run()` has returned, the process can be released and the underlying storage freed using `pctx_release()`. Releasing the process will either allow the controlled process to continue (in the case of an existing captured process and its children) or kill the process (if it and its children were created using `pctx_create()`).

Return Values Upon successful completion, `pctx_capture()` and `pctx_create()` return a valid handle. Otherwise, the functions print a diagnostic message and return `NULL`.

Upon successful completion, `pctx_run()` returns `0` with the controlled process either stopped or exited (if the controlled process has invoked `exit(2)`.) If an error has occurred (for example, if the controlled process has `exec`–ed a set-ID executable, if certain callbacks have returned error indications, or if the process was unable to respond to `proc(4)` requests) an error message is printed and the function returns –1.

Usage Within an event handler in the controlling process, the controlled process can be made to perform various system calls on its behalf. No system calls are directly supported in this version of the API, though system calls are executed by the `cpc_pctx` family of interfaces in `libcpc` such as `cpc_pctx_bind_event(3CPC)`. A specially created agent LWP is used to execute these system calls in the controlled process. See `proc(4)` for more details.

While executing the event handler functions, the library arranges for the signals `SIGTERM`, `SIGQUIT`, `SIGABRT`, and `SIGINT` to be blocked to reduce the likelihood of a keyboard signal killing the controlling process prematurely, thereby leaving the controlled process permanently stopped while the agent LWP is still alive inside the controlled process.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Unsafe

See Also `fork(2)`, `cpc(3CPC)`, `pctx_set_events(3CPC)`, `libpctx(3LIB)`, `proc(4)`, `attributes(5)`

Name pctx_set_events – associate callbacks with process events

Synopsis

```
cc [ flag... ] file... –lpctx [ library... ]
#include <libpctx.h>

typedef    enum {
        PCTX_NULL_EVENT = 0,
        PCTX_SYSC_EXEC_EVENT,
        PCTX_SYSC_FORK_EVENT,
        PCTX_SYSC_EXIT_EVENT,
        PCTX_SYSC_LWP_CREATE_EVENT,
        PCTX_INIT_LWP_EVENT,
        PCTX_FINI_LWP_EVENT,
        PCTX_SYSC_LWP_EXIT_EVENT
} pctx_event_t;

typedef int pctx_sysc_execfn_t(pctx_t *pctx, pid_t pid, id_t lwpid,
     char *cmd, void *arg);

typedef void pctx_sysc_forkfn_t(pctx_t *pctx,
     pid_t pid, id_t lwpid, pid_t child, void *arg);

typedef void pctx_sysc_exitfn_t(pctx_t *pctx, pid_t pid, id_t lwpid,
     void *arg);

typedef int pctx_sysc_lwp_createfn_t(pctx_t *pctx, pid_t pid, id_t lwpid,
     void *arg);

typedef int pctx_init_lwpfn_t(pctx_t *pctx, pid_t pid, id_t lwpid,
     void *arg);

typedef int pctx_fini_lwpfn_t(pctx_t *pctx, pid_t pid, id_t lwpid,
     void *arg);

typedef int pctx_sysc_lwp_exitfn_t(pctx_t *pctx, pid_t pid, id_t lwpid,
     void *arg);

int pctx_set_events(pctx_t *pctx...);
```

Description The `pctx_set_events()` function allows the caller (the controlling process) to express interest in various events in the controlled process. See `pctx_capture(3CPC)` for information about how the controlling process is able to create, capture and manipulate the controlled process.

The `pctx_set_events()` function takes a `pctx_t` handle, followed by a variable length list of pairs of `pctx_event_t` tags and their corresponding handlers, terminated by a `PCTX_NULL_EVENT` tag.

Most of the events correspond closely to various classes of system calls, though two additional pseudo-events (*init_lwp* and *fini_lwp*) are provided to allow callers to perform various housekeeping tasks. The *init_lwp* handler is called as soon as the library identifies a new LWP, while *fini_lwp* is called just before the LWP disappears. Thus the classic “hello world” program

would see an *init_lwp* event, a *fini_lwp* event and (process) *exit* event, in that order. The table below displays the interactions between the states of the controlled process and the handlers executed by users of the library.

System Calls and pctx Handlers		
System call	**Handler**	**Comments**
`exec`,`execve`	*fini_lwp*	Invoked serially on all lwps in the process.
	exec	Only invoked if the `exec()` system call succeeded.
	init_lwp	If the exec succeeds, only invoked on lwp 1. If the exec fails, invoked serially on all lwps in the process.
`fork`, `vfork`, `fork1`	*fork*	Only invoked if the `fork()` system call succeeded.
`exit`	*fini_lwp*	Invoked on all lwps in the process.
	exit	Invoked on the exiting lwp.

Each of the handlers is passed the caller's opaque handle, a `pctx_t` handle, the pid, and lwpid of the process and lwp generating the event. The *lwp_exit*, and (process) `exit` events are delivered *before* the underlying system calls begin, while the `exec`, *fork*, and *lwp_create* events are only delivered after the relevant system calls complete successfully. The `exec` handler is passed a string that describes the command being executed. Catching the *fork* event causes the calling process to `fork(2)`, then capture the child of the controlled process using `pctx_capture( )` before handing control to the *fork* handler. The process is released on return from the handler.

Return Values

Upon successful completion, `pctx_set_events( )` returns 0. Otherwise, the function returns –1.

Examples

EXAMPLE 1 HandleExec example.

This example captures an existing process whose process identifier is *pid*, and arranges to call the *HandleExec* routine when the process performs an `exec(2)`.

```
static void
HandleExec(pctx_t *pctx, pid_t pid, id_t lwpid, char *cmd, void *arg)
{
     (void) printf("pid %d execed '%s'\n", (int)pid, cmd);
}
int
main()
{
     ...
     pctx = pctx_capture(pid, NULL, 1, NULL);
     (void) pctx_set_events(pctx,
```

EXAMPLE 1 HandleExec example. *(Continued)*

```
            PCTX_SYSC_EXEC_EVENT, HandleExec,
            ...
            PCTX_NULL_EVENT);
        (void) pctx_run(pctx, 0, 0, NULL);
        pctx_release(pctx);
}
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Unsafe

See Also

exec(2), exit(2), fork(2), vfork(2), fork1(2), cpc(3CPC), libpctx(3LIB), proc(4), attributes(5)

Name picld_log – log a message in system log

Synopsis

```
cc [flag ...] file ... -lpicltree [library ...]
#include <picltree.h>

void picld_log(const char *msg);
```

Description The `picld_log()` function logs the message specified in msg to the system log file using `syslog(3C)`. This function is used by the PICL daemon and the plug-in modules to log messages to inform users of any error or warning conditions.

Return Values This function does not return a value.

Errors No errors are defined.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `syslog(3C)`, `attributes(5)`

Name picld_plugin_register – register plug-in with the daemon

Synopsis

```
cc [ flag... ] file... -lpicltree [ library... ]
#include <picltree.h>

int picld_plugin_register(picld_plugin_reg_t  *regp);
```

Description The `picld_plugin_register()` function is the function used by a plug-in module to register itself with the PICL daemon upon initialization. The plug-in provides its name and the entry points of the initialization and cleanup routines in the *regp* argument.

```
typedef struct {
    int   version;              /* PICLD_PLUGIN_VERSION */
    int   critical;             /* is plug-in critical? */
    char  *name;                /* name of the plugin module */
    void  (*plugin_init)(void); /* init/reinit function */
    void  (*plugin_fini)(void); /* fini/cleanup function */
} picld_plugin_reg_t;
```

The plug-in module also specifies whether it is a critical module for the proper system operation. The critical field in the registration information is set to `PICLD_PLUGIN_NON_CRITICAL` by plug-in modules that are not critical to system operation, and is set to `PICLD_PLUGIN_CRITICAL` by plug-in modules that are critical to the system operation. An environment control plug-in module is an example for a `PICLD_PLUGIN_CRITICAL` type of plug-in module.

The PICL daemon saves the information passed during registration in *regp* in the order in which the plug-ins registered.

Upon initialization, the PICL daemon invokes the `plugin_init()` routine of each of the registered plug-in modules in the order in which they registered. In their `plugin_init()` routines, the plug-in modules collect the platform configuration data and add it to the PICL tree using PICLTREE interfaces (3PICLTREE).

On reinitialization, the PICL daemon invokes the `plugin_fini()` routines of the registered plug-in modules in the reverse order of registration. Then, the `plugin_init()` entry points are invoked again in the order in which the plug-ins registered.

Return Values Upon successful completion, `0` is returned. On failure, a negative value is returned.

Errors

`PICL_NOTSUPPORTED`	Version not supported
`PICL_FAILURE`	General system failure

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 libpicltree(3PICLTREE), attributes(5)

Name picl_find_node – find node with given property and value

Synopsis

```
cc [ flag... ] file... -l [ library... ]
#include <picl.h>

int picl_find_node(picl_nodehdl_trooth, char *pname,
     picl_prop_type_t ptype, void *pval, size_t valsize,
     picl_nodehdl_t *retnodeh);
```

Description The `picl_find_node()` function visits the nodes in the subtree under the node specified by *rooth*. The handle of the node that has the property whose name, type, and value matches the name, type, and value specified in *pname*, *ptype*, and *pval* respectively, is returned in the location given by *retnodeh*. The *valsize* argument specifies the size of the value in *pval*. The first *valsize* number of bytes of the property value is compared with *pval*.

Return Values Upon successful completion, 0 is returned. Otherwise a non-negative integer is returned to indicate an error.

The value `PICL_NODENOTFOUND` is returned if no node that matches the property criteria can be found.

Errors

`PICL_FAILURE`	General system failure
`PICL_INVALIDHANDLE`	Invalid handle
`PICL_NODENOTFOUND`	Node not found
`PICL_NOTNODE`	Not a node
`PICL_STALEHANDLE`	Stale handle

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

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

See Also `picl_get_propinfo(3PICL)`, `picl_get_propval(3PICL)`, `picl_get_propval_by_name(3PICL)`, `picl_get_prop_by_name(3PICL)`, `attributes(5)`

Name picl_get_first_prop, picl_get_next_prop – get a property handle of a node

Synopsis

```
cc [ flag... ] file... -lpicl [ library... ]
#include <picl.h>

int picl_get_first_prop(picl_nodehdl_t nodeh,
     piclprop_hdl_t *proph);

int picl_get_next_prop(picl_prophdl_t proph,
     picl_prophdl_t *nextprop);
```

Description The `picl_get_first_prop()` function gets the handle of the first property of the node specified by *nodeh* and copies it into the location given by *proph*.

The `picl_get_next_prop()` function gets the handle of the next property after the one specified by *proph* from the property list of the node, and copies it into the location specified by *nextprop*.

If there are no more properties, this function returns `PICL_ENDOFLIST`.

Return Values Upon successful completion, `0` is returned. On failure, a non-negative integer is returned to indicate an error.

`PICL_ENDOFLIST` is returned to indicate that there are no more properties.

`PICL_STALEHANDLE` is returned if the handle is no longer valid. This occurs if the PICL tree was refreshed or reinitialized.

`PICL_INVALIDHANDLE` is returned if the specified handle never existed. This error may be returned for a previously valid handle if the daemon was brought down and restarted. When this occurs a client must revalidate any saved handles.

Errors

`PICL_NOTINITIALIZED`	Session not initialized
`PICL_NORESPONSE`	Daemon not responding
`PICL_NOTNODE`	Not a node
`PICL_NOTPROP`	Not a property
`PICL_INVALIDHANDLE`	Invalid handle
`PICL_STALEHANDLE`	Stale handle
`PICL_FAILURE`	General system failure
`PICL_ENDOFLIST`	End of list

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `picl_get_prop_by_name(3PICL)`, `attributes(5)`

Name picl_get_frutree_parent – get frutree parent node for a given device node

Synopsis

```
cc [ flag... ] file... -lpicl [ library... ]
#include <picl.h>

int picl_get_frutree_parent(picl_nodehdl_t devh,
     picl_nodehdl_t *frutreeh);
```

Description The devices under the `/platform` subtree of the PICLTREE are linked to their FRU containers represented in the `/frutree` using PICL reference properties. The `picl_get_frutree_parent()` function returns the handle of the node in the `/frutree` subtree that is the FRU parent or container of the the device specified by the node handle, *devh*. The handle is returned in the *frutreeh* argument.

Return Values Upon successful completion, 0 is returned. Otherwise a non-negative integer is returned to indicate an error.

Errors

`PICL_FAILURE`	General system failure
`PICL_INVALIDHANDLE`	Invalid handle
`PICL_PROPNOTFOUND`	Property not found
`PICL_STALEHANDLE`	Stale handle

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

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

See Also `picl_get_propinfo(3PICL)`, `picl_get_propval(3PICL)`, `picl_get_propval_by_name(3PICL)`, `picl_get_prop_by_name(3PICL)`, `attributes(5)`

Name picl_get_next_by_row, picl_get_next_by_col – access a table property

Synopsis

```
cc [ flag... ] file... -lpicl [ library... ]
#include <picl.h>

int picl_get_next_by_row(picl_prophdl_t proph,
     picl_prophdl_t *colh);

int picl_get_next_by_col(picl_prophdl_t proph,
     picl_prophdl_t *colh);
```

Description The `picl_get_next_by_row()` function copies the handle of the property that is in the next column of the table and on the same row as the property *proph*. The handle is copied into the location given by *rowh*.

The `picl_get_next_by_col()` function copies the handle of the property that is in the next row of the table and on the same column as the property *proph*. The handle is copied into the location given by *colh*.

If there are no more rows or columns, this function returns the value `PICL_ENDOFLIST`.

Return Values Upon successful completion, `0` is returned. On failure, a non-negative integer is returned to indicate an error.

`PICL_STALEHANDLE` is returned if the handle is no longer valid. This occurs if the PICL tree was refreshed or reinitialized.

`PICL_INVALIDHANDLE` is returned if the specified handle never existed. This error may be returned for a previously valid handle if the daemon was brought down and restarted. When this occurs a client must revalidate any saved handles.

Errors

`PICL_NOTINITIALIZED`	Session not initialized
`PICL_NORESPONSE`	Daemon not responding
`PICL_NOTTABLE`	Not a table
`PICL_INVALIDHANDLE`	Invalid handle
`PICL_STALEHANDLE`	Stale handle
`PICL_FAILURE`	General system failure
`PICL_ENDOFLIST`	General system failure

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also picl_get_propval(3PICL), attributes(5)

Name picl_get_node_by_path – get handle of node specified by PICL tree path

Synopsis

```
cc [ flag... ] file... -lpicl [ library... ]
#include <picl.h>

int picl_get_node_by_path(const char *piclpath,
     picl_nodehdl_t *nodeh);
```

Description The `picl_get_node_by_path()` function copies the handle of the node in the PICL tree specified by the path given in *piclpath* into the location *nodeh*.

The syntax of a PICL tree path is:

`[<def_`*propname*`>:]/[<`*def_propval*`>[<`*match_cond*`>]... ]`

where the <*def_propname*> prefix is a shorthand notation to specify the name of the property whose value is specified in <*def_propval*>, and the <*match_cond*> expression specifies the matching criteria for that node in the form of one or more pairs of property names and values such as

```
[@<address>][?<prop_name>[=<prop_val>]... ]
```

where '@' is a shorthand notation to refer to the device address or a FRU's location label and is followed by <*address*>, which gives the device address or the location label.

For nodes under the `/platform` tree, the address value is matched with the value of the property `bus-addr`, if it exists. If no `bus-addr` property exists, the address value is matched with the value of the property `UnitAddress`. To explicitly limit the comparison to `bus-addr` or `UnitAddress` property, use the '?' notation described below.

For nodes under the `/frutree` tree, the <*address*> value is matched with the value of the `Label` property.

The expression following '?' specifies matching property name and value pairs, where <*prop_name*> specifies the property name and <*prop_val*> specifies the property value for properties not of type `PICL_PTYPE_VOID`. The values for properties of type `PICL_PTYPE_TABLE`, `PICL_PTYPE_BYTEARRAY`, and `PICL_PTYPE_REFERENCE` cannot be specified in the <*match_cond*> expression.

A `_class` property value of `picl` can be used to match nodes of any `PICL` classes. The class `picl` is the base class of all the classes in `PICL`.

All valid paths must begin at the root node denoted by '/'.

If no prefix is specified for the path, the prefix defaults to the `name` property.

Return Values Upon successful completion, 0 is returned. Otherwise a non-negative integer is returned to indicate an error.

The value `PICL_NOTNODE` is returned if there is no node corresponding to the specified path.

Errors

PICL_FAILURE	General system failure
PICL_INVALIDARG	Invalid argument
PICL_NOTNODE	Not a node

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

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

See Also picl_get_propval_by_name(3PICL), attributes(5)

Name picl_get_prop_by_name – get the handle of the property by name

Synopsis

```
cc [ flag... ] file... -lpicl [ library... ]
#include <picl.h>

int picl_get_prop_by_name(picl_nodehdl_t nodeh, char *name,
     picl_prophdl_t *proph);
```

Description The `picl_get_prop_by_name()` function gets the handle of the property of node *nodeh* whose name is specified in *name*. The handle is copied into the location specified by *proph*.

Return Values Upon successful completion, `0` is returned. On failure, a non-negative integer is returned to indicate an error.

`PICL_PROPNOTFOUND` is returned if the property of the specified name does not exist.

`PICL_RESERVEDNAME` is returned if the property name specified is one of the reserved property names.

`PICL_STALEHANDLE` is returned if the handle is no longer valid. This occurs if the PICL tree was refreshed or reinitialized.

`PICL_INVALIDHANDLE` is returned if the specified handle never existed. This error may be returned for a previously valid handle if the daemon was brought down and restarted. When this occurs a client must revalidate any saved handles.

Errors

`PICL_NOTINITIALIZED`	Session not initialized
`PICL_NORESPONSE`	Daemon not responding
`PICL_NOTNODE`	Not a node
`PICL_PROPNOTFOUND`	Property not found
`PICL_RESERVEDNAME`	Reserved property name specified
`PICL_INVALIDHANDLE`	Invalid handle
`PICL_STALEHANDLE`	Stale handle
`PICL_FAILURE`	General system failure

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also attributes(5)

Name picl_get_propinfo – get the information about a property

Synopsis

```
cc [ flag... ] file... -lpicl [ library... ]
#include <picl.h>

int picl_get_propinfo(picl_prophdl_t proph,
     picl_propinfo_t *pinfo);
```

Description The `picl_get_propinfo()` function gets the information about the property specified by handle *proph* and copies it into the location specified by *pinfo*. The property information includes the property type, access mode, size, and the name of the property as described on `libpicl(3PICL)` manual page.

The maximum size of a property value is specified by `PICL_PROPSIZE_MAX`. It is currently set to 512KB.

Return Values Upon successful completion, `0` is returned. On failure, a non-negative integer is returned to indicate an error.

`PICL_STALEHANDLE` is returned if the handle is no longer valid. This occurs if the PICL tree was refreshed or reinitialized.

`PICL_INVALIDHANDLE` is returned if the specified handle never existed. This error may be returned for a previously valid handle if the daemon was brought down and restarted. When this occurs a client must revalidate any saved handles.

Errors

`PICL_NOTINITIALIZED`	Session not initialized
`PICL_NORESPONSE`	Daemon not responding
`PICL_NOTPROP`	Not a property
`PICL_INVALIDHANDLE`	Invalid handle specified
`PICL_STALEHANDLE`	Stale handle specifie
`PICL_FAILURE`	General system failure

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `libpicl(3PICL)`, `picl_get_propval(3PICL)`, `picl_get_propval_by_name(3PICL)`, `attributes(5)`

Name picl_get_propinfo_by_name – get property information and handle of named property

Synopsis

```
cc [ flag... ] file... -lpicl [library... ]
#include <picl.h>

int picl_get_propinfo_by_name(picl_nodehdl_t nodeh,
     const char *pname, picl_propinfo_t *pinfo,
     picl_prophdl_t *proph);
```

Description The `picl_get_propinfo_by_name()` function copies the property information of the property specified by *pname* in the node *nodeh* into the location given by *pinfo*. The handle of the property is returned in the location *proph*.

Return Values Upon successful completion, `0` is returned. On failure, a non-negative integer is returned to indicate an error.

`PICL_PROPNOTFOUND` is returned if the property of the specified name does not exist.

`PICL_RESERVEDNAME` is returned if the property name specified is one of the reserved property names.

`PICL_STALEHANDLE` is returned if the handle is no longer valid. This occurs if the PICL tree was refreshed or reinitialized.

`PICL_INVALIDHANDLE` is returned if the specified handle never existed. This error may be returned for a previously valid handle if the daemon was brought down and restarted. When this occurs a client must revalidate any saved handles.

Errors

`PICL_NOTINITIALIZED`	Session not initialized
`PICL_NORESPONSE`	Daemon not responding
`PICL_NOTNODE`	Not a node
`PICL_PROPNOTFOUND`	Property not found
`PICL_RESERVEDNAME`	Reserved property name specified
`PICL_INVALIDHANDLE`	Invalid handle
`PICL_STALEHANDLE`	Stale handle
`PICL_FAILURE`	General system failure

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `picl_get_propinfo`(3PICL), `picl_get_prop_by_name`(3PICL), `attributes`(5)

Name picl_get_propval, picl_get_propval_by_name – get the value of a property

Synopsis

```
cc [ flag... ] file... -lpicl [ library... ]
#include <picl.h>

int picl_get_propval(picl_prophdl_t proph, void *valbuf,
     size_t nbytes);

int picl_get_propval_by_name(picl_nodehdl_t nodeh,
     char *propname, void *valbuf, size_t nbytes);
```

Description The `picl_get_propval()` function copies the value of the property specified by the handle *proph* into the buffer location given by *valbuf*. The size of the buffer *valbuf* in bytes is specified in *nbytes*.

The `picl_get_propval_by_name()` function gets the value of property named *propname* of the node specified by handle *nodeh*. The value is copied into the buffer location given by *valbuf*. The size of the buffer *valbuf* in bytes is specified in *nbytes*.

The `picl_get_propval_by_name()` function is used to get a reserved property's value. An example of a reserved property is "`_parent`". Please refer to `libpicl(3PICL)` for a complete list of reserved property names.

Return Values Upon successful completion, `0` is returned. On failure, a non-negative integer is returned to indicate an error.

`PICL_PROPNOTFOUND` is returned if the property of the specified name does not exist.

`PICL_PERMDENIED` is returned if the client does not have sufficient permission to access the property.

`PICL_STALEHANDLE` is returned if the handle is no longer valid. This occurs if the PICL tree was refreshed or reinitialized.

`PICL_INVALIDHANDLE` is returned if the specified handle never existed. This error may be returned for a previously valid handle if the daemon was brought down and restarted. When this occurs a client must revalidate any saved handles.

Errors

`PICL_NOTINITIALIZED`	Session not initialized
`PICL_NORESPONSE`	Daemon not responding
`PICL_PERMDENIED`	Insufficient permission
`PICL_VALUETOOBIG`	Value too big for buffer
`PICL_NOTPROP`	Not a property
`PICL_PROPNOTFOUND`	Property node found
`PICL_NOTNODE`	Not a node
`PICL_INVALIDHANDLE`	Invalid handle specified

PICL_STALEHANDLE	Stale handle specified
PICL_FAILURE	General system failure

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also libpicl(3PICL), picl_get_propinfo(3PICL), attributes(5)

Name picl_get_root – get the root handle of the PICL tree

Synopsis

```
cc [ flag... ] file... -lpicl [ library... ]
#include <picl.h>

int picl_get_root(picl_nodehdl_t *nodehandle);
```

Description The `picl_get_root()` function gets the handle of the root node of the PICL tree and copies it into the location given by *nodehandle*.

Return Values Upon successful completion, `0` is returned. On failure, a non-negative integer is returned to indicate an error.

Errors

`PICL_NOTINITIALIZED`	Session not initialized
`PICL_NORESPONSE`	Daemon not responding
`PICL_FAILURE`	General system failure

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `picl_initialize(3PICL)`, `picl_shutdown(3PICL)`, `attributes(5)`

Name picl_initialize – initiate a session with the PICL daemon

Synopsis

```
cc [ flag... ] file... -lpicl [ library... ]
#include <picl.h>

int picl_initialize(void);
```

Description The `picl_initialize()` function opens the daemon door file and initiates a session with the PICL daemon running on the system.

Return Values Upon successful completion, `0` is returned. On failure, this function returns a non-negative integer, `PICL_FAILURE`.

Errors

`PICL_NOTSUPPORTED`	Version not supported
`PICL_FAILURE`	General system failure
`PICL_NORESPONSE`	Daemon not responding

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `picl_shutdown(3PICL)`, `attributes(5)`

Name picl_set_propval, picl_set_propval_by_name – set the value of a property to the specified value

Synopsis

```
cc [ flag... ] file... -lpicl [ library... ]
#include <picl.h>

int picl_set_propval(picl_prophdl_t proph, void *valbuf,
     size_t nbytes);

int picl_set_propval_by_name(picl_nodehdl_t nodeh,
     const char *pname, void *valbuf, size_t nbytes);
```

Description The `picl_set_propval()` function sets the value of the property specified by the handle *proph* to the value contained in the buffer *valbuf*. The argument *nbytes* specifies the size of the buffer *valbuf*.

The `picl_set_propval_by_name()` function sets the value of the property named *pname* of the node specified by the handle *nodeh* to the value contained in the buffer *valbuf*. The argument *nbytes* specifies the size of the buffer *valbuf*.

Return Values Upon successful completion, `0` is returned. On failure, a non-negative integer is returned to indicate an error.

`PICL_PERMDENIED` is returned if the client does not have sufficient permission to access the property.

`PICL_STALEHANDLE` is returned if the handle is no longer valid. This occurs if the PICL tree was refreshed or reinitialized.

`PICL_INVALIDHANDLE` is returned if the specified handle never existed. This error may be returned for a previously valid handle if the daemon was brought down and restarted. When this occurs a client must revalidate any saved handles.

Errors

`PICL_NOTINITIALIZED`	Session not initialized
`PICL_NORESPONSE`	Daemon not responding
`PICL_PERMDENIED`	Insufficient permission
`PICL_NOTWRITABLE`	Property is read-only
`PICL_VALUETOOBIG`	Value too big
`PICL_NOTPROP`	Not a property
`PICL_NOTNODE`	Not a node
`PICL_INVALIDHANDLE`	Invalid handle specified
`PICL_STALEHANDLE`	Stale handle specified
`PICL_FAILURE`	General system failure

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also attributes(5)

Name picl_shutdown – shutdown the session with the PICL daemon

Synopsis

```
cc [ flag... ] file... -lpicl [ library... ]
#include <picl.h>

void picl_shutdown(void);
```

Description The `picl_shutdown()` function terminates the session with the PICL daemon and frees up any resources allocated.

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

Errors

`PICL_NOTINITIALIZED`	Session not initialized
`PICL_FAILURE`	General system failure

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `picl_initialize(3PICL)`, `attributes(5)`

Name picl_strerror – get error message string

Synopsis

```
cc [flag ...] file ... -lpicl [library ...]
#include <picl.h>

char *picl_strerror(int errnum);
```

Description The `picl_strerror()` function maps the error number in *errnum* to an error message string, and returns a pointer to that string. The returned string should not be overwritten.

Return Values The `picl_strerror()` function returns `NULL` if *errnum* is out-of-range.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `libpicl(3PICL)`, `attributes(5)`

Name picl_wait – wait for PICL tree to refresh

Synopsis

```
cc [ flag... ] file... -lpicl [ library... ]
#include <picl.h>

int picl_wait(int to_secs);
```

Description The `picl_wait()` function blocks the calling thread until the PICL tree is refreshed. The *to_secs* argument specifies the timeout for the call in number of seconds. A value of −1 for *to_secs* specifies no timeout.

Return Values The `picl_wait()` function returns `0` to indicate that PICL tree has refreshed. Otherwise, a non-negative integer is returned to indicate error.

Errors

`PICL_NOTINITIALIZED`	Session not initialized
`PICL_NORESPONSE`	Daemon not responding
`PICL_TIMEDOUT`	Timed out waiting for refresh
`PICL_FAILURE`	General system failure

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `attributes(5)`

Name picl_walk_tree_by_class – walk subtree by class

Synopsis

```
cc [ flag... ] file... -lpicl [ library... ]
#include <picl.h>

int picl_walk_tree_by_class(picl_nodehdl_t rooth,
     const char *classname, void *c_args,
     int (*callback)(picl_nodehdl_t nodeh, void *c_args));
```

Description The `picl_walk_tree_by_class()` function visits all the nodes of the subtree under the node specified by *rooth*. The PICL class name of the visited node is compared with the class name specified by *classname*. If the class names match, then the callback function specified by *callback* is called with the matching node handle and the argument provided in *c_args*. If the class name specified in *classname* is `NULL`, then the callback function is invoked for all the nodes.

The return value from the callback function is used to determine whether to continue or terminate the tree walk. The callback function returns `PICL_WALK_CONTINUE` or `PICL_WALK_TERMINATE` to continue or terminate the tree walk.

Return Values Upon successful completion, `0` is returned. On failure, a non-negative integer is returned to indicate an error.

`PICL_STALEHANDLE` is returned if the handle is no longer valid. This occurs if the PICL tree was refreshed or reinitialized.

`PICL_INVALIDHANDLE` is returned if the specified handle never existed. This error may be returned for a previously valid handle if the daemon was brought down and restarted. When this occurs a client must revalidate any saved handles.

Errors

`PICL_NOTINITIALIZED`	Session not initialized
`PICL_NORESPONSE`	Daemon not responding
`PICL_NOTNODE`	Not a node
`PICL_INVALIDHANDLE`	Invalid handle specified
`PICL_STALEHANDLE`	Stale handle specified
`PICL_FAILURE`	General system failure

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also picl_get_propval_by_name(3PICL), attributes(5)

Name pool_associate, pool_create, pool_destroy, pool_dissociate, pool_info, pool_query_pool_resources – resource pool manipulation functions

Synopsis

```
cc [ flag... ] file... -lpool [ library... ]
#include <pool.h>

int pool_associate(pool_conf_t *conf,  pool_t *pool,
     pool_resource_t *resource);

pool_t *pool_create(pool_conf_t *conf, const char *name);

int pool_destroy(pool_conf_t *conf, pool_t *pool);

int pool_dissociate(pool_conf_t *conf, pool_t *pool,
     pool_resource_t *resource);

const char *pool_info(pool_conf_t *conf, pool_t *pool,
     int flags);

pool_resource_t **pool_query_pool_resources(pool_conf_t *conf,
     pool_t *pool, uint_t *nelem, pool_value_t **properties);
```

Description These functions provide mechanisms for constructing and modifying pools entries within a target pools configuration. The *conf* argument for each function refers to the target configuration to which the operation applies.

The `pool_associate()` function associates the specified resource with *pool*. A resource can be associated with multiple pools at the same time. Any resource of this type that was formerly associated with this pool is no longer associated with the pool. The new association replaces the earlier one.

The `pool_create()` function creates a new pool with the supplied name with its default properties initialized, and associated with the default resource of each type.

The `pool_destroy function()` destroys the given pool association. Associated resources are not modified.

The `pool_dissociate()` function removes the association between the given resource and pool. On successful completion, the pool is associated with the default resource of the same type.

The `pool_info()` function returns a string describing the given pool. The string is allocated with `malloc(3C)`. The caller is reponsible for freeing the returned string. If the *flags* option is non-zero, the string returned also describes the associated resources of the pool.

The `pool_query_pool_resources()` function returns a null-terminated array of resources currently associated with the pool that match the given list of properties. The return value must be freed by the caller. The *nelem* argument is set to be the length of the array returned.

Return Values Upon successful completion, `pool_create()` returns a new initialized pool. Otherwise it returns `NULL` and `pool_error(3POOL)` returns the pool-specific error value.

Upon successful completion, `pool_associate()`, `pool_destroy()`, and `pool_dissociate()` return 0. Otherwise, they return -1 and `pool_error()` returns the pool-specific error value.

Upon successful completion, `pool_info()` returns a string describing the given pool. Otherwise it returns `NULL` and `pool_error()` returns the pool-specific error value.

Upon successful completion, `pool_query_pool_resources()` returns a null-terminated array of resources. Otherwise it returns `NULL` and `pool_error()` returns the pool-specific error value.

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

`POE_BADPARAM` The supplied configuration's status is not `POF_VALID` or *name* is already in use.

`POE_SYSTEM` A system error has occurred. Check the system error code for more details.

`POE_INVALID_CONF` The pool element could not be created because the configuration would be invalid.

`POE_PUTPROP` One of the supplied properties could not be set.

The `pool_destroy()` function will fail if:

`POE_BADPARAM` The supplied configuration's status is not `POF_VALID`.

The `pool_associate()` function will fail if:

`POE_BADPARAM` The supplied configuration's status is not `POF_VALID` or the parameters are supplied from a different configuration.

`POE_SYSTEM` A system error has occurred. Check the system error code for more details.

The `pool_disassociate()` function will fail if:

`POE_BADPARAM` The supplied configuration's status is not `POF_VALID` or the parameters are supplied from a different configuration.

`POE_INVALID_CONF` No resources could be located for the supplied configuration or the supplied configuration is not valid (for example, more than one default for a resource type was found.)

`POE_SYSTEM` A system error has occurred. Check the system error code for more details.

The `pool_info()` function will fail if:

`POE_BADPARAM`	The supplied configuration's status is not `POF_VALID` or the *flags* paramter is neither 0 or 1.
`POE_INVALID_CONF`	The configuration is invalid.
`POE_SYSTEM`	A system error has occurred. Check the system error code for more details.

The `pool_query_pool_resources()` function will fail if:

`POE_BADPARAM`	The supplied configuration's status is not `POF_VALID`.
`POE_INVALID_CONF`	The configuration is invalid.
`POE_SYSTEM`	A system error has occurred. Check the system error code for more details.

Usage Pool names are unique across pools in a given configuration file. It is an error to attempt to create a pool with a name that is currently used by another pool within the same configuration.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Unstable
MT-Level	Safe

See Also `libpool(3LIB)`, `pool_error(3POOL)`, `attributes(5)`

Name pool_component_info, pool_get_owning_resource – resource pool component functions

Synopsis

```
cc [ flag... ] file... -lpool [ library... ]
#include <pool.h>

const char *pool_component_info(pool_conf_t *conf,
     pool_component_t *component, int flags);

pool_resource_t *pool_get_owning_resource(pool_conf_t *conf,
     pool_component_t *component);
```

Description Certain resources, such as processor sets, are composed of resource components. Informational and ownership attributes of resource components are made available with the `pool_component_info()` and `pool_get_owning_resource()` functions. The *conf* argument for each function refers to the target configuration to which the operation applies.

The `pool_component_info()` function returns a string describing *component*. The string is allocated with `malloc(3C)`. The caller is reponsible for freeing the returned string. The *flags* argument is ignored.

The `pool_get_owning_resource()` function returns the resource currently containing *component*. Every component is contained by a resource.

Return Values Upon successful completion, `pool_component_info()` returns a string. Otherwise it returns `NULL` and `pool_error(3POOL)` returns the pool-specific error value.

Upon successful completion, `pool_get_owning_resource()` returns the owning resource. Otherwise it returns `NULL` and `pool_error()` returns the pool-specific error value.

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

`POE_BADPARAM` The supplied configuration's status is not `POF_VALID` or the *flags* paramter is neither 0 or 1.

`POE_INVALID_CONF` The configuration is invalid.

`POE_SYSTEM` A system error has occurred. Check the system error code for more details.

The `pool_get_owning_resource()` function will fail if:

`POE_BADPARAM` The supplied configuration's status is not `POF_VALID`.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Unstable

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also libpool(3LIB), pool_error(3POOL), attributes(5)

Name pool_component_to_elem, pool_to_elem, pool_conf_to_elem, pool_resource_to_elem – resource pool element-related functions

Synopsis

```
cc [ flag... ] file... -lpool [ library... ]
#include <pool.h>

pool_elem_t *pool_component_to_elem(pool_conf_t *conf,
     pool_component_t *component);

pool_elem_t *pool_conf_to_elem(pool_conf_t *conf);

pool_elem_t *pool_resource_to_elem(pool_conf_t *conf
      pool_resource_t *resource);

pool_elem_t *pool_to_elem(pool_conf_t *conf, pool_t *pool);
```

Description A pool element, as represented by a `pool_elem_t`, is a common abstraction for any `libpool` entity that contains properties. All such types can be converted to the opaque `pool_elem_t` type using the appropriate conversion functions prototyped above. The *conf* argument for each function refers to the target configuration to which the operation applies.

Return Values Upon successful completion, these functions return a `pool_elem_t` corresponding to the argument passed in. Otherwise they return `NULL` and `pool_error(3POOL)` returns the pool-specific error value.

Errors These function will fail if:

`POE_BADPARAM` The supplied configuration's status is not `POF_VALID`.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Unstable
MT-Level	Safe

See Also `libpool(3LIB)`, `pool_error(3POOL)`, `attributes(5)`

Name pool_conf_alloc, pool_conf_close, pool_conf_commit, pool_conf_export, pool_conf_free, pool_conf_info, pool_conf_location, pool_conf_open, pool_conf_remove, pool_conf_rollback, pool_conf_status, pool_conf_update, pool_conf_validate – manipulate resource pool configurations

Synopsis

```
cc [ flag... ] file... -lpool [ library... ]
#include <pool.h>

pool_conf_t *pool_conf_alloc(void);

int pool_conf_close(pool_conf_t *conf);

int pool_conf_commit(pool_conf_t *conf, int active);

int pool_conf_export(pool_conf_t *conf, const char *location,
     pool_export_format_t format);

void pool_conf_free(pool_conf_t *conf);

char *pool_conf_info(const pool_conf_t *conf, int flags);

const char *pool_conf_location(pool_conf_t *conf);

int pool_conf_open(pool_conf_t *conf, const char *location,
     int flags);

int pool_conf_remove(pool_conf_t *conf);

int pool_conf_rollback(pool_conf_t *conf);

pool_conf_state_t pool_conf_status(const pool_conf_t *conf);

int pool_conf_update(const pool_conf_t *conf, int *changed);

int pool_conf_validate(pool_conf_t *conf,
     pool_valid_level_t level);
```

Description These functions enable the access and creation of configuration files associated with the pools facility. Since the pool configuration is an opaque type, an initial configuration is obtained with `pool_conf_alloc()` and released with `pool_conf_free()` when the configuration is no longer of interest. The *conf* argument for each function refers to the target configuration to which the operation applies.

The `pool_conf_close()` function closes the given configuration, releasing associated resources.

The `pool_conf_commit()` function commits changes made to the given `pool_conf_t` to permanent storage. If the *active* flag is non-zero, the state of the system will be configured to match that described in the supplied `pool_conf_t`. If configuring the system fails, `pool_conf_commit()` will attempt to restore the system to its previous state.

The `pool_conf_export()` function saves the given configuration to the specified location. The only currently supported value of *format* is `POX_NATIVE`, which is the format native to `libpool`, the output of which can be used as input to `pool_conf_open()`.

The `pool_conf_info()` function returns a string describing the entire configuration. The string is allocated with `malloc(3C)`. The caller is responsible for freeing the returned string. If the flags option is non-zero, the string returned also describes the sub-elements (if any) contained in the configuration.

The `pool_conf_location()` function returns the location string provided to `pool_conf_open()` for the given `pool_conf_t`.

The `pool_conf_open()` function creates a `pool_conf_t` given a location at which the configuration is stored. The valid flags are a bitmap of the following:

`PO_RDONLY` Open for reading only.

`PO_RDWR` Open read-write.

`PO_CREAT` Create a configuration at the given location if it does not exist. If it does, truncate it.

`PO_DISCO` Perform 'discovery'. This option only makes sense when used in conjunction with `PO_CREAT`, and causes the returned `pool_conf_t` to contain the resources and components currently active on the system.

The use of this flag is deprecated. `PO_CREAT` always performs discovery. If supplied, this flag is ignored.

`PO_UPDATE` Use when opening the dynamic state file, which is the configuration at `pool_dynamic_location(3POOL)`, to ensure that the contents of the dynamic state file are updated to represent the current state of the system.

The use of this flag is deprecated. The dynamic state is always current and does not require updating. If supplied, this flag is ignored.

A call to `pool_conf_open()` with the pool dynamic location and write permission will hang if the dynamic location has already been opened for writing.

The `pool_conf_remove()` function removes the configuration's permanent storage. If the configuration is still open, it is first closed.

The `pool_conf_rollback()` function restores the configuration state to that held in the configuration's permanent storage. This will either be the state last successfully committed (using `pool_conf_commit()`) or the state when the configuration was opened if there have been no successfully committed changes since then.

The `pool_conf_status()` function returns the status of a configuration, which can be one of the following values:

`POF_INVALID` The configuration is not in a suitable state for use.

`POF_VALID` The configuration is in a suitable state for use.

The `pool_conf_update()` function updates the library snapshot of kernel state. If *changed* is non-null, it is updated to identify which types of configuration elements changed during the update. To check for change, treat the *changed* value as a bitmap of possible element types.

A change is defined for the different element classes as follows:

`POU_SYSTEM` A property on the system element has been created, modified, or removed.

`POU_POOL` A property on a pool element has been created, modified, or removed. A pool has changed a resource association.

`POU_PSET` A property on a pset element has been created, modified, or removed. A pset's resource composition has changed.

`POU_CPU` A property on a CPU element has been created, modified, or removed.

The `pool_conf_validate()` function checks the validity of the contents of the given configuration. The validation can be at several (increasing) levels of strictness:

`POV_LOOSE` Performs basic internal syntax validation.

`POV_STRICT` Performs a more thorough syntax validation and internal consistency checks.

`POV_RUNTIME` Performs an estimate of whether attempting to commit the given configuration on the system would succeed or fail. It is optimistic in that a successful validation does not guarantee a subsequent commit operation will be successful; it is conservative in that a failed validation indicates that a subsequent commit operation on the current system will always fail.

Return Values Upon successful completion, `pool_conf_alloc()` returns an initialized `pool_conf_t` pointer. Otherwise it returns `NULL` and `pool_error(3POOL)` returns the pool-specific error value.

Upon successful completion, `pool_conf_close()`, `pool_conf_commit()`, `pool_conf_export()`, `pool_conf_open()`, `pool_conf_remove()`, `pool_conf_rollback()`, `pool_conf_update()`, and `pool_conf_validate()` return 0. Otherwise they return -1 and `pool_error()` returns the pool-specific error value.

The `pool_conf_status()` function returns either `POF_INVALID` or `POF_VALID`.

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

`POE_SYSTEM` There is not enough memory available to allocate the configuration. Check `errno` for the specific system error code.

`POE_INVALID_CONF` The configuration is invalid.

The `pool_conf_close()` function will fail if:

`POE_BADPARAM` The supplied configuration's status is not `POF_VALID`.

POE_SYSTEM The configuration's permanent store cannot be closed. Check errno for the specific system error code.

The pool_conf_commit() function will fail if:

POE_BADPARAM The supplied configuration's status is not POF_VALID or the active flag is non-zero and the system could not be modified.

POE_SYSTEM The permanent store could not be updated. Check errno for the specific system error code.

POE_INVALID_CONF The configuration is not valid for this system.

POE_ACCESS The configuration was not opened with the correct permissions.

POE_DATASTORE The update of the permanent store has failed and the contents could be corrupted. Check for a .bak file at the datastore location if manual recovery is required.

The pool_conf_export() function will fail if:

POE_BADPARAM The supplied configuration's status is not POF_VALID or the requested export format is not supported.

POE_DATASTORE The creation of the export file failed. A file might have been created at the specified location but the contents of the file might not be correct.

The pool_conf_info() function will fail if:

POE_BADPARAM The supplied configuration's status is not POF_VALID or *flags* is neither 0 nor 1.

POE_SYSTEM There is not enough memory available to allocate the buffer used to build the information string. Check errno for the specific system error code.

POE_INVALID_CONF The configuration is invalid.

The pool_conf_location() function will fail if:

POE_BADPARAM The supplied configuration's status is not POF_VALID.

The pool_conf_open() function will fail if:

POE_BADPARAM The supplied configuration's status is already POF_VALID.

POE_SYSTEM There is not enough memory available to store the supplied location, or the pools facility is not active. Check errno for the specific system error code.

POE_INVALID_CONF The configuration to be opened is at pool_dynamic_location(3POOL) and the configuration is not valid

for this system.

The `pool_conf_remove()` function will fail if:

`POE_BADPARAM` The supplied configuration's status is not `POF_VALID`.

`POE_SYSTEM` The configuration's permanent storage could not be removed. Check `errno` for the specific system error code.

The `pool_conf_rollback()` function will fail if:

`POE_BADPARAM` The supplied configuration's status is not `POF_VALID`.

`POE_SYSTEM` The permanent store could not be accessed. Check `errno` for the specific system error code.

The `pool_conf_update()` function will fail if:

`POE_BADPARAM` The supplied configuration's status is not `POF_VALID` or the configuration is not the dynamic configuration.

`POE_DATASTORE` The kernel snapshot cannot be correctly unpacked.

`POE_INVALID_CONF` The configuration contains uncommitted transactions.

`POE_SYSTEM` A system error occurred during snapshot retrieval and update.

The `pool_conf_validate()` function will fail if:

`POE_BADPARAM` The supplied configuration's status is not `POF_VALID`.

`POE_INVALID_CONF` The configuration is invalid.

Examples

EXAMPLE 1 Create the configuration at the specified location.

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

...

pool_conf_t *pool_conf;
pool_conf = pool_conf_alloc();
char *input_location = "/tmp/poolconf.example";

if (pool_conf_open(pool_conf, input_location,
    PO_RDONLY) < 0) {
        fprintf(stderr, "Opening pool configuration %s
                failed\n", input_location);
}
```

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Uncommitted
MT-Level	Safe

See Also libpool(3LIB), pool_error(3POOL), attributes(5)

Name pool_dynamic_location, pool_static_location, pool_version, pool_get_status, pool_set_status, pool_resource_type_list – resource pool framework functions

Synopsis

```
cc [ flag... ] file... -lpool [ library... ]
#include <pool.h>

const char *pool_dynamic_location(void);

const char *pool_static_location(void);

uint_t pool_version(uint_t ver);

int pool_get_status(int *state);

int pool_set_status(int state);

int pool_resource_type_list(const char **reslist,
     uint_t *numres);
```

Description The `pool_dynamic_location()` function returns the location used by the pools framework to store the dynamic configuration.

