man pages section 3: Extended Library Functions, Volume 3

Beta

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

Part No: 819–2247–30
December 2009

091223@23031

Contents

Preface 7

Extended Library Functions, Volume 3 11
Exacct(3PERL) 12
Exacct::Catalog(3PERL) 15
Exacct::File(3PERL) 17
Exacct::Object(3PERL) 20
Exacct::Object::Group(3PERL) 23
Exacct::Object::Item(3PERL) 25
getprojent(3PROJECT) 27
Kstat(3PERL) 31
Lgrp(3PERL) 34
libpicl(3PICL) 45
libpicltree(3PICLTREE) 48
nvlist_add_boolean(3NVPAIR) 51
nvlist_alloc(3NVPAIR) 55
nvlist_lookup_boolean(3NVPAIR) 62
nvlist_lookup_nvpair(3NVPAIR) 66
nvlist_next_nvpair(3NVPAIR) 67
nvlist_remove(3NVPAIR) 70
nvpair_value_byte(3NVPAIR) 71
pam(3PAM) 74
pam_acct_mgmt(3PAM) 77
pam_authenticate(3PAM) 78
pam_chauthtok(3PAM) 80
pam_getenv(3PAM) 82
pam_getenvlist(3PAM) 83
pam_get_user(3PAM) 84

pam_open_session(3PAM) 86
pam_putenv(3PAM) 88
pam_setcred(3PAM) 90
pam_set_data(3PAM) 92
pam_set_item(3PAM) 94
pam_sm(3PAM) 97
pam_sm_acct_mgmt(3PAM) 100
pam_sm_authenticate(3PAM) 102
pam_sm_chauthtok(3PAM) 104
pam_sm_open_session(3PAM) 107
pam_sm_setcred(3PAM) 109
pam_start(3PAM) 111
pam_strerror(3PAM) 114
papiAttributeListAddValue(3PAPI) 115
papiJobSubmit(3PAPI) 121
papiLibrarySupportedCall(3PAPI) 133
papiPrintersList(3PAPI) 134
papiServiceCreate(3PAPI) 142
papiStatusString(3PAPI) 146
picld_log(3PICLTREE) 147
picld_plugin_register(3PICLTREE) 148
picl_find_node(3PICL) 150
picl_get_first_prop(3PICL) 151
picl_get_frutree_parent(3PICL) 153
picl_get_next_by_row(3PICL) 154
picl_get_node_by_path(3PICL) 156
picl_get_prop_by_name(3PICL) 158
picl_get_propinfo(3PICL) 160
picl_get_propinfo_by_name(3PICL) 161
picl_get_propval(3PICL) 163
picl_get_root(3PICL) 165
picl_initialize(3PICL) 166
picl_set_propval(3PICL) 167
picl_shutdown(3PICL) 169
picl_strerror(3PICL) 170
picl_wait(3PICL) 171

picl_walk_tree_by_class(3PICL) 172
pool_associate(3POOL) 174
pool_component_info(3POOL) 177
pool_component_to_elem(3POOL) 179
pool_conf_alloc(3POOL) 180
pool_dynamic_location(3POOL) 186
pool_error(3POOL) 189
pool_get_binding(3POOL) 191
pool_get_pool(3POOL) 194
pool_get_property(3POOL) 197
pool_resource_create(3POOL) 200
pool_value_alloc(3POOL) 204
pool_walk_components(3POOL) 207
Privilege(3PERL) 209
proc_service(3PROC) 211
Project(3PERL) 214
project_walk(3PROJECT) 217
ps_lgetregs(3PROC) 219
ps_pglobal_lookup(3PROC) 221
ps_pread(3PROC) 222
ps_pstop(3PROC) 223
ptree_add_node(3PICLTREE) 225
ptree_add_prop(3PICLTREE) 226
ptree_create_and_add_node(3PICLTREE) 227
ptree_create_and_add_prop(3PICLTREE) 228
ptree_create_node(3PICLTREE) 230
ptree_create_prop(3PICLTREE) 232
ptree_create_table(3PICLTREE) 234
ptree_find_node(3PICLTREE) 235
ptree_get_first_prop(3PICLTREE) 236
ptree_get_frutree_parent(3PICLTREE) 237
ptree_get_next_by_row(3PICLTREE) 238
ptree_get_node_by_path(3PICLTREE) 239
ptree_get_prop_by_name(3PICLTREE) 241
ptree_get_propinfo(3PICLTREE) 242
ptree_get_propinfo_by_name(3PICLTREE) 243

ptree_get_propval(3PICLTREE) 244
ptree_get_root(3PICLTREE) 246
ptree_init_propinfo(3PICLTREE) 247
ptree_post_event(3PICLTREE) 248
ptree_register_handler(3PICLTREE) 249
ptree_unregister_handler(3PICLTREE) 251
ptree_update_propval(3PICLTREE) 252
ptree_walk_tree_by_class(3PICLTREE) 254
reparse_add(3REPARSE) 255
rsm_create_localmemory_handle(3RSM) 258
rsm_get_controller(3RSM) 260
rsm_get_interconnect_topology(3RSM) 262
rsm_get_segmentid_range(3RSM) 264
rsm_intr_signal_post(3RSM) 266
rsm_intr_signal_wait_pollfd(3RSM) 268
rsm_memseg_export_create(3RSM) 270
rsm_memseg_export_publish(3RSM) 273
rsm_memseg_get_pollfd(3RSM) 276
rsm_memseg_import_connect(3RSM) 277
rsm_memseg_import_get(3RSM) 279
rsm_memseg_import_init_barrier(3RSM) 281
rsm_memseg_import_map(3RSM) 282
rsm_memseg_import_open_barrier(3RSM) 284
rsm_memseg_import_put(3RSM) 286
rsm_memseg_import_putv(3RSM) 288
rsm_memseg_import_set_mode(3RSM) 290
setproject(3PROJECT) 291
Task(3PERL) 294
Ucred(3PERL) 295

