man pages section 3: Extended Library Functions, Volume 1

Part No: E29036
October 2012

130304@25097

Contents

Preface

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

Overview

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

- Section 1 describes, in alphabetical order, commands available with the operating system.
- Section 1M describes, in alphabetical order, commands that are used chiefly for system maintenance and administration purposes.
- Section 2 describes all of the system calls. Most of these calls have one or more error returns. An error condition is indicated by an otherwise impossible returned value.
- Section 3 describes functions found in various libraries, other than those functions that directly invoke UNIX system primitives, which are described in Section 2.
- Section 4 outlines the formats of various files. The C structure declarations for the file formats are given where applicable.
- Section 5 contains miscellaneous documentation such as character-set tables.
- Section 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 section 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 1

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

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

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

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

void auto_ef_free(auto_ef_t *info);

char *auto_ef_get_encoding(auto_ef_t info);

double auto_ef_get_score(auto_ef_t info);
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`EINTR` A signal was caught during the execution.

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

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

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

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

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

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

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

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

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

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

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

Synopsis

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

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

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

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

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

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

void config_unload_libs(void);

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

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

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

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

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

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

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

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

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

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

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

Deprecated Interfaces

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

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

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

Description

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The `cfga_list_data` structure includes the following members:

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

The types are defined as follows:

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

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

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

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

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

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

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

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

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

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

Return Values

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

`CFGA_BUSY` — The command was not completed due to an element of the system configuration administration system being busy.

`CFGA_ATTR_INVAL` — No attachment points with the specified attributes exists

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

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

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

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

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

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

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

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

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

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

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

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/core-os, system/library/platform
MT-Level	Safe

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

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

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

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

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

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

Name cpc – hardware performance counters

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

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

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

Shared Counters or Private Counters

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

Configuration Interfaces

The following configuration interfaces are provided:

`cpc_open(3CPC)` Check the version the application was compiled with against the version of the library.

`cpc_cciname(3CPC)` Return a printable string to describe the performance counters of the processor.

`cpc_npic(3CPC)` Return the number of performance counters on the processor.

`cpc_cpuref(3CPC)` Return a reference to documentation that should be consulted to understand how to use and interpret data from the performance counters.

Performance Counter Access

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

Finding Events

Each different type of processor has its own set of events available for measurement. The `cpc_walk_events_all(3CPC)` and `cpc_walk_events_pic(3CPC)` functions allow an application to determine the names of events supported by the underlying processor. A collection of generic, platform independent event names are defined by `generic_events(3CPC)`. Each generic event maps to an underlying hardware event specific to the underlying processor and any optional attributes. The cpc_walk_generic_events_all(3CPC) and cpc_walk_generic_events_pic(3CPC) functions allow an application to determine the generic events supported on the underlying platform.

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

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

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

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

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

See Also

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

Name cpc_access – test access CPU performance counters

Synopsis

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

int cpc_access(void);
```

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

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

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

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

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

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

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

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

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

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

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

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

Synopsis

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

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

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

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

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

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

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

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

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

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

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

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

The cpc_bind_cpu() function binds the set to the specified CPU and measures events occurring on that CPU regardless of which LWP is running. Only one such binding can be active on the specified CPU at a time. As long as any application has bound a set to a CPU, per-LWP counters are unavailable and any attempt to use either cpc_bind_curlwp() or cpc_bind_pctx() returns EAGAIN.

The purpose of the flags argument is to modify the behavior of cpc_bind_cpu() to adapt to different calling strategies.

Values for the *flags* argument are defined in <libcpc.h> as follows:

```
#define CPC_FLAGS_DEFAULT 0
#define CPC_FLAGS_NORELE  0x01
#define CPC_FLAGS_NOPBIND 0x02
```

When flags is set to CPC_FLAGS_DEFAULT, the library binds the calling LWP to the measured CPU with processor_bind(2). The application must not change its processor binding until after it has unbound the set with cpc_unbind().

The remaining *flags* may be used individually or bitwise-OR'ed together.

When only CPC_FLAGS_NORELE is asserted, the library binds the set to the measured CPU using processor_bind(). When the set is unbound using cpc_unbind(), the library will unbind the set but will not unbind the calling thread from the measured CPU.

When only CPC_FLAGS_NOPBIND is asserted, the library does not bind the calling thread the measured CPU when binding the counter set, with the expectation that the calling thread is already bound to the measured CPU. If the thread is not bound to the CPU, the function will fail. When the set is unbound using cpc_unbind(), the library will unbind the set and the calling thread from the measured CPU.

If both flags are asserted (CPC_FLAGS_NOPBIND|CPC_FLAGS_NORELE), the set is bound and unbound from the measured CPU but the calling thread's CPU binding is never altered.

The intended use of CPC_FLAGS_NOPBIND and CPC_FLAGS_NORELE is to allow a thread to cycle through a collection of counter sets without incurring overhead from altering the calling thread's CPU binding unnecessarily.

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

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

The cpc_unbind() function unbinds the set from the resource to which it is bound. All hardware resources associated with the bound set are freed. If the set was bound to a CPU, the calling LWP is unbound from the corresponding CPU according to the policy requested when the set was bound using cpc_bind_cpu().

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

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

These functions will fail if:

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

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

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

EAGAIN For cpc_bind_curlwp() and cpc_bind_pctx(), the performance counters are not available for use by the application.

For cpc_bind_cpu(), another process has already bound to this CPU. Only one process is allowed to bind to a CPU at a time and only one set can be bound to a CPU at a time.

EINVAL The set does not contain any requests or cpc_set_add_request() was not called.

The value given for an attribute of a request is out of range.

The system could not assign a physical counter to each request in the system. See NOTES.

One or more requests in the set conflict and might not be programmed simultaneously.

The *set* was not created with the same *cpc* handle.

For cpc_bind_cpu(), the specified processor does not exist.

For `cpc_unbind()`, the set is not bound.

For `cpc_request_preset()` and `cpc_set_restart()`, the calling LWP does not have a bound set.

`ENOSYS` For `cpc_bind_cpu()`, the specified processor is not online.

`ENOTSUP` The `cpc_bind_curlwp()` function was called with the `CPC_OVF_NOTIFY_EMT` flag, but the underlying processor is not capable of detecting counter overflow.

`ESRCH` For `cpc_bind_pctx()`, the specified LWP in the target process does not exist.

Examples

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

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

```
#include <inttypes.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <libcpc.h>
#include <errno.h>

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

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

if ((event0 = getenv("EVENT0")) == NULL)
     event0 = "EC_ref";
```

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

```
if ((event1 = getenv("EVENT1")) == NULL)
     event1 = "EC_hit";

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

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

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

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

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

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

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

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

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

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

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

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

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

```
cpc_close(cpc);

return (0);
}
```

EXAMPLE 2 Write a signal handler to catch overflow signals.

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

The signal handler:

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

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

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

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

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

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

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

     /*
     * Update a request's preset and restart the counters. Counters which
     * have not been preset with cpc_request_preset() will resume counting
```

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

```
     * from their current value.
     */
     (cpc_request_preset(cpc, ind1, val1) != 0)
        error("cannot set preset for request %d: %s", ind1,
             strerror(errno));
        if (cpc_set_restart(cpc, set) != 0)
             error("cannot restart lwp%d: %s", _lwp_self(), strerror(errno));
}
```

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

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

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

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

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

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

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

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

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

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

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

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

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

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

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

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

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

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

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

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

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

To measure the shared state on a Pentium 4 processor with HyperThreading, the *count_sibling_usr* and *count_sibling_sys* attributes are provided for use with cpc_bind_cpu(). These attributes behave exactly as the CPC_COUNT_USER and CPC_COUNT_SYSTEM request flags, except that they act on the sibling hardware thread sharing the physical processor with the CPU measured by cpc_bind_cpu(). Some CPC sets will fail to bind due to resource constraints. The most common type of resource constraint is an ESCR conflict among one or more requests in the set. For example, the branch_retired event cannot be measured on counters 12 and 13 simultaneously because both counters require the CRU_ESCR2 ESCR to measure this event. To measure *branch_retired* events simultaneously on more than one counter, use counters such that one counter uses CRU_ESCR2 and the other counter uses CRU_ESCR3. See the processor documentation for details.

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

Synopsis

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

int cpc_bind_event(cpc_event_t *event, int flags);

int cpc_take_sample(cpc_event_t *event);

int cpc_rele(void);
```

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

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

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

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

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

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

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

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

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

EINVAL The cpc_take_sample() function has been invoked before the context is bound.

ENOTSUP The caller has attempted an operation that is illegal or not supported on the current platform, such as attempting to specify signal delivery on counter overflow on a CPU that doesn't generate an interrupt on counter overflow.

Usage

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

Examples

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

EXAMPLE 2 Write a signal handler to catch overflow signals.

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

First the signal handler:

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

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

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

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

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

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

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

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

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

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

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

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

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

Note that a more general version of the signal handler would use `write(2)` directly instead of depending on the signal-unsafe semantics of `stderr` and `stdout`. Most real signal handlers

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

will probably do more with the samples than just print them out.

Attributes

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

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

See Also

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

Notes

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

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

SPARC

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

x86

```
rdpmc                ! Pentium II only
```

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

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

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

Handling counter overflow

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

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

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

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

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

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

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

Inheriting events onto multiple LWPs

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Return Values

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

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

Errors

These functions will fail if:

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

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

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

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

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

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

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

SPARC

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

x86

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

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

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

Name cpc_count_usr_events, cpc_count_sys_events – enable and disable performance counters

Synopsis

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

int cpc_count_usr_events(int enable);

int cpc_count_sys_events(int enable);
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Name cpc_enable, cpc_disable – enable and disable performance counters

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

int cpc_enable(cpc_t *cpc);

int cpc_disable(cpc_t *cpc);
```

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

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

Errors These functions will fail if:

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

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

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

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

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

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

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

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

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

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

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

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also

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

Name cpc_event – data structure to describe CPU performance counters

Synopsis `#include <libcpc.h>`

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

SPARC For UltraSPARC, the structure contains the following members:

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

x86 For Pentium, the structure contains the following members:

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

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

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

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

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

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

Name cpc_event_diff, cpc_event_accum – simple difference and accumulate operations

Synopsis

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

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

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

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

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

SPARC:

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

x86:

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

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

SPARC:

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

x86:

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

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

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

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

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

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

Synopsis

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

int cpc_getcpuver(void);

const char *cpc_getcciname(int cpuver);

const char *cpc_getcpuref(int cpuver);

const char *cpc_getusage(int cpuver);

uint_t cpc_getnpic(int cpuver);

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Synopsis

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

uint_t cpc_npic(cpc_t *cpc);

uint_t cpc_caps(cpc_t *cpc);

const char *cpc_cciname(cpc_t *cpc);

const char *cpc_cpuref(cpc_t *cpc);

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

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

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

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

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

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

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

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

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

The system maintains a list of performance counter events supported by the underlying processor. Some processors are able to count all events on all hardware counters, while other processors restrict certain events to be counted only on specific hardware counters. The system also maintains a list of processor-specific attributes that can be used for advanced configuration of the performance counter hardware. These functions allow applications to determine what events and attributes are supported by the underlying processor. The reference work pointed to by `cpc_cpuref()` should be consulted to understand the reasons for and use of the attributes.

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

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

The system maintains a list of platform independent performance counter events known as generic events (see `generic_events`(3CPC)).

The `cpc_walk_generic_events_all()` function calls the action function on each generic event available on the processor. The action function is called for each generic event, regardless of which counter is capable of counting it. The action function is called only once for each event, even if that event can be counted on more than one counter.

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

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

Return Values

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

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

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

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

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

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

ENOMEM There is not enough memory available.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

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

Name cpc_open, cpc_close – initialize the CPU Performance Counter library

Synopsis

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

cpc_t *cpc_open(int vers);

int cpc_close(cpc_t *cpc);
```

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

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

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

The `cpc_close()` function always returns 0.

Errors These functions will fail if:

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

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also libcpc(3LIB), attributes(5)

Name cpc_pctx_bind_event, cpc_pctx_take_sample, cpc_pctx_rele, cpc_pctx_invalidate – access CPU performance counters in other processes

Synopsis

```
cc [ flag... ] file... −lcpc −lpctx [ library... ]
#include <libpctx.h>
#include <libcpc.h>

int cpc_pctx_bind_event(pctx_t *pctx, id_t lwpid, cpc_event_t *event,
     int flags);

int cpc_pctx_take_sample(pctx_t *pctx, id_t lwpid, cpc_event_t *event);

int cpc_pctx_rele(pctx_t *pctx, id_t lwpid);

int cpc_pctx_invalidate(pctx_t *pctx, id_t lwpid);
```

Description These functions are designed to be run in the context of an event handler created using the `libpctx(3LIB)` family of functions that allow the caller, also known as the *controlling process*, to manipulate the performance counters in the context of a *controlled process*. The controlled process is described by the *pctx* argument, which must be obtained from an invocation of `pctx_capture(3CPC)` or `pctx_create(3CPC)` and passed to the functions described on this page in the context of an event handler.

The semantics of the functions `cpc_pctx_bind_event()`, `cpc_pctx_take_sample()`, and `cpc_pctx_rele()` are directly analogous to those of `cpc_bind_event()`, `cpc_take_sample()`, and `cpc_rele()` described on the `cpc_bind_event(3CPC)` manual page.

The `cpc_pctx_invalidate()` function allows the performance context to be invalidated in an LWP in the controlled process.

Return Values These functions return `0` on success. On failure, they return `−1` and set `errno` to indicate the error.

Errors The `cpc_pctx_bind_event()`, `cpc_pctx_take_sample()`, and `cpc_pctx_rele()` functions return the same `errno` values the analogous functions described on the `cpc_bind_event(3CPC)` manual page. In addition, these function may fail if:

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

`ESRCH` The value of the *lwpid* argument is invalid in the context of the controlled process.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe
Interface Stability	Committed

See Also cpc(3CPC), cpc_bind_event(3CPC), libcpc(3LIB), pctx_capture(3CPC), pctx_create(3CPC), attributes(5)

Notes The cpc_pctx_bind_event(), cpc_pctx_invalidate(), cpc_pctx_rele(), and cpc_pctx_take_sample() functions exist for binary compatibility only. Source containing these functions will not compile. These functions are obsolete and might be removed in a future release. Applications should use cpc_bind_pctx(3CPC), cpc_unbind(3CPC), and cpc_set_sample(3CPC) instead.

The capability to create and analyze overflow events in other processes is not available, though it may be made available in a future version of this API. In the current implementation, the *flags* field must be specified as 0.

Name cpc_set_create, cpc_set_destroy, cpc_set_add_request, cpc_walk_requests – manage sets of counter requests

Synopsis

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

cpc_set_t *cpc_set_create(cpc_t *cpc);

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

int cpc_set_add_request(cpc_t *cpc, cpc_set_t *set,
     const char *event, uint64_t preset, uint_t flags,
     uint_t nattrs, const cpc_attr_t *attrs);

void cpc_walk_requests(cpc_t *cpc, cpc_set_t *set, void *arg,
     void (*action)(void *arg, int index, const char *event,
     uint64_t preset, uint_t flags, int nattrs,
     const cpc_attr_t *attrs));
```

Description The `cpc_set_create()` function returns an initialized and empty CPC set. A CPC set contains some number of requests, where a request represents a specific configuration of a hardware performance instrumentation counter present on the processor. The `cpc_set_t` data structure is opaque and must not be accessed directly by the application.

Applications wanting to program one or more performance counters must create an empty set with `cpc_set_create()` and add requests to the set with `cpc_set_add_request()`. Once all requests have been added to a set, the set must be bound to the hardware performance counters (see `cpc_bind_curlwp()`, `cpc_bind_pctx()`, and `cpc_bind_cpu()`, all described on `cpc_bind_curlwp(3CPC)`) before counting events. At bind time, the system attempts to match each request with an available physical counter capable of counting the event specified in the request. If the bind is successful, a 64-bit virtualized counter is created to store the counts accumulated by the hardware counter. These counts are stored and managed in CPC buffers separate from the CPC set whose requests are being counted. See `cpc_buf_create(3CPC)` and `cpc_set_sample(3CPC)`.

The `cpc_set_add_request()` function specifies a configuration of a hardware counter. The arguments to `cpc_set_add_request()` are:

event — A string containing the name of an event supported by the system's processor. The `cpc_walk_events_all()` and `cpc_walk_events_pic()` functions (both described on `cpc_npic(3CPC)`) can be used to query the processor for the names of available events. Certain processors allow the use of raw event codes, in which case a string representation of an event code in a form acceptable to `strtol(3C)` can be used as the *event* argument.

preset — The value with which the system initializes the counter.

flags — Three flags are defined that modify the behavior of the counter acting on behalf of this request:

`CPC_COUNT_USER`
: The counter should count events that occur while the processor is in user mode.

`CPC_COUNT_SYSTEM`
: The counter should count events that occur while the processor is in privileged mode.

`CPC_OVF_NOTIFY_EMT`
: Request a signal to be sent to the application when the physical counter overflows. A `SIGEMT` signal is delivered if the processor is capable of delivering an interrupt when the counter counts past its maximum value. All requests in the set containing the counter that overflowed are stopped until the set is rebound.

At least one of `CPC_COUNT_USER` or `CPC_COUNT_SYSTEM` must be specified to program the hardware for counting.

nattrs, *attrs*
: The *nattrs* argument specifies the number of attributes pointed to by the *attrs* argument, which is an array of `cpc_attr_t` structures containing processor-specific attributes that modify the request's configuration. The `cpc_walk_attrs()` function (see `cpc_npic`(3CPC)) can be used to query the processor for the list of attributes it accepts. The library makes a private copy of the *attrs* array, allowing the application to dispose of it immediately after calling `cpc_set_add_request()`.

The `cpc_walk_requests()` function calls the action function on each request that has been added to the set. The *arg* argument is passed unmodified to the *action* function with each call.

Return Values

Upon successful completion, `cpc_set_create()` returns a handle to the opaque `cpc_set_t` data structure. Otherwise, NULL is returned and `errno` is set to indicate the error.

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

Upon successful completion, `cpc_set_add_request()` returns an integer index used to refer to the data generated by that request during data retrieval. Otherwise, -1 is returned and `errno` is set to indicate the error.

Errors

These functions will fail if:

`EINVAL`
: An event, attribute, or flag passed to `cpc_set_add_request()` was invalid.

: For `cpc_set_destroy()` and `cpc_set_add_request()`, the set parameter was not created with the given cpc_t.

`ENOMEM`
: There was not enough memory available to the process to create the library's data structures.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also cpc_bind_curlwp(3CPC), cpc_buf_create(3CPC), cpc_npic(3CPC), cpc_seterrhndlr(3CPC), libcpc(3LIB), strtol(3C), attributes(5)

Notes The system automatically determines which particular physical counter to use to count the events specified by each request. Applications can force the system to use a particular counter by specifying the counter number in an attribute named *picnum* that is passed to `cpc_set_add_request()`. Counters are numbered from 0 to *n* - 1, where n is the number of counters in the processor as returned by cpc_npic(3CPC).

Some processors, such as UltraSPARC, do not allow the hardware counters to be programmed differently. In this case, all requests in the set must have the same configuration, or an attempt to bind the set will return `EINVAL`. If a `cpc_errhndlr_t` has been registered with cpc_seterrhndlr(3CPC), the error handler is called with subcode `CPC_CONFLICTING_REQS`. For example, on UltraSPARC `pic0` and `pic1` must both program events in the same processor mode (user mode, kernel mode, or both). For example, `pic0` cannot be programmed with `CPC_COUNT_USER` while `pic1` is programmed with `CPC_COUNT_SYSTEM`. Refer to the hardware documentation referenced by cpc_cpuref(3CPC) for details about a particular processor's performance instrumentation hardware.

Name cpc_seterrfn – control libcpc error reporting

Synopsis

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

typedef void (cpc_errfn_t)(const char *fn, const char *fmt, va_list ap);

void cpc_seterrfn(cpc_errfn_t *errfn);
```

Description For the convenience of programmers instrumenting their code, several `libcpc(3LIB)` functions automatically emit to `stderr` error messages that attempt to provide a more detailed explanation of their error return values. While this can be useful for simple programs, some applications may wish to report their errors differently—for example, to a window or to a log file.

The `cpc_seterrfn()` function allows the caller to provide an alternate function for reporting errors; the type signature is shown above. The *fn* argument is passed the library function name that detected the error, the format string *fmt* and argument pointer *ap* can be passed directly to `vsnprintf(3C)` or similar `varargs`-based routine for formatting.

The default printing routine can be restored by calling the routine with an *errfn* argument of `NULL`.

Examples **EXAMPLE 1** Debugging example.

This example produces error messages only when debugging the program containing it, or when the `cpc_strtoevent()` function is reporting an error when parsing an event specification

```
int debugging;
void
myapp_errfn(const char *fn, const char *fmt, va_list ap)
{
        if (strcmp(fn, "strtoevent") != 0 && !debugging)
            return;
        (void) fprintf(stderr, "myapp: cpc_%s(): ", fn);
        (void) vfprintf(stderr, fmt, ap);
}
```

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe
Interface Stability	Obsolete

See Also cpc(3CPC), cpc_seterrhndlr(3CPC), libcpc(3LIB), vsnprintf(3C), attributes(5)

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

Name cpc_seterrhndlr – control libcpc error reporting

Synopsis

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

typedef void(cpc_errhndlr_t)(cpc_t *cpc, const char *fn, int subcode,
     const char *fmt, va_list ap);

void cpc_seterrhndlr(cpc_t *cpc, cpc_errhndlr_t *errfn);
```

Description For the convenience of programmers instrumenting their code, several `libcpc(3LIB)` functions automatically emit to `stderr` error messages that attempt to provide a more detailed explanation of their error return values. While this can be useful for simple programs, some applications might wanat to report their errors differently, for example, to a window or to a log file.

The `cpc_seterrhndlr()` function allows the caller to provide an alternate function for reporting errors. The type signature is shown in the SYNOPSIS. The *fn* argument is passed the library function name that detected the error, an integer subcode indicating the specific error condidtion that has occurred, and the format string *fmt* that contains a textual description of the integer subcode. The format string *fmt* and argument pointer *ap* can be passed directly to `vsnprintf(3C)` or similar *varargs*-based function for formatting.

The integer subcodes are provided to allow programs to recognize error conditions while using `libcpc`. The *fmt* string is provided as a convenience for easy printing. The error subcodes are:

`CPC_INVALID_EVENT`	A specified event is not supported by the processor.
`CPC_INVALID_PICNUM`	The counter number does not fall in the range of available counters.
`CPC_INVALID_ATTRIBUTE`	A specified attribute is not supported by the processor.
`CPC_ATTRIBUTE_OUT_OF_RANGE`	The value of an attribute is outside the range supported by the processor.
`CPC_RESOURCE_UNAVAIL`	A hardware resource necessary for completing an operation was unavailable.
`CPC_PIC_NOT_CAPABLE`	The requested counter cannot count an assigned event.
`CPC_REQ_INVALID_FLAGS`	One or more requests has invalid flags.
`CPC_CONFLICTING_REQS`	The requests in a set cannot be programmed onto the hardware at the same time.
`CPC_ATTR_REQUIRES_PRIVILEGE`	A request contains an attribute which requires the cpc_cpu privilege, which the process does not have.

The default printing routine can be restored by calling the routine with an *errfn* argument of NULL.

Examples

EXAMPLE 1 Debugging example.

The following example produces error messages only when debugging the program containing it, or when the cpc_bind_curlwp(), cpc_bind_cpu(), or cpc_bind_pctx() functions are reporting an error when binding a cpc_set_t.

```
int debugging;
void
myapp_errfn(const char *fn, int subcode, const char *fmt, va_list ap)
{
        if (strncmp(fn, "cpc_bind", 8) != 0 && !debugging)
            return;
        (void) fprintf(stderr, "myapp: cpc_%s(): ", fn);
        (void) vfprintf(stderr, fmt, ap);
}
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also

cpc_bind_curlwp(3CPC), libcpc(3LIB), vsnprintf(3C), attributes(5)

Name cpc_shared_open, cpc_shared_bind_event, cpc_shared_take_sample, cpc_shared_rele, cpc_shared_close – use CPU performance counters on processors

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

int cpc_shared_open(void);

int cpc_shared_bind_event(int fd, cpc_event_t *event, int flags);

int cpc_shared_take_sample(int fd, cpc_event_t *event);

int cpc_shared_rele(int fd);

void cpc_shared_close(int fd);
```

Description The `cpc_shared_open()` function allows the caller to access the hardware counters in such a way that the performance of the currently bound CPU can be measured. The function returns a file descriptor if successful. Only one such open can be active at a time on any CPU.

The `cpc_shared_bind_event()`, `cpc_shared_take_sample()`, and `cpc_shared_rele()` functions are directly analogous to the corresponding `cpc_bind_event()`, `cpc_take_sample()`, and `cpc_rele()` functions described on the `cpc_bind_event`(3CPC)manual page, except that they operate on the counters of a particular processor.

Usage If a thread wishes to access the counters using this interface, it `must` do so using a thread bound to an lwp, (see the `THR_BOUND` flag to `thr_create`(3C)), that has in turn bound itself to a processor using `processor_bind`(2).

Unlike the `cpc_bind_event`(3CPC) family of functions, no counter context is attached to those lwps, so the performance counter samples from the processors reflects the system-wide usage, instead of per-lwp usage.

The first successful invocation of `cpc_shared_open()` will immediately `invalidate` *all* existing performance counter context on the system, and `prevent` *all* subsequent attempts to bind counter context to lwps from succeeding anywhere on the system until the last caller invokes `cpc_shared_close()`.

This is because it is impossible to simultaneously use the counters to accurately measure per-lwp and system-wide events, so there is an exclusive interlock between these uses.

Access to the shared counters is mediated by file permissions on a `cpc` pseudo device. Only a user with the {`PRIV_SYS_CONFIG`} privilege is allowed to access the shared device. This control prevents use of the counters on a per-lwp basis to other users.

The `CPC_BIND_LWP_INHERIT` and `CPC_BIND_EMT_OVF` flags are invalid for the shared interface.

Return Values On success, the functions (except for `cpc_shared_close()`) return 0. On failure, the functions return −1 and set `errno` to indicate the reason.

Errors

`EACCES` The caller does not have appropriate privilege to access the CPU performance counters system-wide.

`EAGAIN` For cpc_shared_open( ), this value implies that the counters on the bound cpu are busy because they are already being used to measure system-wide events by some other caller.

`EAGAIN` Otherwise, this return value implies that the counters are not available because the thread has been unbound from the processor it was bound to at open time. Robust programs should be coded to expect this behavior, and should invoke `cpc_shared_close( )`, before retrying the operation.

`EINVAL` The counters cannot be accessed on the current CPU because the calling thread is not bound to that CPU using `processor_bind(2)`.

`ENOTSUP` The caller has attempted an operation that is illegal or not supported on the current platform.

`ENXIO` The current machine either has no performance counters, or has been configured to disallow access to them system-wide.

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

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

See Also `processor_bind(2)`, `cpc(3CPC)`, `cpc_bind_cpu(3CPC)`, `cpc_bind_event(3CPC)`, `cpc_set_sample(3CPC)`, `cpc_unbind(3CPC)`, `libcpc(3LIB)`, `thr_create(3C)`, `attributes(5)`

Notes The `cpc_shared_open()`, `cpc_shared_bind_event()`, `cpc_shared_take_sample()`, `cpc_shared_rele()`, and `cpc_shared_close()` functions exist for binary compatibility only. Source containing these functions will not compile. These functions are obsolete and might be removed in a future release. Applications should use `cpc_bind_cpu(3CPC)`, `cpc_set_sample(3CPC)`, and `cpc_unbind(3CPC)` instead.

Name cpc_strtoevent, cpc_eventtostr – translate strings to and from events

Synopsis

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

int cpc_strtoevent(int cpuver, const char *spec, cpc_event_t *event);

char *cpc_eventtostr(cpc_event_t *event);
```

Description The `cpc_strtoevent()` function translates an event specification to the appropriate collection of control bits in a `cpc_event_t` structure pointed to by the *event* argument. The event specification is a `getsubopt(3C)`–style string that describes the event and any attributes that the processor can apply to the event or events. If successful, the funciton returns 0, the `ce_cpuver` field and the ISA-dependent control registers of event are initialized appropriately, and the rest of the `cpc_event_t` structure is initialized to 0.

The `cpc_eventtostr()` function takes an event and constructs a compact canonical string representation for that event.

Return Values Upon successful completion, `cpc_strtoevent()` returns 0. If the string cannot be decoded, a non-zero value is returned and a message is printed using the library's error-reporting mechanism (see `cpc_seterrfn(3CPC)`).

Upon successful completion, `cpc_eventtostr()` returns a pointer to a string. The string returned must be freed by the caller using `free(3C)`. If `cpc_eventtostr()` fails, a null pointer is returned.

Usage The event selection syntax used is processor architecture-dependent. The supported processor families allow variations on how events are counted as well as what events can be counted. This information is available in compact form from the `cpc_getusage()` function (see `cpc_getcpuver(3CPC)`), but is explained in further detail below.

UltraSPARC On UltraSPARC processors, the syntax for setting options is as follows:

```
pic0=<eventspec>,pic1=<eventspec> [,sys] [,nouser]
```

This syntax, which reflects the simplicity of the options available using the `%pcr` register, forces both counter events to be selected. By default only user events are counted; however, the `sys` keyword allows system (kernel) events to be counted as well. User event counting can be disabled by specifying the `nouser` keyword.

The keywords `pic0` and `pic1` may be omitted; they can be used to resolve ambiguities if they exist.

Pentium I On Pentium processors, the syntax for setting counter options is as follows:

```
pic0=<eventspec>,pic1=<eventspec> [,sys[[0|1]]] [,nouser[[0|1]]]
[,noedge[[0|1]]] [,pc[[0|1]]]
```

The syntax and semantics are the same as UltraSPARC, except that is possible to specify whether a particular counter counts user or system events. If unspecified, the specification is presumed to apply to both counters.

There are some additional keywords. The `noedge` keyword specifies that the counter should count clocks (duration) instead of events. The `pc` keyword allows the external pin control pins to be set high (defaults to low). When the pin control register is set high, the external pin will be asserted when the associated register overflows. When the pin control register is set low, the external pin will be asserted when the counter has been incremented. The electrical effect of driving the pin is dependent uptoon how the motherboard manufacturer has chosen to connect it, if it is connected at all.

Pentium II

For Pentium II processors, the syntax is substantially more complex, reflecting the complex configuration options available:

```
pic0=<eventspec>,pic1=<eventspec> [,sys[[0|1]]]
[,nouser[[0|1]]] [,noedge[[0|1]]] [,pc[[0|1]]] [,inv[[0|1]]] [,int[[0|1]]]
[,cmask[0|1]=<maskspec>] [,umask[0|1]=<maskspec>]
```

This syntax is a straightforward extension of the earlier syntax. The additional `inv`, `int`, `cmask0`, `cmask1`, `umask0`, and `umask1` keywords allow extended counting semantics. The mask specification is a number between 0 and 255, expressed in hexadecimal, octal or decimal notation.

Examples

SPARC

EXAMPLE 1 SPARC Example.

```
cpc_event_t event;
char *setting = "pic0=EC_ref,pic1=EC_hit"; /* UltraSPARC-specific */

if (cpc_strtoevent(cpuver, setting, &event) != 0)
        /* can't measure 'setting' on this processor */
else
        setting = cpc_eventtostr(&event);
```

Attributes

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

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

See Also

`cpc(3CPC)`, `cpc_getcpuver(3CPC)`, `cpc_set_add_request(3CPC)`, `cpc_seterrfn(3CPC)`, `free(3C)`, `getsubopt(3C)`, `libcpc(3LIB)`, `attributes(5)`

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

These functions are provided as a convenience only. As new processors are usually released asynchronously with software, the library allows the `pic0` and `pic1` keywords to interpret numeric values specified directly in hexadecimal, octal, or decimal.

Name cpc_version – coordinate CPC library and application versions

Synopsis

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

uint_t cpc_version(uint_t version);
```

Description The `cpc_version()` function takes an interface version as an argument and returns an interface version as a result. Usually, the argument will be the value of `CPC_VER_CURRENT` bound to the application when it was compiled.

Return Values If the version requested is still supported by the implementation, `cpc_version()` returns the requested version number and the application can use the facilities of the library on that platform. If the implementation cannot support the version needed by the application, `cpc_version()` returns `CPC_VER_NONE`, indicating that the application will at least need to be recompiled to operate correctly on the new platform, and may require further changes.

If *version* is `CPC_VER_NONE`, `cpc_version()` returns the most current version of the library.

Examples **EXAMPLE 1** Protect an application from using an incompatible library.

The following lines of code protect an application from using an incompatible library:

```
if (cpc_version(CPC_VER_CURRENT) == CPC_VER_NONE) {
        /* version mismatch - library cannot translate */
        exit(1);
}
```

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe

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

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

The version number is used only to express incompatible semantic changes in the performance counter interfaces on the given platform within a single instruction set architecture, for example, when a new set of performance counter registers are added to an existing processor family that cannot be specified in the existing `cpc_event_t` data structure.

Name crypt, setkey, encrypt, des_crypt, des_setkey, des_encrypt, run_setkey, run_crypt, crypt_close – password and file encryption functions

Synopsis

```
cc [ flag ... ] file ... -lcrypt [ library ... ]
#include <crypt.h>

char *crypt(const char *key, const char *salt);

void setkey(const char *key);

void encrypt(char *block, int flag);

char *des_crypt(const char *key, const char *salt);

void des_setkey(const char *key);

void des_encrypt(char *block, int flag);

int run_setkey(int *p, const char *key);

int run_crypt(long offset, char *buffer, unsigned int count,
     int *p);

int crypt_close(int *p);
```

Description `des_crypt()` is the password encryption function. It is based on a one-way hashing encryption algorithm with variations intended (among other things) to frustrate use of hardware implementations of a key search.

key is a user's typed password. *salt* is a two-character string chosen from the set [`a-zA-Z0-9./`]; this string is used to perturb the hashing algorithm in one of 4096 different ways, after which the password is used as the key to encrypt repeatedly a constant string. The returned value points to the encrypted password. The first two characters are the salt itself.

The `des_setkey()` and `des_encrypt()` entries provide (rather primitive) access to the actual hashing algorithm. The argument of `des_setkey()` is a character array of length 64 containing only the characters with numerical value 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is ignored, thereby creating a 56-bit key that is set into the machine. This key is the key that will be used with the hashing algorithm to encrypt the string *block* with the function `des_encrypt()`.

The argument to the `des_encrypt()` entry is a character array of length 64 containing only the characters with numerical value 0 and 1. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the hashing algorithm using the key set by `des_setkey()`. If *flag* is zero, the argument is encrypted; if non-zero, it is decrypted.

Note that decryption is not provided in the international version of `crypt()`. The international version is part of the C Development Set, and the domestic version is part of the Security Administration Utilities. If decryption is attempted with the international version of `des_encrypt()`, an error message is printed.

crypt(), setkey(), and encrypt() are front-end routines that invoke des_crypt(), des_setkey(), and des_encrypt() respectively.

The routines run_setkey() and run_crypt() are designed for use by applications that need cryptographic capabilities, such as ed(1) and vi(1). run_setkey() establishes a two-way pipe connection with the crypt utility, using *key* as the password argument. run_crypt() takes a block of characters and transforms the cleartext or ciphertext into their ciphertext or cleartext using the crypt utility. *offset* is the relative byte position from the beginning of the file that the block of text provided in *block* is coming from. *count* is the number of characters in *block*, and *connection* is an array containing indices to a table of input and output file streams. When encryption is finished, crypt_close() is used to terminate the connection with the crypt utility.

run_setkey() returns −1 if a connection with the crypt utility cannot be established. This result will occur in international versions of the UNIX system in which the crypt utility is not available. If a null key is passed to run_setkey(), 0 is returned. Otherwise, 1 is returned. run_crypt() returns −1 if it cannot write output or read input from the pipe attached to crypt(). Otherwise it returns 0.

The program must be linked with the object file access routine library libcrypt.a.

Return Values

In the international version of crypt(), a flag argument of 1 to encrypt() or des_encrypt() is not accepted, and errno is set to ENOSYS to indicate that the functionality is not available.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also

ed(1), login(1), passwd(1), vi(1), getpass(3C), passwd(4), attributes(5)

Notes

The return value in crypt() points to static data that are overwritten by each call.

Name ct_ctl_adopt, ct_ctl_abandon, ct_ctl_newct, ct_ctl_ack, ct_ctl_nack, ct_ctl_qack – common contract control functions

Synopsis

```
cc [ flag... ] file... -D_LARGEFILE64_SOURCE  -lcontract  [ library... ]
#include <libcontract.h>

int ct_ctl_adopt(int fd);

int ct_ctl_abandon(int fd);

int ct_ctl_newct(int fd, uint64_t evid, int templatefd);

int ct_ctl_ack(int fd, uint64_t evid);

int ct_ctl_nack(int fd, uint64_t evid);

int ct_ctl_qack(int fd, uint64_t evid);
```

Description These functions operate on contract control file descriptors obtained from the `contract(4)` file system.

The `ct_ctl_adopt()` function adopts the contract referenced by the file descriptor *fd*. After a successful call to `ct_ctl_adopt()`, the contract is owned by the calling process and any events in that contract's event queue are appended to the process's bundle of the appropriate type.

The `ct_ctl_abandon()` function abandons the contract referenced by the file descriptor *fd*. After a successful call to `ct_ctl_abandon()` the process no longer owns the contract, any events sent by that contract are automatically removed from the process's bundle, and any critical events on the contract's event queue are automatically acknowledged. Depending on its type and terms, the contract will either be orphaned or destroyed.

The `ct_ctl_ack()` function acknowledges the critical event specified by*evid*. If the event corresponds to an exit negotiation, `ct_ctl_ack()` also indicates that the caller is prepared for the system to proceed with the referenced reconfiguration.

The `ct_ctl_nack()` function acknowledges the critical negotiation event specified by *evid*. The `ct_ctl_nack()` function also indicates that the caller wishes to block the proposed reconfiguration indicated by event *evid*. Depending on the contract type, this function might require certain privileges to be asserted in the process's effective set. This function will fail and return an error if the event represented by *evid* is not a negotiation event.

The `ct_ctl_qack()` function requests a new quantum of time for the negotiation specified by the event ID *evid*.

The `ct_ctl_newct()` function instructs the contract specified by the file descriptor *fd* that when the current exit negotiation completes, another contract with the terms provided by the template specified by *templatefd* should be automatically written.

Return Values Upon successful completion, `ct_ctl_adopt()`, `ct_ctl_abandon()`, `ct_ctl_newct()`, `ct_ctl_ack()`, and `ct_ctl_qack()` return 0. Otherwise, they return a non-zero error value.

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

`EBUSY` The contract is in the owned state.

`EINVAL` The contract was not inherited by the caller's process contract or was created by a process in a different zone.

The `ct_ctl_abandon()`, `ct_ctl_newct()`, `ct_ctl_ack()`, `ct_ctl_nack()`, and `ct_ctl_qack()` functions will fail if:

`EBUSY` The contract does not belong to the calling process.

The `ct_ctl_newct()` and `ct_ctl_qack()` functions will fail if:

`ESRCH` The event ID specified by *evid* does not correspond to an unacknowledged negotiation event.

The `ct_ctl_newct()` function will fail if:

`EINVAL` The file descriptor specified by *fd* was not a valid template file descriptor.

The `ct_ctl_ack()` and `ct_ctl_nack()` functions will fail if:

`ESRCH` The event ID specified by *evid* does not correspond to an unacknowledged negotiation event.

The `ct_ctl_nack()` function will fail if:

`EPERM` The calling process lacks the appropriate privileges required to block the reconfiguration.

The `ct_ctl_qack()` function will fail if:

`ERANGE` The maximum amount of time has been requested.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libcontract(3LIB)`, `contract(4)`, `attributes(5)`, `lfcompile(5)`

Name ct_dev_status_get_dev_state, ct_dev_status_get_aset, ct_dev_status_get_minor, ct_dev_status_get_noneg – read contract status information from a status object

Synopsis

```
cc [ flag... ] file... -D_LARGEFILE64_SOURCE  -lcontract  [ library... ]
#include <libcontract.h>
#include <sys/contract/device.h>

int ct_dev_status_get_dev_state(ct_stathdl_t stathdl,
     uint_t *statep);

int ct_dev_status_get_aset(ct_stathdl_t stathdl,
     uint_t *asetp);

int ct_dev_status_get_minor(ct_stathdl_t stathdl, char *buf,
     size_t *buflenp);

int ct_dev_status_get_noneg(ct_stathdl_t stathdl,
     uint_t *nonegp);
```

Parameters

asetp a pointer to a `uint_t` variable for receiving the acceptable state set (such as A-set) for the contract

buf a buffer for receiving the `devfs` path of a minor in a contract

buflenp a pointer to a variable of type `size_t` for passing the size of the buffer *buf*. If the buffer is too small (< `PATH_MAX`), the minimum size of the buffer needed (`PATH_MAX`) is passed back to the caller with this argument.

nonegp a pointer to a `uint_t` variable for receiving the setting of the "noneg" term

stathdl a status object returned by `ct_status_read`(3CONTRACT)

statep a pointer to a `uint_t` variable for receiving the current state of the device which is the subject of the contract

Description These functions read contract status information from a status object *stathdl* returned by `ct_status_read()`. The detail level in the call to `ct_status_read()` needs to be at least `CTD_FIXED` for the following calls to be successful. The one exception is `ct_dev_status_get_minor()`, which requires a detail level of `CTD_ALL`.

The `ct_dev_status_get_dev_state()` function returns the current state of the device which is the subject of the contract. This can be one of the following:

`CT_DEV_EV_ONLINE` The device is online and functioning normally.

`CT_DEV_EV_DEGRADED` The device is online but degraded.

`CT_DEV_EV_OFFLINE` The device is offline and not configured.

The `ct_dev_status_get_aset()` function returns the A-set of the contract. This can be the bitset of one or more of the following states: `CT_DEV_EV_ONLINE`, `CT_DEV_EV_DEGRADED`, or `CT_DEV_EV_OFFLINE`.

The `ct_dev_status_get_minor()` function reads the `devfs` path of the minor participating in the contract. The `devfs` path returned does not include the `/devices` prefix. If the buffer passed in by the caller is too small (< `PATH_MAX`), the minimum size of the buffer required (`PATH_MAX`) is returned to the caller via the *buflenp* argument.

The `ct_dev_status_get_noneg()` function returns the "noneg" setting for the contract. A value of 1 is returned in the *nonegp* argument if `NONEG` is set, else 0 is returned.

Return Values Upon successful completion, these functions return 0. Otherwise, they return a non-zero error value.

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

`EOVERFLOW` The buffer size is too small to hold the result.

The `ct_dev_status_get_dev_state()`, `ct_dev_status_get_aset()`, `ct_dev_status_get_minor()` and `ct_dev_status_get_noneg()` functions will fail if:

`EINVAL` An invalid argument was specified.

`ENOENT` The requested data is not present in the status object.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `ct_status_free(3CONTRACT)`, `ct_status_read(3CONTRACT)`, `libcontract(3LIB)`, `contract(4)`, `devices(4)`, `attributes(5)`, `lfcompile(5)`

Name ct_dev_tmpl_set_aset, ct_dev_tmpl_get_aset, ct_dev_tmpl_set_minor, ct_dev_tmpl_get_minor, ct_dev_tmpl_set_noneg, ct_dev_tmpl_clear_noneg, ct_dev_tmpl_get_noneg – device contract template functions

Synopsis

```
cc [ flag... ] file... -D_LARGEFILE64_SOURCE  -lcontract  [ library... ]
#include <libcontract.h>
#include <sys/contract/device.h>

int ct_dev_tmpl_set_aset(int fd, uint_t aset);

int ct_dev_tmpl_get_aset(int fd, uint_t *asetp);

int ct_dev_tmpl_set_minor(int fd, char *minor);

int ct_dev_tmpl_get_minor(int fd, char *buf, size_t *buflenp);

int ct_dev_tmpl_set_noneg(int fd);

int ct_dev_tmpl_clear_noneg(int fd);

int ct_dev_tmpl_get_noneg(int fd, uint_t *nonegp);
```

Parameters

aset a bitset of one or more of device states

asetp a pointer to a variable into which the current A-set is to be returned

buf a buffer into which the minor path is to be returned

buflenp a pointer to variable of type `size_t` in which the size of the buffer *buf* is passed in. If the buffer is too small the size of the buffer needed for a successful call is passed back to the caller.

fd a file descriptor from an open of the device contract template file in the contract filesystem (ctfs)

minor the `devfs` path (the `/devices` path without the "`/devices`" prefix) of a minor which is to be the subject of a contract

nonegp a pointer to a *uint_t* variable for receiving the current setting of the "nonnegotiable" term in the template

Description These functions read and write device contract terms and operate on device contract template file descriptors obtained from the `contract(4)` filesystem (ctfs).

The `ct_dev_tmpl_set_aset()` and `ct_dev_tmpl_get_aset()` functions write and read the "acceptable states" set (or A-set for short). This is the set of device states guaranteed by the contract. Any departure from these states will result in the breaking of the contract and a delivery of a critical contract event to the contract holder. The A-set value is a bitset of one or more of the following device states: `CT_DEV_EV_ONLINE`, `CT_DEV_EV_DEGRADED`, and `CT_DEV_EV_OFFLINE`.

The `ct_dev_tmpl_set_minor()` and `ct_dev_tmpl_get_minor()` functions write and read the minor term (the device resource that is to be the subject of the contract.) The value is a `devfs` path to a device minor node (minus the "`/devices`" prefix). For the `ct_dev_tmpl_get_minor()` function, a buffer at least `PATH_MAX` in size must be passed in. If the buffer is smaller than `PATH_MAX`, then the minimum size of the buffer required (`PATH_MAX`) for this function is passed back to the caller via the *buflenp* argument.

The `ct_dev_tmpl_set_noneg()` and `ct_dev_tmpl_get_noneg()` functions write and read the nonnegotiable term. If this term is set, synchronous negotiation events are automatically NACKed on behalf of the contract holder. For `ct_dev_tmpl_get_noneg()`, the variable pointed to by *nonegp* is set to 1 if the "noneg" term is set or to 0 otherwise. The `ct_dev_tmpl_clear_noneg()` term clears the nonnegotiable term from a template.

Return Values Upon successful completion, these functions return 0. Otherwise, they return a non-zero error value.

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

`EINVAL` A template file descriptor or A-set is invalid

The `ct_dev_tmpl_set_minor()` function will fail if:

`EINVAL` One or more arguments is invalid.

`ENXIO` The minor named by minor path does not exist.

The `ct_dev_tmpl_set_noneg()` function will fail if:

`EPERM` A process lacks sufficient privilege to NACK a device state change.

The `ct_dev_tmpl_get_aset()` and `ct_dev_tmpl_get_minor()` functions will fail if:

`EINVAL` One or more arguments is invalid.

`ENOENT` The requested term is not set.

The `ct_dev_tmpl_get_noneg()` function will fail if:

`EINVAL` One or more arguments is invalid.

The `ct_dev_tmpl_get_minor()` function will fail if:

`EOVEFLOW` The supplied buffer is too small.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also libcontract(3LIB), contract(4), devices(4), attributes(5), lfcompile(5)

Name ct_event_read, ct_event_read_critical, ct_event_reset, ct_event_reliable, ct_event_free, ct_event_get_flags, ct_event_get_ctid, ct_event_get_evid, ct_event_get_type, ct_event_get_nevid, ct_event_get_newct – common contract event functions

Synopsis

```
cc [ flag... ] file... -D_LARGEFILE64_SOURCE  -lcontract  [ library... ]
#include <libcontract.h>

int ct_event_read(int fd, ct_evthdl_t *evthndlp);

int ct_event_read_critical(int fd, ct_evthdl_t *evthndlp);

int ct_event_reset(int fd);

int ct_event_reliable(int fd);

void ct_event_free(ct_evthdl_t evthndl);

ctid_t ct_event_get_ctid(ct_evthdl_t evthndl);

ctevid_t ct_event_get_evid(ct_evthdl_t evthndl);

uint_t ct_event_get_flags(ct_evthdl_t evthndl);

uint_t ct_event_get_type(ct_evthdl_t evthndl);

int ct_event_get_nevid(ct_evthdl_t evthndl, ctevid_t *evidp);

int ct_event_get_newct(ct_evthdl_t evthndl, ctid_t *ctidp);
```

Description These functions operate on contract event endpoint file descriptors obtained from the `contract(4)` file system and event object handles returned by `ct_event_read()` and `ct_event_read_critical()`.

The `ct_event_read()` function reads the next event from the queue referenced by the file descriptor *fd* and initializes the event object handle pointed to by *evthndlp*. After a successful call to `ct_event_read()`, the caller is responsible for calling `ct_event_free()` on this event object handle when it has finished using it.

The `ct_event_read_critical()` function behaves like `ct_event_read()` except that it reads the next critical event from the queue, skipping any intermediate events.

The `ct_event_reset()` function resets the location of the listener to the beginning of the queue. This function can be used to re-read events, or read events that were sent before the event endpoint was opened. Informative and acknowledged critical events, however, might have been removed from the queue.

The `ct_event_reliable()` function indicates that no event published to the specified event queue should be dropped by the system until the specified listener has read the event. This function requires that the caller have the {`PRIV_CONTRACT_EVENT`} privilege in its effective set.

The `ct_event_free()` function frees any storage associated with the event object handle specified by *evthndl*.

The `ct_event_get_ctid()` function returns the ID of the contract that sent the specified event.

The `ct_event_get_evid()` function returns the ID of the specified event.

The `ct_event_get_flags()` function returns the event flags for the specified event. Valid event flags are:

`CTE_INFO` The event is an informative event.

`CTE_ACK` The event has been acknowledged (for critical and negotiation messages).

`CTE_NEG` The message represents an exit negotiation.

The `ct_event_get_type()` function reads the event type. The value is one of the event types described in `contract(4)` or the contract type's manual page.

The `ct_event_get_nevid()` function reads the negotiation ID from an `CT_EV_NEGEND` event.

The `ct_event_get_newct()` function obtains the ID of the contract created when the negotiation referenced by the `CT_EV_NEGEND` event succeeded. If no contract was created, *ctidp* will be 0. If the operation was cancelled, **ctidp* will equal the ID of the existing contract.

Return Values Upon successful completion, `ct_event_read()`, `ct_event_read_critical()`, `ct_event_reset()`, `ct_event_reliable()`, `ct_event_get_nevid()`, and `ct_event_get_newct()` return 0. Otherwise, they return a non-zero error value.

The `ct_event_get_flags()`, `ct_event_get_ctid()`, `ct_event_get_evid()`, and `ct_event_get_type()` functions return data as described in the DESCRIPTION.

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

`EPERM` The caller does not have {`PRIV_CONTRACT_EVENT`} in its effective set.

The `ct_event_read()` and `ct_event_read_critical()` functions will fail if:

`EAGAIN` The event endpoint was opened `O_NONBLOCK` and no applicable events were available to be read.

The The `ct_event_get_nevid()` and `ct_event_get_newct()` functions will fail if:

`EINVAL` The *evthndl* argument is not a `CT_EV_NEGEND` event object.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also `libcontract`(3LIB), `contract`(4), `attributes`(5), `lfcompile`(5)

Name ct_pr_event_get_pid, ct_pr_event_get_ppid, ct_pr_event_get_signal, ct_pr_event_get_sender, ct_pr_event_get_senderct, ct_pr_event_get_exitstatus, ct_pr_event_get_pcorefile, ct_pr_event_get_gcorefile, ct_pr_event_get_zcorefile – process contract event functions

Synopsis

```
cc [ flag... ] file... -D_LARGEFILE64_SOURCE  -lcontract  [ library... ]
#include <libcontract.h>
#include <sys/contract/process.h>

int ct_pr_event_get_pid(ct_evthdl_t evthdl, pid_t *pidp);

int ct_pr_event_get_ppid(ct_evthdl_t evthdl, pid_t *pidp);

int ct_pr_event_get_signal(ct_evthdl_t evthdl, int *signalp);

int ct_pr_event_get_sender(ct_evthdl_t evthdl, pid_t *pidp);

int ct_pr_event_get_senderct(ct_evthdl_t evthdl, ctid_t *pidp);

int ct_pr_event_get_exitstatus(ct_evthdl_t evthdl, int *statusp);

int ct_pr_event_get_pcorefile(ct_evthdl_t evthdl, char **namep);

int ct_pr_event_get_gcorefile(ct_evthdl_t evthdl, char **namep);

int ct_pr_event_get_zcorefile(ct_evthdl_t evthdl, char **namep);
```

Description These functions read process contract event information from an event object returned by `ct_event_read`(3CONTRACT) or `ct_event_read_critical`(3CONTRACT).

The `ct_pr_event_get_pid()` function reads the process ID of the process generating the event.

The `ct_pr_event_get_ppid()` function reads the process ID of the process that forked the new process causing the `CT_PR_EV_FORK` event.

The `ct_pr_event_get_signal()` function reads the signal number of the signal that caused the `CT_PR_EV_SIGNAL` event.

The `ct_pr_event_get_sender()` function reads the process ID of the process that sent the signal that caused the `CT_PR_EV_SIGNAL` event. If the signal's sender was not in the same zone as the signal's recipient, this information is available only to event consumers in the global zone.

The ct_pr_event_get_senderct function reads the contract ID of the process that sent the signal that caused the CT_PR_EV_SIGNAL event. If the signal's sender was not in the same zone as the signal's recipient, this information is available only

The `ct_pr_event_get_exitstatus()` function reads the exit status of the process generating a `CT_PR_EV_EXIT` event.

The `ct_pr_event_get_pcorefile()` function reads the name of the process core file if one was created when the `CT_PR_EV_CORE` event was generated. A pointer to a character array is stored in **namep* and is freed when `ct_event_free`(3CONTRACT) is called on the event handle.

The `ct_pr_event_get_gcorefile()` function reads the name of the zone's global core file if one was created when the `CT_PR_EV_CORE` event was generated. A pointer to a character array is stored in **namep* and is freed when `ct_event_free()` is called on the event handle.

The `ct_pr_event_get_zcorefile()` function reads the name of the system-wide core file in the global zone if one was created when the `CT_PR_EV_CORE` event was generated. This information is available only to event consumers in the global zone. A pointer to a character array is stored in **namep* and is freed when `ct_event_free()` is called on the event handle.

Return Values Upon successful completion, `ct_pr_event_get_pid()`, `ct_pr_event_get_ppid()`, `ct_pr_event_get_signal()`, `ct_pr_event_get_sender()`, `ct_pr_event_get_senderct()`, `ct_pr_event_get_exitstatus()`, `ct_pr_event_get_pcorefile()`, `ct_pr_event_get_gcorefile()`, and `ct_pr_event_get_zcorefile()` return 0. Otherwise, they return a non-zero error value.

Errors The `ct_pr_event_get_pid()`, `ct_pr_event_get_ppid()`, `ct_pr_event_get_signal()`, `ct_pr_event_get_sender()`, `ct_pr_event_get_senderct()`, `ct_pr_event_get_exitstatus()`, `ct_pr_event_get_pcorefile()`, `ct_pr_event_get_gcorefile()`, and `ct_pr_event_get_zcorefile()` functions will fail if:

`EINVAL` The *evthdl* argument is not a process contract event object.

The `ct_pr_event_get_ppid()`, `ct_pr_event_get_signal()`, `ct_pr_event_get_sender()`, `ct_pr_event_get_senderct()`, `ct_pr_event_get_exitstatus()`, `ct_pr_event_get_pcorefile()`, `ct_pr_event_get_gcorefile()`, and `ct_pr_event_get_zcorefile()` functions will fail if:

`EINVAL` The requested data do not match the event type.

The `ct_pr_event_get_sender()`a functions will fail if:

`ENOENT` The process ID of the sender was not available, or the event object was read by a process running in a non-global zone and the sender was in a different zone.

The `ct_pr_event_get_pcorefile()`, `ct_pr_event_get_gcorefile()`, and `ct_pr_event_get_zcorefile()` functions will fail if:

`ENOENT` The requested core file was not created.

The `ct_pr_event_get_zcorefile()` function will fail if:

`ENOENT` The event object was read by a process running in a non-global zone.

Examples

EXAMPLE 1 Print the instigator of all CT_PR_EV_SIGNAL events.

Open the process contract bundle. Loop reading events. Fetch and display the signalled pid and signalling pid for each `CT_PR_EV_SIGNAL` event encountered.