The `pool_static_location()` function returns the location used by the pools framework to store the default configuration used for pools framework instantiation.

The `pool_version()` function can be used to inquire about the version number of the library by specifying `POOL_VER_NONE`. The current (most capable) version is `POOL_VER_CURRENT`. The user can set the version used by the library by specifying the required version number. If this is not possible, the version returned will be `POOL_VER_NONE`.

The `pool_get_status()` function retrieves the current state of the pools facility. If state is non-null, then on successful completion the state of the pools facility is stored in the location pointed to by state.

The `pool_set_status()` function modifies the current state of the pools facility. On successful completion the state of the pools facility is changed to match the value supplied in state. Only two values are valid for state, `POOL_DISABLED` and `POOL_ENABLED`, both of which are defined in `<pool.h>`.

The `pool_resource_type_list()` function enumerates the resource types supported by the pools framework on this platform. If *numres* and *reslist* are both non-null, *reslist* points to a buffer where a list of resource types in the system is to be stored, and *numres* points to the maximum number of resource types the buffer can hold. On successful completion, the list of resource types up to the maximum buffer size is stored in the buffer pointed to by *reslist*.

Return Values The `pool_dynamic_location()` function returns the location used by the pools framework to store the dynamic configuration.

The `pool_static_location()` function returns the location used by the pools framework to store the default configuration used for pools framework instantiation.

The `pool_version()` function returns the version number of the library or `POOL_VER_NONE`.

Upon successful completion, `pool_get_status()`, `pool_set_status()`, and `pool_resource_type_list()` all return 0. Otherwise, −1 is returned and `pool_error(3POOL)` returns the pool specific error.

Errors No errors are defined for `pool_dynamic_location()`, `pool_static_location()`, and `pool_version()`.

The `pool_get_status()` function will fail if:

`POE_SYSTEM` A system error occurred when accessing the kernel pool state.

The `pool_set_status()` function will fail if:

`POE_SYSTEM` A system error occurred when modifying the kernel pool state.

The `pool_resource_type_list()` function will fail if:

`POE_BADPARAM` The *numres* parameter was `NULL`.

Examples **EXAMPLE 1** Get the static location used by the pools framework.

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

...

const char *location = pool_dynamic_location();

...

    (void) fprintf(stderr, "pool dynamic location is %s\n",
           location);
```

EXAMPLE 2 Enable the pools facility.

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

...

   if (pool_set_status(POOL_ENABLED) != 0) {
      (void) fprintf(stderr, "pools could not be enabled %s\n",
             pool_strerror(pool_error()));
             exit(2);
   }
...
```

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Unstable
MT-Level	Safe

See Also libpool(3LIB), pool_error(3POOL), attributes(5)

Name pool_error, pool_strerror – error interface to resource pools library

Synopsis

```
cc [ flag... ] file... -lpool [ library... ]
#include <pool.h>

int pool_error(void);

const char *pool_strerror(int perr);
```

Description The `pool_error()` function returns the error value of the last failure recorded by the invocation of one of the functions of the resource pool configuration library, `libpool`.

The `pool_strerror()` function returns a descriptive null-terminated string for each of the valid pool error codes.

The following error codes can be returned by `pool_error()`:

Return Values The `pool_error()` function returns the current pool error value for the calling thread from among the following:

`POE_ACCESS`	The operation could not be performed because the configuration was not opened with the correct opening permissions.
`POE_BADPARAM`	A bad parameter was supplied.
`POE_BAD_PROP_TYPE`	An incorrect property type was submitted or encountered during the pool operation.
`POE_DATASTORE`	An error occurred within permanent storage.
`POE_INVALID_CONF`	The pool configuration presented for the operation is invalid.
`POE_INVALID_SEARCH`	A query whose outcome set was empty was attempted.
`POE_NOTSUP`	An unsupported operation was attempted.
`POE_PUTPROP`	An attempt to write a read-only property was made.
`POE_OK`	The previous pool operation succeeded.
`POE_SYSTEM`	An underlying system call or library function failed; `errno`(3C) is preserved where possible.

The `pool_strerror()` function returns a pointer to the string corresponding to the requested error value. If the error value has no corresponding string, −1 is returned and `errno` is set to indicate the error.

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

`ESRCH` The specified error value is not defined by the pools error facility.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Unstable
MT-Level	Safe

See Also errno(3C), libpool(3LIB), pool_error(3POOL), attributes(5)

Name pool_get_binding, pool_set_binding, pool_get_resource_binding – set and query process to resource pool bindings

Synopsis

```
cc [ flag... ] file... -lpool [ library... ]
#include <pool.h>

char *pool_get_binding(pid_t pid);

int pool_set_binding(const char *pool, idtype_t idtype,
     id_t id);

char *pool_get_resource_binding(const char *type, pid_t pid);
```

Description The `pool_get_binding()` function returns the name of the pool on the running system that contains the set of resources to which the given process is bound. If no such pool exists on the system or the search returns more than one pool (since the set of resources is referred to by more than one pool), `NULL` is returned and the pool error value is set to `POE_INVALID_SEARCH`.

It is possible that one of the resources to which the given process is bound is not associated with a pool. This could occur if a processor set was created with one of the `pset_()` functions and the process was then bound to that set. It could also occur if the process was bound to a resource set not currently associated with a pool, since resources can exist that are not associated with a pool.

The `pool_set_binding()` function binds the processes matching *idtype* and *id* to the resources associated with *pool* on the running system. This function requires the privilege required by the underlying resource types referenced by the pool; generally, this requirement is equivalent to requiring superuser privilege.

The *idtype* parameter can be of the following types:

`P_PID` The *id* parameter is a pid.

`P_TASKID` The *id* parameter is a taskid.

`P_PROJID` The *id* parameter is a project ID. All currently running processes belonging to the given project will be bound to the pool's resources.

The `pool_get_resource_binding()` function returns the name of the resource of the supplied type to which the supplied process is bound.

The application must explicity free the memory allocated for the return values for `pool_get_binding()` and `pool_get_resource_binding()`.

Return Values Upon successful completion, `pool_get_binding()` returns the name of the pool to which the process is bound. Otherwise it returns `NULL` and `pool_error(3POOL)` returns the pool-specific error value.

Upon successful completion, `pool_set_binding()` returns `PO_SUCCESS`. Otherwise, it returns `PO_FAIL` and `pool_error()` returns the pool-specific error value.

Upon successful completion, `pool_get_resource_binding()` returns the name of the resource of the specified type to which the process is bound. Otherwise it returns `NULL` and `pool_error()` returns the pool-specific error value.

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

`POE_INVALID_CONF`	The configuration is invalid.
`POE_INVALID_SEARCH`	It is not possible to determine the binding for this target due to the overlapping nature of the pools configured for this system, or the pool could not be located.
`POE_SYSTEM`	A system error has occurred. Check the system error code for more details.

The `pool_set_binding()` function will fail if:

`POE_INVALID_SEARCH`	The pool could not be found.
`POE_INVALID_CONF`	The configuration is invalid.
`POE_SYSTEM`	A system error has occurred. Check the system error code for more details.

The `pool_get_resource_binding()` function will fail if:

`POE_INVALID_CONF`	The configuration is invalid.
`POE_INVALID_SEARCH`	The target is not bound to a resource of the specified type.
`POE_SYSTEM`	A system error has occurred. Check the system error code for more details.

Examples **EXAMPLE 1** Bind the current process to the pool named “target”.

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

...

id_t pid = getpid();

...

if (pool_set_binding("target", P_PID, pid) == PO_FAIL) \\{
        (void) fprintf(stderr, "pool binding failed (\\%d)\\B{}n",
                 pool_error());
\\}
```

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Unstable
MT-Level	Safe

See Also libpool(3LIB), pool_error(3POOL), attributes(5)

Name pool_get_pool, pool_get_resource, pool_query_components, pool_query_pools, pool_query_resources – retrieve resource pool configuration elements

Synopsis

```
cc [ flag]... file... -lpool [ library... ]
#include <pool.h>

pool_t *pool_get_pool(pool_conf_t *conf, const char *name);

pool_resource_t *pool_get_resource(pool_conf_t *conf
     const char *type, const char *name);

pool_component_t **pool_query_components(pool_conf_t *conf,
     uint_t *nelem, pool_value_t **props);

pool_t **pool_query_pools(pool_conf_t *conf, uint_t *nelem,
     pool_value_t **props);

pool_component_t **pool_query_resources(pool_conf_t *conf,
     uint_t *nelem, pool_value_t **props);
```

Description These functions provide a means for querying the contents of the specified configuration. The *conf* argument for each function refers to the target configuration to which the operation applies.

The `pool_get_pool()` function returns the pool with the given name from the provided configuration.

The `pool_get_resource()` function returns the resource with the given name and type from the provided configuration.

The `pool_query_components()` function retrieves all resource components that match the given list of properties. If the list of properties is `NULL`, all components are returned. The number of elements returned is stored in the location pointed to by *nelem*. The value returned by `pool_query_components()` is allocated with `malloc(3C)` and must be explicitly freed.

The `pool_query_pools()` function behaves similarly to `pool_query_components()` and returns the list of pools that match the given list of properties. The value returned must be freed by the caller.

The `pool_query_resources()` function similarly returns the list of resources that match the given list of properties. The return value must be freed by the caller.

Return Values The `pool_get_pool()` and `pool_get_resource()` functions return the matching pool and resource, respectively. Otherwise, they return `NULL` and `pool_error(3POOL)` returns the pool-specific error value.

The `pool_query_components()`, `pool_query_pools()`, and `pool_query_resources()` functions return a null-terminated array of components, pools, and resources, respectively. If the query was unsuccessful or there were no matches, `NULL` is returned and `pool_error()` returns the pool-specific error value.

Errors The `pool_get_pool()` will fail if:

`POE_BADPARAM` The supplied configuration's status is not `POF_VALID`.

The `pool_get_resource()` will fail if:

`POE_BADPARAM` The supplied configuration's status is not `POF_VALID`.

`POE_SYSTEM` There is not enough memory available to allocate working buffers. Check `errno` for the specific system error code.

The `pool_query_components()`, `pool_query_pools()`, and `pool_query_resources()` will fail if:

`POE_BADPARAM` The supplied configuration's status is not `POF_VALID`.

`POE_INVALID_CONF` The query generated results that were not of the correct type. The configuration is invalid.

`POE_SYSTEM` There is not enough memory available to allocate working buffers. Check `errno` for the specific system error code.

Examples **EXAMPLE 1** Retrieve the pool named "foo" from a given configuration.

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

...

pool_conf_t *conf;
pool_t *pool;

...

if ((pool = pool_get_pool(conf, "foo")) == NULL) {
        (void) fprintf(stderr, "Cannot retrieve pool named
        'foo'\\B{}n");
        ...
}
```

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Unstable
MT-Level	Safe

See Also libpool(3LIB), pool_error(3POOL), attributes(5)

Name pool_get_property, pool_put_property, pool_rm_property, pool_walk_properties – resource pool element property manipulation

Synopsis

```
cc [ flag... ] file... -lpool [ library... ]
#include <pool.h>

pool_value_class_t pool_get_property(pool_conf_t *conf,
     const pool_elem_t *elem, const char *name,
     pool_value_t *property);

int pool_put_property(pool_conf_t *conf, pool_elem_t *elem,
     const char *name, const pool_value_t *value);

int pool_rm_property(pool_conf_t *conf, pool_elem_t *elem,
     const char *name);

int pool_walk_properties(pool_conf_t *conf, pool_elem_t *elem,
     void *arg, int (*callback)(pool_conf_t *, pool_elem_t *,
     const char *, pool_value_t *, void *));
```

Description The various pool types are converted to the common pool element type (`pool_elem_t`) before property manipulation. A `pool_value_t` is an opaque type that contains a property value of one of the following types:

`POC_UINT`	unsigned 64-bit integer
`POC_INT`	signed 64-bit integer
`POC_DOUBLE`	signed double-precision floating point value
`POC_BOOL`	boolean value: 0 is false, non-zero is true
`POC_STRING`	null-terminated string of characters

The *conf* argument for each function refers to the target configuration to which the operation applies.

The `pool_get_property()` function attempts to retrieve the value of the named property from the element. If the property is not found or an error occurs, the value `POC_INVAL` is returned to indicate error. Otherwise the type of the value retrieved is returned.

The `pool_put_property()` function attempts to set the named property on the element to the specified value. Attempting to set a property that does not currently exist on the element will cause the property with the given name and value to be created on the element and will not cause an error. An attempt to overwrite an existing property with a new property of a different type is an error.

The `pool_rm_property()` function attempts to remove the named property from the element. If the property does not exist or is not removable, -1 is returned and `pool_error(3POOL)` reporst an error of `POE_PUTPROP`.

The pool_walk_properties() function invokes *callback* on all properties defined for the given element. The *callback* is called with the element itself, the name of the property, the value of the property, and the caller-provided opaque argument.

A number of special properties are reserved for internal use and cannot be set or removed. Attempting to do so will fail. These properties are documented on the libpool(3LIB) manual page.

Return Values Upon successful completion, pool_get_property() returns the type of the property. Otherwise it returns POC_INVAL and pool_error() returns the pool-specific error value.

Upon successful completion, pool_put_property(), pool_rm_property(), and pool_walk_properties() return 0. Otherwise they return −1 and pool_error() returns the pool-specific error value.

Errors The pool_get_property() function will fail if:

POE_BADPARAM The supplied configuration's status is not POF_VALID, the supplied *conf* does not contain the supplied *elem*, or the property is restricted and cannot be accessed by the library.

POE_SYSTEM A system error has occurred. Check the system error code for more details.

The pool_put_property() function will fail if:

POE_BADPARAM The supplied configuration's status is not POF_VALID, the supplied *conf* does not contain the supplied *elem*, the property name is not in the correct format, or the property already exists and the supplied type does not match the existing type.

POE_SYSTEM A system error has occurred. Check the system error code for more details.

POE_PUTPROP The property name is reserved by libpool and not available for use.

POE_INVALID_CONF The configuration is invalid.

The pool_rm_property() function will fail if:

POE_BADPARAM The supplied configuration's status is not POF_VALID, the supplied *conf* does not contain the supplied elem, or the property is reserved by libpool and cannot be removed.

POE_SYSTEM A system error has occurred. Check the system error code for more details.

POE_PUTPROP The property name is reserved by libpool and not available for use.

The pool_walk_properties() function will fail if:

POE_BADPARAM The supplied configuration's status is not POF_VALID.

POE_SYSTEM A system error has occurred. Check the system error code for more details.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Unstable
MT-Level	Safe

See Also libpool(3LIB), pool_error(3POOL), attributes(5)

Name pool_resource_create, pool_resource_destroy, pool_resource_info, pool_query_resource_components, pool_resource_transfer, pool_resource_xtransfer – resource pool resource manipulation functions

Synopsis

```
cc [ flag... ] file... -lpool [ library... ]
#include <pool.h>

pool_resource_t *pool_resource_create(pool_conf_t *conf,
     const char *type, const char *name);

int pool_resource_destroy(pool_conf_t *conf,
     pool_resource_t *resource);

const char *pool_resource_info(pool_conf_t *conf
     pool_resource_t *resource, int flags);

pool_component_t **pool_query_resource_components(
     pool_conf_t *conf, pool_resource_t *resource,
     uint_t *nelem, pool_value_t **props);

int pool_resource_transfer(pool_conf_t *conf,
     pool_resource_t *source, pool_resource_t *target,
     uint64_t size);

int pool_resource_xtransfer(pool_conf_t *conf,
     pool_resource_t *source, pool_resource_t *target,
     pool_component_t **components);
```

Description The `pool_resource_create()` function creates and returns a new resource of the given *name* and *type* in the provided configuration. If there is already a resource of the given name, the operation will fail.

The `pool_resource_destroy()` function removes the specified *resource* from its configuration file.

The `pool_resource_info()` function returns a string describing the given *resource*. The string is allocated with `malloc(3C)`. The caller is reponsible for freeing the returned string. If the *flags* argument is non-zero, the string returned also describes the components (if any) contained in the resource.

The `pool_query_resource_components()` function returns a null-terminated array of the components (if any) that comprise the given resource.

The `pool_resource_transfer()` function transfers *size* basic units from the *source* resource to the *target*. Both resources must be of the same type for the operation to succeed. Transferring component resources, such as processors, is always performed as series of `pool_resource_xtransfer()` operations, since discrete resources must be identified for transfer.

The `pool_resource_xtransfer()` function transfers the specific *components* from the *source* resource to the *target*. Both resources must be of the same type, and of a type that contains components (such as processor sets). The *components* argument is a null-terminated list of `pool_component_t`.

The *conf* argument for each function refers to the target configuration to which the operation applies.

Return Values Upon successful completion, `pool_resource_create()` returns a new `pool_resource_t` with default properties initialized. Otherwise, `NULL` is returned and `pool_error(3POOL)` returns the pool-specific error value.

Upon successful completion, `pool_resource_destroy()` returns 0. Otherwise, -1 is returned and `pool_error()` returns the pool-specific error value.

Upon successful completion, `pool_resource_info()` returns a string describing the given resource (and optionally its components). Otherwise, `NULL` is returned and `pool_error()` returns the pool-specific error value.

Upon successful completion, `pool_query_resource_components()` returns a null-terminated array of `pool_component_t *` that match the provided null-terminated property list and are contained in the given resource. Otherwise, `NULL` is returned and `pool_error()` returns the pool-specific error value.

Upon successful completion, `pool_resource_transfer()` and `pool_resource_xtransfer()` return 0. Otherwise -1 is returned and `pool_error()` returns the pool-specific error value.

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

`POE_BADPARAM` The supplied configuration's status is not `POF_VALID` or *name* is in use for this resource type.

`POE_INVALID_CONF` The resource element could not be created because the configuration would be invalid.

`POE_PUTPROP` One of the supplied properties could not be set.

`POE_SYSTEM` A system error has occurred. Check the system error code for more details.

The `pool_resource_destroy()` function will fail if:

`POE_BADPARAM` The supplied configuration's status is not `POF_VALID`.

The `pool_resource_info()` function will fail if:

`POE_BADPARAM` The supplied configuration's status is not `POF_VALID` or the *flags* paramter is neither 0 nor 1.

`POE_INVALID_CONF` The configuration is invalid.

POE_SYSTEM	A system error has occurred. Check the system error code for more details.

The pool_query_resource_components() function will fail if:

POE_BADPARAM	The supplied configuration's status is not POF_VALID.
POE_INVALID_CONF	The configuration is invalid.
POE_SYSTEM	A system error has occurred. Check the system error code for more details.

The pool_resource_transfer() function will fail if:

POE_BADPARAM	The supplied configuration's status is not POF_VALID, the two resources are not of the same type, or the transfer would cause either of the resources to exceed their min and max properties.
POE_SYSTEM	A system error has occurred. Check the system error code for more details.

The pool_resource_xtransfer() function will fail if:

POE_BADPARAM	The supplied configuration's status is not POF_VALID, the two resources are not of the same type, or the supplied resources do not belong to the source.
POE_INVALID_CONF	The transfer operation failed and the configuration may be invalid.
POE_SYSTEM	A system error has occurred. Check the system error code for more details.

Examples

EXAMPLE 1 Create a new resource of type pset named foo.

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

...

pool_conf_t *conf;
pool_resource_t *resource;
...

if ((resource = pool_resource_create(conf, "pset",
    "foo")) == NULL) {
    (void) fprintf(stderr, "Cannot create resource\\B{}n");
    ...
}
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Unstable
MT-Level	Safe

See Also libpool(3LIB), pool_error(3POOL), attributes(5)

Name pool_value_alloc, pool_value_free, pool_value_get_bool, pool_value_get_double, pool_value_get_int64, pool_value_get_name, pool_value_get_string, pool_value_get_type, pool_value_get_uint64, pool_value_set_bool, pool_value_set_double, pool_value_set_int64, pool_value_set_name, pool_value_set_string, pool_value_set_uint64 – resource pool property value manipulation functions

Synopsis

```
cc [ flag... ] file... -lpool [ library... ]
#include <pool.h>

pool_value_t *pool_value_alloc(void);

void pool_value_free(pool_value_t *value);

pool_value_class_t pool_value_get_type(
     const pool_value_t *value);

int pool_value_get_bool(const pool_value_t *value,
     uchar_t *bool);

int pool_value_get_double(const pool_value_t *value,
     double *d);

int pool_value_get_int64(const pool_value_t *value,
     int64_t *i64);

int pool_value_get_string(const pool_value_t *value,
     const char **strp);

int pool_value_get_uint64(const pool_value_t *value,
     uint64_t *ui64);

void pool_value_set_bool(const pool_value_t *value,
     uchar_t bool);

void pool_value_set_double(const pool_value_t *value,
     double d);

void pool_value_set_int64(const pool_value_t *value,
     int64_t i64);

int pool_value_set_string(const pool_value_t *value,
     const char *strp);

void pool_value_set_uint64(const pool_value_t *value,
     uint64_t ui64);

const char *pool_value_get_name(const pool_value_t *value);

int pool_value_set_name(const pool_value_t *value,
     const char *name);
```

Description A `pool_value_t` is an opaque type representing the typed value portion of a pool property. For a list of the types supported by a `pool_value_t`, see `pool_get_property(3POOL)`.

The pool_value_alloc() function allocates and returns an opaque container for a pool property value. The pool_value_free() function must be called explicitly for allocated property values.

The pool_value_get_bool(), pool_value_get_double(), pool_value_get_int64(), pool_value_get_string(), and pool_value_get_uint64() functions retrieve the value contained in the pool_value_t pointed to by *value* to the location pointed to by the second argument. If the type of the value does not match that expected by the function, an error value is returned. The string retrieved by pool_value_get_string() is freed by the library when the value is overwritten or pool_value_free() is called on the pool property value.

The pool_value_get_type() function returns the type of the data contained by a pool_value_t. If the value is unused then a type of POC_INVAL is returned.

The pool_value_set_bool(), pool_value_set_double(), pool_value_set_int64(), pool_value_set_string(), and pool_value_set_uint64() functions set the value and type of the property value to the provided values. The pool_value_set_string() function copies the string passed in and returns -1 if the memory allocation fails.

Property values can optionally have names. These names are used to describe properties as name=value pairs in the various query functions (see pool_query_resources(3POOL)). A copy of the string passed to pool_value_set_name() is made by the library, and the value returned by pool_value_get_name() is freed when the pool_value_t is deallocated or overwritten.

Return Values Upon successful completion, pool_value_alloc() returns a pool property value with type initialized to PVC_INVAL. Otherwise, NULL is returned and pool_error() returns the pool-specific error value.

Upon successful completion, pool_value_get_type() returns the type contained in the property value passed in as an argument. Otherwise, POC_INVAL is returned and pool_error() returns the pool-specific error value.

Upon successful completion, pool_value_get_bool(), pool_value_get_double(), pool_value_get_int64(), pool_value_get_string(), and pool_value_get_uint64() return 0. Otherwise -1 is returned and pool_error(3POOL) returns the pool-specific error value.

Upon successful completion, pool_value_set_string() and pool_value_set_name() return 0. If the memory allocation failed, -1 is returned and pool_error() returns the pool-specific error value.

Errors The pool_value_alloc() function will fail if:

POE_SYSTEM A system error has occurred. Check the system error code for more details.

The pool_value_get_bool(), pool_value_get_double(), pool_value_get_int64(), pool_value_get_string(), and pool_value_get_uint64() functions will fail if:

POE_BADPARAM The supplied *value* does not match the type of the requested operation.

The `pool_value_set_string()` function will fail if:

POE_SYSTEM A system error has occurred. Check the system error code for more details.

The `pool_value_set_name()` function will fail if:

POE_SYSTEM A system error has occurred. Check the system error code for more details.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled
Interface Stability	Unstable
MT-Level	Safe

See Also `libpool(3LIB)`, `pool_error(3POOL)`, `attributes(5)`

REFERENCE

Extended Library Functions - Part 5

Name pool_walk_components, pool_walk_pools, pool_walk_resources – walk objects within resource pool configurations

Synopsis

```
cc [ flag... ] file... -lpool [ library... ]
#include <pool.h>

int pool_walk_components(pool_conf_t *conf,
     pool_resource_t *resource, void *arg,
     int (*callback)(pool_conf_t *, pool_resource_t *, void *));

int pool_walk_pools(pool_conf_t *conf, void *arg,
     int (*callback)(pool_conf_t *, pool_component_t *, void *));

int pool_walk_resources(pool_conf_t *conf, pool_t *pool,
     void *arg, int (*callback)(pool_conf_t *,
     pool_component_t *, void *));
```

Description The walker functions provided with `libpool(3LIB)` visit each associated entity of the given type, and call the caller-provided *callback* function with a user-provided additional opaque argument. There is no implied order of visiting nodes in the walk. If the *callback* function returns a non-zero value at any of the nodes, the walk is terminated, and an error value of -1 returned. The *conf* argument for each function refers to the target configuration to which the operation applies.

The `pool_walk_components()` function invokes *callback* on all components contained in the resource.

The `pool_walk_pools()` function invokes *callback* on all pools defined in the configuration.

The `pool_walk_resources()` function invokes *callback* function on all resources associated with *pool*.

Return Values Upon successful completion of the walk, these functions return 0. Otherwise -1 is returned and `pool_error(3POOL)` returns the pool-specific error value.

Errors These functions will fail if:

`POE_BADPARAM`	The supplied configuration's status is not `POF_VALID`.
`POE_INVALID_CONF`	The configuration is invalid.
`POE_SYSTEM`	A system error has occurred. Check the system error code for more details.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
CSI	Enabled

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Unstable
MT-Level	Safe

See Also libpool(3LIB), pool_error(3POOL), attributes(5)

Name pow, powf, powl – power function

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double pow(double x, double y);

float powf(float x, float y);

long double powl(long double x, long double y);

cc [ flag... ] file... -lm [ library... ]
#include <math.h>

double pow(double x, double y);

float powf(float x, float y);

long double powl(long double x, long double y);
```

Description These functions compute the value of x raised to the power y, x^y>. If x is negative, y must be an integer value.

Return Values Upon successful completion, these functions return the value of x raised to the power y.

For finite values of $x < 0$, and finite non-integer values of y, a domain error occurs and either a NaN (if representable), or an implementation-defined value is returned.

If the correct value would cause overflow, a range error occurs and `pow()`, `powf()`, and `powl()` return `HUGE_VAL`, `HUGE_VALF`, and `HUGE_VALL`, respectively.

If x or y is a NaN, a NaN is returned unless:

- If x is +1 and y is NaN and the application was compiled with the `c99` compiler driver and is therefore SUSv3–conforming (see `standards(5)`), 1.0 is returned.
- For any value of x (including NaN), if y is +0, 1.0 is returned.

For any odd integer value of $y > 0$, if x is ±0, ±0 is returned.

For $y > 0$ and not an odd integer, if x is ±0, +0 is returned.

If x is ±1 and y is ±Inf, and the application was compiled with the `cc` compiler driver, NaN is returned. If, however, the application was compiled with the `c99` compiler driver and is therefore SUSv3-conforming (see`standards(5)`), 1.0 is returned.

For $|x| < 1$, if y is –Inf, +Inf is returned.

For $|x| > 1$, if y is –Inf, +0 is returned.

For $|x| < 1$, if y is +Inf, +0 is returned.

For $|x| > 1$, if y is +Inf, +Inf is returned.

For *y* an odd integer < 0, if *x* is –Inf, –0 is returned.

For *y* < 0 and not an odd integer, if *x* is –Inf, +0 is returned.

For *y* an odd integer > 0, if *x* is –Inf, –Inf is returned.

For *y* > 0 and not an odd integer, if *x* is –Inf, +Inf is returned.

For *y* < 0, if *x* is +Inf, +0 is returned.

For *y* > 0, if *x* is +Inf, +Inf is returned.

For *y* an odd integer < 0, if *x* is ±0, a pole error occurs and ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL are returned for pow(), powf(), and powl(), respectively.

For *y* < 0 and not an odd integer, if *x* is ±0, a pole error occurs and HUGE_VAL, HUGE_VALF, and HUGE_VALL are returned for pow(), powf(), and powl(), respectively.

For exceptional cases, matherr(3M) tabulates the values to be returned by pow() as specified by SVID3 and XPG3.

Errors These functions will fail if:

Domain Error — The value of *x* is negative and *y* is a finite non-integer.

If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, the invalid floating-point exception is raised.

The pow() function sets errno to EDOM if the value of *x* is negative and *y* is non-integral.

Pole Error — The value of *x* is 0 and *y* is negative.

If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, the divide-by-zero floating-point exception is raised.

Range Error — The result overflows.

If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, the overflow floating-point exception is raised.

The pow() function sets errno to EDOM if the value to be returned would cause overflow.

Usage An application wanting to check for exceptions should call feclearexcept(FE_ALL_EXCEPT) before calling these functions. On return, if fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an exception has been raised. An application should either examine the return value or check the floating point exception flags to detect exceptions.

An application can also set `errno` to 0 before calling `pow()`. On return, if `errno` is non-zero, an error has occurred. The `powf()` and `powl()` functions do not set `errno`.

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

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

See Also `exp(3M)`, `feclearexcept(3M)`, `fetestexcept(3M)`, `isnan(3M)`, `math.h(3HEAD)`, `matherr(3M)`, `attributes(5)`, `standards(5)`

Notes Prior to Solaris 2.6, there was a conflict between the `pow()` function in this library and the `pow()` function in the `libmp` library. This conflict was resolved by prepending `mp_` to all functions in the `libmp` library. See `mp(3MP)` for more information.

Name printDmiAttributeValues – print data in input DmiAttributeValues list

Synopsis

```
cc [ flag ... ] file ... -ldmi -lnsl -lrwtool [ library ... ]
#include <dmi/util.hh>

void printDmiAttributeValues(DmiAttributeValues_t *values);
```

Description The `printDmiAttributeValues()` function prints the data in the input `DmiAttributeValues` list. The function prints "unknown data" for those *values* that contain invalid data.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-level	MT-Safe

See Also `libdmi(3LIB)`, `attributes(5)`

Name printDmiDataUnion – print data in input data union

Synopsis

```
cc [ flag ... ] file ... -ldmi -lnsl -lrwtool [ library ... ]
#include <dmi/util.hh>

void printDmiDataUnion(DmiDataUnion_t *data);
```

Description The `printDmiDataUnion()` function prints the data in the input data union. The output depends on the type of DMI data in the union.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-level	MT-Safe

See Also `libdmi(3LIB)`, `attributes(5)`

Name printDmiString – print a DmiString

Synopsis

```
cc [ flag ... ] file ... -ldmi -lnsl -lrwtool [ library ... ]
#include <dmi/util.hh>

void printDmiString(DmiString_t *dstr);
```

Description The `printDmiString()` function prints a `DmiString`.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-level	MT-Safe

See Also `newDmiString(3DMI)`, `libdmi(3LIB)`, `attributes(5)`

Name Privilege – Perl interface to Privileges

Synopsis

```
use Sun::Solaris::Privilege qw(:ALL);
```

Description This module provides wrappers for the Privilege-related system and library calls. Also provided are constants from the various Privilege-related headers and dynamically-generated constants for all the privileges and privilege sets.

Constants `PRIV_STR_SHORT`, `PRIV_STR_LIT`, `PRIV_STR_PORT`, `PRIV_ON`, `PRIV_OFF`, `PRIV_SET`, `PRIV_AWARE`, and `PRIV_DEBUG`.

Functions

`getppriv($which)`	This function returns the process privilege set specified by `$which`.
`setppriv($op, $which, $set)`	This function modified the privilege set specified by $which in the as specified by the `$op` and `$set` arguments. If `$op` is `PRIV_ON`, the privileges in `$set` are added to the set specified. If `$op` is `PRIV_OFF`, the privileges in `$set` are removed from the set specified. If `$op` is `PRIV_SET`, the specified set is made equal to `$set`.
`getpflags($flag)`	This function returns the value associated with process `$flag` or `undef` on error. Possible values for `$flag` are `PRIV_AWARE` and `PRIV_DEBUG`.
`setppflags($flag, $val)`	This function sets the process flag `$flag` to `$val`.
`priv_fillset()`	This function returns a new privilege set with all privileges set.
`priv_emptyset()`	This function returns a new empty privilege set.
`priv_isemptyset($set)`	This function returns whether or not `$set` is empty.
`priv_isfullset($set)`	This function returns whether or not `$set` is full.
`priv_isequalset($a, $b)`	This function returns whether sets `$a` and `$b` are equal.
`priv_issubset($a, $b)`	This function returns whether set `$a` is a subset of `$b`.
`priv_ismember($set, $priv)`	This function returns whether `$priv` is a member of `$set`.
`priv_ineffect($priv)`	This function returned whether `$priv` is in the process's effective set.
`priv_intersect($a, $b)`	This function returns a new privilege set which is the intersection of `$a` and `$b`.
`priv_union($a, $b)`	This function returns a new privilege set which is the union of `$a` and `$b`.

`priv_inverse($a)`	This function returns a new privilege set which is the inverse of `$a`.
`priv_addset($set, $priv)`	This functon adds the privilege `$priv` to `$set`.
`priv_copyset($a)`	This function returns a copy of the privilege set `$a`.
`priv_delset($set, $priv)`	This function remove the privilege `$priv` from `$set`.

Class methods None.

Object methods None.

Exports By default nothing is exported from this module. The following tags can be used to selectively import constants and functions defined in this module:

`:SYSCALLS`	`getppriv(), setppriv()`
`:LIBCALLS`	`priv_addset(), priv_copyset(), priv_delset(), priv_emptyset(), priv_fillset(), priv_intersect(), priv_inverse(), priv_isemptyset(), priv_isequalset(), priv_isfullset(), priv_ismember(), priv_issubset(), priv_gettext(), priv_union(), priv_set_to_str(), priv_str_to_set()`
`:CONSTANTS`	`PRIV_STR_SHORT, PRIV_STR_LIT, PRIV_STR_PORT, PRIV_ON, PRIV_OFF, PRIV_SET, PRIV_AWARE, PRIV_DEBUG`, plus constants for all privileges and privilege sets.
`:VARIABLES`	`%PRIVILEGES, %PRIVSETS`
`:ALL`	`:SYSCALLS, :LIBCALLS, :CONSTANTS, :VARIABLES`

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWpl5u
Interface Stability	Evolving

See Also

`getpflags(2)`, `getppriv(2)`, `priv_addset(3C)`, `priv_set(3C)`, `priv_str_to_set(3C)`, `attributes(5)`, `privileges(5)`

Name Project – Perl interface to Projects

Synopsis

```
use Sun::Solaris::Project qw(:ALL);
my $projid = getprojid();
```

Description This module provides wrappers for the Project-related system calls and the libproject(3LIB) library. Also provided are constants from the various Project-related headers.

Constants MAXPROJID, PROJNAME_MAX, PROJF_PATH, PROJECT_BUFSZ, SETPROJ_ERR_TASK, and SETPROJ_ERR_POOL.

Functions

`getprojid()`

This function returns the numeric project ID of the calling process or `undef` if the underlying getprojid(2) system call is unsuccessful.

`setproject($project, $user, $flags)`

If `$user` is a member of the project specified by `$project`, `setproject()` creates a new task and associates the appropriate resource controls with the process, task, and project. This function returns 0 on success. If the underlying task creation fails, `SETPROJ_ERR_TASK` is returned. If pool assignment fails, `SETPROJ_ERR_POOL` is returned. If any resource attribute assignments fail, an integer value corresponding to the offset of the failed attribute assignment in the project database is returned. See setproject(3PROJECT).

`activeprojects()`

This function returns a list of the currently active projects on the system. Each value in the list is the numeric ID of a currently active project.

`getprojent()`

This function returns the next entry from the project database. When called in a scalar context, `getprojent()` returns only the name of the project. When called in a list context, `getprojent()` returns a 6-element list consisting of:

```
($name, $projid, $comment, \@users, \@groups, $attr)
```

\@users and \@groups are returned as arrays containing the appropriate user or project lists. On end-of-file `undef` is returned.

`setprojent()`

This function rewinds the project database to the beginning of the file.

`endprojent()`	This function closes the project database.
`getprojbyname($name)`	This function searches the project database for an entry with the specified nam. It returns a 6-element list as returned by `getprojent()` if the entry is found and `undef` if it cannot be found.
`getprojbyid($id)`	This function searches the project database for an entry with the specified ID. It returns a 6-element list as returned by `getprojent()` if the entry is found or `undef` if it cannot be found.
`getdefaultproj($user)`	This function returns the default project entry for the specified user in the same format as `getprojent()`. It returns `undef` if the user cannot be found. See `getdefaultproj`(3PROJECT) for information about the lookup process.
`fgetprojent($filehandle)`	This function returns the next project entry from `$filehandle`, a Perl file handle that must refer to a previously opened file in `project`(4) format. Return values are the same as for `getprojent()`.
`inproj($user, $project)`	This function checks whether the specified user is able to use the project. This function returns true if the user can use the project and false otherwise. See `inproj`(3PROJECT).
`getprojidbyname($project)`	This function searches the project database for the specified project. It returns the project ID if the project is found and `undef` if it is not found.

Class methods None.

Object methods None.

Exports By default nothing is exported from this module. The following tags can be used to selectively import constants and functions defined in this module:

`:SYSCALLS`	`getprojid()`
`:LIBCALLS`	`setproject()`, `activeprojects()`, `getprojent()`, `setprojent()`, `endprojent()`, `getprojbyname()`, `getprojbyid()`, `getdefaultproj()`, `fgetprojent()`, `inproj()`, and `getprojidbyname()`

:CONSTANTS	MAXPROJID, PROJNAME_MAX, PROJF_PATH, PROJECT_BUFSZ, SETPROJ_ERR_TASK, and SETPROJ_ERR_POOL
:ALL	:SYSCALLS, :LIBCALLS, and :CONSTANTS

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWpl5u
Interface Stability	Evolving

See Also getprojid(2), getdefaultproj(3PROJECT), inproj(3PROJECT), libproject(3LIB), setproject(3PROJECT), project(4), attributes(5)

Name project_walk – visit active project IDs on current system

Synopsis

```
cc [ flag ... ] file... -lproject [ library ... ]
#include <project.h>

int project_walk(int (*callback)(const projid_t project,
     void *walk_data), void *init_data);
```

Description The `project_walk()` function provides a mechanism for the application author to examine all active projects on the current system. The *callback* function provided by the application is given the ID of an active project at each invocation and can use the *walk_data* to record its own state. The callback function should return non-zero if it encounters an error condition or attempts to terminate the walk prematurely; otherwise the callback function should return 0.

Return Values Upon successful completion, `project_walk()` returns 0. It returns −1 if the *callback* function returned a non-zero value or if the walk encountered an error, in which case `errno` is set to indicate the error.

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

`ENOMEM` There is insufficient memory available to set up the initial data for the walk.

Other returned error values are presumably caused by the *callback* function.

Examples **EXAMPLE 1** Count the number of projects available on the system.

The following example counts the number of projects available on the system.

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

typedef struct wdata {
     uint_t count;
} wdata_t;

wdata_t total_count;

int
simple_callback(const projid_t p, void *pvt)
{
     wdata_t *w = (wdata_t *)pvt;
     w->count++;
     return (0);
}

...

total_count.count = 0;
errno = 0;
```

EXAMPLE 1 Count the number of projects available on the system. *(Continued)*

```
if ((n = project_walk(simple_callback, &total_count)) >= 0)
     (void) printf("count = %u\n", total_count.count);
```

Attributes

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

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

See Also

getprojid(2), libproject(3LIB), settaskid(2), attributes(5)

Name ptree_add_node, ptree_delete_node – add or delete node to or from tree

Synopsis

```
cc [ flag... ] file... -lpicltree [ library... ]
#include <picltree.h>

int ptree_add_node(picl_nodehdl_t parh, picl_nodehdl_t chdh);

int ptree_delete_node(ptree_delete_node nodeh);
```

Description The `ptree_add_node()` function adds the node specified by handle *chdh* as a child node to the node specified by the handle *parh*. `PICL_CANTPARENT` is if the child node already has a parent.

The `ptree_delete_node()` function deletes the node specified by handle *nodeh* and all its descendant nodes from the tree.

Return Values Upon successful completion, `0` is returned. On failure, a non-negative integer is returned to indicate an error.

`PICL_STALEHANDLE` is returned if the handle is no longer valid. This occurs if the PICL tree was refreshed or reinitialized.