Preface

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

Overview

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

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

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

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

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

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

The following special characters are used in this section:

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

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

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

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

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

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

REFERENCE

Extended Library Functions, Volume 3

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
```

EXAMPLE 1 Output of the `dump()` method for a Perl exacct Group object. *(Continued)*

```
    Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_TASKID
    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 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 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
        wait idle)};
```

EXAMPLE 1 Sun::Solaris::Kstat example *(Continued)*

```
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 Lgrp – Perl interface to Solaris liblgrp library

Synopsis

```
use Sun::Solaris::Lgrp qw(:ALL);

# initialize lgroup interface
my $cookie = lgrp_init(LGRP_VIEW_OS | LGRP_VIEW_CALLER);
my $l = Sun::Solaris::Lgrp->new(LGRP_VIEW_OS |
     LGRP_VIEW_CALLER);

my $version = lgrp_version(LGRP_VER_CURRENT | LGRP_VER_NONE);
$version = $l->version(LGRP_VER_CURRENT | LGRP_VER_NONE);

$home = lgrp_home(P_PID, P_MYID);
$home = l->home(P_PID, P_MYID);

lgrp_affinity_set(P_PID, $pid, $lgrp,
      LGRP_AFF_STRONG | LGRP_AFF_WEAK | LGRP_AFF_NONE);
$l->affinity_set(P_PID, $pid, $lgrp,
      LGRP_AFF_STRONG | LGRP_AFF_WEAK | LGRP_AFF_NONE);

my $affinity = lgrp_affinity_get(P_PID, $pid, $lgrp);
$affinity = $l->affinity_get(P_PID, $pid, $lgrp);

my $nlgrps = lgrp_nlgrps($cookie);
$nlgrps = $l->nlgrps();

my $root = lgrp_root($cookie);
$root = l->root();

$latency = lgrp_latency($lgrp1, $lgrp2);
$latency = $l->latency($lgrp1, $lgrp2);

my @children = lgrp_children($cookie, $lgrp);
@children = l->children($lgrp);

my @parents = lgrp_parents($cookie, $lgrp);
@parents = l->parents($lgrp);

my @lgrps = lgrp_lgrps($cookie);
@lgrps = l->lgrps();

@lgrps = lgrp_lgrps($cookie, $lgrp);
@lgrps = l->lgrps($lgrp);

my @leaves = lgrp_leaves($cookie);
@leaves = l->leaves();

my $is_leaf = lgrp_isleaf($cookie, $lgrp);
```

```
$is_leaf = $l->is_leaf($lgrp);

my @cpus = lgrp_cpus($cookie, $lgrp,
     LGRP_CONTENT_HIERARCHY | LGRP_CONTENT_DIRECT);
@cpus = l->cpus($lgrp, LGRP_CONTENT_HIERARCHY |
     LGRP_CONTENT_DIRECT);

my $memsize = lgrp_mem_size($cookie, $lgrp,
      LGRP_MEM_SZ_INSTALLED | LGRP_MEM_SZ_FREE,
      LGRP_CONTENT_HIERARCHY | LGRP_CONTENT_DIRECT);
$memsize = l->mem_size($lgrp,
      LGRP_MEM_SZ_INSTALLED | LGRP_MEM_SZ_FREE,
      LGRP_CONTENT_HIERARCHY | LGRP_CONTENT_DIRECT);

my $is_stale = lgrp_cookie_stale($cookie);
$stale = l->stale();

lgrp_fini($cookie);

# The following is available for API version greater than 1:
my @lgrps = lgrp_resources($cookie, $lgrp, LGRP_RSRC_CPU);

# Get latencies from cookie
$latency = lgrp_latency_cookie($cookie, $from, $to);
```

Description This module provides access to the liblgrp(3LIB) library and to various constants and functions defined in <sys/lgrp_sys.h>. It provides both the procedural and object interface to the library. The procedural interface requires (in most cases) passing around a transparent cookie. The object interface hides all the cookie manipulations from the user.

Functions returning a scalar value indicate an error by returning `undef`. The caller can examine the `$!` variable to get the error value.

Functions returning a list value return the number of elements in the list when called in scalar context. In the event of error, the empty list is returned in the array context and `undef` is returned in the scalar context.

Constants The constants are exported with `:CONSTANTS` or `:ALL` tags:

```
use Sun::Solaris::Lgrp ':ALL';
```

or

```
use Sun::Solaris::Lgrp ':CONSTANTS';
```

The following constants are available for use in Perl programs:

`LGRP_NONE`

LGRP_VER_CURRENT
LGRP_VER_NONE
LGRP_VIEW_CALLER
LGRP_VIEW_OS
LGRP_AFF_NONE
LGRP_AFF_STRONG
LGRP_AFF_WEAK
LGRP_CONTENT_DIRECT
LGRP_CONTENT_HIERARCHY
LGRP_MEM_SZ_FREE
LGRP_MEM_SZ_FREE
LGRP_RSRC_CPU (1)
LGRP_RSRC_MEM (1)
LGRP_CONTENT_ALL (1)
LGRP_LAT_CPU_TO_MEM (1)
P_PID
P_LWPID
P_MYID

(1) Available for versions of the liblgrp(3LIB) API greater than 1.

Functions A detailed description of each function follows. Since this module is intended to provide a Perl interface to the functions in liblgrp(3LIB), a very short description is given for the corresponding functions in this module and a reference is given to the complete description in the liblgrp manual pages. Any differences or additional functionality in the Perl module are highlighted and fully documented here.

lgrp_init([LGRP_VIEW_CALLER | LGRP_VIEW_OS])
This function initializes the lgroup interface and takes a snapshot of the lgroup hierarchy with the given view. 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() function should be called with the cookie when it is no longer needed. Unlike lgrp_init(3LGRP), LGRP_VIEW_OS is assumed as the default if no view is provided.

Upon successful completion, lgrp_init() returns a cookie. Otherwise it returns undef and sets $! to indicate the error.