```
#include <sys/types.h>
#include <fcntl.h>
#include <stdio.h>
#include <libcontract.h>


...
int fd;
ct_evthdl_t event;
pid_t pid, sender;

fd = open("/system/contract/process/bundle", O_RDONLY);
for (;;) {
        ct_event_read(fd, &event);
        if (ct_event_get_type(event) != CT_PR_EV_SIGNAL) {
                ct_event_free(event);
                continue;
        }
        ct_pr_event_get_pid(event, &pid);
        if (ct_pr_event_get_sender(event, &sender) == ENOENT)
                printf("process %ld killed by unknown process\n",
                    (long)pid);
        else
                printf("process %ld killed by process %ld\n",
                    (long)pid, (long)sender);
        ct_event_free(event);
}
        ...
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also

`ct_event_free(3CONTRACT)`, `ct_event_read(3CONTRACT)`, `ct_event_read_critical(3CONTRACT)`, `libcontract(3LIB)`, `contract(4)`, `process(4)`, `attributes(5)`, `lfcompile(5)`

Name ct_pr_status_get_param, ct_pr_status_get_fatal, ct_pr_status_get_members, ct_pr_status_get_contracts, ct_pr_status_get_svc_fmri, ct_pr_status_get_svc_aux, ct_pr_status_get_svc_ctid, ct_pr_status_get_svc_creator – process contract status functions

Synopsis

```
cc [ flag... ] file... -D_LARGEFILE64_SOURCE  -lcontract  [ library... ]
#include <libcontract.h>
#include <sys/contract/process.h>

int ct_pr_status_get_param(ct_stathdl_t stathdl, uint_t *paramp);

int ct_pr_status_get_fatal(ct_stathdl_t stathdl, uint_t *eventsp);

int ct_pr_status_get_members(ct_stathdl_t stathdl,
     pid_t **pidpp, uint_t *n);

int ct_pr_status_get_contracts(ct_stathdl_t stathdl,
     ctid_t **idpp, uint_t *n);

int ct_pr_status_get_svc_fmri(ct_stathdl_t stathdl, char **fmri);

int ct_pr_status_get_svc_aux(ct_stathdl_t stathdl, char **aux);

int ct_pr_status_get_svc_ctid(ct_stathdl_t stathdl, ctid_t *ctid);

int ct_pr_status_get_svc_creator(ct_stathdl_t stathdl,
     char **creator);
```

Description These functions read process contract status information from a status object returned by `ct_status_read`(3CONTRACT).

The `ct_pr_status_get_param()` function reads the parameter set term. The value is a collection of bits as described in `process`(4).

The `ct_pr_status_get_fatal()` function reads the fatal event set term. The value is a collection of bits as described in `process`(4).

The `ct_pr_status_get_members()` function obtains a list of the process IDs of the members of the process contract. A pointer to an array of process IDs is stored in **pidpp*. The number of elements in this array is stored in **n*. These data are freed when the status object is freed by a call to `ct_status_free`(3CONTRACT).

The `ct_pr_status_get_contracts()` function obtains a list of IDs of contracts that have been inherited by the contract. A pointer to an array of IDs is stored in **idpp*. The number of elements in this array is stored in **n*. These data are freed when the status object is freed by a call to `ct_status_free()`.

The `ct_pr_status_get_svc_fmri()`, `ct_pr_status_get_svc_creator()`, and `ct_pr_status_get_svc_aux()` functions read, respectively, the service FMRI, the contract's creator execname and the creator's auxiliary field. The buffer pointed to by *fmri*, *aux* or *creator*, is freed by a call to `ct_status_free()` and should not be modified.

The ct_pr_status_get_svc_ctid() function reads the process contract id for which the service FMRI was first set.

Return Values

Upon successful completion, ct_pr_status_get_param(), ct_pr_status_get_fatal(), ct_pr_status_get_members(), ct_pr_status_get_contracts(), ct_pr_status_get_svc_fmri(), ct_pr_status_get_svc_creator(), ct_pr_status_get_svc_aux(), and ct_pr_status_get_svc_ctid() return 0. Otherwise, they return a non-zero error value.

Errors

The ct_pr_status_get_param(), ct_pr_status_get_fatal(), ct_pr_status_get_members(), ct_pr_status_get_contracts(), ct_pr_status_get_svc_fmri(), ct_pr_status_get_svc_creator(), ct_pr_status_get_svc_aux(), and ct_pr_status_get_svc_ctid() functions will fail if:

EINVAL The *stathdl* argument is not a process contract status object.

The ct_pr_status_get_param(), ct_pr_status_get_fatal(), ct_pr_status_get_members(), ct_r_status_get_contracts(), ct_pr_status_get_svc_fmri(), ct_pr_status_get_svc_creator(), ct_pr_status_get_svc_aux(), and ct_pr_status_get_svc_ctid() functions will fail if:

ENOENT The requested data were not available in the status object.

Examples

EXAMPLE 1 Print members of process contract 1.

Open the status file for contract 1, read the contract's status, obtain the list of processes, print them, and free the status object.

```
#include <sys/types.h>
#include <fcntl.h>
#include <libcontract.h>
#include <stdio.h>

...
int fd;
uint_t i, n;
pid_t *procs;
ct_stathdl_t st;

fd = open("/system/contract/process/1/status");
ct_status_read(fd, &st);
ct_pr_status_get_members(st, &procs, &n);
for (i = 0 ; i < n; i++)
        printf("%ld\n", (long)procs[i]);
ct_status_free(stat);
close(fd);
...
```

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also ct_status_free(3CONTRACT), ct_status_read(3CONTRACT), libcontract(3LIB), contract(4), process(4), attributes(5), lfcompile(5)

Name ct_pr_tmpl_set_transfer, ct_pr_tmpl_set_fatal, ct_pr_tmpl_set_param, ct_pr_tmpl_set_svc_fmri, ct_pr_tmpl_set_svc_aux, ct_pr_tmpl_get_transfer, ct_pr_tmpl_get_fatal, ct_pr_tmpl_get_param, ct_pr_tmpl_get_svc_fmri, ct_pr_tmpl_get_svc_aux – process contract template functions

Synopsis

```
cc [ flag... ] file... -D_LARGEFILE64_SOURCE  -lcontract  [ library... ]
#include <libcontract.h>
#include <sys/contract/process.h>

int ct_pr_tmpl_set_transfer(int fd, ctid_t ctid);

int ct_pr_tmpl_set_fatal(int fd, uint_t events);

int ct_pr_tmpl_set_param(int fd, uint_t params);

int ct_pr_tmpl_set_svc_fmri(int fd, const char *fmri);

int ct_pr_tmpl_set_svc_aux(int fd, const char *aux);

int ct_pr_tmpl_get_transfer(int fd, ctid_t *ctidp);

int ct_pr_tmpl_get_fatal(int fd, uint_t *eventsp);

int ct_pr_tmpl_get_param(int fd, uint_t *paramsp);

int ct_pr_tmpl_get_svc_fmri(int fd, char *fmri, size_t size);

int ct_pr_tmpl_get_svc_aux(int fd, char *aux, size_t size);
```

Description These functions read and write process contract terms and operate on process contract template file descriptors obtained from the `contract(4)` file system.

The `ct_pr_tmpl_set_transfer()` and `ct_pr_tmpl_get_transfer()` functions write and read the transfer contract term. The value is the ID of an empty regent process contract owned by the caller whose inherited contracts are to be transferred to a newly created contract.

The `ct_pr_tmpl_set_fatal()` and `ct_pr_tmpl_get_fatal()` functions write and read the fatal event set term. The value is a collection of bits as described in `process(4)`.

The `ct_pr_tmpl_set_param()` and `ct_pr_tmpl_get_param()` functions write and read the parameter set term. The value is a collection of bits as described in `process(4)`.

The `ct_pr_tmpl_set_svc_fmri()` and `ct_pr_tmpl_get_svc_fmri()` functions write and read the service FMRI value of a process contract template. The `ct_pr_tmpl_set_svc_fmri()` function requires the caller to have the {`PRIV_CONTRACT_IDENTITY`} privilege in its effective set.

The `ct_pr_tmpl_set_svc_aux()` and `ct_pr_tmpl_get_svc_aux()` functions write and read the creator's auxiliary value of a process contract template.

Return Values Upon successful completion, `ct_pr_tmpl_set_transfer()`, `ct_pr_tmpl_set_fatal()`, `ct_pr_tmpl_set_param()`, `ct_pr_tmpl_set_svc_fmri()`, `ct_pr_tmpl_set_svc_aux()`, `ct_pr_tmpl_get_transfer()`, `ct_pr_tmpl_get_fatal()`, and `ct_pr_tmpl_get_param()` return 0. Otherwise, they return a non-zero error value.

Upon successful completion, `ct_pr_tmpl_get_svc_fmri()` and `ct_pr_tmpl_get_svc_aux()` return the size required to store the value, which is the same value return by `strcpy`(3C) + 1. Insufficient buffer size can be checked by:

```
if (ct_pr_tmpl_get_svc_fmri(fd, fmri, size) > size)
        /* buffer is too small */
```

Otherwise, `ct_pr_tmpl_get_svc_fmri()` and `ct_pr_tmpl_get_svc_aux()` return -1 and set `errno` to indicate the error.

Errors The `ct_pr_tmpl_set_param()`, `ct_pr_tmpl_set_svc_fmri()`, `ct_pr_tmpl_set_svc_aux()`, `ct_pr_tmpl_get_svc_fmri()` and `ct_pr_tmpl_get_svc_aux()` functions will fail if:

`EINVAL` An invalid parameter was specified.

The `ct_pr_tmpl_set_fatal()` function will fail if:

`EINVAL` An invalid event was specified.

The `ct_pr_tmpl_set_transfer()` function will fail if:

`ESRCH` The ID specified by *ctid* does not correspond to a process contract.

`EACCES` The ID specified by *ctid* does not correspond to a process contract owned by the calling process.

`ENOTEMPTY` The ID specified by *ctid* does not correspond to an empty process contract.

The `ct_pr_tmpl_set_svc_fmri()` function will fail if:

`EPERM` The calling process does not have {`PRIV_CONTRACT_IDENTITY`} in its effective set.

Examples **EXAMPLE 1** Create and activate a process contract template.

The following example opens a new template, makes hardware errors and signals fatal events, makes hardware errors critical events, and activates the template. It then forks a process in the new contract using the requested terms.

```
#include <libcontract.h>
#include <fcntl.h>
#include <unistd.h>

...
int fd;

fd = open("/system/contract/process/template", O_RDWR);
```

EXAMPLE 1 Create and activate a process contract template. *(Continued)*

```
(void) ct_pr_tmpl_set_fatal(fd, CT_PR_EV_HWERR|CT_PR_EV_SIGNAL);
(void) ct_tmpl_set_critical(fd, CT_PR_EV_HWERR);
(void) ct_tmpl_activate(fd);
close(fd);

if (fork()) {
        /* parent - owns new process contract */
        ...
} else {
        /* child - in new process contract */
        ...
}
...
```

EXAMPLE 2 Clear the process contract template.

The following example opens the template file and requests that the active template be cleared.

```
#include <libcontract.h>
#include <fcntl.h>

...
int fd;

fd = open("/system/contract/process/template", O_RDWR);
(void) ct_tmpl_clear(fd);
close(fd);
...
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also

libcontract(3LIB), strcpy(3C), contract(4), process(4), attributes(5), lfcompile(5)

Name ct_status_read, ct_status_free, ct_status_get_id, ct_status_get_zoneid, ct_status_get_type, ct_status_get_state, ct_status_get_holder, ct_status_get_nevents, ct_status_get_ntime, ct_status_get_qtime, ct_status_get_nevid, ct_status_get_cookie, ct_status_get_informative, ct_status_get_critical – common contract status functions

Synopsis

```
cc [ flag... ] file... -D_LARGEFILE64_SOURCE  -lcontract  [ library... ]
#include <libcontract.h>

int ct_status_read(int fd, int detail, ct_stathdl_t *stathdlp);

void ct_status_free(ct_stathdl_t stathdl);

ctid_t ct_status_get_id(ct_stathdl_t stathdl);

zoneid_t ct_status_get_zoneid(ct_stathdl_t stathdl);

char *ct_status_get_type(ct_stathdl_t stathdl);

uint_t ct_status_get_state(ct_stathdl_t stathdl);

pid_t ct_status_get_holder(ct_stathdl_t stathdl);

int ct_status_get_nevents(ct_stathdl_t stathdl);

int ct_status_get_ntime(ct_stathdl_t stathdl);

int ct_status_get_qtime(ct_stathdl_t stathdl);

ctevid_t ct_status_get_nevid(ct_stathdl_t stathdl);

uint64_t ct_status_get_cookie(ct_stathdl_t stathdl);

ctevid_t ct_status_get_informative(ct_stathdl_t stathdl);

uint_t ct_status_get_critical(ct_stathdl_t stathdl);
```

Description These functions operate on contract status file descriptors obtained from the `contract(4)` file system and status object handles returned by `ct_status_read()`.

The `ct_status_read()` function reads the contract's status and initializes the status object handle pointed to by *stathdlp*. After a successful call to `ct_status_read()`, the caller is responsible for calling `ct_status_free()` on this status object handle when it has finished using it. Because the amount of information available for a contract might be large, the *detail* argument allows the caller to specify how much information `ct_status_read()` should obtain. A value of `CTD_COMMON` fetches only those data accessible by the functions on this manual page. `CTD_FIXED` fetches `CTD_COMMON` data as well as fixed-size contract type-specific data. `CTD_ALL` fetches `CTD_FIXED` data as well as variable lengthed data, such as arrays. See the manual pages for contract type-specific status accessor functions for information concerning which data are fetched by `CTD_FIXED` and `CTD_ALL`.

The `ct_status_free()` function frees any storage associated with the specified status object handle.

The remaining functions all return contract information obtained from a status object.

The ct_status_get_id() function returns the contract's ID.

The ct_status_get_zoneid() function returns the contract's creator's zone ID, or −1 if the creator's zone no longer exists.

The ct_status_get_type() function returns the contract's type. The string should be neither modified nor freed.

The ct_status_get_state() function returns the state of the contract. Valid state values are:

CTS_OWNED	a contract that is currently owned by a process
CTS_INHERITED	a contract that has been inherited by a regent process contract
CTS_ORPHAN	a contract that has no owner and has not been inherited
CTS_DEAD	a contract that is no longer in effect and will be automatically removed from the system as soon as the last reference to it is release (for example, an open status file descriptor)

The ct_status_get_holder() function returns the process ID of the contract's owner if the contract is in the CTS_OWNED state, or the ID of the regent process contract if the contract is in the CTS_INHERITED state.

The ct_status_get_nevents() function returns the number of unacknowledged critical events on the contract's event queue.

The ct_status_get_ntime() function returns the amount of time remaining (in seconds) before the ongoing exit negotiation times out, or -1 if there is no negotiation ongoing.

The ct_status_get_qtime() function returns the amount of time remaining (in seconds) in the quantum before the ongoing exit negotiation times out, or -1 if there is no negotiation ongoing.

The ct_status_get_nevid() function returns the event ID of the ongoing negotiation, or 0 if there are none.

The ct_status_get_cookie() function returns the cookie term of the contract.

The ct_status_get_critical() function is used to read the critical event set term. The value is a collection of bits as described in the contract type's manual page.

The ct_status_get_informative() function is used to read the informative event set term. The value is a collection of bits as described in the contract type's manual page.

Return Values Upon successful completion, ct_status_read() returns 0. Otherwise, it returns a non-zero error value.

Upon successful completion, `ct_status_get_id()`, `ct_status_get_type()`, `ct_status_get_holder()`, `ct_status_get_state()`, `ct_status_get_nevents()`, `ct_status_get_ntime()`, `ct_status_get_qtime()`, `ct_status_get_nevid()`, `ct_status_get_cookie()`, `ct_status_get_critical()`, and `ct_status_get_informative()` return the data described in the DESCRIPTION.

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

`EINVAL` The *detail* level specified is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libcontract(3LIB)`, `contract(4)`, `attributes(5)`, `lfcompile(5)`

Name ct_tmpl_activate, ct_tmpl_clear, ct_tmpl_create, ct_tmpl_set_cookie, ct_tmpl_set_critical, ct_tmpl_set_informative, ct_tmpl_get_cookie, ct_tmpl_get_critical, ct_tmpl_get_informative – common contract template functions

Synopsis

```
cc [ flag... ] file... -D_LARGEFILE64_SOURCE  -lcontract  [ library... ]
#include <libcontract.h>

int ct_tmpl_activate(int fd);

int ct_tmpl_clear(int fd);

int ct_tmpl_create(int fd, ctid_t *idp);

int ct_tmpl_set_cookie(int fd, uint64_t cookie);

int ct_tmpl_set_critical(int fd, uint_t events);

int ct_tmpl_set_informative(int fd, uint_t events);

int ct_tmpl_get_cookie(int fd, uint64_t *cookiep);

int ct_tmpl_get_critical(int fd, uint_t *eventsp);

int ct_tmpl_get_informative(int fd, uint_t *eventsp);
```

Description These functions operate on contract template file descriptors obtained from the `contract(4)` file system.

The `ct_tmpl_activate()` function makes the template referenced by the file descriptor *fd* the active template for the calling thread.

The `ct_tmpl_clear()` function clears calling thread's active template.

The `ct_tmpl_create()` function uses the template referenced by the file descriptor *fd* to create a new contract. If successful, the ID of the newly created contract is placed in **idp*.

The `ct_tmpl_set_cookie()` and `ct_tmpl_get_cookie()` functions write and read the cookie term of a contract template. The cookie term is ignored by the system, except to include its value in a resulting contract's status object. The default cookie term is 0.

The `ct_tmpl_set_critical()` and `ct_tmpl_get_critical()` functions write and read the critical event set term. The value is a collection of bits as described in the contract type's manual page.

The `ct_tmpl_set_informative()` and `ct_tmpl_get_informative()` functions write and read the informative event set term. The value is a collection of bits as described in the contract type's manual page.

Return Values Upon successful completion, `ct_tmpl_activate()`, `ct_tmpl_create()`, `ct_tmpl_set_cookie()`, `ct_tmpl_get_cookie()`, `ct_tmpl_set_critical()`, `ct_tmpl_get_critical()`, `ct_tmpl_set_informative()`, and `ct_tmpl_get_informative()` return 0. Otherwise, they return a non-zero error value.

Errors The ct_tmpl_create() function will fail if:

ERANGE The terms specified in the template were unsatisfied at the time of the call.

EAGAIN The *project.max-contracts* resource control would have been exceeded.

The ct_tmpl_set_critical() and ct_tmpl_set_informative() functions will fail if:

EINVAL An invalid event was specified.

The ct_tmpl_set_critical() function will fail if:

EPERM One of the specified events was disallowed given other contract terms (see contract(4)) and {PRIV_CONTRACT_EVENT} was not in the effective set for the calling process.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also libcontract(3LIB), contract(4), attributes(5), lfcompile(5)

Name dat_cno_create – create a CNO instance

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_cno_create (
    IN     DAT_IA_HANDLE             ia_handle,
    IN     DAT_OS_WAIT_PROXY_AGENT   agent,
    OUT    DAT_CNO_HANDLE            *cno_handle
    )
```

Parameters

ia_handle Handle for an instance of DAT IA.

agent An optional OS Wait Proxy Agent that is to be invoked whenever CNO is invoked. `DAT_OS_WAIT_PROXY_AGENT_NULL` indicates that there is no proxy agent

cno_handle Handle for the created instance of CNO.

Description The `dat_cno_create()` function creates a CNO instance. Upon creation, there are no Event Dispatchers feeding it.

The *agent* parameter specifies the proxy agent, which is OS-dependent and which is invoked when the CNO is triggered. After it is invoked, it is no longer associated with the CNO. The value of `DAT_OS_WAIT_PROXY_AGENT_NULL` specifies that no OS Wait Proxy Agent is associated with the created CNO.

Upon creation, the CNO is not associated with any EVDs, has no waiters and has, at most, one OS Wait Proxy Agent.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INSUFFICIENT_RESOURCES` The operation failed due to resource limitations.

`DAT_INVALID_HANDLE` The *ia_handle* parameter is invalid.

`DAT_INVALID_PARAMETER` One of the parameters was invalid, out of range, or a combination of parameters was invalid, or the *agent* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	uDAPL, 1.1, 1.2

See Also libdat(3LIB), attributes(5)

Name dat_cno_free – destroy an instance of the CNO

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_cno_free (
    IN    DAT_CNO_HANDLE    cno_handle
    )
```

Parameters

cno_handle Handle for an instance of the CNO

Description

The `dat_cno_free()` function destroys a specified instance of the CNO.

A CNO cannot be deleted while it is referenced by an Event Dispatcher or while a thread is blocked on it.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The `cno_handle()` parameter is invalid.

`DAT_INVALID_STATE` Parameter in an invalid state. CNO is in use by an EVD instance or there is a thread blocked on it.

Usage

If there is a thread blocked in `dat_cno_wait(3DAT)`, the Consumer can do the following steps to unblock the waiter:

- Create a temporary EVD that accepts software events. It can be created in advance.
- For a CNO with the waiter, attach that EVD to the CNO and post the software event on the EVD.
- This unblocks the CNO.
- Repeat for other CNOs that have blocked waiters.
- Destroy the temporary EVD after all CNOs are destroyed and the EVD is no longer needed.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.1, 1.2

See Also

`dat_cno_wait(3DAT)`, `libdat(3LIB)`, `attributes(5)`

Name dat_cno_modify_agent – modify the OS Wait Proxy Agent

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_cno_modify_agent (
    IN    DAT_CNO_HANDLE            cno_handle,
    IN    DAT_OS_WAIT_PROXY_AGENT   agent
    )
```

Parameters

cno_handle Handle for an instance of CNO

agent Pointer to an optional OS Wait Proxy Agent to invoke whenever CNO is invoked. `DAT_OS_WAIT_PROXY_AGENT_NULL` indicates that there is no proxy agent.

Description The `dat_cno_modify_agent()` function modifies the OS Wait Proxy Agent associated with a CNO. If non-null, any trigger received by the CNO is also passed to the OS Wait Proxy Agent. This is in addition to any local delivery through the CNO. The Consumer can pass the value of `DAT_OS_WAIT_PROXY_AGENT_NULL` to disassociate the current Proxy agent from the CNO

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *cno_handle* parameter is invalid.

`DAT_INVALID_PARAMETER` One of the parameters was invalid, out of range, or a combination of parameters was invalid, or the *agent* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.1, 1.2

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_cno_query – provide the Consumer parameters of the CNO

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_cno_query (
    IN     DAT_CNO_HANDLE          cno_handle,
    IN     DAT_CNO_PARAM_MASK      cno_param_mask,
    OUT    DAT_CNO_PARAM           *cno_param
    )
```

Parameters

cno_handle Handle for the created instance of the Consumer Notification Object

cno_param_mask Mask for CNO parameters

cno_param Pointer to a Consumer-allocated structure that the Provider fills with CNO parameters

Description The `dat_cno_query()` function provides the Consumer parameters of the CNO. The Consumer passes in a pointer to the Consumer-allocated structures for CNO parameters that the Provider fills.

The *cno_param_mask* parameter allows Consumers to specify which parameters to query. The Provider returns values for *cno_param_mask* requested parameters. The Provider can return values for any other parameters.

A value of `DAT_OS_WAIT_PROXY_AGENT_NULL` in *cno_param* indicates that there are no Proxy Agent associated with the CNO.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_PARAMETER` The *cno_param_mask* parameter is invalid.

`DAT_INVALID_HANDLE` The *cno_handle* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.1, 1.2

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_cno_wait – wait for notification events

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_cno_wait (
    IN     DAT_CNO_HANDLE     cno_handle,
    IN     DAT_TIMEOUT        timeout,
    OUT    DAT_EVD_HANDLE     *evd_handle
    )
```

Parameters

cno_handle Handle for an instance of CNO

timeout The duration to wait for a notification. The value `DAT_TIMEOUT_INFINITE` can be used to wait indefinitely.

evd_handle Handle for an instance of EVD

Description The `dat_cno_wait()` function allows the Consumer to wait for notification events from a set of Event Dispatchers all from the same Interface Adapter. The Consumer blocks until notified or the timeout period expires.

An Event Dispatcher that is disabled or in the "Waited" state does not deliver notifications. A uDAPL Consumer waiting directly upon an Event Dispatcher preempts the CNO.

The consumer can optionally specify a timeout, after which it is unblocked even if there are no notification events. On a timeout, *evd_handle* is explicitly set to a null handle.

The returned *evd_handle* is only a hint. Another Consumer can reap the Event before this Consumer can get around to checking the Event Dispatcher. Additionally, other Event Dispatchers feeding this CNO might have been notified. The Consumer is responsible for ensuring that all EVDs feeding this CNO are polled regardless of whether they are identified as the immediate cause of the CNO unblocking.

All the waiters on the CNO, including the OS Wait Proxy Agent if it is associated with the CNO, are unblocked with the NULL handle returns for an unblocking EVD *evd_handle* when the IA instance is destroyed or when all EVDs the CNO is associated with are freed.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *cno_handle* parameter is invalid.

`DAT_QUEUE_EMPTY` The operation timed out without a notification.

`DAT_INVALID_PARAMETER` One of the parameters was invalid or out of range, a combination of parameters was invalid, or the *timeout* parameter is invalid.

`DAT_INTERRUPTED_CALL` The operation was interrupted by a signal.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	uDAPL, 1.1, 1.2

See Also libdat(3LIB), attributes(5)

Name dat_cr_accept – establishes a Connection between the active remote side requesting Endpoint and the passive side local Endpoint

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_cr_accept (
    IN    DAT_CR_HANDLE    cr_handle,
    IN    DAT_EP_HANDLE    ep_handle,
    IN    DAT_COUNT        private_data_size,
    IN const DAT_PVOID     private_data
    )
```

Parameters

cr_handle — Handle to an instance of a Connection Request that the Consumer is accepting.

ep_handle — Handle for an instance of a local Endpoint that the Consumer is accepting the Connection Request on. If the local Endpoint is specified by the Connection Request, the *ep_handle* shall be `DAT_HANDLE_NULL`.

private_data_size — Size of the *private_data*, which must be nonnegative.

private_data — Pointer to the private data that should be provided to the remote Consumer when the Connection is established. If *private_data_size* is zero, then *private_data* can be `NULL`.

Description The `dat_cr_accept()` function establishes a Connection between the active remote side requesting Endpoint and the passive side local Endpoint. The local Endpoint is either specified explicitly by *ep_handle* or implicitly by a Connection Request. In the second case, *ep_handle* is `DAT_HANDLE_NULL`.

Consumers can specify private data that is provided to the remote side upon Connection establishment.

If the provided local Endpoint does not satisfy the requested Connection Request, the operation fails without any effect on the local Endpoint, Pending Connection Request, private data, or remote Endpoint.

The operation is asynchronous. The successful completion of the operation is reported through a Connection Event of type `DAT_CONNECTION_EVENT_ESTABLISHED` on the *connect_evd* of the local Endpoint.

If the Provider cannot complete the Connection establishment, the connection is not established and the Consumer is notified through a Connection Event of type `DAT_CONNECTION_EVENT_ACCEPT_COMPLETION_ERROR` on the *connect_evd* of the local Endpoint. It can be caused by the active side timeout expiration, transport error, or any other reason. If Connection is not established, Endpoint transitions into Disconnected state and all posted Recv DTOs are flushed to its *recv_evd_handle*.

This operation, if successful, also destroys the Connection Request instance. Use of the handle of the destroyed *cr_handle* in any consequent operation fails.

Return Values

DAT_SUCCESS — The operation was successful.

DAT_INVALID_HANDLE — The *cr_handle* or *ep_handle* parameter is invalid.

DAT_INVALID_PARAMETER — The *private_data_size* or *private_data* parameter is invalid, out of range, or a combination of parameters was invalid

Usage

Consumers should be aware that Connection establishment might fail in the following cases: If the accepting Endpoint has an outstanding RDMA Read outgoing attribute larger than the requesting remote Endpoint or outstanding RDMA Read incoming attribute, or if the outstanding RDMA Read incoming attribute is smaller than the requesting remote Endpoint or outstanding RDMA Read outgoing attribute.

Consumers should set the accepting Endpoint RDMA Reads as the target (incoming) to a number larger than or equal to the remote Endpoint RDMA Read outstanding as the originator (outgoing), and the accepting Endpoint RDMA Reads as the originator to a number smaller than or equal to the remote Endpoint RDMA Read outstanding as the target. DAT API does not define a protocol on how remote peers exchange Endpoint attributes. The exchange of outstanding RDMA Read incoming and outgoing attributes of EPs is left to the Consumer ULP. Consumer can use Private Data for it.

If the Consumer does not care about posting RDMA Read operations or remote RDMA Read operations on the connection, it can set the two outstanding RDMA Read attribute values to 0.

If the Consumer does not set the two outstanding RDMA Read attributes of the Endpoint, the Provider is free to pick up any value for default. The Provider can change these default values during connection setup.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.1, 1.2

See Also

libdat(3LIB), attributes(5)

Name dat_cr_handoff – hand off the Connection Request to another Service Point

Synopsis
```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_cr_handoff (
    IN    DAT_CR_HANDLE    cr_handle,
    IN    DAT_CONN_QUAL    handoff
    )
```

Parameters

cr_handle Handle to an instance of a Connection Request that the Consumer is handing off.

handoff Indicator of another Connection Qualifier on the same IA to which this Connection Request should be handed off.

Description The `dat_cr_handoff()` function hands off the Connection Request to another Service Point specified by the Connection Qualifier *handoff*.

The operation is synchronous. This operation also destroys the Connection Request instance. Use of the handle of the destroyed Connection Request in any consequent operation fails.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *cr_handle* parameter is invalid.

`DAT_INVALID_PARAMETER` The *handoff* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.1, 1.2

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_cr_query – provide parameters of the Connection Request

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_cr_query (
    IN     DAT_CR_HANDLE        cr_handle,
    IN     DAT_CR_PARAM_MASK    cr_param_mask,
    OUT    DAT_CR_PARAM         *cr_param
    )
```

Parameters

cr_handle Handle for an instance of a Connection Request.

cr_param_mask Mask for Connection Request parameters.

cr_param Pointer to a Consumer-allocated structure that the Provider fills for Consumer-requested parameters.

Description The `dat_cr_query()` function provides to the Consumer parameters of the Connection Request. The Consumer passes in a pointer to the Consumer-allocated structures for Connection Request parameters that the Provider fills.

The *cr_param_mask* parameter allows Consumers to specify which parameters to query. The Provider returns values for *cr_param_mask* requested parameters. The Provider can return values for any other parameters.

Return Values

`DAT_SUCCESS` The operation was successful

`DAT_INVALID_HANDLE` The *cr_handle* handle is invalid.

`DAT_INVALID_PARAMETER` The *cr_param_mask* parameter is invalid.

Usage The Consumer uses `dat_cr_query()` to get information about requesting a remote Endpoint as well as a local Endpoint if it was allocated by the Provider for the arrived Connection Request. The local Endpoint is created if the Consumer used PSP with `DAT_PSP_PROVIDER` as the value for *psp_flags*. For the remote Endpoint, `dat_cr_query()` provides *remote_ia_address* and *remote_port_qual*. It also provides remote *peer private_data* and its size.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	uDAPL, 1.1, 1.2

See Also libdat(3LIB), attributes(5)

Name dat_cr_reject – reject a Connection Request from the Active remote side requesting Endpoint

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_cr_reject (
    IN    DAT_CR_HANDLE    cr_handle
    )
```

Parameters *cr_handle* Handle to an instance of a Connection Request that the Consumer is rejecting.

Description The `dat_cr_reject()` function rejects a Connection Request from the Active remote side requesting Endpoint. If the Provider passed a local Endpoint into a Consumer for the Public Service Point-created Connection Request, that Endpoint reverts to Provider Control. The behavior of an operation on that Endpoint is undefined. The local Endpoint that the Consumer provided for Reserved Service Point reverts to Consumer control, and the Consumer is free to use in any way it wants.

The operation is synchronous. This operation also destroys the Connection Request instance. Use of the handle of the destroyed Connection Request in any consequent operation fails.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *cr_handle* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.1, 1.2

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_ep_connect – establish a connection between the local Endpoint and a remote Endpoint

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ep_connect (
    IN    DAT_EP_HANDLE         ep_handle,
    IN    DAT_IA_ADDRESS_PTR    remote_ia_address,
    IN    DAT_CONN_QUAL         remote_conn_qual,
    IN    DAT_TIMEOUT           timeout,
    IN    DAT_COUNT             private_data_size,
    IN const  DAT_PVOID         private_data,
    IN    DAT_QOS               qos,
    IN    DAT_CONNECT_FLAGS     connect_flags
    )
```

Parameters

ep_handle — Handle for an instance of an Endpoint.

remote_ia_address — The Address of the remote IA to which an Endpoint is requesting a connection.

remote_conn_qual — Connection Qualifier of the remote IA from which an Endpoint requests a connection.

timeout — Duration of time, in microseconds, that a Consumer waits for Connection establishment. The value of `DAT_TIMEOUT_INFINITE` represents no timeout, indefinite wait. Values must be positive.

private_data_size — Size of the *private_data*. Must be nonnegative.

private_data — Pointer to the private data that should be provided to the remote Consumer as part of the Connection Request. If *private_data_size* is zero, then *private_data* can be `NULL`.

qos — Requested quality of service of the connection.

connect_flags — Flags for the requested connection. If the least significant bit of `DAT_MULTIPATH_FLAG` is 0, the Consumer does not request multipathing. If the least significant bit of `DAT__MULTIPATH_FLAG` is 1, the Consumer requests multipathing. The default value is `DAT_CONNECT_DEFAULT_FLAG`, which is 0.

Description The `dat_ep_connect()` function requests that a connection be established between the local Endpoint and a remote Endpoint. This operation is used by the active/client side Consumer of the Connection establishment model. The remote Endpoint is identified by Remote IA and Remote Connection Qualifier.

As part of the successful completion of this operation, the local Endpoint is bound to a Port Qualifier of the local IA. The Port Qualifier is passed to the remote side of the requested connection and is available to the remote Consumer in the Connection Request of the DAT_CONNECTION_REQUEST_EVENT.

The Consumer-provided *private_data* is passed to the remote side and is provided to the remote Consumer in the Connection Request. Consumers can encapsulate any local Endpoint attributes that remote Consumers need to know as part of an upper-level protocol. Providers can also provide a Provider on the remote side any local Endpoint attributes and Transport-specific information needed for Connection establishment by the Transport.

Upon successful completion of this operation, the local Endpoint is transferred into DAT_EP_STATE_ACTIVE_CONNECTION_PENDING.

Consumers can request a specific value of *qos*. The Provider specifies which quality of service it supports in documentation and in the Provider attributes. If the local Provider or Transport does not support the requested *qos*, the operation fails and DAT_MODEL_NOT_SUPPORTED is returned synchronously. If the remote Provider does not support the requested *qos*, the local Endpoint is automatically transitioned into the DAT_EP_STATE_DISCONNECTED state, the connection is not established, and the event returned on the *connect_evd_handle* is DAT_CONNECTION_EVENT_NON_PEER_REJECTED. The same DAT_CONNECTION_EVENT_NON_PEER_REJECTED event is returned if the connection cannot be established for all reasons of not establishing the connection, except timeout, remote host not reachable, and remote peer reject. For example, remote Consumer is not listening on the requested Connection Qualifier, Backlog of the requested Service Point is full, and Transport errors. In this case, the local Endpoint is automatically transitioned into DAT_EP_STATE_DISCONNECTED state.

The acceptance of the requested connection by the remote Consumer is reported to the local Consumer through a DAT_CONNECTION_EVENT_ESTABLISHED event on the *connect_evd_handle* of the local Endpoint and the local Endpoint is automatically transitioned into a DAT_EP_STATE_CONNECTED state.

The rejection of the connection by the remote Consumer is reported to the local Consumer through a DAT_CONNECTION_EVENT_PEER_REJECTED event on the *connect_evd_handle* of the local Endpoint and the local Endpoint is automatically transitioned into a DAT_EP_STATE_DISCONNECTED state.

When the Provider cannot reach the remote host or the remote host does not respond within the Consumer requested Timeout, a DAT_CONNECTION_EVENT_UNREACHABLE event is generated on the *connect_evd_handle* of the Endpoint. The Endpoint transitions into a DAT_EP_STATE_DISCONNECTED state.

If the Provider can locally determine that the *remote_ia_address* is invalid, or that the remote_ia_address cannot be converted to a Transport-specific address, the operation can fail synchronously with a `DAT_INVALID_ADDRESS` return.

The local Endpoint is automatically transitioned into a `DAT_EP_STATE_CONNECTED` state when a Connection Request accepted by the remote Consumer and the Provider completes the Transport-specific Connection establishment. The local Consumer is notified of the established connection through a `DAT_CONNECTION_EVENT_ESTABLISHED` event on the *connect_evd_handle* of the local Endpoint.

When the *timeout* expired prior to completion of the Connection establishment, the local Endpoint is automatically transitioned into a `DAT_EP_STATE_DISCONNECTED` state and the local Consumer through a `DAT_CONNECTION_EVENT_TIMED_OUT` event on the *connect_evd_handle* of the local Endpoint.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INSUFFICIENT_RESOURCES`	The operation failed due to resource limitations.
`DAT_INVALID_PARAMETER`	Invalid parameter.
`DAT_INVALID_ADDRESS`	Invalid address.
`DAT_INVALID_HANDLE`	Invalid DAT handle; Invalid Endpoint handle.
`DAT_INVALID_STATE`	Parameter in an invalid state. Endpoint was not in `DAT_EP_STATE_UNCONNECTED` state.
`DAT_MODEL_NOT_SUPPORTED`	The requested Model was not supported by the Provider. For example, the requested qos was not supported by the local Provider.

Usage

It is up to the Consumer to negotiate outstanding RDMA Read incoming and outgoing with a remote peer. The outstanding RDMA Read outgoing attribute should be smaller than the remote Endpoint outstanding RDMA Read incoming attribute. If this is not the case, Connection establishment might fail.

DAT API does not define a protocol on how remote peers exchange Endpoint attributes. The exchange of outstanding RDMA Read incoming and outgoing attributes of EPs is left to the Consumer ULP. The Consumer can use Private Data for it.

If the Consumer does not care about posting RDMA Read operations or remote RDMA Read operations on the connection, it can set the two outstanding RDMA Read attribute values to 0.

If the Consumer does not set the two outstanding RDMA Read attributes of the Endpoint, the Provider is free to pick up any value for default. The Provider is allowed to change these default values during connection setup.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.1, 1.2

See Also libdat(3LIB), attributes(5)

Name dat_ep_create – create an instance of an Endpoint

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ep_create (
    IN     DAT_IA_HANDLE     ia_handle,
    IN     DAT_PZ_HANDLE     pz_handle,
    IN     DAT_EVD_HANDLE    recv_evd_handle,
    IN     DAT_EVD_HANDLE    request_evd_handle,
    IN     DAT_EVD_HANDLE    connect_evd_handle,
    IN     DAT_EP_ATTR       *ep_attributes,
    OUT    DAT_EP_HANDLE     *ep_handle
    )
```

Parameters

ia_handle Handle for an open instance of the IA to which the created Endpoint belongs.

pz_handle Handle for an instance of the Protection Zone.

recv_evd_handle Handle for the Event Dispatcher where events for completions of incoming (receive) DTOs are reported. DAT_HANDLE_NULL specifies that the Consumer is not interested in events for completions of receives.

request_evd_handle Handle for the Event Dispatcher where events for completions of outgoing (Send, RDMA Write, RDMA Read, and RMR Bind) DTOs are reported. DAT_HANDLE_NULL specifies that the Consumer is not interested in events for completions of requests.

connect_evd_handle Handle for the Event Dispatcher where Connection events are reported. DAT_HANDLE_NULL specifies that the Consumer is not interested in connection events for now.

ep_attributes Pointer to a structure that contains Consumer-requested Endpoint attributes. Can be NULL.

ep_handle Handle for the created instance of an Endpoint.

Description The dat_ep_create() function creates an instance of an Endpoint that is provided to the Consumer as *ep_handle*. The value of *ep_handle* is not defined if the DAT_RETURN is not DAT_SUCCESS.

The Endpoint is created in the Unconnected state.

Protection Zone *pz_handle* allows Consumers to control what local memory the Endpoint can access for DTOs and what memory remote RDMA operations can access over the connection of a created Endpoint. Only memory referred to by LMRs and RMRs that match the Endpoint Protection Zone can be accessed by the Endpoint.

The *recv_evd_handle* and *request_evd_handle* parameters are Event Dispatcher instances where the Consumer collects completion notifications of DTOs. Completions of Receive DTOs are reported in *recv_evd_handle* Event Dispatcher, and completions of Send, RDMA Read, and RDMA Write DTOs are reported in *request_evd_handle* Event Dispatcher. All completion notifications of RMR bindings are reported to a Consumer in *request_evd_handle* Event Dispatcher.

All Connection events for the connected Endpoint are reported to the Consumer through *connect_evd_handle* Event Dispatcher.

The *ep_attributes* parameter specifies the initial attributes of the created Endpoint. If the Consumer specifies `NULL`, the Provider fills it with its default Endpoint attributes. The Consumer might not be able to do any posts to the Endpoint or use the Endpoint in connection establishment until certain Endpoint attributes are set. Maximum Message Size and Maximum Recv DTOs are examples of such attributes.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INSUFFICIENT_RESOURCES`	The operation failed due to resource limitations.
`DAT_INVALID_HANDLE`	Invalid DAT handle.
`DAT_INVALID_PARAMETER`	Invalid parameter. One of the requested EP parameters or attributes was invalid or a combination of attributes or parameters is invalid.
`DAT_MODEL_NOT_SUPPORTED`	The requested Provider Model was not supported.

Usage The Consumer creates an Endpoint prior to the establishment of a connection. The created Endpoint is in `DAT_EP_STATE_UNCONNECTED`. Consumers can do the following:

1. Request a connection on the Endpoint through `dat_ep_connect`(3DAT) or `dat_ep_dup_connect`(3DAT) for the active side of the connection model.
2. Associate the Endpoint with the Pending Connection Request that does not have an associated local Endpoint for accepting the Pending Connection Request for the passive/server side of the connection model.
3. Create a Reserved Service Point with the Endpoint for the passive/server side of the connection model. Upon arrival of a Connection Request on the Service Point, the Consumer accepts the Pending Connection Request that has the Endpoint associated with it

The Consumer cannot specify a *request_evd_handle* (*recv_evd_handle*) with Request Completion Flags (Recv Completion Flags) that do not match the other Endpoint Completion Flags for the DTO/RMR completion streams that use the same EVD. If *request_evd_handle* (*recv_evd_handle*) is used for an EVD that is fed by any event stream other than DTO or RMR completion event streams, only `DAT_COMPLETION_THRESHOLD` is valid for Request/Recv Completion Flags for the Endpoint completion streams that use that EVD. If *request_evd_handle* (*recv_evd_handle*) is used for request (recv) completions of an Endpoint whose associated Request (Recv) Completion Flag attribute is `DAT_COMPLETION_UNSIGNALLED_FLAG`, the Request Completion Flags and Recv Completion Flags for all Endpoint completion streams that use the EVD must specify the same. Analogously, if *recv_evd_handle* is used for recv completions of an Endpoint whose associated Recv Completion Flags attribute is `DAT_COMPLETION_SOLICITED_WAIT`, the Recv Completion Flags for all Endpoint Recv completion streams that use the same EVD must specify the same Recv Completion Flags attribute value and the EVD cannot be used for any other event stream types.

If EP is created with `NULL` attributes, Provider can fill them with its own default values. The Consumer should not rely on the Provider-filled attribute defaults, especially for portable applications. The Consumer cannot do any operations on the created Endpoint except for `dat_ep_query`(3DAT), `dat_ep_get_status`(3DAT), `dat_ep_modify`(3DAT), and `dat_ep_free`(3DAT), depending on the values that the Provider picks.

The Provider is encouraged to pick up reasonable defaults because unreasonable values might restrict Consumers to the `dat_ep_query()`, `dat_ep_get_status()`, `dat_ep_modify()`, and `dat_ep_free()` operations. The Consumer should check what values the Provider picked up for the attributes. It is especially important to make sure that the number of posted operations is not too large to avoid EVD overflow. Depending on the values picked up by the Provider, the Consumer might not be able to do any RDMA operations; it might only be able to send or receive messages of very small sizes, or it might not be able to have more than one segment in a buffer. Before doing any operations, except the ones listed above, the Consumer can configure the Endpoint using `dat_ep_modify()` to the attributes they want.

One reason the Consumer might still want to create an Endpoint with Null attributes is for the Passive side of the connection establishment, where the Consumer sets up Endpoint attributes based on the connection request of the remote side.

Consumers might want to create Endpoints with `NULL` attributes if Endpoint properties are negotiated as part the Consumer connection establishment protocol.

Consumers that create Endpoints with Provider default attributes should always verify that the Provider default attributes meet their application's requirements with regard to the number of request/receive DTOs that can be posted, maximum message sizes, maximum request/receive IOV sizes, and maximum RDMA sizes.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Standard	uDAPL, 1.1, 1.2

See Also dat_ep_connect(3DAT), dat_ep_dup_connect(3DAT), dat_ep_free(3DAT), dat_ep_get_status(3DAT), dat_ep_modify(3DAT), dat_ep_query(3DAT), libdat(3LIB), attributes(5)

Name dat_ep_create_with_srq – create an instance of End Point with Shared Receive Queue

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ep_create_with_srq (
        IN      DAT_IA_HANDLE       ia_handle,
        IN      DAT_PZ_HANDLE       pz_handle,
        IN      DAT_EVD_HANDLE      recv_evd_handle,
        IN      DAT_EVD_HANDLE      request_evd_handle,
        IN      DAT_EVD_HANDLE      connect_evd_handle,
        IN      DAT_SRQ_HANDLE      srq_handle,
        IN      DAT_EP_ATTR         *ep_attributes,
        OUT     DAT_EP_HANDLE       *ep_handle
    )
```

Parameters

ia_handle Handle for an open instance of the IA to which the created Endpoint belongs.

pz_handle Handle for an instance of the Protection Zone.

recv_evd_handle Handle for the Event Dispatcher where events for completions of incoming (receive) DTOs are reported. `DAT_HANDLE_NULL` specifies that the Consumer is not interested in events for completions of receives.

request_evd_handle Handle for the Event Dispatcher where events for completions of outgoing (Send, RDMA Write, RDMA Read, and RMR Bind) DTOs are reported. `DAT_HANDLE_NULL` specifies that the Consumer is not interested in events for completions of requests.

connect_evd_handle Handle for the Event Dispatcher where Connection events are reported. `DAT_HANDLE_NULL` specifies that the Consumer is not interested in connection events for now.

srq_handle Handle for an instance of the Shared Receive Queue.

ep_attributes Pointer to a structure that contains Consumer-requested Endpoint attributes. Cannot be `NULL`.

ep_handle Handle for the created instance of an Endpoint.

Description The `dat_ep_create_with_srq()` function creates an instance of an Endpoint that is using SRQ for Recv buffers is provided to the Consumer as *ep_handle*. The value of *ep_handle* is not defined if the `DAT_RETURN` is not `DAT_SUCCESS`.

The Endpoint is created in the Unconnected state.

Protection Zone *pz_handle* allows Consumers to control what local memory the Endpoint can access for DTOs except Recv and what memory remote RDMA operations can access over the connection of a created Endpoint. Only memory referred to by LMRs and RMRs that match the Endpoint Protection Zone can be accessed by the Endpoint. The Recv DTO buffers PZ must match the SRQ PZ. The SRQ PZ might or might not be the same as the EP one. Check Provider attribute for the support of different PZs between SRQ and its EPs.

The *recv_evd_handle* and *request_evd_handle* arguments are Event Dispatcher instances where the Consumer collects completion notifications of DTOs. Completions of Receive DTOs are reported in *recv_evd_handle* Event Dispatcher, and completions of Send, RDMA Read, and RDMA Write DTOs are reported in *request_evd_handle* Event Dispatcher. All completion notifications of RMR bindings are reported to a Consumer in *request_evd_handle* Event Dispatcher.

All Connection events for the connected Endpoint are reported to the Consumer through *connect_evd_handle* Event Dispatcher.

Shared Receive Queue *srq_handle* specifies where the EP will dequeue Recv DTO buffers.

The created EP can be reset. The relationship between SRQ and EP is not effected by `dat_ep_reset`(3DAT).

SRQ can not be disassociated or replaced from created EP. The only way to disassociate SRQ from EP is to destroy EP.

Receive buffers cannot be posted to the created Endpoint. Receive buffers must be posted to the SRQ to be used for the created Endpoint.

The ep_attributes parameter specifies the initial attributes of the created Endpoint. Consumer can not specify `NULL` for *ep_attributes* but can specify values only for the parameters needed and default for the rest.

For *max_request_dtos* and *max_request_iov*, the created Endpoint will have at least the Consumer requested values but might have larger values. Consumer can query the created Endpoint to find out the actual values for these attributes. Created Endpoint has the exact Consumer requested values for *max_recv_dtos*, *max_message_size*, *max_rdma_size*, *max_ rdma_read_in*, and *max_rdma_read_out*. For all other attributes, except *max_recv_iov* that is ignored, the created Endpoint has the exact values requested by Consumer. If Provider cannot satisfy the Consumer requested attribute values the operation fails.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INSUFFICIENT_RESOURCES`	The operation failed due to resource limitations.
`DAT_INVALID_HANDLE`	Invalid DAT handle.

`DAT_INVALID_PARAMETER`	Invalid parameter. One of the requested EP parameters or attributes was invalid or a combination of attributes or parameters is invalid. For example, *pz_handle* specified does not match the one for SRQ or the requested maximum RDMA Read IOV exceeds IA capabilities..
`DAT_MODEL_NOT_SUPPORTED`	The requested Provider Model was not supported.

Usage The Consumer creates an Endpoint prior to the establishment of a connection. The created Endpoint is in `DAT_EP_STATE_UNCONNECTED`. Consumers can do the following:

1. Request a connection on the Endpoint through `dat_ep_connect`(3DAT) or `dat_ep_dup_connect`(3DAT) for the active side of the connection model.
2. Associate the Endpoint with the Pending Connection Request that does not have an associated local Endpoint for accepting the Pending Connection Request for the passive/server side of the con-nection model.
3. Create a Reserved Service Point with the Endpoint for the passive/server side of the connection model. Upon arrival of a Connection Request on the Service Point, the Consumer accepts the Pending Connection Request that has the Endpoint associated with it.