`PICL_INVALIDHANDLE` is returned if the specified handle never existed.

Errors

`PICL_NOTNODE`	Node a node
`PICL_CANTPARENT`	Already has a parent
`PICL_TREEBUSY`	PICL tree is busy
`PICL_INVALIDHANDLE`	Invalid handle
`PICL_STALEHANDLE`	Stale handle
`PICL_FAILURE`	General system failure

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

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

See Also `attributes(5)`

Name ptree_add_prop, ptree_delete_prop – add or delete a property

Synopsis

```
cc [ flag... ] file... -lpicltree [ library... ]
#include <picltree.h>

int ptree_add_prop(picl_nodehdl_t nodeh, picl_prophdl_t proph);

int ptree_delete_prop(picl_prophdl_t proph);
```

Description The `ptree_add_prop()` function adds the property specified by the handle *proph* to the list of properties of the node specified by handle *nodeh*.

The `ptree_delete_prop()` function deletes the property from the property list of the node. For a table property, the entire table is deleted.

Return Values Upon successful completion, 0 is returned. On failure, a non-negative integer is returned to indicate an error.

`PICL_STALEHANDLE` is returned if the handle is no longer valid. This occurs if the PICL tree was refreshed or reinitialized.

`PICL_INVALIDHANDLE` is returned if the specified handle never existed.

Errors

`PICL_NOTTABLE`	Not a table
`PICL_NOTPROP`	Not a property
`PICL_INVALIDHANDLE`	Invalid handle
`PICL_STALEHANDLE`	Stale handle
`PICL_PROPEXISTS`	Property already exists
`PICL_FAILURE`	General system failure

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

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

See Also `ptree_create_prop(3PICLTREE)`, `attributes(5)`

Name ptree_create_and_add_node – create and add node to tree and return node handle

Synopsis

```
cc [ flag... ] file... -lpicltree [ library... ]
#include <picltree.h>

int ptree_create_and_add_node(picl_nodehdl_t parh,
     const char *name, const char *classname,
     picl_nodehdl_t *nodeh);
```

Description The `ptree_create_and_add_node()` function creates a node with the name and PICL class specified by *name* and *classname* respectively. It then adds the node as a a child to the node specified by *parh*. The handle of the new node is returned in *nodeh*.

Return Values Upon successful completion, `0` is returned. On failure, a non-negative integer is returned to indicate an error.

`PICL_STALEHANDLE` is returned if the handle is no longer valid. This occurs if the PICL tree was refreshed or reinitialized.

`PICL_INVALIDHANDLE` is returned if the specified handle never existed.

Errors

`PICL_INVALIDARG`	Invalid argument
`PICL_VALUETOOBIG`	Value exceeds maximum size
`PICL_NOTSUPPORTED`	Property version not supported
`PICL_CANTDESTROY`	Attempting to destroy before delete
`PICL_NOTNODE`	Not a node
`PICL_INVALIDHANDLE`	Invalid handle
`PICL_STALEHANDLE`	Stale handle
`PICL_FAILURE`	General system failure

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

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

See Also `ptree_create_node(3PICLTREE)`, `ptree_add_node(3PICLTREE)`, `attributes(5)`

Name ptree_create_and_add_prop – create and add property to node and return property handle

Synopsis

```
cc [ flag... ] file... -lpicltree [ library... ]
#include <picltree.h>

int ptree_create_and_add_prop(picl_nodehdl_t nodeh,
     ptree_propinfo_t *infop, void *vbuf, picl_prophdl_t *proph);
```

Description The `ptree_create_and_add_prop()` function creates a property using the property information specified in *infop* and the value buffer *vbuf* and adds the property to the node specified by *nodeh*. If *proph* is not `NULL`, the handle of the property added to the node is returned in *proph*.

Return Values Upon successful completion, `0` is returned. On failure, a non-negative integer is returned to indicate an error.

`PICL_STALEHANDLE` is returned if the handle is no longer valid. This occurs if the PICL tree was refreshed or reinitialized.

`PICL_INVALIDHANDLE` is returned if the specified handle never existed.

Errors

`PICL_NOTSUPPORTED`	Property version not supported
`PICL_VALUETOOBIG`	Value exceeds maximum size
`PICL_NOTPROP`	Not a property
`PICL_NOTTABLE`	Not a table
`PICL_PROPEXISTS`	Property already exists
`PICL_RESERVEDNAME`	Property name is reserved
`PICL_INVREFERENCE`	Invalid reference property value
`PICL_INVALIDHANDLE`	Invalid handle
`PICL_STALEHANDLE`	Stale handle
`PICL_FAILURE`	General system failure

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

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

See Also ptree_create_prop(3PICLTREE), ptree_add_prop(3PICLTREE), attributes(5)

Name ptree_create_node, ptree_destroy_node – create or destroy a node

Synopsis

```
cc [ flag... ] file... -lpicltree [ library... ]
#include <picltree.h>

int ptree_create_node(char *name, char *clname,
     picl_nodehdl_t *nodeh);

int ptree_destroy_node(picl_nodehdl_t nodeh);
```

Description The `ptree_create_node()` function creates a node and sets the "name" property value to the string specified in *name* and the "class" property value to the string specified in *clname*. The handle of the new node is copied into the location given by *nodeh*.

The `ptree_destroy_node()` function destroys the node specified by *nodeh* and frees up any allocated space. The node to be destroyed must have been previously deleted by ptree_delete_node (see `ptree_add_node(3PICLTREE)`). Otherwise, `PICL_CANTDESTROY` is returned.

Return Values Upon successful completion, `0` is returned. On failure, a non-negative integer is returned to indicate an error.

`PICL_STALEHANDLE` is returned if the handle is no longer valid. This occurs if the PICL tree was refreshed or reinitialized.

`PICL_INVALIDHANDLE` is returned if the specified handle never existed.

Errors

`PICL_INVALIDARG`	Invalid argument
`PICL_VALUETOOBIG`	Value exceeds maximum size
`PICL_NOTSUPPORTED`	Property version not supported
`PICL_CANTDESTROY`	Attempting to destroy before delete
`PICL_TREEBUSY`	PICL tree is busy
`PICL_NOTNODE`	Not a node
`PICL_INVALIDHANDLE`	Invalid handle
`PICL_STALEHANDLE`	Stale handle
`PICL_FAILURE`	General system failure

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

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

See Also ptree_add_node(3PICLTREE), attributes(5)

Name ptree_create_prop, ptree_destroy_prop – create or destroy a property

Synopsis

```
cc [ flag... ] file... -lpicltree [ library... ]
#include <picltree.h>

int ptree_create_prop(ptree_propinfo_t *pinfo, void *valbuf,
     picl_prophdl_t *proph);

int ptree_destroy_prop(picl_prophdl_t proph);
```

Description The `ptree_create_prop()` function creates a property using the information specified in *pinfo*, which includes the name, type, access mode, and size of the property, as well as the read access function for a volatile property. The value of the property is specified in the buffer *valbuf*, which may be `NULL` for volatile properties. The handle of the property created is copied into the location given by *proph*. See `libpicltree(3PICLTREE)` for more information on the structure of `ptree_propinfo_t` structure.

The `ptree_destroy_prop()` function destroys the property specified by the handle *proph*. For a table property, the entire table is destroyed. The property to be destroyed must have been previously deleted.

Return Values Upon successful completion, `0` is returned. On failure, a non-negative integer is returned to indicate an error.

`PICL_STALEHANDLE` is returned if the handle is no longer valid. This occurs if the PICL tree was refreshed or reinitialized.

`PICL_INVALIDHANDLE` is returned if the specified handle never existed.

Errors

`PICL_NOTSUPPORTED`	Property version not supported
`PICL_VALUETOOBIG`	Value exceeds maximum size
`PICL_NOTPROP`	Not a property
`PICL_CANTDESTROY`	Attempting to destroy before delete
`PICL_RESERVEDNAME`	Property name is reserved
`PICL_INVREFERENCE`	Invalid reference property value
`PICL_INVALIDHANDLE`	Invalid handle
`PICL_STALEHANDLE`	Stale handle
`PICL_FAILURE`	General system failure

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

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

See Also libpicltree(3PICLTREE), ptree_add_prop(3PICLTREE), attributes(5)

Name ptree_create_table, ptree_add_row_to_table – create a table object

Synopsis

```
cc [ flag... ] file... -lpicltree [ library... ]
#include <picltree.h>

int ptree_create_table(picl_prophdl_t *tbl_hdl);

int ptree_add_row_to_table(picl_prophdl_t tbl_hdl, int nprops,
     picl_prophdl_t *proph);
```

Description The `ptree_create_table()` function creates a table object and returns the handle of the table in *tbl_hdl*.

The `ptree_add_row_to_table()` function adds a row of properties to the table specified by *tbl_hdl*. The handles of the properties of the row are specified in the *proph* array and *nprops* specifies the number of handles in the array. The number of columns in the table is determined from the first row added to the table. If extra column values are specified in subsequent rows, they are ignored. The row is appended to the end of the table.

Return Values Upon successful completion, `0` is returned. On failure, a non-negative integer is returned to indicate an error.

`PICL_STALEHANDLE` is returned if the handle is no longer valid. This occurs if the PICL tree was refreshed or reinitialized.

`PICL_INVALIDHANDLE` is returned if the specified handle never existed.

Errors

`PICL_INVALIDARG`	Invalid argument
`PICL_NOTPROP`	Not a property
`PICL_NOTTABLE`	Not a table
`PICL_INVALIDHANDLE`	Invalid handle
`PICL_STALEHANDLE`	Stale handle
`PICL_FAILURE`	General system failure

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

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

See Also `attributes(5)`

Name ptree_find_node – find node with given property and value

Synopsis

```
cc [ flag... ] file... -lpicltree [library... ]
#include <picltree.h>

int ptree_find_node(picl_nodehdl_t rooth, char *pname,
     picl_prop_type_t ptype, void *pval, size_t valsize,
     picl_nodehdl_t *retnodeh);
```

Description The `ptree_find_node()` function visits the nodes in the subtree under the node specified by *rooth*. The handle of the node that has the property whose name, type, and value matches the name, type, and value specified in *pname*, *ptype*, and *pval* respectively, is returned in the location given by *retnodeh*. The argument *valsize* gives the size of the value in *pval*. The first *valsize* number of bytes of the property value is compared with *pval*.

Return Values Upon successful completion, `0` is returned. On failure, a non-negative integer is returned to indicate an error.

`PICL_NODENOTFOUND` is returned if there is no node that matches the property criteria can be found.

Errors

`PICL_NOTNODE`	Not a node
`PICL_INVALIDHANDLE`	Invalid handle
`PICL_STALEHANDLE`	Stale handle
`PICL_PROPNOTFOUND`	Property not found
`PICL_FAILURE`	General system failure

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

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

See Also `ptree_get_prop_by_name(3PICLTREE)`, `ptree_get_propinfo(3PICLTREE)`, `ptree_get_propval(3PICLTREE)`, `ptree_get_propval_by_name(3PICLTREE)`, `attributes(5)`

Name ptree_get_first_prop, ptree_get_next_prop – get a property handle of the node

Synopsis

```
cc [ flag... ] file... -lpicltree [ library... ]
#include <picltree.h>

int ptree_get_first_prop(picl_nodehdl_t nodeh,
     picl_prophdl_t *proph);

int ptree_get_next_prop(picl_prophdl_t proph,
     picl_prophdl_t *nextproph);
```

Description The `ptree_get_first_prop()` function gets the handle of the first property of the node specified by *nodeh* and copies it into the location specified by *proph*.

The `ptree_get_next_prop()` function gets the handle of the next property after the one specified by *proph* from the list of properties of the node and copies it into the location specified by *nextproph*.

Return Values Upon successful completion, `0` is returned. On failure, a non-negative integer is returned to indicate an error.

`PICL_STALEHANDLE` is returned if the handle is no longer valid. This occurs if the PICL tree was refreshed or reinitialized.

`PICL_INVALIDHANDLE` is returned if the specified handle never existed.

Errors

`PICL_NOTPROP`	Not a property
`PICL_NOTNODE`	Not a node
`PICL_ENDOFLIST`	End of list
`PICL_INVALIDHANDLE`	Invalid handle
`PICL_STALEHANDLE`	Stale handle
`PICL_FAILURE`	General system failure

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

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

See Also `ptree_get_prop_by_name(3PICLTREE)`, `attributes(5)`

Name ptree_get_frutree_parent – get frutree parent node for a given device node

Synopsis

```
cc [ flag... ] file... -lpicltree [ library... ]
#include <picltree.h>

int ptree_get_frutree_parent(picl_nodehdl_t devh,
     picl_nodehdl_t *frutreeh);
```

Description The devices under the `/platform` subtree of the PICLTREE are linked to their FRU containers represented in the `/frutree` using PICL reference properties. The `ptree_get_frutree_parent()` function returns the handle of the node in the `/frutree` subtree that is the FRU parent or container of the the device specified by the node handle, *devh*. The handle is returned in the *frutreeh* argument.

Return Values Upon successful completion, 0 is returned. Otherwise a non-negative integer is returned to indicate an error.

Errors

`PICL_FAILURE`	General system failure
`PICL_INVALIDHANDLE`	Invalid handle
`PICL_PROPNOTFOUND`	Property not found
`PICL_STALEHANDLE`	Stale handle

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

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

See Also `ptree_get_propinfo(3PICLTREE)`, `ptree_get_propval(3PICLTREE)`, `ptree_get_propval_by_name(3PICLTREE)`, `ptree_get_prop_by_name(3PICLTREE)`, `attributes(5)`

Name ptree_get_next_by_row, ptree_get_next_by_col – access a table property

Synopsis

```
cc [ flag... ] file... -lpicltree [ library... ]
#include <picltree.h>

int ptree_get_next_by_row(picl_prophdl_t proph,
     picl_prophdl_t *rowh);

int ptree_get_next_by_col(picl_prophdl_t proph,
     picl_prophdl_t *colh);
```

Description The `ptree_get_next_by_row()` function copies the handle of the property that is in the next column of the table and on the same row as the property *proph*. The handle is copied into the location given by *rowh*.

The `ptree_get_next_by_col()` function copies the handle of the property that is in the next row of the table and on the same column as the property *proph*. The handle is copied into the location given by *colh*.

If there are no more rows or columns, this function returns the value `PICL_ENDOFLIST`.

Return Values Upon successful completion, `0` is returned. On failure, a non-negative integer is returned to indicate an error.

`PICL_STALEHANDLE` is returned if the handle is no longer valid. This occurs if the PICL tree was refreshed or reinitialized.

`PICL_INVALIDHANDLE` is returned if the specified handle never existed.

Errors

`PICL_NOTTABLE`	Not a table
`PICL_INVALIDHANDLE`	Invalid handle
`PICL_STALEHANDLE`	Stale handle
`PICL_ENDOFLIST`	End of list
`PICL_FAILURE`	General system failure

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

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

See Also `ptree_create_table(3PICLTREE)`, `attributes(5)`

Name ptree_get_node_by_path – get handle of node specified by PICL tree path

Synopsis

```
cc [ flag... ] file... -lpicltree [ library... ]
#include <picltree.h>

int ptree_get_node_by_path(const char *ptreepath,
     picl_nodehdl_t *nodeh);
```

Description The `ptree_get_node_by_path()` function copies the handle of the node in the PICL tree specified by the path given in *ptreepath* into the location *nodeh*.

The syntax of a PICL tree path is:

`[`*def_propname*`:]/[`*def_propval*`[`*match_cond*`] ... ]`

where *def_propname* prefix is a shorthand notation to specify the name of the property whose value is specified in *def_propval*, and the *match_cond* expression specifies the matching criteria for that node in the form of one or more pairs of property names and values such as

`[@`*address*`][?`*prop_name*`[=`*prop_val*`] ... ]`

where '@' is a shorthand notation to refer to the device address, which is followed by the device addres value address. The address value is matched with the value of the property "bus-addr" if it exists. If no "bus-addr" property exists, then it is matched with the value of the property "UnitAddress". Use the '?' notation to limit explicitly the comparison to "bus-addr" or "UnitAddress" property. The expression following '?' specifies matching property name and value pairs, where *prop_name* gives the property name and *prop_val* gives the property value for non `PICL_PTYPE_VOID` properties. The values for properties of type `PICL_PTYPE_TABLE`, `PICL_PTYPE_BYTEARRAY`, and `PICL_PTYPE_REFERENCE` cannot be specified in the *match_cond* expression.

A "_class" property value of "picl" may be used to match nodes of all PICL classes.

All valid paths must start at the root node denoted by '/'.

If no prefix is specified for the path, then the prefix defaults to the "name" property.

Return Values Upon successful completion, `0` is returned. On failure, a non-negative integer is returned to indicate an error.

`PICL_NOTNODE` is returned if there is no node corresponding to the specified path.

Errors

`PICL_INVALIDARG`	Invalid argument
`PICL_NOTNODE`	Not a node
`PICL_FAILURE`	General system failure

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

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

See Also ptree_get_propval_by_name(3PICLTREE), attributes(5)

Name ptree_get_prop_by_name – get a property handle by name

Synopsis

```
cc [ flag ] file... -lpicltree [ library... ]
#include <picltree.h>

int ptree_get_prop_by_name(picl_nodehdl_t nodeh, char *name,
     picl_prophdl_t *proph);
```

Description The `ptree_get_prop_by_name()` function gets the handle of the property, whose name is specified in *name*, of the node specified by the handle *nodeh*. The property handle is copied into the location specified by *proph*.

Return Values Upon successful completion, `0` is returned. On failure, a non-negative integer is returned to indicate an error.

`PICL_RESERVEDNAME` is returned if the name specified is a PICL reserved name property. Reserved name properties do not have an associated property handle. Use `ptree_get_propval_by_name(3PICLTREE)` to get the value of a reserved property.

`PICL_STALEHANDLE` is returned if the handle is no longer valid. This occurs if the PICL tree was refreshed or reinitialized.

`PICL_INVALIDHANDLE` is returned if the specified handle never existed.

Errors

`PICL_NOTNODE`	Not a node
`PICL_RESERVEDNAME`	Property name is reserved
`PICL_INVALIDHANDLE`	Invalid handle
`PICL_STALEHANDLE`	Stale handle
`PICL_PROPNOTFOUND`	Property not found
`PICL_FAILURE`	General system failure

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

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

See Also `ptree_get_first_prop(3PICLTREE)`, `ptree_get_propval_by_name(3PICLTREE)`, `attributes(5)`

Name ptree_get_propinfo – get property information

Synopsis

```
cc [ flag... ] file... -lpicltree [ library... ]
#include <picltree.h>

int ptree_get_propinfo(picl_prophdl_t proph,
     ptree_propinfo_t *pi);
```

Description The `ptree_get_propinfo()` function gets the information about the property specified by handle *proph* and copies it into the location specified by *pi*. See `libpicltree(3PICLTREE)` for more information about `ptree_propinfo_t` structure.

Return Values Upon successful completion, `0` is returned. On failure, a non-negative integer is returned to indicate an error.

`PICL_STALEHANDLE` is returned if the handle is no longer valid. This occurs if the PICL tree was refreshed or reinitialized.

`PICL_INVALIDHANDLE` is returned if the specified handle never existed.

Errors

`PICL_INVALIDHANDLE`	Invalid handle
`PICL_STALEHANDLE`	Stale handle
`PICL_NOTPROP`	Not a property
`PICL_FAILURE`	General system failure

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

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

See Also `libpicltree(3PICLTREE)`, `ptree_create_prop(3PICLTREE)`, `attributes(5)`

Name ptree_get_propinfo_by_name – get property information and handle of named property

Synopsis

```
cc [ flag... ] file... -lpicltree [ library... ]
#include <picltree.h>

int ptree_get_propinfo_by_name(picl_nodehdl_t nodeh,
     const char *pname, ptree_propinfo_t *pinfo,
     picl_prophdl_t *proph);
```

Description The `ptree_get_propinfo_by_name()` function copies the property information of the property specified by *pname* in the node *nodeh* into the location given by *pinfo*. The handle of the property is returned in the location *proph*.

Return Values Upon successful completion, `0` is returned. On failure, a non-negative integer is returned to indicate an error.

Errors

`PICL_NOTNODE`	Not a node
`PICL_PROPNOTFOUND`	Property not found
`PICL_RESERVEDNAME`	Reserved property name specified
`PICL_INVALIDHANDLE`	Invalid handle
`PICL_STALEHANDLE`	Stale handle
`PICL_FAILURE`	General system failure

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

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

See Also `picl_get_propinfo(3PICL)`, `picl_get_prop_by_name(3PICL)`, `attributes(5)`

Name ptree_get_propval, ptree_get_propval_by_name – get the value of a property

Synopsis

```
cc [ flag... ] file... -lpicltree [ library... ]
#include <picltree.h>

int ptree_get_propval(picl_prophdl_t proph, void *valbuf,
     size_t nbytes);

int ptree_get_propval_by_name(picl_nodehdl_t nodeh,
     void *name, void *valbuf, size_t nbytes);
```

Description The `ptree_get_propval()` function gets the value of the property specified by the handle *proph* and copies it into the buffer specified by *valbuf*. The size of the buffer *valbuf* is specifed in *nbytes*.

The `ptree_get_propval_by_name()` function gets the value of the property, whose name is specified by *name*, from the node specified by handle *nodeh*. The value is copied into the buffer specified by *valbuf*. The size of the buffer is specified by *nbytes*.

For volatile properties, the read access function provided by the plug-in publishing the property is invoked.

Return Values Upon successful completion, `0` is returned. On failure, a non-negative integer is returned to indicate an error.

`PICL_STALEHANDLE` is returned if the handle is no longer valid. This occurs if the PICL tree was refreshed or reinitialized.

`PICL_INVALIDHANDLE` is returned if the specified handle never existed.

Errors

`PICL_VALUETOOBIG`	Value too big
`PICL_NOTPROP`	Not a property
`PICL_NOTNODE`	Not a node
`PICL_INVALIDHANDLE`	Invalid handle
`PICL_STALEHANDLE`	Stale handle
`PICL_PROPNOTFOUND`	Property not found
`PICL_FAILURE`	General system failure

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also ptree_update_propval(3PICLTREE), attributes(5)

Name ptree_get_root – get the root node handle

Synopsis

```
cc [ flag... ] file... -lpicltree [ library... ]
#include <picltree.h>

int ptree_get_root(picl_nodehdl_t *nodeh);
```

Description The `ptree_get_root()` function copies the handle of the root node of the PICL tree into the location specified by *nodeh*.

Return Values Upon successful completion, `0` is returned. On failure, a non-negative integer is returned to indicate an error.

Errors

`PICL_INVALIDARG`	Invalid argument
`PICL_FAILURE`	General system failure

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

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

See Also `libpicltree`(3PICLTREE), `ptree_create_node`(3PICLTREE), `attributes(5)`

Name ptree_init_propinfo – initialize ptree_propinfo_t structure

Synopsis

```
cc [ flag... ] file... -lpicltree [ library... ]
#include <picltree.h>

int ptree_init_propinfo(ptree_propinfo_t *infop, int version,
     int ptype, int pmode, size_t psize, char *pname,
     int (*readfn)(ptree_rarg_t *, void *),
     int (*writefn)(ptree_warg_t *, const void *));
```

Description The `ptree_init_propinfo()` function initializes a `ptree_propinfo_t` property information structure given by location *infop* with the values provided by the arguments.

The *version* argument specifies the version of the `ptree_propinfo_t` structure. `PTREE_PROPINFO_VERSION` gives the current version. The arguments *ptype*, *pmode*, *psize*, and *pname* specify the property's PICL type, access mode, size, and name. The maximum size of a property name is defined by `PICL_PROPNAMELEN_MAX`. The arguments *readfn* and *writefn* specify a volatile property's read and write access functions. For non-volatile properties, these are set to `NULL`.

Return Values Upon successful completion, `0` is returned. On failure, a non-negative integer is returned to indicate an error.

Errors

`PICL_INVALIDARG`	Invalid argument
`PICL_NOTSUPPORTED`	Property version not supported
`PICL_FAILURE`	General system failure

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

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

See Also `ptree_get_propinfo(3PICLTREE)`, `attributes(5)`

Name ptree_post_event – post a PICL event

Synopsis

```
cc [ flag... ] file... -lpicltree [ library... ]
#include <picltree.h>

int ptree_post_event(const char *ename, const void *earg,
     size_t size, void (*completion_handler)(char *ename,
     void *earg, size_t size));
```

Description The `ptree_post_event()` function posts the specified event and its arguments to the PICL framework. The argument *ename* specifies a pointer to a string containing the name of the PICL event. The arguments *earg* and *size* specify a pointer to a buffer containing the event arguments and size of that buffer, respectively. The argument *completion_handler* specifies the completion handler to be called after the event has been dispatched to all handlers. A `NULL` value for a completion handler indicates that no handler should be called. The PICL framework invokes the completion handler of an event with the *ename*, *earg*, and *size* arguments specified at the time of the posting of the event.

PICL events are dispatched in the order in which they were posted. They are dispatched by executing the handlers registered for that event. The handlers are invoked in the order in which they were registered.

New events will not begin execution until all previous events have finished execution. Specifically, an event posted from an event handler will not begin execution until the current event has finished execution.

The caller may not reuse or reclaim the resources associated with the event name and arguments until the invocation of the completion handler. The completion handlers are normally used to reclaim any resources allocated for the posting of an event.

Return Values Upon successful completion, `0` is returned. On failure, a non-negative integer is returned to indicate an error, the event is not posted, and the completion handler is not invoked.

Errors

`PICL_INVALIDARG`	Invalid argument
`PICL_FAILURE`	General system failure

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

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

See Also `ptree_register_handler(3PICLTREE)`, `ptree_unregister_handler(3PICLTREE)`, `attributes(5)`

Name ptree_register_handler – register a handler for the event

Synopsis

```
cc [ flag... ] file... -lpicltree [ library... ]
#include <picltree.h>

int ptree_register_handler(const char *ename,
     void (*evt_handler)(const char *ename, const void *earg,
     size_t size, void *cookie), void *cookie);
```

Description The `ptree_register_handler()` function registers an event handler for a PICL event. The argument *ename* specifies the name of the PICL event for which to register the handler. The argument *evt_handler* specifies the event handler function. The argument *cookie* is a pointer to caller-specific data to be passed as an argument to the event handler when it is invoked.

The event handler function must be defined as

```
void evt_handler(const char *ename, const void *earg, \
        size_t size, void *cookie)
```

where, *ename*, *earg*, *size*, and *cookie* are the arguments passed to the event handler when it is invoked. The argument *ename* is the PICL event name for which the handler is invoked. The arguments *earg* and *size* gives the pointer to the event argument buffer and its size, respectively. The argument *cookie* is the pointer to the caller specific data registered with the handler. The arguments *ename* and *earg* point to buffers that are transient and shall not be modified by the event handler or reused after the event handler finishes execution.

The PICL framework invokes the event handlers in the order in which they were registered when dispatching an event. If the event handler execution order is required to be the same as the plug-in dependency order, then a plug-in should register its handlers from its init function. The handlers that do not have any ordering dependencies on other plug-in handlers can be registered at any time.

The registered handler may be called at any time after this function is called.

Return Values Upon successful completion, `0` is returned. On failure, a non-negative integer is returned to indicate an error and the handler is not registered.

Errors

`PICL_INVALIDARG`	Invalid argument
`PICL_FAILURE`	General system failure

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

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

See Also ptree_unregister_handler(3PICLTREE), attributes(5)

Name ptree_unregister_handler – unregister the event handler for the event

Synopsis

```
cc [flag ...] file ... -lpicltree [library ...]
#include <picltree.h>

void ptree_register_handler(const char *ename,
     void (*evt_handler)(const char *ename, const void *earg,
     size_t size, void *cookie), void *cookie);
```

Description The `ptree_unregister_handler()` function unregisters the event handler for the specified event. The argument *ename* specifies the name of the PICL event for which to unregister the handler. The argument *evt_handler* specifies the event handler function. The argument *cookie* is the pointer to the caller-specific data given at the time of registration of the handler.

If the handler being unregistered is currently executing, then this function will block until its completion. Because of this, locks acquired by the handlers should not be held across the call to `ptree_unregister_handler()` or a deadlock may result.

The `ptree_unregister_handler()` function must not be invoked from the handler that is being unregistered.

Return Values This function does not return a value.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `ptree_register_handler(3PICLTREE)`, `attributes(5)`

Name ptree_update_propval, ptree_update_propval_by_name – update a property value

Synopsis

```
cc [ flag... ] file... -lpicltree [ library... ]
#include <picltree.h>

int ptree_update_propval(picl_prophdl_t proph, void *valbuf,
     size_t nbytes);

int ptree_update_propval_by_name(picl_nodehdl_t nodeh,
     char *name, void *valbuf, size_t nbytes);
```

Description The `ptree_update_propval()` function updates the value of the property specified by *proph* with the value specified in the buffer *valbuf*. The size of the buffer *valbuf* is specified in *nbytes*.

The `ptree_update_propval_by_name()` function updates the value of the property, whose name is specified by *name*, of the node specified by handle *nodeh*. The new value is specified in the buffer *valbuf*, whose size is specified in *nbytes*.

For volatile properties, the write access function provided by the plug-in publishing the property is invoked.

Return Values Upon successful completion, `0` is returned. On failure, a non-negative integer is returned to indicate an error.

`PICL_STALEHANDLE` is returned if the handle is no longer valid. This occurs if the PICL tree was refreshed or reinitialized.

`PICL_INVALIDHANDLE` is returned if the specified handle never existed.

Errors

`PICL_VALUETOOBIG`	Value too big
`PICL_NOTPROP`	Not a property
`PICL_NOTNODE`	Not a node
`PICL_INVALIDHANDLE`	Invalid handle
`PICL_STALEHANDLE`	Stale handle
`PICL_PROPNOTFOUND`	Property not found

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

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

See Also ptree_get_propval(3PICLTREE), attributes(5)

Name ptree_walk_tree_by_class – walk subtree by class

Synopsis

```
cc [ flag... ] file... -lpicltree [ library... ]
#include <picltree.h>

int ptree_walk_tree_by_class(picl_nodehdl_t rooth,
     const char *classname, void *c_args,
     int (*callback)(picl_nodehdl_t nodeh, void *c_args));
```

Description The `ptree_walk_tree_by_class()` function visits all the nodes of the subtree under the node specified by *rooth*. The PICL class name of the visited node is compared with the class name specified by *classname*. If the class names match, the callback function specified by *callback* is called with the matching node handle and the argument provided in *c_args*. If the class name specified in *classname* is `NULL`, then the callback function is invoked for all the nodes.

The return value from the callback function is used to determine whether to continue or terminate the tree walk. The callback function returns `PICL_WALK_CONTINUE` or `PICL_WALK_TERMINATE` to continue or terminate the tree walk.

Return Values Upon successful completion, `0` is returned. On failure, a non-negative integer is returned to indicate an error.

Errors

`PICL_NOTNODE`	Not a node
`PICL_INVALIDHANDLE`	Invalid handle specified
`PICL_STALEHANDLE`	Stale handle specified
`PICL_FAILURE`	General system failure

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

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

See Also `ptree_get_propval_by_name(3PICLTREE)`, `attributes(5)`

Name read_vtoc, write_vtoc – read and write a disk's VTOC

Synopsis

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

int read_vtoc(int fd, struct vtoc *vtoc);

int write_vtoc(int fd, struct vtoc *vtoc);

int read_extvtoc(int fd, struct extvtoc *extvtoc);

int write_extvtoc(int fd, struct extvtoc *extvtoc);
```

Description The `read_vtoc()` and `read_extvtoc()` functions return the VTOC (volume table of contents) structure that is stored on the disk associated with the open file descriptor *fd*. On disks larger than 1 TB `read_extvtoc()` must be used.

The `write_vtoc()` and `write_extvtoc()` function stores the VTOC structure on the disk associated with the open file descriptor *fd*. On disks larger then 1TB `write_extvtoc()` function must be used.

The *fd* argument refers to any slice on a raw disk.

Return Values Upon successful completion, `read_vtoc()` and `read_extvtoc()` return a positive integer indicating the slice index associated with the open file descriptor. Otherwise, they return a negative integer indicating one of the following errors:

`VT_EIO`	An I/O error occurred.
`VT_ENOTSUP`	This operation is not supported on this disk.
`VT_ERROR`	An unknown error occurred.
`VT_OVERFLOW`	The caller attempted an operation that is illegal on the disk and may overflow the fields in the data structure.

Upon successful completion, `write_vtoc()` and `write_extvtoc()` return 0. Otherwise, they return a negative integer indicating one of the following errors:

`VT_EINVAL`	The VTOC contains an incorrect field.
`VT_EIO`	An I/O error occurred.
`VT_ENOTSUP`	This operation is not supported on this disk.
`VT_ERROR`	An unknown error occurred.
`VT_OVERFLOW`	The caller attempted an operation that is illegal on the disk and may overflow the fields in the data structure.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also fmthard(1M), format(1M), prtvtoc(1M), ioctl(2), efi_alloc_and_init(3EXT), attributes(5), dkio(7I)

Bugs The `write_vtoc()` function cannot write a VTOC on an unlabeled disk. Use format(1M) for this purpose.

Name reg_ci_callback – provide a component instrumentation with a transient program number

Synopsis

```
cc [ flag ... ] file ... -ldmici    [ library ... ]
#include <dmi/ci_callback_svc.hh>

u_long reg_ci_callback();
```

Description The `reg_ci_callback()` function provides a component instrumentation with a transient program number. The instrumentation uses this number to register its RPC service provider. The `prognum` member of the `DmiRegisterInfo` structure is populated with the return value of this function

Return Values Upon successful completion, the `reg_ci_callback()` function returns a transient program number of type `u_long`.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-level	Unafe

See Also `attributes(5)`

Name regexpr, compile, step, advance – regular expression compile and match routines

Synopsis

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

#include <regexpr.h>

char *compile(char *instring, char *expbuf, const char *endbuf);

int
step(const char *string, const char *expbuf);

int
advance(const char *string, const char *expbuf);

extern char *loc1, loc2, locs;

extern int nbra, regerrno, reglength;

extern char *braslist[], *braelist[];
```

Description These routines are used to compile regular expressions and match the compiled expressions against lines. The regular expressions compiled are in the form used by `ed(1)`.

The parameter *instring* is a null-terminated string representing the regular expression.

The parameter *expbuf* points to the place where the compiled regular expression is to be placed. If *expbuf* is `NULL`, `compile()` uses `malloc(3C)` to allocate the space for the compiled regular expression. If an error occurs, this space is freed. It is the user's responsibility to free unneeded space after the compiled regular expression is no longer needed.

The parameter *endbuf* is one more than the highest address where the compiled regular expression may be placed. This argument is ignored if *expbuf* is `NULL`. If the compiled expression cannot fit in (*endbuf*–*expbuf*) bytes, `compile()` returns `NULL` and `regerrno` (see below) is set to 50.

The parameter *string* is a pointer to a string of characters to be checked for a match. This string should be null-terminated.

The parameter *expbuf* is the compiled regular expression obtained by a call of the function `compile()`.

The function `step()` returns non-zero if the given string matches the regular expression, and zero if the expressions do not match. If there is a match, two external character pointers are set as a side effect to the call to `step()`. The variables set in `step()` are `loc1` and `loc2`. `loc1` is a pointer to the first character that matched the regular expression. The variable `loc2` points to the character after the last character that matches the regular expression. Thus if the regular expression matches the entire line, `loc1` points to the first character of *string* and `loc2` points to the null at the end of *string*.

The purpose of `step()` is to step through the *string* argument until a match is found or until the end of *string* is reached. If the regular expression begins with `^`, `step()` tries to match the regular expression at the beginning of the string only.

The `advance()` function is similar to `step()`; but, it only sets the variable `loc2` and always restricts matches to the beginning of the string.

If one is looking for successive matches in the same string of characters, `locs` should be set equal to `loc2`, and `step()` should be called with *string* equal to `loc2`. `locs` is used by commands like `ed` and `sed` so that global substitutions like `s/y*//g` do not loop forever, and is `NULL` by default.

The external variable `nbra` is used to determine the number of subexpressions in the compiled regular expression. `braslist` and `braelist` are arrays of character pointers that point to the start and end of the `nbra` subexpressions in the matched string. For example, after calling `step()` or `advance()` with string `sabcdefg` and regular expression `\(abcdef\)`, `braslist[0]` will point at `a` and `braelist[0]` will point at `g`. These arrays are used by commands like `ed` and `sed` for substitute replacement patterns that contain the `\`*n* notation for subexpressions.

Note that it is not necessary to use the external variables `regerrno`, `nbra`, `loc1`, `loc2` `locs`, `braelist`, and `braslist` if one is only checking whether or not a string matches a regular expression.

Examples

EXAMPLE 1 The following is similar to the regular expression code from `grep`:

```
#include<regexpr.h>
 . . .
if(compile(*argv, (char *)0, (char *)0) == (char *)0)
     regerr(regerrno);
 . . .
if (step(linebuf, expbuf))
   succeed( );
```

Return Values

If `compile()` succeeds, it returns a non-`NULL` pointer whose value depends on *expbuf*. If *expbuf* is non-`NULL`, `compile()` returns a pointer to the byte after the last byte in the compiled regular expression. The length of the compiled regular expression is stored in `reglength`. Otherwise, `compile()` returns a pointer to the space allocated by `malloc(3C)`.

The functions `step()` and `advance()` return non-zero if the given string matches the regular expression, and zero if the expressions do not match.

Errors

If an error is detected when compiling the regular expression, a `NULL` pointer is returned from `compile()` and `regerrno` is set to one of the non-zero error numbers indicated below:

ERROR	MEANING
11	Range endpoint too large.
16	Bad Number.
25	"\digit" out or range.
36	Illegal or missing delimiter.

ERROR	MEANING
41	No remembered string search.
42	\(~\) imbalance.
43	Too many \(.
44	More than 2 numbers given in \[~\}.
45	} expected after \.
46	First number exceeds second in \{~\}.
49	[] imbalance.
50	Regular expression overflow.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also ed(1), grep(1), sed(1), malloc(3C), attributes(5), regexp(5)

Notes When compiling multi-threaded applications, the _REENTRANT flag must be defined on the compile line. This flag should only be used in multi-threaded applications.

Name remainder, remainderf, remainderl – remainder function

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double remainder(double x, double y);

float remainderf(float x, float y);

long double remainderl(long double x, long double y);
```

Description These functions return the floating point remainder $r = x - ny$ when y is non-zero. The value n is the integral value nearest the exact value x/y. When $|n - x/y| = ½$, the value n is chosen to be even.

The behavior of `remainder()` is independent of the rounding mode.

Return Values Upon successful completion, these functions return the floating point remainder $r = x - ny$ when y is non-zero.

If x or y is NaN, a NaN is returned.

If x is infinite or y is 0 and the other is non-NaN, a domain error occurs and a NaN is returned.

Errors These functions will fail if:

Domain Error — The x argument is ±Inf, or the y argument is ±0 and the other argument is non-NaN.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, then the invalid floating-point exception is raised.

The `remainder()` function sets `errno` to `EDOM` if y argument is 0 or the x argument is positive or negative infinity.

Usage An application wanting to check for error situations can set `errno` to 0 before calling `remainder()`. On return, if `errno` is non-zero, an error has occurred. The `remainderf()` and `remainderl()` functions do not set `errno`.

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

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

See Also abs(3C), div(3C), feclearexcept(3M), fetestexcept(3M), attributes(5), standards(5)

Name remquo, remquof, remquol – remainder functions

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double remquo(double x, double y, int *quo);

float remquof(float x, float y, int *quo);

long double remquol(long double x, long double y, int *quo);
```

Description The `remquo()`, `remquof()`, and `remquol()` functions compute the same remainder as the `remainder()`, `remainderf()`, and `remainderl()` functions, respectively. See `remainder(3M)`. In the object pointed to by *quo*, they store a value whose sign is the sign of *x*/*y* and whose magnitude is congruent modulo 2^n to the magnitude of the integral quotient of *x*/*y*, where *n* is an integer greater than or equal to 3.

Return Values These functions return *x* REM *y*.

If *x* or *y* is NaN, a NaN is returned.

If *x* is ±Inf or *y* is 0 and the other argument is non-NaN, a domain error occurs and a NaN is returned.

Errors These functions will fail if:

Domain Error — The *x* argument is Inf or the *y* argument is 0 and the other argument is non-NaN.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, then the invalid floating-point exception is raised.

Usage An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. An application should either examine the return value or check the floating point exception flags to detect exceptions.

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

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

See Also `feclearexcept(3M)`, `fetestexcept(3M)`, `math.h(3HEAD)`, `remainder(3M)`, `attributes(5)`, `standards(5)`

Name rint, rintf, rintl – round-to-nearest integral value

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double rint(double x);

float rintf(float x);

long double rintl(long double x);
```

Description These functions return the integral value (represented as a `double`) nearest *x* in the direction of the current rounding mode.

If the current rounding mode rounds toward negative infinity, `rint()` is equivalent to `floor`(3M). If the current rounding mode rounds toward positive infinity, `rint()` is equivalent to `ceil`(3M).

These functions differ from the `nearbyint`(3M), `nearbyintf()`, and `nearbyintl()` functions only in that they might raise the inexact floating-point exception if the result differs in value from the argument.

Return Values Upon successful completion, these functions return the integer (represented as a double precision number) nearest *x* in the direction of the current rounding mode.

If *x* is NaN, a NaN is returned.

If *x* is ±0 or ±Inf, *x* is returned.

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

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

See Also `abs`(3C), `ceil`(3M), `feclearexcept`(3M), `fetestexcept`(3M), `floor`(3M), `isnan`(3M), `math.h`(3HEAD), `nearbyint`(3M), `attributes`(5), `standards`(5)

Name round, roundf, roundl – round to nearest integer value in floating-point format

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double round(double x);

float roundf(float x);

long double roundl(long double x);
```

Description These functions round their argument to the nearest integer value in floating-point format, rounding halfway cases away from 0, regardless of the current rounding direction.

Return Values Upon successful completion, these functions return the rounded integer value.

If *x* is NaN, a NaN is returned.

If *x* is ±0 or ± Inf, *x* is returned.

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

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

See Also feclearexcept(3M), fetestexcept(3M), math.h(3HEAD), attributes(5), standards(5)

Name rsm_create_localmemory_handle, rsm_free_localmemory_handle – create or free local memory handle

Synopsis

```
cc [ flag... ] file... –lrsm [ library... ]
#include <rsmapi.h>

int rsm_create_localmemory_handle(
     rsmapi_controller_handle_t handle,
     rsm_localmemory_handle_t *l_handle,
     caddr_t local_vaddr, size_t length);

int rsm_free_localmemory_handle(
     rsmapi_controller_handle_t handle,
     rsm_localmemory_handle_t l_handle);
```

Description The `rsm_create_localmemory_handle()` and `rsm_free_localmemory_handle()` functions are supporting functions for `rsm_memseg_import_putv(3RSM)` and `rsm_memseg_import_getv(3RSM)`.

The `rsm_create_localmemory_handle()` function creates a local memory handle to be used in the I/O vector component of a scatter-gather list of subsequent `rsm_memseg_import_putv()` and `rsm_memseg_import_getv()` calls. The *handle* argument specifies the controller handle obtained from `rsm_get_controller(3RSM)`. The *l_handle* argument is a pointer to the location for the function to return the local memory handle. The *local_vaddr* argument specifies the local virtual address; it should be aligned at a page boundary. The *length* argument specifies the length of memory spanned by the handle.