See lgrp_init(3LGRP) for more information.

lgrp_fini($*cookie*)
This function takes a cookie, frees the snapshot of the lgroup hierarchy created by lgrp_init(), and cleans up anything else set up by lgrp_init(). After this function is called, the cookie returned by the lgroup interface might no longer be valid and should not be used.

Upon successful completion, 1 is returned. Otherwise, `undef` is returned and `$!` is set to indicate the error.

See `lgrp_fini(3LGRP)` for more information.

`lgrp_view`($*cookie*)
This function takes a cookie representing the snapshot of the lgroup hierarchy 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.

Upon succesful completion, the function returns the view for the snapshot of the lgroup hierarchy represented by the given cookie. Otherwise, `undef` is returned and `$!` is set to indicate the error.

See `lgrp_view(3LGRP)` for more information.

`lgrp_home`($*idtype*, $*id*)
This function returns the home lgroup for the given process or thread. 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.

Upon successful completion, `lgrp_home()` returns the ID of the home lgroup of the specified process or thread. Otherwise, `undef` is returned and `$!` is set to indicate the error.

See `lgrp_home(3LGRP)` for more information.

`lgrp_cookie_stale`($*cookie*)
Upon successful completion, this function returns whether the cookie is stale. Otherwise, it returns `undef` and sets `$!` to indicate the error.

The `lgrp_cookie_stale()` function will fail with `EINVAL` if the cookie is not valid.

See `lgrp_cookie_stale(3LGRP)` for more information.

`lgrp_cpus`($*cookie*, $*lgrp*, $*context*)
This function takes a cookie representing a snapshot of the lgroup hierarchy and returns the list of CPUs in the lgroup specified by $*lgrp*. The $*context* argument should be set to one of the following values to specify whether the direct contents or everything in this lgroup including its children should be returned:

`LGRP_CONTENT_HIERARCHY`	everything within this hierarchy
`LGRP_CONTENT_DIRECT`	directly contained in lgroup

When called in scalar context, `lgrp_cpus()` function returns the number of CPUs contained in the specified lgroup.

In the event of error, `undef` is returned in scalar context and `$!` is set to indicate the error. In list context, the empty list is returned and `$!` is set.

See `lgrp_cpus(3LGRP)` for more information.

`lgrp_children`($*cookie*, $*lgrp*)
This function takes a cookie representing a snapshot of the lgroup hierarchy and returns the list of lgroups that are children of the specified lgroup.

When called in scalar context, `lgrp_children()` returns the number of children lgroups for the specified lgroup.

In the event of error, `undef` or empty list is returned and `$!` is set to indicate the error.

See `lgrp_children(3LGRP)` for more information.

`lgrp_parents`($*cookie*, $*lgrp*)
This function takes a cookie representing a snapshot of the lgroup hierarchy and returns the list of parents of the specified lgroup.

When called in scalar context, `lgrp_parents()` returns the number of parent lgroups for the specified lgroup.

In the event of error, `undef` or an empty list is returned and `$!` is set to indicate the error.

See `lgrp_parents(3LGRP)` for more information.

`lgrp_nlgrps`($*cookie*)
This function takes a cookie representing a snapshot of the lgroup hierarchy. It returns the number of lgroups in the hierarchy, where the number is always at least one.

In the event of error, `undef` is returned and `$!` is set to `EINVAL`, indicating that the cookie is not valid.

See `lgrp_nlgrps(3LGRP)` for more information.

`lgrp_root`($*cookie*)
This function returns the root lgroup ID.

In the event of error, `undef` is returned and `$!` is set to `EINVAL`, indicatng that the cookie is not valid.

See `lgrp_root(3LGRP)` for more information.

`lgrp_mem_size`($*cookie*, $*lgrp*, $*type*, $*content*)
This function takes a cookie representing a snapshot of the lgroup hierarchy. The 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 including its children should be returned:

LGRP_CONTENT_HIERARCHY Return everything within this hierarchy.

LGRP_CONTENT_DIRECT Return that which is directly contained in this lgroup.

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.

Upon successful completion, the size in bytes is returned. Otherwise, `undef` is returned and `$!` is set to indicate the error.

See `lgrp_mem_size(3LGRP)` for more information.

`lgrp_version`([$*version*])
This 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` or `LGRP_VER_NONE` to find out the current lgroup interface version on the running system.

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.

If $*version* is `LGRP_VER_NONE`, `lgrp_version()` returns the current version of the library.

The following example tests whether the version of the interface used by the caller is supported:

```
lgrp_version(LGRP_VER_CURRENT) == LGRP_VER_CURRENT or
    die("Built with unsupported lgroup interface");
```

See `lgrp_version(3LGRP)` for more information.

`lgrp_affinity_set`($*idtype*, $*id*, $*lgrp*, $*affinity*)
This 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.

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

Upon successful completion, lgrp_affinity_set() returns 1. Otherwise, it returns undef and set $! to indicate the error.

See lgrp_affinity_set(3LGRP) for more information.

lgrp_affinity_get($*idtype*, $*id*, $*lgrp*)
This function returns the affinity that the LWP has to a given lgroup.

See lgrp_affinity_get(3LGRP) for more information.

lgrp_latency_cookie($*cookie*, $*from*, $*to*, [$*between*=LGRP_LAT_CPU_TO_MEM])
This function takes a cookie representing a snapshot of the lgroup hierarchy 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 optional $*between* argument should be set to LGRP_LAT_CPU_TO_MEM to specify between which hardware resources the latency should be measured. The only valid value is LGRP_LAT_CPU_TO_MEM, which represents latency from CPU to memory.

Upon successful completion, lgrp_latency_cookie() return 1. Otherwise, it returns undef and set $! to indicate the error. For LGRP API version 1, the lgrp_latency_cookie() is an alias for lgrp_latency.()

See lgrp_latency_cookie(3LGRP) for more information.

lgrp_latency($*from*, $*to*)
This 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, lgrp_latency_cookie() should be used in its place.