The Consumer cannot specify a *request_evd_handle* (*recv_evd_handle*) with Request Completion Flags (Recv Completion Flags) that do not match the other Endpoint Completion Flags for the DTO/RMR completion streams that use the same EVD. If *request_evd_handle* (*recv_evd_ handle*) is used for request (recv) completions of an Endpoint whose associated Request (Recv) Completion Flag attribute is `DAT_COMPLETION_UNSIGNALLED_FLAG`, the Request Completion Flags and Recv Completion Flags for all Endpoint completion streams that use the EVD must specify the same. By definition, completions of all Recv DTO posted to SRQ complete with Signal. Analogously, if *recv_evd_handle* is used for recv completions of an Endpoint whose associated Recv Completion Flag attribute is `DAT_COMPLETION_SOLICITED_WAIT`, the Recv Completion Flags for all Endpoint Recv completion streams that use the same EVD must specify the same Recv Completion Flags attribute value and the EVD cannot be used for any other event stream types. If *recv_evd_handle* is used for Recv completions of an Endpoint that uses SRQ and whose Recv Completion Flag attribute is `DAT_COMPLETION_EVD_THRESHOLD` then all Endpoint DTO completion streams (request and/or recv completion streams) that use that *recv_evd_handle* must specify `DAT_COMPLETION_EVD_THRESHOLD`. Other event stream types can also use the same EVD.

Consumers might want to use `DAT_COMPLETION_UNSIGNALLED_FLAG` for Request and/or Recv completions when they control locally with posted DTO/RMR completion flag (not needed for Recv posted to SRQ) whether posted DTO/RMR completes with Signal or not. Consumers might want to use `DAT_COMPLETION_SOLICITED_WAIT` for Recv completions when the remote sender side control whether posted Recv competes with Signal or not or not. uDAPL

Consumers might want to use DAT_COMPLETION_EVD_THRESHOLD for Request and/or Recv completions when they control waiter unblocking with the *threshold* parameter of the dat_evd_wait(3DAT).

Some Providers might restrict whether multiple EPs that share a SRQ can have different Protection Zones. Check the *srq_ep_pz_difference_support* Provider attribute for it.

Consumers might want to have a different PZ between EP and SRQ. This allows incoming RDMA operations to be specific to this EP PZ and not the same for all EPs that share SRQ. This is critical for servers that supports multiple independent clients.

The Provider is strongly encouraged to create an EP that is ready to be connected. Any effects of previous connections or connection establishment attempts on the underlying Transport-specific Endpoint to which the DAT Endpoint is mapped to should be hidden from the Consumer. The methods described below are examples:

- The Provider does not create an underlying Transport Endpoint until the Consumer is connecting the Endpoint or accepting a connection request on it. This allows the Provider to accumulate Consumer requests for attribute settings even for attributes that the underlying transport does not allow to change after the Transport Endpoint is created.
- The Provider creates the underlying Transport Endpoint or chooses one from a pool of Provider-controlled Transport Endpoints when the Consumer creates the Endpoint. The Provider chooses the Transport Endpoint that is free from any underlying internal attributes that might prevent the Endpoint from being connected. For IB and IP, that means that the Endpoint is not in the TimeWait state. Changing of some of the Endpoint attributes becomes hard and might potentially require mapping the Endpoint to another underlying Transport Endpoint that might not be feasible for all transports.
- The Provider allocates a Transport-specific Endpoint without worrying about impact on it from previous connections or connection establishment attempts. Hide the Transport-specific TimeWait state or CM timeout of the underlying transport Endpoint within dat_ep_connect(3DAT), dat_ep_dup_connect(3DAT), or dat_cr_accept(3DAT). On the Active side of the connection establishment, if the remnants of a previous connection for Transport-specific Endpoint can be hidden within the Timeout parameter, do so. If not, generating DAT_CONNECTION_EVENT_NON_PEER_REJECTED is an option. For the Passive side, generating a DAT_CONNECTION_COMPLETION_ERROR event locally, while sending a non-peer-reject message to the active side, is a way of handling it.

Any transitions of an Endpoint into an Unconnected state can be handled similarly. One transition from a Disconnected to an Unconnected state is a special case.

For dat_ep_reset(3DAT), the Provider can hide any remnants of the previous connection or failed connection establishment in the operation itself. Because the operation is synchronous, the Provider can block in it until the TimeWait state effect of the previous connection or connection setup is expired, or until the Connection Manager timeout of an unsuccessful connection establishment attempt is expired. Alternatively, the Provider can create a new Endpoint for the Consumer that uses the same handle.

DAT Providers are required not to change any Consumer-specified Endpoint attributes during connection establishment. If the Consumer does not specify an attribute, the Provider can set it to its own default. Some EP attributes, like outstanding RDMA Read incoming or outgoing, if not set up by the Consumer, can be changed by Providers to establish connection. It is recommended that the Provider pick the default for outstanding RDMA Read attributes as 0 if the Consumer has not specified them. This ensures that connection establishment does not fail due to insufficient outstanding RDMA Read resources, which is a requirement for the Provider.

The Provider is not required to check for a mismatch between the maximum RDMA Read IOV and maximum RDMA Read outgoing attributes, but is allowed to do so. In the later case it is allowed to return `DAT_INVALID_ PARAMETER` when a mismatch is detected. Provider must allocate resources to satisfy the combination of these two EP attributes for local RDMA Read DTOs.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	uDAPL, 1.2

See Also `dat_ep_create(3DAT)`, `dat_srq_create(3DAT)`, `dat_srq_free(3DAT)`, `dat_srq_query(3DAT)`, `libdat(3LIB)`, `attributes(5)`

Name dat_ep_disconnect – terminate a connection or a connection establishment

Synopsis
```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ep_disconnect (
    IN    DAT_EP_HANDLE      ep_handle,
    IN    DAT_CLOSE_FLAGS    disconnect_flags
    )
```

Parameters

ep_handle Handle for an instance of Endpoint.

disconnect_flags Flags for disconnect. Flag values are as follows:

DAT_CLOSE_ABRUPT_FLAG	Abrupt close. This is the default value.
DAT_CLOSE_GRACEFUL_FLAG	Graceful close.

Description The `dat_ep_disconnect()` function requests a termination of a connection or connection establishment. This operation is used by the active/client or a passive/server side Consumer of the connection model.

The *disconnect_flags* parameter allows Consumers to specify whether they want graceful or abrupt disconnect. Upon disconnect, all outstanding and in-progress DTOs and RMR Binds must be completed.

For abrupt disconnect, all outstanding DTOs and RMR Binds are completed unsuccessfully, and in-progress DTOs and RMR Binds can be completed successfully or unsuccessfully. If an in-progress DTO is completed unsuccessfully, all follow on in-progress DTOs in the same direction also must be completed unsuccessfully. This order is presented to the Consumer through a DTO completion Event Stream of the *recv_evd_handle* and and *request_evd_handle* of the Endpoint.

For graceful disconnect, all outstanding and in-progress request DTOs and RMR Binds must try to be completed successfully first, before disconnect proceeds. During that time, the local Endpoint is in a `DAT_EP_DISCONNECT_PENDING` state.

The Consumer can call abrupt `dat_ep_disconnect()` when the local Endpoint is in the `DAT_EP_DISCONNECT_PENDING` state. This causes the Endpoint to transition into `DAT_EP_STATE_DISCONNECTED` without waiting for outstanding and in-progress request DTOs and RMR Binds to successfully complete. The graceful `dat_ep_disconnect()` call when the local Endpoint is in the `DAT_EP_DISCONNECT_PENDING` state has no effect.

If the Endpoint is not in `DAT_EP_STATE_CONNECTED`, the semantic of the operation is the same for graceful or abrupt *disconnect_flags* value.

No new Send, RDMA Read, and RDMA Write DTOs, or RMR Binds can be posted to the Endpoint when the local Endpoint is in the `DAT_EP_DISCONNECT_PENDING` state.

The successful completion of the disconnect is reported to the Consumer through a DAT_CONNECTION_EVENT_DISCONNECTED event on *connect_evd_handle* of the Endpoint. The Endpoint is automatically transitioned into a DAT_EP_STATE_DISCONNECTED state upon successful asynchronous completion. If the same EVD is used for *connect_evd_handle* and any *recv_evd_handle* and request_evd_handle, all successful Completion events of in-progress DTOs precede the Disconnect Completion event.

Disconnecting an unconnected Disconnected Endpoint is no-op. Disconnecting an Endpoint in DAT_EP_STATE_UNCONNECTED, DAT_EP_STATE_RESERVED, DAT_EP_STATE_PASSIVE_CONNECTION_PENDING, and DAT_EP_STATE_TENTATIVE_CONNECTION_PENDING is disallowed.

Both abrupt and graceful disconnect of the Endpoint during connection establishment, DAT_EP_STATE_ACTIVE_CONNECTION_PENDING and DAT_EP_STATE_COMPLETION_PENDING, "aborts" the connection establishment and transitions the local Endpoint into DAT_EP_STATE_DISCONNECTED. That causes preposted Recv DTOs to be flushed to *recv_evd_handle*.

Return Values

DAT_SUCCESS — The operation was successful.

DAT_INVALID_HANDLE — The *ep_handle* parameter is invalid.

DAT_INSUFFICIENT_RESOURCES — The operation failed due to resource limitations.

DAT_INVALID_PARAMETER — The *disconnect_flags* parameter is invalid.

DAT_INVALID_STATE — A parameter is in an invalid state. Endpoint is not in the valid state for disconnect.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.1, 1.2

See Also

libdat(3LIB), attributes(5)

Name dat_ep_dup_connect – establish a connection between the local Endpoint and a remote Endpoint

Synopsis
```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ep_dup_connect (
    IN    DAT_EP_HANDLE    ep_handle,
    IN    DAT_EP_HANDLE    dup_ep_handle,
    IN    DAT_TIMEOUT      timeout,
    IN    DAT_COUNT        private_data_size,
    IN const  DAT_PVOID    private_data,
    IN    DAT_QOS          qos
    )
```

Parameters

ep_handle Handle for an instance of an Endpoint.

dup_ep_handle Connected local Endpoint that specifies a requested connection remote end.

timeout: Duration of time, in microseconds, that Consumers wait for Connection establishment. The value of `DAT_TIMEOUT_INFINITE` represents no timeout, indefinite wait. Values must be positive.

private_data_size Size of *private_data*. Must be nonnegative.

private_data Pointer to the private data that should be provided to the remote Consumer as part of the Connection Request. If *private_data_size* is zero, then *private_data* can be `NULL`.

qos Requested Quality of Service of the connection.

Description The `dat_ep_dup_connect()` function requests that a connection be established between the local Endpoint and a remote Endpoint. This operation is used by the active/client side Consumer of the connection model. The remote Endpoint is identified by the *dup_ep_handle*. The remote end of the requested connection shall be the same as the remote end of the *dup_ep_handle*. This is equivalent to requesting a connection to the same remote IA, Connection Qualifier, and *connect_flags* as used for establishing the connection on duplicated Endpoints and following the same redirections.

Upon establishing the requested connection as part of the successful completion of this operation, the local Endpoint is bound to a Port Qualifier of the local IA. The Port Qualifier is passed to the remote side of the requested connection and is available to the remote Consumer in the Connection Request of the `DAT_CONNECTION_REQUEST_EVENT`.

The Consumer-provided *private_data* is passed to the remote side and is provided to the remote Consumer in the Connection Request. Consumers can encapsulate any local Endpoint attributes that remote Consumers need to know as part of an upper-level protocol. Providers can also provide a Provider on the remote side any local Endpoint attributes and Transport-specific information needed for Connection establishment by the Transport.

Upon successful completion of this operation, the local Endpoint is transferred into DAT_EP_STATE_ACTIVE_CONNECTION_PENDING.

Consumers can request a specific value of *qos*. The Provider specifies which Quality of Service it supports in documentation and in the Provider attributes. If the local Provider or Transport does not support the requested *qos*, the operation fails and DAT_MODEL_NOT_SUPPORTED is returned synchronously. If the remote Provider does not support the requested *qos*, the local Endpoint is automatically transitioned into a DAT_EP_STATE_UNDISCONNECTED state, the connection is not established, and the event returned on the *connect_evd_handle* is DAT_CONNECTION_EVENT_NON_PEER_REJECTED. The same DAT_CONNECTION_EVENT_NON_PEER_REJECTED event is returned if connection cannot be established for all reasons for not establishing the connection, except timeout, remote host not reachable, and remote peer reject. For example, remote host is not reachable, remote Consumer is not listening on the requested Connection Qualifier, Backlog of the requested Service Point is full, and Transport errors. In this case, the local Endpoint is automatically transitioned into a DAT_EP_STATE_UNDISCONNECTED state.

The acceptance of the requested connection by the remote Consumer is reported to the local Consumer through a DAT_CONNECTION_EVENT_ESTABLISHED event on the *connect_evd_handle* of the local Endpoint.

The rejection of the connection by the remote Consumer is reported to the local Consumer through a DAT_CONNECTION_EVENT_PEER_REJECTED event on the *connect_evd_handle* of the local Endpoint and the local Endpoint is automatically transitioned into a DAT_EP_STATE_UNDISCONNECTED state.

When the Provider cannot reach the remote host or the remote host does not respond within the Consumer-requested *timeout*, a DAT_CONNECTION_EVENT_UNREACHABLE is generated on the *connect_evd_handle* of the Endpoint. The Endpoint transitions into a DAT_EP_STATE_DISCONNECTED state.

The local Endpoint is automatically transitioned into a DAT_EP_STATE_CONNECTED state when a Connection Request is accepted by the remote Consumer and the Provider completes the Transport-specific Connection establishment. The local Consumer is notified of the established connection through a DAT_CONNECTION_EVENT_ESTABLISHED event on the *connect_evd_handle* of the local Endpoint.

When the *timeout* expired prior to completion of the Connection establishment, the local Endpoint is automatically transitioned into a DAT_EP_STATE_UNDISCONNECTED state and the local Consumer through a DAT_CONNECTION_EVENT_TIMED_OUT event on the *connect_evd_handle* of the local Endpoint.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INSUFFICIENT_RESOURCES`	The operation failed due to resource limitations.
`DAT_INVALID_PARAMETER`	Invalid parameter.
`DAT_INVALID_HANDLE`	The *ep_handle* or *dup_ep_handle* parameter is invalid.
`DAT_INVALID_STATE`	A parameter is in an invalid state.
`DAT_MODEL_NOT_SUPPORTED`	The requested Model is not supported by the Provider. For example, requested *qos* was not supported by the local Provider.

Usage It is up to the Consumer to negotiate outstanding RDMA Read incoming and outgoing with a remote peer. The outstanding RDMA Read outgoing attribute should be smaller than the remote Endpoint outstanding RDMA Read incoming attribute. If this is not the case, connection establishment might fail.

DAT API does not define a protocol on how remote peers exchange Endpoint attributes. The exchange of outstanding RDMA Read incoming and outgoing attributes of EPs is left to the Consumer ULP. The Consumer can use Private Data for it.

If the Consumer does not care about posting RDMA Read operations or remote RDMA Read operations on the connection, it can set the two outstanding RDMA Read attribute values to 0.

If the Consumer does not set the two outstanding RDMA Read attributes of the Endpoint, the Provider is free to pick up any values as a default. The Provider is allowed to change these default values during connection setup.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.1, 1.2

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_ep_free – destroy an instance of the Endpoint

Synopsis
```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ep_free (
    IN    DAT_EP_HANDLE    ep_handle
    )
```

Parameters *ep_handle* Handle for an instance of the Endpoint.

Description The `dat_ep_free()` function destroys an instance of the Endpoint.

The Endpoint can be destroyed in any Endpoint state except Reserved, Passive Connection Pending, and Tentative Connection Pending. The destruction of the Endpoint can also cause the destruction of DTOs and RMRs posted to the Endpoint and not dequeued yet. This includes completions for all outstanding and in-progress DTOs/RMRs. The Consumer must be ready for all completions that are not dequeued yet either still being on the Endpoint *recv_evd_handle* and *request_evd_handle* or not being there.

The destruction of the Endpoint during connection setup aborts connection establishment.

If the Endpoint is in the Reserved state, the Consumer shall first destroy the associated Reserved Service Point that transitions the Endpoint into the Unconnected state where the Endpoint can be destroyed. If the Endpoint is in the Passive Connection Pending state, the Consumer shall first reject the associated Connection Request that transitions the Endpoint into the Unconnected state where the Endpoint can be destroyed. If the Endpoint is in the Tentative Connection Pending state, the Consumer shall reject the associated Connection Request that transitions the Endpoint back to Provider control, and the Endpoint is destroyed as far as the Consumer is concerned.

The freeing of an Endpoint also destroys an Event Stream for each of the associated Event Dispatchers.

Use of the handle of the destroyed Endpoint in any subsequent operation except for the `dat_ep_free()` fails.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INVALID_HANDLE`	The *ep_handle* parameter is invalid.
`DAT_INVALID_STATE`	Parameter in an invalid state. The Endpoint is in `DAT_EP_STATE_RESERVED`, `DAT_EP_STATE_PASSIVE_CONNECTION_PENDING`, or `DAT_EP_STATE_TENTATIVE_CONNECTION_PENDING`.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.1, 1.2

See Also libdat(3LIB), attributes(5)

Name dat_ep_get_status – provide a quick snapshot of the Endpoint

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ep_get_status (
    IN     DAT_EP_HANDLE    ep_handle,
    OUT    DAT_EP_STATE     *ep_state,
    OUT    DAT_BOOLEAN      *recv_idle,
    OUT    DAT_BOOLEAN      *request_idle
    )
```

Parameters

ep_handle Handle for an instance of the Endpoint.

ep_state Current state of the Endpoint.

recv_idle Status of the incoming DTOs on the Endpoint.

request_idle Status of the outgoing DTOs and RMR Bind operations on the Endpoint.

Description

the `dat_ep_get_status()` function provides the Consumer a quick snapshot of the Endpoint. The snapshot consists of the Endpoint state and whether there are outstanding or in-progress, incoming or outgoing DTOs. Incoming DTOs consist of Receives. Outgoing DTOs consist of the Requests, Send, RDMA Read, RDMA Write, and RMR Bind.

The *ep_state* parameter returns the value of the current state of the Endpoint *ep_handle*. State value is one of the following: `DAT_EP_STATE_UNCONNECTED`, `DAT_EP_STATE_RESERVED`, `DAT_EP_STATE_PASSIVE_CONNECTION_PENDING`, `DAT_EP_STATE_ACTIVE_CONNECTION_PENDING`, `DAT_EP_STATE_TENTATIVE_CONNECTION_PENDING`, `DAT_EP_STATE_CONNECTED`, `DAT_EP_STATE_DISCONNECT_PENDING`, or `DAT_EP_STATE_DISCONNECTED`.

A *recv_idle* value of `DAT_TRUE` specifies that there are no outstanding or in-progress Receive DTOs at the Endpoint, and `DAT_FALSE` otherwise.

A *request_idle* value of `DAT_TRUE` specifies that there are no outstanding or in-progress Send, RDMA Read, and RDMA Write DTOs, and RMR Binds at the Endpoint, and `DAT_FALSE` otherwise.

This call provides a snapshot of the Endpoint status only. No heroic synchronization with DTO queuing or processing is implied.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *ep_handle* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	uDAPL, 1.1, 1.2

See Also libdat(3LIB), attributes(5)

Name dat_ep_modify – change parameters of an Endpoint

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ep_modify (
    IN    DAT_EP_HANDLE        ep_handle,
    IN    DAT_EP_PARAM_MASK    ep_param_mask,
    IN    DAT_EP_PARAM         *ep_param
    )
```

Parameters

ep_handle Handle for an instance of the Endpoint.

ep_param_mask Mask for Endpoint parameters.

ep_param Pointer to the Consumer-allocated structure that contains Consumer-requested Endpoint parameters.

Description The `dat_ep_modify()` function provides the Consumer a way to change parameters of an Endpoint.

The *ep_param_mask* parameter allows Consumers to specify which parameters to modify. Providers modify values for *ep_param_mask* requested parameters only.

Not all the parameters of the Endpoint can be modified. Some can be modified only when the Endpoint is in a specific state. The following list specifies which parameters can be modified and when they can be modified.

Interface Adapter
Cannot be modified.

Endpoint belongs to an open instance of IA and that association cannot be changed.

Endpoint State
Cannot be modified.

State of Endpoint cannot be changed by a `dat_ep_modify()` operation.

Local IA Address
Cannot be modified.

Local IA Address cannot be changed by a `dat_ep_modify()` operation.

Local Port Qualifier
Cannot be modified.

Local port qualifier cannot be changed by a `dat_ep_modify()` operation.

Remote IA Address
Cannot be modified.

Remote IA Address cannot be changed by a `dat_ep_modify()` operation.

Remote Port Qualifier
Cannot be modified.

Remote port qualifier cannot be changed by a `dat_ep_modify()` operation

Protection Zone
Can be modified when in Quiescent, Unconnected, and Tentative Connection Pending states.

Protection Zone can be changed only when the Endpoint is in quiescent state. The only Endpoint states that isare quiescent isare `DAT_EP_STATE_UNCONNECTED` and `DAT_EP_STATE_TENTATIVE_CONNECTION_PENDING`. Consumers should be aware that any Receive DTOs currently posted to the Endpoint that do not match the new Protection Zone fail with a `DAT_PROTECTION_VIOLATION` return.

In DTO Event Dispatcher
Can be modified when in Unconnected, Reserved, Passive Connection Request Pending, and Tentative Connection Pending states.

Event Dispatcher for incoming DTOs (Receive) can be changed only prior to a request for a connection for an Active side or prior to accepting a Connection Request for a Passive side.

Out DTO Event Dispatcher
Can be modified when in Unconnected, Reserved, Passive Connection Pending, and Tentative Connection Pending states.

Event Dispatcher for outgoing DTOs (Send, RDMA Read, and RDMA Write) can be changed only prior to a request for a connection for an Active side or prior to accepting a Connection Request for a Passive side.

Connection Event Dispatcher
Can be modified when in Unconnected, Reserved, Passive Connection Pending, and Tentative Connection Pending states.

Event Dispatcher for the Endpoint Connection events can be changed only prior to a request for a connection for an Active side or accepting a Connection Request for a Passive side.

Service Type
Can be modified when in Unconnected, Reserved, Passive Connection Pending, and Tentative Connection Pending states.

Service Type can be changed only prior to a request for a connection for an Active side or accepting a Connection Request for a Passive side.

Maximum Message Size
Can be modified when in Unconnected, Reserved, Passive Connection Request Pending, and Tentative Connection Pending states.

Maximum Message Size can be changed only prior to a request for a connection for an Active side or accepting a Connection Request for a Passive side.

Maximum RDMA Size
Can be modified when in Unconnected, Reserved, Passive Connection Pending, and Tentative Connection Pending states.

Maximum RDMA Size can be changed only prior to a request for a connection for an Active side or accepting a Connection Request for a Passive side.

Quality of Service
Can be modified when in Unconnected, Reserved, Passive Connection Pending, and Tentative Connection Pending states.

QoS can be changed only prior to a request for a connection for an Active side or accepting a Connection Request for a Passive side.

Recv Completion Flags
Can be modified when in Unconnected, Reserved, Passive Connection Pending, and Tentative Connection Pending states.

Recv Completion Flags specifies what DTO flags the Endpoint should support for Receive DTO operations. The value can be `DAT_COMPLETION_NOTIFICATION_SUPPRESS_FLAG`, `DAT_COMPLETION_SOLICITED_WAIT_FLAG`, or `DAT_COMPLETION_EVD_THRESHOLD_FLAG`. Recv posting does not support `DAT_COMPLETION_SUPPRESS_FLAG` or `DAT_COMPLETION_BARRIER_FENCE_FLAG` dat_completion_flags values that are only applicable to Request postings. Recv Completion Flags can be changed only prior to a request for a connection for an Active side or accepting a Connection Request for a Passive side, but before posting of any Recvs.

Request Completion Flags
Can be modified when in Unconnected, Reserved, Passive Connection Request Pending, and Tentative Connection Pending states.

Request Completion Flags specifies what DTO flags the Endpoint should support for Send, RDMA Read, RDMA Write, and RMR Bind operations. The value can be: `DAT_COMPLETION_UNSIGNALLED_FLAG` or `DAT_COMPLETION_EVD_THRESHOLD_FLAG`. Request postings always support `DAT_COMPLETION_SUPPRESS_FLAG`, `DAT_COMPLETION_SOLICITED_WAIT_FLAG`, or `DAT_COMPLETION_BARRIER_FENCE_FLAG` completion_flags values. Request Completion Flags can be changed only prior to a request for a connection for an Active side or accepting a Connection Request for a Passive side.

Maximum Recv DTO
Can be modified when in Unconnected, Reserved, Passive Connection Pending, and Tentative Connection Pending states.

Maximum Recv DTO specifies the maximum number of outstanding Consumer-submitted Receive DTOs that a Consumer expects at any time at the Endpoint.

Maximum Recv DTO can be changed only prior to a request for a connection for an Active side or accepting a Connection Request for a Passive side.

Maximum Request DTO
Can be modified when in Unconnected, Reserved, Passive Connection Pending, and Tentative Connection Pending states.

Maximum Request DTO specifies the maximum number of outstanding Consumer-submitted send and RDMA DTOs and RMR Binds that a Consumer expects at any time at the Endpoint. Maximum Out DTO can be changed only prior to a request for a connection for an Active side or accepting a Connection Request for a Passive side.

Maximum Recv IOV
Can be modified when in Unconnected, Reserved, Passive Connection Pending, and Tentative Connection Pending states.

Maximum Recv IOV specifies the maximum number of elements in IOV that a Consumer specifies for posting a Receive DTO for the Endpoint. Maximum Recv IOV can be changed only prior to a request for a connection for an Active side or accepting a Connection Request for a Passive side.

Maximum Request IOV
Can be modified when in Unconnected, Reserved, Passive Connection Pending, and Tentative Connection Pending states.

Maximum Request IOV specifies the maximum number of elements in IOV that a Consumer specifies for posting a Send, RDMA Read, or RDMA Write DTO for the Endpoint. Maximum Request IOV can be changed only prior to a request for a connection for an Active side or accepting a Connection Request for a Passive side.

Maximum outstanding RDMA Read as target
Can be modified when in Unconnected, Reserved, Passive Connection Pending, and Tentative Connection Pending states.

Maximum number of outstanding RDMA Reads for which the Endpoint is the target.

Maximum outstanding RDMA Read as originator
Can be modified when in Unconnected, Reserved, Passive Connection Pending, and Tentative Connection Pending states.

Maximum number of outstanding RDMA Reads for which the Endpoint is the originator.

Num transport-specific attributes
Can be modified when in Quiescent (unconnected) state.

Number of transport-specific attributes to be modified.

Transport-specific endpoint attributes
Can be modified when in Quiescent (unconnected) state.

Transport-specific attributes can be modified only in the transport-defined Endpoint state. The only guaranteed safe state in which to modify transport-specific Endpoint attributes is the quiescent state `DAT_EP_STATE_UNCONNECTED`.

Num provider-specific attributes
Can be modified when in Quiescent (unconnected) state.

Number of Provider-specific attributes to be modified.

Provider-specific endpoint attributes
Can be modified when in Quiescent (unconnected) state.

Provider-specific attributes can be modified only in the Provider-defined Endpoint state. The only guaranteed safe state in which to modify Provider-specific Endpoint attributes is the quiescent state `DAT_EP_STATE_UNCONNECTED`.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *ep_handle* parameter is invalid.

`DAT_INVALID_PARAMETER` The *ep_param_mask* parameter is invalid, or one of the requested Endpoint parameters or attributes was invalid, not supported, or cannot be modified.

`DAT_INVALID_STATE` Parameter in an invalid state. The Endpoint was not in the state that allows one of the parameters or attributes to be modified.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.1, 1.2

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_ep_post_rdma_read – transfer all data to the local data buffer

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ep_post_rdma_read (
    IN    DAT_EP_HANDLE        ep_handle,
    IN    DAT_COUNT            num_segments,
    IN    DAT_LMR_TRIPLET      *local_iov,
    IN    DAT_DTO_COOKIE       user_cookie,
    IN    DAT_RMR_TRIPLET      *remote_buffer,
    IN    DAT_COMPLETION_FLAGS completion_flags
    )
```

Parameters

ep_handle Handle for an instance of the Endpoint.

num_segments Number of *lmr_triplets* in *local_iov*.

local_iov I/O Vector that specifies the local buffer to fill.

user_cookie User-provided cookie that is returned to the Consumer at the completion of the RDMA Read. Can be `NULL`.

remote_buffer A pointer to an RMR Triplet that specifies the remote buffer from which the data is read.

completion_flags Flags for posted RDMA Read. The default `DAT_COMPLETION_DEFAULT_FLAG` is 0x00. Other values are as follows:

Completion Suppression	`DAT_COMPLETION_SUPPRESS_FLAG`	
	0x01	Suppress successful Completion.
Notification of Completion	`DAT_COMPLETION_UNSIGNALLED_FLAG`	
	0x04	Non-notification completion. Local Endpoint must be configured for Notification Suppression.
Barrier Fence	`DAT_COMPLETION_BARRIER_FENCE_FLAG`	
	0x08	Request for Barrier Fence.

Description The `dat_ep_post_rdma_read()` function requests the transfer of all the data specified by the *remote_buffer* over the connection of the *ep_handle* Endpoint into the *local_iov*.

The *num_segments* parameter specifies the number of segments in the *local_iov*. The *local_iov* segments are filled in the I/O Vector order until the whole message is received. This ensures that all the "front" segments of the *local_iov* I/O Vector are completely filled, only one segment is partially filled, if needed, and all segments that follow it are not filled at all.

The *user_cookie* allows Consumers to have unique identifiers for each DTO. These identifiers are completely under user control and are opaque to the Provider. There is no requirement on the Consumer that the value *user_cookie* should be unique for each DTO. The *user_cookie* is returned to the Consumer in the Completion event for the posted RDMA Read.

A Consumer must not modify the *local_iov* or its content until the DTO is completed. When a Consumer does not adhere to this rule, the behavior of the Provider and the underlying Transport is not defined. Providers that allow Consumers to get ownership of the *local_iov* but not the memory it specifies back after the `dat_ep_post_rdma_read()` returns should document this behavior and also specify its support in Provider attributes. This behavior allows Consumers full control of the *local_iov* after `dat_ep_post_rdma_read()` returns. Because this behavior is not guaranteed by all Providers, portable Consumers should not rely on this behavior. Consumers should not rely on the Provider copying *local_iov* information.

The completion of the posted RDMA Read is reported to the Consumer asynchronously through a DTO Completion event based on the specified *completion_flags* value. The value of `DAT_COMPLETION_UNSIGNALLED_FLAG` is only valid if the Endpoint Request Completion Flags `DAT_COMPLETION_UNSIGNALLED_FLAG`. Otherwise, `DAT_INVALID_PARAMETER` is returned.

The `DAT_SUCCESS` return of the `dat_ep_post_rdma_read()` is at least the equivalent of posting an RDMA Read operation directly by native Transport. Providers should avoid resource allocation as part of `dat_ep_post_rdma_read()` to ensure that this operation is nonblocking and thread safe for an UpCall.

The operation is valid for the Endpoint in the `DAT_EP_STATE_CONNECTED` and `DAT_EP_STATE_DISCONNECTED` states. If the operation returns successfully for the Endpoint in the `DAT_EP_STATE_DISCONNECTED` state, the posted RDMA Read is immediately flushed to *request_evd_handle*.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INSUFFICIENT_RESOURCES`	The operation failed due to resource limitations.
`DAT_INVALID_PARAMETER`	Invalid parameter. For example, one of the IOV segments pointed to a memory outside its LMR.
`DAT_INVALID_HANDLE`	The *ep_handle* parameter is invalid.
`DAT_INVALID_STATE`	A parameter is in an invalid state. Endpoint was not in the `DAT_EP_STATE_CONNECTED` or `DAT_EP_STATE_DISCONNECTED` state.
`DAT_LENGTH_ERROR`	The size of the receiving buffer is too small for sending buffer data. The size of the local buffer is too small for the data of the remote buffer.

`DAT_PROTECTION_VIOLATION` Protection violation for local or remote memory access. Protection Zone mismatch between either an LMR of one of the *local_iov* segments and the local Endpoint or the *rmr_context* and the remote Endpoint.

`DAT_PRIVILEGES_VIOLATION` Privileges violation for local or remote memory access. Either one of the LMRs used in *local_iov* is invalid or does not have the local write privileges, or *rmr_context* does not have the remote read privileges.

Usage

For best RDMA Read operation performance, the Consumer should align each buffer segment of *local_iov* to the Optimal Buffer Alignment attribute of the Provider. For portable applications, the Consumer should align each buffer segment of *local_iov* to the `DAT_OPTIMAL_ALIGNMENT`.

If connection was established without outstanding RDMA Read attributes matching on Endpoints on both sides (outstanding RDMA Read outgoing on one end is larger than the outstanding RDMA Read incoming on the other end), connection is broken when the number of incoming RDMA Read exceeds the outstanding RDMA Read incoming attribute of the Endpoint. The Consumer can use its own flow control to ensure that it does not post more RDMA Reads then the remote EP outstanding RDMA Read incoming attribute is. Thus, they do not rely on the underlying Transport enforcing it.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.1, 1.2

See Also

`libdat(3LIB)`, `attributes(5)`

Name dat_ep_post_rdma_write – write all data to the remote data buffer

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ep_post_rdma_read (
    IN    DAT_EP_HANDLE        ep_handle,
    IN    DAT_COUNT            num_segments,
    IN    DAT_LMR_TRIPLET      *local_iov,
    IN    DAT_DTO_COOKIE       user_cookie,
    IN    DAT_RMR_TRIPLET      *remote_buffer,
    IN    DAT_COMPLETION_FLAGS completion_flags
    )
```

Parameters

ep_handle Handle for an instance of the Endpoint.

num_segments Number of *lmr_triplets* in *local_iov*.

local_iov I/O Vector that specifies the local buffer from which the data is transferred.

user_cookie User-provided cookie that is returned to the Consumer at the completion of the RDMA Write.

remote_buffer A pointer to an RMR Triplet that specifies the remote buffer from which the data is read.

completion_flags Flags for posted RDMA read. The default `DAT_COMPLETION_DEFAULT_FLAG` is 0x00. Other values are as follows:

Completion Suppression	`DAT_COMPLETION_SUPPRESS_FLAG`	
	0x01	Suppress successful Completion.
Notification of Completion	`DAT_COMPLETION_UNSIGNALLED_FLAG`	
	0x04	Non-notification completion. Local Endpoint must be configured for Notification Suppression.
Barrier Fence	`DAT_COMPLETION_BARRIER_FENCE_FLAG`	
	0x08	Request for Barrier Fence.

Description The `dat_ep_post_rdma_write()` function requests the transfer of all the data specified by the *local_iov* over the connection of the *ep_handle* Endpoint into the *remote_buffer*.

The *num_segments* parameter specifies the number of segments in the *local_iov*. The *local_iov* segments are traversed in the I/O Vector order until all the data is transferred.

A Consumer must not modify the *local_iov* or its content until the DTO is completed. When a Consumer does not adhere to this rule, the behavior of the Provider and the underlying Transport is not defined. Providers that allow Consumers to get ownership of the *local_iov* but not the memory it specifies back after the `dat_ep_post_rdma_write()` returns should document this behavior and also specify its support in Provider attributes. This behavior allows Consumers full control of the *local_iov* after `dat_ep_post_rdma_write()` returns. Because this behavior is not guaranteed by all Providers, portable Consumers should not rely on this behavior. Consumers should not rely on the Provider copying *local_iov* information.

The `DAT_SUCCESS` return of the `dat_ep_post_rdma_write()` is at least the equivalent of posting an RDMA Write operation directly by native Transport. Providers should avoid resource allocation as part of `dat_ep_post_rdma_write()` to ensure that this operation is nonblocking and thread safe for an UpCall.

The completion of the posted RDMA Write is reported to the Consumer asynchronously through a DTO Completion event based on the specified *completion_flags* value. The value of `DAT_COMPLETION_UNSIGNALLED_FLAG` is only valid if the Endpoint Request Completion Flags `DAT_COMPLETION_UNSIGNALLED_FLAG`. Otherwise, `DAT_INVALID_PARAMETER` is returned.

The *user_cookie* allows Consumers to have unique identifiers for each DTO. These identifiers are completely under user control and are opaque to the Provider. There is no requirement on the Consumer that the value *user_cookie* should be unique for each DTO. The *user_cookie* is returned to the Consumer in the Completion event for the posted RDMA Write.

The operation is valid for the Endpoint in the `DAT_EP_STATE_CONNECTED` and `DAT_EP_STATE_DISCONNECTED` states. If the operation returns successfully for the Endpoint in the `DAT_EP_STATE_DISCONNECTED` state, the posted RDMA Write is immediately flushed to *request_evd_handle*.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INSUFFICIENT_RESOURCES`	The operation failed due to resource limitations.
`DAT_INVALID_PARAMETER`	Invalid parameter. For example, one of the IOV segments pointed to a memory outside its LMR.
`DAT_INVALID_HANDLE`	The *ep_handle* parameter is invalid.
`DAT_INVALID_STATE`	A parameter is in an invalid state. Endpoint was not in the `DAT_EP_STATE_CONNECTED` or `DAT_EP_STATE_DISCONNECTED` state.
`DAT_LENGTH_ERROR`	The size of the receiving buffer is too small for sending buffer data. The size of the remote buffer is too small for the data of the local buffer.
`DAT_PROTECTION_VIOLATION`	Protection violation for local or remote memory access. Protection Zone mismatch between either an LMR of one

of the *local_iov* segments and the local Endpoint or the *rmr_context* and the remote Endpoint.

DAT_PRIVILEGES_VIOLATION
: Privileges violation for local or remote memory access. Either one of the LMRs used in *local_iov* is invalid or does not have the local read privileges, or *rmr_context* does not have the remote write privileges.

Usage

For best RDMA Write operation performance, the Consumer should align each buffer segment of *local_iov* to the Optimal Buffer Alignment attribute of the Provider. For portable applications, the Consumer should align each buffer segment of *local_iov* to the DAT_OPTIMAL_ALIGNMENT.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.1, 1.2

See Also

libdat(3LIB), attributes(5)

Name dat_ep_post_recv – receive data over the connection of the Endpoint

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ep_post_recv (
    IN    DAT_EP_HANDLE        ep_handle,
    IN    DAT_COUNT            num_segments,
    IN    DAT_LMR_TRIPLET      *local_iov,
    IN    DAT_DTO_COOKIE       user_cookie,
    IN    DAT_COMPLETION_FLAGS completion_flags
    )
```

Parameters

ep_handle — Handle for an instance of the Endpoint.

num_segments — Number of *lmr_triplets* in *local_iov*. Can be 0 for receiving a 0 size message.

local_iov — I/O Vector that specifies the local buffer to be filled. Can be `NULL` for receiving a 0 size message.

user_cookie: — User-provided cookie that is returned to the Consumer at the completion of the Receive DTO. Can be `NULL`.

completion_flags — Flags for posted Receive. The default `DAT_COMPLETION_DEFAULT_FLAG` is 0x00. Other values are as follows:

Notification of Completion — `DAT_COMPLETION_UNSIGNALLED_FLAG`

0x04 — Non-notification completion. Local Endpoint must be configured for Unsignaled CompletionNotification Suppression.

Description The `dat_ep_post_recv()` function requests the receive of the data over the connection of the *ep_handle* Endpoint of the incoming message into the *local_iov*.

The *num_segments* parameter specifies the number of segments in the *local_iov*. The *local_iov* segments are filled in the I/O Vector order until the whole message is received. This ensures that all the "front" segments of the *local_iov* I/O Vector are completely filled, only one segment is partially filled, if needed, and all segments that follow it are not filled at all.

The *user_cookie* allows Consumers to have unique identifiers for each DTO. These identifiers are completely under user control and are opaque to the Provider. There is no requirement on the Consumer that the value *user_cookie* should be unique for each DTO. The *user_cookie* is returned to the Consumer in the Completion event for the posted Receive.

The completion of the posted Receive is reported to the Consumer asynchronously through a DTO Completion event based on the configuration of the connection for Solicited Wait and the specified *completion_flags* value for the matching Send. The value of `DAT_COMPLETION _UNSIGNALLED_FLAG` is only valid if the Endpoint Recv Completion Flags `DAT_COMPLETION_UNSIGNALLED_FLAG`. Otherwise, `DAT_INVALID_PARAMETER` is returned.

A Consumer must not modify the *local_iov* or its content until the DTO is completed. When a Consumer does not adhere to this rule, the behavior of the Provider and the underlying Transport is not defined. Providers that allow Consumers to get ownership of the *local_iov* but not the memory it specified back after the `dat_ep_post_recv()` returns should document this behavior and also specify its support in Provider attributes. This behavior allows Consumer full control of the *local_iov* content after `dat_ep_post_recv()` returns. Because this behavior is not guaranteed by all Providers, portable Consumers should not rely on this behavior. Consumers shouldnot rely on the Provider copying *local_iov* information.

The `DAT_SUCCESS` return of the `dat_ep_post_recv()` is at least the equivalent of posting a Receive operation directly by native Transport. Providers should avoid resource allocation as part of `dat_ep_post_recv()` to ensure that this operation is nonblocking and thread safe for an UpCall.

If the size of an incoming message is larger than the size of the *local_iov*, the reported status of the posted Receive DTO in the corresponding Completion DTO event is `DAT_DTO_LENGTH_ERROR`. If the reported status of the Completion DTO event corresponding to the posted Receive DTO is not `DAT_DTO_SUCCESS`, the content of the *local_iov* is not defined.

The operation is valid for all states of the Endpoint. The actual data transfer does not take place until the Endpoint is in the `DAT_EP_STATE_CONNECTED` state. The operation on the Endpoint in `DAT_EP_STATE_DISCONNECTED` is allowed. If the operation returns successfully, the posted Recv is immediately flushed to *recv_evd_handle*.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INSUFFICIENT_RESOURCES`	The operation failed due to resource limitations.
`DAT_INVALID_PARAMETER`	Invalid parameter. For example, one of the IOV segments pointed to a memory outside its LMR.
`DAT_INVALID_HANDLE`	The *ep_handle* parameter is invalid.
`DAT_PROTECTION_VIOLATION`	Protection violation for local or remote memory access. Protection Zone mismatch between an LMR of one of the *local_iov* segments and the local Endpoint.
`DAT_PRIVILEGES_VIOLATION`	Privileges violation for local or remote memory access. One of the LMRs used in *local_iov* was either invalid or did not have the local read privileges.

Usage For best Recv operation performance, the Consumer should align each buffer segment of *local_iov* to the Optimal Buffer Alignment attribute of the Provider. For portable applications, the Consumer should align each buffer segment of *local_iov* to the `DAT_OPTIMAL_ALIGNMENT`.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.1, 1.2

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_ep_post_send – transfer data to the remote side

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ep_post_send (
    IN    DAT_EP_HANDLE        ep_handle,
    IN    DAT_COUNT            num_segments,
    IN    DAT_LMR_TRIPLET      *local_iov,
    IN    DAT_DTO_COOKIE       user_cookie,
    IN    DAT_COMPLETION_FLAGS completion_flags
    )
```

Parameters

ep_handle Handle for an instance of the Endpoint.

num_segments Number of *lmr_triplets* in *local_iov*. Can be 0 for 0 size message.

local_iov I/O Vector that specifies the local buffer that contains data to be transferred. Can be `NULL` for 0 size message.

user_cookie: User-provided cookie that is returned to the Consumer at the completion of the send. Can be `NULL`.

completion_flags Flags for posted Send. The default `DAT_COMPLETION_DEFAULT_FLAG` is 0x00. Other values are as follows:

Completion Suppression `DAT_COMPLETION_SUPPRESS_FLAG`

0x01 Suppress successful Completion.

Solicited Wait `DAT_COMPLETION_SOLICITED_WAIT_FLAG`

0x02 Request for notification completion for matching receive on the other side of the connection.

Notification of Completion `DAT_COMPLETION_UNSIGNALLED_FLAG`

0x04 Non-notification completion. Local Endpoint must be configured for Notification Suppression.

Barrier Fence `DAT_COMPLETION_BARRIER_FENCE_FLAG`

0x08 Request for Barrier Fence.

Description The `dat_ep_post_send()` function requests a transfer of all the data from the *local_iov* over the connection of the *ep_handle* Endpoint to the remote side.

The *num_segments* parameter specifies the number of segments in the *local_iov*. The *local_iov* segments are traversed in the I/O Vector order until all the data is transferred.

A Consumer cannot modify the *local_iov* or its content until the DTO is completed. When a Consumer does not adhere to this rule, the behavior of the Provider and the underlying Transport is not defined. Providers that allow Consumers to get ownership of the *local_iov* back after the `dat_ep_post_send()` returns should document this behavior and also specify its support in Provider attributes. This behavior allows Consumers full control of the *local_iov*, but not the memory it specifies after `dat_ep_post_send()` returns. Because this behavior is not guaranteed by all Providers, portable Consumers should not rely on this behavior. Consumers should not rely on the Provider copying *local_iov* information.

The `DAT_SUCCESS` return of the `dat_ep_post_send()` is at least the equivalent of posting a Send operation directly by native Transport. Providers should avoid resource allocation as part of `dat_ep_post_send()` to ensure that this operation is nonblocking and thread safe for an UpCall.

The completion of the posted Send is reported to the Consumer asynchronously through a DTO Completion event based on the specified *completion_flags* value. The value of `DAT_COMPLETION _UNSIGNALLED_FLAG` is only valid if the Endpoint Request Completion Flags `DAT_COMPLETION_UNSIGNALLED_FLAG`. Otherwise, `DAT_INVALID_PARAMETER` is returned.

The *user_cookie* allows Consumers to have unique identifiers for each DTO. These identifiers are completely under user control and are opaque to the Provider. There is no requirement on the Consumer that the value *user_cookie* should be unique for each DTO. The *user_cookie* is returned to the Consumer in the Completion event for the posted Send.

The operation is valid for the Endpoint in the `DAT_EP_STATE_CONNECTED` and `DAT_EP_STATE_DISCONNECTED` states. If the operation returns successfully for the Endpoint in the `DAT_EP_STATE_DISCONNECTED` state, the posted Send is immediately flushed to *request_evd_handle*.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INSUFFICIENT_RESOURCES`	The operation failed due to resource limitations.
`DAT_INVALID_PARAMETER`	Invalid parameter. For example, one of the IOV segments pointed to a memory outside its LMR.
`DAT_INVALID_HANDLE`	The *ep_handle* parameter is invalid.
`DAT_INVALID_STATE`	A parameter is in an invalid state. Endpoint was not in the `DAT_EP_STATE_CONNECTED` or `DAT_EP_STATE_DISCONNECTED` state.

`DAT_PROTECTION_VIOLATION`	Protection violation for local or remote memory access. Protection Zone mismatch between an LMR of one of the *local_iov* segments and the local Endpoint.
`DAT_PRIVILEGES_VIOLATION`	Privileges violation for local or remote memory access. One of the LMRs used in *local_iov* was either invalid or did not have the local read privileges.

Usage For best Send operation performance, the Consumer should align each buffer segment of *local_iov* to the Optimal Buffer Alignment attribute of the Provider. For portable applications, the Consumer should align each buffer segment of *local_iov* to the `DAT_OPTIMAL_ALIGNMENT`.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.1, 1.2

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_ep_query – provide parameters of the Endpoint

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ep_query (
    IN    DAT_EP_HANDLE       ep_handle,
    IN    DAT_EP_PARAM_MASK   ep_param_mask,
    OUT   DAT_EP_PARAM        *ep_param
    )
```

Parameters

ep_handle Handle for an instance of the Endpoint.

ep_param_mask Mask for Endpoint parameters.

ep_param Pointer to a Consumer-allocated structure that the Provider fills with Endpoint parameters.

Description The `dat_ep_query()` function provides the Consumer parameters, including attributes and status, of the Endpoint. Consumers pass in a pointer to Consumer-allocated structures for Endpoint parameters that the Provider fills.

The *ep_param_mask* parameter allows Consumers to specify which parameters to query. The Provider returns values for *ep_param_mask* requested parameters. The Provider can return values for any other parameters.

Some of the parameters only have values for certain Endpoint states. Specifically, the values for *remote_ia_address* and *remote_port_qual* are valid only for Endpoints in the `DAT_EP_STATE_PASSIVE_CONNECTION_PENDING`, `DAT_EP_STATE_ACTIVE_CONNECTION_PENDING`, `DAT_EP_STATE_TENTATIVE_CONNECTION_PENDING`, `DAT_EP_STATE_DISCONNECT_PENDING`, `DAT_EP_STATE_COMPLETION_PENDING`, or `DAT_EP_STATE_CONNECTED` states. The values of *local_port_qual* is valid only for Endpoints in the `DAT_EP_STATE_PASSIVE_CONNECTION_PENDING`, `DAT_EP_STATE_ACTIVE_CONNECTION_PENDING`, `DAT_EP_STATE_DISCONNECT_PENDING`, `DAT_EP_STATE_COMPLETION_PENDING`, or `DAT_EP_STATE_CONNECTED` states, and might be valid for `DAT_EP_STATE_UNCONNECTED`, `DAT_EP_STATE_RESERVED`, `DAT_EP_STATE_TENTATIVE_CONNECTION_PENDING`, `DAT_EP_STATE_PASSIVE_CONNECTION_PENDING`, and `DAT_EP_STATE_UNCONNECTED` states.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *ep_handle* parameter is invalid.

`DAT_INVALID_PARAMETER` The *ep_param_mask* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.1, 1.2

See Also libdat(3LIB), attributes(5)

Name dat_ep_recv_query – provide Endpoint receive queue consumption on SRQ

Synopsis
```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ep_recv_query (
        IN      DAT_EP_HANDLE       ep_handle,
        OUT     DAT_COUNT           *nbufs_allocated,
        OUT     DAT_COUNT           *bufs_alloc_span
    )
```

Parameters

ep_handle Handle for an instance of the EP.

nbufs_allocated The number of buffers at the EP for which completions have not yet been generated.

bufs_alloc_span The span of buffers that EP needs to complete arriving messages.

Description The `dat_ep_recv_query()` function provides to the Consumer a snapshot for Recv buffers on EP. The values for *nbufs_allocated* and *bufs_alloc_span* are not defined when `DAT_RETURN` is not `DAT_SUCCESS`.

The Provider might not support *nbufs_allocated*, *bufs_alloc_span* or both. Check the Provider attribute for EP Recv info support. When the Provider does not support both of these counts, the return value for the operation can be `DAT_MODEL_NOT_SUPPORTED`.

If *nbufs_allocated* is not `NULL`, the count pointed to by *nbufs_allocated* will return a snapshot count of the number of buffers allocated to *ep_handle* but not yet completed.

Once a buffer has been allocated to an EP, it will be completed to the EP *recv_evd* if the EVD has not overflowed. When an EP does not use SRQ, a buffer is allocated as soon as it is posted to the EP. For EP that uses SRQ, a buffer is allocated to the EP when EP removes it from SRQ.

If *bufs_alloc_span* is not `NULL`, then the count to which *bufs_alloc_span* pointed will return the span of buffers allocated to the *ep_handle*. The span is the number of additional successful Recv completions that EP can generate if all the messages it is currently receiving will complete successfully.

If a message sequence number is assigned to all received messages, the buffer span is the difference between the latest message sequence number of an allocated buffer minus the latest message sequence number for which completion has been generated. This sequence number only counts Send messages of remote Endpoint of the connection.

The Message Sequence Number (MSN) represents the order that Send messages were submitted by the remote Consumer. The ordering of sends is intrinsic to the definition of a reliable service. Therefore every send message does have a MSN whether or not the native transport has a field with that name.

For both *nbufs_allocated* and *bufs_alloc_span*, the Provider can return the reserved value `DAT_VALUE_UNKNOWN` if it cannot obtain the requested count at a reasonable cost.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INVALID_PARAMETER`	Invalid parameter.
`DAT_INVALID_HANDLE`	The DAT handle ep_handle is invalid.
`DAT_MODEL_NOT_SUPPORTED`	The requested Model was not supported by the Provider.

Usage If the Provider cannot support the query for *nbufs_allocated* or *bufs_alloc_span*, the value returned for that attribute must be `DAT_VALUE_UNKNOWN`.

An implementation that processes incoming packets out of order and allocates from SRQs on an arrival basis can have gaps in the MSNs associated with buffers allocated to an Endpoint.

For example, suppose Endpoint X has received buffer fragments for MSNs 19, 22, and 23. With arrival ordering, the EP would have allocated three buffers from the SRQ for messages 19, 22, and 23. The number allocated would be 3, but the span would be 5. The difference of two represents the buffers that will have to be allocated for messages 20 and 21. They have not yet been allocated, but messages 22 and 23 will not be delivered until after messages 20 and 21 have not only had their buffers allocated but have also completed.

An implementation can choose to allocate 20 and 21 as soon as any higher buffer is allocated. This makes sense if you presume that this is a valid connection, because obviously 20 and 21 are in flight. However, it creates a greater vulnerability to Denial Of Service attacks. There are also other implementation tradeoffs, so the Consumer should accept that different RNICs for iWARP will employ different strategies on when to perform these allocations.

Each implementation will have some method of tracking the receive buffers already associated with an EP and knowing which buffer matches which incoming message, though those methods might vary. In particular, there are valid implementations such as linked lists, where a count of the outstanding buffers is not instantly available. Such implementations would have to scan the allocated list to determine both the number of buffers and their span. If such a scan is necessary, it is important that it be only a single scan. The set of buffers that was counted must be the same set of buffers for which the span is reported.

The implementation should not scan twice, once to count the buffers and then again to determine their span. Not only is it inefficient, but it might produce inconsistent results if buffers were completed or arrived between the two scans.

Other implementations can simply maintain counts of these values to easily filter invalid packets. If so, these status counters should be updated and referenced atomically.

The implementation must never report *n* buffers in a span that is less than *n*.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.2

See Also dat_ep_create(3DAT), dat_srq_create(3DAT), dat_srq_free(3DAT), dat_srq_query(3DAT), dat_ep_set_watermark(3DAT), libdat(3LIB), attributes(5)

Name dat_ep_reset – transition the local Endpoint from a Disconnected to an Unconnected state

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ep_reset (
    IN     DAT_EP_HANDLE     ep_handle
    )
```

Parameters

ep_handle Handle for an instance of Endpoint.