The `rsm_free_localmemory_handle()` function unlocks the memory range for the local handle specified by *l_handle* and releases the associated system resources. The *handle* argument specifies the controller handle. All handles created by a process are freed when the process exits, but the process should call `rsm_free_localmemory_handle()` as soon as possible to free the system resources.

Return Values Upon successful completion, these functions return 0. Otherwise, an error value is returned to indicate the error.

Errors The `rsm_create_localmemory_handle()` and `rsm_free_localmemory_handle()` functions can return the following errors:

`RSMERR_BAD_CTLR_HNDL`	Invalid controller handle.
`RSMERR_BAD_LOCALMEM_HNDL`	Invalid local memory handle.

The `rsm_create_localmemory_handle()` function can return the following errors:

`RSMERR_BAD_LENGTH`	Invalid length.
`RSMERR_BAD_ADDRESS`	Invalid address.
`RSMERR_INSUFFICIENT_MEM`	Insufficient memory.

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

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

See Also rsm_memseg_import_putv(3RSM), attributes(5)

Name rsm_get_controller, rsm_get_controller_attr, rsm_release_controller – get or release a controller handle

Synopsis

```
cc [ flag... ] file... –lrsm [ library... ]
#include <rsmapi.h>

int rsm_get_controller(char *name,
     rsmapi_controller_handle_t *controller);

int rsm_get_controller_attr(rsmapi_controller_handle_t chdl,
     rsmapi_controller_attr_t *attr);

int rsm_release_controller(rsmapi_controller_handle_t chdl);
```

Description The controller functions provide mechanisms for obtaining access to a controller, determining the characteristics of the controller, and releasing the controller.

The `rsm_get_controller()` function acquires a controller handle through the *controller* argument. The *name* argument is the specific controller instance (for example, "sci0" or "loopback"). This controller handle is used for subsequent RSMAPI calls.

The `rsm_get_controller_attr()` function obtains a controller's attributes through the *attr* argument. The *chdl* argument is the controller handle obtained by the `rsm_get_controller()` call. The attribute structure is defined in the `<rsmapi>` header.

The `rsm_release_controller()` function releases the resources associated with the controller identified by the controller handle *chdl*, obtained by calling `rsm_get_controller()`. Each `rsm_release_controller()` call must have a corresponding `rsm_get_controller()` call. It is illegal to access a controller or segments exported or imported using a released controller.

Return Values Upon successful completion, these functions return 0. Otherwise, an error value is returned to indicate the error.

Errors The `rsm_get_controller()`, `rsm_get_controller_attr()`, and `rsm_release_controller()` functions can return the following errors:

`RSMERR_BAD_CTLR_HNDL` Invalid controller handle.

The `rsm_get_controller()` and `rsm_get_controller_attr()` functions can return the following errors:

`RSMERR_BAD_ADDR` Bad address.

The `rsm_get_controller()` function can return the following errors:

`RSMERR_CTLR_NOT_PRESENT` Controller not present.

`RSMERR_INSUFFICIENT_MEM` Insufficient memory.

`RSMERR_BAD_LIBRARY_VERSION` Invalid library version.

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

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

See Also rsm_memseg_export_create(3RSM), rsm_memseg_import_connect(3RSM), attributes(5)

Name rsm_get_interconnect_topology, rsm_free_interconnect_topology – get or free interconnect topology

Synopsis

```
cc [ flag... ] file... –lrsm [ library... ]
#include <rsmapi.h>

int rsm_get_interconnect_topology(rsm_topology_t **topology_data);

void rsm_free_interconnect_topology(rsm_topology_t *topology_data);
```

Description The `rsm_get_interconnect_topology(3RSM)` and `rsm_free_interconnect_topology(3RSM)` functions provide for access to the interconnect controller and connection data. The key interconnect data required for export and import operations includes the respective cluster nodeids and the controller names. To facilitate applications in the establishment of proper and efficient export and import policies, a delineation of the interconnect topology is provided by this interface. The data provided includes local nodeid, local controller name, its hardware address, and remote connection specification for each local controller. An application component exporting memory can thus find the set of existing local controllers and correctly assign controllers for the creation and publishing of segments. Exported segments may also be efficiently distributed over the set of controllers consistent with the hardware interconnect and application software. An application component which is to import memory must be informed of the segment id(s) and controller(s) used in the exporting of memory, this needs to be done using some out-of-band mechanism. The topology data structures are defined in the `<rsmapi.h>` header.

The `rsm_get_interconnect_topology()` returns a pointer to the topology data in a location specified by the *topology_data* argument.

The `rsm_free_interconnect_topology()` frees the resources allocated by `rsm_get_interconnect_topology()`.

Return Values Upon successful completion, `rsm_get_interconnect_topology()` returns 0. Otherwise, an error value is returned to indicate the error.

Errors The `rsm_get_interconnect_topology()` function can return the following errors:

`RSMERR_BAD_TOPOLOGY_PTR`	Invalid topology pointer.
`RSMERR_INSUFFICIENT_MEM`	Insufficient memory.
`RSMERR_BAD_ADDR`	Bad address.

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

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

See Also attributes(5)

Name rsm_get_segmentid_range – get segment ID range

Synopsis

```
cc [ flag... ] file... –lrsm [ library... ]
#include <rsmapi.h>

int rsm_get_segmentid_range(const char *appid,
     rsm_memseg_id_t *baseid, uint32_t *length);
```

Description RSM segment IDs can be either specified by the application or generated by the system using the `rsm_memseg_export_publish(3RSM)` function. Applications that specify segment IDs require a reserved range of segment IDs that they can use. This can be achieved by using `rsm_get_segmentid_range()` and by reserving a range of segment IDs in the segment ID configuration file, `/etc/rsm/rsm.segmentid`. The `rsm_get_segmentid_range()` function can be used by applications to obtain the segment ID range reserved for them. The *appid* argument is a null-terminated string that identifies the application. The *baseid* argument points to the location where the starting segment ID of the reserved range is returned. The *length* argument points to the location where the number of reserved segment IDs is returned.

The application can use any value starting at *baseid* and less than *baseid+length*. The application should use an offset within the range of reserved segment IDs to obtain a segment ID such that if the *baseid* or *length* is modified, it will still be within its reserved range.

It is the responsibility of the system administrator to make sure that the segment ID ranges are properly administered (such that they are non-overlapping, the file on various nodes of the cluster have identical entries, and so forth.) Entries in the `/etc/rsm/rsm.segmentid` file are of the form:

```
#keyword         appid    baseid    length
reserved     SUNWfoo  0x600000  1000
```

The fields in the file are separated by tabs or blanks. The first string is a keyword "reserve", followed by the application identifier (a string without spaces), the baseid (the starting segment ID of the reserved range in hexadecimal), and the length (the number of segmentids reserved). Comment lines contain a "#" in the first column. The file should not contain blank or empty lines. Segment IDs reserved for the system are defined in the `</usr/include/rsm/rsm_common.h>` header and cannot be used by the applications.

Return Values Upon successful completion, these functions return 0. Otherwise, an error value is returned to indicate the error.

Errors The `rsm_get_segmentid_range()` function can return the following errors:

`RSMERR_BAD_ADDR`	The address passed is invalid.
`RSMERR_BAD_APPID`	The *appid* is not defined in configuration file.
`RSMERR_BAD_CONF`	The configuration file is not present or not readable, or the configuration file format is incorrect.

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

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

See Also rsm_memseg_export_publish(3RSM), attributes(5)

Name rsm_intr_signal_post, rsm_intr_signal_wait – signal or wait for an event

Synopsis

```
cc [ flag... ] file... –lrsm [ library... ]
#include <rsmapi.h>

int rsm_intr_signal_post(void *memseg, uint_t flags);

int rsm_intr_signal_wait(void *memseg, int timeout);
```

Description The `rsm_intr_signal_post()` and `rsm_intr_signal_wait()` functions are event functions that allow synchronization between importer processes and exporter processes. A process may block to wait for an event occurance by calling `rsm_intr_signal_wait()`. A process can signal a waiting process when an event occurs by calling `rsm_intr_signal_post()`.

The `rsm_intr_signal_post()` function signals an event occurance. Either an import segment handle (`rsm_memseg_import_handle_t`) or an export segment handle (`rsm_memseg_export_handle_t`) may be type cast to a void pointer for the *memseg* argument. If *memseg* refers to an import handle, the exporting process is signalled. If *memseg* refers to an export handle, all importers of that segment are signalled. The *flags* argument may be set to `RSM_SIGPOST_NO_ACCUMULATE`; this will cause this event to be discarded if an event is already pending for the target segment.

The `rsm_intr_signal_wait()` function allows a process to block and wait for an event occurance. Either an import segment handle (`rsm_memseg_import_handle_t`) or an export segment handle (`rsm_memseg_export_handle_t`) may be type cast to a void pointer for the *memseg* argument. The process blocks for up to *timeout* milliseconds for an event to occur; if the timeout value is -1, the process blocks until an event occurs or until interrupted.

Return Values Upon successful completion, these functions return 0. Otherwise, an error value is returned to indicate the error.

Errors The `rsm_intr_signal_post()` and `rsm_intr_signal_wait()` functions can return the following error:

`RSMERR_BAD_SEG_HNDL` Invalid segment handle.

The `rsm_intr_signal_post()` function can return the following error:

`RSMERR_CONN_ABORTED` Connection aborted.

`RSMERR_REMOTE_NODE_UNREACHABL` Remote node not reachable.

The `rsm_intr_signal_wait()` function can return the following errors:

`RSMERR_INTERRUPTED` Wait interrupted.

`RSMERR_TIMEOUT` Timer expired.

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

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

See Also rsm_memseg_get_pollfd(3RSM), attributes(5)

Name rsm_intr_signal_wait_pollfd – wait for events on a list of file descriptors

Synopsis

```
cc [ flag... ] file... -lrsm [ library ... ]
#include <rsmapi.h>

int rsm_intr_signal_wait_pollfd(struct pollfd fds[],
     nfds_t nfds, int timeout, int *numfdsp);
```

Description The `rsm_intr_signal_wait_pollfd()` function is similar to `rsm_intr_signal_wait(3RSM)`, except that it allows an application to multiplex I/O over various types of file descriptors. Applications can use this function to wait for interrupt signals on RSMAPI segments as well as poll for I/O events on other non-RSMAPI file descriptors.

The *fds* argument is an array of `pollfd` structures that correspond to both RSMAPI segments and other file descriptors. The `rsm_memseg_get_pollfd(3RSM)` is used to obtain a `pollfd` structure corresponding to an RSMAPI segment.

The number of file descriptors that have events is returned in *numfdsp*. This parameter can be set to `NULL` if the application is not interested in the number of file descriptors that have events. See `poll(2)` for descriptions of the `pollfd` structure as well as the *nfds* and *timeout* parameters.

It is the application's responsibility to establish the validity of a `pollfd` structure corresponding to an RSMAPI segment by ensuring that `rsm_memseg_release_pollfd(3RSM)` has not been called on the segment or that the segment has not been destroyed.

For file descriptors other than RSMAPI segments, the behavior of `rsm_intr_signal_wait_pollfd()` is similar to `poll()`.

Return Values Upon successful completion, `rsm_intr_signal_wait_pollfd()` returns 0 and the `revents` member of the `pollfd` struct corresponding to an RSMAPI segment is set to `POLLRDNORM`, indicating that an interrupt signal for that segment was received. Otherwise, an error value is returned.

For file descriptors other than RSMAPI segments, the `revents` member of the `pollfd` struct is identical to that returned by `poll(2)`.

Errors The `rsm_intr_signal_wait_pollfd()` function can return the following errors:

`RSMERR_TIMEOUT`	Timeout has occured.
`RSMERR_BAD_ARGS_ERRORS`	Invalid arguments passed.
`RSMERR_BAD_ADDR`	An argument points to an illegal address.
`RSMERR_INTERRUPTED`	The call was interrupted.
`RSMERR_INSUFFICIENT_MEM`	Insufficient memory.
`RSMERR_INSUFFICIENT_RESOURCES`	Insufficient resources.

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

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

See Also poll(2), rsm_intr_signal_wait(3RSM), rsm_memseg_get_pollfd(3RSM), rsm_memseg_release_pollfd(3RSM), attributes(5)

Name rsm_memseg_export_create, rsm_memseg_export_destroy, rsm_memseg_export_rebind – resource allocation and management functions for export memory segments

Synopsis

```
cc [ flag... ] file... -lrsm [ library... ]
#include <rsmapi.h>

int rsm_memseg_export_create(
     rsmapi_controller_handle_t controller,
     rsm_memseg_export_handle_t *memseg, void *vaddr,
     size_t length, uint_t flags);

int rsm_memseg_export_destroy(
     rsm_memseg_export_handle_t memseg);

int rsm_memseg_export_rebind(
     rsm_memseg_export_handle_t memseg,
     void *vaddr, offset_t off, size_t length);
```

Description The `rsm_memseg_export_create()`, `rsm_memseg_export_destroy()`, and `rsm_memseg_export_rebind()` functions provide for allocation and management of resources supporting export memory segments. Exporting a memory segment involves the application allocating memory in its virtual address space through the System V shared memory interface or normal operating system memory allocation functions. This is followed by the calls to create the export segment and bind physical pages to back to allocated virtual address space.

The `rsm_memseg_export_create()` creates a new memory segment. Physical memory pages are allocated and are associated with the segment. The segment lifetime is the same as the lifetime of the creating process or until a destroy operation is performed. The *controller* argument is the controller handle obtained from a prior call to `rsm_get_controller(3RSM)`. The export memory segment handle is obtained through the *memseg* argument for use in subsequent operations. The *vaddr* argument specifies the process virtual address for the segment. It must be aligned according to the controller page size attribute. The *length* argument specifies the size of the segment in bytes and must be in multiples of the controller page size. The *flags* argument is a bitmask of flags. Possible values are:

`RSM_ALLOW_REBIND`	Unbind and rebind is allowed on the segment during its lifetime.
`RSM_CREATE_SEG_DONTWAIT`	The RSMAPI segment create operations `rsm_memseg_export_create()` and `rsm_memseg_export_publish(3RSM)` should not block for resources and return `RSMERR_INSUFFICIENT_RESOURCES` to indicate resource shortage.
`RSM_LOCK_OPS`	This segment can be used for lock operations.

The `rsm_memseg_export_destroy()` function deallocates the physical memory pages associated with the segment and disconnects all importers of the segment. The *memseg* argument is the export memory segment handle obtained by a call to `rsm_memseg_export_create()`.

The `rsm_memseg_export_rebind()` function releases the current backing pages associated with the segment and allocates new physical memory pages. This operation is transparent to the importers of the segment. It is the responsibility of the application to prevent data access to the export segment until the rebind operation has completed. Segment data access during rebind does not cause a system failure but data content results are undefined. The *memseg* argument is the export segment handle pointer obtained from `rsm_memseg_export_create()`. The *vaddr* argument must be aligned with respect to the page size attribute of the controller. The *length* argument modulo controller page size must be 0. The *off* argument is currently unused.

Return Values Upon successful completion, these functions return 0. Otherwise, an error value is returned to indicate the error.

Errors The `rsm_memseg_export_create()`, `rsm_memseg_export_destroy()`, and `rsm_memseg_export_rebind()` functions can return the following errors:

`RSMERR_BAD_SEG_HNDL` Invalid segment handle.

The `rsm_memseg_export_create()` and `rsm_memseg_export_rebind()` functions can return the following errors:

`RSMERR_BAD_CTLR_HNDL`	Invalid controller handle.
`RSMERR_CTLR_NOT_PRESENT`	Controller not present.
`RSMERR_BAD_LENGTH`	Length zero or length exceeds controller limits.
`RSMERR_BAD_ADDR`	Invalid address.
`RSMERR_INSUFFICIENT_MEM`	Insufficient memory.
`RSMERR_INSUFFICIENT_RESOURCES`	Insufficient resources.
`RSMERR_PERM_DENIED`	Permission denied.
`RSMERR_NOT_CREATOR`	Not creator of segment.
`RSMERR_REBIND_NOT_ALLOWED`	Rebind not allowed.

The `rsm_memseg_export_create()` function can return the following errors:

`RSMERR_BAD_MEM_ALIGNMENT` The address is not aligned on a page boundary.

The `rsm_memseg_export_rebind()` function can return the following errors:

`RSMERR_INTERRUPTED` The operation was interrupted by a signal.

The `rsm_memseg_export_destroy()` function can return the following errors:

`RSMERR_POLLFD_IN_USE` Poll file descriptor in use.

Usage Exporting a memory segment involves the application allocating memory in its virtual address space through the System V Shared Memory interface or other normal operating system memory allocation methods such as `valloc()` (see `malloc(3C)`) or `mmap(2)`. Memory for a file mapped with `mmap()` must be mapped `MAP_PRIVATE`.

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

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

See Also `rsm_get_controller(3RSM)`, `rsm_memseg_export_publish(3RSM)`, `attributes(5)`

Name rsm_memseg_export_publish, rsm_memseg_export_unpublish, rsm_memseg_export_republish – allow or disallow a memory segment to be imported by other nodes

Synopsis

```
cc [ flag... ] file... –lrsm [ library... ]
#include <rsmapi.h>

int rsm_memseg_export_publish(
     rsm_memseg_export_handle_t memseg,
     rsm_memseg_id_t *segment_id,
     rsmapi_access_entry_t access_list[],
     uint_t access_list_length);

int rsm_memseg_export_unpublish(
     rsm_memseg_export_handle_t memseg);

int rsm_memseg_export_republish(
     rsm_memseg_export_handle_t memseg,
     rsmapi_access_entry_t access_list[],
     uint_t access_list_length);
```

Description The `rsm_memseg_export_publish()`, `rsm_memseg_export_unpublish()`, and `rsm_memseg_export_republish()` functions allow or disallow a memory segment to be imported by other nodes.

The `rsm_memseg_export_publish(3RSM)` function allows the export segment specified by the *memseg* argument to be imported by other nodes. It also assigns a unique segment identifier to the segment and defines the access control list for the segment. The *segment_id* argument is a pointer to an identifier which is unique on the publishing node. It is the responsibility of the application to manage the assignment of unique segment identifiers. The identifier can be optionally initialized to 0, in which case the system will return a unique segment identifier value. The *access_list* argument is composed of pairs of nodeid and access permissions. For each nodeid specified in the list, the associated read/write permissions are provided by three octal digits for owner, group, and other, as for Solaris file permissions. In the access control each octal digit may have the following values:

2 write access

4 read only access

6 read and write access

An access permissions value of 0624 specifies: (1) an importer with the same uid as the exporter has read and write access; (2) an importer with the same gid as the exporter has write access only; and (3) all other importers have read access only. When an access control list is provided, nodes not included in the list will be prevented from importing the segment. However, if the access list is `NULL` (this will require the length *access_list_length* to be specified as 0 as well), then no nodes will be excluded from importing and the access permissions on all

nodes will equal the owner-group-other file creation permissions of the exporting process. Corresponding to the *access_list* argument, the *access_list_length* argument specifies the number of entries in the *access_list* array.

The `rsm_memseg_export_unpublish()` function disallows the export segment specified by *memseg* from being imported. All the existing import connections are forcibly disconnected.

The `rsm_memseg_export_republish()` function changes the access control list for the exported and published segment. Although the current import connections remain unaffected by this call, new connections are constrained by the new access list.

Return Values Upon successful completion, these functions return 0. Otherwise, an error value is returned to indicate the error.

Errors The `rsm_memseg_export_publish()`, `rsm_memseg_export_unpublish()`, and `rsm_memseg_export_republish()` functions can return the following errors:

`RSMERR_BAD_SEG_HNDL`	Invalid segment handle.
`RSMERR_NOT_CREATOR`	Not creator of segment.

The `rsm_memseg_export_publish()` and `rsm_memseg_export_republish()` functions can return the following errors, with the exception that only `rsm_memseg_export_publish()` can return the errors related to the segment identifier:

`RSMERR_SEGID_IN_USE`	Segment identifier in use.
`RSMERR_RESERVED_SEGID`	Segment identifier reserved.
`RSMERR_BAD_SEGID`	Invalid segment identifier.
`RSMERR_BAD_ACL`	Invalid access control list.
`RSMERR_SEG_ALREADY_PUBLISHED`	Segment already published.
`RSMERR_INSUFFICIENT_MEM`	Insufficient memory.
`RSMERR_INSUFFICIENT_RESOURCES`	Insufficient resources.
`RSMERR_LOCKS_NOT_SUPPORTED`	Locks not supported.
`RSMERR_BAD_ADDR`	Bad address.

The `rsm_memseg_export_republish()` and `rsm_memseg_export_unpublish()` functions can return the following errors:

`RSMERR_SEG_NOT_PUBLISHED`	Segment not published.
`RSMERR_INTERRUPTED`	The operation was interrupted by a signal.

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

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

See Also rsm_memseg_export_create(3RSM), attributes(5)

Notes An attempt to publish a segment might block until sufficient resources become available.

Name rsm_memseg_get_pollfd, rsm_memseg_release_pollfd – get or release a poll descriptor

Synopsis

```
cc [ flag... ] file... –lrsm [ library... ]
#include <rsmapi.h>

int rsm_memseg_get_pollfd(void *memseg, struct pollfd *fd);

int rsm_memseg_release_pollfd(void *memseg);
```

Description The `rsm_memseg_get_pollfd()` and `rsm_memseg_release_pollfd()` functions provide an alternative to `rsm_intr_signal_wait(3RSM)`. The waiting process can multiplex event waiting using the `poll(2)` function after first obtaining a poll descriptor using `rsm_memseg_get_pollfd()`. The descriptor can subsequently be released using `rsm_memseg_release_pollfd()`.

As a result of a call `rsm_memseg_get_pollfd()`, the specified `pollfd` structure is initialized with a descriptor for the specified segment (*memseg*) and the event generated by `rsm_intr_signal_post(3RSM)`. Either an export segment handle or an import segment handle can be type cast to a void pointer. The *pollfd* argument can subsequently be used with the `rsm_intr_signal_wait_pollfd(3RSM)` function to wait for the event; it cannot be used with `poll()`. If *memseg* references an export segment, the segment must be currently published. If *memseg* references an import segment, the segment must be connected.

The `rsm_memseg_reslease_pollfd()` function decrements the reference count of the `pollfd` structure associated with the specified segment. A segment unpublish, destroy or unmap operation will fail if the reference count is non-zero.

Return Values Upon successful completion, these functions return 0. Otherwise, an error value is returned to indicate the error.

Errors The `rsm_memseg_get_pollfd()` and `rsm_memseg_release_pollfd()` function can return the following error:

`RSMERR_BAD_SEG_HNDL` Invalid segment handle.

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

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

See Also `poll(2)`, `rsm_intr_signal_post(3RSM)`, `rsm_intr_signal_wait_pollfd(3RSM)`, `attributes(5)`

Name rsm_memseg_import_connect, rsm_memseg_import_disconnect – create or break logical commection between import and export segments

Synopsis

```
cc [ flag... ] file... –lrsm [ library... ]
#include <rsmapi.h>

int rsm_memseg_import_connect(
     rsmapi_controller_handle_t controller,
     rsm_node_id_t nodeid, rsm_memseg_id_t segment_id,
     rsm_permission_t perm, rsm_memseg_import_handle_t *memseg);

int rsm_memseg_import_disconnect(
     rsm_memseg_import_handle_t memseg);
```

Description The `rsm_memseg_import_connect()` function provides a means of creating an import segment called *memseg* and establishing a logical connection with an export segment identified by the *segment_id* on the node specified by *node_id*. The controller specified by *controller* must have a physical connection with the controller (see `rsm_get_interconnect_topology(3RSM)`) used while exporting the segment identified by *segment_id* on node specified by *node_id*. The *perm* argument specifies the mode of access that the importer is requesting for this connection. In the connection process, the mode of access and the importers userid and groupid are compared with the access permissions specified by the exporter. If the request mode is not valid, the connection request is denied. The *perm* argument is limited to the following octal values:

0400 read mode

0200 write mode

0600 read/write mode

The `rsm_memseg_import_disconnect()` function breaks the logical connection between the import segment and the exported segment and deallocates the resources associated with the import segment handle *memseg*.

Return Values Upon successful completion, these functions return 0. Otherwise, an error value is returned to indicate the error.

Errors The `rsm_memseg_import_connect()` and `rsm_memseg_import_disconnect()` functions can return the following errors:

`RSMERR_BAD_SEG_HNDL` Invalid segment handle.

The `rsm_memseg_import_connect()` function can return the following errors:

`RSMERR_BAD_CTLR_HNDL` Invalid controller handle.

`RSMERR_CTLR_NOT_PRESENT` Controller not present.

`RSMERR_PERM_DENIED` Permission denied.

RSMERR_INSUFFICIENT_MEM	Insufficient memory.
RSMERR_INSUFFICIENT_RESOURCES	Insufficient resources.
RSMERR_SEG_NOT_PUBLISHED_TO_NODE	Segment not published to node.
RSMERR_SEG_NOT_PUBLISHED	Segment not published at all.
RSMERR_BAD_ADDR	Bad address.
RSMERR_REMOTE_NODE_UNREACHABLE	Remote not not reachable.
RSMERR_INTERRUPTED	Connection interrupted.

The rsm_memseg_import_disconnect() function can return the following errors:

RSMERR_SEG_STILL_MAPPED	Segment still mapped, need to unmap before disconnect.
RSMERR_POLLFD_IN_USE	Poll file descriptor in use.

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

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

See Also rsm_memseg_import_map(3RSM), attributes(5)

Name rsm_memseg_import_get, rsm_memseg_import_get8, rsm_memseg_import_get16, rsm_memseg_import_get32, rsm_memseg_import_get64 – read from a segment

Synopsis

```
cc [ flag... ] file... –lrsm [ library... ]
#include <rsmapi.h>

int rsm_memseg_import_get(rsm_memseg_import_handle_t im_memseg,
     off_t offset, void *dest_addr, size_t length);

int rsm_memseg_import_get8(rsm_memseg_import_handle_t im_memseg,
     off_t offset, uint8_t *datap, ulong_t rep_cnt);

int rsm_memseg_import_get16(rsm_memseg_import_handle_t im_memseg,
     off_t offset, uint16_t *datap, ulong_t rep_cnt);

int rsm_memseg_import_get32(rsm_memseg_import_handle_t im_memseg,
     off_t offset, uint32_t *datap, ulong_t rep_cnt);

int rsm_memseg_import_get64(rsm_memseg_import_handle_t im_memseg,
     off_t offset, uint64_t *datap, ulong_t rep_cnt);
```

Description When using interconnects that allow memory mapping (see `rsm_memseg_import_map(3RSM)`), standard CPU memory operations may be used for accessing memory of a segment. If a mapping is not provided, then explicitly calling these functions facilitates reading from a segment. Depending on the attributes of the extension library of the specific interconnect, these functions may involve performing an implicit mapping before performing the data transfer. Applications can be made interconnect-independent with respect to segment reads by using these functions. The data access error detection is performed through the use of barriers (see `rsm_memseg_import_open_barrier(3RSM)`). The default barrier operation mode is `RSM_BARRIER_MODE_IMPLICIT`, meaning that around every get operation open and close barrier are performed automatically. Alternatively, explicit error handling may be set up for these functions (see `rsm_memseg_import_set_mode(3RSM)`). In either case the barrier should be initialized prior to using these functions using `rsm_memseg_import_init_barrier(3RSM)`.

The `rsm_memseg_import_get()` function copies *length* bytes from the imported segment *im_memseg* beginning at location *offset* from the start of the segment to a local memory buffer pointed to by *dest_addr*.

The `rsm_memseg_import_get8()` function copies *rep_cnt* number of 8-bit quantities from successive locations starting from *offset* in the imported segment to successive local memory locations pointed to by *datap*.

The `rsm_memseg_import_get16()` functions copies *rep_cnt* number of 16-bit quantities from successive locations starting from *offset* in the imported segment to successive local memory locations pointed to by *datap*. The offset must be aligned at half-word address boundary.

The `rsm_memseg_import_get32()` function copies *rep_cnt* number of 32-bit quantities from successive locations starting from *offset* in the imported segment to successive local memory locations pointed to by *datap*. The offset must be aligned at word address boundary.

The `rsm_memseg_import_get64()` function copies *rep_cnt* number of -bit quantities from successive locations starting from *offset* in the imported segment to successive local memory locations pointed to by *datap*. The offset must be aligned at double-word address boundary.

The data transfer functions that transfer small quantities of data (that is, 8-, 16-, 32-, and 64-bit quantities) perform byte swapping prior to the data transfer, in the event that the source and destination have incompatible endian characteristics.

Return Values Upon successful completion, these functions return 0. Otherwise, an error value is returned to indicate the error.

Errors These functions can return the following errors:

`RSMERR_BAD_SEG_HNDL`	Invalid segment handle.
`RSMERR_BAD_ADDR`	Bad address.
`RSMERR_BAD_MEM_ALIGNMENT`	Invalid memory alignment for pointer.
`RSMERR_BAD_OFFSET`	Invalid offset.
`RSMERR_BAD_LENGTH`	Invalid length.
`RSMERR_PERM_DENIED`	Permission denied.
`RSMERR_INSUFFICIENT_RESOURCES`	Insufficient resources.
`RSMERR_BARRIER_UNINITIALIZED`	Barrier not initialized.
`RSMERR_BARRIER_FAILURE`	I/O completion error.
`RSMERR_CONN_ABORTED`	Connection aborted.

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

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

See Also `rsm_memseg_import_init_barrier(3RSM)`, `rsm_memseg_import_open_barrier(3RSM)`, `rsm_memseg_import_set_mode(3RSM)`, `attributes(5)`

Name rsm_memseg_import_init_barrier, rsm_memseg_import_destroy_barrier – create or destroy barrier for imported segment

Synopsis

```
cc [ flag... ] file... –lrsm [ library... ]
#include <rsmapi.h>

int rsm_memseg_import_init_barrier(
     rsm_memseg_import_handle_t memseg, rsm_barrier_type_t type,
     rsmapi_barrier_t *barrier);

int rsm_memseg_import_destroy_barrier(rsmapi_barrier_t *barrier);
```

Description The `rsm_memseg_import_init_barrier()` function creates a barrier for the imported segment specified by *memseg*. The barrier type is specified by the *type* argument. Currently, only `RSM_BAR_DEFAULT` is supported as a barrier type. A handle to the barrier is obtained through the *barrier* argument and is used in subsequent barrier calls.

The `rsm_memseg_import_destroy_barrier()` function deallocates all the resources associated with the barrier.

Return Values Upon successful completion, these functions return 0. Otherwise, an error value is returned to indicate the error.

Errors The `rsm_memseg_import_init_barrier()` and `rsm_memseg_import_destroy_barrier()` functions can return the following errors:

`RSMERR_BAD_SEG_HNDL` Invalid segment handle.

`RSMERR_BAD_BARRIER_PTR` Invalid barrier pointer.

The `rsm_memseg_import_init_barrier()` function can return the following errors:

`RSMERR_INSUFFICIENT_MEM` Insufficient memory.

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

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

See Also `rsm_memseg_import_open_barrier(3RSM)`, `rsm_memseg_import_set_mode(3RSM)`, `attributes(5)`

Name rsm_memseg_import_map, rsm_memseg_import_unmap – map or unmap imported segment

Synopsis

```
cc [ flag... ] file... –lrsm [ library... ]
#include <rsmapi.h>

int rsm_memseg_import_map(rsm_memseg_import_handle_t im_memseg,
     void **address, rsm_attribute_t attr,
     rsm_permission_t perm, off_t offset, size_t length);

int rsm_memseg_import_unmap(rsm_memseg_import_handle_t im_memseg);
```

Description The `rsm_memseg_import_map()` and `rsm_memseg_import_unmap()` functions provide for mapping and unmapping operations on imported segments. The mapping operations are only available for native architecture interconnects such as Dolphin-SCI or Sun Fire Link. Mapping a segment allows that segment to be accessed by CPU memory operations, saving the overhead of calling the memory access primitives described on the `rsm_memseg_import_get(3RSM)` and `rsm_memseg_import_put(3RSM)` manual pages.

The `rsm_memseg_import_map()` function maps an import segment into caller's address space for the segment to be accessed by CPU memory operations. The *im_memseg* argument represents the import segment that is being mapped. The location where the process's address space is mapped to the segment is pointed to by the *address* argument. The *attr* argiment can be one fo the following:

`RSM_MAP_NONE` The system will choose available virtual address to map and return its value in the *address* argument.

`RSM_MAP_FIXED` The import segment should be mapped at the requested virtual address specified in the *address* argument.

The *perm* argument determines whether read, write or a combination of accesses are permitted to the data being mapped. It can be either `RSM_PERM_READ`, `RSM_PERM_WRITE`, or `RSM_PERM_RDWR`.

The *offset* argument is the byte offset location from the base of the segment being mapped to *address*. The *length* argument indicates the number of bytes from offset to be mapped.

The `rsm_memseg_import_unmap()` function unmaps a previously mapped import segment.

Return Values Upon successful completion, these functions return 0. Otherwise, an error value is returned to indicate the error.

Errors The `rsm_memseg_import_map()` and `rsm_memseg_import_unmap()` functions can return the following errors:

`RSMERR_BAD_SEG_HNDL` Invalid segment handle.

The `rsm_memseg_import_map()` function can return the following errors:

RSMERR_BAD_ADDR	Invalid address.
RSMERR_BAD_LENGTH	Invalid length.
RSMERR_BAD_MEM_ALIGNMENT	The address is not aligned on a page boundary.
RSMERR_BAD_OFFSET	Invalid offset.
RSMERR_BAD_PERMS	Invalid permissions.
RSMERR_CONN_ABORTED	Connection aborted.
RSMERR_MAP_FAILED	Map failure.
RSMERR_SEG_ALREADY_MAPPED	Segment already mapped.
RSMERR_SEG_NOT_CONNECTED	Segment not connected.

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

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

See Also rsm_memseg_import_connect(3RSM), rsm_memseg_import_get(3RSM), rsm_memseg_import_put(3RSM), rsm_memseg_get_pollfd(3RSM), attributes(5)

Name rsm_memseg_import_open_barrier, rsm_memseg_import_order_barrier, rsm_memseg_import_close_barrier – remote memory access error detection functions

Synopsis

```
cc [ flag... ] file... –lrsm [ library... ]
#include <rsmapi.h>

int rsm_memseg_import_open_barrier(rsmapi_barrier_t *barrier);

int rsm_memseg_import_order_barrier(rsmapi_barrier_t *barrier);

int rsm_memseg_import_close_barrier(rsmapi_barrier_t *barrier);
```

Description The `rsm_memseg_import_open_barrier()` and `rsm_memseg_import_close_barrier()` functions provide a means of remote memory access error detection when the barrier mode is set to `RSM_BARRIER_MODE_EXPLICIT`. Open and close barrier operations define a span-of-time interval for error detection. A successful close barrier guarantees that remote memory access covered between the open barrier and close barrier have completed successfully. Any individual failures which may have occured between the open barrier and close barrier occur without any notification and the failure is not reported until the close barrier.

The `rsm_memseg_import_order_barrier()` function imposes the order-of-write completion whereby, with an order barrier, the write operations issued before the order barrier are all completed before the operations after the order barrier. Effectively, with the order barrier call, all writes within one barrier scope are ordered with respect to those in another barrier scope.

Return Values Upon successful completion, these functions return 0. Otherwise, an error value is returned to indicate the error.

Errors The `rsm_memseg_import_open_barrier()`, `rsm_memseg_import_order_barrier()`, and `rsm_memseg_import_close_barrier()` functions can return the following errors:

`RSMERR_BAD_SEG_HNDL`	Invalid segment handle
`RSMERR_BAD_BARRIER_PTR`	Invalid barrier pointer.

The `rsm_memseg_close_barrier()` and `rsm_memseg_order_barrier()` functions can return the following errors:

`RSMERR_BARRIER_UNINITIALIZED`	Barrier not initialized.
`RSMERR_BARRIER_NOT_OPENED`	Barrier not opened.
`RSMERR_BARRIER_FAILURE`	Memory access error.
`RSMERR_CONN_ABORTED`	Connection aborted.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE

Interface Stability	Evolving
MT-Level	MT-Safe

See Also rsm_memseg_import_init_barrier(3RSM), rsm_memseg_import_set_mode(3RSM), attributes(5)

Name rsm_memseg_import_put, rsm_memseg_import_put8, rsm_memseg_import_put16, rsm_memseg_import_put32, rsm_memseg_import_put64 – write to a segment

Synopsis

```
cc [ flag... ] file... -lrsm [ library... ]
#include <rsmapi.h>

int rsm_memseg_import_put(rsm_memseg_import_handle_t im_memseg,
     off_t offset, void *src_addr, size_t length);

int rsm_memseg_import_put8(rsm_memseg_import_handle_t im_memseg,
     off_t offset, uint8_t datap, ulong_t rep_cnt);

int rsm_memseg_import_put16(rsm_memseg_import_handle_t im_memseg,
     off_t offset, uint16_t datap, ulong_t rep_cnt);

int rsm_memseg_import_put32(rsm_memseg_import_handle_t im_memseg,
     off_t offset, uint32_t datap, ulong_t rep_cnt);

int rsm_memseg_import_put64(rsm_memseg_import_handle_t im_memseg,
     off_t offset, uint64_t datap, ulong_t rep_cnt);
```

Description When using interconnects that allow memory mapping (see `rsm_memseg_import_map(3RSM)`), standard CPU memory operations may be used for accessing memory of a segment. If, however, a mapping is not provided, then explicitly calling these functions facilitates writing to a segment. Depending on the attributes of the extension library for the interconnect, these functions may involve doing an implicit mapping before performing the data transfer. Applications can be made interconnect-independent with respect to segment writes by using these functions. The data access error detection is performed through the use of barriers (see `rsm_memseg_import_open_barrier(3RSM)`). The default barrier operation mode is `RSM_BARRIER_MODE_IMPLICIT`, which means that around every put operation open and close barrier operations are performed automatically. Explicit error handling may also be set up for these functions (see `rsm_memseg_import_set_mode(3RSM)`).

The `rsm_memseg_import_put()` function copies *length* bytes from local memory with start address *src_addr* to the imported segment *im_memseg* beginning at location *offset* from the start of the segment.

The `rsm_memseg_import_put8()` function copies *rep_cnt* number of 8-bit quantities from successive local memory locations pointed to by *datap* to successive locations starting from *offset* in the imported segment.

The `rsm_memseg_import_put16()` function copies *rep_cnt* number of 16-bit quantities from successive local memory locations pointed to by *datap* to successive locations starting from *offset* in the imported segment. The offset must be aligned at half-word address boundary.

The `rsm_memseg_import_put32()` function copies *rep_cnt* number of 32-bit quantities from successive local memory locations pointed to by *datap* to successive locations starting from *offset* in the imported segment. The offset must be aligned at word address boundary.

The `rsm_memseg_import_put64()` function copies *rep_cnt* number of 64-bit quantities from successive local memory locations pointed to by *datap* to successive locations starting from *offset* in the imported segment. The offset must be aligned at double-word address boundary.

The data transfer functions that transfer small quantities of data (that is, 8-, 16-, 32-, and 64-bit quantities) perform byte swapping prior to the data transfer, in the event that the source and destination have incompatible endian characteristics.

Return Values Upon successful completion, these functions return 0. Otherwise, an error value is returned to indicate the error.

Errors These functions can return the following errors:

`RSMERR_BAD_SEG_HNDL`	Invalid segment handle.
`RSMERR_BAD_ADDR`	Bad address.
`RSMERR_BAD_MEM_ALIGNMENT`	Invalid memory alignment for pointer.
`RSMERR_BAD_OFFSET`	Invalid offset.
`RSMERR_BAD_LENGTH`	Invalid length.
`RSMERR_PERM_DENIED`	Permission denied.
`RSMERR_INSUFFICIENT_RESOURCES`	Insufficient resources.
`RSMERR_BARRIER_UNINITIALIZED`	Barrier not initialized.
`RSMERR_BARRIER_FAILURE`	I/O completion error.
`RSMERR_CONN_ABORTED`	Connection aborted.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

See Also `rsm_memseg_import_get(3RSM)`, `rsm_memseg_import_init_barrier(3RSM)`, `rsm_memseg_import_open_barrier(3RSM)`, `rsm_memseg_import_set_mode(3RSM)`, `attributes(5)`

Name rsm_memseg_import_putv, rsm_memseg_import_getv – write to a segment using a list of I/O requests

Synopsis

```
cc [ flag... ] file... –lrsm [ library... ]
#include <rsmapi.h>

int rsm_memseg_import_putv(rsm_scat_gath_t *sg_io);

int rsm_memseg_import_getv(rsm_scat_gath_t *sg_io);
```

Description The `rsm_memseg_import_putv()` and `rsm_memseg_import_getv()` functions provide for using a list of I/O requests rather than a single source and destination address as is done for the`rsm_memseg_import_put(3RSM)` and `rsm_memseg_import_get(3RSM)` functions.

The I/O vector component of the scatter-gather list (*sg_io*), allows specifying local virtual addresses or local_memory_handles. When a local address range is used repeatedly, it is efficient to use a handle because allocated system resources (that is, locked down local memory) are maintained until the handle is freed. The supporting functions for handles are `rsm_create_localmemory_handle(3RSM)` and `rsm_free_localmemory_handle(3RSM)`.

Virtual addresses or handles may be gathered into the vector for writing to a single remote segment, or a read from a single remote segment may be scattered to the vector of virtual addresses or handles.

Implicit mapping is supported for the scatter-gather type of access. The attributes of the extension library for the specific interconnect are used to determine whether mapping is necessary before any scatter-gather access. If mapping of the imported segment is a prerequisite for scatter-gather access and the mapping has not already been performed, an implicit mapping is performed for the imported segment. The I/O for the vector is then initiated.

I/O for the entire vector is initiated before returning. The barrier mode attribute of the import segment determines if the I/O has completed before the function returns. A barrier mode attribute setting of `IMPLICIT` guarantees that the transfer of data is completed in the order as entered in the I/O vector. An implicit barrier open and close surrounds each list entry. If an error is detected, I/O for the vector is terminated and the function returns immediately. The residual count indicates the number of entries for which the I/O either did not complete or was not initiated.

The number of entries in the I/O vector component of the scatter-gather list is specified in the `io_request_count` field of the `rsm_scat_gath_t` pointed to by *sg_io*. The `io_request_count` is valid if greater than 0 and less than or equal to `RSM_MAX_SGIOREQS`. If `io_request_count` is not in the valid range, `rsm_memseg_import_putv()` and `rsm_memseg_import_getv()` returns `RSMERR_BAD_SGIO`.

Optionally, the scatter-gather list allows support for an implicit signal post after the I/O for the entire vector has completed. This alleviates the need to do an explicit signal post after ever I/O transfer operation. The means of enabling the implicit signal post involves setting the `flags`

field within the scatter-gather list to `RSM_IMPLICIT_SIGPOST`. The `flags` field may also be set to `RSM_SIG_POST_NO_ACCUMULATE`, which will be passed on to the signal post operation when `RSM_IMPLICIT_SIGPOST` is set.

Return Values Upon successful completion, these functions return 0. Otherwise, an error value is returned to indicate the error.