See lgrp_latency(3LGRP) for more information.

lgrp_resources($*cookie*, $*lgrp*, $*type*)
This function returns the list of lgroups directly containing resources of the specified type. The resources are represented by a set of lgroups in which each lgroup directly contains CPU and/or memory resources.

The *type* can be specified as:

LGRP_RSRC_CPU CPU resources

LGRP_RSRC_MEM memory resources

In the event of error, undef or an empty list is returned and $! is set to indicate the error.

This function is available only for API version 2 and returns `undef` or an empty list for API version 1 and sets `$!` to `EINVAL`.

See `lgrp_resources(3LGRP)` for more information.

`lgrp_lgrps`($*cookie*, [$*lgrp*])
This function returns a list of all lgroups in a hierarchy starting from $*lgrp*. If $*lgrp* is not specified, uses the value of `lgrp_root`($*cookie*). This function returns the empty list on failure.

When called in scalar context, this function returns the total number of lgroups in the system.

`lgrp_leaves`($*cookie*, [$*lgrp*])
This function returns a list of all leaf lgroups in a hierarchy starting from $*lgrp*. If $*lgrp* is not specified, this function uses the value of `lgrp_root`($*cookie*). It returns `undef` or an empty list on failure.

When called in scalar context, this function returns the total number of leaf lgroups in the system.

`lgrp_isleaf`($*cookie*, $*lgrp*)
This function returns True if $*lgrp* is a leaf (has no children). Otherwise it returns False.

Object methods

`new`([$*view*])	This method creates a new `Sun::Solaris::Lgrp` object. An optional argument is passed to the `lgrp_init()` function. By default this method uses `LGRP_VIEW_OS`.
`cookie()`	This method returns a transparent cookie that can be passed to functions accepting the cookie.
`version`([$*version*])	Without the argument, this method returns the current version of the `liblgrp(3LIB)` library. This method is a wrapper for `lgrp_version()` with `LGRP_VER_NONE` as the default version argument.
`stale()`	This method returns T if the lgroup information in the object is stale and F otherwise. It is a wrapper for `lgrp_cookie_stale()`.
`view()`	This method returns the snapshot's view of the lgroup hierarchy. It is a wrapper for `lgrp_view()`.

`root()`	This method returns the root lgroup. It is a wrapper for `lgrp_root()`.
`children`($*lgrp*)	This method returns the list of lgroups that are children of the specified lgroup. It is a wrapper for `lgrp_children()`.
`parents`($*lgrp*)	This method returns the list of lgroups that are parents of the specified lgroup. It is a wrapper for `lgrp_parents()`.
`nlgrps()`	This method returns the number of lgroups in the hierarchy. It is a wrapper for `lgrp_nlgrps()`.
`mem_size`($*lgrp*, $*type*, $*content*)	This method returns the memory size of the given lgroup in bytes. It is a wrapper for `lgrp_mem_size()`.
`cpus`($*lgrp*, $*context*)	This method returns the list of CPUs in the lgroup specified by $lgrp. It is a wrapper for `lgrp_cpus()`.
`resources`($*lgrp*, $*type*)	This method returns the list of lgroups directly containing resources of the specified type. It is a wrapper for `lgrp_resources()`.
`home`($*idtype*, $*id*)	This method returns the home lgroup for the given process or thread. It is a wrapper for `lgrp_home()`.
`affinity_get`($*idtype*, $*id*, $*lgrp*)	This method returns the affinity that the LWP has to a given lgrp. It is a wrapper for `lgrp_affinity_get()`.
`affinity_set`($*idtype*, $*id*, $*lgrp*, $*affinity*)	This method sets the affinity that the LWP or set of LWPs specified by $*idtype* and $*id* have for the given lgroup. It is a wrapper for lgrp_affinity_set.
`lgrps`([$*lgrp*])	This method returns list of all lgroups in a hierarchy starting from $*lgrp* or the `lgrp_root()` if $*lgrp* is not specified. It is a wrapper for `lgrp_lgrps()`.
`leaves`([$*lgrp*])	This method returns a list of all leaf lgroups in a hierarchy starting from $*lgrp*. If $*lgrp* is not

	specified, this method uses the value of `lgrp_root()`. It is a wrapper for `lgrp_leaves()`.
`isleaf`($*lgrp*)	This method returns True if $*lgrp* is leaf (has no children) and False otherwise. It is a wrapper for `lgrp_isleaf()`.
`latency`($*from*, $*to*)	This method returns the latency value between a hardware resource in the $*from* lgroup to a hardware resource in the $*to* lgroup. It uses `lgrp_latency()` for version 1 of `liblgrp` and `lgrp_latency_cookie()` for newer versions.

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:

`:LGRP_CONSTANTS`	`LGRP_AFF_NONE`, `LGRP_AFF_STRONG`, `LGRP_AFF_WEAK`, `LGRP_CONTENT_DIRECT`, `LGRP_CONTENT_HIERARCHY`, `LGRP_MEM_SZ_FREE`, `LGRP_MEM_SZ_INSTALLED`, `LGRP_VER_CURRENT`, `LGRP_VER_NONE`, `LGRP_VIEW_CALLER`, `LGRP_VIEW_OS`, `LGRP_NONE`, `LGRP_RSRC_CPU`, `LGRP_RSRC_MEM`, `LGRP_CONTENT_ALL`, `LGRP_LAT_CPU_TO_MEM`
`:PROC_CONSTANTS`	`P_PID`, `P_LWPID`, `P_MYID`
`:CONSTANTS`	`:LGRP_CONSTANTS`, `:PROC_CONSTANTS`
`:FUNCTIONS`	`lgrp_affinity_get()`, `lgrp_affinity_set()`, `lgrp_children()`, `lgrp_cookie_stale()`, `lgrp_cpus()`, `lgrp_fini()`, `lgrp_home()`, `lgrp_init()`, `lgrp_latency()`, `lgrp_latency_cookie()`, `lgrp_mem_size()`, `lgrp_nlgrps()`, `lgrp_parents()`, `lgrp_root()`, `lgrp_version()`, `lgrp_view()`, `lgrp_resources()`, `lgrp_lgrps()`, `lgrp_leaves()`, `lgrp_isleaf()`
`:ALL`	`:CONSTANTS`, `:FUNCTIONS`

Error values

The functions in this module return `undef` or an empty list when an underlying library function fails. The `$!` is set to provide more information values for the error. The following error codes are possible:

`EINVAL`	The value supplied is not valid.
`ENOMEM`	There was not enough system memory to complete an operation.
`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.