Description

The `dat_ep_reset()` function transitions the local Endpoint from a Disconnected to an Unconnected state.

The operation might cause the loss of any completions of previously posted DTOs and RMRs that were not dequeued yet.

The `dat_ep_reset()` function is valid for both Disconnected and Unconnected states. For Unconnected state, the operation is no-op because the Endpoint is already in an Unconnected state. For an Unconnected state, the preposted Recvs are not affected by the call.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` ep_handle is invalid.

`DAT_INVALID_STATE` Parameter in an invalid state. Endpoint is not in the valid state for reset.

Usage

If the Consumer wants to ensure that all Completions are dequeued, the Consumer can post DTO or RMR operations as a "marker" that are flushed to *recv_evd_handle* or *request_evd_handle*. Now, when the Consumer dequeues the completion of the "marker" from the EVD, it is guaranteed that all previously posted DTO and RMR completions for the Endpoint were dequeued for that EVD. Now, it is safe to reset the Endpoint without losing any completions.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.1, 1.2

See Also

`libdat(3LIB)`, `attributes(5)`

Name dat_ep_set_watermark – set high watermark on Endpoint

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ep_set_watermark (
    IN      DAT_EP_HANDLE      ep_handle,
    IN      DAT_COUNT          soft_high_watermark,
    IN      DAT_COUNT          hard_high_watermark
    )
```

Parameters

ep_handle The handle for an instance of an Endpoint.

soft_high_watermark The soft high watermark for the number of Recv buffers consumed by the Endpoint.

hard_high_watermark The hard high watermark for the number of Recv buffers consumed by the Endpoint.

Description

The `dat_ep_set_watermark()` function sets the soft and hard high watermark values for EP and arms EP for generating asynchronous events for high watermarks. An asynchronous event will be generated for IA *async_evd* when the number of Recv buffers at EP exceeds the soft high watermark for the first time. A connection broken event will be generated for EP *connect_evd* when the number of Recv buffers at EP exceeds the hard high watermark. These can occur during this call or when EP takes a buffer from the SRQ or EP RQ. The soft and hard high watermark asynchronous event generation and setting are independent of each other.

The asynchronous event for a soft high watermark is generated only once per setting. Once an event is generated, no new asynchronous events for the soft high watermark is generated until the EP is again set for the soft high watermark. If the Consumer is once again interested in the event, the Consumer should again set the soft high watermark.

If the Consumer is not interested in a soft or hard high watermark, the value of `DAT_WATERMARK_INFINITE` can be specified for the case that is the default value. This value specifies that a non-asynchronous event will be generated for a high watermark EP attribute for which this value is set. It does not prevent generation of a connection broken event for EP when no Recv buffer is available for a message arrived on the EP connection.

The operation is supported for all states of Endpoint.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *ep_handle* argument is an invalid DAT handle.

`DAT_INVALID_PARAMETER` One of the parameters is invalid.

`DAT_MODEL_NOT_SUPPORTED` The requested Model was not supported by the Provider. The Provider does not support EP Soft or Hard High

Watermarks.

Usage For a hard high watermark, the Provider is ready to generate a connection broken event as soon as the connection is established.

If the asynchronous event for a soft or hard high watermark has not yet been generated, this call simply modifies the values for these attributes. The Provider remains armed for generation of these asynchronous events.

Regardless of whether an asynchronous event for the soft and hard high watermark has been generated, this operation will set the generation of an asynchronous event with the Consumer-provided high watermark values. If the new high watermark values are below the current number of Receive DTOs at EP, an asynchronous event will be generated immediately. Otherwise the old soft or hard (or both) high watermark values are simply replaced with the new ones.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.2

See Also `dat_ep_create`(3DAT), `dat_ep_recv_query`(3DAT), `dat_srq_create`(3DAT), `dat_srq_free`(3DAT), `dat_srq_post_recv`(3DAT), `dat_srq_query`(3DAT), `dat_srq_resize`(3DAT), `dat_srq_set_lw`(3DAT), `libdat`(3LIB), `attributes(5)`

Name dat_evd_clear_unwaitable – transition the Event Dispatcher into a waitable state

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_evd_clear_unwaitable(
    IN    DAT_EVD_HANDLE     evd_handle
    )
```

Parameters *evd_handle* Handle for an instance of Event Dispatcher.

Description The `dat_evd_clear_unwaitable()` transitions the Event Dispatcher into a waitable state. In this state, calls to `dat_evd_wait`(3DAT) are permitted on the EVD. The actual state of the Event Dispatcher is accessible through `dat_evd_query`(3DAT) and is `DAT_EVD_WAITABLE` after the return of this operation.

This call does not affect a CNO associated with this EVD at all. Events arriving on the EVD after it is set waitable still trigger the CNO (if appropriate), and can be retrieved with `dat_evd_dequeue`(3DAT).

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *evd_handle* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	uDAPL, 1.1, 1.2

See Also `dat_evd_dequeue`(3DAT), `dat_evd_query`(3DAT), `dat_evd_set_unwaitable`(3DAT), `dat_evd_wait`(3DAT), `libdat`(3LIB), `attributes`(5)

Name dat_evd_dequeue – remove the first event from the Event Dispatcher event queue

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_evd_dequeue(
    IN     DAT_EVD_HANDLE     evd_handle,
    OUT    DAT_EVENT          *event
    )
```

Parameters

evd_handle Handle for an instance of the Event Dispatcher.

event Pointer to the Consumer-allocated structure that Provider fills with the event data.

Description

The `dat_evd_dequeue()` function removes the first event from the Event Dispatcher event queue and fills the Consumer allocated *event* structure with event data. The first element in this structure provides the type of the event; the rest provides the event-type-specific parameters. The Consumer should allocate an *event* structure big enough to hold any event that the Event Dispatcher can deliver.

For all events the Provider fills the `dat_event` that the Consumer allocates. So for all events, all fields of `dat_event` are OUT from the Consumer point of view. For `DAT_CONNECTION_REQUEST_EVENT`, the Provider creates a Connection Request whose *cr_handle* is returned to the Consumer in `DAT_CR_ARRIVAL_EVENT_DATA`. That object is destroyed by the Provider as part of `dat_cr_accept`(3DAT), `dat_cr_reject`(3DAT), or `dat_cr_handoff`(3DAT). The Consumer should not use *cr_handle* or any of its parameters, including *private_data*, after one of these operations destroys the Connection Request.

For `DAT_CONNECTION_EVENT_ESTABLISHED` for the Active side of connection establishment, the Provider returns the pointer for *private_data* and the *private_data_size*. For the Passive side, `DAT_CONNECTION_EVENT_ESTABLISHED` event *private_data* is not defined and *private_data_size* returns zero. The Provider is responsible for the memory allocation and deallocation for *private_data*. The *private_data* is valid until the Active side Consumer destroys the connected Endpoint (`dat_ep_free`(3DAT)), or transitions the Endpoint into Unconnected state so it is ready for the next connection. So while the Endpoint is in Connected, Disconnect Pending, or Disconnected state, the private_data of `DAT_CONNECTION_REQUEST_EVENT` is still valid for Active side Consumers.

Provider must pass to the Consumer the entire Private Data that the remote Consumer provided for `dat_ep_connect`(3DAT), `dat_ep_dup_connect`(3DAT), and `dat_cr_accept()`. If the Consumer provides more data than the Provider and Transport can support (larger than IA Attribute of *max_private_data_size*), `DAT_INVALID_PARAMETER` is returned for that operation.

The returned event that was posted from an Event Stream guarantees Consumers that all events that were posted from the same Event Stream prior to the returned event were already returned to a Consumer directly through a `dat_evd_dequeue()` or `dat_evd_wait(3DAT)` operation.

The ordering of events dequeued by overlapping calls to `dat_evd_wait()` or `dat_evd_dequeue()` is not specified.

Return Values

`DAT_SUCCESS`	The operation was successful. An event was returned to a Consumer.
`DAT_INVALID_HANDLE`	Invalid DAT handle; evd_handle is invalid.
`DAT_QUEUE_EMPTY`	There are no entries on the Event Dispatcher queue.
`DAT_INVALID_STATE`	One of the parameters was invalid for this operation. There is already a waiter on the EVD.

Usage

No matter how many contexts attempt to dequeue from an Event Dispatcher, each event is delivered exactly once. However, which Consumer receives which event is not defined. The Provider is not obligated to provide the first caller the first event unless it is the only caller. The Provider is not obligated to ensure that the caller receiving the first event executes earlier than contexts receiving later events.

Preservation of event ordering within an Event Stream is an important feature of the DAT Event Model. Consumers are cautioned that overlapping or concurrent calls to `dat_evd_dequeue()` from multiple contexts can undermine this ordering information. After multiple contexts are involved, the Provider can only guarantee the order that it delivers events into the EVD. The Provider cannot guarantee that they are processed in the correct order.

Although calling `dat_evd_dequeue()` does not cause a context switch, the Provider is under no obligation to prevent one. A context could successfully complete a dequeue, and then reach the end of its timeslice, before returning control to the Consumer code. Meanwhile, a context receiving a later event could be executing.

The Event ordering is preserved when dequeueing is serialized. Potential Consumer serialization methods include, but are not limited to, performing all dequeueing from a single context or protecting dequeueing by way of lock or semaphore.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Standard	uDAPL, 1.1, 1.2

See Also dat_cr_accept(3DAT), dat_cr_handoff(3DAT), dat_cr_reject(3DAT), dat_ep_connect(3DAT), dat_ep_dup_connect(3DAT), dat_ep_free(3DAT), dat_evd_wait(3DAT)libdat(3LIB), attributes(5)

Name dat_evd_disable – disable the Event Dispatcher

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_evd_disable(
    IN    DAT_EVD_HANDLE     evd_handle
    )
```

Parameters *evd_handle* Handle for an instance of Event Dispatcher.

Description The `dat_evd_disable()` function disables the Event Dispatcher so that the arrival of an event does not affect the associated CNO.

If the Event Dispatcher is already disabled, this operation is no-op.

Events arriving on this EVD might cause waiters on the associated CNO to be awakened after the return of this routine because an unblocking a CNO waiter is already "in progress" at the time this routine is called or returned.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *evd_handle* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	uDAPL, 1.1, 1.2

See Also `dat_evd_enable`(3DAT), `libdat`(3LIB), `attributes`(5)

Name dat_evd_enable – enable the Event Dispatcher

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_evd_enable(
    IN    DAT_EVD_HANDLE     evd_handle
    )
```

Parameters *evd_handle* Handle for an instance of Event Dispatcher.

Description The `dat_evd_enable()` function enables the Event Dispatcher so that the arrival of an event can trigger the associated CNO. The enabling and disabling EVD has no effect on direct waiters on the EVD. However, direct waiters effectively take ownership of the EVD, so that the specified CNO is not triggered even if is enabled.

If the Event Dispatcher is already enabled, this operation is no-op.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *evd_handle* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	uDAPL, 1.1, 1.2

See Also `dat_evd_disable(3DAT)`, `libdat(3LIB)`, `attributes(5)`

Name dat_evd_free – destroy an instance of the Event Dispatcher

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_evd_free (
    IN    DAT_EVD_HANDLE    evd_handle
    )
```

Parameters *evd_handle* Handle for an instance of the Event Dispatcher.

Description The `dat_evd_free()` function destroys a specified instance of the Event Dispatcher.

All events on the queue of the specified Event Dispatcher are lost. The destruction of the Event Dispatcher instance does not have any effect on any DAT Objects that originated an Event Stream that had fed events to the Event Dispatcher instance. There should be no event streams feeding the Event Dispatcher and no threads blocked on the Event Dispatcher when the EVD is being closed as at the time when it was created.

Use of the handle of the destroyed Event Dispatcher in any consequent operation fails.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *evd_handle* parameter is invalid

`DAT_INVALID_STATE` Invalid parameter. There are Event Streams associated with the Event Dispatcher feeding it.

Usage Consumers are advised to destroy all Objects that originate Event Streams that feed an instance of the Event Dispatcher before destroying it. An exception to this rule is Event Dispatchers of an IA.

Freeing an IA automatically destroys all Objects associated with it directly and indirectly, including Event Dispatchers.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.1, 1.2

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_evd_modify_cno – change the associated CNO for the Event Dispatcher

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_evd_modify_cno (
    IN    DAT_EVD_HANDLE    evd_handle,
    IN    DAT_CNO_HANDLE    cno_handle
    )
```

Parameters

evd_handle Handle for an instance of the Event Dispatcher.

cno_handle Handle for a CNO. The value of `DAT_NULL_HANDLE` specifies no CNO.

Description The `dat_evd_modify_cno()` function changes the associated CNO for the Event Dispatcher.

A Consumer can specify the value of `DAT_HANDLE_NULL` for *cno_handle* to associate not CNO with the Event Dispatcher instance.

Upon completion of the `dat_evd_modify_cno()` operation, the passed IN new CNO is used for notification. During the operation, an event arrival can be delivered to the old or new CNO. If Notification is generated by EVD, it is delivered to the new or old CNO.

If the EVD is enabled at the time `dat_evd_modify_cno()` is called, the Consumer must be prepared to collect a notification event on the EVD's old CNO as well as the new one. Checking immediately prior to calling `dat_evd_modify_cno()` is not adequate. A notification could have been generated after the prior check and before the completion of the change.

The Consumer can avoid the risk of missed notifications either by temporarily disabling the EVD, or by checking the prior CNO after invoking this operation. The Consumer can disable EVD before a `dat_evd_modify_cno()` call and enable it afterwards. This ensures that any notifications from the EVD are delivered to the new CNO only.

If this function is used to disassociate a CNO from the EVD, events arriving on this EVD might cause waiters on that CNO to awaken after returning from this routine because of unblocking a CNO waiter already "in progress" at the time this routine is called. If this is the case, the events causing that unblocking are present on the EVD upon return from the `dat_evd_modify_cno()` call and can be dequeued at that time

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` Invalid DAT handle.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.1, 1.2

See Also libdat(3LIB), attributes(5)

Name dat_evd_post_se – post Software event to the Event Dispatcher event queue

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_evd_post_se(
    IN        DAT_EVD_HANDLE     evd_handle,
    IN const  DAT_EVENT          *event
    )
```

Parameters

evd_handle Handle for an instance of the Event Dispatcher

event A pointer to a Consumer created Software Event.

Description The `dat_evd_post_se()` function posts Software events to the Event Dispatcher event queue. This is analogous to event arrival on the Event Dispatcher software Event Stream. The *event* that the Consumer provides adheres to the event format as defined in `<dat.h>`. The first element in the *event* provides the type of the event (`DAT_EVENT_TYPE_SOFTWARE`); the rest provide the event-type-specific parameters. These parameters are opaque to a Provider. Allocation and release of the memory referenced by the *event* pointer in a software event are the Consumer's responsibility.

There is no ordering between events from different Event Streams. All the synchronization issues between multiple Consumer contexts trying to post events to an Event Dispatcher instance simultaneously are left to a Consumer.

If the event queue is full, the operation is completed unsuccessfully and returns `DAT_QUEUE_FULL`. The *event* is not queued. The queue overflow condition does takes place and, therefore, the asynchronous Event Dispatcher is not effected.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *evd_handle* parameter is invalid.

`DAT_INVALID_PARAMETER` The *event* parameter is invalid.

`DAT_QUEUE_FULL` The Event Dispatcher queue is full.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.1, 1.2

See Also libdat(3LIB), attributes(5)

Name dat_evd_query – provide parameters of the Event Dispatcher,

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_evd_query (
    IN     DAT_EVD_HANDLE        evd_handle,
    IN     DAT_EVD_PARAM_MASK    evd_param_mask,
    OUT    DAT_EVD_PARAM         *evd_param
    )
```

Parameters

evd_handle Handle for an instance of Event Dispatcher.

evd_param_mask Mask for EVD parameters

evd_param Pointer to a Consumer-allocated structure that the Provider fills for Consumer-requested parameters.

Description The `dat_evd_query()` function provides to the Consumer parameters of the Event Dispatcher, including the state of the EVD (enabled/disabled). The Consumer passes in a pointer to the Consumer-allocated structures for EVD parameters that the Provider fills.

The *evd_param_mask* parameter allows Consumers to specify which parameters to query. The Provider returns values for *evd_param_mask* requested parameters. The Provider can return values for any of the other parameters.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *evd_handle* parameter is invalid.

`DAT_INVALID_PARAMETER` The *evd_param_mask* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.1, 1.2

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_evd_resize – modify the size of the event queue of Event Dispatcher

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_evd_resize(
    IN     DAT_EVD_HANDLE     evd_handle,
    IN     DAT_COUNT          evd_min_qlen
    )
```

Parameters

evd_handle Handle for an instance of Event Dispatcher.

evd_min_qlen New number of events the Event Dispatcher event queue must hold.

Description The `dat_evd_resize()` function modifies the size of the event queue of Event Dispatcher.

Resizing of Event Dispatcher event queue should not cause any incoming or current events on the event queue to be lost. If the number of entries on the event queue is larger then the requested evd_min_qlen, the operation can return `DAT_INVALID_STATE` and not change an instance of Event Dispatcher

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *evd_handle* parameter is invalid.

`DAT_INVALID_PARAMETER` The *evd_min_qlen* parameter is invalid

`DAT_INSUFFICIENT_RESOURCES` The operation failed due to resource limitations

`DAT_INVALID_STATE` Invalid parameter. The number of entries on the event queue of the Event Dispatcher exceeds the requested event queue length.

Usage This operation is useful when the potential number of events that could be placed on the event queue changes dynamically.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.1, 1.2

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_evd_set_unwaitable – transition the Event Dispatcher into an unwaitable state

Synopsis
```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_evd_set_unwaitable(
    IN    DAT_EVD_HANDLE     evd_handle
    )
```

Parameters *evd_handle* Handle for an instance of Event Dispatcher.

Description The `dat_evd_set_unwaitable()` transitions the Event Dispatcher into an unwaitable state. In this state, calls to `dat_evd_wait(3DAT)` return synchronously with a `DAT_INVALID_STATE` error, and threads already blocked in `dat_evd_wait()` are awakened and return with a `DAT_INVALID_STATE` error without any further action by the Consumer. The actual state of the Event Dispatcher is accessible through `dat_evd_query(3DAT)` and is `DAT_EVD_UNWAITABLE` after the return of this operation.

This call does not affect a CNO associated with this EVD at all. Events arriving on the EVD after it is set unwaitable still trigger the CNO (if appropriate), and can be retrieved with `dat_evd_dequeue(3DAT)`. Because events can arrive normally on the EVD, the EVD might overflow; the Consumer is expected to protect against this possibility.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *evd_handle* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	uDAPL, 1.1, 1.2

See Also `dat_evd_clear_unwaitable(3DAT)`, `dat_evd_dequeue(3DAT)`, `dat_evd_query(3DAT)`, `dat_evd_wait(3DAT)`, `libdat(3LIB)`, `attributes(5)`

Name dat_evd_wait – remove first event from the Event Dispatcher event queue

Synopsis
```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_evd_wait(
    IN     DAT_EVD_HANDLE     evd_handle,
    IN     DAT_TIMEOUT        timeout,
    IN     DAT_COUNT          threshold,
    OUT    DAT_EVENT          *event,
    OUT    DAT_COUNT          *nmore
    )
```

Parameters

evd_handle Handle for an instance of the Event Dispatcher.

timeout The duration of time, in microseconds, that the Consumer is willing to wait for the event.

threshold The number of events that should be on the EVD queue before the operation should return with `DAT_SUCCESS`. The threshold must be at least 1.

event Pointer to the Consumer-allocated structure that the Provider fills with the event data.

nmore The snapshot of the queue size at the time of the operation return.

Description The `dat_evd_wait()` function removes the first event from the Event Dispatcher event queue and fills the Consumer-allocated *event* structure with event data. The first element in this structure provides the type of the event; the rest provides the event type-specific parameters. The Consumer should allocate an event structure big enough to hold any event that the Event Dispatcher can deliver.

For all events, the Provider fills the *dat_event* that the Consumer allocates. Therefore, for all events, all fields of *dat_event* are OUT from the Consumer point of view. For `DAT_CONNECTION_REQUEST_EVENT`, the Provider creates a Connection Request whose *cr_handle* is returned to the Consumer in `DAT_CR_ARRIVAL_EVENT_DATA`. That object is destroyed by the Provider as part of `dat_cr_accept`(3DAT), `dat_cr_reject`(3DAT), or `dat_cr_handoff`(3DAT). The Consumer should not use *cr_handle* or any of its parameters, including *private_data*, after one of these operations destroys the Connection Request.

For `DAT_CONNECTION_EVENT_ESTABLISHED` for the Active side of connection establishment, the Provider returns the pointer for *private_data* and the *private_data_size*. For the Passive side, `DAT_CONNECTION_EVENT_ESTABLISHED` event *private_data* is not defined and *private_data_size* returns zero. The Provider is responsible for the memory allocation and deallocation for *private_data*. The *private_data* is valid until the Active side Consumer destroys the connected Endpoint (`dat_ep_free`(3DAT)), or transitions the Endpoint into Unconnected state so it is ready for the next connection. So, while the Endpoint is in

Connected, Disconnect Pending, or Disconnected state, the *private_data* of `DAT_CONNECTION_REQUEST_EVENT` is still valid for Active side Consumers.

Provider must pass to the Consumer the entire Private Data that the remote Consumer provided for `dat_ep_connect(3DAT)`, `dat_ep_dup_connect(3DAT)`, and `dat_cr_accept()`. If the Consumer provides more data than the Provider and Transport can support (larger than IA Attribute of *max_private_data_size*), `DAT_INVALID_PARAMETER` is returned for that operation.

A Consumer that blocks performing a `dat_evd_wait()` on an Event Dispatcher effectively takes exclusive ownership of that Event Dispatcher. Any other dequeue operation (`dat_evd_wait()` or `dat_evd_dequeue(3DAT)`) on the Event Dispatcher is rejected with a `DAT_INVALID_STATE` error code.

The CNO associated with the `evd_handle()` is not triggered upon event arrival if there is a Consumer blocked on `dat_evd_wait()` on this Event Dispatcher.

The *timeout* allows the Consumer to restrict the amount of time it is blocked waiting for the event arrival. The value of `DAT_TIMEOUT_INFINITE` indicates that the Consumer waits indefinitely for an event arrival. Consumers should use extreme caution in using this value.

When *timeout* value is reached and the number of events on the EVD queue is below the *threshold* value, the operation fails and returns `DAT_TIMEOUT_EXPIRED`. In this case, no event is dequeued from the EVD and the return value for the *event* argument is undefined. However, an *nmore* value is returned that specifies the snapshot of the number of the events on the EVD queue that is returned.

The *threshold* allows the Consumer to wait for a requested number of event arrivals prior to waking the Consumer. If the value of the *threshold* is larger than the Event Dispatcher queue length, the operation fails with the return `DAT_INVALID_PARAMETER`. If a non-positive value is specified for *threshold*, the operation fails and returns `DAT_INVALID_PARAMETER`.

If EVD is used by an Endpoint for a DTO completion stream that is configured for a Consumer-controlled event Notification (`DAT_COMPLETION_UNSIGNALLED_FLAG` or `DAT_COMPLETION_SOLICITED_WAIT_FLAG` for Receive Completion Type for Receives; `DAT_COMPLETION_UNSIGNALLED_FLAG` for Request Completion Type for Send, RDMA Read, RDMA Write and RMR Bind), the *threshold* value must be 1. An attempt to specify some other value for *threshold* for this case results in `DAT_INVALID_STATE`.

The returned value of *nmore* indicates the number of events left on the Event Dispatcher queue after the `dat_evd_wait()` returns. If the operation return value is `DAT_SUCCESS`, the *nmore* value is at least the value of (*threshold* -1). Notice that *nmore* is only a snapshot and the number of events can be changed by the time the Consumer tries to dequeue events with `dat_evd_wait()` with timeout of zero or with `dat_evd_dequeue()`.

For returns other than DAT_SUCCESS, DAT_TIMEOUT_EXPIRED, and DAT_INTERRUPTED_CALL, the returned value of *nmore* is undefined.

The returned event that was posted from an Event Stream guarantees Consumers that all events that were posted from the same Event Stream prior to the returned event were already returned to a Consumer directly through a dat_evd_dequeue() or dat_evd_wait() operation.

If the return value is neither DAT_SUCCESS nor DAT_TIMEOUT_EXPIRED, then returned values of *nmore* and event are undefined. If the return value is DAT_TIMEOUT_EXPIRED, then the return value of event is undefined, but the return value of *nmore* is defined. If the return value is DAT_SUCCESS, then the return values of *nmore* and *event* are defined.

If this function is called on an EVD in an unwaitable state, or if dat_evd_set_unwaitable(3DAT) is called on an EVD on which a thread is blocked in this function, the function returns with DAT_INVALID_STATE.

The ordering of events dequeued by overlapping calls to dat_evd_wait() or dat_evd_dequeue() is not specified.

Return Values

DAT_SUCCESS	The operation was successful. An event was returned to a Consumer.
DAT_INVALID_HANDLE	The *evd_handle* parameter is invalid.
DAT_INVALID_PARAMETER	The *timeout* or *threshold* parameter is invalid. For example, *threshold* is larger than the EVD's *evd_min_qlen*.
DAT_ABORT	The operation was aborted because IA was closed or EVD was destroyed
DAT_INVALID_STATE	One of the parameters was invalid for this operation. There is already a waiter on the EVD, or the EVD is in an unwaitable state.
DAT_TIMEOUT_EXPIRED	The operation timed out.
DAT_INTERRUPTED_CALL	The operation was interrupted by a signal.

Usage Consumers should be cautioned against using threshold combined with infinite *timeout*.

Consumers should not mix different models for control of unblocking a waiter. If the Consumer uses Notification Suppression or Solicited Wait to control the Notification events for unblocking a waiter, the *threshold* must be set to 1. If the Consumer uses *threshold* to control when a waiter is unblocked, DAT_COMPLETION_UNSIGNALLED_FLAG locally and DAT_COMPLETION_SOLICITED_WAIT remotely shall not be used. By default, all completions are Notification events.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	uDAPL, 1.1, 1.2

See Also dat_cr_accept(3DAT), dat_cr_handoff(3DAT), dat_cr_reject(3DAT), dat_ep_connect(3DAT), dat_ep_dup_connect(3DAT),dat_ep_free(3DAT), dat_evd_dequeue(3DAT), dat_evd_set_unwaitable(3DAT), libdat(3LIB), attributes(5)

Name dat_get_consumer_context – get Consumer context

Synopsis
```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_get_consumer_context (
    IN     DAT_HANDLE     dat_handle,
    OUT    DAT_CONTEXT    *context
    )
```

Parameters

dat_handle Handle for a DAT Object associated with *context*.

context Pointer to Consumer-allocated storage where the current value of the *dat_handle* context will be stored.

Description The `dat_get_consumer_context()` function gets the Consumer context from the specified *dat_handle*. The *dat_handle* can be one of the following handle types: `DAT_IA_HANDLE`, `DAT_EP_HANDLE`, `DAT_EVD_HANDLE`, `DAT_CR_HANDLE`, `DAT_RSP_HANDLE`, `DAT_PSP_HANDLE`, `DAT_PZ_HANDLE`, `DAT_LMR_HANDLE`, `DAT_RMR_HANDLE`, or `DAT_CNO_HANDLE`.

Return Values

`DAT_SUCCESS` The operation was successful. The Consumer context was successfully retrieved from the specified handle.

`DAT_INVALID_HANDLE` The *dat_handle* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.1, 1.2

See Also `dat_set_consumer_context(3DAT)`, `libdat(3LIB)`, `attributes(5)`

Name dat_get_handle_type – get handle type

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_get_handle_typet (
    IN     DAT_HANDLE         dat_handle,
    OUT    DAT_HANDLE_TYPE    *handle_type
    )
```

Parameters

dat_handle Handle for a DAT Object.

handle_type Type of the handle of *dat_handle*.

Description The `dat_get_handle_type()` function allows the Consumer to discover the type of a DAT Object using its handle.

The *dat_handle* can be one of the following handle types: `DAT_IA_HANDLE`, `DAT_EP_HANDLE`, `DAT_EVD_HANDLE`, `DAT_CR_HANDLE`, `DAT_RSP_HANDLE`, `DAT_PSP_HANDLE`, `DAT_PZ_HANDLE`, `DAT_LMR_HANDLE`, or `DAT_RMR_HANDLE`.

The *handle_type* is one of the following handle types: `DAT_HANDLE_TYPE_IA`, `DAT_HANDLE_TYPE_EP`, `DAT_HANDLE_TYPE_EVD`, `DAT_HANDLE_TYPE_CR`, `DAT_HANDLE_TYPE_PSP`, `DAT_HANDLE_TYPE_RSP`, `DAT_HANDLE_TYPE_PZ`, `DAT_HANDLE_TYPE_LMR`, `DAT_HANDLE_TYPE_RMR`, or `DAT_HANDLE_TYPE_CNO`.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *dat_handle* parameter is invalid.

Usage Consumers can use this operation to determine the type of Object being returned. This is needed for calling an appropriate query or any other operation on the Object handle.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.1, 1.2

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_ia_close – close an IA

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ia_close (
    IN    DAT_IA_HANDLE    ia_handle,
    IN    DAT_CLOSE_FLAGS  ia_flags
    )
```

Parameters

ia_handle Handle for an instance of a DAT IA.

ia_flags Flags for IA closure. Flag definitions are:

`DAT_CLOSE_ABRUPT_FLAG` Abrupt close. Abrupt cascading close of IA including all Consumer created DAT objects.

`DAT_CLOSE_GRACEFUL_FLAG` Graceful close. Closure is successful only if all DAT objects created by the Consumer have been freed before the graceful closure call.

Default value of `DAT_CLOSE_DEFAULT` = `DAT_CLOSE_ABRUPT_FLAG` represents abrupt closure of IA.

Description The `dat_ia_close()` function closes an IA (destroys an instance of the Interface Adapter).

The *ia_flags* specify whether the Consumer wants abrupt or graceful close.

The abrupt close does a phased, cascading destroy. All DAT Objects associated with an IA instance are destroyed. These include all the connection oriented Objects: public and reserved Service Points; Endpoints, Connection Requests, LMRs (including lmr_contexts), RMRs (including rmr_contexts), Event Dispatchers, CNOs, and Protection Zones. All the waiters on all CNOs, including the OS Wait Proxy Agents, are unblocked with the DAT_HANDLE_NULL handle returns for an unblocking EVD. All direct waiters on all EVDs are also unblocked and return with DAT_ABORT.

The graceful close does a destroy only if the Consumer has done a cleanup of all DAT objects created by the Consumer with the exception of the asynchronous EVD. Otherwise, the operation does not destroy the IA instance and returns the `DAT_INVALID_STATE`.

If async EVD was created as part of the of `dat_ia_open`(3DAT), `dat_ia_close()` must destroy it. If *async_evd_handle* was passed in by the Consumer at `dat_ia_open()`, this handle is not destroyed. This is applicable to both abrupt and graceful *ia_flags* values.

Because the Consumer did not create async EVD explicitly, the Consumer does not need to destroy it for graceful close to succeed.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INSUFFICIENT_RESOURCES`	The operation failed due to resource limitations. This is a catastrophic error.
`DAT_INVALID_HANDLE`	Invalid DAT handle; *ia_handle* is invalid.
`DAT_INVALID_PARAMETER`	Invalid parameter; *ia_flags* is invalid.
`DAT_INVALID_STATE`	Parameter in an invalid state. IA instance has Consumer-created objects associated with it.

Usage The `dat_ia_close()` function is the root cleanup method for the Provider, and, thus, all Objects.

Consumers are advised to explicitly destroy all Objects they created prior to closing the IA instance, but can use this function to clean up everything associated with an open instance of IA. This allows the Consumer to clean up in case of errors.

Note that an abrupt close implies destruction of EVDs and CNOs. Just as with explicit destruction of an EVD or CNO, the Consumer should take care to avoid a race condition where a Consumer ends up attempting to wait on an EVD or CNO that has just been deleted.

The techniques described in `dat_cno_free`(3DAT) and `dat_evd_free`(3DAT) can be used for these purposes.

If the Consumer desires to shut down the IA as quickly as possible, the Consumer can call dat_ia_close(abrupt) without unblocking CNO and EVD waiters in an orderly fashion. There is a slight chance that an invalidated DAT handle will cause a memory fault for a waiter. But this might be an acceptable behavior, especially if the Consumer is shutting down the process.

No provision is made for blocking on event completion or pulling events from queues.

This is the general cleanup and last resort method for Consumer recovery. An implementation must provide for successful completion under all conditions, avoiding hidden resource leakage (dangling memory, zombie processes, and so on) eventually leading to a reboot of the operating system.

The `dat_ia_close()` function deletes all Objects that were created using the IA handle.

The `dat_ia_close()` function can decrement a reference count for the Provider Library that is incremented by `dat_ia_open()` to ensure that the Provider Library cannot be removed when it is in use by a DAT Consumer.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.1, 1.2

See Also dat_cno_free(3DAT), dat_evd_free(3DAT), dat_ia_open(3DAT), libdat(3LIB), attributes(5)

Name dat_ia_open – open an Interface Adapter (IA)

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ia_open (
    IN const DAT_NAME_PTR      ia_name_ptr,
    IN       DAT_COUNT         async_evd_min_qlen,
    INOUT    DAT_EVD_HANDLE    *async_evd_handle,
    OUT      DAT_IA_HANDLE     *ia_handle
    )
```

Parameters

ia_name_ptr — Symbolic name for the IA to be opened. The name should be defined by the Provider registration.

If the name is prefixed by the string `RO_AWARE_`, then the prefix is removed prior to being passed down and the existence of the prefix indicates that the application has been coded to correctly deal with relaxed ordering constraints. If the prefix is not present and the platform on which the application is running is utilizing relaxed ordering, the open will fail with `DAT_INVALID_PARAMETER` (with `DAT_SUBTYPE_STATUS` of `DAT_INVALID_RO_COOKIE`). This setting also affects `dat_lmr_create`(3DAT).

async_evd_min_qlen — Minimum length of the Asynchronous Event Dispatcher queue.

async_evd_handle — Pointer to a handle for an Event Dispatcher for asynchronous events generated by the IA. This parameter can be `DAT_EVD_ASYNC_EXISTS` to indicate that there is already EVD for asynchronous events for this Interface Adapter or `DAT_HANDLE_NULL` for a Provider to generate EVD for it.

ia_handle — Handle for an open instance of a DAT IA. This handle is used with other functions to specify a particular instance of the IA.

Description The `dat_ia_open()` function opens an IA by creating an IA instance. Multiple instances (opens) of an IA can exist.

The value of `DAT_HANDLE_NULL` for *async_evd_handle* (**async_evd_handle* == `DAT_HANDLE_NULL`) indicates that the default Event Dispatcher is created with the requested *async_evd_min_qlen*. The *async_evd_handle* returns the handle of the created Asynchronous Event Dispatcher. The first Consumer that opens an IA must use `DAT_HANDLE_NULL` because no EVD can yet exist for the requested *ia_name_ptr*.

The Asynchronous Event Dispatcher (*async_evd_handle*) is created with no CNO (`DAT_HANDLE_NULL`). Consumers can change these values using `dat_evd_modify_cno`(3DAT). The Consumer can modify parameters of the Event Dispatcher using `dat_evd_resize`(3DAT) and `dat_evd_modify_cno()`.

The Provider is required to provide a queue size at least equal to *async_evd_min_qlen*, but is free to provide a larger queue size or dynamically enlarge the queue when needed. The Consumer can determine the actual queue size by querying the created Event Dispatcher instance.

If *async_evd_handle* is not `DAT_HANDLE_NULL`, the Provider does not create an Event Dispatcher for an asynchronous event and the Provider ignores the *async_evd_min_qlen* value. The *async_evd_handle* value passed in by the Consumer must be an asynchronous Event Dispatcher created for the same Provider (*ia_name_ptr*). The Provider does not have to check for the validity of the Consumer passed in *async_evd_handle*. It is the Consumer responsibility to guarantee that *async_evd_handle* is valid and for this Provider. How the *async_evd_handle* is passed between DAT Consumers is out of scope of the DAT specification. If the Provider determines that the Consumer-provided *async_evd_handle* is invalid, the operation fails and returns `DAT_INVALID_HANDLE`. The *async_evd_handle* remains unchanged, so the returned *async_evd_handle* is the same the Consumer passed in. All asynchronous notifications for the open instance of the IA are directed by the Provider to the Consumer passed in Asynchronous Event Dispatcher specified by *async_evd_handle*.

Consumer can specify the value of `DAT_EVD_ASYNC_EXISTS` to indicate that there exists an event dispatcher somewhere else on the host, in user or kernel space, for asynchronous event notifications. It is up to the Consumer to ensure that this event dispatcher is unique and unambiguous. A special handle may be returned for the Asynchronous Event Dispatcher for this scenario, `DAT_EVD_OUT_OF_SCOPE`, to indicate that there is a default Event Dispatcher assigned for this Interface Adapter, but that it is not in a scope where this Consumer may directly invoke it.

The Asynchronous Event Dispatcher is an Object of both the Provider and IA. Each Asynchronous Event Dispatcher bound to an IA instance is notified of all asynchronous events, such that binding multiple Asynchronous Event Dispatchers degrades performance by duplicating asynchronous event notifications for all Asynchronous Event Dispatchers. Also, transport and memory resources can be consumed per Event Dispatcher bound to an IA

As with all Event Dispatchers, the Consumer is responsible for synchronizing access to the event queue.

Valid IA names are obtained from `dat_registry_list_providers`(3DAT).

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INSUFFICIENT_RESOURCES`	The operation failed due to resource limitations.

DAT_INVALID_PARAMETER	Invalid parameter.
DAT_PROVIDER_NOT_FOUND	The specified provider was not registered in the registry.
DAT_INVALID_HANDLE	Invalid DAT handle; async_evd_handle is invalid.

Usage The dat_ia_open() function is the root method for the Provider, and, thus, all Objects. It is the root handle through which the Consumer obtains all other DAT handles. When the Consumer closes its handle, all its DAT Objects are released.

The dat_ia_open() function is the workhorse method that provides an IA instance. It can also initialize the Provider library or do any other registry-specific functions.

The dat_ia_open() function creates a unique handle for the IA to the Consumer. All further DAT Objects created for this Consumer reference this handle as their owner.

The dat_ia_open() function can use a reference count for the Provider Library to ensure that the Provider Library cannot be removed when it is in use by a DAT Consumer.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	uDAPL, 1.1, 1.2 (except RO_AWARE_)

See Also dat_evd_modify_cno(3DAT), dat_evd_resize(3DAT), dat_ia_close(3DAT), dat_registry_list_providers(3DAT), libdat(3LIB), attributes(5)

Name dat_ia_query – query an IA

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_ia_query (
    IN    DAT_IA_HANDLE            ia_handle,
    OUT   DAT_EVD_HANDLE           *async_evd_handle,
    IN    DAT_IA_ATTR_MASK         ia_attr_mask,
    OUT   DAT_IA_ATTR              *ia_attributes,
    IN    DAT_PROVIDER_ATTR_MASK   provider_attr_mask,
    OUT   DAT_PROVIDER_ATTR        *provider_attributes
    )
```

Parameters

ia_handle Handle for an open instance of an IA.

async_evd_handle Handle for an Event Dispatcher for asynchronous events generated by the IA.

ia_attr_mask Mask for the *ia_attributes*.

ia_attributes Pointer to a Consumer-allocated structure that the Provider fills with IA attributes.

provider_attr_mask Mask for the *provider_attributes*.

provider_attributes Pointer to a Consumer-allocated structure that the Provider fills with Provider attributes.

Description The `dat_ia_query()` functions provides the Consumer with the IA parameters, as well as the IA and Provider attributes. Consumers pass in pointers to Consumer-allocated structures for the IA and Provider attributes that the Provider fills.

The *ia_attr_mask* and *provider_attr_mask* parameters allow the Consumer to specify which attributes to query. The Provider returns values for requested attributes. The Provider can also return values for any of the other attributes.

Interface Adapter Attributes The IA attributes are common to all open instances of the IA. DAT defines a method to query the IA attributes but does not define a method to modify them.

If IA is multiported, each port is presented to a Consumer as a separate IA.

Adapter name: The name of the IA controlled by the Provider. The same as *ia_name_ptr*.

Vendor name: Vendor if IA hardware.

HW version major: Major version of IA hardware.

HW version minor:	Minor version of IA hardware.
Firmware version major:	Major version of IA firmware.
Firmware version minor:	Minor version of IA firmware.
IA_address_ptr:	An address of the interface Adapter.
Max EPs:	Maximum number of Endpoints that the IA can support. This covers all Endpoints in all states, including the ones used by the Providers, zero or more applications, and management.
Max DTOs per EP:	Maximum number of DTOs and RMR_binds that any Endpoint can support for a single direction. This means the maximum number of outstanding and in-progress Send, RDMA Read, RDMA Write DTOs, and RMR Binds at any one time for any Endpoint; and maximum number of outstanding and in-progress Receive DTOs at any one time for any Endpoint.
Max incoming RDMA Reads per EP:	Maximum number of RDMA Reads that can be outstanding per (connected) Endpoint with the IA as the target.
Max outgoing RDMA Reads per EP:	Maximum number of RDMA Reads that can be outstanding per (connected) Endpoint with the IA as the originator.
Max EVDs:	Maximum number of Event Dispatchers that an IA can support. An IA cannot support an Event Dispatcher directly, but indirectly by Transport-specific Objects, for example, Completion Queues for Infiniband™ and VI. The Event Dispatcher Objects can be shared among multiple Providers and similar Objects from other APIs, for example, Event Queues for uDAPL.
Max EVD queue size:	Maximum size of the EVD queue supported by an IA.
Max IOV segments per DTO:	Maximum entries in an IOV list that an IA supports. Notice that this number cannot be explicit but must be implicit to transport-specific Object entries. For example, for IB, it is the maximum number of

	scatter/gather entries per Work Request, and for VI it is the maximum number of data segments per VI Descriptor.
Max LMRs:	Maximum number of Local Memory Regions IA supports among all Providers and applications of this IA.
Max LMR block size:	Maximum contiguous block that can be registered by the IA.
Mac LMR VA:	Highest valid virtual address within the context of an LMR. Frequently, IAs on 32–bit architectures support only 32–bit local virtual addresses.
Max PZs:	Maximum number of Protection Zones that the IA supports.
Max MTU size:	Maximum message size supported by the IA
Max RDMA size:	Maximum RDMA size supported by the IA
Max RMRs:	Maximum number of RMRs an IA supports among all Providers and applications of this IA.
Max RMR target address:	Highest valid target address with the context of a local RMR. Frequently, IAs on 32–bit architectures support only 32–bit local virtual addresses.
Num transport attributes:	Number of transport-specific attributes.
Transport-specific attributes:	Array of transport-specific attributes. Each entry has the format of `DAT_NAMED_ATTR`, which is a structure with two elements. The first element is the name of the attribute. The second element is the value of the attribute as a string.
Num vendor attributes:	Number of vendor-specific attributes.
Vendor-specific attributes:	Array of vendor-specific attributes. Each entry has the format of `DAT_NAMED_ATTR`, which is a structure with two elements. The first element is the name of the attribute. The second element is the value of the attribute as a string.

DAPL Provider Attributes

The provider attributes are specific to the open instance of the IA. DAT defines a method to query Provider attributes but does not define a method to modify them.

Provider name:	Name of the Provider vendor.
Provider version major:	Major Version of uDAPL Provider.

Provider version minor: Minor Version of uDAPL Provider.

DAPL API version major: Major Version of uDAPL API supported.

DAPL API version minor: Minor Version of uDAPL API supported.

LMR memory types supported: Memory types that LMR Create supports for memory registration. This value is a union of LMR Memory Types DAT_MEM_TYPE_VIRTUAL, DAT_MEM_TYPE_LMR, and DAT_MEM_TYPE_SHARED_VIRTUAL that the Provider supports. All Providers must support the following Memory Types: DAT_MEM_TYPE_VIRTUAL, DAT_MEM_TYPE_LMR, and DAT_MEM_TYPE_SHARED_VIRTUAL.

IOV ownership: An enumeration flag that specifies the ownership of the local buffer description (IOV list) after post DTO returns. The three values are as follows:

- DAT_IOV_CONSUMER indicates that the Consumer has the ownership of the local buffer description after a post returns.
- DAT_IOV_PROVIDER_NOMOD indicates that the Provider still has ownership of the local buffer description of the DTO when the post DTO returns, but the Provider does not modify the buffer description.
- DAT_IOV_PROVIDER_MOD indicates that the Provider still has ownership of the local buffer description of the DTO when the post DTO returns and can modify the buffer description.

In any case, the Consumer obtains ownership of the local buffer description after the DTO transfer is completed and the Consumer is notified through a DTO completion event.

QOS supported: The union of the connection QOS supported by the Provider.

Completion flags supported: The following values for the completion flag DAT_COMPLETION_FLAGS are supported by the Provider: DAT_COMPLETION_SUPPRESS_FLAG, DAT_COMPLETION_UNSIGNALLED_FLAG, DAT_COMPLETION_SOLICITED_WAIT_FLAG, and DAT_COMPLETION_BARRIER_FENCE_FLAG.

Thread safety: Provider Library thread safe or not. The Provider Library is not required to be thread safe.

Max private data size: Maximum size of private data the Provider supports. This value is at least 64 bytes.

Multipathing support:	Capability of the Provider to support Multipathing for connection establishment.
EP creator for PSP:	Indicator for who can create an Endpoint for a Connection Request. For the Consumer it is `DAT_PSP_CREATES_EP_NEVER`. For the Provider it is `DAT_PSP_CREATES_EP_ALWAYS`. For both it is `DAT_PSP_CREATES_EP_IFASKED`. This attribute is used for Public Service Point creation.
PZ support:	Indicator of what kind of protection the Provider's PZ provides.
Optimal Buffer Alignment:	Local and remote DTO buffer alignment for optimal performance on the Platform. The `DAT_OPTIMAL_ALIGMNEMT` must be divisible by this attribute value. The maximum allowed value is `DAT_OPTIMAL_ALIGMNEMT`, or 256.
EVD stream merging support:	A 2D binary matrix where each row and column represent an event stream type. Each binary entry is 1 if the event streams of its row and column can be fed to the same EVD, and 0 otherwise. More than two different event stream types can feed the same EVD if for each pair of the event stream types the entry is 1. The Provider should support merging of all event stream types. The Consumer should check this attribute before requesting an EVD that merges multiple event stream types.
Num provider attributes:	Number of Provider-specific attributes.
Provider-specific attributes:	Array of Provider-specific attributes. Each entry has the format of `DAT_NAMED_ATTR`, which is a structure with two elements. The first element is the name of the attribute. The second element is the value of the attribute as a string.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INVALID_PARAMETER`	Invalid parameter;
`DAT_INVALID_HANDLE`	Invalid DAT handle; ia_handle is invalid.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	uDAPL, 1.1, 1.2

See Also libdat(3LIB), attributes(5)

Name dat_lmr_create – register a memory region with an IA

Synopsis
```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_lmr_create (
    IN     DAT_IA_HANDLE          ia_handle,
    IN     DAT_MEM_TYPE           mem_type,
    IN     DAT_REGION_DESCRIPTION region_description,
    IN     DAT_VLEN               length,
    IN     DAT_PZ_HANDLE          pz_handle,
    IN     DAT_MEM_PRIV_FLAGS     mem_privileges,
    OUT    DAT_LMR_HANDLE         *lmr_handle,
    OUT    DAT_LMR_CONTEXT        *lmr_context,
    OUT    DAT_RMR_CONTEXT        *rmr_context,
    OUT    DAT_VLEN               *registered_size,
    OUT    DAT_VADDR              *registered_address
    )
```

Parameters

ia_handle
Handle for an open instance of the IA.

mem_type
Type of memory to be registered. The following list outlines the memory type specifications.

`DAT_MEM_TYPE_VIRTUAL`
Consumer virtual memory.

Region description: A pointer to a contiguous user virtual range.

Length: Length of the Memory Region.

`DAT_MEM_TYPE_SO_VIRTUAL`
Consumer virtual memory with strong memory ordering. This type is a Solaris specific addition. If the *ia_handle* was opened without `RO_AWARE_` (see `dat_ia_open`(3DAT)), then type `DAT_MEM_TYPE_VIRTUAL` is implicitly converted to this type.

Region description: A pointer to a contiguous user virtual range.

Length: Length of the Memory Region.

`DAT_MEM_TYPE_LMR`
LMR.

Region description: An LMR_handle.

Length: Length parameter is ignored.

DAT_MEM_TYPE_SHARED_VIRTUAL

Shared memory region. All DAT Consumers of the same uDAPL Provider specify the same Consumer cookie to indicate who is sharing the shared memory region. This supports a peer-to-peer model of shared memory. All DAT Consumers of the shared memory must allocate the memory region as shared memory using Platform-specific primitives.

Region description: A structure with 2 elements, where the first one is of type DAT_LMR_COOKIE and is a unique identifier of the shared memory region, and the second one is a pointer to a contiguous user virtual range.

Length: Length of the Memory Region

region_description

Pointer to type-specific data describing the memory in the region to be registered. The type is derived from the *mem_type* parameter.

length

Length parameter accompanying the region_description.

pz_handle

Handle for an instance of the Protection Zone.

mem_privileges:

Consumer-requested memory access privileges for the registered local memory region. The Default value is DAT_MEM_PRIV_NONE_FLAG. The constant value DAT_MEM_PRIV_ALL_FLAG = 0x33, which specifies both Read and Write privileges, is also defined. Memory privilege definitions are as follows:

Local Read	DAT_MEM_PRIV_LOCAL_READ_FLAG	
	0x01	Local read access requested.
Local Write	DAT_MEM_PRIV_LOCAL_WRITE_FLAG	
	0x10	Local write access requested.
Remote Read	DAT_MEM_PRIV_REMOTE_READ_FLAG	
	0x02	Remote read access requested.
Remote Write	DAT_MEM_PRIV_REMOTE_WRITE_FLAG	
	0x20	Remote write access requested.

lmr_handle

Handle for the created instance of the LMR.

lmr_context

Context for the created instance of the LMR to use for DTO local buffers.

registered_size
Actual memory size registered by the Provider.

registered_address
Actual base address of the memory registered by the Provider.

Description The `dat_lmr_create()` function registers a memory region with an IA. The specified buffer must have been previously allocated and pinned by the uDAPL Consumer on the platform. The Provider must do memory pinning if needed, which includes whatever OS-dependent steps are required to ensure that the memory is available on demand for the Interface Adapter. uDAPL does not require that the memory never be swapped out; just that neither the hardware nor the Consumer ever has to deal with it not being there. The created *lmr_context* can be used for local buffers of DTOs and for binding RMRs, and *lmr_handle* can be used for creating other LMRs. For uDAPL the scope of the *lmr_context* is the address space of the DAT Consumer.

The return values of *registered_size* and *registered_address* indicate to the Consumer how much the contiguous region of Consumer virtual memory was registered by the Provider and where the region starts in the Consumer virtual address.

The *mem_type* parameter indicates to the Provider the kind of memory to be registered, and can take on any of the values defined in the table in the PARAMETERS section.

The *pz_handle* parameter allows Consumers to restrict local accesses to the registered LMR by DTOs.

`DAT_LMR_COOKIE` is a pointer to a unique identifier of the shared memory region of the `DAT_MEM_TYPE_SHARED_VIRTUAL` DAT memory type. The identifier is an array of 40 bytes allocated by the Consumer. The Provider must check the entire 40 bytes and shall not interpret it as a null-terminated string.

The return value of *rmr_context* can be transferred by the local Consumer to a Consumer on a remote host to be used for an RDMA DTO.

If *mem_privileges* does not specify remote Read and Write privileges, *rmr_context* is not generated and `NULL` is returned. No remote privileges are given for Memory Region unless explicitly asked for by the Consumer.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_UNSUFFICIENT_RESOURCES`	The operation failed due to resource limitations.
`DAT_INVALID_PARAMETER`	Invalid parameter.
`DAT_INVALID_HANDLE`	Invalid DAT handle.
`DAT_INVALID_STATE`	Parameter in an invalid state. For example, shared virtual buffer was not created shared by the platform.

`DAT_MODEL_NOT_SUPPORTED` The requested Model was not supported by the Provider. For example, requested Memory Type was not supported by the Provider.

Usage Consumers can create an LMR over the existing LMR memory with different Protection Zones and privileges using previously created IA translation table entries.

The Consumer should use *rmr_context* with caution. Once advertised to a remote peer, the *rmr_context* of the LMR cannot be invalidated. The only way to invalidate it is to destroy the LMR with `dat_lmr_free`(3DAT).