Errors The `rsm_memseg_import_putv()` and `rsm_memseg_import_getv()` functions can return the following errors:

`RSMERR_BAD_SGIO`	Invalid scatter-gather structure pointer.
`RSMERR_BAD_SEG_HNDL`	Invalid segment handle.
`RSMERR_BAD_CTLR_HNDL`	Invalid controller handle.
`RSMERR_BAD_OFFSET`	Invalid offset.
`RSMERR_BAD_LENGTH`	Invalid length.
`RSMERR_BAD_ADDR`	Bad address.
`RSMERR_INSUFFICIENT_RESOURCES`	Insufficient resources.
`RSMERR_INTERRUPTED`	The operation was interrupted by a signal.
`RSMERR_PERM_DENIED`	Permission denied.
`RSMERR_BARRIER_FAILURE`	I/O completion error.
`RSMERR_REMOTE_NODE_UNREACHABLE`	Remote node not reachable.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

See Also `rsm_create_localmemory_handle(3RSM)`, `rsm_free_localmemory_handle(3RSM)`, `attributes(5)`

Name rsm_memseg_import_set_mode, rsm_memseg_import_get_mode – set or get mode for barrier scoping

Synopsis

```
cc [ flag... ] file... –lrsm [ library... ]
#include <rsmapi.h>

int rsm_memseg_import_set_mode(rsm_memseg_import_handle_t memseg,
     rsm_barrier_mode_t mode);

int rsm_memseg_import_get_mode(rsm_memseg_import_handle_t memseg,
     rsm_barrier_mode_t *mode);
```

Description The `rsm_memseg_import_set_mode()` function provides support for optional explicit barrier scoping in the functions described on the `rsm_memseg_import_get(3RSM)` and `rsm_memseg_import_put(3RSM)` manual pages.. The two valid barrier modes are `RSM_BARRIER_MODE_EXPLICIT` and `RSM_BARRIER_MODE_IMPLICIT`. By default, the barrier mode is set to `RSM_BARRIER_MODE_IMPLICIT`. When the mode is `RSM_BARRIER_MODE_IMPLICIT`, an implicit barrier open and barrier close is applied to the put operation. Irrespective of the mode set, the barrier must be initialized using the `rsm_memseg_import_init_barrier(3RSM)` function before any barrier operations, either implicit or explicit, are used.

The `rsm_memseg_import_get_mode()` function obtains the current value of the mode used for barrier scoping in put functions.

Return Values Upon successful completion, these functions return 0. Otherwise, an error value is returned to indicate the error.

Errors The `rsm_memseg_import_set_mode()` and `rsm_memseg_import_get_mode()` functions can return the following errors:

`RSMERR_BAD_SEG_HNDL` Invalid segment handle.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

See Also `rsm_memseg_import_get(3RSM)`, `rsm_memseg_import_init_barrier(3RSM)`, `rsm_memseg_import_put(3RSM)`, `attributes(5)`

Name rtld_audit, la_activity, la_i86_pltenter, la_objsearch, la_objopen, la_objfilter, la_pltexit, la_pltexit64, la_preinit, la_sparcv8_pltenter, la_sparcv9_pltenter, la_amd64_pltenter, la_symbind32, la_symbind64, la_version – runtime linker auditing functions

Synopsis

```
void la_activity(uintptr_t *cookie, uint_t flag);

uintptr_t la_i86_pltenter(Elf32_Sym *sym, uint_t ndx, uintptr_t *refcook,
     uintptr_t *defcook, La_i86_regs *regs, uint_t *flags);

char *la_objsearch(const char *name, uintptr_t *cookie, uint_t flag);

uint_t la_objopen(Link_map  *lmp, Lmid_t lmid, uintptr_t *cookie);

int la_objfilter(uintptr_t  *fltrcook, uintptr_t *fltecook,
     uint_t *flags);

uintptr_t la_pltexit(Elf32_Sym *sym, uint_t ndx, uintptr_t *refcook,
     uintptr_t *defcook, uintptr_t retval);

uintptr_t la_pltexit64(Elf64_Sym *sym, uint_t ndx, uintptr_t *refcook,
     uintptr_t *defcook, uintptr_t retval, const char *sym_name);

void la_preinit(uintptr_t *cookie);

uintptr_t la_sparcv8_pltenter(Elf32_Sym *sym, uint_t ndx,
     uintptr_t *refcook, uintptr_t *defcook, La_amd64_regs *regs,
     uint_t *flags);

uintptr_t la_sparcv9_pltenter(Elf64_Sym *sym, uint_t ndx,
     uintptr_t *refcook, uintptr_t *defcook, La_sparcv8_regs *regs,
     uint_t *flags, const char *sym_name);

uintptr_t la_amd64_pltenter(Elf32_Sym *sym, uint_t ndx,
     uintptr_t *refcook, uintptr_t *defcook, La_sparcv8_regs *regs,
     uint_t *flags, const char *sym_name);

uintptr_t la_symbind32(Elf32_Sym *sym, uint_t ndx, uintptr_t *refcook,
     uintptr_t *defcook, uint_t *flags);

uintptr_t la_symbind64(Elf64_Sym *sym, uint_t ndx,
     uintptr_t *refcook,uintptr_t *defcook, uint_t *flags,
     const char *sym_name);

uint_t la_version(uint_t version);
```

Description A runtime linker auditing library is a user-created shared object offering one or more of these interfaces. The runtime linker `ld.so.1(1)`, calls these interfaces during process execution. See the *Linker and Libraries Guide* for a full description of the link auditing mechanism.

See Also `ld.so.1(1)`

Linker and Libraries Guide

Name rtld_db, rd_delete, rd_errstr, rd_event_addr, rd_event_enable, rd_event_getmsg, rd_init, rd_loadobj_iter, rd_log, rd_new, rd_objpad_enable, rd_plt_resolution, rd_reset – runtime linker debugging functions

Synopsis

```
cc [ flag ... ] file ... -lrtld_db [ library ... ]
#include <proc_service.h>
#include <rtld_db.h>

void rd_delete(struct rd_agent *rdap);

char *rd_errstr(rd_err_e rderr);

rd_err_e rd_event_addr(rd_agent *rdap, rd_notify_t *notify);

rd_err_e rd_event_enable(struct rd_agent *rdap, int onoff);

rd_err_e rd_event_getmsg(struct rd_agent *rdap,
     rd_event_msg_t *msg);

rd_err_e rd_init(int version);

typedef int rl_iter_f(const rd_loadobj_t *, void *);

rd_err_e rd_loadobj_iter(rd_agent_t *rap, rl_iter_f *cb,
     void *clnt_data);

void rd_log(const int onoff);

rd_agent_t *rd_new(struct ps_prochandle *php);

rd_err_e rd_objpad_enable(struct rd_agent *rdap, size_t padsize);

rd_err_e rd_plt_resolution(rd_agent *rdap, paddr_t pc,
     lwpid_t lwpid, paddr_t plt_base, rd_plt_info_t *rpi);

rd_err_e rd_reset(struct rd_agent *rdap);
```

Description The `librtld_db` library provides support for monitoring and manipulating runtime linking aspects of a program. There are at least two processes involved, the controlling process and one or more target processes. The controlling process is the `librtld_db` client that links with `librtld_db` and uses `librtld_db` to inspect or modify runtime linking aspects of one or more target processes. See the *Linker and Libraries Guide* for a full description of the runtime linker debugger interface mechanism.

Usage To use `librtld_db`, applications need to implement the interfaces documented in `ps_pread(3PROC)` and `proc_service(3PROC)`.

Attributes See `attributes(5)` for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also `ld.so.1`(1), `libc_db`(3LIB), `librtld_db`(3LIB), `proc_service`(3PROC), `ps_pread`(3PROC), `attributes`(5)

Linker and Libraries Guide

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 scalb, scalbf, scalbl – load exponent of a radix-independent floating-point number

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double scalb(double x, double n);

float scalbf(float x, float n);

long double scalbl(long double x, long double n);
```

Description These functions compute x * r^n, where *r* is the radix of the machine's floating point arithmetic. When *r* is 2, `scalb()` is equivalent to `ldexp(3M)`. The value of *r* is `FLT_RADIX` which is defined in `<float.h>`.

Return Values Upon successful completion, the `scalb()` function returns $x * r^n$.

If *x* or *n* is NaN, a NaN is returned.

If *n* is 0, *x* is returned.

If *x* is ±Inf and *n* is not –Inf, *x* is returned.

If *x* is ±0 and *n* is not +Inf, *x* is returned.

If *x* is ±0 and *n* is +Inf, a domain error occurs and a NaN is returned.

If *x* is ±Inf and *n* is –Inf, a domain error occurs and a NaN is returned.

If the result would cause an overflow, a range error occurs and ±`HUGE_VAL` (according to the sign of *x*) is returned.

For exceptional cases, `matherr(3M)` tabulates the values to be returned by `scalb()` as specified by SVID3 and XPG3. See `standards(5)`.

Errors These functions will fail if:

Domain Error — If *x* is 0 and *n* is +Inf, or *x* is Inf and *n* is –Inf.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, then the invalid floating-point exception is raised.

Range Error — The result would overflow.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, then the overflow floating-point exception is raised.

Usage An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. An application should either examine the return value or check the floating point exception flags to detect

exceptions.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	See below.
MT-Level	MT-Safe

The `scalb()` function is Standard. The `scalbf()` and `scalbl()` functions are Stable.

See Also feclearexcept(3M), fetestexcept(3M), ilogb(3M), ldexp(3M), logb(3M), math.h(3HEAD), matherr(3M), scalbln(3M), attributes(5), standards(5)

Name scalbln, scalblnf, scalblnl, scalbn, scalbnf, scalbnl – compute exponent using FLT_RADIX

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double scalbln(double x, long n);

float scalblnf(float x, long n);

long double scalblnl(long double x, long n);

double scalbn(double x, int n);

float scalbnf(float x, int n);

long double scalbnl(long double x, int n);
```

Description These functions compute $x * \texttt{FLT\_RADIX}^{n}$ efficiently, not normally by computing $\texttt{FLT\_RADIX}^{n}$ explicitly.

Return Values Upon successful completion, these functions return $x * \texttt{FLT\_RADIX}^{n}$>.

If the result would cause overflow, a range error occurs and these functions return ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL (according to the sign of *x*) as appropriate for the return type of the function.

If *x* is NaN, a NaN is returned.

If *x* is ±0 or ±Inf, *x* is returned.

If *x* is 0, *x* is returned.

Errors These functions will fail if:

Range Error — The result overflows.

If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception is raised.

Usage An application wanting to check for exceptions should call feclearexcept(FE_ALL_EXCEPT) before calling these functions. On return, if fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an exception has been raised. An application should either examine the return value or check the floating point exception flags to detect exceptions.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also feclearexcept(3M), fetestexcept(3M), math.h(3HEAD), scalb(3M), attributes(5), standards(5)

Name SCF_Card_exchangeAPDU – send a command APDU to a card and read the card's response

Synopsis

```
cc [ flag... ] file... -lsmartcard [ library...]
#include <smartcard/scf.h>

SCF_Status_t SCF_Card_exchangeAPDU(SCF_Card_t card,
     const uint8_t *sendBuffer, size_t sendLength,
     uint8_t *recvBuffer, size_t *recvLength);
```

Parameters

card The card (from `SCF_Terminal_getCard(3SMARTCARD)`) to communicate with.

sendBuffer A pointer to a buffer containing the command APDU.

sendLength The number of bytes in the sendBuffer (that is, the size of the command APDU).

recvBuffer A pointer to a buffer in which the card's reply APDU should be stored. This buffer can be the same as the *sendBuffer* to allow the application to conserve memory usage. The buffer must be large enough to store the expected reply.

recvLength The caller specifies the maximum size of the recvBuffer in *recvLength*. The library uses this value to prevent overflowing the buffer. When the reply is received, the library sets *recvLength* to the actual size of the reply APDU that was stored in the *recvBuffer*.

Description The `SCF_Card_exchangeAPDU()` function sends a binary command to the card and reads the reply. The application is responsible for constructing a valid command and providing a receive buffer large enough to hold the reply. Generally, the command and reply will be ISO7816-formatted APDUs (Application Protocol Data Units), but the SCF library does not examine or verify the contents of the buffers.

If the caller needs to perform a multi-step transaction that must not be interrupted, `SCF_Card_lock(3SMARTCARD)` should be used to prevent other applications from communicating with the card during the transaction. Similarly, calls to `SCF_Card_exchangeAPDU()` must be prepared to retry the call if `SCF_STATUS_CARDLOCKED` is returned.

An ISO7816-formatted command APDU always begins with a mandatory 4 byte header (CLA, INS, P1, and P2), followed by a variable length body (zero or more bytes). For details on the APDUs supported by a specific card, consult the documentation provided by the card manufacturer or applet vendor.

An ISO7816-formatted reply APDU consists of zero or more bytes of data, followed by a manditory 2 byte status trailer (SW1 and SW2).

Return Values If the APDU is successfully sent and a reply APDU is successfully read, `SCF_STATUS_SUCCESS` is returned with *recvBuffer* and *recvLength* set appropriately. Otherwise, an error value is returned and both *recvBuffer* and *recvLength* remain unaltered.

Errors The `SCF_Card_exchangeAPDU()` function will fail if:

`SCF_STATUS_BADARGS`	Neither *sendBuffer*, *recvBuffer*, nor *recvLength* can be null pointers. The value of recvLength must be at least 2.
`SCF_STATUS_BADHANDLE`	The card has been closed or is invalid.
`SCF_STATUS_CARDLOCKED`	The APDU cannot be sent because the card is locked by another application.
`SCF_STATUS_CARDREMOVED`	The card object cannot be used because the card represented by the `SCF_Card_t` has been removed
`SCF_STATUS_COMMERROR`	The connection to the server was closed.
`SCF_STATUS_FAILED`	An internal error occurred.
`SCF_STATUS_NOSPACE`	The specified size of *recvBuffer* is too small to hold the complete reply APDU.

Examples **EXAMPLE 1** Send a command to the card.

```
SCF_Status_t status;
SCF_Card_t myCard;
uint8_t commandAPDU[] = {0x00, 0xa4, 0x00, 0x00, 0x02, 0x3f, 0x00};
uint8_t replyAPDU[256];
uint32_t commandSize = sizeof(commandAPDU);
uint32_t replySize = sizeof(replyAPDU);
/* (...call SCF_Terminal_getCard to open myCard...) */

/* Send the ISO7816 command to select the card's MF. */
status = SCF_Card_exchangeAPDU(myCard, commandAPDU, commandSize,
    replyAPDU, &replySize);
if (status != SCF_STATUS_SUCCESS) exit(1);

printf("Received a %d byte reply.\n", replySize);
printf("SW1=0x%02.2x SW2=0x%02.2x\n",
    replyAPDU[replySize-2], replyAPDU[replySize-1]);

/* ... */
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

See Also libsmartcard(3LIB), SCF_Card_lock(3SMARTCARD), SCF_Terminal_getCard(3SMARTCARD), attributes(5)

Name SCF_Card_lock, SCF_Card_unlock – perform mutex locking on a card

Synopsis

```
cc [ flag... ] file... -lsmartcard [ library...]
#include <smartcard/scf.h>

SCF_Status_t SCF_Card_lock(SCF_Card_t card, unsigned int timeout);

SCF_Status_t SCF_Card_unlock(SCF_Card_t card);
```

Parameters

card The card (from SCF_Terminal_getCard(3SMARTCARD)) to be locked.

timeout The maximum number of seconds SCF_Card_lock() should wait for a card locked by another application to become unlocked. A value of 0 results in SCF_Card_lock() returning immediately if a lock cannot be immediately obtained. A value of SCF_TIMEOUT_MAX results in SCF_Card_lock() waiting forever to obtain a lock.

Description Locking a card allows an application to perform a multi-APDU transaction (that is, multiple calls to SCF_Card_exchangeAPDU(3SMARTCARD)) without interference from other smartcard applications. The lock is enforced by the server, so that other applications that attempt to call SCF_Card_exchangeAPDU() or SCF_Card_reset(3SMARTCARD) will be denied access to the card. Applications should restrict use of locks only to brief critical sections. Otherwise it becomes difficult for multiple applications to share the same card.

When a lock is granted to a specific SCF_Card_t card object, only that object can be used to access the card and subsquently release the lock. If a misbehaving application holds a lock for an extended period, the lock can be broken by having the user remove and reinsert the smartcard.

It is an error to attempt to lock a card when the caller already holds a lock on the card (that is, calling SCF_Card_lock() twice in a succession). Unlocking a card that is not locked (or was already unlocked) can be performed without causing an error.

An application might find that it is unable to lock the card, or communicate with it because SCF_Card_exchangeAPDU() keeps returning SCF_STATUS_CARDLOCKED. If this situation persists, it might indicate that another application has not released its lock on the card. The user is able to forcably break a lock by removing the card and reinserting it, after which the application must call SCF_Terminal_getCard(3SMARTCARD) to access the "new" card. In this situation an application should retry for a reasonable period of time, and then alert the user that the operation could not be completed because the card is in use by another application and that removing or reinserting the card will resolve the problem.

Return Values If the card is successfully locked or unlocked, SCF_STATUS_SUCCESS is returned. Otherwise, the lock status of the card remains unchanged and an error value is returned.

Errors The SCF_Card_lock() and SCF_Card_lock() functions will fail if:

SCF_STATUS_BADHANDLE The specified card has been closed or is invalid.

SCF_STATUS_CARDLOCKED	There is a lock present on the card, but it is not held by the specified card object. For example, the caller is attempting to unlock a card locked by another application.
SCF_STATUS_CARDREMOVED	The card object cannot be used because the card represented by the SCF_Card_t has been removed.
SCF_STATUS_COMMERROR	The connection to the server was lost.
SCF_STATUS_DOUBLELOCK	The caller has already locked this card and is attempting to lock it again.
SCF_STATUS_FAILED	An internal error occured.
SCF_STATUS_TIMEOUT	The *timeout* expired before the call was able to obtain the lock.

Examples

EXAMPLE 1 Use a card lock.

```
SCF_Status_t status;
SCF_Card_t myCard;

/* (...call SCF_Terminal_getCard to open myCard...) */

status = SCF_Card_lock(myCard, 15);
if (status == SCF_STATUS_TIMEOUT) {
    printf("Unable to get a card lock, someone else has a lock.\n");
    exit(0);
}
else if (status != SCF_STATUS_SUCCESS) exit(1);

/* Send the first APDU */
SCF_Card_exchangeAPDU(myCard, ...);

/* Send the second APDU */
SCF_Card_exchangeAPDU(myCard, ...);

status = SCF_Card_unlock(myCard);

/* ... */
```

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

See Also libsmartcard(3LIB), SCF_Card_exchangeAPDU(3SMARTCARD), SCF_Card_reset(3SMARTCARD), SCF_Terminal_getCard(3SMARTCARD), attributes(5)

Name SCF_Card_reset – perform a reset of a smartcard

Synopsis

```
cc [ flag... ] file... -lsmartcard [ library...]
#include <smartcard/scf.h>

SCF_Status_t SCF_Card_reset(SCF_Card_t card);
```

Parameters *card* The card (from SCF_Terminal_getCard(3SMARTCARD)) to be reset

Description The SCF_Card_reset() function causes the specified smartcard to be reset by the terminal.

A card can be reset only if it has not been locked (with SCF_Card_lock(3SMARTCARD)) by another client. A client wishing to reset a card should either first call SCF_Card_lock() to obtain the card lock, or be prepared to retry the reset operation if it fails because another client holds the card lock.

When the card is reset, any SCF_Card_t object representing the card will continue to remain valid after the reset. When the reset occurs, an SCF_EVENT_CARDRESET event will be sent to all registered event listeners for the terminal (assuming they registered for this event). This is the only notification of a reset provided to SCF clients. When a client receives this event, it should be prepared to reinitialize any state on the card that might have been interrupted by the reset. New information about the card (for example, ATR, if it changed) can also be available from SCF_Card_getInfo(3SMARTCARD).

Return Values If the card is successfully reset, SCF_STATUS_SUCCESS is returned. Otherwise, the status of the card remains unchanged and an error value is returned.

Errors The SCF_Card_reset() function will fail if:

SCF_STATUS_BADHANDLE	The specified card has been closed or is invalid.
SCF_STATUS_CARDLOCKED	The card cannot be reset because another client holds a lock on the card.
SCF_STATUS_CARDREMOVED	The card cannot be reset because the card represented by the SCF_Card_t has been removed.
SCF_STATUS_COMMERROR	The connection to the server was lost.
SCF_STATUS_FAILED	An internal error occured.

Examples **EXAMPLE 1** Reset a card.

```
SCF_Status_t status;
SCF_Card_t myCard;

/* (...call SCF_Terminal_getCard to open myCard...) */

status = SCF_Card_lock(myCard, SCF_TIMEOUT_MAX);
if (status != SCF_STATUS_SUCCESS) exit(1);
```

EXAMPLE 1 Reset a card. *(Continued)*

```
status = SCF_Card_reset(myCard);
if (status != SCF_STATUS_SUCCESS) exit(1);

status = SCF_Card_unlock(myCard);
if (status != SCF_STATUS_SUCCESS) exit(1);

/* ... */
```

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

See Also

libsmartcard(3LIB), SCF_Card_getInfo(3SMARTCARD), SCF_Card_lock(3SMARTCARD), SCF_Terminal_addEventListener(3SMARTCARD), SCF_Terminal_getCard(3SMARTCARD), attributes(5)

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.

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_IN_USE` The value has been added to another entry.

`SCF_ERROR_NOT_SET` The transaction entry is not associated with a transaction.

`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_HANDLE_MISMATCH` The *value* and *entry* arguments are not derived from the same handle.

`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	Evolving
MT-Level	Safe

See Also `libscf(3LIB)`, `scf_error(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>

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 *inst,
     char *buffer, size_t sz);

ssize_t scf_pg_to_fmri(const scf_scope_t *object, char *buffer,
     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_NO_RESOURCES`	The server does not have adequate resources to complete the request.
`SCF_ERROR_INVALID_ARGUMENT`	The *fmri* argument is not a valid FMRI.
`SCF_ERROR_CONSTRAINT_VIOLATED`	The FMRI does not meet the restrictions requested in the flag argument.
`SCF_ERROR_NOT_FOUND`	The FMRI is well-formed but there is no object in the repository matching it.
`SCF_ERROR_NOT_BOUND`	The handle is not currently bound.
`SCF_ERROR_CONNECTION_BROKEN`	The connection to the repository was lost.
`SCF_ERROR_HANDLE_MISMATCH`	One or more of the entity handles was not derived from handle.

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_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_NO_RESOURCES`
The server does not have the resources to complete the request.

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_NOT_FOUND`
No instance specified by *name* was found.

`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.

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_pg_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 of *parent* of a particular type.

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_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_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) {
          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);
```

EXAMPLE 2 Connect to the repository, walk all services and instances and print their FMRIs. *(Continued)*

```
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	Evolving
MT-Level	Safe

See Also

libscf(3LIB), 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, 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_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);

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_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.

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_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.
`SCF_ERROR_CONNECTION_BROKEN`	The connection to the repository was lost.

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_NOT_SET`	The property group specified by *pg* is not set.
`SCF_ERROR_NOT_FOUND`	The property specified by *name* was not found.
`SCF_ERROR_INVALID_ARGUMENT`	The value of the *name* argument is not a valid property name.
`SCF_ERROR_HANDLE_MISMATCH`	The property group and property are not derived from the same handle.

SCF_ERROR_CONNECTION_BROKEN	The connection to the repository was lost.
SCF_ERROR_NOT_BOUND	The handle was never bound or has been unbound.
SCF_ERROR_DELETED	The property group or an ancestor has been deleted.

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_NOT_FOUND	The property has no values associated with it. The *value* argument will be reset.
SCF_ERROR_HANDLE_MISMATCH	The property and value are derived from different handles.

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_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.

The `scf_scope_add_service()` function will fail if:

`SCF_ERROR_EXISTS`	A {service,instance,property group} named *name* already exists.

`SCF_ERROR_DELETED` The parent entity has been deleted.

The `scf_scope_add_service()` and `scf_scope_get_service()` functions will fail if:

`SCF_ERROR_NO_RESOURCES` The server does not have the resources to complete the request.

`SCF_ERROR_NOT_SET` The scope is not set.

`SCF_ERROR_NOT_FOUND` The service specified by *name* was not found.

`SCF_ERROR_INVALID_ARGUMENT` The value of the *name* argument is not a valid service name.

`SCF_ERROR_HANDLE_MISMATCH` The scope and service 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_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	Evolving
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_Session_close, SCF_Terminal_close, SCF_Card_close – close a smartcard session, terminal, or card

Synopsis

```
cc [ flag... ] file... -lsmartcard [ library...]
#include <smartcard/scf.h>

SCF_Status_t SCF_Session_close(SCF_Session_t session);

SCF_Status_t SCF_Terminal_close(SCF_Terminal_t terminal);

SCF_Status_t SCF_Card_close(SCF_Card_t card);
```

Parameters

card An object that was returned from SCF_Terminal_getCard(3SMARTCARD)

session An object that was returned from SCF_Session_getSession(3SMARTCARD)

terminal An object that was returned from SCF_Session_getTerminal(3SMARTCARD)

Description These functions release the resources (memory, threads, and others) that were allocated within the library when the session, terminal, or card was opened. Any storage allocated by calls to SCF_Session_getInfo(3SMARTCARD), SCF_Session_getInfo(3SMARTCARD), or SCF_Card_getInfo(3SMARTCARD) is deallocated when the associated object is closed. Attempts to access results from these interfaces after the object has been closed results in undefined behavior.

If a card that was locked by SCF_Card_lock(3SMARTCARD) is closed, the lock is automatically released. When a terminal is closed, any event listeners on that terminal object are removed and any cards that were obtained with the terminal are closed. Similarly, closing a session will close any terminals or cards obtained with that session. These are the only cases where the library will automatically perform a close.

Once closed, a session, terminal, or card object can no longer be used by an SCF function. Any attempt to do so results in an SCF_STATUS_BADHANDLE error. The sole exception is that closing an object, even if already closed, is always a successful operation.

Return Values Closing a handle is always a successful operation that returns SCF_STATUS_SUCCESS. The library can safely detect handles that are invalid or already closed.

Examples **EXAMPLE 1** Close each object explicitly.

```
SCF_Status_t status;
SCF_Session_t mySession;
SCF_Terminal_t myTerminal;
SCF_Card_t myCard;

status = SCF_Session_getSession(&mySession);
if (status != SCF_STATUS_SUCCESS) exit(1);
status = SCF_Session_getTerminal(mySession, NULL, &myTerminal);
if (status != SCF_STATUS_SUCCESS) exit(1);
status = SCF_Terminal_getCard(myTerminal, &myCard);
```

EXAMPLE 1 Close each object explicitly. *(Continued)*

```
if (status != SCF_STATUS_SUCCESS) exit(1);

/* (Do interesting things with smartcard...) */

SCF_Card_close(myCard);
SCF_Terminal_close(myTerminal);
SCF_Session_close(mySession);
```

EXAMPLE 2 Allow the library to close objects.

```
SCF_Status_t status;
SCF_Session_t mySession;
SCF_Terminal_t myTerminal;
SCF_Card_t myCard;

status = SCF_Session_getSession(&mySession);
if (status != SCF_STATUS_SUCCESS) exit(1);
status = SCF_Session_getTerminal(mySession, NULL, &myTerminal);
if (status != SCF_STATUS_SUCCESS) exit(1);
status = SCF_Terminal_getCard(myTerminal, &myCard);
if (status != SCF_STATUS_SUCCESS) exit(1);

/* (Do interesting things with smartcard...) */

SCF_Session_close(mySession);
/* myTerminal and myCard have been closed by the library. */
```

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

See Also

libsmartcard(3LIB), SCF_Card_getInfo(3SMARTCARD), SCF_Card_lock(3SMARTCARD), SCF_Session_getInfo(3SMARTCARD), SCF_Session_getSession(3SMARTCARD), SCF_Session_getTerminal(3SMARTCARD), SCF_Terminal_getCard(3SMARTCARD), SCF_Session_getInfo(3SMARTCARD), attributes(5)

Name SCF_Session_freeInfo, SCF_Terminal_freeInfo, SCF_Card_freeInfo – deallocate information storage

Synopsis

```
cc [ flag... ] file... -lsmartcard [ library...]
#include <smartcard/scf.h>

SCF_Status_t SCF_Session_freeInfo(SCF_Session_t session, void *value);

SCF_Status_t SCF_Terminal_freeInfo(SCF_Terminal_t terminal, void *value);

SCF_Status_t SCF_Card_freeInfo(SCF_Card_t card, void *value);
```

Parameters

card An object that was returned from SCF_Terminal_getCard(3SMARTCARD). This object must be associated with the information value being freed.

session An object that was returned from SCF_Session_getSession(3SMARTCARD). This object must be associated with the information value being freed.

terminal An object that was returned from SCF_Session_getTerminal(3SMARTCARD). This object must be associated with the information value being freed.

value A pointer that was returned from a call to SCF_Session_getInfo(3SMARTCARD), SCF_Session_getInfo(3SMARTCARD), or SCF_Card_getInfo(3SMARTCARD).

Description When information is requested for an object (for example, by using SCF_Session_getInfo()), the result is placed in memory allocated for that request. This memory must eventually be deallocated, or a memory leak will result. The deallocation of memory can occur in one of two ways.

- The simplest method is to allow the smartcard library to automatically deallocate memory when the object associated with the information is closed. For example, when SCF_Card_close(3SMARTCARD) is called, any information obtained from SCF_Card_getInfo() for that card object is deallocated. The application is not required to call SCF_Card_freeInfo() at all.
- If the object persists for a long period of time, the application can explicitly request the information to be deallocated without closing the object, so that memory is not wasted on unneeded storage. Similarly, if an application repeatedly requests information about an object (even the same information), the application can explicitly request deallocation as needed, so that memory usage does not continue to increase until the object is closed. In general, requesting information to be deallocated can be used to reduce runtime memory bloat.

Attempts to access deallocated memory result in undefined behavior.

Return Values If the information is successfully deallocated, SCF_STATUS_SUCCESS is returned. Otherwise, an error value is returned.

Errors These functions will fail if:

`SCF_STATUS_BADARGS`	The specified value cannot be deallocated, possibly because of an invalid pointer, a value already deallocated, or because the value is not associated with the specified session, terminal, or card.
`SCF_STATUS_BADHANDLE`	The specified session, terminal, or card has been closed or is invalid.
`SCF_STATUS_FAILED`	An internal error occured.

Examples **EXAMPLE 1** Free information.

```
char *terminalName;
SCF_Status_t status;
SCF_Terminal_t myTerminal;

/* (...call SCF_Session_getTerminal to open myTerminal...) */

status = SCF_Terminal_getInfo(myTerminal, "name", &terminalName);
if (status != SCF_STATUS_SUCCESS) exit(1);

printf("The terminal name is %s\n", terminalName);

status = SCF_Terminal_freeInfo(myTerminal, terminalName);
if (status != SCF_STATUS_SUCCESS) exit(1);
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

See Also `libsmartcard(3LIB)`, `SCF_Session_getInfo(3SMARTCARD)`, `SCF_Session_getSession(3SMARTCARD)`, `SCF_Session_getTerminal(3SMARTCARD)`, `SCF_Terminal_getCard(3SMARTCARD)`, `attributes(5)`

Name SCF_Session_getInfo, SCF_Terminal_getInfo, SCF_Card_getInfo – retrieve information about a session, terminal, or card

Synopsis

```
cc [ flag... ] file... -lsmartcard [ library...]
#include <smartcard/scf.h>

SCF_Status_t SCF_Session_getInfo(SCF_Session_t session, const char *name,
     void *value);

SCF_Status_t SCF_Terminal_getInfo(SCF_Terminal_t terminal,
     const char *name, void *value);

SCF_Status_t SCF_Card_getInfo(SCF_Card_t card, const char *name,
     void *value);
```

Parameters

card An object that was returned from `SCF_Terminal_getCard(3SMARTCARD)`.

name The name of a property for which a value is to be returned. The name is case-sensitive.

session An object that was returned from `SCF_Session_getSession(3SMARTCARD)`.

terminal An object that was returned from `SCF_Session_getTerminal(3SMARTCARD)`.

value The value of the property. The actual type of the value depends on what property was being queried.

Description These functions obtain information about a session, terminal, or card. The information returned represents the current state of the object and can change between calls.

Each call allocates new storage for the returned result. This storage is tracked internally and is deallocated when the object is closed. An application repeatedly asking for information can cause memory bloat until the object is closed. The application can optionally call `SCF_Session_freeInfo(3SMARTCARD)`, `SCF_Terminal_freeInfo(3SMARTCARD)`, or `SCF_Card_freeInfo(3SMARTCARD)` to cause immediate deallocation of the value. Applications must not use other means such as`free(3C)` to deallocate the memory.

Applications must not access values that have been deallocated. For example, accessing a Card's ATR after the card has been closed results in undefined behavior.

For a session, the valid property names and value types are:

terminalnames (pointer to `char **`) The list of terminal names that can currently be used in this session. The returned value is an array of `char *`, each element of the list is a pointer to a terminal name. The end of the array is denoted by a null pointer. The first element of the list is the default terminal for the session, which will be used when `SCF_Session_getTerminal()` is called with a null pointer for the terminal name.

For a terminal, the standard property names and value types are as follows. Some terminal drivers can define additional driver-specific properties.

name (pointer to `char *`)	The name of the terminal. If the default terminal was used (a null pointer was passed to `SCF_Session_getTerminal()`), the value will contain the actual name of the default terminal. For example, "MyInternalCardReader".
type (pointer to `char *`)	The type of the terminal. For example, "SunISCRI".
devname (pointer to `char *`)	Information about how the device is attached to the system. This can be a UNIX device name (for example, "/dev/scmi2c0") or some other terminal-specific string describing its relation to the system.

For a card, the valid property names and value types are:

type (pointer to `char *`)	The type of the smartcard, as recognized by the framework (For example, "Cyberflex"). If the framework does not recognize the card type, "UnknownCard" is returned.
atr (pointer to `struct SCF_BinaryData_t *`)	The Answer To Reset (ATR) data returned by the card when it was last inserted or reset. The structure member `length` denotes how many bytes are in the ATR. The structure member `data` is a pointer to the actual ATR bytes.

Return Values Upon success, `SCF_STATUS_SUCCESS` is returned and *value* will contain the requested information. Otherwise, an error value is returned and *value* remains unaltered.

Errors These functions will fail if:

`SCF_STATUS_BADARGS`	Either *name* or *value* is a null pointer.
`SCF_STATUS_BADHANDLE`	The session, terminal, or card has been closed or is invalid.
`SCF_STATUS_FAILED`	An internal error occurred.
`SCF_STATUS_UNKNOWNPROPERTY`	The property specified by *name* was not found.

Examples **EXAMPLE 1** Simple string information.

```
SCF_Status_t status;
SCF_Terminal_t myTerminal;
const char *myName, *myType;

/* (...call SCF_Session_getTerminal to open myTerminal...) */
```

EXAMPLE 1 Simple string information. *(Continued)*

```
status = SCF_Terminal_getInfo(myTerminal, "name", &myName);
if (status != SCF_STATUS_SUCCESS) exit(1);
status = SCF_Terminal_getInfo(myTerminal, "type", &myType);
if (status != SCF_STATUS_SUCCESS) exit(1);

printf("The terminal called %s is a %s\n", myName, myType);
```

EXAMPLE 2 Display the names of all terminals available in the session.

```
SCF_Status_t status;
SCF_Session_t mySession;
const char **myList;  /* Technically "const char * const *". */
int i;

/* (...call SCF_Session_getSession to open mySession...) */

status = SCF_Session_getInfo(mySession, "terminalnames", &myList);
if (status != SCF_STATUS_SUCCESS) exit(1);

printf("The following terminals are available:\n");
for (i=0; myList[i] != NULL; i++) {
    printf("%d: %s\n", i, myList[i]);
}
```

EXAMPLE 3 Display the card's ATR.

```
SCF_Status_t status;
SCF_Card_t myCard;
struct SCF_BinaryData_t *myATR;
int i;

/* (...call SCF_Terminal_getCard to open myCard...) */

status = SCF_Card_getInfo(myCard, "atr", &myATR);
if (status != SCF_STATUS_SUCCESS) exit(1);

printf("The card's ATR is: 0x");
for(i=0; i < myATR->length; i++) {
    printf("%02.2x", myATR->data[i]);
}
printf("\n");
```

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

See Also libsmartcard(3LIB), SCF_Session_freeInfo(3SMARTCARD), SCF_Session_getSession(3SMARTCARD), SCF_Session_getTerminal(3SMARTCARD), SCF_Terminal_getCard(3SMARTCARD), attributes(5)

Name SCF_Session_getSession – establish a context with a system's smartcard framework

Synopsis

```
cc [ flag... ] file... -lsmartcard [ library...]
#include <smartcard/scf.h>

SCF_Status_t SCF_Session_getSession(SCF_Session_t *session);
```

Parameters

session A pointer to an `SCF_Session_t`. If a session is successfully established, the session will be returned through this parameter.

Description The `SCF_Session_getSession()` function establishes a session with the Solaris Smart Card Framework (SCF). Once a session has been opened, the session can be used with `SCF_Session_getTerminal(3SMARTCARD)` to access a smartcard terminal (reader). Information about the session can be obtained by calling `SCF_Session_getInfo(3SMARTCARD)`.

When the session is no longer needed, `SCF_Session_close(3SMARTCARD)` should be called to end the session and release session resources. Closing a session will also close any terminals and cards opened within the session.

An application usually needs to open only a single session. For example, multiple terminals can be opened from the same session. If an appication opens additional sessions, each call will return independent (different) sessions.

Return Values Upon success, `SCF_STATUS_SUCCESS` is returned and *session* contains a valid session. If a session could not be established, an error value is returned and *session* remains unaltered.

Errors The `SCF_Session_getSession()` function will fail if:

`SCF_STATUS_BADARGS`	The *session* argument is a null pointer.
`SCF_STATUS_COMMERROR`	The library was unable to contact the smartcard server daemon (`ocfserv(1M)`), or the library was unable to obtain a session from the server.
`SCF_STATUS_FAILED`	An internal error occurred.

Examples **EXAMPLE 1** Establish a session with the framework.

```
SCF_Status_t status;
SCF_Session_t mySession;

status = SCF_Session_getSession(&mySession);
if (status != SCF_STATUS_SUCCESS) exit(1);

/* Proceed with other smartcard operations. */
```

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

See Also libsmartcard(3LIB), SCF_Session_close(3SMARTCARD), SCF_Session_getInfo(3SMARTCARD), SCF_Session_getTerminal(3SMARTCARD), attributes(5)

Name SCF_Session_getTerminal – establish a context with a smartcard terminal (reader)

Synopsis

```
cc [ flag... ] file... -lsmartcard [ library...]
#include <smartcard/scf.h>

SCF_Status_t SCF_Session_getTerminal(SCF_Session_t session,
     const char *terminalName, SCF_Terminal_t *terminal);
```

Parameters

session The session (from `SCF_Session_getSession(3SMARTCARD)`) containing a terminal to be opened.

terminal A pointer to an `SCF_Terminal_t`. If the terminal is successfully opened, a handle for the terminal will be returned through this parameter.

terminalName Specifies the name of the terminal to access. If *terminalName* is a null pointer, it indicates that the library should connect with the default terminal for the session.

Description The `SCF_Session_getTerminal()` function establishes a context with a specific smartcard terminal (also known as a reader) in the session. Terminal objects are used for detecting card movement (insertion or removal) and to create card objects for accessing a specific card.

The list of available terminal names can be retrieved by calling `SCF_Session_getInfo(3SMARTCARD)`. Unless the user explicitly requests a specific terminal, applications should use the session's default terminal by calling `SCF_Session_getTerminal()` with a null pointer for the terminal name. This eliminates the need to first process an available-terminal list with just one element on systems with only a single smartcard terminal. On multi-terminal systems, the user can preconfigure one of the terminals as the default (or preferred) terminal. See USAGE below.

If `SCF_Session_getTerminal()` is called multiple times in the same session to access the same physical terminal, the same `SCF_Terminal_t` will be returned in each call. Multithreaded applications must take care to avoid having one thread close a terminal that is still needed by another thread. This can be accomplished by coordination within the application or by having each thread open a separate session to avoid interference.

When the terminal is no longer needed, `SCF_Terminal_close(3SMARTCARD)` should be called to release terminal resources. Closing a terminal will also close any cards opened from the terminal.

Return Values Upon success, `SCF_STATUS_SUCCESS` is returned and *terminal* contains the opened terminal. Otherwise, an error value is returned and *terminal* remains unaltered.

Errors The `SCF_Session_getTerminal()` function will fail if:

`SCF_STATUS_BADARGS` The *terminal* argument is a null pointer.

`SCF_STATUS_BADHANDLE` The session was closed or is invalid.

SCF_STATUS_BADTERMINAL	The specified *terminalName* is not valid for this session, or the default terminal could not be opened because there are no terminals available in this session.
SCF_STATUS_COMMERROR	The connection to the server was lost.
SCF_STATUS_FAILED	An internal error occurred.

Examples

EXAMPLE 1 Use the default terminal.

```
SCF_Status_t status;
SCF_Session_t mySession;
SCF_Terminal_t myTerminal;
char *myName;

/* (...call SCF_Session_getSession to open mySession...) */

status = SCF_Session_getTerminal(mySession, NULL, &myTerminal);
if (status != SCF_STATUS_SUCCESS) exit(1);

status = SCF_Terminal_getInfo(myTerminal, "name", &myName);
if (status != SCF_STATUS_SUCCESS) exit(1);

printf("Please insert a card into the terminal named %s\n", myName);

/* ... */
```

EXAMPLE 2 Open a terminal by name.

```
SCF_Status_t status;
SCF_Session_t mySession;
SCF_Terminal_t myTerminal;
char *myName;

/* (...call SCF_Session_getSession to open mySession...) */

/*
 * The name should be selected from the list of terminal names
 * available from SCF_Session_getInfo, but it could also be
 * read from an appliation's config file or from user input.
 */
myName = "SunInternalReader";

status = SCF_Session_getTerminal(mySession, myName, &myTerminal);
if (status == SCF_STATUS_BADTERMINAL) {
    printf("There is no terminal named %s.\n", myName);
    exit(1);
} else if (status != SCF_STATUS_SUCCESS) exit(2);
```

EXAMPLE 2 Open a terminal by name. *(Continued)*

```
/* ... */
```

Usage When using the Solaris OCF smartcard framework, the default reader is specified by the `ocf.client.default.defaultreader` property. If this property is not set, the first available reader is chosen as the default. Users can set the `SCF_DEFAULT_TERMINAL` environment variable to the name of a terminal to override the normal default. The `smartcard` utility can also be used to add terminals to or remove terminals from the system. See `smartcard(1M)` for information on how to add or modify the OCF property.

Terminals can be accessed only by the user who expected to have physical access to the terminal. By default, this user is assumed to be the owner of `/dev/console` and the superuser. Certain terminals such as Sun Ray appliances can use a different method to restrict access to the terminal.

The framework also uses the `DISPLAY` environment variable to further restrict which terminals are listed for a user. By default, terminals are associated with the ":0" display. Sun Ray terminals are associated with the display for that session, for example ":25". If the `DISPLAY` environment variable is not set or is a display on another host, it is treated as though it were set to ":0". Terminals not associated with the user's `DISPLAY` are not listed. To override this behaviour, the `SCF_FILTER_KEY` environment variable can be set to the desired display, for example ":0", ":25", and so on. To list all terminals to which a user has access, `SCF_FILTER_KEY` can be set to the special value of ":*".