Difference in the API versions

The liblgrp(3LIB) library is versioned. The exact version that was used to compile a module is available through the lgrp_version() function.

Version 2 of the lgrp_user API introduced the following constants and functions not present in version 1:

LGRP_RSRC_CPU constant
LGRP_RSRC_MEM constant
LGRP_CONTENT_ALL constant
LGRP_LAT_CPU_TO_MEM constant
lgrp_resources() function
lgrp_latency_cookie() function

The LGRP_RSRC_CPU and LGRP_RSRC_MEM constants are not defined for version 1. The lgrp_resources() function is defined for version 1 but always returns an empty list. The lgrp_latency_cookie() function is an alias for lgrp_latency() for version 1.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWpl5u
Interface Stability	Unstable

See Also

lgrp_affinity_get(3LGRP), lgrp_affinity_set(3LGRP), lgrp_children(3LGRP), lgrp_cookie_stale(3LGRP), lgrp_cpus(3LGRP), lgrp_fini(3LGRP), lgrp_home(3LGRP), lgrp_init(3LGRP), lgrp_latency(3LGRP), lgrp_latency_cookie(3LGRP), lgrp_mem_size(3LGRP), lgrp_nlgrps(3LGRP), lgrp_parents(3LGRP), lgrp_resources(3LGRP), lgrp_root(3LGRP), lgrp_version(3LGRP), lgrp_view(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 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_double, 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_double(nvlist_t *nvl, const char *name,
     double 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 list that is added to the parent `nvlist_t` by calling `nvlist_add_nvlist()` is copied and thus is not freed when `nvlist_free()` is called on the parent list. To prevent memory leaks, your code needs to look like the following (error handling elided for clarity):

```
nvlist_t *parent_nvl;
nvlist_t *child_nvl;

/* create parent list, add an entry */
(void) nvlist_alloc(&parent_nvl, NV_UNIQUE_NAME, KM_SLEEP);
(void) nvlist_add_boolean(parent_nvl, "parent_bool", 0);

/* create child list, add an entry */
(void) nvlist_alloc(&child_nvl, NV_UNIQUE_NAME, KM_SLEEP);
(void) nvlist_add_boolean(child_nvl, "child_bool", 0);

/* add the child to the parent */
(void) nvlist_add_nvlist(parent_nvl, "child_nvlist", child_nvl);

/* do stuff .. */

/* free nvlist(s) */
(void) nvlist_free(child_nvl); /* required, but not obvious */
(void) nvlist_free(parent_nvl);
```

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	Committed

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also libnvpair(3LIB), nvlist_alloc(3NVPAIR), 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_double, 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_double(nvlist_t *nvl, const char *name,
     double *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(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	Committed
MT-Level	MT-Safe

See Also libnvpair(3LIB), nvlist_alloc(3NVPAIR), nvpair_type(3NVPAIR), attributes(5)

Name nvlist_lookup_nvpair, nvlist_exists – lookup named pairs

Synopsis

```
cc [ flag... ] file... -lnvpair [ library... ]
#include <libnvpair.h>

int nvlist_lookup_nvpair(nvlist_t *nvl, const char *name,
     nvpair_t **nvp);

boolean_t nvlist_exists(nvlist_t *nvl, const char *name);
```

Description The `nvlist_lookup_nvpair()` function returns the nvpair with the matching name, regardless of type. It is valid only for lists allocated with `NV_UNIQUE_NAME`. See `nvlist_alloc(3NVPAIR)`.

The `nvlist_exists()` function returns success if any nvpair exists with the given name. It is valid for all types of lists.

Return Values The `nvlist_lookup_nvpair()` function returns 0 on success and an error value on failure.

The `nvlist_exists()` function returns `B_TRUE` if an nvpair with the given name exists and `B_FALSE` otherwise.

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

`EINVAL` There is an invalid argument.

`ENOENT` No matching name-value pair is found.

`ENOTSUP` The list was not allocated with `NV_UNIQUE_NAME`.

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

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

See Also `libnvpair(3LIB)`, `nvlist_alloc(3NVPAIR)`, `attributes(5)`, `nvlist_lookup_nvpair(9F)`

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_double, 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_double(nvpair_t *nvpair, double *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	Committed
MT-Level	MT-Safe

See Also libnvpair(3LIB), nvlist_alloc(3NVPAIR), attributes(5)

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 *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)`

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.
`PAM_NO_AUTHTOK_CHECK`	The password service should not perform conformance checks on the password entered.

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 flag PAM_NO_AUTHTOK_CHECK is typically used by programs that allow an administrator to bypass various password conformance checks when setting a password for a user.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
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.