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	uDAPL, 1.1, 1.2 (except `DAT_MEM_TYPE_SO_VIRTUAL`)

See Also `dat_lmr_free`(3DAT), `libdat`(3LIB), `attributes`(5)

Name dat_lmr_free – destroy an instance of the LMR

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_lmr_free (
    IN    DAT_LMR_HANDLE    lmr_handle
    )
```

Parameters *lmr_handle:* Handle for an instance of LMR to be destroyed.

Description The `dat_lmr_free()` function destroys an instance of the LMR. The LMR cannot be destroyed if it is in use by an RMR. The operation does not deallocate the memory region or unpin memory on a host.

Use of the handle of the destroyed LMR in any subsequent operation except for `dat_lmr_free()` fails. Any DTO operation that uses the destroyed LMR after the `dat_lmr_free()` is completed shall fail and report a protection violation. The use of *rmr_context* of the destroyed LMR by a remote peer for an RDMA DTO results in an error and broken connection on which it was used. Any remote RDMA operation that uses the destroyed LMR rmr_context, whose Transport-specific request arrived to the local host after the `dat_lmr_free()` has completed, fails and reports a protection violation. Remote RDMA operation that uses the destroyed LMR *rmr_context*, whose Transport-specific request arrived to the local host prior to the `dat_lmr_free()` returns, might or might not complete successfully. If it fails, `DAT_DTO_ERR_REMOTE_ACCESS` is reported in `DAT_DTO_COMPLETION_STATUS` for the remote RDMA DTO and the connection is broken.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *lmr_handle* parameter is invalid.

`DAT_INVALID_STATE` Parameter in an invalid state; LMR is in use by an RMR instance.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.1, 1.2

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_lmr_query – provide LMR parameters

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_lmr_query (
    IN     DAT_LMR_HANDLE        lmr_handle,
    IN     DAT_LMR_PARAM_MASK    lmr_param_mask,
    OUT    DAT_LMR_PARAM         *lmr_param
    )
```

Parameters

lmr_handle Handle for an instance of the LMR.

lmr_param_mask Mask for LMR parameters.

lmr_param Pointer to a Consumer-allocated structure that the Provider fills with LMR parameters.

Description The `dat_lmr_query()` function provides the Consumer LMR parameters. The Consumer passes in a pointer to the Consumer-allocated structures for LMR parameters that the Provider fills.

The *lmr_param_mask* parameter allows Consumers to specify which parameters to query. The Provider returns values for *lmr_param_mask* requested parameters. The Provider can return values for any other parameters.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_PARAMETER` The *lmr_param_mask* function is invalid.

`DAT_INVALID_HANDLE` The *lmr_handle* function is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.1, 1.2

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_lmr_sync_rdma_read – synchronize local memory with RDMA read on non-coherent memory

Synopsis
```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_lmr_sync_rdma_read (
    IN DAT_IA_HANDLE ia_handle,
    IN const DAT_LMR_TRIPLET *local_segments,
    IN DAT_VLEN num_segments
    )
```

Parameters

ia_handle A handle for an open instance of the IA.

local_segments An array of buffer segments.

num_segments The number of segments in the *local_segments* argument.

Description The `dat_lmr_sync_rdma_read()` function makes memory changes visible to an incoming RDMA Read operation. This operation guarantees consistency by locally flushing the non-coherent cache prior to it being retrieved by remote peer RDMA read operations.

The `dat_lmr_sync_rdma_read()` function is needed if and only if the Provider attribute specifies that this operation is needed prior to an incoming RDMA Read operation. The Consumer must call `dat_lmr_sync_rdma_read()` after modifying data in a memory range in this region that will be the target of an incoming RDMA Read operation. The `dat_lmr_sync_rdma_read()` function must be called after the Consumer has modified the memory range but before the RDMA Read operation begins. The memory range that will be accessed by the RDMA read operation must be supplied by the caller in the *local_segments* array. After this call returns, the RDMA Read operation can safely see the modified contents of the memory range. It is permissible to batch synchronizations for multiple RDMA Read operations in a single call by passing a *local_segments* array that includes all modified memory ranges. The *local_segments* entries need not contain the same LMR and need not be in the same Protection Zone.

If the Provider attribute specifying that this operation is required attempts to read from a memory range that is not properly synchronized using `dat_lmr_sync_rdma_read()`, the returned contents are undefined.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The DAT handle is invalid.

`DAT_INVALID_PARAMETER` One of the parameters is invalid. For example, the address range for a local segment fell outside the boundaries of the corresponding Local Memory Region or the LMR handle was invalid.

Usage Determining when an RDMA Read will start and what memory range it will read is the Consumer's responsibility. One possibility is to have the Consumer that is modifying memory call `dat_lmr_sync_rdma_read()` and then post a Send DTO message that identifies the range in the body of the Send. The Consumer wanting to perform the RDMA Read can receive this message and know when it is safe to initiate the RDMA Read operation.

This call ensures that the Provider receives a coherent view of the buffer contents upon a subsequent remote RDMA Read operation. After the call completes, the Consumer can be assured that all platform-specific buffer and cache updates have been performed, and that the LMR range has consistency with the Provider hardware. Any subsequent write by the Consumer can void this consistency. The Provider is not required to detect such access.

The action performed on the cache before the RDMA Read depends on the cache type:

- I/O noncoherent cache will be invalidated.
- CPU noncoherent cache will be flushed.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.2

See Also `dat_lmr_sync_rdma_write(3DAT)`, `libdat(3LIB)`, `attributes(5)`

Name dat_lmr_sync_rdma_write – synchronize local memory with RDMA write on non-coherent memory

Synopsis
```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_lmr_sync_rdma_write (
    IN DAT_IA_HANDLE ia_handle,
    IN const DAT_LMR_TRIPLET *local_segments,
    IN DAT_VLEN num_segments
    )
```

Parameters

ia_handle A handle for an open instance of the IA.

local_segments An array of buffer segments.

num_segments The number of segments in the *local_segments* argument.

Description The `dat_lmr_sync_rdma_write()` function makes effects of an incoming RDMA Write operation visible to the Consumer. This operation guarantees consistency by locally invalidating the non-coherent cache whose buffer has been populated by remote peer RDMA write operations.

The `dat_lmr_sync_rdma_write()` function is needed if and only if the Provider attribute specifies that this operation is needed after an incoming RDMA Write operation. The Consumer must call `dat_lmr_sync_rdma_write()` before reading data from a memory range in this region that was the target of an incoming RDMA Write operation. The `dat_lmr_sync_rdma_write()` function must be called after the RDMA Write operation completes, and the memory range that was modified by the RDMA Write must be supplied by the caller in the *local_ segments* array. After this call returns, the Consumer may safely see the modified contents of the memory range. It is permissible to batch synchronizations of multiple RDMA Write operations in a single call by passing a *local_segments* array that includes all modified memory ranges. The *local_segments* entries need not contain the same LMR and need not be in the same Protection Zone.

The Consumer must also use `dat_lmr_sync_rdma_write()` when performing local writes to a memory range that was or will be the target of incoming RDMA writes. After performing the local write, the Consumer must call `dat_lmr_sync_rdma_write()` before the RDMA Write is initiated. Conversely, after an RDMA Write completes, the Consumer must call `dat_lmr_sync_rdma_write()` before performing a local write to the same range.

If the Provider attribute specifies that this operation is needed and the Consumer attempts to read from a memory range in an LMR without properly synchronizing using `dat_lmr_sync_rdma_write()`, the returned contents are undefined. If the Consumer attempts to write to a memory range without properly synchronizing, the contents of the memory range become undefined.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INVALID_HANDLE`	The DAT handle is invalid.
`DAT_INVALID_PARAMETER`	One of the parameters is invalid. For example, the address range for a local segment fell outside the boundaries of the corresponding Local Memory Region or the LMR handle was invalid.

Usage

Determining when an RDMA Write completes and determining which memory range was modified is the Consumer's responsibility. One possibility is for the RDMA Write initiator to post a Send DTO message after each RDMA Write that identifies the range in the body of the Send. The Consumer at the target of the RDMA Write can receive the message and know when and how to call `dat_lmr_sync_rdma_write()`.

This call ensures that the Provider receives a coherent view of the buffer contents after a subsequent remote RDMA Write operation. After the call completes, the Consumer can be assured that all platform-specific buffer and cache updates have been performed, and that the LMR range has consistency with the Provider hardware. Any subsequent read by the Consumer can void this consistency. The Provider is not required to detect such access.

The action performed on the cache before the RDMA Write depends on the cache type:

- I/O noncoherent cache will be flushed.
- CPU noncoherent cache will be invalidated.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.2

See Also

`dat_lmr_sync_rdma_read(3DAT)`, `libdat(3LIB)`, `attributes(5)`

Name dat_provider_fini – disassociate the Provider from a given IA name

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

void
    dat_provider_fini (
    IN    const DAT_PROVIDER_INFO    *provider_info
    )
```

Parameters *provider_info* The information that was provided when dat_provider_init was called.

Description A destructor the Registry calls on a Provider before it disassociates the Provider from a given IA name.

The Provider can use this method to undo any initialization it performed when `dat_provider_init(3DAT)` was called for the same IA name. The Provider's implementation of this method should call `dat_registry_remove_provider(3DAT)` to unregister its IA Name. If it does not, the Registry might remove the entry itself.

This method can be called for a given IA name at any time after all open instances of that IA are closed, and is certainly called before the Registry unloads the Provider library. However, it is not called more than once without an intervening call to `dat_provider_init()` for that IA name.

Return Values No values are returned.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	
Standard	uDAPL, 1.1, 1.2

See Also `dat_provider_init(3DAT)`, `dat_registry_remove_provider(3DAT)`, `libdat(3LIB)`, `attributes(5)`

Name dat_provider_init – locate the Provider in the Static Registry

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

void
    dat_provider_init (
    IN    const DAT_PROVIDER_INFO    *provider_info,
    IN    const char *               instance_data
    )
```

Parameters

provider_info The information that was provided by the Consumer to locate the Provider in the Static Registry.

instance_data The instance data string obtained from the entry found in the Static Registry for the Provider.

Description A constructor the Registry calls on a Provider before the first call to dat_ia_open(3DAT) for a given IA name when the Provider is auto-loaded. An application that explicitly loads a Provider on its own can choose to use dat_provider_init() just as the Registry would have done for an auto-loaded Provider.

The Provider's implementation of this method must call dat_registry_add_provider(3DAT), using the IA name in the provider_info.ia_name field, to register itself with the Dynamic Registry. The implementation must not register other IA names at this time. Otherwise, the Provider is free to perform any initialization it finds useful within this method.

This method is called before the first call to dat_ia_open() for a given IA name after one of the following has occurred:

- The Provider library was loaded into memory.
- The Registry called dat_provider_fini(3DAT) for that IA name.
- The Provider called dat_registry_remove_provider(3DAT) for that IA name (but it is still the Provider indicated in the Static Registry).

If this method fails, it should ensure that it does not leave its entry in the Dynamic Registry.

Return Values No values are returned.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Standard	uDAPL, 1.1, 1.2

See Also dat_ia_open(3DAT), dat_provider_fini(3DAT), dat_registry_add_provider(3DAT), dat_registry_remove_provider(3DAT), libdat(3LIB), attributes(5)

Name dat_psp_create – create a persistent Public Service Point

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_psp_create(
    IN    DAT_IA_HANDLE     ia_handle,
    IN    DAT_CONN_QUAL     conn_qual,
    IN    DAT_EVD_HANDLE    evd_handle,
    IN    DAT_PSP_FLAGS     psp_flags,
    OUT   DAT_PSP_HANDLE    *psp_handle
    )
```

Parameters

ia_handle Handle for an instance of DAT IA.

conn_qual Connection Qualifier of the IA on which the Public Service Point is listening.

evd_handle Event Dispatcher that provides the Connection Requested Events to the Consumer. The size of the event queue for the Event Dispatcher controls the size of the backlog for the created Public Service Point.

psp_flags Flag that indicates whether the Provider or Consumer creates an Endpoint per arrived Connection Request. The value of `DAT_PSP_PROVIDER` indicates that the Consumer wants to get an Endpoint from the Provider; a value of `DAT_PSP_CONSUMER` means the Consumer does not want the Provider to provide an Endpoint for each arrived Connection Request.

psp_handle Handle to an opaque Public Service Point.

Description The `dat_psp_create()` function creates a persistent Public Service Point that can receive multiple requests for connection and generate multiple Connection Request instances that are delivered through the specified Event Dispatcher in Notification events.

The `dat_psp_create()` function is blocking. When the Public Service Point is created, `DAT_SUCCESS` is returned and *psp_handle* contains a handle to an opaque Public Service Point Object.

There is no explicit backlog for a Public Service Point. Instead, Consumers can control the size of backlog through the queue size of the associated Event Dispatcher.

The *psp_flags* parameter allows Consumers to request that the Provider create an implicit Endpoint for each incoming Connection Request, or request that the Provider should not create one per Connection Request. If the Provider cannot satisfy the request, the operation shall fail and `DAT_MODEL_NOT_SUPPORTED` is returned.

All Endpoints created by the Provider have `DAT_HANDLE_NULL` for the Protection Zone and all Event Dispatchers. The Provider sets up Endpoint attributes to match the Active side connection request. The Consumer can change Endpoint parameters. Consumers should change Endpoint parameters, especially PZ and EVD, and are advised to change parameters for local accesses prior to the connection request acceptance with the Endpoint.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INSUFFICIENT_RESOURCES`	The operation failed due to resource limitations.
`DAT_INVALID_HANDLE`	The *ia_handle* or *evd_handle* parameter is invalid.
`DAT_INVALID_PARAMETER`	The *conn_qual* or *psp_flags* parameter is invalid.
`DAT_CONN_QUAL_IN_USE`	The specified Connection Qualifier was in use.
`DAT_MODEL_NOT_SUPPORTED`	The requested Model was not supported by the Provider.

Usage Two uses of a Public Service Point are as follows:

Model 1 For this model, the Provider manipulates a pool of Endpoints for a Public Service Point. The Provider can use the same pool for more than one Public Service Point.

- The DAT Consumer creates a Public Service Point with a *flag* set to `DAT_PSP_PROVIDER`.
- The Public Service Point does the following:
  - Collects native transport information reflecting a received Connection Reques
  - Creates an instance of Connection Reques
  - Creates a Connection Request Notice (event) that includes the Connection Request instance (thatwhich includes, among others, Public Service Point, its Connection Qualifier, Provider-generated Local Endpoint, and information about remote Endpoint)
  - Delivers the Connection Request Notice to the Consumer-specified target (CNO) *evd_handle*

    The Public Service Point is persistent and continues to listen for incoming requests for connection.
- Upon receiving a connection request, or at some time subsequent to that, the DAT Consumer can modify the provided local Endpoint to match the Connection Request and must either `accept()` or `reject()` the pending Connection Request.
- If accepted, the provided Local Endpoint is now in a "connected" state and is fully usable for this connection, pending only any native transport mandated RTU (ready-to-use) messages. This includes binding it to the IA port if that was not done previously. The Consumer is notified that the Endpoint is in Connected state by a Connection Established Event on the Endpoint *connect_evd_handle*.

- If rejected, control of the Local Endpoint point is returned back to the Provider and its *ep_handle* is no longer usable by the Consumer.

Model 2 For this model, the Consumer manipulates a pool of Endpoints. Consumers can use the same pool for more than one Service Point.

- DAT Consumer creates a Public Service Point with a *flag* set to `DAT_PSP_CONSUMER`.
- Public Service Point:
  - Collects native transport information reflecting a received Connection Request
  - Creates an instance of Connection Request
  - Creates a Connection Request Notice (event) that includes the Connection Request instance (which includes, among others, Public Service Point, its Connection Qualifier, Provider-generated Local Endpoint and information about remote Endpoint)
  - Delivers the Connection Request Notice to the Consumer-specified target (CNO) *evd_handle*

    The Public Service Point is persistent and continues to listen for incoming requests for connection.
- The Consumer creates a pool of Endpoints that it uses for accepting Connection Requests. Endpoints can be created and modified at any time prior to accepting a Connection Request with that Endpoint.
- Upon receiving a connection request or at some time subsequent to that, the DAT Consumer can modify its local Endpoint to match the Connection Request and must either `accept()` or `reject()` the pending Connection Request.
- If accepted, the provided Local Endpoint is now in a "connected" state and is fully usable for this connection, pending only any native transport mandated RTU messages. This includes binding it to the IA port if that was not done previously. The Consumer is notified that the Endpoint is in Connected state by a Connection Established Event on the Endpoint *connect_evd_handle*.
- If rejected, the Consumer does not have to provide any Endpoint for `dat_cr_reject`(3DAT).

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe
Standard	uDAPL, 1.1, 1.2

See Also `dat_cr_reject`(3DAT), `libdat`(3LIB), `attributes`(5)

Name dat_psp_create_any – create a persistent Public Service Point

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_psp_create_any(
    IN    DAT_IA_HANDLE     ia_handle,
    IN    DAT_CONN_QUAL     conn_qual,
    IN    DAT_EVD_HANDLE    evd_handle,
    IN    DAT_PSP_FLAGS     psp_flags,
    OUT   DAT_PSP_HANDLE    *psp_handle
    )
```

Parameters

ia_handle Handle for an instance of DAT IA.

conn_qual Connection Qualifier of the IA on which the Public Service Point is listening.

evd_handle Event Dispatcher that provides the Connection Requested Events to the Consumer. The size of the event queue for the Event Dispatcher controls the size of the backlog for the created Public Service Point.

psp_flags Flag that indicates whether the Provider or Consumer creates an Endpoint per arrived Connection Request. The value of `DAT_PSP_PROVIDER` indicates that the Consumer wants to get an Endpoint from the Provider; a value of `DAT_PSP_CONSUMER` means the Consumer does not want the Provider to provide an Endpoint for each arrived Connection Request.

psp_handle Handle to an opaque Public Service Point.

Description The `dat_psp_create_any()` function creates a persistent Public Service Point that can receive multiple requests for connection and generate multiple Connection Request instances that are delivered through the specified Event Dispatcher in Notification events.

The `dat_psp_create_any()` function allocates an unused Connection Qualifier, creates a Public Service point for it, and returns both the allocated Connection Qualifier and the created Public Service Point to the Consumer.

The allocated Connection Qualifier should be chosen from "nonprivileged" ports that are not currently used or reserved by any user or kernel Consumer or host ULP of the IA. The format of allocated Connection Qualifier returned is specific to IA transport type.

The `dat_psp_create_any()` function is blocking. When the Public Service Point is created, `DAT_SUCCESS` is returned, *psp_handle* contains a handle to an opaque Public Service Point Object, and *conn_qual* contains the allocated Connection Qualifier. When return is not `DAT_SUCCESS`, *psp_handle* and *conn_qual* return values are undefined.

There is no explicit backlog for a Public Service Point. Instead, Consumers can control the size of backlog through the queue size of the associated Event Dispatcher.

The *psp_flags* parameter allows Consumers to request that the Provider create an implicit Endpoint for each incoming Connection Request, or request that the Provider should not create one per Connection Request. If the Provider cannot satisfy the request, the operation shall fail and `DAT_MODEL_NOT_SUPPORTED` is returned.

All Endpoints created by the Provider have `DAT_HANDLE_NULL` for the Protection Zone and all Event Dispatchers. The Provider sets up Endpoint attributes to match the Active side connection request. The Consumer can change Endpoint parameters. Consumers should change Endpoint parameters, especially PZ and EVD, and are advised to change parameters for local accesses prior to the connection request acceptance with the Endpoint.

Return Values

`DAT_SUCCESS` — The operation was successful.

`DAT_INSUFFICIENT_RESOURCES` — The operation failed due to resource limitations.

`DAT_INVALID_HANDLE` — The *ia_handle* or *evd_handle* parameter is invalid.

`DAT_INVALID_PARAMETER` — The *conn_qual* or *psp_flags* parameter is invalid.

`DAT_CONN_QUAL_UNAVAILABLE` — No Connection Qualifiers available.

`DAT_MODEL_NOT_SUPPORTED` — The requested Model was not supported by the Provider.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	uDAPL, 1.1, 1.2

See Also

`libdat(3LIB)`, `attributes(5)`

Name dat_psp_free – destroy an instance of the Public Service Point

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_psp_free (
    IN    DAT_PSP_HANDLE    psp_handle
    )
```

Parameters *psp_handle* Handle for an instance of the Public Service Point.

Description The `dat_psp_free()` function destroys a specified instance of the Public Service Point.

Any incoming Connection Requests for the Connection Qualifier on the destroyed Service Point it had been listening on are automatically rejected by the Provider with the return analogous to the no listening Service Point.

The behavior of the Connection Requests in progress is undefined and left to an implementation. But it must be consistent. This means that either a Connection Requested Event has been generated for the Event Dispatcher associated with the Service Point, including the creation of the Connection Request instance, or the Connection Request is rejected by the Provider without any local notification.

This operation shall have no effect on previously generated Connection Requested Events. This includes Connection Request instances and, potentially, Endpoint instances created by the Provider.

The behavior of this operation with creation of a Service Point on the same Connection Qualifier at the same time is not defined. Consumers are advised to avoid this scenario.

Use of the handle of the destroyed Public Service Point in any consequent operation fails.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *psp_handle* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.1, 1.2

See Also libdat(3LIB), attributes(5)

Name dat_psp_query – provide parameters of the Public Service Point

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_psp_query (
    IN     DAT_PSP_HANDLE        psp_handle,
    IN     DAT_PSP_PARAM_MASK    psp_param_mask,
    OUT    DAT_PSP_PARAM         *psp_param
    )
```

Parameters

psp_handle Handle for an instance of Public Service Point.

psp_param_mask Mask for PSP parameters.

psp_param Pointer to a Consumer-allocated structure that Provider fills for Consumer-requested parameters.

Description The `dat_psp_query()` function provides to the Consumer parameters of the Public Service Point. Consumer passes in a pointer to the Consumer allocated structures for PSP parameters that Provider fills.

The *psp_param_mask* parameter allows Consumers to specify which parameters they would like to query. The Provider will return values for *psp_param_mask* requested parameters. The Provider may return the value for any of the other parameters.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *psp_handle* parameter is invalid.

`DAT_INVALID_PARAMETER` The *psp_param_mask* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	uDAPL, 1.1, 1.2

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_pz_create – create an instance of the Protection Zone

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_pz_create (
    IN     DAT_IA_HANDLE     ia_handle,
    OUT    DAT_PZ_HANDLE     *pz_handle
    )
```

Parameters

ia_handle Handle for an open instance of the IA.

pz_handle Handle for the created instance of Protection Zone.

Description The `dat_pz_create()` function creates an instance of the Protection Zone. The Protection Zone provides Consumers a mechanism for association Endpoints with LMRs and RMRs to provide protection for local and remote memory accesses by DTOs.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INSUFFICIENT_RESOURCES` The operation failed due to resource limitations.

`DAT_INVALID_PARAMETER` Invalid parameter.

`DAT_INVALID_HANDLE` The *ia_handle* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	uDAPL, 1.1, 1.2

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_pz_free – destroy an instance of the Protection Zone

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_pz_free (
    IN    DAT_PZ_HANDLE    pz_handle
    )
```

Parameters

pz_handle Handle for an instance of Protection Zone to be destroyed.

Description

The `dat_pz_free()` function destroys an instance of the Protection Zone. The Protection Zone cannot be destroyed if it is in use by an Endpoint, LMR, or RMR.

Use of the handle of the destroyed Protection Zone in any subsequent operation except for `dat_pz_free()` fails.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_STATE` Parameter in an invalid state. The Protection Zone was in use by Endpoint, LMR, or RMR instances.

`DAT_INVALID_HANDLE` The *pz_handle* parameter is invalid.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.1, 1.2

See Also

`libdat(3LIB)`, `attributes(5)`

Name dat_pz_query – provides parameters of the Protection Zone

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_pz_query (
    IN    DAT_PZ_HANDLE       pz_handle,
    IN    DAT_PZ_PARAM_MASK   pz_param_mask,
    OUT   DAT_PZ_PARAM        *pz_param
    )
```

Parameters

pz_handle: Handle for the created instance of the Protection Zone.

pz_param_mask: Mask for Protection Zone parameters.

pz_param: Pointer to a Consumer-allocated structure that the Provider fills with Protection Zone parameters.

Description The `dat_pz_query()` function provides the Consumer parameters of the Protection Zone. The Consumer passes in a pointer to the Consumer-allocated structures for Protection Zone parameters that the Provider fills.

The *pz_param_mask* parameter allows Consumers to specify which parameters to query. The Provider returns values for *pz_param_mask* requested parameters. The Provider can return values for any other parameters.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_PARAMETER` The *pz_param_mask* parameter is invalid.

`DAT_INVALID_HANDLE` The *pz_handle* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	uDAPL, 1.1, 1.2

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_registry_add_provider – declare the Provider with the Dynamic Registry

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_registry_add_provider (
    IN    const DAT_PROVIDER          *provider,
    IN    const DAT_PROVIDER_INFO     *provider_info
    )
```

Parameters

provider Self-description of a Provider.

provider_info Attributes of the Provider.

Description The Provider declares itself with the Dynamic Registry. Note that the caller can choose to register itself multiple times, for example once for each port. The choice of what to virtualize is up to the Provider. Each registration provides an Interface Adapter to DAT. Each Provider must have a unique name.

The same IA Name cannot be added multiple times. An attempt to register the same IA Name again results in an error with the return value `DAT_PROVIDER_ALREADY_REGISTERED`.

The contents of provider_info must be the same as those the Consumer uses in the call to `dat_ia_open`(3DAT) directly, or the ones provided indirectly defined by the header files with which the Consumer compiled.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INSUFFICIENT_RESOURCES` The maximum number of Providers was already registered.

`DAT_INVALID_PARAMETER` Invalid parameter.

`DAT_PROVIDER_ALREADY_REGISTERED` Invalid or nonunique name.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	
Standard	uDAPL, 1.1, 1.2

See Also `dat_ia_open`(3DAT), `libdat`(3LIB), `attributes`(5)

Name dat_registry_list_providers – obtain a list of available pProviders from the Static Registry

Synopsis

```
typedef struct dat_provider_info {
    char ia_name[DAT_NAME_MAX_LENGTH];
    DAT_UINT32      dapl_version_major;
    DAT_UINT32      dapl_version_minor;
    DAT_BOOLEAN     is_thread_safe;
    } DAT_PROVIDER_INFO;

cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_registry_list_providers (
    IN    DAT_COUNT          max_to_return,
    OUT   DAT_COUNT          *number_entries,
    OUT   DAT_PROVIDER_INFO  *(dat_provider_list[])
    )
```

Parameters

max_to_return Maximum number of entries that can be returned to the Consumer in the *dat_provider_list*.

number_entries The actual number of entries returned to the Consumer in the *dat_provider_list* if successful or the number of Providers available.

dat_provider_list Points to an array of `DAT_PROVIDER_INFO` pointers supplied by the Consumer. Each Provider's information will be copied to the destination specified.

Description The `dat_registry_list_providers()` function allows the Consumer to obtain a list of available Providers from the Static Registry. The information provided is the Interface Adapter name, the uDAPL/kDAPL API version supported, and whether the provided version is thread-safe. The Consumer can examine the attributes to determine which (if any) Interface Adapters it wants to open. This operation has no effect on the Registry itself.

The Registry can open an IA using a Provider whose *dapl_version_minor* is larger than the one the Consumer requests if no Provider entry matches exactly. Therefore, Consumers should expect that an IA can be opened successfully as long as at least one Provider entry returned by `dat_registry_list_providers()` matches the *ia_name*, *dapl_version_major*, and *is_thread_safe* fields exactly, and has a *dapl_version_minor* that is equal to or greater than the version requested.

If the operation is successful, the returned value is `DAT_SUCCESS` and *number_entries* indicates the number of entries filled by the registry in *dat_provider_list*.

If the operation is not successful, then *number_entries* returns the number of entries in the registry. Consumers can use this return to allocate *dat_provider_list* large enough for the registry entries. This number is just a snapshot at the time of the call and may be changed by the time of the next call. If the operation is not successful, then the content of *dat_provider_list* is not defined.

If *dat_provider_list* is too small, including pointing to `NULL` for the registry entries, then the operation fails with the return `DAT_INVALID_PARAMETER`.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_PARAMETER` Invalid parameter. For example, *dat_provider_list* is too small or `NULL`.

`DAT_INTERNAL_ERROR` Internal error. The DAT static registry is missing.

Usage `DAT_NAME_MAX_LENGTH` includes the null character for string termination.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	uDAPL, 1.1, 1.2

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_registry_remove_provider – unregister the Provider from the Dynamic Registry

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_registry_remove_provider  (
    IN       DAT_PROVIDER         *provider
    IN const DAT_PROVIDER_INFO    *provider_info
    )
```

Parameters

provider Self-description of a Provider.

provider_info Attributes of the Provider.

Description The Provider removes itself from the Dynamic Registry. It is the Provider's responsibility to complete its sessions. Removal of the registration only prevents new sessions.

The Provider cannot be removed while it is in use. An attempt to remove the Provider while it is in use results in an error with the return code `DAT_PROVIDER_IN_USE`.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_PARAMETER` Invalid parameter. The Provider was not found.

`DAT_PROVIDER_IN_USE` The Provider was in use.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	
Standard	uDAPL, 1.1, 1.2

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_rmr_bind – bind the RMR to the specified memory region within an LMR

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_rmr_bind(
    IN    DAT_RMR_HANDLE        rmr_handle,
    IN    DAT_LMR_TRIPLET       *lmr_triplet,
    IN    DAT_MEM_PRIV_FLAGS    mem_privileges,
    IN    DAT_EP_HANDLE         ep_handle,
    IN    DAT_RMR_COOKIE        user_cookie,
    IN    DAT_COMPLETION_FLAGS  completion_flags,
    OUT   DAT_RMR_CONTEXT       *rmr_context
    )
```

Parameters

rmr_handle Handle for an RMR instance.

lmr_triplet A pointer to an *lmr_triplet* that defines the memory region of the LMR.

mem_privileges Consumer-requested memory access privileges for the registered remote memory region. The Default value is `DAT_MEM_PRIV_NONE_FLAG`. The constant value `DAT_MEM_PRIV_ALL_FLAG` = 0x33, which specifies both Read and Write privileges, is also defined. Memory privilege definitions are as follows:

Remote Read `DAT_MEM_PRIV_REMOTE_READ_FLAG`

0x02 Remote read access requested.

Remote Write `DAT_MEM_PRIV_REMOTE_WRITE_FLAG`

0x20 Remote write access requested.

ep_handle Endpoint to which `dat_rmr_bind()` is posted.

user_cookie User-provided cookie that is returned to a Consumer at the completion of the `dat_rmr_bind()`. Can be `NULL`.

completion_flags Flags for RMR Bind. The default `DAT_COMPLETION_DEFAULT_FLAG` is 0. Flag definitions are as follows:

Completion Suppression `DAT_COMPLETION_SUPPRESS_FLAG`

0x01 Suppress successful Completion.

Notification of Completion `DAT_COMPLETION_UNSIGNALLED_FLAG`

0x04 Non-notification completion. Local Endpoint must be configured for Notification

			Suppression.
	Barrier Fence	`DAT_COMPLETION_BARRIER_FENCE_FLAG`	
		0x08	Request for Barrier Fence.
rmr_context	New *rmr_context* for the bound RMR suitable to be shared with a remote host.		

Description The `dat_rmr_bind()` function binds the RMR to the specified memory region within an LMR and provides the new *rmr_context* value. The `dat_rmr_bind()` operation is a lightweight asynchronous operation that generates a new *rmr_context*. The Consumer is notified of the completion of this operation through a *rmr_bind* Completion event on the *request_evd_handle* of the specified Endpoint *ep_handle*.

The return value of *rmr_context* can be transferred by local Consumer to a Consumer on a remote host to be used for an RDMA DTO. The use of *rmr_context* by a remote host for an RDMA DTO prior to the completion of the `dat_rmr_bind()` can result in an error and a broken connection. The local Consumer can ensure that the remote Consumer does not have *rmr_context* before `dat_rmr_bind()` is completed. One way is to "wait" for the completion `dat_rmr_bind()` on the *rmr_bind* Event Dispatcher of the specified Endpoint *ep_handle*. Another way is to send *rmr_context* in a Send DTO over the connection of the Endpoint *ep_handle*. The barrier-fencing behavior of the `dat_rmr_bind()` with respect to Send and RDMA DTOs ensures that a Send DTO does not start until `dat_rmr_bind()` completed.

The `dat_rmr_bind()` function automatically fences all Send, RDMA Read, and RDMA Write DTOs and `dat_rmr_bind()` operations submitted on the Endpoint *ep_handle* after the `dat_rmr_bind()`. Therefore, none of these operations starts until `dat_rmr_bind()` is completed.

If the RMR Bind fails after `dat_rmr_bind()` returns, connection of *ep_handle* is broken. The Endpoint transitions into a `DAT_EP_STATE_DISCONNECTED` state and the `DAT_CONNECTION_EVENT_BROKEN` event is delivered to the *connect_evd_handle* of the Endpoint.

The `dat_rmr_bind()` function employs fencing to ensure that operations sending the RMR Context on the same Endpoint as the bind specified cannot result in an error from the peer side using the delivered RMR Context too soon. One method, used by InfiniBand, is to ensure that none of these operations start on the Endpoint until after the bind is completed. Other transports can employ different methods to achieve the same goal.

Any RDMA DTO that uses the previous value of *rmr_context* after the `dat_rmr_bind()` is completed fail and report a protection violation.

By default, `dat_rmr_bind()` generates notification completions.

The *mem_privileges* parameter allows Consumers to restrict the type of remote accesses to the registered RMR by RDMA DTOs. Providers whose underlying Transports require that privileges of the requested RMR and the associated LMR match, that is

- Set RMR's `DAT_MEM_PRIV_REMOTE_READ_FLAG` requires that LMR's `DAT_MEM_PRIV_LOCAL_READ_FLAG` is also set,
- Set RMR's `DAT_MEM_PRIV_REMOTE_WRITE_FLAG` requires that LMR's `DAT_MEM_PRIV_LOCAL_WRITE_FLAG` is also set,

or the operation fails and returns `DAT_PRIVILEGES_VIOLATION`.

In the *lmr_triplet*, the value of *length* of zero means that the Consumer does not want to associate an RMR with any memory region within the LMR and the return value of *rmr_context* for that case is undefined.

The completion of the posted RMR Bind is reported to the Consumer asynchronously through a DTO Completion event based on the specified *completion_flags* value. The value of `DAT_COMPLETION_UNSIGNALLED_FLAG` is only valid if the Endpoint Request Completion Flags `DAT_COMPLETION_UNSIGNALLED_FLAG`. Otherwise, `DAT_INVALID_PARAMETER` is returned.

The *user_cookie* parameter allows Consumers to have unique identifiers for each `dat_rmr_bind()`. These identifiers are completely under user control and are opaque to the Provider. The Consumer is not required to ensure the uniqueness of the *user_cookie* value. The *user_cookie* is returned to the Consumer in the *rmr_bind* Completion event for this operation.

The operation is valid for the Endpoint in the `DAT_EP_STATE_CONNECTED` and `DAT_EP_STATE_DISCONNECTED` states. If the operation returns successfully for the Endpoint in `DAT_EP_STATE_DISCONNECTED` state, the posted RMR Bind is immediately flushed to *request_evd_handle*.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INSUFFICIENT_RESOURCES`	The operation failed due to resource limitations.
`DAT_INVALID_PARAMETER`	Invalid parameter. For example, the *target_address* or *segment_length* exceeded the limits of the existing LMR.
`DAT_INVALID_HANDLE`	Invalid DAT handle.
`DAT_INVALID_STATE`	Parameter in an invalid state. Endpoint was not in the a `DAT_EP_STATE_CONNECTED` or `DAT_EP_STATE_DISCONNECTED` state.
`DAT_MODEL_NOT_SUPPORTED`	The requested Model was not supported by the Provider.
`DAT_PRIVILEGES_VIOLATION`	Privileges violation for local or remote memory access.
`DAT_PROTECTION_VIOLATION`	Protection violation for local or remote memory access.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.1, 1.2

See Also libdat(3LIB), attributes(5)

Name dat_rmr_create – create an RMR for the specified Protection Zone

Synopsis
```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_rmr_create(
    IN     DAT_PZ_HANDLE      pz_handle,
    OUT    DAT_RMR_HANDLE     *rmr_handle
    )
```

Parameters

pz_handle Handle for an instance of the Protection Zone.

rmr_handle Handle for the created instance of an RMR.

Description The `dat_rmr_create()` function creates an RMR for the specified Protection Zone. This operation is relatively heavy. The created RMR can be bound to a memory region within the LMR through a lightweight `dat_rmr_bind(3DAT)` operation that generates *rmr_context*.

If the operation fails (does not return `DAT_SUCCESS`), the return values of *rmr_handle* are undefined and Consumers should not use them.

The *pz_handle* parameter provide Consumers a way to restrict access to an RMR by authorized connection only.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INSUFFICIENT_RESOURCES` The operation failed due to resource limitations.

`DAT_INVALID_HANDLE` The *pz_handle* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	uDAPL, 1.1, 1.2

See Also `dat_rmr_bind(3DAT)`, `libdat(3LIB)`, `attributes(5)`

Name dat_rmr_free – destroy an instance of the RMR

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_rmr_free (
    IN    DAT_RMR_HANDLE    rmr_handle
    )
```

Parameters

rmr_handle Handle for an instance of the RMR to be destroyed.

Description

The `dat_rmr_free()` function destroys an instance of the RMR.

Use of the handle of the destroyed RMR in any subsequent operation except for the `dat_rmr_free()` fails. Any remote RDMA operation that uses the destroyed RMR *rmr_context*, whose Transport-specific request arrived to the local host after the `dat_rmr_free()` has completed, fails and reports a protection violation. Remote RDMA operation that uses the destroyed RMR *rmr_context*, whose Transport-specific request arrived to the local host prior to the `dat_rmr_free()` return, might or might not complete successfully. If it fails, `DAT_DTO_ERR_REMOTE_ACCESS` is reported in `DAT_DTO_COMPLETION_STATUS` for the remote RDMA DTO and the connection is broken.

The `dat_rmr_free()` function is allowed on either bound or unbound RMR. If RMR is bound, `dat_rmr_free()` unbinds (free HCA TPT and other resources and whatever else binds with length of 0 should do), and then free RMR.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *rmr_handle* handle is invalid.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.1, 1.2

See Also

`libdat(3LIB)`, `attributes(5)`

Name dat_rmr_query – provide RMR parameters

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_rmr_query (
    IN     DAT_RMR_HANDLE        rmr_handle,
    IN     DAT_RMR_PARAM_MASK    rmr_param_mask,
    OUT    DAT_RMR_PARAM         *rmr_param
    )
```

Parameters

rmr_handle Handle for an instance of the RMR.

rmr_param_mask Mask for RMR parameters.

rmr_param Pointer to a Consumer-allocated structure that the Provider fills with RMR parameters.

Description The `dat_rmr_query()` function provides RMR parameters to the Consumer. The Consumer passes in a pointer to the Consumer-allocated structures for RMR parameters that the Provider fills.

The *rmr_param_mask* parameter allows Consumers to specify which parameters to query. The Provider returns values for *rmr_param_mask* requested parameters. The Provider can return values for any other parameters.

Not all parameters can have a value at all times. For example, *lmr_handle*, *target_address*, *segment_length*, *mem_privileges*, and *rmr_context* are not defined for an unbound RMR.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_PARAMETER` The *rmr_param_mask* parameter is invalid.

`DAT_INVALID_HANDLE` The *mr_handle* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.1, 1.2

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_rsp_create – create a Reserved Service Point

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_rsp_create (
    IN    DAT_IA_HANDLE    ia_handle,
    IN    DAT_CONN_QUAL    conn_qual,
    IN    DAT_EP_HANDLE    ep_handle,
    IN    DAT_EVD_HANDLE   evd_handle,
    OUT   DAT_RSP_HANDLE   *rsp_handle
    )
```

Parameters

ia_handle Handle for an instance of DAT IA.

conn_qual Connection Qualifier of the IA the Reserved Service Point listens to.

ep_handle Handle for the Endpoint associated with the Reserved Service Point that is the only Endpoint that can accept a Connection Request on this Service Point. The value `DAT_HANDLE_NULL` requests the Provider to associate a Provider-created Endpoint with this Service Point.

evd_handle The Event Dispatcher to which an event of Connection Request arrival is generated.

rsp_handle Handle to an opaque Reserved Service Point.

Description The `dat_rsp_create()` function creates a Reserved Service Point with the specified Endpoint that generates, at most, one Connection Request that is delivered to the specified Event Dispatcher in a Notification event.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INSUFFICIENT_RESOURCES` The operation failed due to resource limitations.

`DAT_INVALID_HANDLE` The *ia_handle*, *evd_handle*, or *ep_handle* parameter is invalid.

`DAT_INVALID_PARAMETER` The *conn_qual* parameter is invalid.

`DAT_INVALID_STATE` Parameter in an invalid state. For example, an Endpoint was not in the Idle state.

`DAT_CONN_QUAL_IN_USE` Specified Connection Qualifier is in use.

Usage The usage of a Reserve Service Point is as follows:

- The DAT Consumer creates a Local Endpoint and configures it appropriately.
- The DAT Consumer creates a Reserved Service Point specifying the Local Endpoint.
- The Reserved Service Point performs the following:

- Collects native transport information reflecting a received Connection Request.
- Creates a Pending Connection Request.
- Creates a Connection Request Notice (event) that includes the Pending Connection Request (which includes, among others, Reserved Service Point Connection Qualifier, its Local Endpoint, and information about remote Endpoint).
- Delivers the Connection Request Notice to the Consumer-specified target (CNO) *evd_handle*. The Local Endpoint is transitioned from Reserved to Passive Connection Pending state.

- Upon receiving a connection request, or at some time subsequent to that, the DAT Consumer must either `accept()` or `reject()` the Pending Connection Request.
- If accepted, the original Local Endpoint is now in a *Connected* state and fully usable for this connection, pending only native transport mandated RTU messages. This includes binding it to the IA port if that was not done previously. The Consumer is notified that the Endpoint is in a *Connected* state by a Connection Established Event on the Endpoint *connect_evd_handle*.
- If rejected, the Local Endpoint point transitions into *Unconnected* state. The DAT Consumer can elect to destroy it or reuse it for other purposes.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	uDAPL, 1.1, 1.2

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_rsp_free – destroy an instance of the Reserved Service Point

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_rsp_free (
    IN    DAT_RSP_HANDLE    rsp_handle
    )
```

Parameters *rsp_handle* Handle for an instance of the Reserved Service Point.

Description The `dat_rsp_free()` function destroys a specified instance of the Reserved Service Point.

Any incoming Connection Requests for the Connection Qualifier on the destroyed Service Point was listening on are automatically rejected by the Provider with the return analogous to the no listening Service Point.

The behavior of the Connection Requests in progress is undefined and left to an implementation, but it must be consistent. This means that either a Connection Requested Event was generated for the Event Dispatcher associated with the Service Point, including the creation of the Connection Request instance, or the Connection Request is rejected by the Provider without any local notification.

This operation has no effect on previously generated Connection Request Event and Connection Request.

The behavior of this operation with creation of a Service Point on the same Connection Qualifier at the same time is not defined. Consumers are advised to avoid this scenario.

For the Reserved Service Point, the Consumer-provided Endpoint reverts to Consumer control. Consumers shall be aware that due to a race condition, this Reserved Service Point might have generated a Connection Request Event and passed the associated Endpoint to a Consumer in it.

Use of the handle of the destroyed Service Point in any consequent operation fails.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *rsp_handle* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Standard	uDAPL, 1.1, 1.2

See Also libdat(3LIB), attributes(5)

Name dat_rsp_query – provide parameters of the Reserved Service Point

Synopsis
```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_rsp_query (
    IN     DAT_RSP_HANDLE        rsp_handle,
    IN     DAT_RSP_PARAM_MASK    rsp_param_mask,
    OUT    DAT_RSP_PARAM         *rsp_param
    )
```

Parameters

rsp_handle Handle for an instance of Reserved Service Point

rsp_param_mask Mask for RSP parameters.

rsp_param Pointer to a Consumer-allocated structure that the Provider fills for Consumer-requested parameters.

Description The `dat_rsp_query()` function provides to the Consumer parameters of the Reserved Service Point. The Consumer passes in a pointer to the Consumer-allocated structures for RSP parameters that the Provider fills.

The *rsp_param_mask* parameter allows Consumers to specify which parameters to query. The Provider returns values for *rsp_param_mask* requested parameters. The Provider can return values for any other parameters.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *rsp_handle* parameter is invalid.

`DAT_INVALID_PARAMETER` The *rsp_param_mask* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	uDAPL, 1.1, 1.2

See Also `libdat(3LIB)`, `attributes(5)`

Name dat_set_consumer_context – set Consumer context

Synopsis
```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_set_consumer_context (
    IN     DAT_HANDLE     dat_handle,
    IN     DAT_CONTEXT    context
    )
```

Parameters

dat_handle Handle for a DAT Object associated with *context*.

context Consumer context to be stored within the associated *dat_handle*. The Consumer context is opaque to the uDAPL Provider. `NULL` represents no context.

Description The `dat_set_consumer_context()` function associates a Consumer context with the specified *dat_handle*. The *dat_handle* can be one of the following handle types: `DAT_IA_HANDLE`, `DAT_EP_HANDLE`, `DAT_EVD_HANDLE`, `DAT_CR_HANDLE`, `DAT_RSP_HANDLE`, `DAT_PSP_HANDLE`, `DAT_PZ_HANDLE`, `DAT_LMR_HANDLE`, `DAT_RMR_HANDLE`, or `DAT_CNO_HANDLE`.

Only a single Consumer context is provided for any *dat_handle*. If there is a previous Consumer context associated with the specified handle, the new context replaces the old one. The Consumer can disassociate the existing context by providing a NULL pointer for the *context*. The Provider makes no assumptions about the contents of *context*; no check is made on its value. Furthermore, the Provider makes no attempt to provide any synchronization for access or modification of the *context*.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_PARAMETER` The *context* parameter is invalid.

`DAT_INVALID_HANDLE` The *dat_handle* parameter is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.1, 1.2

See Also `dat_get_consumer_context(3DAT)`, `libdat(3LIB)`, `attributes(5)`

Name dat_srq_create – create an instance of a shared receive queue

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_srq_create (
        IN      DAT_IA_HANDLE       ia_handle,
        IN      DAT_PZ_HANDLE       pz_handle,
        IN      DAT_SRQ_ATTR        *srq_attr,
        OUT     DAT_SRQ_HANDLE      *srq_handle
    )
```

Parameters

ia_handle A handle for an open instance of the IA to which the created SRQ belongs.

pz_handle A handle for an instance of the Protection Zone.

srq_attr A pointer to a structure that contains Consumer-requested SRQ attributes.

srq_handle A handle for the created instance of a Shared Receive Queue.

Description The `dat_srq_create()` function creates an instance of a Shared Receive Queue (SRQ) that is provided to the Consumer as *srq_handle*. If the value of `DAT_RETURN` is not `DAT_SUCCESS`, the value of *srq_handle* is not defined.

The created SRQ is unattached to any Endpoints.

The Protection Zone *pz_handle* allows Consumers to control what local memory can be used for the Recv DTO buffers posted to the SRQ. Only memory referred to by LMRs of the posted Recv buffers that match the SRQ Protection Zone can be accessed by the SRQ.

The *srq_attributes* argument specifies the initial attributes of the created SRQ. If the operation is successful, the created SRQ will have the queue size at least *max_recv_dtos* and the number of entries on the posted Recv scatter list of at lease *max_recv_iov*. The created SRQ can have the queue size and support number of entries on post Recv buffers larger than requested. Consumer can query SRQ to find out the actual supported queue size and maximum Recv IOV.

The Consumer must set *low_watermark* to `DAT_SRQ_LW_DEFAULT` to ensure that an asynchronous event will not be generated immediately, since there are no buffers in the created SRQ. The Consumer should set the Maximum Receive DTO attribute and the Maximum number of elements in IOV for posted buffers as needed.

When an associated EP tries to get a buffer from SRQ and there are no buffers available, the behavior of the EP is the same as when there are no buffers on the EP Recv Work Queue.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INSUFFICIENT_RESOURCES` The operation failed due to resource limitations.

`DAT_INVALID_HANDLE`	Either *ia_handle* or *pz_handle* is an invalid DAT handle.
`DAT_INVALID_PARAMETER`	One of the parameters is invalid. Either one of the requested SRQ attributes was invalid or a combination of attributes is invalid.
`DAT_MODEL_NOT_SUPPORTED`	The requested Model was not supported by the Provider.

Usage SRQ is created by the Consumer prior to creation of the EPs that will be using it. Some Providers might restrict whether multiple EPs that share a SRQ can have different Protection Zones. Check the *srq_ep_pz_difference_support* Provider attribute. The EPs that use SRQ might or might not use the same *recv_evd*.

Since a Recv buffer of SRQ can be used by any EP that is using SRQ, the Consumer should ensure that the posted Recv buffers are large enough to receive an incoming message on any of the EPs.

If Consumers do not want to receive an asynchronous event when the number of buffers in SRQ falls below the Low Watermark, they should leave its value as `DAT_SRQ_LW_DEFAULT`. If Consumers do want to receive a notification, they can set the value to the desired one by calling `dat_srq_set_lw`(3DAT).

SRQ allows the Consumer to use fewer Recv buffers then posting the maximum number of buffers for each connection. If the Consumer can upper bound the number of incoming messages over all connections whose local EP is using SRQ, then instead of posting this maximum for each connection the Consumer can post them for all connections on SRQ. For example, the maximum utilized link bandwidth divided over the message size can be used for an upper bound.

Depending on the underlying Transport, one or more messages can arrive simultaneously on an EP that is using SRQ. Thus, the same EP can have multiple Recv buffers in its possession without these buffers being on SRQ or *recv_evd*.

Since Recv buffers can be used by multiple connections of the local EPs that are using SRQ, the completion order of the Recv buffers is no longer guaranteed even when they use of the same *recv_evd*. For each connection the Recv buffers completion order is guaranteed to be in the order of the posted matching Sends to the other end of the connection. There is no ordering guarantee that Receive buffers will be returned in the order they were posted even if there is only a single connection (Endpoint) associated with the SRQ. There is no ordering guarantee between different connections or between different *recv_evd*s.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe
Standard	uDAPL, 1.2

See Also dat_srq_free(3DAT), dat_srq_post_recv(3DAT), dat_srq_query(3DAT), dat_srq_resize(3DAT), dat_srq_set_lw(3DAT), libdat(3LIB), attributes(5)

Name dat_srq_free – destroy an instance of the shared receive queue

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_srq_free (
    IN      DAT_SRQ_HANDLE      srq_handle
    )
```

Parameters *srq_handle* A handle for an instance of SRQ to be destroyed.

Description The `dat_srq_free()` function destroys an instance of the SRQ. The SRQ cannot be destroyed if it is in use by an EP.

It is illegal to use the destroyed handle in any consequent operation.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *srq_handle* argument is an invalid DAT handle.

`DAT_SRQ_IN_USE` The Shared Receive Queue can not be destroyed because it is in still associated with an EP instance.