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

See Also `smartcard(1M)`, `libsmartcard(3LIB)`, `SCF_Session_getInfo(3SMARTCARD)`, `SCF_Session_getSession(3SMARTCARD)`, `SCF_Terminal_close(3SMARTCARD)`, `attributes(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.

Thescf_simple_app_props_search() function queries for an exact match on a property in a property group. It takes a service instance FMRI, 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.

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()` function will fail if:

`SCF_ERROR_NOT_FOUND` The specified instance or property does not exist.

`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_CONNECTION_BROKEN` The connection to the datastore is broken.

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...)


prop = scf_simple_app_props_next(appprops, prop);
```

EXAMPLE 2 Property Iteration *(Continued)*

```
}

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	Evolving
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_DELETED	The snapshot has been deleted.
SCF_ERROR_NOT_SET	The snapshot is not set.
SCF_ERROR_HANDLE_MISMATCH	The snapshot and snaplevel are not derived from the same handle.
SCF_ERROR_NOT_FOUND	There are no snaplevels in this snapshot.
SCF_ERROR_NOT_BOUND	The handle is not bound.
SCF_ERROR_CONNECTION_BROKEN	The connection to the repository was lost.

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	Evolving
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_DELETED`
The instance has been deleted.

`SCF_ERROR_NOT_SET`
The instance is not set.

`SCF_ERROR_NOT_FOUND`
The snapshot specified by *name* was not found.

`SCF_ERROR_INVALID_ARGUMENT`
The value of the *name* argument is not a valid snapshot name.

`SCF_ERROR_HANDLE_MISMATCH`
The instance and snapshot 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_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_strerror – get a string describing a status code

Synopsis

```
cc [ flag... ] file... -lsmartcard [ library...]
#include <smartcard/scf.h>

const char *SCF_strerror(SCF_Status_t error);
```

Parameters

error A value returned from a smartcard SCF function call. A list of all current codes is contained in <smartcard/scf.h>

Description The `SCF_strerror()` function provides a mechanism for generating a brief message that describes each `SCF_Status_t` error code. An application might use the message when displaying or logging errors.

The string returned by the function does not contain any newline characters. Returned strings must not be modified or freed by the caller.

Return Values A pointer to a valid string is always returned. If the provided *error* is not a valid SCF error code, a string is returned stating that the error code is unknown. A null pointer is never returned.

Examples **EXAMPLE 1** Report a fatal error.

```
SCF_Status_t status;
SCF_Session_t mySession;

status = SCF_Session_getSession(&mySession);
if (status != SCF_STATUS_SUCCESS) {
    printf("Smartcard startup error: %s\n", SCF_strerror(status));
    exit(1);
}

/* ... */
```

Usage Messages returned from `SCF_strerror()` are in the native language specified by the `LC_MESSAGES` locale category; see `setlocale(3C)`. The C locale is used if the native strings could not be loaded.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

See Also libsmartcard(3LIB), SCF_Session_getSession(3SMARTCARD), strerror(3C), attributes(5)

Name SCF_Terminal_addEventListener, SCF_Terminal_updateEventListener, SCF_Terminal_removeEventListener – receive asychronous event notification

Synopsis

```
cc [ flag... ] file... -lsmartcard [ library...]
#include <smartcard/scf.h>

SCF_Status_t SCF_Terminal_addEventListener(SCF_Terminal_t terminal,
     SCF_Event_t events, void(*callback)
     (SCF_Event_t, SCF_Terminal_t, void *), void *userData, SCF_ListenerHandle_t *listenerHandle);

SCF_Status_t SCF_Terminal_updateEventListener(SCF_Terminal_t terminal,
     SCF_ListenerHandle_t listenerHandle, SCF_Event_t events);

SCF_Status_t SCF_Terminal_removeEventListener(SCF_Terminal_t terminal,
     SCF_ListenerHandle_t listenerHandle);
```

Parameters

terminal A terminal (from SCF_Session_getTerminal(3SMARTCARD)) to which the event listener should be added or removed.

events Events to deliver to the callback. An event will not be delivered if it is not listed. The caller can register for multiple events by performing a bitwise OR of the desired events. The valid events are:

SCF_EVENT_ALL	All of the events listed below will be delivered.
SCF_EVENT_CARDINSERTED	A smartcard was inserted into the terminal.
SCF_EVENT_CARDREMOVED	A smartcard was removed from the terminal.
SCF_EVENT_CARDPRESENT	Indicates that a card was present in the terminal when the event listener was first added. This event allows event listeners to determine the initial state of the terminal before an insert or remove event occurs. Either this event or the SCF_EVENT_CARDABSENT (see below) event will be delivered only once upon adding an event listener and immediately before any other events are delivered. Future card movements will generate SCF_EVENT_CARDINSERTED and SCF_EVENT_CARDREMOVED events, but not SCF_EVENT_CARDPRESENT or SCF_EVENT_CARDABSENT events. An event listener can assume that if a SCF_EVENT_CARDPRESENT event is

		delivered, the next card movement event will be a `SCF_EVENT_CARDREMOVED`.
	`SCF_EVENT_CARDABSENT`	Indicates that a card was not present in the terminal when the event listener was first added. This event allows event listeners to determine the initial state of the terminal before an insert or remove event occurs. Either this event or the `SCF_EVENT_CARDPRESENT` event (see above) will be delivered only once upon adding an event listener and immediately before any other events are delivered. Future card movements will generate `SCF_EVENT_CARDINSERTED` and `SCF_EVENT_CARDREMOVED` events, but not `SCF_EVENT_CARDPRESENT` or `SCF_EVENT_CARDABSENT` events. An event listener can assume that if a `SCF_EVENT_CARDABSENT` event is delivered, the next card movement event will be a `SCF_EVENT_CARDINSERTED`.
	`SCF_EVENT_CARDRESET`	The smartcard currently present has been reset (see `SCF_Card_reset`(3SMARTCARD)).
	`SCF_EVENT_TERMINALCLOSED`	The terminal is in the process of being closed (due to a call to `SCF_Session_close`(3SMARTCARD) or `SCF_Terminal_close`(3SMARTCARD)), so no further events will be delivered. The *terminal* argument provided to the callback will still be valid.
	`SCF_EVENT_COMMERROR`	The connection to the server has been lost. No further events will be delivered.
callback	A function pointer that will be executed when the desired event occurs. The function must take three arguments. The first is a `SCF_Event_t` containing the event that occured. The second argument is an `SCF_Terminal_t` containing the terminal on which the event occured. The third is a `void *` that can be used to provide arbitrary data to the *callback* when it is executed.	

userData A pointer to arbitrary user data. The data is not accessed by the library. The pointer is simply provided to the callback when an event is issued. This argument can safely be set to `NULL` if not needed. The callback must be able to handle this case.

listenerHandle A unique "key" that is provided by `SCF_Terminal_addEventListener()` to refer to a specific event listener registration. This allows multiple event listeners to be selectivly updated or removed.

Description These functions allow an application to receive notification of events on a terminal as they occur. The concept is similar to a signal handler. When an event occurs, a thread in the SCF library will execute the provided *callback* function. Once added, the listener will receive events until it is removed or either the terminal or session is closed.

When the callback function is executed, the callback arguments specify the event that occured and the terminal on which it occurred. Additionally, each callback will receive the *userData* pointer that was provided when the listener was added. The library does not make a copy of the memory pointed to by *userData*, so applications must take care not to deallocate that memory until it is known that the callback will no longer access it (for example, by removing the event listener). Each invocation of the callback will be for exactly one event. If the library needs to deliver multiple events, they will be dispatched one at a time. Because the callback is executed from a thread, any operations it performs must be thread safe. For each callback registration, the library creates a new thread to deliver events to that callback. The callback is expected to perform minimal work and return quickly.

An application can add multiple callbacks on a terminal. Any event that occurs will be delivered to all listeners that registered for that event type. The same callback can be registered multiple times. Each call to `SCF_Terminal_addEventListener()` will result in a new `SCF_ListenerHandle_t`. The events a callback receives can be changed by calling `SCF_Terminal_updateEventListener()` with the handle that was returned when the listener was initially added. If the listener is set to receive no events (that is, the events parameter has no bits set), the listener will remain registered but will not receive any events. To remove a listener and release allocated resources, use `SCF_Terminal_removeEventListener()` or close the terminal.

Return Values If the event listener was successfully added or removed, `SCF_STATUS_SUCCESS` is returned. Otherwise, an error value is returned and the internal list of registered event listeners remains unaltered.

Errors These functions will fail if:

`SCF_STATUS_BADARGS` The callback function pointer and/or *listenerHandle* is null, or an unknown event was specified.

`SCF_STATUS_BADHANDLE` The specified terminal has been closed or is invalid, or the event listener handle could not be found to update or remove.

SCF_STATUS_COMMERROR	The connection to the server was lost.
SCF_STATUS_FAILED	An internal error occurred.

Examples

EXAMPLE 1 Register for card movements.

```
struct myState_t {
    int isStateKnown;
    int isCardPresent;
};

void myCallback(SCF_Event_t event, SCF_Terminal_t eventTerminal,
    void *data) {
    struct myState_t *state = data;
    if (event == SCF_EVENT_CARDINSERTED) {
        printf("--- Card inserted ---\n");
        state->isCardPresent = 1;
    }
    else if (event == SCF_EVENT_CARDREMOVED) {
        printf("--- Card removed ---\n");
        state->isCardPresent = 0;
    }
    state->isStateKnown = 1;
}

main() {
    SCF_Status_t status;
    SCF_Terminal_t myTerminal;
    SCF_ListenerHandle_t myListener;
    struct myState_t myState;

    /* (...call SCF_Session_getTerminal to open myTerminal...) */

    myState.isStateKnown = 0;
    status = SCF_Terminal_addEventListener(myTerminal,
        SCF_EVENT_CARDINSERTED|SCF_EVENT_CARDREMOVED, &myCallback,
        &myState, &myListener);
    if (status != SCF_STATUS_SUCCESS) exit(1);

    while(1) {
        if (!myState.isStateKnown)
            printf("Waiting for first event...\n");
        else {
            if (myState.isCardPresent)
                printf("Card is present.\n");
            else
                printf("Card is not present.\n");
        }
```

EXAMPLE 1 Register for card movements. *(Continued)*

```
        sleep(1);
    }
}
```

EXAMPLE 2 Use different callbacks for each event.

```
void myInsertCallback(SCF_Event_t event, SCF_Terminal_t eventTerminal,
    void *data) {

    /* ... */
}

void myRemoveCallback(SCF_Event_t event, SCF_Terminal_t eventTerminal,
    void *data) {
    /* ... */
}

main () {
    SCF_Status_t status;
    SCF_Terminal_t terminal;
    SCF_ListenerHandle_t myListener1, myListener2, myListener3;
    int foo, bar;

    /* (...call SCF_Session_getTerminal to open myTerminal...) */

    status = SCF_Terminal_addEventListener(myTerminal,
        SCF_EVENT_CARDINSERTED, &myInsertCallback, &foo,
        &myListener1);
    if (status != SCF_STATUS_SUCCESS) exit(1);

    status = SCF_Terminal_addEventListener(myTerminal,
        SCF_EVENT_CARDREMOVED, &myRemoveCallback, &foo,
        &myListener2);
    if (status != SCF_STATUS_SUCCESS) exit(1);

    status = SCF_Terminal_addEventListener(myTerminal,
        SCF_EVENT_CARDREMOVED, &myRemoveCallback, &bar,
        &myListener3);
    if (status != SCF_STATUS_SUCCESS) exit(1);

    /*
     * At this point, when each insertion occurs, myInsertCallback
     * will be called once (with a pointer to foo). When each removal
     * occurs, myRemoveCallback will be called twice. One call will
     * be given a pointer to foo, and the other will be given a
     * pointer to bar.
```

EXAMPLE 2 Use different callbacks for each event. *(Continued)*

```
     */

    status = SCF_Terminal_removeEventListener(myTerminal,
        myListener2);
    if (status != SCF_STATUS_SUCCESS) exit(1);

    /*
     * Now, when a removal occurs, myRemoveCallback will only be
     * called once, with a pointer to bar.
     */

    /* ... */
}
```

EXAMPLE 3 Use initial state events to show user the terminal state in a GUI.

```
void myCallback(SCF_Event_t event, SCF_Terminal_t eventTerminal,
    void *unused) {
    if (event == SCF_EVENT_CARDPRESENT) {
        /* Set initial icon to a terminal with a card present. */
    }
    else if (event == SCF_EVENT_CARDABSENT) {
        /* Set initial icon to a terminal without a card present. */
    }
    else if (event == SCF_EVENT_CARDINSERTED) {
        /* Show animation for card being inserted into a terminal. */
    }
    else if (event == SCF_EVENT_CARDREMOVED) {
        /* Show animation for card being removed from a terminal. */
    }
}

main() {
    SCF_Terminal_t myTerminal;
    SCF_ListenerHandle_t myListener;

    /* (...call SCF_Session_getTerminal to open myTerminal...) */

    status = SCF_Terminal_addEventListener(myTerminal,
        SCF_EVENT_ALL, &myCallback, NULL, &myListener);
    if (status != SCF_STATUS_SUCCESS) exit(1);


    /* ... */
}
```

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

See Also libsmartcard(3LIB), SCF_Card_reset(3SMARTCARD), SCF_Session_close(3SMARTCARD), SCF_Session_getTerminal(3SMARTCARD), SCF_Terminal_updateEventListener(3SMARTCARD), SCF_Terminal_close(3SMARTCARD), SCF_Terminal_removeEventListener(3SMARTCARD), attributes(5)

Name SCF_Terminal_getCard – establish a context with a smartcard

Synopsis

```
cc [ flag... ] file... -lsmartcard [ library...]
#include <smartcard/scf.h>

SCF_Status_t SCF_Terminal_getCard(SCF_Terminal_t terminal,
     SCF_Card_t *card);
```

Parameters

card A pointer to a `SCF_Card_t`. If the smartcard is successfully opened, a handle for the card will be returned through this parameter.

terminal The terminal (from `SCF_Session_getTerminal(3SMARTCARD)`) containing a smartcard to open.

Description The `SCF_Terminal_getCard()` function establishes a context with a specific smartcard in a terminal. Card objects can be used to send APDUs (Application Protocol Data Units) to the card with `SCF_Card_exchangeAPDU(3SMARTCARD)`. When the card is no longer needed, `SCF_Card_close(3SMARTCARD)` should be called to release allocated resources.

If `SCF_Terminal_getCard()` is called multiple times in the same session to access the same physical card (while the card remains inserted), the same `SCF_Card_t` will be returned in each call. The library cannot identifty specific cards, so when a card is reinserted it will be represented by a new `SCF_Card_t`. Multithreaded applications must take care to avoid having one thread close a card that is still needed by another thread. This can be accomplished by coordination within the application, or by having each thread open a separate session to avoid interference.

Return Values If a working card is present in the reader, `SCF_STATUS_SUCCESS` is returned and *card* is a valid reference to the card. Otherwise, an error value is returned and card remains unaltered.

Errors The `SCF_Terminal_getCard()` function will fail if:

`SCF_STATUS_BADARGS`	The *card* argument is a null pointer.
`SCF_STATUS_BADHANDLE`	The specified terminal has been closed or is invalid.
`SCF_STATUS_FAILED`	An internal error occured.
`SCF_STATUS_NOCARD`	No card is present in the terminal.

Examples **EXAMPLE 1** Access a smartcard.

```
SCF_Status_t status;
SCF_Terminal_t myTerminal;
SCF_Card_t myCard;

/* (...call SCF_Session_getTerminal to open myTerminal...) */

status = SCF_Terminal_getCard(myTerminal, &myCard);
if (status == SCF_STATUS_NOCARD) {
```

EXAMPLE 1 Access a smartcard. *(Continued)*

```
    printf("Please insert your smartcard and try again.\n");
    exit(0);
}
else if (status != SCF_STATUS_SUCCESS) exit(1);

/* (...go on to use the card with SCF_Card_exchangeAPDU()...) */
```

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

See Also

libsmartcard(3LIB), SCF_Card_close(3SMARTCARD), SCF_Card_exchangeAPDU(3SMARTCARD), SCF_Card_getInfo(3SMARTCARD), SCF_Card_lock(3SMARTCARD), SCF_Session_getTerminal(3SMARTCARD), attributes(5)

Name SCF_Terminal_waitForCardPresent, SCF_Terminal_waitForCardAbsent, SCF_Card_waitForCardRemoved – wait for a card to be inserted or removed

Synopsis

```
cc [ flag... ] file... -lsmartcard [ library...]
#include <smartcard/scf.h>

SCF_Status_t SCF_Terminal_waitForCardPresent(SCF_Terminal_t terminal,
     unsigned int timeout);

SCF_Status_t SCF_Terminal_waitForCardAbsent(SCF_Terminal_t terminal,
     unsigned int timeout);

SCF_Status_t SCF_Card_waitForCardRemoved(SCF_Card_t card,
     unsigned int timeout);
```

Parameters

card A card that was returned from SCF_Terminal_getCard(3SMARTCARD).

terminal A terminal that was returned from SCF_Session_getTerminal(3SMARTCARD).

timeout The maximum number or seconds to wait for the desired state to be reached. If the timeout is 0, the function will immediately return SCF_STATUS_TIMEOUT if the terminal or card is not in the desired state. A timeout of SCF_TIMEOUT_MAX can be specified to indicate that the function should never timeout.

Description These functions determine if a card is currently available in the specified terminal.

The SCF_Card_waitForCardRemoved() function differs from SCF_Terminal_waitForCardAbsent() in that it checks to see if a specific card has been removed. If another card (or even the same card) has since been reinserted, SCF_Card_waitForCardRemoved() will report that the old card was removed, while the SCF_Terminal_waitForCardAbsent() will instead report that there is a card present.

If the desired state is already true, the function will immediately return SCF_STATUS_SUCCESS. Otherwise it will wait for a change to the desired state, or for the timeout to expire, whichever occurs first.

Unlike an event listener (SCF_Terminal_addEventListener(3SMARTCARD)), these functions return the state of the terminal, not just events. To use an electronics analogy, event listeners are edge-triggered, while these functions are level-triggered.

Return Values If the desired state is reached before the timeout expires, SCF_STATUS_SUCCESS is returned. If the timeout expires, SCF_STATUS_TIMEOUT is returned. Otherwise, an error value is returned.

Errors These functions will fail if:

SCF_STATUS_BADHANDLE The specified *terminal* or *card* has been closed or is invalid.

SCF_STATUS_COMMERROR The server closed the connection.

SCF_STATUS_FAILED	An internal error occured.

Examples

EXAMPLE 1 Determine if a card is currently inserted.

```
int isCardCurrentlyPresent(SCF_Terminal_t myTerminal) {
    SCF_Status_t status;

    /*
     * The timeout of zero makes sure this call will always
     * return immediately.
     */
    status = SCF_Terminal_waitForCardPresent(myTerminal, 0);

    if (status == SCF_STATUS_SUCCESS) return (TRUE);
    else if (status == SCF_STATUS_TIMEOUT) return (FALSE);

    /*
     * For other errors, this example just assumes no card
     * is present. We don't really know.
     */
    return (FALSE);
}
```

EXAMPLE 2 Remind the user every 5 seconds to remove their card.

```
SCF_Status_t status;
SCF_Terminal_t myTerminal;

/* (...call SCF_Session_getTerminal to open myTerminal...) */

status = SCF_Terminal_waitForCardAbsent(myTerminal, 0);
while (status == SCF_STATUS_TIMEOUT) {
    printf("Please remove the card from the terminal!\n");
    status = SCF_Terminal_waitForCardAbsent(myTerminal, 5);
}

if (status == SCF_STATUS_SUCCESS)
    printf("Thank you.\n");
else
    exit(1);

/* ... */
```

EXAMPLE 3 Demonstrate the difference between the card-specific and terminal-specific calls.

```
SCF_Status_t status;
SCF_Terminal_t myTerminal;
SCF_Card_t myCard;
```

EXAMPLE 3 Demonstrate the difference between the card-specific and terminal-specific calls. *(Continued)*

```
/* (...call SCF_Session_getTerminal to open myTerminal...) */

status = SCF_Terminal_getCard(myTerminal, &myCard);
if (status != SCF_STATUS_SUCCESS) exit(1);

/*
 * While we sleep, assume user removes the card
 * and inserts another card.
 */
sleep(10);

status = SCF_Terminal_waitForCardAbsent(myTerminal, 0);
/*
 * In this case, status is expected to be SCF_STATUS_TIMEOUT, as there
 * is a card present.
 */

status = SCF_Card_waitForCardRemoved(myCard, 0);
/*
 * In this case, status is expected to be SCF_STATUS_SUCCESS, as the
 * card returned from SCF_Terminal_getCard was indeed removed (even
 * though another card is currently in the terminal).
 */

/* ... */
```

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

See Also

libsmartcard(3LIB), SCF_Session_getTerminal(3SMARTCARD), SCF_Terminal_addEventListener(3SMARTCARD), SCF_Terminal_getCard(3SMARTCARD), attributes(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_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_NOT_SET` The transaction has not been started.

`SCF_ERROR_DELETED` The property group the transaction is changing has been deleted.

`SCF_ERROR_IN_USE` The property already has an entry in the transaction.

`SCF_ERROR_INVALID_ARGUMENT` The *prop_name* argument is not a valid property name.

`SCF_ERROR_HANDLE_MISMATCH` The transaction and entry 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_transaction_property_delete()`, `scf_transaction_property_change()`, and `scf_transaction_property_change_type()` functions will fail if:

`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*.

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 */
```

EXAMPLE 1 Set an existing boolean value to true. *(Continued)*

```
     if (scf_transaction_property_change(tx, e1, "property",
        SCF_TYPE_BOOLEAN) == -1) {
            scf_transaction_reset(tx);
            goto fail;
        }
        scf_value_set_boolean(v1, B_TRUE);
        scf_entry_add_value(e1, v1);

     if (scf_transaction_add(tx, e1) == -1) {
          scf_transaction_reset(tx);
          goto fail;
     }

     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	Evolving
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 string written, 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_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 *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 sendfile – send files over sockets or copy files to files

Synopsis

```
cc [ flag... ] file... -lsendfile [ library... ]
#include <sys/sendfile.h>

ssize_t sendfile(int out_fd, int in_fd, off_t *off, size_t len);
```

Description The `sendfile()` function copies data from *in_fd* to *out_fd* starting at offset *off* and of length *len* bytes. The *in_fd* argument should be a file descriptor to a regular file opened for reading. See `open(2)`. The *out_fd* argument should be a file descriptor to a regular file opened for writing or to a connected `AF_INET` or `AF_INET6` socket of `SOCK_STREAM` type. See `socket(3SOCKET)`. The *off* argument is a pointer to a variable holding the input file pointer position from which the data will be read. After `sendfile()` has completed, the variable will be set to the offset of the byte following the last byte that was read. The `sendfile()` function does not modify the current file pointer of *in_fd*, but does modify the file pointer for *out_fd* if it is a regular file.

The `sendfile()` function can also be used to send buffers by pointing *in_fd* to `SFV_FD_SELF`.

Return Values Upon successful completion, `sendfile()` returns the total number of bytes written to *out_fd* and also updates the offset to point to the byte that follows the last byte read. Otherwise, it returns –1, and `errno` is set to indicate the error.

Errors The `sendfile()` function will fail if:

`EAFNOSUPPORT`	The implementation does not support the specified address family for socket.
`EAGAIN`	Mandatory file or record locking is set on either the file descriptor or output file descriptor if it points at regular files. `O_NDELAY` or `O_NONBLOCK` is set, and there is a blocking record lock. An attempt has been made to write to a stream that cannot accept data with the `O_NDELAY` or the `O_NONBLOCK` flag set.
`EBADF`	The *out_fd* or *in_fd* argument is either not a valid file descriptor, *out_fd* is not opened for writing. or *in_fd* is not opened for reading.
`EINVAL`	The offset cannot be represented by the `off_t` structure, or the length is negative when cast to `ssize_t`.
`EIO`	An I/O error occurred while accessing the file system.
`ENOTCONN`	The socket is not connected.
`EOPNOTSUPP`	The socket type is not supported.
`EPIPE`	The *out_fd* argument is no longer connected to the peer endpoint.
`EINTR`	A signal was caught during the write operation and no data was transferred.

Usage The `sendfile()` function has a transitional interface for 64-bit file offsets. See `lf64(5)`.

Examples **EXAMPLE 1** Sending a Buffer Over a Socket

The following example demonstrates how to send the buffer *buf* over a socket. At the end, it prints the number of bytes transferred over the socket from the buffer. It assumes that *addr* will be filled up appropriately, depending upon where to send the buffer.

```
int tfd;
off_t baddr;
struct sockaddr_in sin;
char buf[64 * 1024];
in_addr_t addr;
size_t len;

tfd = socket(AF_INET, SOCK_STREAM, 0);
if (tfd == -1) {
    perror("socket");
    exit(1);
}

sin.sin_family = AF_INET;
sin.sin_addr.s_addr = addr;    /* Fill in the  appropriate address. */
sin.sin_port = htons(2345);
if (connect(tfd, (struct sockaddr *)&sin, sizeof(sin))<0) {
    perror("connect");
    exit(1);
}

baddr = (off_t)buf;
len = sizeof(buf);
while (len > 0) {
    ssize_t res;
    res = sendfile(tfd, SFV_FD_SELF, &baddr, len);
    if (res == -1)
            if (errno != EINTR) {
                    perror("sendfile");
                    exit(1);
            } else continue;
    len -= res;
}
```

EXAMPLE 2 Transferring Files to Sockets

The following program demonstrates a transfer of files to sockets:

```
int ffd, tfd;
off_t off;
struct sockaddr_in sin;
```

EXAMPLE 2 Transferring Files to Sockets *(Continued)*

```
in_addr_t  addr;
int len;
struct stat stat_buf;
ssize_t len;

ffd = open("file", O_RDONLY);
if (ffd == -1) {
    perror("open");
    exit(1);
}

tfd = socket(AF_INET, SOCK_STREAM, 0);
if (tfd == -1) {
    perror("socket");
    exit(1);
}

sin.sin_family = AF_INET;
sin.sin_addr = addr;    /* Fill in the  appropriate address. */
sin.sin_port = htons(2345);
if (connect(tfd, (struct sockaddr *) &sin, sizeof(sin)) <0) {
    perror("connect");
    exit(1);
}

if (fstat(ffd, &stat_buf) == -1) {
    perror("fstat");
    exit(1);
}

len = stat_buf.st_size;
while (len > 0) {
    ssize_t res;
    res = sendfile(tfd, ffd, &off, len);
    if (res == -1)
            if (errno != EINTR) {
                    perror("sendfile");
                    exit(1);
            } else continue;
    len -= res;
}
```

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

See Also open(2), libsendfile(3LIB), sendfilev(3EXT), socket(3SOCKET), attributes(5), lf64(5)

Name sendfilev – send a file

Synopsis

```
cc [ flag... ] file... -lsendfile  [ library... ]
#include <sys/sendfile.h>

ssize_t sendfilev(int fildes, const struct sendfilevec *vec,
     int sfvcnt, size_t *xferred);
```

Parameters The `sendfilev()` function supports the following parameters:

fildes A file descriptor to a regular file or to a `AF_NCA`, `AF_INET`, or `AF_INET6` family type `SOCK_STREAM` socket that is open for writing. For `AF_NCA`, the protocol type should be zero.

vec An array of `SENDFILEVEC_T`, as defined in the `sendfilevec` structure above.

sfvcnt The number of members in *vec*.

xferred The total number of bytes written to `out_fd`.

Description The `sendfilev()` function attempts to write data from the *sfvcnt* buffers specified by the members of *vec* array: `vec[0], vec[1], ... , vec[sfvcnt–1]`. The *fildes* argument is a file descriptor to a regular file or to an `AF_NCA`, `AF_INET`, or `AF_INET6` family type `SOCK_STREAM` socket that is open for writing.

This function is analogous to `writev(2)`, but can read from both buffers and file descriptors. Unlike `writev()`, in the case of multiple writers to a file the effect of `sendfilev()` is not necessarily atomic; the writes may be interleaved. Application-specific synchronization methods must be employed if this causes problems.

The following is the `sendfilevec` structure:

```
typedef struct sendfilevec {
        int     sfv_fd;         /* input fd */
        uint_t  sfv_flag;       /* Flags. see below */
        off_t   sfv_off;        /* offset to start reading from */
        size_t  sfv_len;        /* amount of data */
} sendfilevec_t;

#define SFV_FD_SELF     (-2)
```

To send a file, open the file for reading and point `sfv_fd` to the file descriptor returned as a result. See `open(2)`. `sfv_off` should contain the offset within the file. `sfv_len` should have the length of the file to be transferred.

The *xferred* argument is updated to record the total number of bytes written to `out_fd`.

The `sfv_flag` field is reserved and should be set to zero.

To send data directly from the address space of the process, set `sfv_fd` to `SFV_FD_SELF`. `sfv_off` should point to the data, with `sfv_len` containing the length of the buffer.

Return Values Upon successful completion, the `sendfilev()` function returns total number of bytes written to `out_fd`. Otherwise, it returns `-1`, and `errno` is set to indicate the error. The *xferred* argument contains the amount of data successfuly transferred, which can be used to discover the error vector.

Errors

`EACCES`	The process does not have appropriate privileges or one of the files pointed by `sfv_fd` does not have appropriate permissions.
`EAFNOSUPPORT`	The implementation does not support the specified address family for socket.
`EAGAIN`	Mandatory file or record locking is set on either the file descriptor or output file descriptor if it points at regular files. `O_NDELAY` or `O_NONBLOCK` is set, and there is a blocking record lock. An attempt has been made to write to a stream that cannot accept data with the `O_NDELAY` or the `O_NONBLOCK` flag set.
`EBADF`	The *fildes* argument is not a valid descriptor open for writing or an `sfv_fd` is invalid or not open for reading.
`EFAULT`	The *vec* argument points to an illegal address.
	The *xferred* argument points to an illegal address.
`EINTR`	A signal was caught during the write operation and no data was transferred.
`EINVAL`	The *sfvcnt* argument was less than or equal to `0`. One of the `sfv_len` values in *vec* array was less than or equal to `0`, or greater than the file size. An `sfv_fd` is not seekable.
	Fewer bytes were transferred than were requested.
`EIO`	An I/O error occurred while accessing the file system.
`EPIPE`	The *fildes* argument is a socket that has been shut down for writing.
`EPROTOTYPE`	The socket type is not supported.

Usage The `sendfilev()` function has a transitional interface for 64-bit file offsets. See `lf64(5)`.

Examples The following example sends 2 vectors, one of HEADER data and a file of length 100 over `sockfd`. `sockfd` is in a connected state, that is, `socket()`, `accept()`, and `bind()` operation are complete.

```
#include <sys/sendfile.h>
.
```

```
.
.
int
main (int argc, char *argv[]){
  int sockfd;
  ssize_t ret;
  size_t xfer;
  struct sendfilevec vec[2];
    .
    .
    .
  vec[0].sfv_fd = SFV_FD_SELF;
  vec[0].sfv_flag = 0;
  vec[0].sfv_off = "HEADER_DATA";
  vec[0].sfv_len = strlen("HEADER_DATA");
  vec[1].sfv_fd = open("input_file",.... );
  vec[1].sfv_flag = 0;
  vec[1].sfv_off = 0;
  vec[1].sfv_len = 100;

  ret = sendfilev(sockfd, vec, 2, &xfer);
.
.
.
}
```

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also open(2), writev(2), libsendfile(3LIB), sendfile(3EXT), socket(3SOCKET), 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 *Oracle 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 setproject – associate a user process with a project

Synopsis

```
cc [ flag ... ] file... -lproject [ library ... ]
#include <project.h>

int setproject(const char *project_name, const char *user_name,
     uint_t flags);
```

Description The `setproject()` function provides a simplified method for the association of a user process with a project and its various resource management attributes, as stored in the `project(4)` name service database. These attributes include resource control settings, resource pool membership, and third party attributes (which are ignored by `setproject()`.)

If *user_name* is a valid member of the project specified by *project_name*, as determined by `inproj(3PROJECT)`, `setproject()` will create a new task with `settaskid(2)` using task flags specified by *flags*, use `setrctl(2)` to associate various resource controls with the process, task, and project, and bind the calling process to the appropriate resource pool with `pool_set_binding(3POOL)`. Resource controls not explicitly specified in the project entry will be preserved. If *user_name* is a name of the superuser (user with UID equal to 0), the `setproject()` function skips the `inproj(3PROJECT)` check described above and allows the superuser to join any project.

The current process will not be bound to a resource pool if the resource pools facility (see `pooladm(1M)`) is inactive. The `setproject()` function will succeed whether or not the project specified by *project_name* specifies a `project.pool` attribute. If the resource pools facility is active, `setproject()` will fail if the project does not specify a `project.pool` attribute and there is no designated pool accepting default assignments. The `setproject()` function will also fail if there is a specified `project.pool` attribute for a nonexistent pool.

Return Values Upon successful completion, `setproject()` returns 0. If any of the resource control assignments failed but the project assignment, pool binding, and task creation succeeded, an integer value corresponding to the offset into the key-value pair list of the failed attribute assignment is returned. If the project assignment or task creation was not successful, `setproject()` returns `SETPROJ_ERR_TASK` and sets `errno` to indicate the error. In the event of a pool binding failure, `setproject()` returns `SETPROJ_ERR_POOL` and sets `errno` to indicate the error. Additional error information can be retrieved from `pool_error(3POOL)`.

Errors The `setproject()` function will fail during project assignment or task creation if:

`EACCES` The invoking task was created with the `TASK_FINAL` flag.

`EAGAIN` A resource control limiting the number of LWPs or tasks in the target project or zone has been exceeded.

A resource control on the given project would be exceeded.

`EINVAL` The project ID associated with the given project is not within the range of valid project IDs, invalid flags were specified, or *user_name* is `NULL`.

EPERM The effective user of the calling process is not superuser.

ESRCH The specified user is not a valid user of the given project, *user_name* is not valid user name, or *project_name* is not valid project name.

The `setproject()` function will fail during pool binding if:

EACCES No resource pool accepting default bindings exists.

EPERM The effective user of the calling process is not superuser.

ESRCH The specified resource pool is unknown

If `setproject()` returns an offset into the key-value pair list, the returned error value is associated with `setrctl(2)` for resource control attributes.

Usage The `setproject()` function recognizes a name-structured value pair for the attributes in the `project(4)` database with the following format:

`entity.control=(`*privilege*`,`*value*`,`*action*`,`*action*`,...),...`

where *privilege* is one of `BASIC` or `PRIVILEGED`, *value* is a numeric value with optional units, and *action* is one of `none`, `deny`, and `signal=`*signum* or `signal=`*SIGNAME*. For instance, to set a series of progressively more assertive control values on a project's per-process CPU time, specify

```
process.max-cpu-time=(PRIVILEGED,1000s,signal=SIGXRES), \
(PRIVILEGED,1250, signal=SIGTERM),(PRIVILEGED,1500,
     signal=SIGKILL)
```

To prevent a task from exceeding a total of 128 LWPs, specify a resource control with

```
task.max-lwps=(PRIVILEGED,128,deny)
```

Specifying a resource control name with no values causes all resource control values for that name to be cleared on the given project, leaving only the system resource control value on the specified resource control name.

For example, to remove all resource control values on shared memory, specify:

```
project.max-shm-memory
```

The project attribute, `project.pool`, specifies the pool to which processes associated with the project entry should be bound. Its format is:

```
project.pool=pool_name
```

where `pool_name` is a valid resource pool within the active configuration enabled with `pooladm(1M)`.

The final attribute is used to finalize the task created by `setproject()`. See`settaskid(2)`.

`task.final`

All further attempts to create new tasks, such as using `newtask(1)` and `su(1M)`, will fail.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

See Also

`pooladm(1M)`, `setrctl(2)`, `settaskid(2)`, `inproj(3PROJECT)`, `libproject(3LIB)`, `pool_error(3POOL)`, `pool_set_binding(3POOL)`, `passwd(4)`, `project(4)`, `attributes(5)`

Name sha1, SHA1Init, SHA1Update, SHA1Final – SHA1 digest functions

Synopsis

```
cc [ flag ... ] file ... -lmd [ library ... ]
#include <sha1.h>

void SHA1Init(SHA1_CTX *context);

void SHA1Update(SHA1_CTX *context, unsigned char *input,
     unsigned int inlen);

void SHA1Final(unsigned char *output, SHA1_CTX *context);
```

Description The SHA1 functions implement the SHA1 message-digest algorithm. The algorithm takes as input a message of arbitrary length and produces a 200-bit "fingerprint" or "message digest" as output. The SHA1 message-digest algorithm is intended for digital signature applications in which large files are "compressed" in a secure manner before being encrypted with a private (secret) key under a public-key cryptosystem such as RSA.

`SHA1Init()`, `SHA1Update()`, `SHA1Final()`

The `SHA1Init()`, `SHA1Update()`, and `SHA1Final()` functions allow a SHA1 digest to be computed over multiple message blocks. Between blocks, the state of the SHA1 computation is held in an SHA1 context structure allocated by the caller. A complete digest computation consists of calls to SHA1 functions in the following order: one call to `SHA1Init()`, one or more calls to `SHA1Update()`, and one call to `SHA1Final()`.

The `SHA1Init()` function initializes the SHA1 context structure pointed to by *context*.

The `SHA1Update()` function computes a partial SHA1 digest on the *inlen*-byte message block pointed to by *input*, and updates the SHA1 context structure pointed to by *context* accordingly.

The `SHA1Final()` function generates the final SHA1 digest, using the SHA1 context structure pointed to by *context*. The 16-bit SHA1 digest is written to output. After a call to `SHA1Final()`, the state of the context structure is undefined. It must be reinitialized with `SHA1Init()` before it can be used again.

Security The SHA1 algorithm is also believed to have some weaknesses. Migration to one of the SHA2 algorithms–including SHA256, SHA386 or SHA512–is highly recommended when compatibility with data formats and on wire protocols is permitted.

Return Values These functions do not return a value.

Examples **EXAMPLE 1** Authenticate a message found in multiple buffers

The following is a sample function that authenticates a message found in multiple buffers. The calling function provides an authentication buffer to contain the result of the SHA1 digest.

```
#include <sys/types.h>
#include <sys/uio.h>
#include <sha1.h>

int
AuthenticateMsg(unsigned char *auth_buffer, struct iovec
                *messageIov, unsigned int num_buffers)
{
    SHA1_CTX sha1_context;
    unsigned int i;

    SHA1Init(&sha1_context);

    for(i=0; i<num_buffers; i++)
    {
         SHA1Update(&sha1_context, messageIov->iov_base,
                    messageIov->iov_len);
         messageIov += sizeof(struct iovec);
    }

    SHA1Final(auth_buffer, &sha1_context);

    return 0;
}
```

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also sha2(3EXT), libmd(3LIB)

RFC 1374

Name sha2, SHA2Init, SHA2Update, SHA2Final, SHA256Init, SHA256Update, SHA256Final, SHA384Init, SHA384Update, SHA384Final, SHA512Init, SHA512Update, SHA512Final – SHA2 digest functions

Synopsis

```
cc [ flag ... ] file ... -lmd [ library ... ]
#include <sha2.h>

void SHA2Init(uint64_t mech, SHA2_CTX *context);

void SHA2Update(SHA2_CTX *context, unsigned char *input,
     unsigned int inlen);

void SHA2Final(unsigned char *output, SHA2_CTX *context);

void SHA256Init(SHA256_CTX *context);

void SHA256Update(SHA256_CTX *context, unsigned char *input,
     unsigned int inlen);

void SHA256Final(unsigned char *output, SHA256_CTX *context);

void SHA384Init(SHA384_CTX *context);

void SHA384Update(SHA384_CTX *context, unsigned char *input,
     unsigned int inlen);

void SHA384Final(unsigned char *output, 384_CTX *context);

void SHA512Init(SHA512_CTX *context);

void SHA512Update(SHA512_CTX *context, unsigned char *input,
     unsigned int inlen);

void SHA512Final(unsigned char *output, 512_CTX *context);
```

Description The `SHA2Init()`, `SHA2Update()`, `SHA2Final()` functions implement the `SHA256`, `SHA384` and `SHA512` message-digest algorithms. The algorithms take as input a message of arbitrary length and produces a 200-bit "fingerprint" or "message digest" as output. The `SHA2` message-digest algorithms are intended for digital signature applications in which large files are "compressed" in a secure manner before being encrypted with a private (secret) key under a public-key cryptosystem such as RSA.

`SHA2Init()`, `SHA2Update()`, `SHA2Final()`

The `SHA2Init()`, `SHA2Update()`, and `SHA2Final()` functions allow an `SHA2` digest to be computed over multiple message blocks. Between blocks, the state of the `SHA2` computation is held in an `SHA2` context structure allocated by the caller. A complete digest computation consists of calls to `SHA2` functions in the following order: one call to `SHA2Init()`, one or more calls to `SHA2Update()`, and one call to `SHA2Final()`.

The `SHA2Init()` function initializes the SHA2 context structure pointed to by *context*. The *mech* argument is one of `SHA256`, `SHA512`, `SHA384`.

The `SHA2Update()` function computes a partial `SHA2` digest on the *inlen*-byte message block pointed to by *input*, and updates the `SHA2` context structure pointed to by *context* accordingly.

The `SHA2Final()` function generates the final `SHA2Final` digest, using the `SHA2` context structure pointed to by *context*. The `SHA2` digest is written to output. After a call to `SHA2Final()`, the state of the context structure is undefined. It must be reinitialized with `SHA2Init()` before it can be used again.

`SHA256Init()`, `SHA256Update()`, `SHA256Final()`, `SHA384Init()`, `SHA384Update()`, `SHA384Final()`, `SHA512Init()`, `SHA512Update()`, `SHA512Final()`

Alternative APIs exist as named above. The `Update()` and `Final()` sets of functions operate exactly as the previously described `SHA2Update()` and `SHA2Final()` functions. The `SHA256Init()`, `SHA384Init()`, and `SHA512Init()` functions do not take the *mech* argument as it is implicit in the function names.

Return Values These functions do not return a value.