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_NO_AUTHTOK_CHECK`	The password service should not perform conformance checks on the structure of the password. Conformance checks do not apply to verification that the same password was entered during both passes.
`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 *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)

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 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
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_t rooth, 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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Unstable
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)`

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
Interface Stability	Unstable
MT-Level	Safe

See Also libpool(3LIB), pool_error(3POOL), 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 proc_service – process service interfaces

Synopsis

```
#include <proc_service.h>

ps_err_e ps_pdmodel(struct ps_prochandle *ph,
     int *data_model);

ps_err_e ps_pglobal_lookup(struct ps_prochandle *ph,
     const char *object_name, const char *sym_name,
     psaddr_t *sym_addr);

ps_err_e ps_pglobal_sym(struct ps_prochandle *ph,
     const char *object_name, const char *sym_name,
     ps_sym_t *sym);

ps_err_e ps_pread(struct ps_prochandle *ph, psaddr_t addr,
     void *buf, size_t size);

ps_err_e ps_pwrite(struct ps_prochandle *ph, psaddr_t addr,
     const void *buf, size_t size);

ps_err_e ps_pdread(struct ps_prochandle *ph, psaddr_t addr,
     void *buf, size_t size);

ps_err_e ps_pdwrite(struct ps_prochandle *ph, psaddr_t addr,
     const void *buf, size_t size);

ps_err_e ps_ptread(struct ps_prochandle *ph, psaddr_t addr,
     void *buf, size_t size);

ps_err_e ps_ptwrite(struct ps_prochandle *ph, psaddr_t addr,
     const void *buf, size_t size);

ps_err_e ps_pstop(struct ps_prochandle *ph);

ps_err_e ps_pcontinue(struct ps_prochandle *ph);

ps_err_e ps_lstop(struct ps_prochandle *ph, lwpid_t lwpid);

ps_err_e ps_lcontinue(struct ps_prochandle *ph, lwpid_t lwpid);

ps_err_e ps_lgetregs(struct ps_prochandle *ph, lwpid_t lwpid,
     prgregset_t gregset);

ps_err_e ps_lsetregs(struct ps_prochandle *ph, lwpid_t lwpid,
     const prgregset_t gregset);

ps_err_e ps_lgetfpregs(struct ps_prochandle *ph, lwpid_t lwpid,
     prfpregset_t *fpregset);

ps_err_e ps_lsetfpregs(struct ps_prochandle *ph, lwpid_t lwpid,
     const prfpregset_t *fpregset);

ps_err_e ps_pauxv(struct ps_prochandle *ph,
     const auxv_t **auxp);

ps_err_e ps_kill(struct ps_prochandle *ph, int sig);
```

```
ps_err_e ps_lrolltoaddr(struct ps_prochandle *ph,
     lwpid_t lwpid, psaddr_t go_addr, psaddr_t stop_addr);

void ps_plog(const char *fmt);
```

SPARC

```
ps_err_e ps_lgetxregsize(struct ps_prochandle *ph,
     lwpid_t lwpid, int *xregsize);

ps_err_e ps_lgetxregs(struct ps_prochandle *ph,
     lwpid_t lwpid, caddr_t xregset);

ps_err_e ps_lsetxregs(struct ps_prochandle *ph,
     lwpid_t lwpid, caddr_t xregset);
```

x86

```
ps_err_e ps_lgetLDT(struct ps_prochandle *ph, lwpid_t lwpid,
     struct ssd *ldt);
```

Description Every program that links `libthread_db` or `librtld_db` must provide a set of process control primitives that allow `libthread_db` and `librtld_db` to access memory and registers in the target process, to start and to stop the target process, and to look up symbols in the target process. See `libc_db`(3LIB). For information on `librtld_db`, refer to the *Linker and Libraries Guide*.

Refer to the individual reference manual pages that describe these routines for a functional specification that clients of `libthread_db` and `librtld_db` can use to implement this required interface. The <`proc_service.h`> header lists the C declarations of these routines.

Functions

`ps_pdmodel()`	Returns the data model of the target process.
`ps_pglobal_lookup()`	Looks up the symbol in the symbol table of the load object in the target process and returns its address.
`ps_pglobal_sym()`	Looks up the symbol in the symbol table of the load object in the target process and returns its symbol table entry.
`ps_pread()`	Copies *size* bytes from the target process to the controlling process.
`ps_pwrite()`	Copies *size* bytes from the controlling process to the target process.
`ps_pdread()`	Identical to `ps_pread()`.
`ps_pdwrite()`	Identical to `ps_pwrite()`.
`ps_ptread()`	Identical to `ps_pread()`.
`ps_ptwrite()`	Identical to `ps_pwrite()`.
`ps_pstop()`	Stops the target process.
`ps_pcontinue()`	Resumes target process.

	`ps_lstop()`	Stops a single lightweight process (LWP) within the target process.
	`ps_lcontinue()`	Resumes a single LWP within the target process.
	`ps_lgetregs()`	Gets the general registers of the LWP.
	`ps_lsetregs()`	Sets the general registers of the LWP.
	`ps_lgetfpregs()`	Gets the LWP‘s floating point register set.
	`ps_lsetfpregs()`	Sets the LWP‘s floating point register set.
	`ps_pauxv()`	Returns a pointer to a read-only copy of the target process's auxiliary vector.
	`ps_kill()`	Sends signal to target process.
	`ps_lrolltoaddr()`	Rolls the LWP out of a critical section when the process is stopped.
	`ps_plog()`	Logs a message.
SPARC	`ps_lgetxregsize()`	Returns the size of the architecture-dependent extra state registers.
	`ps_lgetxregs()`	Gets the extra state registers of the LWP.
	`ps_lsetxregs()`	Sets the extra state registers of the LWP.
x86	`ps_lgetLDT()`	Reads the local descriptor table of the LWP.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also

`libc_db(3LIB)`, `librtld_db(3LIB)`, `ps_pread(3PROC)`, `rtld_db(3EXT)`, `attributes(5)`

Linker and Libraries Guide

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;
```

EXAMPLE 1 Count the number of projects available on the system. *(Continued)*

```
errno = 0;
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 ps_lgetregs, ps_lsetregs, ps_lgetfpregs, ps_lsetfpregs, ps_lgetxregsize, ps_lgetxregs, ps_lsetxregs – routines that access the target process register in libthread_db