Usage If the Provider detects the use of a deleted object handle, it should return `DAT_INVALID_HANDLE`. The Provider should avoid assigning the used handle as long as possible. Once reassigned the handle is no longer a handle of a destroyed object.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.2

See Also `dat_srq_create(3DAT)`, `dat_srq_post_recv(3DAT)`, `dat_srq_query(3DAT)`, `dat_srq_resize(3DAT)`, `dat_srq_set_lw(3DAT)`, `libdat(3LIB)`, `attributes(5)`

Name dat_srq_post_recv – add receive buffers to shared receive queue

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_srq_post_recv (
    IN      DAT_SRQ_HANDLE      srq_handle,
    IN      DAT_COUNT           num_segments,
    IN      DAT_LMR_TRIPLET     *local_iov,
    IN      DAT_DTO_COOKIE      user_cookie
    )
```

Parameters

srq_handle A handle for an instance of the SRQ.

num_segments The number of *lmr_triplets* in *local_iov*. Can be 0 for receiving a zero-size message.

local_iov An I/O Vector that specifies the local buffer to be filled. Can be `NULL` for receiving a zero-size message.

user_cookie A user-provided cookie that is returned to the Consumer at the completion of the Receive DTO. Can be `NULL`.

Description The `dat_srq_post_recv()` function posts the receive buffer that can be used for the incoming message into the *local_iov* by any connected EP that uses SRQ.

The *num_segments* argument specifies the number of segments in the *local_iov*. The *local_iov* segments are filled in the I/O Vector order until the whole message is received. This ensures that all the front segments of the *local_iov* I/O Vector are completely filled, only one segment is partially filled, if needed, and all segments that follow it are not filled at all. The actual order of segment fillings is left to the implementation.

The *user_cookie* argument allows Consumers to have unique identifiers for each DTO. These identifiers are completely under user control and are opaque to the Provider. There is no requirement on the Consumer that the value *user_cookie* should be unique for each DTO. The *user_cookie* is returned to the Consumer in the Completion event for the posted Receive.

The completion of the posted Receive is reported to the Consumer asynchronously through a DTO Completion event based on the configuration of the EP that dequeues the posted buffer and the specified *completion_flags* value for Solicited Wait for the matching Send. If EP Recv Completion Flag is `DAT_COMPLETION_UNSIGNALLED_FLAG`, which is the default value for SRQ EP, then all posted Recvs will generate completions with Signal Notifications.

A Consumer should not modify the *local_iov* or its content until the DTO is completed. When a Consumer does not adhere to this rule, the behavior of the Provider and the underlying Transport is not defined. Providers that allow Consumers to get ownership of the *local_iov* but not the memory it specified back after the `dat_srq_post_recv()` returns should document this behavior and also specify its support in Provider attributes. This behavior allows Consumer full control of the *local_iov* content after `dat_srq_post_recv()` returns. Because this behavior is not guaranteed by all Providers, portable Consumers shall not rely on this behavior. Consumers shall not rely on the Provider copying *local_iov* information.

The `DAT_SUCCESS` return of the `dat_srq_post_recv()` is at least the equivalent of posting a Receive operation directly by native Transport. Providers shall avoid resource allocation as part of `dat_srq_post_recv()` to ensure that this operation is nonblocking.

The completion of the Receive posted to the SRQ is equivalent to what happened to the Receive posted to the Endpoint for the Endpoint that dequeued the Receive buffer from the Shared Receive queue.

The posted Recv DTO will complete with signal, equivalently to the completion of Recv posted directly to the Endpoint that dequeued the Recv buffer from SRQ with `DAT_COMPLETION_UNSIGNALLED_FLAG` value not set for it.

The posted Recv DTOs will complete in the order of Send postings to the other endpoint of each connection whose local EP uses SRQ. There is no ordering among different connections regardless if they share SRQ and *recv_evd* or not.

If the reported status of the Completion DTO event corresponding to the posted RDMA Read DTO is not `DAT_DTO_SUCCESS`, the content of the *local_iov* is not defined and the *transfered_length* in the DTO Completion event is not defined.

The operation is valid for all states of the Shared Receive Queue.

The `dat_srq_post_recv()` function is asynchronous, nonblocking, and its thread safety is Provider-dependent.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INVALID_HANDLE`	The *srq_handle* argument is an invalid DAT handle.
`DAT_INSUFFICIENT_RESOURCES`	The operation failed due to resource limitations.
`DAT_INVALID_PARAMETER`	Invalid parameter. For example, one of the IOV segments pointed to a memory outside its LMR.
`DAT_PROTECTION_VIOLATION`	Protection violation for local or remote memory access.
	Protection Zone mismatch between an LMR of one of the *local_iov* segments and the SRQ.
`DAT_PRIVILEGES_VIOLATION`	Privileges violation for local or remote memory access. One of the LMRs used in local_iov was either invalid or did not have the local write privileges.

Usage For the best Recv operation performance, the Consumer should align each buffer segment of *local_iov* to the Optimal Buffer Alignment attribute of the Provider. For portable applications, the Consumer should align each buffer segment of *local_iov* to the `DAT_OPTIMAL_ALIGNMENT`.

Since any of the Endpoints that use the SRQ can dequeue the posted buffer from SRQ, Consumers should post a buffer large enough to handle incoming message on any of these Endpoint connections.

The buffer posted to SRQ does not have a DTO completion flag value. Posting Recv buffer to SRQ is semantically equivalent to posting to EP with `DAT_COMPLETION_UNSIGNALLED_FLAG` is not set. The configuration of the Recv Completion flag of an Endpoint that dequeues the posted buffer defines how DTO completion is generated. If the Endpoint Recv Completion flag is `DAT_COMPLETION_SOLICITED_WAIT_FLAG` then matching Send DTO completion flag value for Solicited Wait determines if the completion will be Signalled or not. If the Endpoint Recv Completion flag is not `DAT_COMPLETION_SOLICITED_WAIT_FLAG`, the posted Recv completion will be generated with Signal. If the Endpoint Recv Completion flag is `DAT_COMPLETION_EVD_THRESHOLD_FLAG`, the posted Recv completion will be generated with Signal and *dat_evd_wait* threshold value controls if the waiter will be unblocked or not.

Only the Endpoint that is in Connected or Disconnect Pending states can dequeue buffers from SRQ. When an Endpoint is transitioned into Disconnected state, all the buffers that it dequeued from SRQ are queued on the Endpoint *recv_evd*. All the buffers that the Endpoint has not completed by the time of transition into Disconnected state and that have not completed message reception will be flushed.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.2

See Also `dat_srq_create(3DAT)`, `dat_srq_free(3DAT)`, `dat_srq_query(3DAT)`, `dat_srq_resize(3DAT)`, `dat_srq_set_lw(3DAT)`, `libdat(3LIB)`, `attributes(5)`

Name dat_srq_query – provide parameters of the shared receive queue

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_srq_query (
    IN      DAT_SRQ_HANDLE      srq_handle,
    IN      DAT_SRQ_PARAM_MASK  srq_param_mask,
    OUT     DAT_SRQ_PARAM       *srq_param
    )
```

Parameters

srq_handle A handle for an instance of the SRQ.

srq_param_mask The mask for SRQ parameters.

srq_param A pointer to a Consumer-allocated structure that the Provider fills with SRQ parameters.

Description The `dat_srq_query()` function provides to the Consumer SRQ parameters. The Consumer passes a pointer to the Consumer-allocated structures for SRQ parameters that the Provider fills.

The *srq_param_mask* argument allows Consumers to specify which parameters to query. The Provider returns values for the requested *srq_param_mask* parameters. The Provider can return values for any other parameters.

In addition to the elements in SRQ attribute, `dat_srq_query()` provides additional information in the `srq_param` structure if Consumer requests it with *srq_param_mask* settings. The two that are related to entry counts on SRQ are the number of Receive buffers (*available_dto_count*) available for EPs to dequeue and the number of occupied SRQ entries (*outstanding_dto_count*) not available for new Recv buffer postings.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_PARAMETER` The *srq_param_mask* argument is invalid.

`DAT_INVALID_HANDLE` The *srq_handle* argument is an invalid DAT handle.

Usage The Provider might not be able to provide the number of outstanding Recv of SRQ or available Recvs of SRQ. The Provider attribute indicates if the Provider does not support the query for one or these values. Even when the Provider supports the query for one or both of these values, it might not be able to provide this value at this moment. In either case, the return value for the attribute that cannot be provided will be `DAT_VALUE_UNKNOWN`.

Example: Consumer created SRQ with 10 entries and associated 1 EP with it. 3 Recv buffers have been posted to it. The query will report:

```
max_recv_dtos=10,
available_dto_count=3,
outstanding_dto_count=3.
```

After a Send message arrival the query will report:

```
max_recv_dtos=10,
available_dto_count=2,
outstanding_dto_count=3.
```

After Consumer dequeues Recv completion the query will report:

```
max_recv_dtos=10,
available_dto_count=2,
outstanding_dto_count=2.
```

In general, each EP associated with SRQ can have multiple buffers in progress of receiving messages as well completed Recv on EVDs. The watermark setting helps to control how many Recv buffers posted to SRQ an Endpoint can own.

If the Provider cannot support the query for the number of outstanding Recv of SRQ or available Recvs of SRQ, the value return for that attribute should be `DAT_VALUE_UNKNOWN`.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.2

See Also `dat_srq_create(3DAT)`, `dat_srq_free(3DAT)`, `dat_srq_post_recv(3DAT)`, `dat_srq_resize(3DAT)`, `dat_srq_set_lw(3DAT)`, `libdat(3LIB)`, `attributes(5)`

Name dat_srq_resize – modify the size of the shared receive queue

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_srq_resize (
    IN      DAT_SRQ_HANDLE      srq_handle,
    IN      DAT_COUNT           srq_max_recv_dto
    )
```

Parameters

srq_handle A handle for an instance of the SRQ.

srq_max_recv_dto The new maximum number of Recv DTOs that Shared Receive Queue must hold.

Description The `dat_srq_resize()` function modifies the size of the queue of SRQ.

Resizing of Shared Receive Queue should not cause any incoming messages on any of the EPs that use the SRQ to be lost. If the number of outstanding Recv buffers on the SRQ is larger then the requested *srq_max_recv_dto*, the operation returns `DAT_INVALID_STATE` and do not change SRQ. This includes not just the buffers on the SRQ but all outstanding Receive buffers that had been posted to the SRQ and whose completions have not reaped yet. Thus, the outstanding buffers include the buffers on SRQ, the buffers posted to SRQ at are at SRQ associated EPs, and the buffers posted to SRQ for which completions have been generated but not yet reaped by Consumer from recv_evds of the EPs that use the SRQ.

If the requested *srq_max_recv_dto* is below the SRQ low watermark, the operation returns `DAT_INVALID_STATE` and does not change SRQ.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *srq_handle* argument is an invalid DAT handle.

`DAT_INVALID_PARAMETER` The *srq_max_recv_dto* argument is invalid.

`DAT_INSUFFICIENT_RESOURCES` The operation failed due to resource limitations.

`DAT_INVALID_STATE` Invalid state. Either the number of entries on the SRQ exceeds the requested SRQ queue length or the requested SRQ queue length is smaller than the SRQ low watermark.

Usage The `dat_srq_resize()` function is required not to lose any buffers. Thus, it cannot shrink below the outstanding number of Recv buffers on SRQ. There is no requirement to shrink the SRQ to return `DAT_SUCCESS`.

The quality of the implementation determines how closely to the Consumer-requested value the Provider shrinks the SRQ. For example, the Provider can shrink the SRQ to the Consumer-requested value and if the requested value is smaller than the outstanding buffers on SRQ, return DAT_INVALID_STATE; or the Provider can shrink to some value larger than that requested by the Consumer but below current SRQ size; or the Provider does not change the SRQ size and still returns DAT_SUCCESS.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.2

See Also dat_srq_create(3DAT), dat_srq_free(3DAT), dat_srq_post_recv(3DAT), dat_srq_query(3DAT), dat_srq_set_lw(3DAT), libdat(3LIB), attributes(5)

Name dat_srq_set_lw – set low watermark on shared receive queue

Synopsis

```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_srq_set_lw (
        IN      DAT_SRQ_HANDLE      srq_handle,
        IN      DAT_COUNT           low_watermark
    )
```

Parameters

srq_handle A handle for an instance of a Shared Receive Queue.

low_watermark The low watermark for the number of Recv buffers on SRQ.

Description The `dat_srq_set_lw()` function sets the low watermark value for the SRQ and arms the SRQ for generating an asynchronous event for the low watermark. An asynchronous event will be generated when the number of buffers on the SRQ is below the low watermark for the first time. This can occur during the current call or when an associated EP takes a buffer from the SRQ.

The asynchronous event will be generated only once per setting of the low watermark. Once an event is generated, no new asynchronous events for the number of buffers inthe SRQ below the specified value will be generated until the SRQ is again set for the Low Watermark. If the Consumer is again interested in the event, the Consumer should set the low watermark again.

Return Values

`DAT_SUCCESS` The operation was successful.

`DAT_INVALID_HANDLE` The *srq_handle* argument is an invalid DAT handle.

`DAT_INVALID_PARAMETER` Invalid parameter; the value of *low_watermark* is exceeds the value of *max_recv_dtos*.

`DAT_MODEL_NOT_SUPPORTED` The requested Model was not supported by the Provider. The Provider does not support SRQ Low Watermark.

Usage Upon receiving the asynchronous event for the SRQ low watermark, the Consumer can replenish Recv buffers on the SRQ or take any other action that is appropriate.

Regardless of whether an asynchronous event for the low watermark has been generated, this operation will set the generation of an asynchronous event with the Consumer-provided low watermark value. If the new low watermark value is below the current number of free Receive DTOs posted to the SRQ, an asynchronous event will be generated immediately. Otherwise the old low watermark value is simply replaced with the new one.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe
Standard	uDAPL, 1.2

See Also dat_srq_create(3DAT), dat_srq_free(3DAT), dat_srq_post_recv(3DAT), dat_srq_query(3DAT), dat_srq_resize(3DAT), libdat(3LIB), attributes(5)

Name dat_strerror – convert a DAT return code into human readable strings

Synopsis
```
cc [ flag... ] file... -ldat [ library... ]
#include <dat/udat.h>

DAT_RETURN
    dat_strerror(
    IN    DAT_RETURN    return,
    OUT   const char    **major_message,
    OUT   const char    **minor_message
    )
```

Parameters

return	DAT function return value.
message	A pointer to a character string for the return.

Description The `dat_strerror()` function converts a DAT return code into human readable strings. The *major_message* is a string-converted `DAT_TYPE_STATUS`, while *minor_message* is a string-converted `DAT_SUBTYPE_STATUS`. If the return of this function is not `DAT_SUCCESS`, the values of *major_message* and *minor_message* are not defined.

If an undefined `DAT_RETURN` value was passed as the return parameter, the operation fails with `DAT_INVALID_PARAMETER` returned. The operation succeeds when `DAT_SUCCESS` is passed in as the return parameter.

Return Values

`DAT_SUCCESS`	The operation was successful.
`DAT_INVALID_PARAMETER`	Invalid parameter. The *return* value is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	uDAPL, 1.1, 1.2

See Also `libdat(3LIB)`, `attributes(5)`

Name demangle, cplus_demangle – decode a C++ encoded symbol name

Synopsis `cc [ flag ... ] file[ library ... ] -ldemangle`

```
#include <demangle.h>



int cplus_demangle(const char *symbol, char *prototype, size_t size);
```

Description The `cplus_demangle()` function decodes (demangles) a C++ linker symbol name (mangled name) into a (partial) C++ prototype, if possible. C++ mangled names may not have enough information to form a complete prototype.

The *symbol* string argument points to the input mangled name.

The *prototype* argument points to a user-specified output string buffer, of `size` bytes.

The `cplus_demangle()` function operates on mangled names generated by SPARCompilers C++ 3.0.1, 4.0.1, 4.1 and 4.2.

The `cplus_demangle()` function improves and replaces the `demangle()` function.

Refer to the `CC.1`, `dem.1`, and `c++filt.1` manual pages in the `/opt/SUNWspro/man/man1` directory. These pages are only available with the SPROcc package.

Return Values The `cplus_demangle()` function returns the following values:

`0` — The *symbol* argument is a valid mangled name and *prototype* contains a (partial) prototype for the symbol.

`DEMANGLE_ENAME` — The *symbol* argument is not a valid mangled name and the content of *prototype* is a copy of the symbol.

`DEMANGLE_ESPACE` — The *prototype* output buffer is too small to contain the prototype (or the symbol), and the content of *prototype* is undefined.

Name devid_get, devid_compare, devid_deviceid_to_nmlist, devid_free, devid_free_nmlist, devid_get_minor_name, devid_sizeof, devid_str_decode, devid_str_free, devid_str_encode, devid_valid – device ID interfaces for user applications

Synopsis

```
cc [ flag... ] file... -ldevid [ library... ]
#include <devid.h>

int devid_get(int fd, ddi_devid_t *retdevid);

void devid_free(ddi_devid_t devid);

int devid_get_minor_name(int fd, char **retminor_name);

int devid_deviceid_to_nmlist(char *search_path, ddi_devid_t devid,
     char *minor_name, devid_nmlist_t **retlist);

void devid_free_nmlist(devid_nmlist_t *list);

int devid_compare(ddi_devid_t devid1, ddi_devid_t devid2);

size_t devid_sizeof(ddi_devid_t devid);

int devid_valid(ddi_devid_t devid);

char *devid_str_encode(ddi_devid_t devid, char *minor_name);

int devid_str_decode(char *devidstr, ddi_devid_t *retdevid,
     char **retminor_name);

void devid_str_free(char *str);
```

Description These functions provide unique identifiers (device IDs) for devices. Applications and device drivers use these functions to identify and locate devices, independent of the device's physical connection or its logical device name or number.

The `devid_get()` function returns in *retdevid* the device ID for the device associated with the open file descriptor *fd*, which refers to any device. It returns an error if the device does not have an associated device ID. The caller must free the memory allocated for *retdevid* using the `devid_free()` function.

The `devid_free()` function frees the space that was allocated for the returned *devid* by `devid_get()` and `devid_str_decode()`.

The `devid_get_minor_name()` function returns the minor name, in *retminor_name*, for the device associated with the open file descriptor *fd*. This name is specific to the particular minor number, but is “instance number” specific. The caller of this function must free the memory allocated for the returned *retminor_name* string using `devid_str_free()`.

The `devid_deviceid_to_nmlist()` function returns an array of *devid_nmlist* structures, where each entry matches the *devid* and *minor_name* passed in. If the *minor_name* specified is one of the special values (`DEVID_MINOR_NAME_ALL`, `DEVID_MINOR_NAME_ALL_CHR`, or `DEVID_MINOR_NAME_ALL_BLK`) , then all minor names associated with *devid* which also meet the special *minor_name* filtering requirements are returned. The *devid_nmlist* structure contains the device name and device number. The last entry of the array contains a null pointer for the *devname* and `NODEV` for the device number. This function traverses the file tree, starting at *search_path*. For each device with a matching device ID and minor name tuple, a device name and device number are added to the *retlist*. If no matches are found, an error is returned. The caller of this function must free the memory allocated for the returned array with the `devid_free_nmlist()` function. This function may take a long time to complete if called with the device ID of an unattached device.

The `devid_free_nmlist()` function frees the memory allocated by the `devid_deviceid_to_nmlist()` function.

The `devid_compare()` function compares two device IDs and determines both equality and sort order. The function returns an integer greater than 0 if the device ID pointed to by *devid1* is greater than the device ID pointed to by *devid2*. It returns 0 if the device ID pointed to by *devid1* is equal to the device ID pointed to by *devid2*. It returns an integer less than 0 if the device ID pointed to by *devid1* is less than the device ID pointed to by *devid2*. This function is the only valid mechanism to determine the equality of two devids. This function may indicate equality for arguments which by simple inspection appear different.

The `devid_sizeof()` function returns the size of *devid* in bytes.

The `devid_valid()` function validates the format of a *devid*. It returns 1 if the format is valid, and 0 if invalid. This check may not be as complete as the corresponding kernel function `ddi_devid_valid()` (see `ddi_devid_compare(9F)`).

The `devid_str_encode()` function encodes a *devid* and *minor_name* into a null-terminated ASCII string, returning a pointer to that string. To avoid shell conflicts, the *devid* portion of the string is limited to uppercase and lowercase letters, digits, and the plus (+), minus (-), period (.), equals (=), underscore (_), tilde (~), and comma (,) characters. If there is an ASCII quote character in the binary form of a *devid*, the string representation will be in `hex_id` form, not `ascii_id` form. The comma (,) character is added for "id1," at the head of the string *devid*. If both a *devid* and a *minor_name* are non-null, a slash (/)is used to separate the *devid* from the *minor_name* in the encoded string. If *minor_name* is null, only the *devid* is encoded. If the *devid* is null then the special string "id0" is returned. Note that you cannot compare the returned string against another string with `strcmp(3C)` to determine devid equality. The string returned must be freed by calling `devid_str_free()`.

The `devid_str_decode()` function takes a string previously produced by the `devid_str_encode()` or `ddi_devid_str_encode()` (see `ddi_devid_compare(9F)`) function and decodes the contained device ID and minor name, allocating and returning pointers to the extracted parts via the *retdevid* and *retminor_name* arguments. If the special *devidstr* "id0" was specified, the returned device ID and minor name will both be null. A non-null returned devid must be freed by the caller by the `devid_free()` function. A non-null returned minor name must be freed by calling `devid_str_free()`.

The `devid_str_free()` function frees the character string returned by `devid_str_encode()` and the *retminor_name* argument returned by `devid_str_decode()`.

Return Values Upon successful completion, the `devid_get()`, `devid_get_minor_name()`, `devid_str_decode()`, and `devid_deviceid_to_nmlist()` functions return `0`. Otherwise, they return `−1`.

The `devid_compare()` function returns the following values:

`−1` The device ID pointed to by *devid1* is less than the device ID pointed to by *devid2*.

`0` The device ID pointed to by *devid1* is equal to the device ID pointed to by *devid2*.

`1` The device ID pointed to by *devid1* is greater than the device ID pointed to by *devid2*.

The `devid_sizeof()` function returns the size of *devid* in bytes. If *devid* is null, the number of bytes that must be allocated and initialized to determine the size of a complete device ID is returned.

The `devid_valid()` function returns 1 if the *devid* is valid and 0 if the *devid* is invalid.

The `devid_str_encode()` function returns `NULL` to indicate failure. Failure may be caused by attempting to encode an invalid string. If the return value is non-null, the caller must free the returned string by using the `devid_str_free()` function.

Examples **EXAMPLE 1** Using `devid_get()`, `devid_get_minor_name()`, and `devid_str_encode()`

The following example shows the proper use of `devid_get()`, `devid_get_minor_name()`, and `devid_str_encode()` to free the space allocated for *devid*, *minor_name* and encoded *devid*.

```
int fd;
ddi_devid_t    devid;
char        *minor_name, *devidstr;
if ((fd = open("/dev/dsk/c0t3d0s0", O_RDONLY|O_NDELAY)) < 0) {
    ...
}
if (devid_get(fd, &devid) != 0) {
    ...
}
if (devid_get_minor_name(fd, &minor_name) != 0) {
    ...
}
if ((devidstr = devid_str_encode(devid, minor_name)) == 0) {
    ...
}
printf("devid %s\n", devidstr);
devid_str_free(devidstr);
devid_free(devid);
devid_str_free(minor_name);
```

EXAMPLE 2 Using devid_deviceid_to_nmlist() and devid_free_nmlist()

The following example shows the proper use of devid_deviceid_to_nmlist() and devid_free_nmlist():

```
devid_nmlist_t  *list = NULL;
int    err;
if (devid_deviceid_to_nmlist("/dev/rdsk", devid,
    minor_name, &list))
        return (-1);
/* loop through list and process device names and numbers */
devid_free_nmlist(list);
```

Attributes

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

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

See Also

free(3C), libdevid(3LIB), attributes(5), ddi_devid_compare(9F)

Name di_binding_name, di_bus_addr, di_compatible_names, di_devid, di_driver_name, di_driver_ops, di_driver_major, di_instance, di_nodeid, di_node_name – return libdevinfo node information

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

char *di_binding_name(di_node_t node);

char *di_bus_addr(di_node_t node);

int di_compatible_names(di_node_t node, char **names);

ddi_devid_t di_devid(di_node_t node);

char *di_driver_name(di_node_t node);

uint_t di_driver_ops(di_node_t node);

int di_driver_major(di_node_t node);

int di_instance(di_node_t node);

int di_nodeid(di_node_t node);

char *di_node_name(di_node_t node);
```

Parameters

names The address of a pointer.

node A handle to a device node.

Description These functions extract information associated with a device node.

Return Values The `di_binding_name()` function returns a pointer to the binding name. The binding name is the name used by the system to select a driver for the device.

The `di_bus_addr()` function returns a pointer to a null-terminated string containing the assigned bus address for the device. `NULL` is returned if a bus address has not been assigned to the device. A zero-length string may be returned and is considered a valid bus address.

The return value of `di_compatible_names()` is the number of compatible names. *names* is updated to point to a buffer contained within the snapshot. The buffer contains a concatenation of null-terminated strings, for example:

```
<name1>/0<name2>/0...<namen>/0
```

See the discussion of generic names in *Writing Device Drivers* for a description of how compatible names are used by Solaris to achieve driver binding for the node.

The `di_devid()` function returns the device ID for *node*, if it is registered. Otherwise, a null pointer is returned. Interfaces in the `libdevid(3LIB)` library may be used to manipulate the handle to the device id. This function is obsolete and might be removed from a future Solaris release. Applications should use the “devid” property instead.

The `di_driver_name()` function returns the name of the driver bound to the *node*. A null pointer is returned if *node* is not bound to any driver.

The `di_driver_ops()` function returns a bit array of device driver entry points that are supported by the driver bound to this *node*. Possible bit fields supported by the driver are `DI_CB_OPS`, `DI_BUS_OPS`, `DI_STREAM_OPS`.

The `di_driver_major()` function returns the major number associated with the driver bound to *node*. If there is no driver bound to the node, this function returns −1.

The `di_instance()` function returns the instance number of the device. A value of `-1` indicates an instance number has not been assigned to the device by the system.

The `di_nodeid()` function returns the type of device, which may be one of the following possible values: `DI_PSEUDO_NODEID`, `DI_PROM_NODEID`, and `DI_SID_NODEID`. Devices of type `DI_PROM_NODEID` may have additional properties that are defined by the PROM. See `di_prom_prop_data`(3DEVINFO) and `di_prom_prop_lookup_bytes`(3DEVINFO).

The `di_node_name()` function returns a pointer to a null-terminated string containing the node name.

Examples See `di_init`(3DEVINFO) for an example demonstrating typical use of these functions.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed (`di_devid()` is obsolete)
MT-Level	Safe

See Also `di_init`(3DEVINFO), `di_prom_init`(3DEVINFO), `di_prom_prop_data`(3DEVINFO), `di_prom_prop_lookup_bytes`(3DEVINFO), `libdevid`(3LIB), `libdevinfo`(3LIB), `attributes`(5)

Writing Device Drivers

Name di_child_node, di_parent_node, di_sibling_node, di_drv_first_node, di_drv_next_node – libdevinfo node traversal functions

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

di_node_t di_child_node(di_node_t node);

di_node_t di_parent_node(di_node_t node);

di_node_t di_sibling_node(di_node_t node);

di_node_t di_drv_first_node(const char *drv_name, di_node_t root);

di_node_t di_drv_next_node(di_node_t node);
```

Parameters

drv_name The name of the driver of interest.

node A handle to any node in the snapshot.

root The handle of the root node for the snapshot returned by di_init(3DEVINFO).

Description The kernel device configuration data may be viewed in two ways, either as a tree of device configuration nodes or as a list of nodes associated with each driver. In the tree view, each node may contain references to its parent, the next sibling in a list of siblings, and the first child of a list of children. In the per-driver view, each node contains a reference to the next node associated with the same driver. Both views are captured in the snapshot, and the interfaces are provided for node access.

The `di_child_node()` function obtains a handle to the first child of *node*. If no child node exists in the snapshot, `DI_NODE_NIL` is returned and `errno` is set to `ENXIO` or `ENOTSUP`.

The `di_parent_node()` function obtains a handle to the parent node of *node*. If no parent node exists in the snapshot, `DI_NODE_NIL` is returned and `errno` is set to `ENXIO` or `ENOTSUP`.

The `di_sibling_node()` function obtains a handle to the next sibling node of *node*. If no next sibling node exists in the snapshot, `DI_NODE_NIL` is returned and `errno` is set to `ENXIO` or `ENOTSUP`.

The `di_drv_first_node()` function obtains a handle to the first node associated with the driver specified by `drv_name`. If there is no such driver, `DI_NODE_NIL` is returned with `errno` is set to `EINVAL`. If the driver exists but there is no node associated with this driver, `DI_NODE_NIL` is returned and `errno` is set to `ENXIO` or `ENOTSUP`.

The `di_drv_next_node()` function returns a handle to the next node bound to the same driver. If no more nodes exist, `DI_NODE_NIL` is returned.

Return Values Upon successful completion, a handle is returned. Otherwise, `DI_NODE_NIL` is returned and `errno` is set to indicate the error.

Errors These functions will fail if:

`EINVAL` The argument is invalid.

`ENXIO` The requested node does not exist.

`ENOTSUP` The node was not found in the snapshot, but it may exist in the kernel. This error may occur if the snapshot contains a partial device tree.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `di_init(3DEVINFO)`, `libdevinfo(3LIB)`, `attributes(5)`

Writing Device Drivers

Name di_devfs_path, di_devfs_minor_path, di_path_devfs_path, di_path_client_devfs_path, di_devfs_path_free – generate and free path names

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include &lt;libdevinfo.h&gt;

char *di_devfs_path(di_node_t node);

char *di_devfs_minor_path(di_minor_t minor);

char *di_path_devfs_path(di_path_t path);

char *di_path_client_devfs_path(di_path_t path);

void di_devfs_path_free(char *path_buf);
```

Parameters

node The handle to a device node in a `di_init`(3DEVINFO) snapshot.

minor The handle to a device minor node in a snapshot.

path The handle to a device path node in a snapshot.

path_buf A pointer returned by `di_devfs_path()`, `di_devfs_minor_path()`, `di_path_devfs_path()`, or `di_path_client_devfs_path()`.

Description The `di_devfs_path()` function generates the physical path of the device node specified by *node*.

The `di_devfs_minor_path()` function generates the physical path of the device minor node specified by *minor*.

The `di_path_devfs_path()` function generates the pHCI physical path to the device associated with the specified path node. The returned string is identical to the `di_devfs_path()` for the device had the device not been supported by multipath.

The `di_path_client_devfs_path()` function generates the vHCI physical path of the multipath client device node associated with the device identity of the specified path node. The returned string is identical to the `di_devfs_path()` of the multipath client device node.

The `di_devfs_path_free()` function frees memory that was allocated to store the path returned by `di_devfs_path()`, `di_devfs_minor_path()`, `di_path_devfs_path()`, and `di_path_client_devfs_path()`. The caller is responsible for freeing this memory by calling `di_devfs_path_free()`.

Return Values Upon successful completion, the `di_devfs_path()`, `di_devfs_minor_path()`, `di_path_devfs_path()`, and `di_path_client_devfs_path()` functions return a pointer to the string containing the path to a device node, a device minor node, or a device path node, respectively. Otherwise, they return `NULL` and `errno` is set to indicate the error. For a non-NULL return, the path will not have a "`/devices`" prefix.

Errors The `di_devfs_path()`, `di_devfs_minor_path()`, `di_path_devfs_path()`, and `di_path_client_devfs_path()` functions will fail if:

`EINVAL` The *node*, *minor*, or *path* argument is not a valid handle.

The `di_devfs_path()`, `di_devfs_minor_path()`, `di_path_devfs_path()`, and `di_path_client_devfs_path()` functions can also return any error value returned by `malloc`(3C).

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `di_init`(3DEVINFO), `libdevinfo`(3LIB), `malloc`(3C), `attributes`(5)

Writing Device Drivers

Name di_devlink_dup, di_devlink_free – copy and free a devlink object

Synopsis
```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

di_devlink_t di_devlink_dup(di_devlink_t devlink);

int di_devlink_free(di_devlink_t devlink);
```

Parameters *devlink* An opaque handle to a devlink.

Description Typically, a `di_devlink_t` object is only accessible from within the scope of the `di_devlink_walk`(3DEVINFO) callback function. The `di_devlink_dup()` function allows the callback function implementation to make a duplicate copy of the `di_devlink_t` object. The duplicate copy is valid and accessible until `di_devlink_free()` is called.

The `di_devlink_dup()` function returns a copy of a *devlink* object. The `di_devlink_free()` function frees this copy.

Return Values Upon successful completion, `di_devlink_dup()` returns a copy of the *devlink* object passed in. Otherwise, `NULL` is returned and `errno` is set to indicate the error.

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

Errors The `di_devlink_dup()` and `di_devlink_free()` functions will fail if:

`EINVAL` The *devlink* argument is not a valid handle.

The `di_devlink_dup()` function can set `errno` to any error value that can also be set by `malloc`(3C).

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `di_devlink_init`(3DEVINFO), `di_devlink_path`(3DEVINFO), `di_devlink_walk`(3DEVINFO), `libdevinfo`(3LIB), `malloc`(3C), `attributes`(5)

Name di_devlink_init, di_devlink_fini – create and destroy a snapshot of devlinks

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

di_devlink_handle_t di_devlink_init(const char *name,
     uint_t flags);

int di_devlink_fini(di_devlink_handle_t *hdlp);
```

Parameters

flags The following values are supported:

`DI_MAKE_LINK` Synchronize with devlink management before taking the snapshot. The name argument determines which devlink management activities must complete before taking a devlink snapshot. Appropriate privileges are required to use this flag.

name If flags is `DI_MAKE_LINK`, *name* determines which devlink management activity must complete prior to snapshot.

- If *name* is `NULL` then all devlink management activities must complete. The devlink snapshot returned accurately reflects the entire kernel device tree.
- If *name* is a driver name, devlink management activities associated with nodes bound to that driver must complete.
- If *name* is a path to a node in the kernel device tree (no "`/devices`" prefix), devlink management activities below node must complete.
- If *name* is a path to a minor node in the kernel device tree (no "`/devices`"prefix), devlink management activities on that minor node must complete.

hdlp The handle to the snapshot obtained by calling `di_devlink_init()`.

Description System management applications often need to map a "`/devices`" path to a minor node to a public "`/dev`" device name. The `di_devlink_*()` functions provide an efficient way to accomplish this.

The `di_devlink_init()` function takes a snapshot of devlinks and returns a handle to this snapshot.

The `di_devlink_fini()` function destroys the devlink snapshot and frees the associated memory.

Return Values Upon successful completion, `di_devlink_init()` returns a handle to a devlink snapshot. Otherwise, `DI_LINK_NIL` is returned and `errno` is set to indicate the error.

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

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

`EINVAL` One or more arguments is invalid.

The `di_devlink_init()` function with `DI_MAKE_LINK` can also fail if:

`EPERM` The user does no have appropriate privileges.

The `di_devlink_init()` function can set `errno` to any error value that can also be set by `malloc`(3C), `open`(2), `ioctl`(2), or `mmap`(2).

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `ioctl`(2), `mmap`(2), `open`(2), `di_devlink_path`(3DEVINFO), `di_devlink_walk`(3DEVINFO), `libdevinfo`(3LIB), `malloc`(3C), `attributes`(5)

Name di_devlink_path, di_devlink_content, di_devlink_type – get devlink attributes

Synopsis
```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

const char *di_devlink_path(di_devlink_t devlink);

const char *di_devlink_content(di_devlink_t devlink);

int di_devlink_type(di_devlink_t devlink);
```

Parameters *devlink* An opaque handle to a devlink.

Description These functions return various attributes of a devlink.

Return Values The `di_devlink_path()` function returns the absolute path of a devlink. On error, `NULL` is returned and `errno` is set to indicate the error.

The `di_devlink_content()` function returns the content of the symbolic link represented by *devlink*. On error, `NULL` is returned and `errno` is set to indicate the error.

The `di_devlink_type()` function returns the devlink type, either `DI_PRIMARY_LINK` or `DI_SECONDARY_LINK`. On error, -1 is returned and `errno` is set to indicate the error.

Errors These functions will fail if:

`EINVAL` The *devlink* argument is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `di_devlink_init(3DEVINFO)`, `di_devlink_walk(3DEVINFO)`, `libdevinfo(3LIB)`, `malloc(3C)`, `attributes(5)`

Name di_devlink_walk – walk through links in a devlink snapshot

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

int di_devlink_walk(di_devlink_handle_t hdl, const char *re,
     const char *mpath, uint_t flags, void *arg,
     int (*devlink_callback)(di_devlink_t devlink, void *arg));
```

Parameters

hdl A handle to a snapshot of devlinks in "`/dev`".

re An extended regular expression as specified in `regex(5)` describing the paths of devlinks to visit. A null value matches all devlinks. The expression should not involve the "`/dev`" prefix. For example, the "`^dsk/`" will invoke *devlink_callback*() for all "`/dev/dsk/`" links.

mpath A path to a minor node below "`/devices`" for which "`/dev`" links are to be looked up. A null value selects all devlinks. This path should not have a "`/devices`" prefix.

flags Specify the type of devlinks to be selected. If `DI_PRIMARY_LINK` is used, only primary links (for instance, links which point only to "`/devices`" entries) are selected. If `DI_SECONDARY_LINK` is specified, only secondary links (for instance, devlinks which point to other devlinks) are selected. If neither flag is specified, all devlinks are selected.

arg A pointer to caller private data.

devlink The devlink being visited.

Description The `di_devlink_walk()` function visits every link in the snapshot that meets the criteria specified by the caller. For each such devlink, the caller-supplied function *devlink_callback*() is invoked. The return value of *devlink_callback*() determines subsequent walk behavior.

Return Values Upon success, the `di_devlink_walk()` function returns 0. Otherwise, -1 is returned and `errno` is set to indicate the error.

The *devlink_callback*() function can return the following values:

`DI_WALK_CONTINUE` Continue walking.

`DI_WALK_TERMINATE` Terminate the walk immediately.

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

`EINVAL` One or more arguments is invalid.

`ENOMEM` Insufficient memory is available.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also di_devlink_init(3DEVINFO), di_devlink_path(3DEVINFO), libdevinfo(3LIB), malloc(3C), attributes(5), regex(5)

Name di_init, di_fini – create and destroy a snapshot of kernel device tree

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

di_node_t di_init(const char *phys_path, uint_t flags);

void di_fini(di_node_t root);
```

Parameters

flags — Snapshot content specification. The possible values can be a bitwise OR of at least one of the following:

`DINFOSUBTREE`	Include subtree.
`DINFOPROP`	Include properties.
`DINFOMINOR`	Include minor node data.
`DINFOCPYALL`	Include all of the above.
`DINFOPATH`	Include multipath path node data.
`DINFOLYR`	Include device layering data.
`DINFOCPYONE`	Include only a single node without properties, minor nodes, or path nodes.

phys_path — Physical path of the *root* device node of the snapshot. See `di_devfs_path`(3DEVINFO).

root — Handle obtained by calling `di_init()`.

Description The `di_init()` function creates a snapshot of the kernel device tree and returns a handle of the *root* device node. The caller specifies the contents of the snapshot by providing *flag* and *phys_path*.

The `di_fini()` function destroys the snapshot of the kernel device tree and frees the associated memory. All handles associated with this snapshot become invalid after the call to `di_fini()`.

Return Values Upon success, `di_init()` returns a handle. Otherwise, `DI_NODE_NIL` is returned and `errno` is set to indicate the error.

Errors The `di_init()` function can set `errno` to any error code that can also be set by `open`(2), `ioctl`(2) or `mmap`(2). The most common error codes include:

`EACCES`	Insufficient privilege for accessing device configuration data.
`ENXIO`	Either the device named by *phys_path* is not present in the system, or the `devinfo`(7D) driver is not installed properly.
`EINVAL`	Either *phys_path* is incorrectly formed or the *flags* argument is invalid.

Examples

EXAMPLE 1 Using the libdevinfo Interfaces To Print All Device Tree Node Names

The following is an example using the `libdevinfo` interfaces to print all device tree device node names:

```
/*
 * Code to print all device tree device node names
 */

#include <stdio.h>
#include <libdevinfo.h>

int
prt_nodename(di_node_t node, void *arg)
{
     printf("%s\n", di_node_name(node));
     return (DI_WALK_CONTINUE);
}

main()
{
     di_node_t root_node;
     if((root_node = di_init("/", DINFOSUBTREE)) == DI_NODE_NIL) {
           fprintf(stderr, "di_init() failed\n");
           exit(1);
     }
     di_walk_node(root_node, DI_WALK_CLDFIRST, NULL, prt_nodename);
     di_fini(root_node);
}
```

EXAMPLE 2 Using the libdevinfo Interfaces To Print The Physical Path Of SCSI Disks

The following example uses the `libdevinfo` interfaces to print the physical path of SCSI disks:

```
/*
 * Code to print physical path of scsi disks
 */

#include <stdio.h>
#include <libdevinfo.h>
#define    DISK_DRIVER    "sd"    /* driver name */

void
prt_diskinfo(di_node_t node)
{
    int instance;
        char *phys_path;

    /*
```

EXAMPLE 2 Using the libdevinfo Interfaces To Print The Physical Path Of SCSI Disks *(Continued)*

```
     * If the device node exports no minor nodes,
     * there is no physical disk.
     */
     if (di_minor_next(node, DI_MINOR_NIL) == DI_MINOR_NIL) {
              return;
         }

         instance = di_instance(node);
         phys_path = di_devfs_path(node);
         printf("%s%d: %s\n", DISK_DRIVER, instance, phys_path);
         di_devfs_path_free(phys_path);
}

void
walk_disknodes(di_node_t node)
{
        node = di_drv_first_node(DISK_DRIVER, node);
        while (node != DI_NODE_NIL) {
             prt_diskinfo(node);
             node = di_drv_next_node(node);
        }
}

main()
{
    di_node_t root_node;
    if ((root_node = di_init("/", DINFOCPYALL)) == DI_NODE_NIL) {
        fprintf(stderr, "di_init() failed\n");
        exit(1);
    }
        walk_disknodes(root_node);
        di_fini(root_node);
}
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also

open(2), ioctl(2), mmap(2), libdevinfo(3LIB), attributes(5)

Writing Device Drivers

Name di_link_next_by_node, di_link_next_by_lnode – libdevinfo link traversal functions

Synopsis
```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

di_link_t di_link_next_by_node(di_lnode_t node, di_link_t link,
     uint_t endpoint);

di_link_t di_link_next_by_lnode(di_node_t lnode, di_link_t link,
     uint_t endpoint);
```

Parameters

link — The handle to the current the link or `DI_LINK_NIL`.

endpoint — Specify which endpoint of the link the node or lnode should correspond to, either `DI_LINK_TGT` or `DI_LINK_SRC`.

node — The device node with which the link is associated.

lnode — The lnode with which the link is associated.

Description The `di_link_next_by_node()` function returns a handle to the next link that has the same endpoint node as *link*. If *link* is `DI_LINK_NIL`, a handle is returned to the first link whose endpoint specified by *endpoint* matches the node specified by *node*.

The `di_link_next_by_lnode()` function returns a handle to the next link that has the same endpoint lnode as *link*. If *link* is `DI_LINK_NIL`, a handle is returned to the first link whose endpoint specified by *endpoint* matches the lnode specified by *lnode*.

Return Values Upon successful completion, a handle to the next link is returned. Otherwise, `DI_LINK_NIL` is returned and `errno` is set to indicate the error.

Errors The `di_link_next_by_node()` and `di_link_next_by_lnode()` functions will fail if:

`EINVAL` — An argument is invalid.

`ENXIO` — The end of the link list has been reached.

The `di_link_next_by_node()` function will fail if:

`ENOTSUP` — Device usage information is not available in snapshot.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also di_init(3DEVINFO), libdevinfo(3LIB), attributes(5)

Name di_link_spectype, di_link_to_lnode – return libdevinfo link information

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

int di_link_spectype(di_link_t link);

di_lnode_t di_link_to_lnode(di_link_t link, uint_t endpoint);
```

Parameters

link A handle to a link.

endpoint specifies the endpoint of the link, which should correspond to either `DI_LINK_TGT` or `DI_LINK_SRC`

Description The `di_link_spectype()` function returns libdevinfo link information.

The `di_link_to_lnode()` function takes a link specified by *link* and returns the lnode corresponding to the link endpoint specified by *endpoint*.

Return Values The `di_link_spectype()` function returns the spectype parameter flag that was used to open the target device of a link, either `S_IFCHR` or `S_IFBLK`.

Upon successful completion, `di_link_to_lnode()` returns a handle to an lnode. Otherwise, `DI_LINK_NIL` is returned and `errno` is set to indicate the error.

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

`EINVAL` An argument is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `di_init(3DEVINFO)`, `libdevinfo(3LIB)`, `attributes(5)`

Name di_lnode_name, di_lnode_devinfo, di_lnode_devt – return libdevinfo lnode information

Synopsis
```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

char *di_lnode_name(di_lnode_t lnode);

di_node_t di_lnode_devinfo(di_lnode_t lnode);

int di_lnode_devt(di_lnode_t lnode, dev_t *devt);
```

Parameters

lnode A handle to an lnode.

devt A pointer to a `dev_t` that can be returned.

Description These functions return `libdevinfo` lnode information.

The `di_lnode_name()` function returns a pointer to the name associated with *lnode*.

The `di_lnode_devinfo()` function returns a handle to the device node associated with *lnode*.

The `di_lnode_devt()` function sets the `dev_t` pointed to by the *devt* parameter to the `dev_t` associated with *lnode*.

Return Values The `di_lnode_name()` function returns a pointer to the name associated with *lnode*.

The `di_lnode_devinfo()` function returns a handle to the device node associated with *lnode*.

The `di_lnode_devt()` function returns 0 if the requested attribute exists in *lnode* and was returned. It returns −1 if the requested attribute does not exist and sets `errno` to indicate the error.

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

`EINVAL` An argument was invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `di_init(3DEVINFO)`, `libdevinfo(3LIB)`, `attributes(5)`

Name di_lnode_next – libdevinfo lnode traversal function

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

di_lnode_t di_lnode_next(di_node_t node, di_lnode_t lnode);
```

Parameters

node A handle to a di_node.

lnode A handle to an lnode.

Description The `di_lnode_next()` function returns a handle to the next lnode for the device node specified by *node*. If *lnode* is `DI_LNODE_NIL`, a handle to the first lnode is returned.

Return Values Upon successful completion, a handle to an lnode is returned. Otherwise, `DI_LNODE_NIL` is returned and `errno` is set to indicate the error.

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

`EINVAL` An argument is invalid.

`ENOTSUP` Device usage information is not available in snapshot.

`ENXIO` The end of the lnode list has been reached.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `di_init`(3DEVINFO), `libdevinfo`(3LIB), `attributes`(5)

Name di_minor_devt, di_minor_name, di_minor_nodetype, di_minor_spectype – return libdevinfo minor node information

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

dev_t di_minor_devt(di_minor_t minor);

char *di_minor_name(di_minor_t minor);

char *di_minor_nodetype(di_minor_t minor);

int di_minor_spectype(di_minor_t minor);
```

Parameters *minor* A handle to minor data node.

Description These functions return `libdevinfo` minor node information.

Return Values The `di_minor_name()` function returns the minor *name*. See `ddi_create_minor_node(9F)` for a description of the *name* parameter.

The `di_minor_devt()` function returns the `dev_t` value of the minor node that is specified by SYS V ABI. See `getmajor(9F)`, `getminor(9F)`, and `ddi_create_minor_node(9F)` for more information.

The `di_minor_spectype()` function returns the *spec_type* of the file, either `S_IFCHR` or `S_IFBLK`. See `ddi_create_minor_node(9F)` for a description of the *spec_type* parameter.

The `di_minor_nodetype()` function returns the minor *node_type* of the minor node. See `ddi_create_minor_node(9F)` for a description of the *node_type* parameter.

Errors No errors are defined.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `attributes(5)`, `ddi_create_minor_node(9F)`, `getmajor(9F)`, `getminor(9F)`

Writing Device Drivers

Name di_minor_next – libdevinfo minor node traversal functions

Synopsis
```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

di_minor_t di_minor_next(di_node_t node, di_minor_t minor);
```

Parameters

minor Handle to the current minor node or `DI_MINOR_NIL`.

node Device node with which the minor node is associated.

Description The `di_minor_next()` function returns a handle to the next minor node for the device node *node*. If *minor* is `DI_MINOR_NIL`, a handle to the first minor node is returned.

Return Values Upon successful completion, a handle to the next minor node is returned. Otherwise, `DI_MINOR_NIL` is returned and `errno` is set to indicate the error.

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

`EINVAL` Invalid argument.

`ENOTSUP` Minor node information is not available in snapshot.

`ENXIO` End of minor node list.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libdevinfo(3LIB)`, `attributes(5)`

Writing Device Drivers

Name di_node_private_set, di_node_private_get, di_path_private_set, di_path_private_get, di_minor_private_set, di_minor_private_get, di_link_private_set, di_link_private_get, di_lnode_private_set, di_lnode_private_get – manipulate libdevinfo user traversal pointers

Synopsis
```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

void di_node_private_set(di_node_t node, void *data);

void *di_node_private_get(di_node_t node);

void di_path_private_set(di_path_t path, void *data);

void *di_path_private_get(di_path_t path);

void di_minor_private_set(di_minor_t minor, void *data);

void *di_minor_private_get(di_minor_t minor);

void di_link_private_set(di_link_t link, void *data);

void *di_link_private_get(di_link_t link);

void di_lnode_private_set(di_lnode_t lnode, void *data);

void *di_lnode_private_get(di_lnode_t lnode);
```

Parameters

node The handle to a devinfo node in a `di_init(3DEVINFO)` snapshot.

path The handle to a path node in a snapshot.

minor The handle to a minor node in a snapshot.

link The handle to a link in a snapshot.

lnode The handle to an lnode in a snapshot.

data A pointer to caller-specific data.

Description The `di_node_private_set()` function allows a caller to associate caller-specific data pointed to by *data* with a devinfo node, thereby facilitating traversal of devinfo nodes in the snapshot.

The `di_node_private_get()` function allows a caller to retrieve a data pointer that was associated with a devinfo node obtained by a call to `di_node_private_set()`.

The `di_path_private_set()` function allows a caller to associate caller-specific data pointed to by *data* with a devinfo path node, thereby facilitating traversal of path nodes in the snapshot.

The `di_path_private_get()` function allows a caller to retrieve a data pointer that was associated with a path node obtained by a call to `di_path_private_set()`.

The `di_minor_private_set()` function allows a caller to associate caller-specific data pointed to by *data* with a minor node specified by *minor*, thereby facilitating traversal of minor nodes in the snapshot.

The `di_minor_private_get()` function allows a caller to retrieve a data pointer that was associated with a minor node obtained by a call to `di_minor_private_set()`.

The `di_link_private_set()` function allows a caller to associate caller-specific data pointed to by *data* with a link, thereby facilitating traversal of links in the snapshot.

The `di_link_private_get()` function allows a caller to retrieve a data pointer that was associated with a link obtained by a call to `di_link_private_set()`.

The `di_lnode_private_set()` function allows a caller to associate caller-specific data pointed to by *data* with an lnode specified by *lnode*, thereby facilitating traversal of lnodes in the snapshot.

The `di_lnode_private_get()` function allows a caller to retrieve a data pointer that was associated with an lnode by a call to `di_lnode_private_set()`.

These functions do not perform any type of locking. It is up to the caller to satisfy any locking needs.

Return Values The `di_node_private_set()`, `di_path_private_set()`, `di_minor_private_set()`, `di_link_private_set()`, and `di_lnode_private_set()` functions do not return values.

The `di_node_private_get()`, `di_path_private_get()`, `di_minor_private_get()`, `di_link_private_get()`, and `di_lnode_private_get()` functions return a pointer to caller-specific data that was initialized with their corresponding `*_set()` function. If no caller-specific data was assigned with a `*_set()` function, the results are undefined.

Errors No errors are defined.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `di_init(3DEVINFO)`, `libdevinfo(3LIB)`, `attributes(5)`

Name di_path_bus_addr, di_path_client_node, di_path_instance, di_path_node_name, di_path_phci_node, di_path_state – return libdevinfo path node information

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

char *di_path_bus_addr(di_path_t path);

di_node_t di_path_client_node(di_path_t path);

int di_path_instance(di_path_t path);

char *di_path_node_name(di_path_t path);

di_node_t di_path_phci_node(di_path_t path);

di_path_state_t di_path_state(di_path_t path);
```

Parameters *path* The handle to a path node in a `di_init`(3DEVINFO) snapshot.

Description These functions extract information associated with a path node.

Return Values The `di_path_bus_addr()` function returns a string representing the pHCI child path node's unit-address. This function is the `di_path_t` peer of `di_bus_addr`(3DEVINFO).

The `di_path_client_node()` function returns the `di_node_t` of the 'client' device node associated with the given path node. If the client device node is not present in the current device tree snapshot, `DI_NODE_NIL` is returned and `errno` is set to `ENOTSUP`.

The `di_path_node_name()` function returns a pointer to a null-terminated string containing the path node name. This function is the `di_path_t` peer of `di_node_name`(3DEVINFO).

The `di_path_instance()` function returns the instance number associated with the given path node. A path node instance is persistent across `attach`(9E)/`detach`(9E) and device reconfigurations, but not across reboot. A path node instance is unrelated to a device node `di_instance`(3DEVINFO).

The `di_path_phci_node()` function returns the `di_node_t` of the pHCI host adapter associated with the given path node. If the pHCI device node is not present in the current device tree snapshot, `DI_NODE_NIL` is returned and `errno` is set to `ENOTSUP`.

The `di_path_state()` function returns the state of an I/O path. This function may return one of the following values:

`DI_PATH_STATE_ONLINE`
: Identifies that the `path_info` node is online and I/O requests can be routed through this path.

`DI_PATH_STATE_OFFLINE`
: Identifies that the `path_info` node is in offline state.

DI_PATH_STATE_FAULT
Identifies that the path_info node is in faulted state and not ready for I/O operations.

DI_PATH_STATE_STANDBY
Identifies that the path_info node is in standby state and not ready for I/O operations.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also di_bus_addr(3DEVINFO), di_devfs_path(3DEVINFO), di_init(3DEVINFO), di_instance(3DEVINFO), di_node_name(3DEVINFO), di_path_client_next_path(3DEVINFO), di_path_prop_next(3DEVINFO), di_path_prop_bytes(3DEVINFO), di_path_prop_lookup_bytes(3DEVINFO), di_path_prop_next(3DEVINFO), libdevinfo(3LIB), attributes(5)

Writing Device Drivers

Name di_path_client_next_path, di_path_phci_next_path – libdevinfo path node traversal functions

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

di_path_t di_path_client_next_path(di_node_t node node,
     di_path_t path);

di_path_t di_path_phci_next_path(di_node_t node node,
     di_path_t path);
```

Parameters

node The handle to a device node in a `di_init(3DEVINFO)` snapshot. For `di_path_client_next_path()`, *node* must be a client device node. For `di_path_phci_next_path()`, *node* must be a pHCI device node.

path `DI_PATH_NIL`, or the handle to a path node in a snapshot.

Description Each path node is an element in a pHCI-client matrix. The matrix is implemented by dual linked lists: one list links path nodes related to a common client head, and the other links path nodes related to a common pHCI head.

The `di_path_client_next_path()` function is called on a multipathing 'client' device node, where a 'client' is the child of a vHCI device node, and is associated with a specific endpoint device identity (independent of physical paths). If the path argument is `NULL`, `di_path_client_next_path()` returns the first path node associated with the client. To walk all path nodes associated with a client, returned `di_path_t` values are fed back into `di_path_client_next_path()`, via the *path* argument, until a null path node is returned. For each path node, `di_path_bus_addr(3DEVINFO)` returns the pHCI child path node unit-address.

The `di_path_phci_next_path()` function is called on a multipathing pHCI device node. If the path argument is `NULL`, `di_path_phci_next_path()` returns the first path node associated with the pHCI. To walk all path nodes associated with a pHCI, returned `di_path_t` values are fed back into `di_path_phci_next_path()`, via the *path* argument, until a null path node is returned. For each path node, `di_path_client_node(3DEVINFO)` provides a pointer to the associated client device node.

A device node can be a client device node of one multipathing class and a pHCI device node of another class.

Return Values Upon successful completion, a handle to the next path node is returned. Otherwise, `DI_PATH_NIL` is returned and `errno` is set to indicate the error.

Errors These functions will fail if:

`EINVAL` One or more argument was invalid.

`ENOTSUP` Path node information is not available in the snapshot.

ENXIO The end of the path node list was reached.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also

di_init(3DEVINFO), di_path_bus_addr(3DEVINFO), di_path_client_node(3DEVINFO), libdevinfo(3LIB), attributes(5)

Writing Device Drivers

Name di_path_prop_bytes, di_path_prop_ints, di_path_prop_int64s, di_path_prop_name, di_path_prop_strings, di_path_prop_type – access path property information

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

int di_path_prop_bytes(di_path_prop_t prop, uchar_t **prop_data);

int di_path_prop_ints(di_path_prop_t prop, int **prop_data);

int di_path_prop_int64s(di_path_prop_t prop, int64_t **prop_data);

char *di_path_prop_name(di_path_prop_t prop);

int di_path_prop_strings(di_path_prop_t prop, char **prop_data);

int di_path_prop_type(di_path_prop_t prop);
```

Parameters

prop A handle to a property returned by `di_path_prop_next`(3DEVINFO).

prop_data For `di_path_prop_bytes()`, the address of a pointer to an unsigned character.