Examples **EXAMPLE 1** Authenticate a message found in multiple buffers

The following is a sample function that authenticates a message found in multiple buffers. The calling function provides an authentication buffer to contain the result of the `SHA2` digest.

```
#include <sys/types.h>
#include <sys/uio.h>
#include <sha2.h>

int
AuthenticateMsg(unsigned char *auth_buffer, struct iovec
                *messageIov, unsigned int num_buffers)
{
    SHA2_CTX sha2_context;
    unsigned int i;

    SHA2Init(SHA384, &sha2_context);

    for(i=0; i<num_buffers; i++)
    {
```

EXAMPLE 1 Authenticate a message found in multiple buffers *(Continued)*

```
        SHA2Update(&sha2_context, messageIov->iov_base,
                  messageIov->iov_len);
        messageIov += sizeof(struct iovec);
    }

    SHA2Final(auth_buffer, &sha2_context);

    return 0;
}
```

EXAMPLE 2 Authenticate a message found in multiple buffers

The following is a sample function that authenticates a message found in multiple buffers. The calling function provides an authentication buffer that will contain the result of the SHA384 digest, using alternative interfaces.

```
int
AuthenticateMsg(unsigned char *auth_buffer, struct iovec
                *messageIov, unsigned int num_buffers)
{
    SHA384_CTX ctx;
    unsigned int i;

    SHA384Init(&ctx);

    for(i=0, i<num_buffers; i++
    {
        SHA384Update(&ctx, messageIov->iov_base,
                  messageIov->iov_len);
        messageIov += sizeof(struct iovec);
    }

    SHA384Final(auth_buffer, &ctx);

    return 0;
}
```

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also libmd(3LIB)

FIPS 180–2

Name signbit – test sign

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

int signbit(real-floating x);
```

Description The `signbit()` macro determines whether the sign of its argument value is negative. NaNs, zeros, and infinities have a sign bit.

Return Values The `signbit()` macro returns a non-zero value if and only if the sign of its argument value is negative.

Errors No errors are defined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also `fpclassify(3M)`, `isfinite(3M)`, `isinf(3M)`, `isnan(3M)`, `isnormal(3M)`, `math.h(3HEAD)`, `attributes(5)`, `standards(5)`

Name significand, significandf, significandl – significand function

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double significand(double x);

float significandf(float x);

long double significandl(long double x);
```

Description If x equals $sig * 2^n$ with $1 \leq sig < 2$, then these functions return sig.

Return Values Upon successful completion, these functions return *sig*.

If x is either 0, ±Inf or NaN, x is returned.

Errors No errors are defined.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Stable
MT-Level	MT-Safe

See Also logb(3M), scalb(3M), attributes(5)

REFERENCE

Extended Library Functions - Part 6

Name sin, sinf, sinl – sine function

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double sin(double x);

float sinf(float x);

long double sinl(long double x);
```

Description These functions compute the sine of its argument *x*, measured in radians.

Return Values Upon successful completion, these functions return the sine of *x*.

If *x* is NaN, a NaN is returned.

If *x* is ±0, *x* is returned.

If *x* is ±Inf, a domain error occurs and a NaN is returned.

Errors These functions will fail if:

Domain Error — The *x* argument is ±Inf.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, then the invalid floating-point exception is raised.

Usage An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. An application should either examine the return value or check the floating point exception flags to detect exceptions.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also `asin(3M)`, `feclearexcept(3M)`, `fetestexcept(3M)`, `isnan(3M)`, `math.h(3HEAD)`, `attributes(5)`, `standards(5)`

Name sincos, sincosf, sincosl – combined sine and cosine function

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

void sincos(double x, double *s, double *c);

void sincosf(float x, float *s, float *c);

void sincosl(long double x, long double *s, long double *c);
```

Description These functions compute the sine and cosine of the first argument *x*, measured in radians.

Return Values Upon successful completion, these functions return the sine of *x* in **s* and cosine of *x* in **c*.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Stable
MT-Level	MT-Safe

See Also cos(3M), sin(3M), math.h(3HEAD), attributes(5)

Name sinh, sinhf, sinhl – hyperbolic sine function

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double sinh(double x);

float sinhf(float x);

long double sinhl(long double x);
```

Description These functions compute the hyperbolic sine of *x*.

Return Values Upon successful completion, these functions return the hyperbolic sine of *x*.

If the result would cause an overflow, a range error occurs and ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL (with the same sign as *x*) is returned as appropriate for the type of the function.

If *x* is NaN, a NaN is returned.

If *x* is ±0 or ±Inf, *x* is returned.

For exceptional cases, matherr(3M) tabulates the values to be returned by acos() as specified by SVID3 and XPG3.

Errors These functions will fail if:

Range Error — The result would cause an overflow.

If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, the overflow floating-point exception is raised.

The asinh() function sets errno to ERANGE if the result would cause an overflow.

Usage An application wanting to check for exceptions should call feclearexcept(FE_ALL_EXCEPT) before calling these functions. On return, if fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an exception has been raised. An application should either examine the return value or check the floating point exception flags to detect exceptions.

An application can also set errno to 0 before calling asinh(). On return, if errno is non-zero, an error has occurred. The asinhf() and asinhl() functions do not set errno.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also asinh(3M), cosh(3M), feclearexcept(3M), fetestexcept(3M), isnan(3M), math.h(3HEAD), matherr(3M), tanh(3M), attributes(5), standards(5)

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 sqrt, sqrtf, sqrtl – square root function

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double sqrt(double x);

float sqrtf(float x);

long double sqrtl(long double x);
```

Description These functions compute the square root of their argument *x*.

Return Values Upon successful completion, these functions return the square root of *x*.

For finite values of $x < -0$, a domain error occurs and either a NaN (if supported) or an implementation-defined value is returned.

If *x* is NaN, a NaN is returned.

If *x* is ±0 or +Inf, *x* is returned.

If *x* is –Inf, a domain error occurs and a NaN is returned.

Errors These functions will fail if:

Domain Error — The finite value of *x* is < –0 or *x* is –Inf.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, the invalid floating-point exception is raised.

The `sqrt()` function sets `errno` to `EDOM` if the value of *x* is negative.

Usage An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. An application should either examine the return value or check the floating point exception flags to detect exceptions.

An application can also set `errno` to 0 before calling `sqrt()`. On return, if `errno` is non-zero, an error has occurred. The `sqrtf()` and `sqrtl()` functions do not set `errno`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also feclearexcept(3M), fetestexcept(3M), isnan(3M), math.h(3HEAD), attributes(5), standards(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
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
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
MT-Level	Unsafe

See Also `attributes(5)`

Name stdarg – handle variable argument list

Synopsis

```
#include <stdarg.h>
va_list pvar;

void va_start(va_list pvar, void name);

(type *) va_arg(va_list pvar, type);

void va_copy(va_list dest, va_list src);

void va_end(va_list pvar);
```

Description This set of macros allows portable procedures that accept variable numbers of arguments of variable types to be written. Routines that have variable argument lists (such as `printf`) but do not use *stdarg* are inherently non-portable, as different machines use different argument-passing conventions.

`va_list` is a type defined for the variable used to traverse the list.

The `va_start` macro is invoked before any access to the unnamed arguments and initializes *pvar* for subsequent use by `va_arg()` and `va_end()`. The parameter *name* is the identifier of the rightmost parameter in the variable parameter list in the function definition (the one just before the `, ...`). If this parameter is declared with the `register` storage class or with a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions, the behavior is undefined.

The parameter *name* is required under strict ANSI C compilation. In other compilation modes, *name* need not be supplied and the second parameter to the `va_start()` macro can be left empty (for example, `va_start(pvar, );`). This allows for routines that contain no parameters before the `...` in the variable parameter list.

The `va_arg()` macro expands to an expression that has the type and value of the next argument in the call. The parameter `pvar` should have been previously initialized by `va_start()`. Each invocation of `va_arg()` modifies `pvar` so that the values of successive arguments are returned in turn. The parameter `type` is the type name of the next argument to be returned. The type name must be specified in such a way so that the type of a pointer to an object that has the specified type can be obtained simply by postfixing a * to *type*. If there is no actual next argument, or if *type* is not compatible with the type of the actual next argument (as promoted according to the default argument promotions), the behavior is undefined.

The `va_copy()` macro saves the state represented by the `va_list`*src* in the `va_list` *dest*. The `va_list` passed as *dest* should not be initialized by a previous call to `va_start()`, and must be passed to `va_end()` before being reused as a parameter to `va_start()` or as the *dest* parameter of a subsequent call to `va_copy()`. The behavior is undefined should any of these restrictions not be met.

The `va_end()` macro is used to clean up.

Multiple traversals, each bracketed by `va_start()` and `va_end()`, are possible.

Examples

EXAMPLE 1 A sample program.

This example gathers into an array a list of arguments that are pointers to strings (but not more than MAXARGS arguments) with function f1, then passes the array as a single argument to function f2. The number of pointers is specified by the first argument to f1.

```
#include <stdarg.h>
#define MAXARGS    31
void f1(int n_ptrs, ...)
{
     va_list ap;
     char *array[MAXARGS];
     int ptr_no = 0;

     if (n_ptrs > MAXARGS)
          n_ptrs = MAXARGS;
     va_start(ap, n_ptrs);
     while (ptr_no < n_ptrs)
          array[ptr_no++] = va_arg(ap, char*);
     va_end(ap);
     f2(n_ptrs, array);
}
```

Each call to f1 shall have visible the definition of the function or a declaration such as

```
void f1(int, ...)
```

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

See Also

vprintf(3C), attributes(5), standards(5)

Notes

It is the responsibility of the calling routine to specify in some manner how many arguments there are, since it is not always possible to determine the number of arguments from the stack frame. For example, *execl* is passed a zero pointer to signal the end of the list. The *printf* function can determine the number of arguments by the format. It is non-portable to specify a second argument of char, short, or float to va_arg(), because arguments seen by the called function are not char, short, or float. C converts char and short arguments to int and converts float arguments to double before passing them to a function.

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

strccpy, streadd, strcadd, strecpy – copy strings, compressing or expanding escape codes

Synopsis

```
cc [ flag ... ] file ... -lgen [ library ... ]
#include <libgen.h>

char *strccpy(char *output, const char *input);

char *strcadd(char *output, const char *input);

char *strecpy(char *output, const char *input, const char *exceptions);

char *streadd(char *output, const char *input, const char *exceptions);
```

Description

`strccpy()` copies the *input* string, up to a null byte, to the *output* string, compressing the C-language escape sequences (for example, `\n`, `\001`) to the equivalent character. A null byte is appended to the output. The *output* argument must point to a space big enough to accommodate the result. If it is as big as the space pointed to by *input* it is guaranteed to be big enough. `strccpy()` returns the *output* argument.

`strcadd()` is identical to `strccpy()`, except that it returns the pointer to the null byte that terminates the output.

`strecpy()` copies the *input* string, up to a null byte, to the *output* string, expanding non-graphic characters to their equivalent C-language escape sequences (for example, `\n`, `\001`). The *output* argument must point to a space big enough to accommodate the result; four times the space pointed to by *input* is guaranteed to be big enough (each character could become \ and 3 digits). Characters in the *exceptions* string are not expanded. The *exceptions* argument may be zero, meaning all non-graphic characters are expanded. `strecpy()` returns the *output* argument.

`streadd()` is identical to `strecpy()`, except that it returns the pointer to the null byte that terminates the output.

Examples

EXAMPLE 1 Example of expanding and compressing escape codes.

```
/* expand all but newline and tab */
strecpy( output, input, "\n\t" );

/* concatenate and compress several strings */
cp = strcadd( output, input1 );
cp = strcadd( cp, input2 );
cp = strcadd( cp, input3 );
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `string(3C)`, `strfind(3GEN)`, `attributes(5)`

Notes When compiling multi-thread applications, the `_REENTRANT` flag must be defined on the compile line. This flag should only be used in multi-thread applications.

Name strfind, strrspn, strtrns – string manipulations

Synopsis

```
cc [ flag ... ] file ... -lgen [ library ... ]
#include <libgen.h>

int strfind(const char *as1, const char *as2);

char *strrspn(const char *string, const char *tc);

char * strtrns(const char *string, const char *old,
     const char *new, char *result);
```

Description The `strfind()` function returns the offset of the first occurrence of the second string, *as2*, if it is a substring of string *as1*. If the second string is not a substring of the first string `strfind()` returns −1.

The `strrspn()` function trims chartacters from a string. It searches from the end of *string* for the first character that is not contained in *tc*. If such a character is found, `strrspn()` returns a pointer to the next character; otherwise, it returns a pointer to *string*.

The `strtrns()` function transforms *string* and copies it into *result*. Any character that appears in *old* is replaced with the character in the same position in *new*. The *new* result is returned.

Usage When compiling multithreaded applications, the `_REENTRANT` flag must be defined on the compile line. This flag should only be used in multithreaded applications.

Examples **EXAMPLE 1** An example of the `strfind()` function.

```
/* find offset to substring "hello" within as1 */
i = strfind(as1, "hello");
/* trim junk from end of string */
s2 = strrspn(s1, "*?#$%");
*s2 = '\0';
/* transform lower case to upper case */
a1[] = "abcdefghijklmnopqrstuvwxyz";
a2[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
s2 = strtrns(s1, a1, a2, s2);
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `string(3C)`, `attributes(5)`

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*.

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.

`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 *Oracle 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 Sun_MP_SendScsiCmd – send a SCSI command to a logical unit

Synopsis

```
cc [ flag... ] file... -lMPAPI [ library... ]
#include <mpapi.h>
#include <mpapi_sun.h>

MP_STATUS MP_SendScsiCmd(MP_OID oid, struct uscsi_cmd *cmd);
```

Parameters

oid The object ID of the logical unit path.

cmd A `uscsi_cmd` structure. See `uscsi(7I)`.

Description The `Sun_MP_SendScsiCmd()` function sends a SCSI command on a specific path to a logical unit. This function is applicable only to an OID whose `MP_PLUGIN_PROPERTIES` *driverVendor*, as defined by the Multipath Management API, is equal to "Sun Microsystems". See MP_GetPluginProperties(3MPAPI) and *Multipath Management API Version 1.0*.

Return Values

`MP_STATUS_INVALID_PARAMETER`
The *pProps* is null or specifies a memory area to which data cannot be written, or the *oid* has a type subfield other than `MP_OBJECT_TYPE_PLUGIN`.

`MP_STATUS_INVALID_OBJECT_TYPE`
The *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.

`MP_STATUS_OBJECT_NOT_FOUND`
The *oid* owner ID or object sequence number is invalid.

`MP_STATUS_SUCCESS`
The operation is successful.

Warnings The `uscsi` command is very powerful but somewhat dangerous. See the WARNINGS section on `attributes(5)` before using this interface.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libMPAPI(3LIB)`, MP_GetPluginProperties(3MPAPI), `attributes(5)`, `uscsi(7I)`

Multipath Management API Version 1.0

Name SUNW_C_GetMechSession, SUNW_C_KeyToObject – PKCS#11 Cryptographic Framework functions

Synopsis

```
cc [ flag ... ] file... -lpkcs11 [ library... ]
#include <security/cryptoki.h>
#include <security/pkcs11.h>

CK_RV SUNW_C_GetMechSession(CK_MECHANISM_TYPE mech,
     CK_SESSION_HANDLE_PTR hSession);

CK_RV SUNW_C_KeyToObject(CK_SESSION_HANDLE hSession,
     CK_MECHANISM_TYPE mech, const void *rawkey, size_t rawkey_len,
     CK_OBJECT_HANDLE_PTR obj);
```

Description These functions implement the RSA PKCS#11 v2.20 specification by using plug-ins to provide the slots.

The `SUNW_C_GetMechSession()` function initializes the PKCS#11 cryptographic framework and performs all necessary calls to Standard PKCS#11 functions (see `libpkcs11(3LIB)`) to create a session capable of providing operations on the requested mechanism. It is not neccessary to call `C_Initalize()` or `C_GetSlotList()` before the first call to `SUNW_C_GetMechSession()`.

If the `SUNW_C_GetMechSession()` function is called multiple times, it will return a new session each time without re-initalizing the framework. If it is unable to return a new session, `CKR_SESSION_COUNT` is returned.

The `C_CloseSession()` function should be called to release the session when it is no longer required.

The `SUNW_C_KeyToObject()` function creates a key object for the specified mechanism from the *rawkey* data. The object should be destroyed with `C_DestroyObject()` when it is no longer required.

Return Values The `SUNW_C_GetMechSession()` function returns the following values:

`CKR_OK`	The function completed successfully.
`CKR_SESSION_COUNT`	No sessions are available.
`CKR_ARGUMENTS_BAD`	A null pointer was passed for the return session handle.
`CKR_MECHANISM_INVALID`	The requested mechanism is invalid or no available plug-in provider supports it.
`CKR_FUNCTION_FAILED`	The function failed.
`CKR_GENERAL_ERROR`	A general error occurred.

The `SUNW_C_KeyToObject()` function returns the following values:

`CKR_OK`	The function completed successfully.

`CKR_ARGUMENTS_BAD`	A null pointer was passed for the session handle or the key material.
`CKR_MECHANISM_INVALID`	The requested mechanism is invalid or no available plug-in provider supports it.
`CKR_FUNCTION_FAILED`	The function failed.
`CKR_GENERAL_ERROR`	A general error occurred.

The return values of each of the implemented functions are defined and listed in the RSA PKCS#11 v2.20 specification. See http://www.rsasecurity.com.

Usage These functions are not part of the RSA PKCS#11 v2.20 specification. They are not likely to exist on non-Solaris systems. They are provided as a convenience to application programmers. Use of these functions will make the application non-portable to other systems.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

See Also `libpkcs11(3LIB)`, `attributes(5)`

http://www.rsasecurity.com

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	Evolving
MT-Level	MT-Safe

See Also syseventd(1M), sysevent_subscribe_event(3SYSEVENT), attributes(5)

Notes Event notifications are revoked by syseventd 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.

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	Evolving
MT-Level	MT-Safe

See Also `attributes(5)`

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	Evolving
MT-Level	MT-Safe

See Also `syseventd(1M)`, `nvlist_alloc(3NVPAIR)`, `nvlist_lookup_boolean(3NVPAIR)`, `attributes(5)`

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	Evolving
MT-Level	MT-Safe

See Also

attributes(5)

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	Evolving
MT-Level	MT-Safe

See Also

`malloc(3MALLOC)`, `attributes(5)`

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 `syseventd(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 `syseventd` 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	Evolving
MT-Level	MT-Safe

See Also

devfsadm(1M), syseventd(1M), nvlist_add_boolean(3NVPAIR), nvlist_alloc(3NVPAIR), attributes(5)

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(1M), sysevent_bind_handle(3SYSEVENT), sysevent_get_attr_list(3SYSEVENT), sysevent_get_class_name(3SYSEVENT), sysevent_get_vendor_name(3SYSEVENT), attributes(5)

Name tan, tanf, tanl – tangent function

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double tan(double x);

float tanf(float x);

long double tanl(long double x);
```

Description These functions compute the tangent of their argument *x*, measured in radians.

Return Values Upon successful completion, these functions return the tangent of *x*.

If *x* is NaN, a NaN is returned.

If *x* is ±0, *x* is returned.

If *x* is ±Inf, a domain error occurs and a NaN is returned.

Errors These functions will fail if:

Domain Error — The value of *x* is ±Inf.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, the invalid floating-point exception is raised.

Usage There are no known floating-point representations such that for a normal argument, `tan`(*x*) is either overflow or underflow.

An application wanting to check for exceptions should call `feclearexcept`(`FE_ALL_EXCEPT`) before calling these functions. On return, if `fetestexcept`(`FE_INVALID` | `FE_DIVBYZERO` | `FE_OVERFLOW` | `FE_UNDERFLOW`) is non-zero, an exception has been raised. An application should either examine the return value or check the floating point exception flags to detect exceptions.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also `atan(3M)`, `feclearexcept(3M)`, `fetestexcept(3M)`, `isnan(3M)`, `math.h(3HEAD)`, `attributes(5)`, `standards(5)`

Name tanh, tanhf, tanhl – hyperbolic tangent function

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double tanh(double x);

float tanhf(float x);

long double tanhl(long double x);
```

Description These functions compute the hyperbolic tangent of their argument *x*.

Return Values Upon successful completion, these functions return the hyperbolic tangent of *x*.

If *x* is NaN, a NaN is returned.

If *x* is ±0, *x* is returned.

If *x* is ±Inf, ±1 is returned.

Errors No errors are defined.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also atanh(3M), isnan(3M), math.h(3HEAD), tan(3M), attributes(5), standards(5)

Name Task – Perl interface to Tasks

Synopsis

```
use Sun::Solaris::Task qw(:ALL);
my $taskid = gettaskid();
```

Description This module provides wrappers for the gettaskid(2) and settaskid(2) system calls.

Constants `TASK_NORMAL`, `TASK_FINAL`.

Functions

`settaskid($project, $flags)` The `$project` parameter must be a valid project ID and the `$flags` parameter must be `TASK_NORMAL` or `TASK_FINAL`. The parameters are passed through directly to the underlying `settaskid()` system call. The new task ID is returned if the call succeeds. On failure −1 is returned.

`gettaskid()` This function returns the numeric task ID of the calling process, or `undef` if the underlying `gettaskid()` system call is unsuccessful.

Class methods None.

Object methods None.

Exports By default nothing is exported from this module. The following tags can be used to selectively import constants and functions defined in this module:

`:SYSCALLS`	`settaskid()` and `gettaskid()`
`:CONSTANTS`	`TASK_NORMAL` and `TASK_FINAL`
`:ALL`	`:SYSCALLS` and `:CONSTANTS`

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWpl5u
Interface Stability	Evolving

See Also gettaskid(2), settaskid(2), attributes(5)

Name tgamma, tgammaf, tgammal – compute gamma function

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double tgamma(double x);

float tgammaf(float x);

long double tgammal(long double x);
```

Description These functions compute the `gamma()` function of *x*.

Return Values Upon successful completion, these functions return `gamma`(*x*).

If *x* is a negative integer, a domain error occurs and a NaN is returned.

If the correct value would cause overflow, a range error occurs and `tgamma()`, `tgammaf()`, and `tgammal()` return the value of the macro ±`HUGE_VAL`, ±`HUGE_VALF`, or ±`HUGE_VALL`, respectively.

If *x* is NaN, a NaN is returned.

If *x* is ±Inf, *x* is returned.

If x is ±0, a pole error occurs and `tgamma()`, `tgammaf()`, and `tgammal()` return ±`HUGE_VAL`, ±`HUGE_VALF`, and ±`HUGE_VALL`, respectively.

If *x* is +Inf, a domain error occurs and a NaN is returned.

Errors These functions will fail if:

Domain Error — The value of *x* is a negative integer or *x* is –Inf.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, then the invalid floating-point exception is raised.

Pole Error — The value of *x* is zero.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, then the divide–by-zero floating-point exception is raised.

Range Error — The value overflows.

If the integer expression (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, then the overflow floating-point exception is raised.

Usage An application wanting to check for exceptions should call `feclearexcept`(`FE_ALL_EXCEPT`) before calling these functions. On return, if `fetestexcept`(`FE_INVALID` | `FE_DIVBYZERO` | `FE_OVERFLOW` | `FE_UNDERFLOW`) is non-zero, an exception has been raised. An application should either examine the return value or check the floating point exception flags to detect

exceptions.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also feclearexcept(3M), fetestexcept(3M), lgamma(3M), math.h(3HEAD), attributes(5), standards(5)

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 (see cc(1B)), 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 don't 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` (see cc(1B)), 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` (see cc(1B)), 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
MT Level	MT-Safe

See Also cc(1B), 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
prex> ktrace on                # turn on kernel tracing
```

EXAMPLE 2 Tracing the Kernel *(Continued)*

```
<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 trunc, truncf, truncl – round to truncated integer value

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double trunc(double x);

float truncf(float x);

long double truncl(long double x);
```

Description These functions round their argument to the integer value, in floating format, nearest to but no larger in magnitude than the argument.

Return Values Upon successful completion, these functions return the truncated integer value.

If *x* is NaN, a NaN is returned.

If *x* is ±0 or ±Inf, *x* is returned.

Errors No errors are defined.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also math.h(3HEAD), attributes(5), standards(5)

Name tsalarm_get, tsalarm_set – get or set alarm relays

Synopsis

```
cc [ flag... ] file... -ltsalarm [ library... ]
#include <tsalarm.h>

int tsalarm_get(uint32_t alarm_type, uint32_t *alarm_state);

int tsalarm_set(uint32_t alarm_type, uint32_t alarm_state);
```

Parameters

alarm_type

The alarm type whose state is retrieved or set. Valid settings are:

`TSALARM_CRITICAL`	critical
`TSALARM_MAJOR`	major
`TSALARM_MINOR`	minor
`TSALARM_USER`	user

alarm_state

The state of the alarm. Valid settings are:

`TSALARM_STATE_ON`	The alarm state needs to be changed to “on”, or is returned as “on”.
`TSALARM_STATE_OFF`	The alarm state needs to be changed to “off”, or is returned as “off”.
`TSALARM_STATE_UNKNOWN`	The alarm state is returned as unknown.

Description The TSALARM interface provides functions through which alarm relays can be controlled. The set of functions and data structures of this interface are defined in the `<tsalarm.h>` header.

There are four alarm relays that are controlled by ILOM. Each alarm can be set to “on” or “off” by using `tsalarm` interfaces provided from the host. The four alarms are labeled as `critical`, `major`, `minor`, and `user`. The user alarm is set by a user application depending on system condition. LEDs in front of the box provide a visual indication of the four alarms. The number of alarms and their meanings and labels can vary across platforms.

The `tsalarm_get()` function gets the state of *alarm_type* and returnsit in *alarm_state*. If successful, the function returns 0.

The `tsalarm_set()` function sets the state of *alarm_type* to the value in *alarm_state*. If successful, the function returns 0.

The following structures are defined in `<tsalarm.h>`:

```
typedef struct tsalarm_req {
     uint32_t     alarm_id;
```

```
     uint32_t     alarm_action;
} tsalarm_req_t;

typedef struct tsalarm_resp {
     uint32_t     status;
     uint32_t     alarm_id;
     uint32_t     alarm_state;
} tsalarm_resp_t;
```

Return Values The `tsalarm_get()` and `tsalarm_set()` functions return the following values:

`TSALARM_CHANNEL_INIT_FAILURE`	Channel initialization failed.
`TSALARM_COMM_FAILURE`	Channel communication failed.
`TSALARM_NULL_REQ_DATA`	Allocating memory for request data failed.
`TSALARM_SUCCESS`	Successful completion.
`TSALARM_UNBOUND_PACKET_RECVD`	An incorrect packet was received.

The `tsalarm_get()` function returns the following value:

`TSALARM_GET_ERROR`	An error occurred while getting the alarm state.

The `tsalarm_set()` function returns the following value:

`TSALARM_SET_ERROR`	An error occurred while setting the alarm state.

Examples **EXAMPLE 1** Get and set an alarm state.

The following example demonstrates how to get and set an alarm state.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <tsalarm.h>

void help(char *name) {
   printf("Syntax:  %s [get <type> | set <type> <state>]\n\n", name);
   printf("         type  = { critical, major, minor, user }\n");
   printf("         state = { on, off }\n\n");

   exit(0);
}

int main(int argc, char **argv) {

   uint32_t alarm_type, alarm_state;

   if (argc < 3)
```

EXAMPLE 1 Get and set an alarm state. *(Continued)*

```
        help(argv[0]);

    if (strncmp(argv[2], "critical", 1) == 0)
        alarm_type = TSALARM_CRITICAL;
    else if (strncmp(argv[2], "major", 2) == 0)
        alarm_type = TSALARM_MAJOR;
    else if (strncmp(argv[2], "minor", 2) == 0)
        alarm_type = TSALARM_MINOR;
    else if (strncmp(argv[2], "user", 1) == 0)
        alarm_type = TSALARM_USER;
    else
        help(argv[0]);

    if (strncmp(argv[1], "get", 1) == 0) {
        tsalarm_get(alarm_type, &alarm_state);
        printf("alarm = %d\tstate = %d\n", alarm_type, alarm_state);
    }
    else if (strncmp(argv[1], "set", 1) == 0) {
        if (strncmp(argv[3], "on", 2) == 0)
            alarm_state = TSALARM_STATE_ON;
        else if (strncmp(argv[3], "off", 2) == 0)
            alarm_state = TSALARM_STATE_OFF;
        else
            help(argv[0]);

        tsalarm_set(alarm_type, alarm_state);
    }
    else {
        help(argv[0]);
    }

    return 0;
}
```

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Uncommitted
MT-Level	Safe

See Also

libtsalarm(3LIB), 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 *Oracle 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 Ucred – Perl interface to User Credentials

Synopsis

```
use Sun::Solaris::Ucred qw(:ALL);
```

Description This module provides wrappers for the Ucred-related system and library calls.

Constants None.

Functions

`ucred_get($pid)`	This function returns the credential of the process specified by `$pid` if the process exists and the calling process is permitted to obtain the credentials of that process.
`getpeerucred($fd)`	If `$fd` is a connected connection-oriented TLI endpoint, a connected `SOCK_STREAM`, or a `SOCK_SEQPKT` socket, `getpeerucred()` returns the user credential of the peer at the time the connection was established, if availble.
`ucred_geteuid($ucred)`	This function returns the effective uid of a user credential, if available.
`ucred_getruid($ucred)`	This function returns the real uid of a user credential, if available.
`ucred_getsuid($ucred)`	This function returns the saved uid of a user credential, if available.
`ucred_getegid($ucred)`	This function returns the effective group of a user credential, if available.
`ucred_getrgid($ucred)`	This function returns the real group of a user credential, if available.
`ucred_getsgid($ucred)`	This function returns the saved group of a user credential, if available.
`ucred_getgroups($ucred)`	This function returns the list of supplemental groups of a user credential, if available.An array of groups is returned in `ARRAY` context; the number of groups is returned in `SCALAR` context.
`ucred_getprivset($ucred, $which)`	This function returns the privilege set specified by `$which` of a user credential, if available.
`ucred_getpflags($ucred, $flags)`	This function returns the value of a specific process flag of a user credential, if available.
`ucred_getpid($ucred)`	This function returns the process ID of a user credential, if available.

`ucred_getprojid($ucred)`	This function returns the project ID of a user credential, if available.
`ucred_getzoneid($ucred)`	This function returns the zone ID of a user credential, if available.

Class methods None.

Object methods None.

Exports By default nothing is exported from this module. The following tags can be used to selectively import constants and functions defined in this module:

`:SYSCALLS`	`ucred_get(),getpeerucred()`
`:LIBCALLS`	`ucred_geteuid(),ucred_getruid(),ucred_getsuid(),ucred_getegid(), ucred_getrgid(),ucred_getsgid(),ucred_getgroups(), ucred_getprivset(),ucred_getpflags(),ucred_getpid(), ucred_getzone()`
`:ALL`	`:SYSCALLS(),:LIBCALLS()`

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWpl5u
Interface Stability	Evolving

See Also `getpeerucred(3C)`, `ucred_get(3C)`, `attributes(5)`

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 v12n, v12n_capabilities, v12n_domain_roles, v12n_domain_name, v12n_domain_uuid, v12n_ctrl_domain, v12n_chassis_serialno – return virtualization environment domain parameters

Synopsis

```
cc [ flag... ] file... -lv12n [ library... ]
#include <libv12n.h>

int v12n_capabilities();

int v12n_domain_roles();

int v12n_domain_uuid(uuid_t uuid);

size_t v12n_domain_name(char *buf, size_t buflen);

size_t v12n_ctrl_domain(char *buf, size_t buflen);

size_t v12n_chassis_serialno(char *buf, size_t buflen);
```

Description The `v12n_capabilities()` function returns the virtualization capabilities mask of the current domain. The virtualization capabilities bit mask consists of the following values:

`V12N_CAP_SUPPORTED`	Virtualization is supported on this domain.
`V12N_CAP_ENABLED`	Virtualization is enabled on this domain.
`V12N_CAP_IMPL_LDOMS`	Logical Domains is the supported virtualization implementation.

The `v12n_domain_roles()` function returns the virtualization domain role mask. The virtualization domain role mask consists of the following values:

`V12N_ROLE_CONTROL`	If the virtualization implementation is Logical Domains, and this bit is one, the current domain is a control domain. If this bit is zero, the current domain is a guest domain.
`V12N_ROLE_IO`	Current domain is an I/O domain.
`V12N_ROLE_SERVICE`	Current domain is a service domain.
`V12N_ROLE_ROOT`	Current domain is an root I/O domain.

The `v12n_domain_uuid()` function stores the universally unique identifier (UUID) for the current virtualization domain in the `uuid` argument. See the `libuuid(3LIB)` manual page.

The `v12n_domain_name()` function stores the name of the current virtualization domain in the location specified by *buf*. *buflen* specifies the size in bytes of the buffer. If the buffer is too small to hold the complete null-terminated name, the first *buflen* bytes of the name are stored in the buffer. A buffer of size `V12N_NAME_MAX` is sufficient to hold any domain name. If *buf* is NULL or *buflen* is 0, the name is not copied into the buffer.

The `v12n_ctrl_domain()` function stores the control domain or dom0 network node name of the current domain in the location specified by *buf*. Note that a domain's control domain is

volatile during a domain migration. The information returned by this function might be stale if the domain was in the process of migrating. *buflen* specifies the size in bytes of the buffer. If the buffer is too small to hold the complete null-terminated name, the first *buflen* bytes of the name are stored in the buffer. A buffer of size `V12N_NAME_MAX` is sufficient to hold the control domain node name string. If *buf* is NULL or *buflen* is 0, the name is not copied into the buffer.

The `v12n_chassis_serialno()` function stores the chassis serial number of the platform on which the current domain is running in the location specified by *buf*. Note that the chassis serial number is volatile during a domain migration. The information returned by this function might be stale if the domain was in the process of migrating. *buflen* specifies the size in bytes of the buffer. If the buffer is too small to hold the complete null-terminated name, the first *buflen* bytes of the name are stored in the buffer. A buffer of size `V12N_NAME_MAX` is sufficient to hold any chassis serial number string. If *buf* is NULL or *buflen* is 0, the name is not copied into the buffer.

Return Values On successful completion, the `v12n_capabilties()` and `v12n_domain_roles()` functions return a non-negative bit mask. Otherwise, the `v12n_domain_roles()` function returns -1 and sets `errno` to indicate the error.

On successful completion, the `v12n_domain_uuid()` function returns 0. Otherwise, the `v12n_domain_uuid()` function returns -1 and sets `errno` to indicate the error.

On successful completion, the `v12n_domain_name()`, `v12n_ctrl_domain()`, and `v12n_chassis_serialno()` functions return the buffer size required to hold the full non-terminated string. Otherwise, these functions return -1 and set `errno` to indicate the error.

Errors The `v12n_domain_roles()` function fails with `EPERM` when the calling process has an ID other than the privileged user.

The `v12n_domain_name()` function will fail if:

`EPERM` The calling process has an ID other than the privileged user.

`ENOTSUP` Virtualization is not supported or enabled on this domain.

`EFAULT` *buf* points to an illegal address.

`ENOENT` The `sun4v` machine description is inaccessible or has no `uuid` node.

The `v12n_domain_uuid()` function will fail if:

`EPERM` The calling process has an ID other than the privileged user.

`ENOTSUP` Virtualization is not supported or enabled on this domain.

`EFAULT` *buf* points to an illegal address.

`ENOENT` The `sun4v` machine description is inaccessible or has no `uuid` node.

The `v12n_ctrl_domain()` function will fail if:

`EPERM` The calling process has an ID other than the privileged user.

`ENOTSUP` Virtualization is not supported or enabled on this domain.

`EFAULT` *buf* points to an illegal address.

`ETIME` The domain service on the control domain did not respond within the timeout value.

The `v12n_chassis_serialno()` function will fail if:

`EPERM` The calling process has an ID other than the privileged user.

`ENOTSUP` Virtualization is not supported or enabled on this domain.

`EFAULT` *buf* points to an illegal address.

`ETIME` The domain service on the control domain did not respond within the timeout value.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also

`virtinfo(1M)`, `libuuid(3LIB)`, `libv12n(3LIB)`, `attributes(5)`

Name varargs – handle variable argument list

Synopsis

```
#include <varargs.h>
va_alist
va_dcl
va_list pvar;

void va_start(va_listpvar);

type va_arg(va_list pvar, type);

void va_end(va_list pvar);
```

Description This set of macros allows portable procedures that accept variable argument lists to be written. Routines that have variable argument lists (such as `printf(3C)`) but do not use `varargs` are inherently non-portable, as different machines use different argument-passing conventions.

`va_alist` is used as the parameter list in a function header.

`va_dcl` is a declaration for `va_alist`. No semicolon should follow `va_dcl`.

`va_list` is a type defined for the variable used to traverse the list.

`va_start` is called to initialize `pvar` to the beginning of the list.

`va_arg` will return the next argument in the list pointed to by `pvar`. `type` is the type the argument is expected to be. Different types can be mixed, but it is up to the routine to know what type of argument is expected, as it cannot be determined at runtime.

`va_end` is used to clean up.

Multiple traversals, each bracketed by `va_start` and `va_end`, are possible.

Examples **EXAMPLE 1** A sample program.

This example is a possible implementation of `execl` (see `exec(2)`).

```
#include <unistd.h>
#include <varargs.h>
#define MAXARGS    100
/*    execl is called by
        execl(file, arg1, arg2, ..., (char *)0);
*/
execl(va_alist)
va_dcl
{
    va_list ap;
    char *file;
    char *args[MAXARGS];        /* assumed big enough*/
    int argno = 0;
```

EXAMPLE 1 A sample program. *(Continued)*

```
    va_start(ap);
    file = va_arg(ap, char *);
    while ((args[argno++] = va_arg(ap, char *)) != 0)
        ;
    va_end(ap);
    return execv(file, args);
}
```

See Also exec(2), printf(3C), vprintf(3C), stdarg(3EXT)

Notes It is up to the calling routine to specify in some manner how many arguments there are, since it is not always possible to determine the number of arguments from the stack frame. For example, `execl` is passed a zero pointer to signal the end of the list. `printf` can tell how many arguments are there by the format.

It is non-portable to specify a second argument of `char`, `short`, or `float` to `va_arg`, since arguments seen by the called function are not `char`, `short`, or `float`. C converts `char` and `short` arguments to `int` and converts `float` arguments to `double` before passing them to a function.

`stdarg` is the preferred interface.

Name vatan2_, vatan2f_ – vector atan2 functions

Synopsis

```
cc [ flag... ] file... -lmvec [ library... ]

void vatan2_(int *n, double * restrict y, int *stridey,
     double * restrict x, int *stridex, double * restrict z,
     int *stridez);

void vatan2f_(int *n, float * restrict y, int *stridey,
     float * restrict x, int *stridex, float * restrict z,
     int *stridez);
```

Description These functions evaluate the function `atan2`(y, x) for an entire vector of values at once. The first parameter specifies the number of values to compute. Subsequent parameters specify the argument and result vectors. Each vector is described by a pointer to the first element and a stride, which is the increment between successive elements.

Specifically, `vatan2_` (n, y, sy, x, sx, z, sz) computes $z[i * *sz]$ = `atan2`($y[i * *sy]$, $x[i * *sx]$) for each $i = 0, 1, ..., *n - 1$. The `vatan2f_()` function performs the same computation for single precision data.

These functions are not guaranteed to deliver results that are identical to the results of the atan2(3M) functions given the same arguments. Non-exceptional results, however, are accurate to within a unit in the last place.

Usage The element count $*n$ must be greater than zero. The strides for the argument and result arrays can be arbitrary integers, but the arrays themselves must not be the same or overlap. A zero stride effectively collapses an entire vector into a single element. A negative stride causes a vector to be accessed in descending memory order, but note that the corresponding pointer must still point to the first element of the vector to be used; if the stride is negative, this will be the highest-addressed element in memory. This convention differs from the Level 1 BLAS, in which array parameters always refer to the lowest-addressed element in memory even when negative increments are used.

These functions assume that the default round-to-nearest rounding direction mode is in effect. On x86, these functions also assume that the default round-to-64-bit rounding precision mode is in effect. The result of calling a vector function with a non-default rounding mode in effect is undefined.

These functions handle special cases and exceptions in the same way as the `atan2()` functions when `c99 MATHERREXCEPT` conventions are in effect. See atan2(3M) for the results for special cases.

An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. The application can then examine the result or argument vectors for exceptional values. Some vector functions can raise the inexact exception even if all elements of the argument array are such that the

numerical results are exact.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also atan2(3M), feclearexcept(3M), fetestexcept(3M), attributes(5)

Name vatan_, vatanf_ – vector arctangent functions

Synopsis

```
cc [ flag... ] file... -lmvec [ library... ]

void vatan_(int *n,double * restrict x, int *stridex,
     double * restrict y, int *stridey);

void vatanf_(int *n, float * restrict x, int *stridex,
     float * restrict y, int *stridey);
```

Description These functions evaluate the function `atan`(*x*) for an entire vector of values at once. The first parameter specifies the number of values to compute. Subsequent parameters specify the argument and result vectors. Each vector is described by a pointer to the first element and a stride, which is the increment between successive elements.

Specifically, `vatan_`(*n*, *x*, *sx*, *y*, *sy*) computes *y*[*i* * *sy*] = `atan`(*x*[*i* * *sx*]) for each *i* = 0, 1, ..., **n* - 1. The `vatanf_()` function performs the same computation for single precision data.

These functions are not guaranteed to deliver results that are identical to the results of the `atan`(3M) functions given the same arguments. Non-exceptional results, however, are accurate to within a unit in the last place.

Usage The element count **n* must be greater than zero. The strides for the argument and result arrays can be arbitrary integers, but the arrays themselves must not be the same or overlap. A zero stride effectively collapses an entire vector into a single element. A negative stride causes a vector to be accessed in descending memory order, but note that the corresponding pointer must still point to the first element of the vector to be used; if the stride is negative, this will be the highest-addressed element in memory. This convention differs from the Level 1 BLAS, in which array parameters always refer to the lowest-addressed element in memory even when negative increments are used.

These functions assume that the default round-to-nearest rounding direction mode is in effect. On x86, these functions also assume that the default round-to-64-bit rounding precision mode is in effect. The result of calling a vector function with a non-default rounding mode in effect is undefined.

These functions handle special cases and exceptions in the same way as the `atan()` functions when `c99` `MATHERREXCEPT` conventions are in effect. See `atan`(3M) for the results for special cases.

An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. The application can then examine the result or argument vectors for exceptional values. Some vector functions can raise the inexact exception even if all elements of the argument array are such that the numerical results are exact.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also atan(3M), feclearexcept(3M), fetestexcept(3M), attributes(5)

Name vcos_, vcosf_ – vector cosine functions

Synopsis

```
cc [ flag... ] file... -lmvec [ library... ]

void vcos_(int *n, double * restrict x, int *stridex,
     double * restrict y, int *stridey);

void vcosf_(int *n, float * restrict x, int *stridex,
     float * restrict y, int *stridey);
```

Description These functions evaluate the function `cos`(*x*) for an entire vector of values at once. The first parameter specifies the number of values to compute. Subsequent parameters specify the argument and result vectors. Each vector is described by a pointer to the first element and a stride, which is the increment between successive elements.

Specifically, `vcos_`(*n*, *x*, *sx*, *y*, *sy*) computes *y*[*i* * *sy*] = `cos`(*x*[*i* * *sx*]) for each *i* = 0, 1, ..., **n* - 1. The `vcosf_()` function performs the same computation for single precision data.

These functions are not guaranteed to deliver results that are identical to the results of the cos(3M) functions given the same arguments. Non-exceptional results, however, are accurate to within a unit in the last place.

Usage The element count **n* must be greater than zero. The strides for the argument and result arrays can be arbitrary integers, but the arrays themselves must not be the same or overlap. A zero stride effectively collapses an entire vector into a single element. A negative stride causes a vector to be accessed in descending memory order, but note that the corresponding pointer must still point to the first element of the vector to be used; if the stride is negative, this will be the highest-addressed element in memory. This convention differs from the Level 1 BLAS, in which array parameters always refer to the lowest-addressed element in memory even when negative increments are used.

These functions assume that the default round-to-nearest rounding direction mode is in effect. On x86, these functions also assume that the default round-to-64-bit rounding precision mode is in effect. The result of calling a vector function with a non-default rounding mode in effect is undefined.

These functions handle special cases and exceptions in the same way as the `cos()` functions when `c99` `MATHERREXCEPT` conventions are in effect. See cos(3M) for the results for special cases.

An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. The application can then examine the result or argument vectors for exceptional values. Some vector functions can raise the inexact exception even if all elements of the argument array are such that the numerical results are exact.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also cos(3M), feclearexcept(3M), fetestexcept(3M), attributes(5)

Name vcospi_, vcospif_ – vector cospi functions

Synopsis