Synopsis

```
#include <proc_service.h>

 ps_err_e ps_lgetregs(struct ps_prochandle *ph, lwpid_t lid,
     prgregset_t gregset);

ps_err_e ps_lsetregs(struct ps_prochandle *ph, lwpid_t lid,
     static prgregset_t gregset);

ps_err_e ps_lgetfpregs(struct ps_prochandle *ph, lwpid_t lid,
     prfpregset_t *fpregs);

ps_err_e ps_lsetfpregs(struct ps_prochandle *ph, lwpid_t lid,
     static prfpregset_t *fpregs);

ps_err_e ps_lgetxregsize(struct ps_prochandle *ph, lwpid_t lid,
     int *xregsize);

ps_err_e ps_lgetxregs(struct ps_prochandle *ph, lwpid_t lid,
     caddr_t xregset);

ps_err_e ps_lsetxregs(struct ps_prochandle *ph, lwpid_t lid,
     caddr_t xregset);
```

Description `ps_lgetregs()`, `ps_lsetregs()`, `ps_lgetfpregs()`, `ps_lsetfpregs()`, `ps_lgetxregsize()`, `ps_lgetxregs()`, `ps_lsetxregs()` read and write register sets from lightweight processes (LWPs) within the target process identified by *ph*. `ps_lgetregs()` gets the general registers of the LWP identified by *lid*, and `ps_lsetregs()` sets them. `ps_lgetfpregs()` gets the LWP's floating point register set, while `ps_lsetfpregs()` sets it.

SPARC Only `ps_lgetxregsize()`,`ps_lgetxregs()`, and`ps_lsetxregs()` are SPARC-specific. They do not need to be defined by a controlling process on non-SPARC architecture. `ps_lgetxregsize()` returns in **xregsize* the size of the architecture-dependent extra state registers. `ps_lgetxregs()` gets the extra state registers, and `ps_lsetxregs()` sets them.

Return Values

`PS_OK`	The call returned successfully.
`PS_NOFPREGS`	Floating point registers are neither available for this architecture nor for this process.
`PS_NOXREGS`	Extra state registers are not available on this architecture.
`PS_ERR`	The function did not return successfully.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also libc_db(3LIB), proc_service(3PROC), attributes(5), threads(5)

Name ps_pglobal_lookup, ps_pglobal_sym – look up a symbol in the symbol table of the load object in the target process

Synopsis

```
#include <proc_service.h>

ps_err_e ps_pglobal_lookup(struct ps_prochandle *ph,
     const char *object_name, const char *sym_name,
     psaddr_t *sym_addr);

ps_err_e ps_pglobal_sym(struct ps_prochandle *ph,
     const char *object_name, const char *sym_name,
     ps_sym_t *sym);
```

Description `ps_pglobal_lookup()` looks up the symbol *sym_name* in the symbol table of the load object *object_name* in the target process identified by *ph*. It returns the symbol's value as an address in the target process in **sym_addr*.

`ps_pglobal_sym()` looks up the symbol *sym_name* in the symbol table of the load object *object_name* in the target process identified by *ph*. It returns the symbol table entry in **sym*. The value in the symbol table entry is the symbol's value as an address in the target process.

Return Values

`PS_OK`	The call completed successfully.
`PS_NOSYM`	The specified symbol was not found.
`PS_ERR`	The function did not return successfully.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also `kill(2)`, `libc_db(3LIB)`, `proc_service(3PROC)`, `attributes(5)`, `threads(5)`

Name ps_pread, ps_pwrite, ps_pdread, ps_pdwrite, ps_ptread, ps_ptwrite – interfaces in libthread_db that target process memory access

Synopsis

```
#include <proc_service.h>

ps_err_e ps_pread(struct ps_prochandle *ph, psaddr_t addr,
     void *buf, size_t size);

ps_err_e ps_pwrite(struct ps_prochandle *ph, psaddr_t addr,
     const void *buf, size_t size);

ps_err_e ps_pdread(struct ps_prochandle *ph, psaddr_t addr,
     void *buf, size_t size);

ps_err_e ps_pdwrite(struct ps_prochandle *ph, psaddr_t addr,
     const void *buf, size_t size);

ps_err_e ps_ptread(struct ps_prochandle *ph, psaddr_t addr,
     void *buf, size_t size);

ps_err_e ps_ptwrite(struct ps_prochandle *ph, psaddr_t addr,
     const void *buf, size_t size);
```

Description These routines copy data between the target process's address space and the controlling process. `ps_pread()` copies *size* bytes from address *addr* in the target process into *buf* in the controlling process. `pr_pwrite()` is like `ps_pread()` except that the direction of the copy is reversed; data is copied from the controlling process to the target process.

`ps_pdread()` and `ps_ptread()` behave identically to `ps_pread()`. `ps_pdwrite()` and `ps_ptwrite()` behave identically to `ps_pwrite()`. These functions can be implemented as simple aliases for the corresponding primary functions. They are artifacts of history that must be maintained.

Return Values

`PS_OK`	The call returned successfully. *size* bytes were copied.
`PS_BADADDR`	Some part of the address range from *addr* through *addr*+*size*−1 is not part of the target process's address space.
`PS_ERR`	The function did not return successfully.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also `libc_db(3LIB)`, `librtld_db(3LIB)`, `proc_service(3PROC)`, `rtld_db(3EXT)`, `attributes(5)`, `threads(5)`

Name ps_pstop, ps_pcontinue, ps_lstop, ps_lcontinue, ps_lrolltoaddr, ps_kill – process and LWP control in libthread_db

Synopsis

```
#include <proc_service.h>

ps_err_e ps_pstop(struct ps_prochandle *ph);

ps_err_e ps_pcontinue(struct ps_prochandle *ph);

ps_err_e ps_lstop(struct ps_prochandle *ph, lwpid_t lwpid);

ps_err_e ps_lcontinue(struct ps_prochandle *ph,
     lwpid_t lwpid);

ps_err_e ps_lrolltoaddr(struct ps_prochandle *ph,
     lwpid_t lwpid, psaddr_t go_addr, psaddr_t stop_addr);

ps_err_e ps_kill(struct ps_prochandle *ph, int signum);
```

Description The `ps_pstop()` function stops the target process identified by *ph*, while the `ps_pcontinue()` function allows it to resume.