For `di_path_prop_ints()`, the address of a pointer to an integer.

For `di_path_prop_int64()`, the address of a pointer to a 64-bit integer.

For `di_path_prop_strings()`, the address of pointer to a character.

Description These functions access information associated with path property values and attributes such as the property name or data type.

The `di_path_prop_name()` function returns a pointer to a string containing the name of the property.

The `di_path_prop_type()` function returns the type of the path property. The type determines the appropriate interface to access property values. Possible property types are the same as for `di_prop_type`(3DEVINFO), excluding `DI_PROP_TYPE_UNKNOWN` and `DI_PROP_UNDEFINED`. Thus, `di_path_prop_type()` can return one of the following constants:

`DI_PROP_TYPE_INT` Use `di_path_prop_ints()` to access property data.

`DI_PROP_TYPE_INT64` Use `di_path_prop_int64s()` to access property data.

`DI_PROP_TYPE_STRING` Use `di_path_prop_strings()` to access property data.

`DI_PROP_TYPE_BYTE` Use `di_path_prop_bytes()` to access property data.

The `di_path_prop_bytes()` function returns the property data as a series of unsigned characters.

The `di_path_prop_ints()` function returns the property data as a series of integers.

The `di_path_prop_int64s()` function returns the property data as a series of integers.

The `di_path_prop_strings()` function returns the property data as a concatenation of null-terminated strings.

Return Values Upon successful completion, `di_path_prop_bytes()`, `di_path_prop_ints()`, `di_path_prop_int64s()`, and `di_path_prop_strings()` return a non-negative value, indicating the number of entries in the property value buffer. If the property is found, the number of entries in *prop_data* is returned. Otherwise, -1 is returned and `errno` is set to indicate the error.

For `di_path_prop_bytes()`, the number of entries is the number of unsigned characters contained in the buffer pointed to by *prop_data*.

For `di_path_prop_ints()`, the number of entries is the number of integers contained in the buffer pointed to by *prop_data*.

For `di_path_prop_int64s()`, the number of entries is the number of 64-bit integers contained in the buffer pointed to by *prop_data*.

For `di_path_prop_strings()`, the number of entries is the number of null-terminated strings contained in the buffer. The strings are stored in a concatenated format in the buffer.

The `di_path_prop_name()` function returns the name of the property.

The `di_path_prop_type()` function can return one of types described in the Description.

Errors These functions will fail if:

`EINVAL` One of the arguments is invalid. For example, the property type does not match the interface.

`ENOTSUP` The snapshot contains no property information.

`ENXIO` The path property does not exist.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `di_path_prop_next(3DEVINFO)`, `di_prop_type(3DEVINFO)`, `libdevinfo(3LIB)`, `attributes(5)`

Writing Device Drivers

Name di_path_prop_lookup_bytes, di_path_prop_lookup_int64s, di_path_prop_lookup_ints, di_path_prop_lookup_strings – search for a path property

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

int di_path_prop_lookup_bytes(di_path_t path,
     const char *prop_name);

int di_path_prop_lookup_int64s(di_path_t path,
     const char *prop_name);

int di_path_prop_lookup_ints(di_path_t path,
     const char *prop_name, char **prop_data);

int di_path_prop_lookup_strings(di_path_t path,
     const char *prop_name, char **prop_data);
```

Parameters

path The handle to a path node in a `di_init`(3DEVINFO).

prop_name The name of property for which to search.

prop_data For `di_path_prop_lookup_bytes()`, the address to a pointer to an array of unsigned characters containing the property data.

For `di_path_prop_lookup_int64()`, the address to a pointer to an array of 64-bit integers containing the property data.

For `di_path_prop_lookup_ints()`, the address to a pointer to an array of integers containing the property data.

For `di_path_prop_lookup_strings()`, the address to a pointer to a buffer containing a concatenation of null-terminated strings containing the property data.

Description These functions return the value of a known property name and type.

All memory allocated by these functions is managed by the library and must not be freed by the caller.

Return Values If the property is found, the number of entries in *prop_data* is returned. Otherwise, -1 is returned and `errno` is set to indicate the error.

Errors These functions will fail if:

`EINVAL` One of the arguments is invalid.

`ENOTSUP` The snapshot contains no property information.

`ENXIO` The path property does not exist.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also di_init(3DEVINFO), libdevinfo(3LIB), attributes(5)

Writing Device Drivers

Name di_path_prop_next – libdevinfo path property traversal function

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

di_path_prop_t di_path_prop_next(di_path_t path,
     di_path_prop_t prop);
```

Parameters

path The handle to a path node in a `di_init`(3DEVINFO).

prop The handle to a property.

Description The `di_prop_next()` function returns a handle to the next property on the property list. If prop is `DI_PROP_NIL`, the handle to the first property is returned.

Return Values Upon successful completion, `di_path_prop_next()` returns a handle to a path property object. Otherwise `DI_PROP_NIL` is returned, and `errno` is set to indicate the error.

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

`EINVAL` An argument is invalid.

`ENOTSUP` The snapshot does not contain path property information (`DINFOPROP` was not passed to `di_init()`).

`ENXIO` There are no more properties.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libdevinfo`(3LIB), `attributes`(5)

Writing Device Drivers

Name di_prom_init, di_prom_fini – create and destroy a handle to the PROM device information

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

di_prom_handle_t di_prom_init(void);

void di_prom_fini(di_prom_handle_t  ph);
```

Parameters *ph* Handle to prom returned by di_prom_init().

Description For device nodes whose nodeid value is DI_PROM_NODEID (see di_nodeid(3DEVINFO)), additional properties can be retrieved from the PROM. The di_prom_init() function returns a handle that is used to retrieve such properties. This handle is passed to di_prom_prop_lookup_bytes(3DEVINFO) and di_prom_prop_next(3DEVINFO).

The di_prom_fini() function destroys the handle and all handles to the PROM device information obtained from that handle.

Return Values Upon successful completion, di_prom_init() returns a handle. Otherwise, DI_PROM_HANDLE_NIL is returned and errno is set to indicate the error.

Errors The di_prom_init() sets errno function to any error code that can also be set by openprom(7D) or malloc(3C).

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also di_nodeid(3DEVINFO), di_prom_prop_next(3DEVINFO), di_prom_prop_lookup_bytes(3DEVINFO), libdevinfo(3LIB), malloc(3C), attributes(5), openprom(7D)

Name di_prom_prop_data, di_prom_prop_next, di_prom_prop_name – access PROM device information

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

di_prom_prop_t di_prom_prop_next(di_prom_handle_t ph, di_node_t node,
     di_prom_prop_t prom_prop);

char *di_prom_prop_name(di_prom_prop_t prom_prop);

int di_prom_prop_data(di_prom_prop_t prom_prop, uchar_t **prop_data);
```

Parameters

node	Handle to a device node in the snapshot of kernel device tree.
ph	PROM handle
prom_prop	Handle to a PROM property.
prop_data	Address of a pointer.

Description The `di_prom_prop_next()` function obtains a handle to the next property on the PROM property list associated with *node*. If *prom_prop* is `DI_PROM_PROP_NIL`, the first property associated with *node* is returned.

The `di_prom_prop_name()` function returns the name of the *prom_prop* property.

The `di_prom_prop_data()` function returns the value of the *prom_prop* property. The return value is a non-negative integer specifying the size in number of bytes in *prop_data*.

All memory allocated by these functions is managed by the library and must not be freed by the caller.

Return Values The `di_prom_prop_data()` function returns the number of bytes in *prop_data* and *prop_data* is updated to point to a byte array containing the property value. If 0 is returned, the property is a boolean property and the existence of this property indicates the value is true.

The `di_prom_prop_name()` function returns a pointer to a string that contains the name of *prom_prop*.

The `di_prom_prop_next()` function returns a handle to the next PROM property. `DI_PROM_PROP_NIL` is returned if no additional properties exist.

Errors See `openprom(7D)` for a description of possible errors.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also attributes(5), openprom(7D)

Writing Device Drivers

Name di_prom_prop_lookup_bytes, di_prom_prop_lookup_ints, di_prom_prop_lookup_strings – search for a PROM property

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

int di_prom_prop_lookup_bytes(di_prom_handle_t ph, di_node_t node,
     const char *prop_name, uchar_t **prop_data);

int di_prom_prop_lookup_ints(di_prom_handle_t ph, di_node_t node,
     const char *prop_name, int **prop_data);

int di_prom_prop_lookup_strings(di_prom_handle_t ph, di_node_t node,
     const char *prop_name, char **prop_data);
```

Parameters

node Handle to device node in snapshot created by `di_init`(3DEVINFO).

ph Handle returned by `di_prom_init`(3DEVINFO).

prop_data For `di_prom_prop_lookup_bytes()`, the address of a pointer to an array of unsigned characters.

For `di_prom_prop_lookup_ints()`, the address of a pointer to an integer.

For `di_prom_prop_lookup_strings()`, the address of pointer to a buffer.

prop_name The name of the property being searched.

Description These functions return the value of a known PROM property name and value type and update the *prop_data* pointer to reference memory that contains the property value. All memory allocated by these functions is managed by the library and must not be freed by the caller.

Return Values If the property is found, the number of entries in *prop_data* is returned. If the property is a boolean type, 0 is returned and the existence of this property indicates the value is true. Otherwise, -1 is returned and `errno` is set to indicate the error.

For `di_prom_prop_lookup_bytes()`, the number of entries is the number of unsigned characters contained in the buffer pointed to by *prop_data*.

For `di_prom_prop_lookup_ints()`, the number of entries is the number of integers contained in the buffer pointed to by *prop_data*.

For `di_prom_prop_lookup_strings()`, the number of entries is the number of null-terminated strings contained in the buffer. The strings are stored in a concatenated format in the buffer.

Errors These functions will fail if::

`EINVAL` Invalid argument.

`ENXIO` The property does not exist.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also di_init(3DEVINFO), di_prom_prop_next(3DEVINFO), libdevinfo(3LIB), attributes(5), openprom(7D)

Writing Device Drivers

Name di_prop_bytes, di_prop_devt, di_prop_ints, di_prop_name, di_prop_strings, di_prop_type, di_prop_int64 – access property values and attributes

Synopsis
```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

int di_prop_bytes(di_prop_t prop, uchar_t **prop_data);

dev_t di_prop_devt(di_prop_t prop);

int di_prop_ints(di_prop_t prop, int **prop_data);

int di_prop_int64(di_prop_t prop, int64_t **prop_data);

char *di_prop_name(di_prop_t prop);

int di_prop_strings(di_prop_t prop, char **prop_data);

int di_prop_type(di_prop_t prop);
```

Parameters

prop Handle to a property returned by `di_prop_next`(3DEVINFO).

prop_data For `di_prop_bytes()`, the address of a pointer to an unsigned character.

For `di_prop_ints()`, the address of a pointer to an integer.

For `di_prop_int64()`, the address of a pointer to a 64–bit integer.

For `di_prop_strings()`, the address of pointer to a character.

Description These functions access information associated with property values and attributes. All memory allocated by these functions is managed by the library and must not be freed by the caller.

The `di_prop_bytes()` function returns the property data as a series of unsigned characters.

The `di_prop_devt()` function returns the `dev_t` with which this property is associated. If the value is `DDI_DEV_T_NONE`, the property is not associated with any specific minor node.

The `di_prop_ints()` function returns the property data as a series of integers.

The `di_prop_int64()` function returns the property data as a series of 64–bit integers.

The `di_prop_name()` function returns the name of the property.

The `di_prop_strings()` function returns the property data as a concatenation of null-terminated strings.

The `di_prop_type()` function returns the type of the property. The type determines the appropriate interface to access property values. The following is a list of possible types:

`DI_PROP_TYPE_BOOLEAN`	There is no interface to call since there is no property data associated with boolean properties. The existence of the property defines a `TRUE` value.
`DI_PROP_TYPE_INT`	Use `di_prop_ints()` to access property data.
`DI_PROP_TYPE_INT64`	Use `di_prop_int64()` to access property data.
`DI_PROP_TYPE_STRING`	Use `di_prop_strings()` to access property data.
`DI_PROP_TYPE_BYTE`	Use `di_prop_bytes()` to access property data.
`DI_PROP_TYPE_UNKNOWN`	Use `di_prop_bytes()` to access property data. Since the type of property is unknown, the caller is responsible for interpreting the contents of the data.
`DI_PROP_TYPE_UNDEF_IT`	The property has been undefined by the driver. No property data is available.

Return Values

Upon successful completion, `di_prop_bytes()`, `di_prop_ints()`, `di_prop_int64()`, and `di_prop_strings()` return a non-negative value, indicating the number of entries in the property value buffer. See `di_prom_prop_lookup_bytes`(3DEVINFO) for a description of the return values. Otherwise, -1 is returned and `errno` is set to indicate the error.

The `di_prop_devt()` function returns the `dev_t` value associated with the property.

The `di_prop_name()` function returns a pointer to a string containing the name of the property.

The `di_prop_type()` function can return one of types described in the DESCRIPTION section.

Errors

These functions will fail if:

`EINVAL` Invalid argument. For example, the property type does not match the interface.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also

`di_prom_prop_lookup_bytes`(3DEVINFO), `di_prop_next`(3DEVINFO), `libdevinfo`(3LIB), `attributes`(5)

Writing Device Drivers

Name di_prop_lookup_bytes, di_prop_lookup_ints, di_prop_lookup_int64, di_prop_lookup_strings, di_prop_exists – search for a property

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

int di_prop_lookup_bytes(dev_t dev, di_node_t node,
     const char *prop_name, uchar_t **prop_data);

int di_prop_lookup_ints(dev_t dev, di_node_t node,
     const char *prop_name, int **prop_data);

int di_prop_lookup_int64(dev_t dev, di_node_t node,
     const char *prop_name, int64_t **prop_data);

int di_prop_lookup_strings(dev_t dev, di_node_t node,
     const char *prop_name, char **prop_data);

int di_prop_exists(dev_t dev, di_node_t node,
     const char *prop_name);
```

Parameters

dev — `dev_t` of minor node with which the property is associated. `DDI_DEV_T_ANY` is a wild card that matches all `dev_t`'s, including `DDI_DEV_T_NONE`.

node — Handle to the device node with which the property is associated.

prop_data — For `di_prop_lookup_bytes()`, the address to a pointer to an array of unsigned characters containing the property data.

For `di_prop_lookup_ints()`, the address to a pointer to an array of integers containing the property data.

For `di_prop_lookup_int64()`, the address to a pointer to an array of 64–bit integers containing the property data.

For `di_prop_lookup_strings()`, the address to a pointer to a buffer containing a concatenation of null-terminated strings containing the property data.

prop_name — Name of the property for which to search.

Description These functions return the value of a known property name type and `dev_t` value. All memory allocated by these functions is managed by the library and must not be freed by the caller.

Return Values If the property is found, the number of entries in *prop_data* is returned. If the property is a boolean type, 0 is returned and the existence of this property indicates the value is true. Otherwise, -1 is returned and `errno` is set to indicate the error.

The `ddi_prop_exists()` returns 1 if a property, including a boolean property, exists, and 0 otherwise.

Errors These functions will fail if:

EINVAL Invalid argument.

ENOTSUP The snapshot contains no property information.

ENXIO The property does not exist; try `di_prom_prop_lookup_*()`.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `di_init`(3DEVINFO), `di_prom_prop_lookup_bytes`(3DEVINFO), `libdevinfo`(3LIB), `attributes`(5)

Writing Device Drivers

Name di_prop_next – libdevinfo property traversal function

Synopsis
```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

di_prop_t di_prop_next(di_node_t node, di_prop_t prop);
```

Parameters

node Handle to a device node.

prop Handle to a property.

Description The `di_prop_next()` function returns a handle to the next property on the property list. If *prop* is `DI_PROP_NIL`, the handle to the first property is returned.

Return Values Upon successful completion, `di_prop_next()` returns a handle. Otherwise `DI_PROP_NIL` is returned and `errno` is set to indicate the error.

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

`EINVAL` Invalid argument.

`ENOTSUP` The snapshot does not contain property information.

`ENXIO` There are no more properties.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `di_init`(3DEVINFO), `libdevinfo`(3LIB), `attributes`(5)

Writing Device Drivers

Name di_walk_link – traverse libdevinfo links

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

int di_walk_link(di_node_t root, uint_t flag, uint_t endpoint, void *arg,
     int (*link_callback)(di_link_t link, void *arg));
```

Parameters

root	The handle to the root node of the subtree to visit.
flag	Specify 0. Reserved for future use.
endpoint	Specify if the current node being visited should be the target or source of an link, either `DI_LINK_TGT` or `DI_LINK_SRC`
arg	A pointer to caller-specific data.
link_callback	The caller-supplied callback function.

Description The `di_walk_link()` function visits all nodes in the subtree rooted at *root*. For each node found, the caller-supplied function *link_callback*() is invoked for each link associated with that node where that node is the specified *endpoint* of the link. The return value of *link_callback*() specifies subsequent walking behavior. See RETURN VALUES.

Return Values Upon successful completion, `di_walk_link()` returns 0. Otherwise, -1 is returned and `errno` is set to indicate the error.

The callback function, *link_callback*(), can return one of the following:

`DI_WALK_CONTINUE`	Continue walking.
`DI_WALK_TERMINATE`	Terminate the walk immediately.

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

`EINVAL`	An argument is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `di_init(3DEVINFO)`, `libdevinfo(3LIB)`, `attributes(5)`

Name di_walk_lnode – traverse libdevinfo lnodes

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

int di_walk_lnode(di_node_t root, uint_t flag, void *arg,
     int (*lnode_callback)(di_lnode_t link, void *arg));
```

Parameters

root	The handle to the root node of the subtree to visit.
flag	Specify 0. Reserved for future use.
arg	A pointer to caller-specific data.
lnode_callback	The caller-supplied callback function.

Description The `di_walk_lnode()` function visits all nodes in the subtree rooted at *root*. For each node found, the caller-supplied function *lnode_callback*() is invoked for each lnode associated with that node. The return value of *lnode_callback*() specifies subsequent walking behavior where that node is the specified *endpoint* of the link.

Return Values Upon successful completion, `di_walk_lnode()` returns 0. Otherwise, -1 is returned and `errno` is set to indicate the error.

The callback function *lnode_callback*() can return one of the following:

`DI_WALK_CONTINUE`	Continue walking.
`DI_WALK_TERMINATE`	Terminate the walk immediately.

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

`EINVAL`	An argument is invalid.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `di_init(3DEVINFO)`, `libdevinfo(3LIB)`, `attributes(5)`

Name di_walk_minor – traverse libdevinfo minor nodes

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

int di_walk_minor(di_node_t root, const char *minor_nodetype,
     uint_t flag, void *arg, int (*minor_callback)(di_node_t node,
     di_minor_t minor, void *arg));
```

Parameters

arg Pointer to caller– specific user data.

flag Specify 0. Reserved for future use.

minor The minor node visited.

minor_nodetype A character string specifying the minor data type, which may be one of the types defined by the Solaris DDI framework, for example, `DDI_NT_BLOCK`. `NULL` matches all *minor_node* types. See `ddi_create_minor_node(9F)`.

node The device node with which to the minor node is associated.

root Root of subtree to visit.

Description The `di_walk_minor()` function visits all minor nodes attached to device nodes in a subtree rooted at *root*. For each minor node that matches *minor_nodetype*, the caller-supplied function *minor_callback*() is invoked. The walk terminates immediately when *minor_callback*() returns `DI_WALK_TERMINATE`.

Return Values Upon successful completion, `di_walk_minor()` returns 0. Otherwise, -1 is returned and errno is set to indicate the error.

The `minor_callback()` function returns one of the following:

`DI_WALK_CONTINUE` Continue to visit subsequent minor data nodes.

`DI_WALK_TERMINATE` Terminate the walk immediately.

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

`EINVAL` Invalid argument.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also dladm(1M), di_minor_nodetype(3DEVINFO), dlpi_walk(3DLPI), libdevinfo(3LIB), attributes(5), filesystem(5), ddi_create_minor_node(9F)

Writing Device Drivers

Notes The di_walk_minor() function is no longer an accurate method for walking network datalink interfaces on the system. Applications should use dlpi_walk(3DLPI) instead. It has been common for applications to use di_walk_minor() to walk networking devices by passing in a minor_nodetype of DDI_NT_NET, in most cases to discover the set of DLPI devices on the system. Solaris now makes a layering distinction between networking devices (the objects displayed in the DEVICE field by dladm show-phys) and network datalink interfaces (the objects displayed by dladm show-link). Datalink interfaces are represented as the set of DLPI device nodes that applications can open by using dlpi_open(3DLPI) or by opening DLPI nodes out of the /dev/net filesystem (see filesystem(5)). The dlpi_walk(3DLPI) function is the proper function to walk these nodes.

Name di_walk_node – traverse libdevinfo device nodes

Synopsis

```
cc [ flag... ] file... -ldevinfo [ library... ]
#include <libdevinfo.h>

int di_walk_node(di_node_t root, uint_t flag, void *arg,
     int (*node_callback)(di_node_t node, void *arg));
```

Description The `di_walk_node()` function visits all nodes in the subtree rooted at *root*. For each node found, the caller-supplied function *node_callback*() is invoked. The return value of *node_callback*() specifies subsequent walking behavior.

Parameters

arg Pointer to caller–specific data.

flag Specifies walking order, either `DI_WALK_CLDFIRST` (depth first) or `DI_WALK_SIBFIRST` (breadth first). `DI_WALK_CLDFIRST` is the default.

node The node being visited.

root The handle to the root node of the subtree to visit.

Return Values Upon successful completion, `di_walk_node()` returns 0. Otherwise, `-1` is returned and `errno` is set to indicate the error.

The *node_callback*() function can return one of the following:

`DI_WALK_CONTINUE` Continue walking.

`DI_WALK_PRUNESIB` Continue walking, but skip siblings and their child nodes.

`DI_WALK_PRUNECHILD` Continue walking, but skip subtree rooted at current node.

`DI_WALK_TERMINATE` Terminate the walk immediately.

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

`EINVAL` Invalid argument.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `di_init(3DEVINFO)`, `libdevinfo(3LIB)`, `attributes(5)`

Writing Device Drivers

Name ea_error – error interface to extended accounting library

Synopsis

```
cc [ flag... ] file... -lexacct [ library ... ]
#include <exacct.h>

int ea_error(void);
```

Description The `ea_error()` function returns the error value of the last failure recorded by the invocation of one of the functions of the extended accounting library, `libexacct`.

Return Values

`EXR_CORRUPT_FILE` A function failed because the file was not a valid exacct file.

`EXR_EOF` A function detected the end of the file, either when reading forwards or backwards through the file.

`EXR_INVALID_BUF` When unpacking an object, an invalid unpack buffer was specified.

`EXR_INVALID_OBJ` The object type passed to the function is not valid for the requested operation, for example passing a group object to `ea_set_item`(3EXACCT).

`EXR_NO_CREATOR` When creating a new file no creator was specified, or when opening a file for reading the creator value did not match the value in the file.

`EXR_NOTSUPP` An unsupported type of access was attempted, for example attempting to write to a file that was opened read-only.

`EXR_OK` The function completed successfully.

`EXR_SYSCALL_FAIL` A system call invoked by the function failed. The `errno` variable contains the error value set by the underlying call.

`EXR_UNKN_VERSION` The file referred to by name uses an `exacct` file version that cannot be processed by this library.

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

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

See Also `read`(2), `libexacct`(3LIB), `attributes`(5)

Name ea_open, ea_close – open or close exacct files

Synopsis
```
cc [ flag... ] file... -lexacct [ library... ]
#include <exacct.h>

int ea_open(ea_file_t *ef, char *name, char *creator, int aflags,
     int oflags, mode_t mode);

int ea_close(ea_file_t *ef);
```

Description The `ea_open()` function provides structured access to `exacct` files. The *aflags* argument contains the appropriate `exacct` flags necessary to describe the file. The *oflags* and *mode* arguments contain the appropriate flags and mode to open the file; see `<fcntl.h>`. If `ea_open()` is invoked with `EO_HEAD` specified in *aflags*, the resulting file is opened with the object cursor located at the first object of the file. If `ea_open()` is invoked with `EO_TAIL` specified in *aflags*, the resulting file is opened with the object cursor positioned beyond the last object in the file. If `EO_NO_VALID_HDR` is set in *aflags* along with `EO_HEAD`, the initial header record will be returned as the first item read from the file. When creating a file, the *creator* argument should be set (system generated files use the value "SunOS"); when reading a file, this argument should be set to `NULL` if no validation is required; otherwise it should be set to the expected value in the file.

The `ea_close()` function closes an open `exacct` file.

Return Values Upon successful completion, `ea_open()` and `ea_close()` return `0`. Otherwise they return `−1` and call `ea_error(3EXACCT)` to return the extended accounting error value describing the error.

Errors The `ea_open()` and `ea_close()` functions may fail if:

`EXR_SYSCALL_FAIL` A system call invoked by the function failed. The `errno` variable contains the error value set by the underlying call.

The `ea_open()` function may fail if:

`EXR_CORRUPT_FILE` The file referred to by *name* is not a valid `exacct` file.

`EXR_NO_CREATOR` In the case of file creation, the *creator* argument was `NULL`. In the case of opening an existing file, a *creator* argument was not NULL and does not match the *creator* item of the `exacct` file.

`EXR_UNKN_VERSION` The file referred to by *name* uses an `exacct` file version that cannot be processed by this library.

Usage The `exacct` file format can be used to represent data other than that in the extended accounting format. By using a unique creator type in the file header, application writers can develop their own format suited to the needs of their application.

Examples

EXAMPLE 1 Open and close exacct file.

The following example opens the extended accounting data file for processes. The `exacct` file is then closed.

```
#include <exacct.h>

ea_file_t ef;
if (ea_open(&ef, "/var/adm/exacct/proc", NULL, EO_HEAD,
    O_RDONLY, 0) == -1)
        exit(1);
(void) ea_close(&ef);
```

Attributes

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

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

See Also

`ea_error(3EXACCT)`, `ea_pack_object(3EXACCT)`, `ea_set_item(3EXACCT)`, `libexacct(3LIB)`, `attributes(5)`

Name ea_pack_object, ea_unpack_object, ea_get_creator, ea_get_hostname, ea_next_object, ea_previous_object, ea_get_object, ea_write_object, ea_copy_object, ea_copy_object_tree, ea_get_object_tree – construct, read, and write extended accounting records

Synopsis

```
cc [ flag... ] file... -lexacct [ library... ]
#include <exacct.h>

size_t ea_pack_object(ea_object_t *obj, void *buf,
     size_t bufsize);

ea_object_type_t ea_unpack_object(ea_object_t **objp, int flag,
     void *buf, size_t bufsize);

const char *ea_get_creator(ea_file_t *ef);

const char *ea_get_hostname(ea_file_t *ef);

ea_object_type_t ea_next_object(ea_file_t *ef, ea_object_t *obj);

ea_object_type_t ea_previous_object(ea_file_t *ef,
     ea_object_t *obj);

ea_object_type_t ea_get_object(ea_file_t *ef, ea_object_t *obj);

int ea_write_object(ea_file_t *ef, ea_object_t *obj);

ea_object_type_t *ea_copy_object(const ea_object_t *src);

ea_object_type_t *ea_copy_object_tree(const ea_object_t *src);

ea_object_type_t *ea_get_object_tree(ea_file_t *ef,
     uint32_tnobj);
```

Description The `ea_pack_object()` function converts `exacct` objects from their in-memory representation to their file representation. It is passed an object pointer that points to the top of an `exacct` object hierarchy representing one or more `exacct` records. It returns the size of the buffer required to contain the packed buffer representing the object hierarchy. To obtain the correct size of the required buffer, the *buf* and *bufsize* parameters can be set to `NULL` and 0 respectively, and the required buffer size will be returned. The resulting packed record can be passed to `putacct(2)` or to `ea_set_item(3EXACCT)` when constructing an object of type `EXT_EXACCT_OBJECT`.

The `ea_unpack_object()` function reverses the packing process performed by `ea_pack_object()`. A packed buffer passed to `ea_unpack_object()` is unpacked into the original hierarchy of objects. If the unpack operation fails (for example, due to a corrupted or incomplete buffer), it returns `EO_ERROR`; otherwise, the object type of the first object in the hierarchy is returned. If `ea_unpack_object()` is invoked with *flag* equal to `EUP_ALLOC`, it allocates memory for the variable-length data in the included objects. Otherwise, with *flag* equal to `EUP_NOALLOC`, it sets the variable length data pointers within the unpacked object structures to point within the buffer indicated by *buf*. In both cases, `ea_unpack_object()` allocates all the necessary `exacct` objects to represent the unpacked record. The resulting object hierarchy can be freed using `ea_free_object(3EXACCT)` with the same *flag* value.

The ea_get_creator() function returns a pointer to a string representing the recorded creator of the exacct file. The ea_get_hostname() function returns a pointer to a string representing the recorded hostname on which the exacct file was created. These functions will return NULL if their respective field was not recorded in the exacct file header.

The ea_next_object() function reads the basic fields (eo_catalog and eo_type) into the ea_object_t indicated by *obj* from the exacct file referred to by *ef* and rewinds to the head of the record. If the read object is corrupted, ea_next_object() returns EO_ERROR and records the extended accounting error code, accessible with ea_error(3EXACCT). If end-of-file is reached, EO_ERROR is returned and the extended accounting error code is set to EXR_EOF.

The ea_previous_object() function skips back one object in the file and reads its basic fields (eo_catalog and eo_type) into the indicated ea_object_t. If the read object is corrupted, ea_previous_object() returns EO_ERROR and records the extended accounting error code, accessible with ea_error(3EXACCT). If end-of-file is reached, EO_ERROR is returned and the extended accounting error code is set to EXR_EOF.

The ea_get_object() function reads the value fields into the ea_object_t indicated by *obj*, allocating memory as necessary, and advances to the head of the next record. Once a record group object is retrieved using ea_get_object(), subsequent calls to ea_get_object() and ea_next_object() will track through the objects within the record group, and on reaching the end of the group, will return the next object at the same level as the group from the file. If the read object is corrupted, ea_get_object() returns EO_ERROR and records the extended accounting error code, accessible with ea_error(3EXACCT). If end-of-file is reached, EO_ERROR is returned and the extended accounting error code is set to EXR_EOF.

The ea_write_object() function appends the given object to the open exacct file indicated by *ef* and returns 0. If the write fails, ea_write_object() returns −1 and sets the extended accounting error code to indicate the error, accessible with ea_error(3EXACCT).

The ea_copy_object() function copies an ea_object_t. If the source object is part of a chain, only the current object is copied. If the source object is a group, only the group object is copied without its list of members and the eg_nobjs and eg_objs fields are set to 0 and NULL, respectively. Use ea_copy_tree() to copy recursively a group or a list of items.

The ea_copy_object_tree() function recursively copies an ea_object_t. All elements in the eo_next list are copied, and any group objects are recursively copied. The returned object can be completely freed with ea_free_object(3EXACCT) by specifying the EUP_ALLOC flag.

The ea_get_object_tree() function reads in *nobj* top-level objects from the file, returning the same data structure that would have originally been passed to ea_write_object(). On encountering a group object, the ea_get_object() function reads only the group header part of the group, whereas ea_get_object_tree() reads the group and all its member items, recursing into sub-records if necessary. The returned object data structure can be completely freed with ea_free_object() by specifying the EUP_ALLOC flag.

Return Values The `ea_pack_object()` function returns the number of bytes required to hold the `exacct` object being operated upon. If the returned size exceeds *bufsize*, the pack operation does not complete and the function returns `(size_t)` −1 and sets the extended accounting error code to indicate the error.

The `ea_get_object()` function returns the `ea_object_type` of the object if the object was retrieved successfully. Otherwise, it returns `EO_ERROR` and sets the extended accounting error code to indicate the error.

The `ea_next_object()` function returns the `ea_object_type` of the next `exacct` object in the file. It returns `EO_ERROR` if the `exacct` file is corrupted sets the extended accounting error code to indicate the error.

The `ea_unpack_object()` function returns the `ea_object_type` of the first `exacct` object unpacked from the buffer. It returns `EO_ERROR` if the exacct file is corrupted, and sets the extended accounting error code to indicate the error.

The `ea_write_object()` function returns 0 on success. Otherwise it returns −1 and sets the extended accounting error code to indicate the error.

The `ea_copy_object()` and `ea_copy_object_tree()` functions return the copied object on success. Otherwise they return `NULL` and set the extended accounting error code to indicate the error.

The `ea_get_object_tree()` function returns the list of objects read from the file on success. Otherwise it returns `NULL` and sets the extended accounting error code to indicate the error.

The extended account error code can be retrieved using `ea_error`(3EXACCT).

Errors These functions may fail if:

`EXR_SYSCALL_FAIL`
: A system call invoked by the function failed. The `errno` variable contains the error value set by the underlying call. On memory allocation failure, `errno` will be set to `ENOMEM`.

`EXR_CORRUPT_FILE`
: The file referred to by *name* is not a valid `exacct` file, or is unparsable, and therefore appears corrupted. This error is also used by `ea_unpack_buffer()` to indicate a corrupted buffer.

`EXR_EOF`
: The end of the file has been reached. In the case of `ea_previous_record()`, the previous record could not be reached, either because the head of the file was encountered or because the previous record could not be skipped over.

Usage The `exacct` file format can be used to represent data other than that in the extended accounting format. By using a unique creator type in the file header, application writers can develop their own format suited to the needs of their application.

Examples

EXAMPLE 1 Open and close exacct file.

The following example opens the extended accounting data file for processes. The `exacct` file is then closed.

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

ea_file_t ef;
ea_object_t *obj;

...

ea_open(&ef, "foo", O_RDONLY, ...);

while ((obj = ea_get_object_tree(&ef, 1)) != NULL) {
    if (obj->eo_type == EO_ITEM) {
        /* handle item */
    } else {
        /* handle group */
    }
    ea_free_object(obj, EUP_ALLOC);
}

if (ea_error() != EXR_EOF) {
    /* handle error */
}

ea_close(&ef);
```

EXAMPLE 2 Construct an exacct file consisting of a single object containing the current process ID.

```
#include <sys/types.h>
#include <unistd.h>
#include <exacct.h>

...

ea_file_t ef;
ea_object_t obj;
pid_t my_pid;

ea_open(&ef, "foo", O_CREAT | O_WRONLY, ...);

my_pid = getpid();
ea_set_item(&obj, EXT_UINT32 | EXC_DEFAULT | EXT_PROC_PID, &my_pid, 0);
(void) ea_write_object(&ef, &obj);

ea_close(&ef);
```

EXAMPLE 2 Construct an exacct file consisting of a single object containing the current process ID. *(Continued)*

```
...
```

Attributes

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

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

See Also

`read(2)`, `ea_error(3EXACCT)`, `ea_open(3EXACCT)`, `ea_set_item(3EXACCT)`, `libexacct(3LIB)`, `attributes(5)`

Name ea_set_item, ea_alloc, ea_strdup, ea_set_group, ea_match_object_catalog, ea_attach_to_object, ea_attach_to_group, ea_free, ea_strfree, ea_free_item, ea_free_object – create, destroy and manipulate exacct objects

Synopsis

```
cc [ flag... ] file... -lexacct [ library... ]
#include <exacct.h>

int ea_set_item(ea_object_t *obj, ea_catalog_t tag, void *value,
     size_t valsize);

void *ea_alloc(size_t size);

char *ea_strdup(char *ptr);

int ea_set_group(ea_object_t *obj, ea_catalog_t tag);

int ea_match_object_catalog(ea_object_t *obj, ea_catalog_t catmask);

void ea_attach_to_object(ea_object_t *head_obj, ea_object_t *obj);

void ea_attach_to_group(ea_object_t *group_obj, ea_object_t *obj);

void ea_free(void *ptr, size_t size);

void ea_strfree(char *ptr);

int ea_free_item(ea_object_t *obj, int flag);

void ea_free_object(ea_object_t *obj, int flag);
```

Description The `ea_alloc()` function allocates a block of memory of the requested size. This block can be safely passed to `libexacct` functions, and can be safely freed by any of the `ea_free()` functions.

The `ea_strdup()` function can be used to duplicate a string that is to be stored inside an `ea_object_t` structure.

The `ea_set_item()` function assigns the given `exacct` object to be a data item with *value* set according to the remaining arguments. For buffer-based data values (`EXT_STRING`, `EXT_EXACCT_OBJECT`, and `EXT_RAW`), a copy of the passed buffer is taken. In the case of `EXT_EXACCT_OBJECT`, the passed buffer should be a packed `exacct` object as returned by `ea_pack_object`(3EXACCT). Any item assigned with `ea_set_item()` should be freed with `ea_free_item()` specifying a flag value of `EUP_ALLOC` when the item is no longer needed.

The `ea_match_object_catalog()` function returns `TRUE` if the `exacct` object specified by *obj* has a catalog tag that matches the mask specified by *catmask*.

The `ea_attach_to_object()` function attaches an object to the given object. The `ea_attach_to_group()` function attaches a chain of objects as member items of the given group. Objects are inserted at the end of the list of any previously attached objects.

The `ea_free()` function frees a block of memory previously allocated by `ea_alloc()`.

The `ea_strfree()` function frees a string previously copied by `ea_strdup()`.

The `ea_free_item()` function frees the *value* fields in the `ea_object_t` indicated by *obj*, if `EUP_ALLOC` is specified. The object itself is not freed. The `ea_free_object()` function frees the specified object and any attached hierarchy of objects. If the *flag* argument is set to `EUP_ALLOC`, `ea_free_object()` will also free any variable-length data in the object hierarchy; if set to `EUP_NOALLOC`, `ea_free_object()` will not free variable-length data. In particular, these flags should correspond to those specified in calls to `ea_unpack_object`(3EXACCT).

Return Values The `ea_match_object_catalog()` function returns 0 if the object's catalog tag does not match the given mask, and 1 if there is a match.

Other integer-valued functions return 0 if successful. Otherwise these functions return -1 and set the extended accounting error code appropriately. Pointer-valued functions return a valid pointer if successful and NULL otherwise, setting the extended accounting error code appropriately. The extended accounting error code can be examined with `ea_error`(3EXACCT).

Errors The `ea_set_item()`, `ea_set_group()`, and `ea_match_object_catalog()` functions may fail if:

`EXR_SYSCALL_FAIL` A system call invoked by the function failed. The `errno` variable contains the error value set by the underlying call.

`EXR_INVALID_OBJECT` The passed object is of an incorrect type, for example passing a group object to `ea_set_item()`.

Usage The `exacct` file format can be used to represent data other than that in the extended accounting format. By using a unique creator type in the file header, application writers can develop their own format suited to the needs of their application.

Examples **EXAMPLE 1** Open and close exacct file.

Construct an exacct file consisting of a single object containing the current process ID.

```
#include <sys/types.h>
#include <unistd.h>
#include <exacct.h>

...

ea_file_t ef;
ea_object_t obj;
pid_t my_pid;

my_pid = getpid();
ea_set_item(&obj, EXT_UINT32 | EXC_DEFAULT | EXT_PROC_PID,
       &my_pid, sizeof(my_pid));

...
```

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

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

See Also read(2), ea_error(3EXACCT), ea_open(3EXACCT), ea_pack_object(3EXACCT), libexacct(3LIB), attributes(5)

Name ecb_crypt, cbc_crypt, des_setparity, DES_FAILED – fast DES encryption

Synopsis

```
#include <rpc/des_crypt.h>

int ecb_crypt(char *key, char *data, unsigned datalen,
     unsigned mode);

int cbc_crypt(char *key, char *data, unsigned datalen,
     unsigned mode, char *ivec);

void des_setparity(char *key);

int DES_FAILED(int stat);
```

Description `ecb_crypt()` and `cbc_crypt()` implement the NBS DES (Data Encryption Standard). These routines are faster and more general purpose than `crypt(3C)`. They also are able to utilize DES hardware if it is available. `ecb_crypt()` encrypts in ECB (Electronic Code Book) mode, which encrypts blocks of data independently. `cbc_crypt()` encrypts in CBC (Cipher Block Chaining) mode, which chains together successive blocks. CBC mode protects against insertions, deletions, and substitutions of blocks. Also, regularities in the clear text will not appear in the cipher text.

The first parameter, *key*, is the 8-byte encryption key with parity. To set the key's parity, which for DES is in the low bit of each byte, use `des_setparity()`. The second parameter, *data*, contains the data to be encrypted or decrypted. The third parameter, *datalen*, is the length in bytes of *data*, which must be a multiple of 8. The fourth parameter, *mode*, is formed by OR'ing together the DES_ENCRYPT or DES_DECRYPT to specify the encryption direction and DES_HW or DES_SW to specify software or hardware encryption. If DES_HW is specified, and there is no hardware, then the encryption is performed in software and the routine returns DESERR_NOHWDEVICE.

For `cbc_crypt()`, the parameter *ivec* is the 8-byte initialization vector for the chaining. It is updated to the next initialization vector upon successful return.

Return Values Given a result status *stat*, the macro `DES_FAILED` is false only for the first two statuses.

`DESERR_NONE` No error.

`DESERR_NOHWDEVICE` Encryption succeeded, but done in software instead of the requested hardware.

`DESERR_HWERROR` An error occurred in the hardware or driver.

`DESERR_BADPARAM` Bad parameter to routine.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also crypt(3C), attributes(5)

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

Name efi_alloc_and_init, efi_alloc_and_read, efi_free, efi_write, efi_use_whole_disk – manipulate a disk's EFI Partition Table

Synopsis

```
cc [ flag ... ] file... -lefi [ library ... ]
#include <sys/vtoc.h>
#include <sys/efi_partition.h>

int efi_alloc_and_init(int fd, uint32_t nparts, dk_gpt_t **vtoc);

int efi_alloc_and_read(int fd, dk_gpt_t **vtoc);

void efi_free(dk_gpt_t *vtoc);

int efi_write(int fd, dk_gpt_t *vtoc);

int efi_use_whole_disk(int fd);
```

Description The `efi_alloc_and_init()` function initializes the `dk_gpt_t` structure specified by *vtoc* in preparation for a call to `efi_write()`. It calculates and initializes the `efi_version`, `efi_lbasize`, `efi_nparts`, `efi_first_u_lba`, `efi_last_lba`, and `efi_last_u_lba` members of this sturcture. The caller can then set the `efi_nparts` member.

The `efi_alloc_and_read()` function allocates memory and returns the partition table.

The `efi_free()` function frees the memory allocated by `efi_alloc_and_init()` and `efi_alloc_and_read()`.

The `efi_write()` function writes the EFI partition table.

The `efi_use_whole_disk()` function takes any space that is not contained in the disk label and adds it into the EFI label. If the reserved partition is right before the backup label, add the space to the last physically non-zero area before the reserved partition. Otherwise, add the space to the last physically non-zero area before the backup label.

The *fd* argument refers to any partition on a raw disk, opened with `O_NDELAY`. See open(2).

The *nparts* argument specifies the number of desired partitions.

The *vtoc* argument is a `dk_gpt_t` structure that describes an EFI partition table and contains at least the following members:

```
uint_t        efi_version;     /* set to EFI_VERSION_CURRENT */
uint_t        efi_nparts;      /* index of last user-defined  */
                               /* (non-zero) partition in efi_parts */
                               /* plus one */
uint_t        efi_lbasize;     /* size of block in bytes */
diskaddr_t    efi_last_lba;    /* last block on the disk */
diskaddr_t    efi_first_u_lba; /* first block after labels */
diskaddr_t    efi_last_u_lba;  /* last block before backup labels */
uint_t        efi_num_of_partition_entries /* number of partitions */
                               /* in efi_parts, representing actual */
```

```
                                    /* GUID partition entries allocated */
                                    /* on disk */
struct dk_part efi_parts[];         /* array of partitions */
```

Return Values Upon successful completion, `efi_alloc_and_init()` returns 0. Otherwise it returns `VT_EIO` if an I/O operation to the disk fails.

Upon successful completion, `efi_alloc_and_read()` returns a positive integer indicating the partition index associated with the open file descriptor. Otherwise, it returns a negative integer to indicate one of the following:

`VT_EIO` An I/O error occurred.

`VT_ERROR` An unknown error occurred.

`VT_EINVAL` An EFI label was not found.

Upon successful completion, `efi_write()` returns 0. Otherwise, it returns a negative integer to indicate one of the following:

`VT_EIO` An I/O error occurred.

`VT_ERROR` An unknown error occurred.

`VT_EINVAL` The label contains incorrect data.

Upon successful completion, `efi_use_whole_disk()` returns 0. Otherwise, it returns a negative integer to indicate one of the following:

`VT_EIO` An I/O error occurred.

`VT_ERROR` An unknown error occurred.

`VT_EINVAL` The label contains incorrect data.

`VT_ENOSPC` Space out of label was not found.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe

See Also `fmthard(1M)`, `format(1M)`, `prtvtoc(1M)`, `ioctl(2)`, `open(2)`, `libefi(3LIB)`, `read_vtoc(3EXT)`, `attributes(5)`, `dkio(7I)`

Name elf32_checksum, elf64_checksum – return checksum of elf image

Synopsis

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

long elf32_checksum(Elf *elf);

long elf64_checksum(Elf *elf);
```

Description The `elf32_checksum()` function returns a simple checksum of selected sections of the image identified by *elf*. The value is typically used as the `.dynamic` tag `DT_CHECKSUM`, recorded in dynamic executables and shared objects.

Selected sections of the image are used to calcluate the checksum in order that its value is not affected by utilities such as `strip(1)`.

For the 64–bit class, replace 32 with 64 as appropriate.

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

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

See Also `elf(3ELF)`, `elf_version(3ELF)`, `gelf(3ELF)`, `libelf(3LIB)`, `attributes(5)`

Name elf32_fsize, elf64_fsize – return the size of an object file type

Synopsis

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

size_t elf32_fsize(Elf_Type type, size_t count, unsigned ver);

size_t elf64_fsize(Elf_Type type, size_t count, unsigned ver);
```

Description `elf32_fsize()` gives the size in bytes of the 32-bit file representation of *count* data objects with the given `type`. The library uses version `ver` to calculate the size. See `elf(3ELF)` and `elf_version(3ELF)`.

Constant values are available for the sizes of fundamental types:

```
Elf_Type    File Size    Memory Size
ELF_T_ADDR    ELF32_FSZ_ADDR    sizeof(Elf32_Addr)
ELF_T_BYTE    1    sizeof(unsigned char)
ELF_T_HALF    ELF32_FSZ_HALF    sizeof(Elf32_Half)
ELT_T_OFF    ELF32_FSZ_OFF    sizeof(Elf32_Off)
ELF_T_SWORD    ELF32_FSZ_SWORD    sizeof(Elf32_Sword)
ELF_T_WORD    ELF32_FSZ_WORD    sizeof(Elf32_Word)
```

`elf32_fsize()` returns `0` if the value of `type` or `ver` is unknown. See `elf32_xlatetof(3ELF)` for a list of the `type` values.

For the 64–bit class, replace 32 with 64 as appropriate.

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

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

See Also `elf(3ELF)`, `elf32_xlatetof(3ELF)`, `elf_version(3ELF)`, `libelf(3LIB)`, `attributes(5)`

Name elf32_getehdr, elf32_newehdr, elf64_getehdr, elf64_newehdr – retrieve class-dependent object file header

Synopsis
```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

Elf32_Ehdr *elf32_getehdr(Elf *elf);

Elf32_Ehdr *elf32_newehdr(Elf *elf);

Elf64_Ehdr *elf64_getehdr(Elf *elf);

Elf64_Ehdr *elf64_newehdr(Elf *elf);
```

Description For a 32-bit class file, `elf32_getehdr()` returns a pointer to an ELF header, if one is available for the ELF descriptor *elf*. If no header exists for the descriptor, `elf32_newehdr()` allocates a clean one, but it otherwise behaves the same as `elf32_getehdr()`. It does not allocate a new header if one exists already. If no header exists for `elf32_getehdr()`, one cannot be created for `elf32_newehdr()`, a system error occurs, the file is not a 32-bit class file, or *elf* is `NULL`, both functions return a null pointer.

For the 64–bit class, replace 32 with 64 as appropriate.

The header includes the following members:

```
unsigned char    e_ident[EI_NIDENT];
Elf32_Half    e_type;
Elf32_Half    e_machine;
Elf32_Word    e_version;
Elf32_Addr    e_entry;
Elf32_Off     e_phoff;
Elf32_Off     e_shoff;
Elf32_Word    e_flags;
Elf32_Half    e_ehsize;
Elf32_Half    e_phentsize;
Elf32_Half    e_phnum;
Elf32_Half    e_shentsize;
Elf32_Half    e_shnum;
Elf32_Half    e_shstrndx;
```

The `elf32_newehdr()` function automatically sets the `ELF_F_DIRTY` bit. See `elf_flagdata`(3ELF).

An application can use `elf_getident()` to inspect the identification bytes from a file.

An application can use `elf_getshnum()` and `elf_getshstrndx()` to obtain section header information. The location of this section header information differs between standard ELF files to those that require Extended Sections.

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

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

See Also elf(3ELF), elf_begin(3ELF), elf_flagdata(3ELF), elf_getident(3ELF), elf_getshnum(3ELF), elf_getshstrndx(3ELF), libelf(3LIB), attributes(5)

Name elf32_getphdr, elf32_newphdr, elf64_getphdr, elf64_newphdr – retrieve class-dependent program header table

Synopsis
```
cc [ flag ... ] file... -lelf [ library ... ]
#include <libelf.h>

Elf32_Phdr *elf32_getphdr(Elf *elf);

Elf32_Phdr *elf32_newphdr(Elf *elf, size_t count);

Elf64_Phdr *elf64_getphdr(Elf *elf);

Elf64_Phdr *elf64_newphdr(Elf *elf, size_t count);
```

Description For a 32-bit class file, `elf32_getphdr()` returns a pointer to the program execution header table, if one is available for the ELF descriptor *elf*.

`elf32_newphdr()` allocates a new table with *count* entries, regardless of whether one existed previously, and sets the `ELF_F_DIRTY` bit for the table. See `elf_flagdata(3ELF)`. Specifying a zero *count* deletes an existing table. Note this behavior differs from that of `elf32_newehdr()` allowing a program to replace or delete the program header table, changing its size if necessary. See `elf32_getehdr(3ELF)`.

If no program header table exists, the file is not a 32-bit class file, an error occurs, or *elf* is `NULL,` both functions return a null pointer. Additionally, `elf32_newphdr()` returns a null pointer if *count* is `0`.

The table is an array of `Elf32_Phdr` structures, each of which includes the following members:

```
Elf32_Word    p_type;
Elf32_Off    p_offset;
Elf32_Addr    p_vaddr;
Elf32_Addr    p_paddr;
Elf32_Word    p_filesz;
Elf32_Word    p_memsz;
Elf32_Word    p_flags;
Elf32_Word    p_align;
```

The `Elf64_Phdr` structures include the following members:

```
Elf64_Word    p_type;
Elf64_Word    p_flags;
Elf64_Off    p_offset;
Elf64_Addr    p_vaddr;
Elf64_Addr    p_paddr;
Elf64_Xword    p_filesz;
Elf64_Xword    p_memsz;
Elf64_Xword    p_align;
```

For the 64–bit class, replace 32 with 64 as appropriate.

The ELF header's e_phnum member tells how many entries the program header table has. See elf32_getehdr(3ELF). A program may inspect this value to determine the size of an existing table; elf32_newphdr() automatically sets the member's value to *count*. If the program is building a new file, it is responsible for creating the file's ELF header before creating the program header table.