```
cc [ flag... ] file... -lmvec [ library... ]

void vcospi_(int *n, double * restrict x, int *stridex,
     double * restrict y, int *stridey);

void vcosfpi_(int *n, float * restrict x, int *stridex,
     float * restrict y, int *stridey);
```

Description These functions evaluate the function `cospi`(*x*), defined by `cospi`(*x*) = `cos`(π * *x*), for an entire vector of values at once. The first parameter specifies the number of values to compute. Subsequent parameters specify the argument and result vectors. Each vector is described by a pointer to the first element and a stride, which is the increment between successive elements.

Specifically, `vcospi_`(*n*, *x*, *sx*, *y*, *sy*) computes *y*[*i* * *sy*] = `cospi`(*x*[*i* * *sx*]) for each *i* = 0, 1, ..., **n* - 1. The `vcospif_()` function performs the same computation for single precision data.

Non-exceptional results are accurate to within a unit in the last place.

Usage The element count **n* must be greater than zero. The strides for the argument and result arrays can be arbitrary integers, but the arrays themselves must not be the same or overlap. A zero stride effectively collapses an entire vector into a single element. A negative stride causes a vector to be accessed in descending memory order, but note that the corresponding pointer must still point to the first element of the vector to be used; if the stride is negative, this will be the highest-addressed element in memory. This convention differs from the Level 1 BLAS, in which array parameters always refer to the lowest-addressed element in memory even when negative increments are used.

These functions assume that the default round-to-nearest rounding direction mode is in effect. On x86, these functions also assume that the default round-to-64-bit rounding precision mode is in effect. The result of calling a vector function with a non-default rounding mode in effect is undefined.

These functions handle special cases and exceptions in the spirit of IEEE 754. In particular,

- `cospi`(NaN) is NaN,
- `cospi`(±Inf) is NaN, and an invalid operation exception is raised.

An application wanting to check for exceptions should call `feclearexcept`(`FE_ALL_EXCEPT`) before calling these functions. On return, if `fetestexcept`(`FE_INVALID` | `FE_DIVBYZERO` | `FE_OVERFLOW` | `FE_UNDERFLOW`) is non-zero, an exception has been raised. The application can then examine the result or argument vectors for exceptional values. Some vector functions can raise the inexact exception even if all elements of the argument array are such that the numerical results are exact.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also feclearexcept(3M), fetestexcept(3M), attributes(5)

Name vexp_, vexpf_ – vector exponential functions

Synopsis

```
cc [ flag... ] file... -lmvec [ library... ]

void vexp_(int *n, double * restrict x, int *stridex,
     double * restrict y, int *stridey);

void vexpf_(int *n, float * restrict x, int *stridex,
     float * restrict y, int *stridey);
```

Description These functions evaluate the function exp(*x*) for an entire vector of values at once. The first parameter specifies the number of values to compute. Subsequent parameters specify the argument and result vectors. Each vector is described by a pointer to the first element and a stride, which is the increment between successive elements.

Specifically, vexp_(*n*, *x*, *sx*, *y*, *sy*) computes *y*[*i* * *sy*] = exp(*x*[*i* * *sx*]) for each *i* = 0, 1, ..., **n* - 1. The vexpf_() function performs the same computation for single precision data.

These functions are not guaranteed to deliver results that are identical to the results of the exp(3M) functions given the same arguments. Non-exceptional results, however, are accurate to within a unit in the last place.

Usage The element count **n* must be greater than zero. The strides for the argument and result arrays can be arbitrary integers, but the arrays themselves must not be the same or overlap. A zero stride effectively collapses an entire vector into a single element. A negative stride causes a vector to be accessed in descending memory order, but note that the corresponding pointer must still point to the first element of the vector to be used; if the stride is negative, this will be the highest-addressed element in memory. This convention differs from the Level 1 BLAS, in which array parameters always refer to the lowest-addressed element in memory even when negative increments are used.

These functions assume that the default round-to-nearest rounding direction mode is in effect. On x86, these functions also assume that the default round-to-64-bit rounding precision mode is in effect. The result of calling a vector function with a non-default rounding mode in effect is undefined.

On SPARC, the vexpf_() function delivers +0 rather than a subnormal result for arguments in the range –103.2789 <= *x* <= -87.3365. Otherwise, these functions handle special cases and exceptions in the same way as the exp() functions when c99 MATHERREXCEPT conventions are in effect. See exp(3M) for the results for special cases.

An application wanting to check for exceptions should call feclearexcept(FE_ALL_EXCEPT) before calling these functions. On return, if fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an exception has been raised. The application can then examine the result or argument vectors for exceptional values. Some vector functions can raise the inexact exception even if all elements of the argument array are such that the numerical results are exact.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also exp(3M), feclearexcept(3M), fetestexcept(3M), attributes(5)

Name vhypot_, vhypotf_ – vector hypotenuse functions

Synopsis

```
cc [ flag... ] file... -lmvec [ library... ]

void vhypot_(int *n, double * restrict x, int *stridex,
     double * restrict y, int *stridey, double * restrict z,
     int *stridez);

void vhypotf_(int *n, float * restrict x, int *stridex,
     float * restrict y, int *stridey, float * restrict z,
     int *stridez);
```

Description These functions evaluate the function `hypot`(*x*, *y*) for an entire vector of values at once. The first parameter specifies the number of values to compute. Subsequent parameters specify the argument and result vectors. Each vector is described by a pointer to the first element and a stride, which is the increment between successive elements.

Specifically, `vhypot_`(*n*, *x*, *sx*, *y*, *sy*, *z*, *sz*) computes *z*[*i* * *sz*] = `hypot`(*x*[*i* * *sx*], *y*[*i* * *sy*]) for each *i* = 0, 1, ..., **n* - 1. The `vhypotf_()` function performs the same computation for single precision data.

These functions are not guaranteed to deliver results that are identical to the results of the `hypot(3M)` functions given the same arguments. Non-exceptional results, however, are accurate to within a unit in the last place.

Usage The element count **n* must be greater than zero. The strides for the argument and result arrays can be arbitrary integers, but the arrays themselves must not be the same or overlap. A zero stride effectively collapses an entire vector into a single element. A negative stride causes a vector to be accessed in descending memory order, but note that the corresponding pointer must still point to the first element of the vector to be used; if the stride is negative, this will be the highest-addressed element in memory. This convention differs from the Level 1 BLAS, in which array parameters always refer to the lowest-addressed element in memory even when negative increments are used.

These functions assume that the default round-to-nearest rounding direction mode is in effect. On x86, these functions also assume that the default round-to-64-bit rounding precision mode is in effect. The result of calling a vector function with a non-default rounding mode in effect is undefined.

These functions handle special cases and exceptions in the same way as the `hypot()` functions when `c99 MATHERREXCEPT` conventions are in effect. See `hypot(3M)` for the results for special cases.

An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. The application can then examine the result or argument vectors for exceptional values. Some vector functions can raise the inexact exception even if all elements of the argument array are such that the

numerical results are exact.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also hypot(3M), feclearexcept(3M), fetestexcept(3M), attributes(5)

Name vlog_, vlogf_ – vector logarithm functions

Synopsis

```
cc [ flag... ] file... -lmvec [ library... ]

void vlog_(int *n, double * restrict x, int *stridex,
     double * restrict y, int *stridey);

void vlogf_(int *n, float * restrict x, int *stridex,
     float * restrict y, int *stridey);
```

Description These functions evaluate the function `log`(*x*) for an entire vector of values at once. The first parameter specifies the number of values to compute. Subsequent parameters specify the argument and result vectors. Each vector is described by a pointer to the first element and a stride, which is the increment between successive elements.

Specifically, `vlog_`(*n*, *x*, *sx*, *y*, *sy*) computes *y*[*i* * *sy*] = `log`(*x*[*i* * *sx*]) for each *i* = 0, 1, ..., **n* - 1. The `vlogf_()` function performs the same computation for single precision data.

These functions are not guaranteed to deliver results that are identical to the results of the log(3M) functions given the same arguments. Non-exceptional results, however, are accurate to within a unit in the last place.

Usage The element count **n* must be greater than zero. The strides for the argument and result arrays can be arbitrary integers, but the arrays themselves must not be the same or overlap. A zero stride effectively collapses an entire vector into a single element. A negative stride causes a vector to be accessed in descending memory order, but note that the corresponding pointer must still point to the first element of the vector to be used; if the stride is negative, this will be the highest-addressed element in memory. This convention differs from the Level 1 BLAS, in which array parameters always refer to the lowest-addressed element in memory even when negative increments are used.

These functions assume that the default round-to-nearest rounding direction mode is in effect. On x86, these functions also assume that the default round-to-64-bit rounding precision mode is in effect. The result of calling a vector function with a non-default rounding mode in effect is undefined.

These functions handle special cases and exceptions in the same way as the `log()` functions when `c99` `MATHERREXCEPT` conventions are in effect. See log(3M) for the results for special cases.

An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. The application can then examine the result or argument vectors for exceptional values. Some vector functions can raise the inexact exception even if all elements of the argument array are such that the numerical results are exact.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also log(3M), feclearexcept(3M), fetestexcept(3M), 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 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. See `vold(1M)`.

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, `/vol/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 Upon successful completion, `volmgt_acquire()` returns a non-zero value.

Upon failure, `volmgt_acquire()` returns `0`. If the return value is `0`, and `errno` is set to `EBUSY`, the address pointed to by *err* contains the string that was specified as `id` (when the device was reserved by the process holding the reservation).

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

See Also vold(1M), free(3C), malloc(3C), volmgt_release(3VOLMGT), attributes(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 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 any "actions" specified in `/etc/vold.conf` (see `vold.conf(4)`).

Return Values This routine returns `0` if no media was found, and a non-zero value if any media was found.

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 specified in `/etc/vold.conf` for any media found.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `cc(1B)`, `volcheck(1)`, `vold(1M)`, `open(2)`, `stat(2)`, `volmgt_inuse(3VOLMGT)`, `volmgt_running(3VOLMGT)`, `vold.conf(4)`, `attributes(5)`, `volfs(7FS)`

Notes Volume Management must be running for this routine to work.

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 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 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()`.

The list of features that `volmgt_feature_enabled()` checks for is expected to expand in the future.

Return Values `0` is returned if the specified feature is not currently available. A non-zero value indicates that the specified feature is currently available.

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

See Also `volmgt_acquire(3VOLMGT)`, `volmgt_release(3VOLMGT)`, `attributes(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 `volmgt_inuse()` checks whether Volume Management is managing the specified *pathname*.

Return Values A non-zero value is returned if Volume Management is managing the specified *pathname*, otherwise `0` is returned.

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

See Also `cc(1B)`, `vold(1M)`, `open(2)`, `stat(2)`, `errno(3C)`, `volmgt_check(3VOLMGT)`, `volmgt_running(3VOLMGT)`, `attributes(5)`, `volfs(7FS)`

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 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 `volgmt_ownspath()` function returns a non-zero value if *path* is owned by Volume Management. It returns 0 if *path* is not in its name space or Volume Management is not running.

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
Commitment Level	Public

See Also `volmgt_root(3VOLMGT)`, `volmgt_running(3VOLMGT)`, `attributes(5)`

Name volmgt_release – release removable media device reservation

Synopsis

```
cc [ flag ... ] file ... -lvolmgt [ library ... ]
#include <volmgt.h>

int volmgt_release(char *dev);
```

Description 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 Upon successful completion, `volmgt_release` returns a non-zero value. Upon failure, `0` is returned.

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	Stable

See Also vold(1M), volmgt_acquire(3VOLMGT), attributes(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 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 `volmgt_root()` function returns pointer to a static string containing the root directory for Volume Management.

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 `/vol` 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

See Also `cc(1B)`, `vold(1M)`, `open(2)`, `volmgt_check(3VOLMGT)`, `volmgt_inuse(3VOLMGT)`, `volmgt_running(3VOLMGT)`, `attributes(5)`, `volfs(7FS)`

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 `volmgt_running()` tells whether or not Volume Management is running.

Return Values A non-zero value is returned if Volume Management is running, else `0` is returned.

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

See Also `cc(1B)`, `vold(1M)`, `open(2)`, `stat(2)`, `volmgt_check(3VOLMGT)`, `volmgt_inuse(3VOLMGT)`, `attributes(5)`, `volfs(7FS)`

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 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 (see `volfs(7FS)` for a description of Volume Management symbolic names).

`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 `volmgt_symname( )` returns the symbolic name for the device pathname supplied, and `volmgt_symdev( )` returns the device pathname for the supplied symbolic name.

These strings are allocated upon success, and therefore must be freed by the caller when they are no longer needed (see `free(3C)`).

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) {
```

EXAMPLE 1 Testing Floppies *(Continued)*

```
                (void) printf("volume %s is in drive %d\n",
                    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

See Also

`cc(1B)`, `vold(1M)`, `open(2)`, `stat(2)`, `free(3C)`, `malloc(3C)`, `volmgt_check(3VOLMGT)`, `volmgt_inuse(3VOLMGT)`, `volmgt_running(3VOLMGT)`, `attributes(5)`, `volfs(7FS)`

Notes

These routines only work when Volume Management is running.

Bugs

There should be a straightforward way to query Volume Management for a list of all media types it's managing, and how many of each type are being managed.

Name vpow_, vpowf_ – vector power functions

Synopsis

```
cc [ flag... ] file... -lmvec [ library... ]

void vpow_(int *n, double * restrict x, int *stridex,
     double * restrict y, int *stridey, double * restrict z,
     int *stridez);

void vpowf_(int *n, float * restrict x, int *stridex,
     float * restrict y, int *stridey, float * restrict z,
     int *stridez);
```

Description These functions evaluate the function pow(*x*, *y*) for an entire vector of values at once. The first parameter specifies the number of values to compute. Subsequent parameters specify the argument and result vectors. Each vector is described by a pointer to the first element and a stride, which is the increment between successive elements.

Specifically, vpow_(*n*, *x*, *sx*, *y*, *sy*, *z*, *sz*) computes *z*[*i* * *sz*] = pow(*x*[*i* * *sx*], *y*[*i* * *sy*]) for each *i* = 0, 1, ..., **n* - 1. The vpowf_() function performs the same computation for single precision data.

These functions are not guaranteed to deliver results that are identical to the results of the pow(3M) functions given the same arguments. Non-exceptional results, however, are accurate to within a unit in the last place.

Usage The element count **n* must be greater than zero. The strides for the argument and result arrays can be arbitrary integers, but the arrays themselves must not be the same or overlap. A zero stride effectively collapses an entire vector into a single element. A negative stride causes a vector to be accessed in descending memory order, but note that the corresponding pointer must still point to the first element of the vector to be used; if the stride is negative, this will be the highest-addressed element in memory. This convention differs from the Level 1 BLAS, in which array parameters always refer to the lowest-addressed element in memory even when negative increments are used.

These functions assume that the default round-to-nearest rounding direction mode is in effect. On x86, these functions also assume that the default round-to-64-bit rounding precision mode is in effect. The result of calling a vector function with a non-default rounding mode in effect is undefined.

These functions handle special cases and exceptions in the same way as the pow() functions when c99 MATHERREXCEPT conventions are in effect. See pow(3M) for the results for special cases.

An application wanting to check for exceptions should call feclearexcept(FE_ALL_EXCEPT) before calling these functions. On return, if fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an exception has been raised. The application can then examine the result or argument vectors for exceptional values. Some vector functions can raise the inexact exception even if all elements of the argument array are such that the

numerical results are exact.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also pow(3M), feclearexcept(3M), fetestexcept(3M), attributes(5)

Name vrhypot_, vrhypotf_ – vector reciprocal hypotenuse functions

Synopsis

```
cc [ flag... ] file... -lmvec [ library... ]

void vrhypot_(int *n, double * restrict x, int *stridex,
     double * restrict y, int *stridey, double * restrict z,
     int *stridez);

void vrhypotf_(int *n, float * restrict x, int *stridex,
     float * restrict y, int *stridey, float * restrict z,
     int *stridez);
```

Description These functions evaluate the function rhypot(*x*, *y*), defined by rhypot(*x*, *y*) = 1 / hypot(*x*, *y*), for an entire vector of values at once. The first parameter specifies the number of values to compute. Subsequent parameters specify the argument and result vectors. Each vector is described by a pointer to the first element and a stride, which is the increment between successive elements.

Specifically, vrhypot_(*n*, *x*, *sx*, *y*, *sy*, *z*, *sz*) computes *z*[*i* * *sz*] = rhypot(*x*[*i* * *sx*], *y*[*i* * *sy*]) for each *i* = 0, 1, ..., **n* - 1. The vrhypotf_() function performs the same computation for single precision data.

These functions are not guaranteed to deliver results that are identical to the results of evaluating 1.0 / hypot(*x*, *y*) given the same arguments. Non-exceptional results, however, are accurate to within a unit in the last place.

Usage The element count **n* must be greater than zero. The strides for the argument and result arrays can be arbitrary integers, but the arrays themselves must not be the same or overlap. A zero stride effectively collapses an entire vector into a single element. A negative stride causes a vector to be accessed in descending memory order, but note that the corresponding pointer must still point to the first element of the vector to be used; if the stride is negative, this will be the highest-addressed element in memory. This convention differs from the Level 1 BLAS, in which array parameters always refer to the lowest-addressed element in memory even when negative increments are used.

These functions assume that the default round-to-nearest rounding direction mode is in effect. On x86, these functions also assume that the default round-to-64-bit rounding precision mode is in effect. The result of calling a vector function with a non-default rounding mode in effect is undefined.

These functions handle special cases and exceptions in the spirit of IEEE 754. In particular,

- if x or *y* is ±Inf, rhypot(*x*, *y*) is +0, even if the other of *x* or *y* is NaN,
- if x or *y* is NaN and neither is infinite, rhypot(*x*, *y*) is NaN
- if *x* and *y* are both zero, rhypot(*x*, *y*) is +0, and a division-by-zero exception is raised.

An application wanting to check for exceptions should call feclearexcept(FE_ALL_EXCEPT) before calling these functions. On return, if fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an exception has been raised. The application can

then examine the result or argument vectors for exceptional values. Some vector functions can raise the inexact exception even if all elements of the argument array are such that the numerical results are exact.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also hypot(3M), feclearexcept(3M), fetestexcept(3M), attributes(5)

Name vrsqrt_, vrsqrtf_ – vector reciprocal square root functions

Synopsis

```
cc [ flag... ] file... -lmvec [ library... ]

void vrsqrt_(int *n, double * restrict x, int *stridex,
     double * restrict y, int *stridey);

void vrsqrtf_(int *n, float * restrict x, int *stridex,
     float * restrict y, int *stridey);
```

Description These functions evaluate the function `rsqrt`(*x*), defined by `rsqrt`(*x*) = 1 / `sqrt`(*x*), for an entire vector of values at once. The first parameter specifies the number of values to compute. Subsequent parameters specify the argument and result vectors. Each vector is described by a pointer to the first element and a stride, which is the increment between successive elements.

Specifically, `vrsqrt_`(*n*, *x*, *sx*, *y*, *sy*) computes *y*[*i* * *sy*] = `rsqrt`(*x*[*i* * *sx*]) for each *i* = 0, 1, ..., **n* - 1. The `vrsqrtf_()` function performs the same computation for single precision data.

These functions are not guaranteed to deliver results that are identical to the results of evaluating 1.0 / `sqrt`(*x*) given the same arguments. Non-exceptional results, however, are accurate to within a unit in the last place.

Usage The element count **n* must be greater than zero. The strides for the argument and result arrays can be arbitrary integers, but the arrays themselves must not be the same or overlap. A zero stride effectively collapses an entire vector into a single element. A negative stride causes a vector to be accessed in descending memory order, but note that the corresponding pointer must still point to the first element of the vector to be used; if the stride is negative, this will be the highest-addressed element in memory. This convention differs from the Level 1 BLAS, in which array parameters always refer to the lowest-addressed element in memory even when negative increments are used.

These functions assume that the default round-to-nearest rounding direction mode is in effect. On x86, these functions also assume that the default round-to-64-bit rounding precision mode is in effect. The result of calling a vector function with a non-default rounding mode in effect is undefined.

These functions handle special cases and exceptions in the spirit of IEEE 754. In particular,

- if *x* < 0, `rsqrt`(*x*) is NaN, and an invalid operation exception is raised,
- `rsqrt`(NaN) is NaN,
- `rsqrt`(+Inf) is +0,
- `rsqrt`(±0) is ±Inf, and a division-by-zero exception is raised.

An application wanting to check for exceptions should call `feclearexcept`(`FE_ALL_EXCEPT`) before calling these functions. On return, if `fetestexcept`(`FE_INVALID` | `FE_DIVBYZERO` | `FE_OVERFLOW` | `FE_UNDERFLOW`) is non-zero, an exception has been raised. The application can then examine the result or argument vectors for exceptional values. Some vector functions can raise the inexact exception even if all elements of the argument array are such that the

numerical results are exact.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also sqrt(3M), feclearexcept(3M), fetestexcept(3M), attributes(5)

Name vsin_, vsinf_ – vector sine functions

Synopsis

```
cc [ flag... ] file... -lmvec [ library... ]

void vsin_(int *n, double * restrict x, int *stridex,
     double * restrict y, int *stridey);

void vsinf_(int *n, float * restrict x, int *stridex,
     float * restrict y, int *stridey);
```

Description These functions evaluate the function `sin`(*x*) for an entire vector of values at once. The first parameter specifies the number of values to compute. Subsequent parameters specify the argument and result vectors. Each vector is described by a pointer to the first element and a stride, which is the increment between successive elements.

Specifically, `vsin_`(*n*, *x*, *sx*, *y*, *sy*) computes *y*[*i* * *sy*] = `sin`(*x*[*i* * *sx*]) for each *i* = 0, 1, ..., **n* - 1. The `vsinf_()` function performs the same computation for single precision data.

These functions are not guaranteed to deliver results that are identical to the results of the sin(3M) functions given the same arguments. Non-exceptional results, however, are accurate to within a unit in the last place.

Usage The element count **n* must be greater than zero. The strides for the argument and result arrays can be arbitrary integers, but the arrays themselves must not be the same or overlap. A zero stride effectively collapses an entire vector into a single element. A negative stride causes a vector to be accessed in descending memory order, but note that the corresponding pointer must still point to the first element of the vector to be used; if the stride is negative, this will be the highest-addressed element in memory. This convention differs from the Level 1 BLAS, in which array parameters always refer to the lowest-addressed element in memory even when negative increments are used.

These functions assume that the default round-to-nearest rounding direction mode is in effect. On x86, these functions also assume that the default round-to-64-bit rounding precision mode is in effect. The result of calling a vector function with a non-default rounding mode in effect is undefined.

These functions handle special cases and exceptions in the same way as the `sin()` functions when `c99` `MATHERREXCEPT` conventions are in effect. See sin(3M) for the results for special cases.

An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. The application can then examine the result or argument vectors for exceptional values. Some vector functions can raise the inexact exception even if all elements of the argument array are such that the numerical results are exact.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also sin(3M), feclearexcept(3M), fetestexcept(3M), attributes(5)

Name vsincos_, vsincosf_ – vector sincos functions

Synopsis

```
cc [ flag... ] file... -lmvec [ library... ]

void vsincos_(int *n, double * restrict x, int *stridex,
     double * restrict s, int *strides, double * restrict c,
     int *stridec);

void vsincosf_(int *n, float * restrict x, int *stridex,
     float * restrict s, int *strides, float * restrict c,
     int *stridec);
```

Description These functions evaluate both `sin`(*x*) and `cos`(*x*) for an entire vector of values at once. The first parameter specifies the number of values to compute. Subsequent parameters specify the argument and result vectors. Each vector is described by a pointer to the first element and a stride, which is the increment between successive elements.

Specifically, `vsincos_`(*n*, *x*, *sx*, *s*, *ss*, *c*, *sc*) simultaneously computes *s*[*i* * *ss*] = `sin`(*x*[*i* * *sx*]) and *c*[*i* * *sc*] = `cos`(*x*[*i* * *sx*]) for each *i* = 0, 1, ..., **n* - 1. The `vsincosf_()` function performs the same computation for single precision data.

These functions are not guaranteed to deliver results that are identical to the results of the `sincos(3M)` functions given the same arguments. Non-exceptional results, however, are accurate to within a unit in the last place.

Usage The element count **n* must be greater than zero. The strides for the argument and result arrays can be arbitrary integers, but the arrays themselves must not be the same or overlap. A zero stride effectively collapses an entire vector into a single element. A negative stride causes a vector to be accessed in descending memory order, but note that the corresponding pointer must still point to the first element of the vector to be used; if the stride is negative, this will be the highest-addressed element in memory. This convention differs from the Level 1 BLAS, in which array parameters always refer to the lowest-addressed element in memory even when negative increments are used.

These functions assume that the default round-to-nearest rounding direction mode is in effect. On x86, these functions also assume that the default round-to-64-bit rounding precision mode is in effect. The result of calling a vector function with a non-default rounding mode in effect is undefined.

These functions handle special cases and exceptions in the same way as the `sin()` and `cos()` functions when `c99` `MATHERREXCEPT` conventions are in effect. See `sin(3M)` and `cos(3M)` for the results for special cases.

An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. The application can then examine the result or argument vectors for exceptional values. Some vector functions can raise the inexact exception even if all elements of the argument array are such that the

numerical results are exact.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also cos(3M), sin(3M), sincos(3M), feclearexcept(3M), fetestexcept(3M), attributes(5)

Name vsincospi_, vsincospif_ – vector sincospi functions

Synopsis

```
cc [ flag... ] file... -lmvec [ library... ]

void vsincospi_(int *n, double * restrict x, int *stridex,
     double * restrict s, int *strides, double * restrict c,
     int *stridec);

void vsincospif_(int *n, float * restrict x, int *stridex,
     float * restrict s, int *strides, float * restrict c,
     int *stridec);
```

Description These functions evaluate both `sinpi`(*x*) and `cospi`(*x*), defined by `sinpi`(*x*) = `sin`(π * *x*) and `cospi`(*x*) = `cos`(π * *x*), for an entire vector of values at once. The first parameter specifies the number of values to compute. Subsequent parameters specify the argument and result vectors. Each vector is described by a pointer to the first element and a stride, which is the increment between successive elements.

Specifically, `vsincospi_`(*n*, *x*, *sx*, *s*, *ss*, *c*, *sc*) simultaneously computes *s*[*i* * **ss*] = `sinpi`(*x*[*i* * **sx*]) and *c*[*i* * **sc*] = `cospi`(*x*[*i* * **sx*]) for each *i* = 0, 1, ..., **n* - 1. The `vsincosf_()` function performs the same computation for single precision data.

Non-exceptional results are accurate to within a unit in the last place.

Usage The element count **n* must be greater than zero. The strides for the argument and result arrays can be arbitrary integers, but the arrays themselves must not be the same or overlap. A zero stride effectively collapses an entire vector into a single element. A negative stride causes a vector to be accessed in descending memory order, but note that the corresponding pointer must still point to the first element of the vector to be used; if the stride is negative, this will be the highest-addressed element in memory. This convention differs from the Level 1 BLAS, in which array parameters always refer to the lowest-addressed element in memory even when negative increments are used.

These functions assume that the default round-to-nearest rounding direction mode is in effect. On x86, these functions also assume that the default round-to-64-bit rounding precision mode is in effect. The result of calling a vector function with a non-default rounding mode in effect is undefined.

These functions handle special cases and exceptions in the spirit of IEEE 754. In particular,

- `sinpi`(NaN), `cospi`(NaN) are NaN,
- `sinpi`(±0) is ±0,
- `sinpi`(±Inf), `cospi`(±Inf) are NaN, and an invalid operation exception is raised.

An application wanting to check for exceptions should call `feclearexcept`(`FE_ALL_EXCEPT`) before calling these functions. On return, if `fetestexcept`(`FE_INVALID` | `FE_DIVBYZERO` | `FE_OVERFLOW` | `FE_UNDERFLOW`) is non-zero, an exception has been raised. The application can then examine the result or argument vectors for exceptional values. Some vector functions can

raise the inexact exception even if all elements of the argument array are such that the numerical results are exact.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also feclearexcept(3M), fetestexcept(3M), attributes(5)

Name vsinpi_, vsinpif_ – vector sinpi functions

Synopsis

```
cc [ flag... ] file... -lmvec [ library... ]

void vsinpi_(int *n, double * restrict x, int *stridex,
     double * restrict y, int *stridey);

void vsinpif_(int *n, float * restrict x, int *stridex,
     float * restrict y, int *stridey);
```

Description These functions evaluate the function `sinpi`(*x*), defined by `sinpi`(*x*) = `sin`(π * *x*), for an entire vector of values at once. The first parameter specifies the number of values to compute. Subsequent parameters specify the argument and result vectors. Each vector is described by a pointer to the first element and a stride, which is the increment between successive elements.

Specifically, `vsinpi_`(*n*, *x*, *sx*, *y*, *sy*) computes *y*[*i* * *sy*] = `sinpi`(*x*[*i* * *sx*]) for each *i* = 0, 1, ..., **n* - 1. The `vsinpif_()` function performs the same computation for single precision data.

Non-exceptional results are accurate to within a unit in the last place.

Usage The element count **n* must be greater than zero. The strides for the argument and result arrays can be arbitrary integers, but the arrays themselves must not be the same or overlap. A zero stride effectively collapses an entire vector into a single element. A negative stride causes a vector to be accessed in descending memory order, but note that the corresponding pointer must still point to the first element of the vector to be used; if the stride is negative, this will be the highest-addressed element in memory. This convention differs from the Level 1 BLAS, in which array parameters always refer to the lowest-addressed element in memory even when negative increments are used.

These functions assume that the default round-to-nearest rounding direction mode is in effect. On x86, these functions also assume that the default round-to-64-bit rounding precision mode is in effect. The result of calling a vector function with a non-default rounding mode in effect is undefined.

These functions handle special cases and exceptions in the spirit of IEEE 754. In particular,

- `sinpi`(NaN) is NaN,
- `sinpi`(±0) is ±0,
- `sinpi`(±Inf) is NaN, and an invalid operation exception is raised.

An application wanting to check for exceptions should call `feclearexcept`(`FE_ALL_EXCEPT`) before calling these functions. On return, if `fetestexcept`(`FE_INVALID` | `FE_DIVBYZERO` | `FE_OVERFLOW` | `FE_UNDERFLOW`) is non-zero, an exception has been raised. The application can then examine the result or argument vectors for exceptional values. Some vector functions can raise the inexact exception even if all elements of the argument array are such that the numerical results are exact.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also feclearexcept(3M), fetestexcept(3M), attributes(5)

Name vsqrt_, vsqrtf_ – vector square root functions

Synopsis

```
cc [ flag... ] file... -lmvec [ library... ]

void vsqrt_(int *n, double * restrict x, int *stridex,
     double * restrict y, int *stridey);

void vsqrtf_(int *n, float * restrict x, int *stridex,
     float * restrict y, int *stridey);
```

Description These functions evaluate the function `sqrt`(*x*) for an entire vector of values at once. The first parameter specifies the number of values to compute. Subsequent parameters specify the argument and result vectors. Each vector is described by a pointer to the first element and a stride, which is the increment between successive elements.

Specifically, `vsqrt_`(*n*, *x*, *sx*, *y*, *sy*) computes *y*[*i* * *sy*] = `sqrt`(*x*[*i* * *sx*]) for each *i* = 0, 1, ..., **n* - 1. The `vsqrtf_()` function performs the same computation for single precision data.

Unlike their scalar counterparts, these functions do not always deliver correctly rounded results. However, the error in each non-exceptional result is less than one unit in the last place.

Usage The element count **n* must be greater than zero. The strides for the argument and result arrays can be arbitrary integers, but the arrays themselves must not be the same or overlap. A zero stride effectively collapses an entire vector into a single element. A negative stride causes a vector to be accessed in descending memory order, but note that the corresponding pointer must still point to the first element of the vector to be used; if the stride is negative, this will be the highest-addressed element in memory. This convention differs from the Level 1 BLAS, in which array parameters always refer to the lowest-addressed element in memory even when negative increments are used.

These functions assume that the default round-to-nearest rounding direction mode is in effect. On x86, these functions also assume that the default round-to-64-bit rounding precision mode is in effect. The result of calling a vector function with a non-default rounding mode in effect is undefined.

These functions handle special cases and exceptions in the same way as the `sqrt()` functions when `c99 MATHERREXCEPT` conventions are in effect. See `sqrt(3M)` for the results for special cases.

An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. The application can then examine the result or argument vectors for exceptional values. Some vector functions can raise the inexact exception even if all elements of the argument array are such that the numerical results are exact.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also sqrt(3M), feclearexcept(3M), fetestexcept(3M), attributes(5)

Name vz_abs_, vc_abs_ – vector complex absolute value functions

Synopsis

```
cc [ flag... ] file... -lmvec [ library... ]

void vz_abs_(int *n, double complex * restrict z,
     int *stridez, double * restrict y, int *stridey);

void vc_abs_(int *n, float complex * restrict z,
     int *stridez, float * restrict y, int *stridey);
```

Description These functions compute the magnitude (or modulus) | *z* | for an entire vector of values at once. The first parameter specifies the number of values to compute. Subsequent parameters specify the argument and result vectors. Each vector is described by a pointer to the first element and a stride, which is the increment between successive elements.

Specifically, `vz_abs_`(*n*, *z*, *sz*, *y*, *sy*) computes *y*[*i* * *sy*] = | *z*[*i* * *sz*] | for each *i* = 0, 1, ..., **n* - 1. The `vc_abs_()` function performs the same computation for single precision data.

These functions are not guaranteed to deliver results that are identical to the results of the `cabs`(3M) functions given the same arguments. Non-exceptional results, however, are accurate to within a unit in the last place.

Usage The element count **n* must be greater than zero. The strides for the argument and result arrays can be arbitrary integers, but the arrays themselves must not be the same or overlap. A zero stride effectively collapses an entire vector into a single element. A negative stride causes a vector to be accessed in descending memory order, but note that the corresponding pointer must still point to the first element of the vector to be used; if the stride is negative, this will be the highest-addressed element in memory. This convention differs from the Level 1 BLAS, in which array parameters always refer to the lowest-addressed element in memory even when negative increments are used.

These functions assume that the default round-to-nearest rounding direction mode is in effect. On x86, these functions also assume that the default round-to-64-bit rounding precision mode is in effect. The result of calling a vector function with a non-default rounding mode in effect is undefined.

These functions handle special cases and exceptions in the spirit of IEEE 754. See `cabs`(3M) for the results for special cases.

An application wanting to check for exceptions should call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an exception has been raised. The application can then examine the result or argument vectors for exceptional values. Some vector functions can raise the inexact exception even if all elements of the argument array are such that the numerical results are exact.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also cabs(3M), feclearexcept(3M), fetestexcept(3M), attributes(5)

Name vz_exp_, vc_exp_ – vector complex exponential functions

Synopsis

```
cc [ flag... ] file... -lmvec [ library... ]

void vz_exp_(int *n, double complex * restrict z,
     int *stridez, double  complex * restrict w int *stridew,
     double * tmp);

void vc_exp_(int *n, float complex * restrict z,
     int *stridez, float complex * restrict w, int *stridew,
     float * tmp);
```

Description These functions evaluate the complex function exp(*z*) for an entire vector of values at once. The first parameter specifies the number of values to compute. Subsequent parameters specify the argument and result vectors. Each vector is described by a pointer to the first element and a stride, which is the increment between successive elements. The last argument is a pointer to scratch storage; this storage must be large enough to hold **n* consecutive values of the real type corresponding to the complex type of the argument and result.

Specifically, vz_exp_(*n*, *z*, *sz*, *w*, *sw*, *tmp*) computes *w*[*i* * **sw*] = exp(*z*[*i* * **sz*]) for each *i* = 0, 1, ..., **n* - 1. The vc_exp_() function performs the same computation for single precision data.

These functions are not guaranteed to deliver results that are identical to the results of the cexp(3M) functions given the same arguments.

Usage The element count **n* must be greater than zero. The strides for the argument and result arrays can be arbitrary integers, but the arrays themselves must not be the same or overlap. A zero stride effectively collapses an entire vector into a single element. A negative stride causes a vector to be accessed in descending memory order, but note that the corresponding pointer must still point to the first element of the vector to be used; if the stride is negative, this will be the highest-addressed element in memory. This convention differs from the Level 1 BLAS, in which array parameters always refer to the lowest-addressed element in memory even when negative increments are used.

These functions assume that the default round-to-nearest rounding direction mode is in effect. On x86, these functions also assume that the default round-to-64-bit rounding precision mode is in effect. The result of calling a vector function with a non-default rounding mode in effect is undefined.

Unlike the c99 cexp(3M) functions, the vector complex exponential functions make no attempt to handle special cases and exceptions; they simply use textbook formulas to compute a complex exponential in terms of real elementary functions. As a result, these functions can raise different exceptions and/or deliver different results from cexp().

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also cexp(3M), attributes(5)

Name vz_log_, vc_log_ – vector complex logarithm functions

Synopsis

```
cc [ flag... ] file... -lmvec [ library... ]

void vz_log_(int *n, double complex * restrict z,
     int *stridez, double _complex * restrict w, int *stridew);

void vc_log_(int *n, float complex * restrict z,
     int *stridez, float complex * restrict w, int *stridew);
```

Description These functions evaluate the complex function log(*z*) for an entire vector of values at once. The first parameter specifies the number of values to compute. Subsequent parameters specify the argument and result vectors. Each vector is described by a pointer to the first element and a stride, which is the increment between successive elements.

Specifically, vz_log_(*n*, *z*, *sz*, *w*, *sw*) computes *w*[*i* * *sw*] = log(*z*[*i* * *sz*]) for each *i* = 0, 1, ..., **n* - 1. The vc_log_() function performs the same computation for single precision data.

These functions are not guaranteed to deliver results that are identical to the results of the clog(3M) functions given the same arguments.

Usage The element count **n* must be greater than zero. The strides for the argument and result arrays can be arbitrary integers, but the arrays themselves must not be the same or overlap. A zero stride effectively collapses an entire vector into a single element. A negative stride causes a vector to be accessed in descending memory order, but note that the corresponding pointer must still point to the first element of the vector to be used; if the stride is negative, this will be the highest-addressed element in memory. This convention differs from the Level 1 BLAS, in which array parameters always refer to the lowest-addressed element in memory even when negative increments are used.

These functions assume that the default round-to-nearest rounding direction mode is in effect. On x86, these functions also assume that the default round-to-64-bit rounding precision mode is in effect. The result of calling a vector function with a non-default rounding mode in effect is undefined.

Unlike the c99 clog(3M) functions, the vector complex exponential functions make no attempt to handle special cases and exceptions; they simply use textbook formulas to compute a complex exponential in terms of real elementary functions. As a result, these functions can raise different exceptions and/or deliver different results from clog().

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also clog(3M), attributes(5)

Name vz_pow_, vc_pow_ – vector complex power functions

Synopsis

```
cc [ flag... ] file... -lmvec [ library... ]

void vz_pow_(int *n, double complex * restrict z,
     int *stridez, double complex * restrict w, int *stridew,
     double complex * restrict u, int *strideu,
     double * tmp);

void vc_pow_(int *n, float complex * restrict z,
     int *stridez, float complex * restrict w, int *stridew,
     float complex * restrict u, int *strideu,
     float * tmp);
```

Description These functions evaluate the complex function *z*^*w* for an entire vector of values at once. The first parameter specifies the number of values to compute. Subsequent parameters specify the argument and result vectors. Each vector is described by a pointer to the first element and a stride, which is the increment between successive elements. The last argument is a pointer to scratch storage; this storage must be large enough to hold 3 * **n* consecutive values of the real type corresponding to the complex type of the argument and result.

Specifically, `vz_pow_`(*n*, *z*, *sz*, *w*, *sw*, *u*, *su*, *tmp*) computes *u*[*i* * **su*] = (*z*[*i* * **sz*])^(*w*[*i* * **sw*]) for each *i* = 0, 1, ..., **n* - 1. The `vc_pow_()` function performs the same computation for single precision data.

These functions are not guaranteed to deliver results that are identical to the results of the cpow(3M) functions given the same arguments.

Usage The element count **n* must be greater than zero. The strides for the argument and result arrays can be arbitrary integers, but the arrays themselves must not be the same or overlap. A zero stride effectively collapses an entire vector into a single element. A negative stride causes a vector to be accessed in descending memory order, but note that the corresponding pointer must still point to the first element of the vector to be used; if the stride is negative, this will be the highest-addressed element in memory. This convention differs from the Level 1 BLAS, in which array parameters always refer to the lowest-addressed element in memory even when negative increments are used.

These functions assume that the default round-to-nearest rounding direction mode is in effect. On x86, these functions also assume that the default round-to-64-bit rounding precision mode is in effect. The result of calling a vector function with a non-default rounding mode in effect is undefined.

Unlike the c99 cpow(3M) functions, the vector complex exponential functions make no attempt to handle special cases and exceptions; they simply use textbook formulas to compute a complex exponential in terms of real elementary functions. As a result, these functions can raise different exceptions and/or deliver different results from `cpow()`.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also cpow(3M), attributes(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);
         wsreg_free_component(comp);
```

EXAMPLE 1 Create and register a component. *(Continued)*

```
        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 *Oracle 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 *Oracle 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 *Oracle 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 *Oracle 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 *Oracle 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 *Oracle 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 *Oracle 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 *Oracle 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 *Oracle 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 *Oracle 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 *Oracle 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 *Oracle 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 *Oracle 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 *Oracle 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 y0, y0f, y0l, y1, y1f, y1l, yn, ynf, ynl – Bessel functions of the second kind

Synopsis

```
c99 [ flag... ] file... -lm [ library... ]
#include <math.h>

double y0(double x);

float y0f(float x);

long double y0l(long double x);

double y1(double x);

float y1f(float x);

long double y1l(long double x);

double yn(int n, double x);

float ynf(int n, float x);

long double ynl(int n, long double x);
```

Description These functions compute Bessel functions of *x* of the second kind of orders 0, 1 and *n*, respectively.

Return Values Upon successful completion, these functions return the relevant Bessel value of *x* of the second kind.

If *x* is NaN, a NaN is returned.

If *x* is negative, –HUGE_VAL or NaN is returned.

If *x* is 0.0, –HUGE_VAL is returned.

If the correct result would cause overflow, –HUGE_VAL is returned.

For exceptional cases, matherr(3M) tabulates the values to be returned as specified by SVID3 and XPG3.

Errors No errors are returned.

Usage An application wanting to check for exceptions should call feclearexcept(FE_ALL_EXCEPT) before calling these functions. On return, if fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an exception has been raised. An application should either examine the return value or check the floating point exception flags to detect exceptions.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	See below.
MT-Level	MT-Safe

The `y0()`, `y1()`, and `yn()` functions are Standard. The `y0f()`, `y0l()`, `y1f()`, `y1l()`, `ynf()`, and `ynl()` functions are Stable.

See Also `isnan`(3M), `feclearexcept`(3M), `fetestexcept`(3M), `j0`(3M), `math.h`(3HEAD), `matherr`(3M), `attributes`(5), `standards`(5)