The `libthread_db()` function uses `ps_pstop()` to freeze the target process while it is under inspection. Within the scope of any single call from outside `libthread_db` to a `libthread_db` routine, `libthread_db` will call `ps_pstop()`, at most once. If it does, it will call `ps_pcontinue()` within the scope of the same routine.

The controlling process may already have stopped the target process when it calls `libthread_db`. In that case, it is not obligated to resume the target process when `libthread_db` calls `ps_pcontinue()`. In other words, `ps_pstop()` is mandatory, while `ps_pcontinue()` is advisory. After `ps_pstop()`, the target process must be stopped; after `ps_pcontinue()`, the target process may be running.

The `ps_lstop()` and `ps_lcontinue()` functions stop and resume a single lightweight process (LWP) within the target process *ph*.

The `ps_lrolltoaddr()` function is used to roll an LWP forward out of a critical section when the process is stopped. It is also used to run the `libthread_db` agent thread on behalf of `libthread`. The `ps_lrolltoaddr()` function is always called with the target process stopped, that is, there has been a preceding call to `ps_pstop()`. The specified LWP must be continued at the address *go_addr*, or at its current address if *go_addr* is `NULL`. It should then be stopped when its execution reaches *stop_addr*. This routine does not return until the LWP has stopped at *stop_addr*.

The `ps_kill()` function directs the signal *signum* to the target process for which the handle is *ph*. It has the same semantics as `kill(2)`.

Return Values

`PS_OK`	The call completed successfully. In the case of `ps_pstop()`, the target process is stopped.
`PS_BADLID`	For `ps_lstop()`, `ps_lcontinue()` and `ps_lrolltoaddr()`; there is no LWP with id *lwipd* in the target process.
`PS_ERR`	The function did not return successfully.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also

`kill(2)`, `libc_db(3LIB)`, `proc_service(3PROC)`, `attributes(5)`, `threads(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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
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 reparse_add, reparse_create, reparse_delete, reparse_deref, reparse_free, reparse_init, reparse_parse, reparse_remove, reparse_unparse, reparse_validate, rp_plugin_init – reparse point operations

Synopsis

```
cc [ flag ... ] file... -lreparse [ library ... ]
#include <sys/fs_reparse.h>
#include <rp_plugin.h>

int reparse_add(nvlist_t *list, const char *svc_type,
     const char *string);

int reparse_create(const char *path, const char *string);

int reparse_delete(const char *path);

int reparse_deref(const char *svc_type, const char *svc_data,
     const char *svc_data);

void reparse_free(nvlist_t *list);

nvlist_t *reparse_init(void);

int reparse_parse(const char *string, nvlist *list);

int reparse_remove(nvlist_t *list, const char *svc_type);

int reparse_unparse(const nvlist_t *list, char **stringp);

int reparse_validate(const char *string);

int rp_plugin_init(void);
```

Description The `reparse_add()` function adds a service type entry to an `nvlist` with a copy of *string*, replacing one of the same type if already present. This routine will allocate and free memory as needed. It will fail with a non-zero value from `</usr/include/sys/errno.h>` if it is unsuccessful.

The `reparse_create()` function create a reparse point at a given pathname; the string format is validated. This function will fail if path refers to an existing file system object or an object named string already exists at the given path. It will fail with a non-zero value from `</usr/include/sys/errno.h>` if it is unsuccessful.

The `reparse_delete()` function delete a reparse point at a given pathname. It will fail if the pathname is not a symlink. It will fail with a non-zero value from `</usr/include/sys/errno.h>` if it is unsuccessful.

The `reparse_deref()` function accepts and parses the symlink data, and returns a type-specific piece of data in *buf*. The caller specifies the size of the buffer provided via **bufsize*; the routine will fail with `EOVERFLOW` if the results will not fit in the buffer, in which case, **bufsize* will contain the number of bytes needed to hold the results. It can fail with other non-zero values from `</usr/include/sys/errno.h>` if it is unsuccessful.

The `reparse_free()` function frees all of the resources in the `nvlist`.

The `reparse_init()` function allocates an empty `nvlist_t` suitable for `libreparse.so` routines to manipulate. This routine will allocate memory, which must be freed by `reparse_free()`. It will return `NULL` on failure.

The `reparse_parse()` function parses the specified string and populates the nvlist with the *svc_type*s and data from the string. The string could be read from the reparse point symlink body. Existing or duplicate *svc_type* entries in the nvlist will be replaced. This routine will allocate memory that must be freed by `reparse_free()`. It will fail with a non-zero value from `</usr/include/sys/errno.h>` if it is unsuccessful.

The `reparse_remove()` function removes a service type entry from the `nvlist`, if present. This routine will free memory that is no longer needed. It will fail with a non-zero value from `</usr/include/sys/errno.h>` if it is unsuccessful.

The `reparse_unparse()` function converts an `nvlist` back to a string format suitable to write to the reparse point symlink body. The string returned is in allocated memory and must be freed by the caller. It will fail with a non-zero value from `</usr/include/sys/errno.h>` if it is unsuccessful.

The `reparse_validate()` function checks the syntax of a reparse point as it would be read from or written to the symlink body. It will fail with a non-zero value from `</usr/include/sys/errno.h>` if it is unsuccessful.

The `rp_plugin_init()` function loads reparse point "plugins" from `/usr/lib/reparse` to permit reparse point manipulation to start. It will fail with a non-zero value from `</usr/include/sys/errno.h>` if it is unsuccessful.

Examples

EXAMPLE 1 Set up a reparse point.

A service would set up a reparse point this way:

```
nvlist_t        *nvp;
char            *text;
int             rc;

nvp = reparse_init();
rc = reparse_add(nvp, "smb-ad", smb_ad_data);
rc = reparse_add(nvp, "nfs-fedfs", nfs_fedfs_data);
rc = reparse_unparse(nvp, &text);
rc = reparse_create(path, text);
reparse_free(nvp);
free(text);
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also reparsed(1M), libreparse(3LIB), attributes(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 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 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 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)`