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

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

See Also elf(3ELF), elf32_getehdr(3ELF), elf_begin(3ELF), elf_flagdata(3ELF), libelf(3LIB), attributes(5)

Name elf32_getshdr, elf64_getshdr – retrieve class-dependent section header

Synopsis

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

Elf32_Shdr *elf32_getshdr(Elf_Scn *scn);

Elf64_Shdr *elf64_getshdr(Elf_Scn *scn);
```

Description For a 32-bit class file, `elf32_getshdr()` returns a pointer to a section header for the section descriptor *scn*. Otherwise, the file is not a 32-bit class file, *scn* was `NULL`, or an error occurred; `elf32_getshdr()` then returns `NULL`.

The `elf32_getshdr` header includes the following members:

```
Elf32_Word    sh_name;
Elf32_Word    sh_type;
Elf32_Word    sh_flags;
Elf32_Addr    sh_addr;
Elf32_Off    sh_offset;
Elf32_Word    sh_size;
Elf32_Word    sh_link;
Elf32_Word    sh_info;
Elf32_Word    sh_addralign;
Elf32_Word    sh_entsize;
```

while the `elf64_getshdr` header includes the following members:

```
Elf64_Word    sh_name;
Elf64_Word    sh_type;
Elf64_Xword    sh_flags;
Elf64_Addr    sh_addr;
Elf64_Off    sh_offset;
Elf64_Xword    sh_size;
Elf64_Word    sh_link;
Elf64_Word    sh_info;
Elf64_Xword    sh_addralign;
Elf64_Xword    sh_entsize;
```

For the 64–bit class, replace 32 with 64 as appropriate.

If the program is building a new file, it is responsible for creating the file's ELF header before creating sections.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `elf`(3ELF), `elf_flagdata`(3ELF), `elf_getscn`(3ELF), `elf_strptr`(3ELF), `libelf`(3LIB), `attributes`(5)

Name elf32_xlatetof, elf32_xlatetom, elf64_xlatetof, elf64_xlatetom – class-dependent data translation

Synopsis

```
cc [ flag ... ] file... -lelf [ library ... ]
#include <libelf.h>

Elf_Data *elf32_xlatetof(Elf_Data *dst, const Elf_Data *src,
     unsigned encode);

Elf_Data *elf32_xlatetom(Elf_Data *dst, const Elf_Data *src,
     unsigned encode);

Elf_Data *elf64_xlatetof(Elf_Data *dst, const Elf_Data *src,
     unsigned encode);

Elf_Data *elf64_xlatetom(Elf_Data *dst, const Elf_Data *src,
     unsigned encode);
```

Description `elf32_xlatetom()` translates various data structures from their 32-bit class file representations to their memory representations; `elf32_xlatetof()` provides the inverse. This conversion is particularly important for cross development environments. *src* is a pointer to the source buffer that holds the original data; *dst* is a pointer to a destination buffer that will hold the translated copy. *encode* gives the byte encoding in which the file objects are to be represented and must have one of the encoding values defined for the ELF header's `e_ident[EI_DATA]` entry (see `elf_getident(3ELF)`). If the data can be translated, the functions return *dst*. Otherwise, they return `NULL` because an error occurred, such as incompatible types, destination buffer overflow, etc.

`elf_getdata(3ELF)` describes the `Elf_Data` descriptor, which the translation routines use as follows:

`d_buf` Both the source and destination must have valid buffer pointers.

`d_type` This member's value specifies the type of the data to which `d_buf` points and the type of data to be created in the destination. The program supplies a `d_type` value in the source; the library sets the destination's `d_type` to the same value. These values are summarized below.

`d_size` This member holds the total size, in bytes, of the memory occupied by the source data and the size allocated for the destination data. If the destination buffer is not large enough, the routines do not change its original contents. The translation routines reset the destination's `d_size` member to the actual size required, after the translation occurs. The source and destination sizes may differ.

`d_version` This member holds the version number of the objects (desired) in the buffer. The source and destination versions are independent.

Translation routines allow the source and destination buffers to coincide. That is, `dst→d_buf` may equal `src→d_buf`. Other cases where the source and destination buffers overlap give undefined behavior.

```
Elf_Type       32-Bit Memory Type
ELF_T_ADDR    Elf32_Addr
ELF_T_BYTE    unsigned char
ELF_T_DYN    Elf32_Dyn
ELF_T_EHDR    Elf32_Ehdr
ELF_T_HALF    Elf32_Half
ELT_T_OFF    Elf32_Off
ELF_T_PHDR    Elf32_Phdr
ELF_T_REL    Elf32_Rel
ELF_T_RELA    Elf32_Rela
ELF_T_SHDR    Elf32_Shdr
ELF_T_SWORD    Elf32_Sword
ELF_T_SYM    Elf32_Sym
ELF_T_WORD    Elf32_Word
```

Translating buffers of type `ELF_T_BYTE` does not change the byte order.

For the 64–bit class, replace 32 with 64 as appropriate.

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

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

See Also `elf(3ELF)`, `elf32_fsize(3ELF)`, `elf_getdata(3ELF)`, `elf_getident(3ELF)`, `libelf(3LIB)`, `attributes(5)`

Name elf – object file access library

Synopsis

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>
```

Description Functions in the ELF access library let a program manipulate ELF (Executable and Linking Format) object files, archive files, and archive members. The header provides type and function declarations for all library services.

Programs communicate with many of the higher-level routines using an *ELF descriptor*. That is, when the program starts working with a file, `elf_begin(3ELF)` creates an ELF descriptor through which the program manipulates the structures and information in the file. These ELF descriptors can be used both to read and to write files. After the program establishes an ELF descriptor for a file, it may then obtain *section descriptors* to manipulate the sections of the file (see `elf_getscn(3ELF)`). Sections hold the bulk of an object file's real information, such as text, data, the symbol table, and so on. A section descriptor "belongs" to a particular ELF descriptor, just as a section belongs to a file. Finally, *data descriptors* are available through section descriptors, allowing the program to manipulate the information associated with a section. A data descriptor "belongs" to a section descriptor.

Descriptors provide private handles to a file and its pieces. In other words, a data descriptor is associated with one section descriptor, which is associated with one ELF descriptor, which is associated with one file. Although descriptors are private, they give access to data that may be shared. Consider programs that combine input files, using incoming data to create or update another file. Such a program might get data descriptors for an input and an output section. It then could update the output descriptor to reuse the input descriptor's data. That is, the descriptors are distinct, but they could share the associated data bytes. This sharing avoids the space overhead for duplicate buffers and the performance overhead for copying data unnecessarily.

File Classes ELF provides a framework in which to define a family of object files, supporting multiple processors and architectures. An important distinction among object files is the *class*, or capacity, of the file. The 32-bit class supports architectures in which a 32-bit object can represent addresses, file sizes, and so on, as in the following:

Name	Purpose
`Elf32_Addr`	Unsigned address
`Elf32_Half`	Unsigned medium integer
`Elf32_Off`	Unsigned file offset
`Elf32_Sword`	Signed large integer
`Elf32_Word`	Unsigned large integer
`unsigned char`	Unsigned small integer

The 64–bit class works the same as the 32–bit class, substituting 64 for 32 as necessary. Other classes will be defined as necessary, to support larger (or smaller) machines. Some library services deal only with data objects for a specific class, while others are class-independent. To make this distinction clear, library function names reflect their status, as described below.

Data Representation

Conceptually, two parallel sets of objects support cross compilation environments. One set corresponds to file contents, while the other set corresponds to the native memory image of the program manipulating the file. Type definitions supplied by the headers work on the native machine, which may have different data encodings (size, byte order, and so on) than the target machine. Although native memory objects should be at least as big as the file objects (to avoid information loss), they may be bigger if that is more natural for the host machine.

Translation facilities exist to convert between file and memory representations. Some library routines convert data automatically, while others leave conversion as the program's responsibility. Either way, programs that create object files must write file-typed objects to those files; programs that read object files must take a similar view. See `elf32_xlatetof`(3ELF) and `elf32_fsize`(3ELF) for more information.

Programs may translate data explicitly, taking full control over the object file layout and semantics. If the program prefers not to have and exercise complete control, the library provides a higher-level interface that hides many object file details. `elf_begin()` and related functions let a program deal with the native memory types, converting between memory objects and their file equivalents automatically when reading or writing an object file.

ELF Versions

Object file versions allow ELF to adapt to new requirements. *Three independent versions* can be important to a program. First, an application program knows about a particular version by virtue of being compiled with certain headers. Second, the access library similarly is compiled with header files that control what versions it understands. Third, an ELF object file holds a value identifying its version, determined by the ELF version known by the file's creator. Ideally, all three versions would be the same, but they may differ.

If a program's version is newer than the access library, the program might use information unknown to the library. Translation routines might not work properly, leading to undefined behavior. This condition merits installing a new library.

The library's version might be newer than the program's and the file's. The library understands old versions, thus avoiding compatibility problems in this case.

Finally, a file's version might be newer than either the program or the library understands. The program might or might not be able to process the file properly, depending on whether the file has extra information and whether that information can be safely ignored. Again, the safe alternative is to install a new library that understands the file's version.

To accommodate these differences, a program must use `elf_version`(3ELF) to pass its version to the library, thus establishing the *working version* for the process. Using this, the library accepts data from and presents data to the program in the proper representations.

When the library reads object files, it uses each file's version to interpret the data. When writing files or converting memory types to the file equivalents, the library uses the program's working version for the file data.

System Services

As mentioned above, `elf_begin()` and related routines provide a higher-level interface to ELF files, performing input and output on behalf of the application program. These routines assume a program can hold entire files in memory, without explicitly using temporary files. When reading a file, the library routines bring the data into memory and perform subsequent operations on the memory copy. Programs that wish to read or write large object files with this model must execute on a machine with a large process virtual address space. If the underlying operating system limits the number of open files, a program can use `elf_cntl`(3ELF) to retrieve all necessary data from the file, allowing the program to close the file descriptor and reuse it.

Although the `elf_begin()` interfaces are convenient and efficient for many programs, they might be inappropriate for some. In those cases, an application may invoke the `elf32_xlatetom`(3ELF) or `elf32_xlatetof`(3ELF) data translation routines directly. These routines perform no input or output, leaving that as the application's responsibility. By assuming a larger share of the job, an application controls its input and output model.

Library Names

Names associated with the library take several forms.

`elf_`*name*
: These class-independent names perform some service, *name*, for the program.

`elf32_`*name*
: Service names with an embedded class, `32` here, indicate they work only for the designated class of files.

`Elf_`*Type*
: Data types can be class-independent as well, distinguished by *Type*.

`Elf32_`*Type*
: Class-dependent data types have an embedded class name, `32` here.

`ELF_C_`*CMD*
: Several functions take commands that control their actions. These values are members of the `Elf_Cmd` enumeration; they range from zero through `ELF_C_NUM−1`.

`ELF_F_`*FLAG*
: Several functions take flags that control library status and/or actions. Flags are bits that may be combined.

`ELF32_FSZ_`*TYPE*
: These constants give the file sizes in bytes of the basic ELF types for the 32-bit class of files. See `elf32_fsize()` for more information.

`ELF_K_`*KIND*
: The function `elf_kind()` identifies the *KIND* of file associated with an ELF descriptor. These values are members of the `Elf_Kind` enumeration; they range from zero through `ELF_K_NUM−1`.

`ELF_T_`*TYPE*
: When a service function, such as `elf32_xlatetom()` or `elf32_xlatetof()`, deals with multiple types, names of this form specify the desired *TYPE*. Thus, for example, `ELF_T_EHDR` is directly

related to `Elf32_Ehdr`. These values are members of the `Elf_Type` enumeration; they range from zero through `ELF_T_NUM−1.`

Examples

EXAMPLE 1 An interpretation of elf file.

The basic interpretation of an ELF file consists of:

- opening an ELF object file
- obtaining an ELF descriptor
- analyzing the file using the descriptor.

The following example opens the file, obtains the ELF descriptor, and prints out the names of each section in the file.

```
#include    <fcntl.h>
#include    <stdio.h>
#include    <libelf.h>
#include    <stdlib.h>
#include    <string.h>
static void failure(void);
void
main(int argc, char ** argv)
{
    Elf32_Shdr *    shdr;
    Elf32_Ehdr *    ehdr;
    Elf *        elf;
    Elf_Scn *    scn;
    Elf_Data *    data;
    int        fd;
    unsigned int    cnt;

         /* Open the input file */
    if ((fd = open(argv[1], O_RDONLY)) == -1)
        exit(1);

        /* Obtain the ELF descriptor */
    (void) elf_version(EV_CURRENT);
    if ((elf = elf_begin(fd, ELF_C_READ, NULL)) == NULL)
        failure();

        /* Obtain the .shstrtab data buffer */
    if (((ehdr = elf32_getehdr(elf)) == NULL) ||
        ((scn = elf_getscn(elf, ehdr->e_shstrndx)) == NULL) ||
        ((data = elf_getdata(scn, NULL)) == NULL))
        failure();

        /* Traverse input filename, printing each section */
    for (cnt = 1, scn = NULL; scn = elf_nextscn(elf, scn); cnt++) {
```

EXAMPLE 1 An interpretation of elf file. *(Continued)*

```
        if ((shdr = elf32_getshdr(scn)) == NULL)
            failure();
    (void) printf("[%d]    %s\n", cnt,
        (char *)data->d_buf + shdr->sh_name);
    }
}        /* end main */

static void
failure()
{
    (void) fprintf(stderr, "%s\n", elf_errmsg(elf_errno()));
    exit(1);
}
```

Attributes

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

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

See Also

ar.h(3HEAD), elf32_checksum(3ELF), elf32_fsize(3ELF), elf32_getshdr(3ELF), elf32_xlatetof(3ELF), elf_begin(3ELF), elf_cntl(3ELF), elf_errmsg(3ELF), elf_fill(3ELF), elf_getarhdr(3ELF), elf_getarsym(3ELF), elf_getbase(3ELF), elf_getdata(3ELF), elf_getident(3ELF), elf_getscn(3ELF), elf_hash(3ELF), elf_kind(3ELF), elf_memory(3ELF), elf_rawfile(3ELF), elf_strptr(3ELF), elf_update(3ELF), elf_version(3ELF), gelf(3ELF), libelf(3LIB), attributes(5), lfcompile(5)

ANSI C Programmer's Guide

SPARC only a.out(4)

Notes

Information in the ELF headers is separated into common parts and processor-specific parts. A program can make a processor's information available by including the appropriate header: <sys/elf_*NAME*.h> where *NAME* matches the processor name as used in the ELF file header.

Name	Processor
M32	AT&T WE 32100
SPARC	SPARC

Name	Processor
386	Intel 80386, 80486, Pentium

Other processors will be added to the table as necessary.

To illustrate, a program could use the following code to “see" the processor-specific information for the SPARC based system.

```
#include <libelf.h>
#include <sys/elf_SPARC.h>
```

Without the <sys/elf_SPARC.h> definition, only the common ELF information would be visible.

A program could use the following code to “see" the processor-specific information for the Intel 80386:

```
#include <libelf.h>
#include <sys/elf_386.h>
```

Without the <sys/elf_386.h> definition, only the common ELF information would be visible.

Although reading the objects is rather straightforward, writing/updating them can corrupt the shared offsets among sections. Upon creation, relationships are established among the sections that must be maintained even if the object's size is changed.

Name elf_begin, elf_end, elf_memory, elf_next, elf_rand – process ELF object files

Synopsis

```
cc [ flag... ] file ... -lelf [ library ... ]
#include <libelf.h>

Elf *elf_begin(int fildes, Elf_Cmd cmd, Elf *ref);

int elf_end(Elf *elf);

Elf *elf_memory(char *image, size_t sz);

Elf_Cmd elf_next(Elf *elf);

size_t elf_rand(Elf *elf, size_t offset);
```

Description The `elf_begin()`, `elf_end()`, `elf_memory()`, `elf_next()`, and `elf_rand()` functions work together to process Executable and Linking Format (ELF) object files, either individually or as members of archives. After obtaining an ELF descriptor from `elf_begin()` or `elf_memory()`, the program can read an existing file, update an existing file, or create a new file. The *fildes* argument is an open file descriptor that `elf_begin()` uses for reading or writing. The *elf* argument is an ELF descriptor previously returned from `elf_begin()`. The initial file offset (see `lseek(2)`) is unconstrained, and the resulting file offset is undefined.

The *cmd* argument can take the following values:

`ELF_C_NULL` When a program sets *cmd* to this value, `elf_begin()` returns a null pointer, without opening a new descriptor. *ref* is ignored for this command. See the examples below for more information.

`ELF_C_READ` When a program wants to examine the contents of an existing file, it should set *cmd* to this value. Depending on the value of *ref*, this command examines archive members or entire files. Three cases can occur.

- If *ref* is a null pointer, `elf_begin()` allocates a new ELF descriptor and prepares to process the entire file. If the file being read is an archive, `elf_begin()` also prepares the resulting descriptor to examine the initial archive member on the next call to `elf_begin()`, as if the program had used `elf_next()` or `elf_rand()` to "move" to the initial member.
- If *ref* is a non-null descriptor associated with an archive file, `elf_begin()` lets a program obtain a separate ELF descriptor associated with an individual member. The program should have used `elf_next()` or `elf_rand()` to position *ref* appropriately (except for the initial member, which `elf_begin()` prepares; see the example below). In this case, *fildes* should be the same file descriptor used for the parent archive.
- If *ref* is a non-null ELF descriptor that is not an archive, `elf_begin()` increments the number of activations for the descriptor and returns *ref*, without allocating a new descriptor and without changing the descriptor's read/write permissions. To terminate the descriptor for *ref*,

the program must call `elf_end()` once for each activation. See the examples below for more information.

`ELF_C_RDWR` This command duplicates the actions of `ELF_C_READ` and additionally allows the program to update the file image (see `elf_update`(3ELF)). Using `ELF_C_READ` gives a read-only view of the file, while `ELF_C_RDWR` lets the program read *and* write the file. `ELF_C_RDWR` is not valid for archive members. If *ref* is non-null, it must have been created with the `ELF_C_RDWR` command.

`ELF_C_WRITE` If the program wants to ignore previous file contents, presumably to create a new file, it should set *cmd* to this value. *ref* is ignored for this command.

The `elf_begin()` function operates on all files (including files with zero bytes), providing it can allocate memory for its internal structures and read any necessary information from the file. Programs reading object files can call `elf_kind`(3ELF) or `elf32_getehdr`(3ELF) to determine the file type (only object files have an ELF header). If the file is an archive with no more members to process, or an error occurs, `elf_begin()` returns a null pointer. Otherwise, the return value is a non-null ELF descriptor.

Before the first call to `elf_begin()`, a program must call `elf_version()` to coordinate versions.

The `elf_end()` function is used to terminate an ELF descriptor, *elf*, and to deallocate data associated with the descriptor. Until the program terminates a descriptor, the data remain allocated. A null pointer is allowed as an argument, to simplify error handling. If the program wants to write data associated with the ELF descriptor to the file, it must use `elf_update()` before calling `elf_end()`.

Calling `elf_end()` removes one activation and returns the remaining activation count. The library does not terminate the descriptor until the activation count reaches `0`. Consequently, a `0` return value indicates the ELF descriptor is no longer valid.

The `elf_memory()` function returns a pointer to an ELF descriptor. The ELF image has read operations enabled (`ELF_C_READ`). The *image* argument is a pointer to an image of the Elf file mapped into memory. The *sz* argument is the size of the ELF image. An ELF image that is mapped in with `elf_memory()` can be read and modified, but the ELF image size cannot be changed.

The `elf_next()` function provides sequential access to the next archive member. Having an ELF descriptor, *elf*, associated with an archive member, `elf_next()` prepares the containing archive to access the following member when the program calls `elf_begin()`. After successfully positioning an archive for the next member, `elf_next()` returns the value `ELF_C_READ`. Otherwise, the open file was not an archive, *elf* was `NULL`, or an error occurred, and the return value is `ELF_C_NULL`. In either case, the return value can be passed as an argument to `elf_begin()`, specifying the appropriate action.

The `elf_rand()` function provides random archive processing, preparing *elf* to access an arbitrary archive member. The *elf* argument must be a descriptor for the archive itself, not a member within the archive. The *offset* argument specifies the byte offset from the beginning of the archive to the archive header of the desired member. See `elf_getarsym(3ELF)` for more information about archive member offsets. When `elf_rand()` works, it returns *offset*. Otherwise, it returns `0`, because an error occurred, *elf* was `NULL`, or the file was not an archive (no archive member can have a zero offset). A program can mix random and sequential archive processing.

System Services

When processing a file, the library decides when to read or write the file, depending on the program's requests. Normally, the library assumes the file descriptor remains usable for the life of the ELF descriptor. If, however, a program must process many files simultaneously and the underlying operating system limits the number of open files, the program can use `elf_cntl()` to let it reuse file descriptors. After calling `elf_cntl()` with appropriate arguments, the program can close the file descriptor without interfering with the library.

All data associated with an ELF descriptor remain allocated until `elf_end()` terminates the descriptor's last activation. After the descriptors have been terminated, the storage is released; attempting to reference such data gives undefined behavior. Consequently, a program that deals with multiple input (or output) files must keep the ELF descriptors active until it finishes with them.

Examples

EXAMPLE 1 A sample program of calling the `elf_begin()` function.

A prototype for reading a file appears on the next page. If the file is a simple object file, the program executes the loop one time, receiving a null descriptor in the second iteration. In this case, both `elf` and `arf` will have the same value, the activation count will be 2, and the program calls `elf_end()` twice to terminate the descriptor. If the file is an archive, the loop processes each archive member in turn, ignoring those that are not object files.

```
if (elf_version(EV_CURRENT) == EV_NONE)
{
    /* library out of date */
    /* recover from error */
}
cmd = ELF_C_READ;
arf = elf_begin(fildes, cmd, (Elf *)0);
while ((elf = elf_begin(fildes, cmd, arf)) != 0)
{
    if ((ehdr = elf32_getehdr(elf)) != 0)
    {
        /* process the file  . . . */
    }
    cmd = elf_next(elf);
    elf_end(elf);
}
elf_end(arf);
```

EXAMPLE 1 A sample program of calling the `elf_begin()` function. *(Continued)*

Alternatively, the next example illustrates random archive processing. After identifying the file as an archive, the program repeatedly processes archive members of interest. For clarity, this example omits error checking and ignores simple object files. Additionally, this fragment preserves the ELF descriptors for all archive members, because it does not call `elf_end()` to terminate them.

```
elf_version(EV_CURRENT);
arf = elf_begin(fildes, ELF_C_READ, (Elf *)0);
if (elf_kind(arf) != ELF_K_AR)
{
    /* not an archive */
}
/* initial processing */
/* set offset =  . . . for desired member header */
while (elf_rand(arf, offset) == offset)
{
    if ((elf = elf_begin(fildes, ELF_C_READ, arf)) == 0)
        break;
    if ((ehdr = elf32_getehdr(elf)) != 0)
    {
        /* process archive member  . . . */
    }
    /* set offset =  . . . for desired member header */
}
```

An archive starts with a "magic string" that has `SARMAG` bytes; the initial archive member follows immediately. An application could thus provide the following function to rewind an archive (the function returns −1 for errors and `0` otherwise).

```
#include <ar.h>
#include <libelf.h>
int
rewindelf(Elf *elf)
{
     if (elf_rand(elf, (size_t)SARMAG) == SARMAG)
          return 0;
     return −1;
}
```

The following outline shows how one might create a new ELF file. This example is simplified to show the overall flow.

```
elf_version(EV_CURRENT);
fildes = open("path/name", O_RDWR|O_TRUNC|O_CREAT, 0666);
if ((elf = elf_begin(fildes, ELF_C_WRITE, (Elf *)0)) == 0)
    return;
```

EXAMPLE 1 A sample program of calling the `elf_begin()` function. *(Continued)*

```
ehdr = elf32_newehdr(elf);
phdr = elf32_newphdr(elf, count);
scn = elf_newscn(elf);
shdr = elf32_getshdr(scn);
data = elf_newdata(scn);
elf_update(elf, ELF_C_WRITE);
elf_end(elf);
```

Finally, the following outline shows how one might update an existing ELF file. Again, this example is simplified to show the overall flow.

```
elf_version(EV_CURRENT);
fildes = open("path/name", O_RDWR);
elf = elf_begin(fildes, ELF_C_RDWR, (Elf *)0);
/* add new or delete old information */
 . . .
/* ensure that the memory image of the file is complete */
elf_update(elf, ELF_C_NULL);
elf_update(elf, ELF_C_WRITE);   /* update file */
elf_end(elf);
```

Notice that both file creation examples open the file with write *and* read permissions. On systems that support `mmap(2)`, the library uses it to enhance performance, and `mmap(2)` requires a readable file descriptor. Although the library can use a write-only file descriptor, the application will not obtain the performance advantages of `mmap(2)`.

Attributes

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

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

See Also

`creat(2)`, `lseek(2)`, `mmap(2)`, `open(2)`, `ar.h(3HEAD)`, `elf(3ELF)`, `elf32_getehdr(3ELF)`, `elf_cntl(3ELF)`, `elf_getarhdr(3ELF)`, `elf_getarsym(3ELF)`, `elf_getbase(3ELF)`, `elf_getdata(3ELF)`, `elf_getscn(3ELF)`, `elf_kind(3ELF)`, `elf_rawfile(3ELF)`, `elf_update(3ELF)`, `elf_version(3ELF)`, `libelf(3LIB)`, `attributes(5)`

Name elf_cntl – control an elf file descriptor

Synopsis
```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

int elf_cntl(Elf *elf, Elf_Cmd cmd);
```

Description `elf_cntl()` instructs the library to modify its behavior with respect to an ELF descriptor, *elf*. As `elf_begin(3ELF)` describes, an ELF descriptor can have multiple activations, and multiple ELF descriptors may share a single file descriptor. Generally, `elf_cntl()` commands apply to all activations of *elf*. Moreover, if the ELF descriptor is associated with an archive file, descriptors for members within the archive will also be affected as described below. Unless stated otherwise, operations on archive members do not affect the descriptor for the containing archive.

The *cmd* argument tells what actions to take and may have the following values:

`ELF_C_FDDONE` This value tells the library not to use the file descriptor associated with *elf*. A program should use this command when it has requested all the information it cares to use and wishes to avoid the overhead of reading the rest of the file. The memory for all completed operations remains valid, but later file operations, such as the initial `elf_getdata()` for a section, will fail if the data are not in memory already.

`ELF_C_FDREAD` This command is similar to `ELF_C_FDDONE`, except it forces the library to read the rest of the file. A program should use this command when it must close the file descriptor but has not yet read everything it needs from the file. After `elf_cntl()` completes the `ELF_C_FDREAD` command, future operations, such as `elf_getdata()`, will use the memory version of the file without needing to use the file descriptor.

If `elf_cntl()` succeeds, it returns `0`. Otherwise *elf* was `NULL` or an error occurred, and the function returns `−1`.

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

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

See Also `elf(3ELF)`, `elf_begin(3ELF)`, `elf_getdata(3ELF)`, `elf_rawfile(3ELF)`, `libelf(3LIB)`, `attributes(5)`

Notes If the program wishes to use the "raw" operations (see `elf_rawdata()`, which `elf_getdata(3ELF)` describes, and `elf_rawfile(3ELF)`) after disabling the file descriptor with `ELF_C_FDDONE` or `ELF_C_FDREAD`, it must execute the raw operations explicitly beforehand. Otherwise, the raw file operations will fail. Calling `elf_rawfile()` makes the entire image available, thus supporting subsequent `elf_rawdata()` calls.

Name elf_errmsg, elf_errno – error handling

Synopsis

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

const char *elf_errmsg(int err);

int elf_errno(void);
```

Description If an ELF library function fails, a program can call `elf_errno()` to retrieve the library's internal error number. As a side effect, this function resets the internal error number to `0`, which indicates no error.

The `elf_errmsg()` function takes an error number, *err*, and returns a null-terminated error message (with no trailing new-line) that describes the problem. A zero *err* retrieves a message for the most recent error. If no error has occurred, the return value is a null pointer (not a pointer to the null string). Using *err* of `−1` also retrieves the most recent error, except it guarantees a non-null return value, even when no error has occurred. If no message is available for the given number, `elf_errmsg()` returns a pointer to an appropriate message. This function does not have the side effect of clearing the internal error number.

Examples **EXAMPLE 1** A sample program of calling the `elf_errmsg()` function.

The following fragment clears the internal error number and checks it later for errors. Unless an error occurs after the first call to `elf_errno()`, the next call will return `0`.

```
(void)elf_errno( );
/* processing  . . . */
while (more_to_do)
{
    if ((err = elf_errno( )) != 0)
    {
        /* print msg */
        msg = elf_errmsg(err);
    }
}
```

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

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

See Also `elf(3ELF)`, `libelf(3LIB)`, `attributes(5)`

Name elf_fill – set fill byte

Synopsis

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

void elf_fill(int fill);
```

Description Alignment constraints for ELF files sometimes require the presence of "holes." For example, if the data for one section are required to begin on an eight-byte boundary, but the preceding section is too "short," the library must fill the intervening bytes. These bytes are set to the *fill* character. The library uses zero bytes unless the application supplies a value. See `elf_getdata(3ELF)` for more information about these holes.

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

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

See Also `elf(3ELF)`, `elf_flagdata(3ELF)`, `elf_getdata(3ELF)`, `elf_update(3ELF)`, `libelf(3LIB)`, `attributes(5)`

Notes An application can assume control of the object file organization by setting the `ELF_F_LAYOUT` bit (see `elf_flagdata(3ELF)`). When this is done, the library does *not* fill holes.

Name elf_flagdata, elf_flagehdr, elf_flagelf, elf_flagphdr, elf_flagscn, elf_flagshdr – manipulate flags

Synopsis

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

unsigned elf_flagdata(Elf_Data *data, Elf_Cmd cmd, unsigned flags);

unsigned elf_flagehdr(Elf *elf, Elf_Cmd cmd, unsigned flags);

unsigned elf_flagelf(Elf *elf, Elf_Cmd cmd, unsigned flags);

unsigned elf_flagphdr(Elf *elf, Elf_Cmd cmd, unsigned flags);

unsigned elf_flagscn(Elf_Scn *scn, Elf_Cmd cmd, unsigned flags);

unsigned elf_flagshdr(Elf_Scn *scn, Elf_Cmd cmd, unsigned flags);
```

Description These functions manipulate the flags associated with various structures of an ELF file. Given an ELF descriptor (*elf*), a data descriptor (*data*), or a section descriptor (*scn*), the functions may set or clear the associated status bits, returning the updated bits. A null descriptor is allowed, to simplify error handling; all functions return `0` for this degenerate case.

cmd may have the following values:

`ELF_C_CLR` The functions clear the bits that are asserted in *flags*. Only the non-zero bits in *flags* are cleared; zero bits do not change the status of the descriptor.

`ELF_C_SET` The functions set the bits that are asserted in *flags*. Only the non-zero bits in *flags* are set; zero bits do not change the status of the descriptor.

Descriptions of the defined *flags* bits appear below:

`ELF_F_DIRTY` When the program intends to write an ELF file, this flag asserts the associated information needs to be written to the file. Thus, for example, a program that wished to update the ELF header of an existing file would call `elf_flagehdr()` with this bit set in *flags* and *cmd* equal to `ELF_C_SET`. A later call to `elf_update()` would write the marked header to the file.

`ELF_F_LAYOUT` Normally, the library decides how to arrange an output file. That is, it automatically decides where to place sections, how to align them in the file, etc. If this bit is set for an ELF descriptor, the program assumes responsibility for determining all file positions. This bit is meaningful only for `elf_flagelf()` and applies to the entire file associated with the descriptor.

When a flag bit is set for an item, it affects all the subitems as well. Thus, for example, if the program sets the `ELF_F_DIRTY` bit with `elf_flagelf()`, the entire logical file is “dirty."

Examples

EXAMPLE 1 A sample display of calling the `elf_flagdata()` function.

The following fragment shows how one might mark the ELF header to be written to the output file:

```
/* dirty ehdr  . . . */
ehdr = elf32_getehdr(elf);
elf_flagehdr(elf, ELF_C_SET, ELF_F_DIRTY);
```

Attributes

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

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

See Also

`elf(3ELF)`, `elf32_getehdr(3ELF)`, `elf_getdata(3ELF)`, `elf_update(3ELF)`, `attributes(5)`

Name elf_getarhdr – retrieve archive member header

Synopsis

```
cc [ flag ... ] file ... -lelf [ library... ]
#include <libelf.h>

Elf_Arhdr *elf_getarhdr(Elf *elf);
```

Description `elf_getarhdr()` returns a pointer to an archive member header, if one is available for the ELF descriptor *elf*. Otherwise, no archive member header exists, an error occurred, or *elf* was null; `elf_getarhdr()` then returns a null value. The header includes the following members.

```
char    *ar_name;
time_t    ar_date;
uid_t    ar_uid;
gid_t    ar_gid;
mode_t    ar_mode;
off_t    ar_size;
char    *ar_rawname;
```

An archive member name, available through `ar_name`, is a null-terminated string, with the `ar` format control characters removed. The `ar_rawname` member holds a null-terminated string that represents the original name bytes in the file, including the terminating slash and trailing blanks as specified in the archive format.

In addition to “regular" archive members, the archive format defines some special members. All special member names begin with a slash (`/`), distinguishing them from regular members (whose names may not contain a slash). These special members have the names (`ar_name`) defined below.

`/` This is the archive symbol table. If present, it will be the first archive member. A program may access the archive symbol table through `elf_getarsym()`. The information in the symbol table is useful for random archive processing (see `elf_rand()` on `elf_begin`(3ELF)).

`//` This member, if present, holds a string table for long archive member names. An archive member's header contains a 16-byte area for the name, which may be exceeded in some file systems. The library automatically retrieves long member names from the string table, setting `ar_name` to the appropriate value.

Under some error conditions, a member's name might not be available. Although this causes the library to set `ar_name` to a null pointer, the `ar_rawname` member will be set as usual.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also ar.h(3HEAD), elf(3ELF), elf_begin(3ELF), elf_getarsym(3ELF), libelf(3LIB), attributes(5)

Name elf_getarsym – retrieve archive symbol table

Synopsis

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

Elf_Arsym *elf_getarsym(Elf *elf, size_t *ptr);
```

Description The `elf_getarsym()` function returns a pointer to the archive symbol table, if one is available for the ELF descriptor *elf*. Otherwise, the archive doesn't have a symbol table, an error occurred, or *elf* was null; `elf_getarsym()` then returns a null value. The symbol table is an array of structures that include the following members.

```
char    *as_name;
size_t    as_off;
unsigned long    as_hash;
```

These members have the following semantics:

`as_name` A pointer to a null-terminated symbol name resides here.

`as_off` This value is a byte offset from the beginning of the archive to the member's header. The archive member residing at the given offset defines the associated symbol. Values in `as_off` may be passed as arguments to `elf_rand()`. See `elf_begin(3ELF)` to access the desired archive member.

`as_hash` This is a hash value for the name, as computed by `elf_hash()`.

If *ptr* is non-null, the library stores the number of table entries in the location to which *ptr* points. This value is set to `0` when the return value is `NULL`. The table's last entry, which is included in the count, has a null `as_name`, a zero value for `as_off`, and `~0UL` for `as_hash`.

The hash value returned is guaranteed not to be the bit pattern of all ones (`~0UL`).

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

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

See Also `ar.h(3HEAD)`, `elf(3ELF)`, `elf_begin(3ELF)`, `elf_getarhdr(3ELF)`, `elf_hash(3ELF)`, `libelf(3LIB)`, `attributes(5)`

Name elf_getbase – get the base offset for an object file

Synopsis

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

off_t elf_getbase(Elf *elf);
```

Description The `elf_getbase()` function returns the file offset of the first byte of the file or archive member associated with *elf*, if it is known or obtainable, and −1 otherwise. A null *elf* is allowed, to simplify error handling; the return value in this case is −1. The base offset of an archive member is the beginning of the member's information, *not* the beginning of the archive member header.

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

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

See Also `ar.h(3HEAD)`, `elf(3ELF)`, `elf_begin(3ELF)`, `libelf(3LIB)`, `attributes(5)`

Name elf_getdata, elf_newdata, elf_rawdata – get section data

Synopsis
```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

Elf_Data *elf_getdata(Elf_Scn *scn, Elf_Data *data);

Elf_Data *elf_newdata(Elf_Scn *scn);

Elf_Data *elf_rawdata(Elf_Scn *scn, Elf_Data *data);
```

Description These functions access and manipulate the data associated with a section descriptor, *scn*. When reading an existing file, a section will have a single data buffer associated with it. A program may build a new section in pieces, however, composing the new data from multiple data buffers. For this reason, the data for a section should be viewed as a list of buffers, each of which is available through a data descriptor.

The `elf_getdata()` function lets a program step through a section's data list. If the incoming data descriptor, *data*, is null, the function returns the first buffer associated with the section. Otherwise, *data* should be a data descriptor associated with *scn*, and the function gives the program access to the next data element for the section. If *scn* is null or an error occurs, `elf_getdata()` returns a null pointer.

The `elf_getdata()` function translates the data from file representations into memory representations (see `elf32_xlatetof`(3ELF)) and presents objects with memory data types to the program, based on the file's *class* (see `elf`(3ELF)). The working library version (see `elf_version`(3ELF)) specifies what version of the memory structures the program wishes `elf_getdata()` to present.

The `elf_newdata()` function creates a new data descriptor for a section, appending it to any data elements already associated with the section. As described below, the new data descriptor appears empty, indicating the element holds no data. For convenience, the descriptor's type (`d_type` below) is set to `ELF_T_BYTE`, and the version (`d_version` below) is set to the working version. The program is responsible for setting (or changing) the descriptor members as needed. This function implicitly sets the `ELF_F_DIRTY` bit for the section's data (see `elf_flagdata`(3ELF)). If *scn* is null or an error occurs, `elf_newdata()` returns a null pointer.

The `elf_rawdata()` function differs from `elf_getdata()` by returning only uninterpreted bytes, regardless of the section type. This function typically should be used only to retrieve a section image from a file being read, and then only when a program must avoid the automatic data translation described below. Moreover, a program may not close or disable (see `elf_cntl`(3ELF)) the file descriptor associated with *elf* before the initial raw operation, because `elf_rawdata()` might read the data from the file to ensure it doesn't interfere with `elf_getdata()`. See `elf_rawfile`(3ELF) for a related facility that applies to the entire file. When `elf_getdata()` provides the right translation, its use is recommended over `elf_rawdata()`. If *scn* is null or an error occurs, `elf_rawdata()` returns a null pointer.

The `Elf_Data` structure includes the following members:

```
void        *d_buf;
Elf_Type    d_type;
size_t      d_size;
off_t       d_off;
size_t      d_align;
unsigned    d_version;
```

These members are available for direct manipulation by the program. Descriptions appear below.

`d_buf` — A pointer to the data buffer resides here. A data element with no data has a null pointer.

`d_type` — This member's value specifies the type of the data to which `d_buf` points. A section's type determines how to interpret the section contents, as summarized below.

`d_size` — This member holds the total size, in bytes, of the memory occupied by the data. This may differ from the size as represented in the file. The size will be zero if no data exist. (See the discussion of `SHT_NOBITS` below for more information.)

`d_off` — This member gives the offset, within the section, at which the buffer resides. This offset is relative to the file's section, not the memory object's.

`d_align` — This member holds the buffer's required alignment, from the beginning of the section. That is, `d_off` will be a multiple of this member's value. For example, if this member's value is `4`, the beginning of the buffer will be four-byte aligned within the section. Moreover, the entire section will be aligned to the maximum of its constituents, thus ensuring appropriate alignment for a buffer within the section and within the file.

`d_version` — This member holds the version number of the objects in the buffer. When the library originally read the data from the object file, it used the working version to control the translation to memory objects.

Data Alignment As mentioned above, data buffers within a section have explicit alignment constraints. Consequently, adjacent buffers sometimes will not abut, causing "holes" within a section. Programs that create output files have two ways of dealing with these holes.

First, the program can use `elf_fill()` to tell the library how to set the intervening bytes. When the library must generate gaps in the file, it uses the fill byte to initialize the data there. The library's initial fill value is `0`, and `elf_fill()` lets the application change that.

Second, the application can generate its own data buffers to occupy the gaps, filling the gaps with values appropriate for the section being created. A program might even use different fill values for different sections. For example, it could set text sections' bytes to no-operation instructions, while filling data section holes with zero. Using this technique, the library finds no holes to fill, because the application eliminated them.

Section and Memory Types

The `elf_getdata()` function interprets sections' data according to the section type, as noted in the section header available through `elf32_getshdr()`. The following table shows the section types and how the library represents them with memory data types for the 32-bit file class. Other classes would have similar tables. By implication, the memory data types control translation by `elf32_xlatetof`(3ELF)

Section Type	Elf_Type	32-bit Type
SHT_DYNAMIC	ELF_T_DYN	Elf32_Dyn
SHT_DYNSYM	ELF_T_SYM	Elf32_Sym
SHT_FINI_ARRAY	ELF_T_ADDR	Elf32_Addr
SHT_GROUP	ELF_T_WORD	Elf32_Word
SHT_HASH	ELF_T_WORD	Elf32_Word
SHT_INIT_ARRAY	ELF_T_ADDR	Elf32_Addr
SHT_NOBITS	ELF_T_BYTE	unsigned char
SHT_NOTE	ELF_T_NOTE	unsigned char
SHT_NULL	*none*	*none*
SHT_PREINIT_ARRAY	ELF_T_ADDR	Elf32_Addr
SHT_PROGBITS	ELF_T_BYTE	unsigned char
SHT_REL	ELF_T_REL	Elf32_Rel
SHT_RELA	ELF_T_RELA	Elf32_Rela
SHT_STRTAB	ELF_T_BYTE	unsigned char
SHT_SYMTAB	ELF_T_SYM	Elf32_Sym
SHT_SUNW_comdat	ELF_T_BYTE	unsigned char
SHT_SUNW_move	ELF_T_MOVE	Elf32_Move (sparc)
SHT_SUNW_move	ELF_T_MOVEP	Elf32_Move (ia32)
SHT_SUNW_syminfo	ELF_T_SYMINFO	Elf32_Syminfo
SHT_SUNW_verdef	ELF_T_VDEF	Elf32_Verdef
SHT_SUNW_verneed	ELF_T_VNEED	Elf32_Verneed
SHT_SUNW_versym	ELF_T_HALF	Elf32_Versym
other	ELF_T_BYTE	unsigned char

The `elf_rawdata()` function creates a buffer with type ELF_T_BYTE.

As mentioned above, the program's working version controls what structures the library creates for the application. The library similarly interprets section types according to the versions. If a section type belongs to a version newer than the application's working version, the library does not translate the section data. Because the application cannot know the data format in this case, the library presents an untranslated buffer of type `ELF_T_BYTE`, just as it would for an unrecognized section type.

A section with a special type, `SHT_NOBITS`, occupies no space in an object file, even when the section header indicates a non-zero size. `elf_getdata()` and `elf_rawdata()` work on such a section, setting the *data* structure to have a null buffer pointer and the type indicated above. Although no data are present, the `d_size` value is set to the size from the section header. When a program is creating a new section of type `SHT_NOBITS`, it should use `elf_newdata()` to add data buffers to the section. These empty data buffers should have the `d_size` members set to the desired size and the `d_buf` members set to `NULL`.

Examples

EXAMPLE 1 A sample program of calling `elf_getdata()`.

The following fragment obtains the string table that holds section names (ignoring error checking). See `elf_strptr(3ELF)` for a variation of string table handling.

```
ehdr = elf32_getehdr(elf);
scn = elf_getscn(elf, (size_t)ehdr->e_shstrndx);
shdr = elf32_getshdr(scn);
if (shdr->sh_type != SHT_STRTAB)
{
/* not a string table */
}
data = 0;
if ((data = elf_getdata(scn, data)) == 0 || data->d_size == 0)
{
/* error or no data */
}
```

The `e_shstrndx` member in an ELF header holds the section table index of the string table. The program gets a section descriptor for that section, verifies it is a string table, and then retrieves the data. When this fragment finishes, `data->d_buf` points at the first byte of the string table, and `data->d_size` holds the string table's size in bytes.

Attributes

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

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

See Also elf(3ELF), elf32_getehdr(3ELF), elf64_getehdr(3ELF), elf32_getshdr(3ELF), elf64_getshdr(3ELF), elf32_xlatetof(3ELF), elf64_xlatetof(3ELF), elf_cntl(3ELF), elf_fill(3ELF), elf_flagdata(3ELF), elf_getscn(3ELF), elf_rawfile(3ELF), elf_strptr(3ELF), elf_version(3ELF), libelf(3LIB), attributes(5)

Name elf_getident, elf_getphdrnum, elf_getshdrnum, elf_getshdrstrndx, elf_getphnum, elf_getshnum, elf_getshstrndx – retrieve ELF header data

Synopsis

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

char *elf_getident(Elf *elf, size_t *dst);

int elf_getphdrnum(Elf *elf, size_t *dst);

int elf_getshdrnum(Elf *elf, size_t *dst);

int elf_getshdrstrndx(Elf *elf, size_t *dst);
```

Obsolete Interfaces

```
int elf_getphnum(Elf *elf, size_t *dst);

int elf_getshnum(Elf *elf, size_t *dst);

int elf_getshstrndx(Elf *elf, size_t *dst);
```

Description As elf(3ELF) explains, ELF provides a framework for various classes of files, where basic objects might have 32 or 64 bits. To accommodate these differences, without forcing the larger sizes on smaller machines, the initial bytes in an ELF file hold identification information common to all file classes. The e_ident of every ELF header has EI_NIDENT bytes with interpretations described in the following table.

e_ident Index	Value	Purpose
EI_MAG0	ELFMAG0	File identification
EI_MAG1	ELFMAG1	
EI_MAG2	ELFMAG2	
EI_MAG3	ELFMAG3	
EI_CLASS	ELFCLASSNONE	File class
	ELFCLASS32	
	ELFCLASS64	
EI_DATA	ELFDATANONE	Data encoding
	ELFDATA2LSB	
	ELFDATA2MSB	

EI_VERSION	EV_CURRENT	File version
7-15	0	Unused, set to zero

Other kinds of files might have identification data, though they would not conform to e_ident. See elf_kind(3ELF) for information on other kinds of files.

The elf_getident() function returns a pointer to the initial bytes of the file. If the library recognizes the file, a conversion from the file image to the memory image can occur. The identification bytes are guaranteed to be unmodified, though the size of the unmodified area depends on the file type. If the *dst* argument is non-null, the library stores the number of identification bytes in the location to which *dst* points. If no data are present, *elf* is NULL, or an error occurs, the return value is a null pointer, with 0 stored through *dst*, if *dst* is non-null.

The elf_getphdrnum() function obtains the number of program headers recorded in the ELF file. The number of sections in a file is typically recorded in the e_phnum field of the ELF header. A file that requires the ELF extended program header records the value PN_XNUM in the e_phnum field and records the number of sections in the sh_info field of section header 0. See USAGE. The *dst* argument points to the location where the number of sections is stored. If *elf* is NULL or an error occurs, elf_getphdrnum() returns −1.

The elf_getshdrnum() function obtains the number of sections recorded in the ELF file. The number of sections in a file is typically recorded in the e_shnum field of the ELF header. A file that requires ELF extended section records the value 0 in the e_shnum field and records the number of sections in the sh_size field of section header 0. See USAGE. The *dst* argument points to the location where the number of sections is stored. If a call to elf_newscn(3ELF) that uses the same *elf* descriptor is performed, the value obtained by elf_getshnum() is valid only after a successful call to elf_update(3ELF). If *elf* is NULL or an error occurs, elf_getshdrnum() returns −1.

The elf_getshdrstrndx() function obtains the section index of the string table associated with the section headers in the ELF file. The section header string table index is typically recorded in the e_shstrndx field of the ELF header. A file that requires ELF extended section records the value SHN_XINDEX in the e_shstrndx field and records the string table index in the sh_link field of section header 0. See USAGE. The *dst* argument points to the location where the section header string table index is stored. If *elf* is NULL or an error occurs, elf_getshdrstrndx() returns −1.

The elf_getphnum(), elf_getshnum(), and elf_getshstrndx() functions behave in a manner similar to elf_getphdrnum(), elf_getshdrnum(), and elf_getshdrstrndx(), respectively, except that they return 0 if *elf* is NULL or an error occurs. Because these return values differ from those used by some other systems, they are therefore non-portable and their use is discouraged. The elf_getphdrnum(), elf_getshdrnum(), and elf_getshdrstrndx() functions should be used instead.

Usage ELF extended sections allow an ELF file to contain more than `0xff00` (`SHN_LORESERVE`) section. ELF extended program headers allow an ELF file to contain `0xffff` (`PN_XNUM`) or more program headers. See the *Linker and Libraries Guide* for more information.

Return Values Upon successful completion, the `elf_getident()` function returns 1. Otherwise, it return 0.

Upon successful completion, the `elf_getphdrnum()`, `elf_getshdrnum()`, and `elf_getshdrstrndx()` functions return 0. Otherwise, they return -1.

Upon successful completion, the `elf_getphnum()`, `elf_getshnum()`, and `elf_getshstrndx()` functions return 1. Otherwise, they return 0.

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

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

The `elf_getident()`, `elf_getphdrnum()`, `elf_getshdrnum()`, and `elf_getshdrstrndx()` functions are Committed. The `elf_getphnum()`, `elf_getshnum()`, and `elf_getshstrndx()` functions are Committed (Obsolete).

See Also `elf(3ELF)`, `elf32_getehdr(3ELF)`, `elf_begin(3ELF)`, `elf_kind(3ELF)`, `elf_newscn(3ELF)`, `elf_rawfile(3ELF)`, `elf_update(3ELF)`, `libelf(3LIB)`, `attributes(5)`

Linker and Libraries Guide

Name elf_getscn, elf_ndxscn, elf_newscn, elf_nextscn – get section information

Synopsis
```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

Elf_Scn *elf_getscn(Elf *elf, size_t index);

size_t elf_ndxscn(Elf_Scn *scn);

Elf_Scn *elf_newscn(Elf *elf);

Elf_Scn *elf_nextscn(Elf *elf, Elf_Scn *scn);
```

Description These functions provide indexed and sequential access to the sections associated with the ELF descriptor *elf*. If the program is building a new file, it is responsible for creating the file's ELF header before creating sections; see `elf32_getehdr`(3ELF).

The `elf_getscn()` function returns a section descriptor, given an *index* into the file's section header table. Note that the first "real" section has an index of `1`. Although a program can get a section descriptor for the section whose *index* is `0` (`SHN_UNDEF`, the undefined section), the section has no data and the section header is "empty" (though present). If the specified section does not exist, an error occurs, or *elf* is `NULL`, `elf_getscn()` returns a null pointer.

The `elf_newscn()` function creates a new section and appends it to the list for *elf*. Because the `SHN_UNDEF` section is required and not "interesting" to applications, the library creates it automatically. Thus the first call to `elf_newscn()` for an ELF descriptor with no existing sections returns a descriptor for section 1. If an error occurs or *elf* is `NULL`, `elf_newscn()` returns a null pointer.

After creating a new section descriptor, the program can use `elf32_getshdr()` to retrieve the newly created, "clean" section header. The new section descriptor will have no associated data (see `elf_getdata`(3ELF)). When creating a new section in this way, the library updates the `e_shnum` member of the ELF header and sets the `ELF_F_DIRTY` bit for the section (see `elf_flagdata`(3ELF)). If the program is building a new file, it is responsible for creating the file's ELF header (see `elf32_getehdr`(3ELF)) before creating new sections.

The `elf_nextscn()` function takes an existing section descriptor, *scn*, and returns a section descriptor for the next higher section. One may use a null *scn* to obtain a section descriptor for the section whose index is `1` (skipping the section whose index is `SHN_UNDEF`). If no further sections are present or an error occurs, `elf_nextscn()` returns a null pointer.

The `elf_ndxscn()` function takes an existing section descriptor, *scn*, and returns its section table index. If *scn* is null or an error occurs, `elf_ndxscn()` returns `SHN_UNDEF`.

Examples **EXAMPLE 1** A sample of calling `elf_getscn()` function.

An example of sequential access appears below. Each pass through the loop processes the next section in the file; the loop terminates when all sections have been processed.

EXAMPLE 1 A sample of calling `elf_getscn()` function. *(Continued)*

```
scn = 0;
while ((scn = elf_nextscn(elf, scn)) != 0)
{
    /* process section */
}
```

Attributes

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

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

See Also

`elf(3ELF)`, `elf32_getehdr(3ELF)`, `elf32_getshdr(3ELF)`, `elf_begin(3ELF)`, `elf_flagdata(3ELF)`, `elf_getdata(3ELF)`, `libelf(3LIB)`, `attributes(5)`

Name elf_hash – compute hash value

Synopsis

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

unsigned long elf_hash(const char *name);
```

Description The `elf_hash()` function computes a hash value, given a null terminated string, *name*. The returned hash value, *h*, can be used as a bucket index, typically after computing *h* mod x to ensure appropriate bounds.

Hash tables may be built on one machine and used on another because `elf_hash()` uses unsigned arithmetic to avoid possible differences in various machines' signed arithmetic. Although *name* is shown as `char*` above, `elf_hash()` treats it as `unsigned char*` to avoid sign extension differences. Using `char*` eliminates type conflicts with expressions such as `elf_hash(`*name*`)`.

ELF files' symbol hash tables are computed using this function (see `elf_getdata`(3ELF) and `elf32_xlatetof`(3ELF)). The hash value returned is guaranteed not to be the bit pattern of all ones (`~0UL`).

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

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

See Also `elf`(3ELF), `elf32_xlatetof`(3ELF), `elf_getdata`(3ELF), `libelf`(3LIB), `attributes`(5)

Name elf_kind – determine file type

Synopsis

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

Elf_Kind elf_kind(Elf *elf);
```

Description This function returns a value identifying the kind of file associated with an ELF descriptor (*elf*). Defined values are below:

`ELF_K_AR` The file is an archive (see `ar.h`(3HEAD)). An ELF descriptor may also be associated with an archive *member*, not the archive itself, and then `elf_kind()` identifies the member's type.

`ELF_K_ELF` The file is an ELF file. The program may use `elf_getident()` to determine the class. Other functions, such as `elf32_getehdr()`, are available to retrieve other file information.

`ELF_K_NONE` This indicates a kind of file unknown to the library.

Other values are reserved, to be assigned as needed to new kinds of files. *elf* should be a value previously returned by `elf_begin()`. A null pointer is allowed, to simplify error handling, and causes `elf_kind()` to return `ELF_K_NONE`.

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

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

See Also `ar.h`(3HEAD), `elf`(3ELF), `elf32_getehdr`(3ELF), `elf_begin`(3ELF), `elf_getident`(3ELF), `libelf`(3LIB), `attributes`(5)

Name elf_rawfile – retrieve uninterpreted file contents

Synopsis
```
cc [ flag... ] file ... -lelf [ library ... ]
#include <libelf.h>

char *elf_rawfile(Elf *elf, size_t *ptr);
```

Description The `elf_rawfile()` function returns a pointer to an uninterpreted byte image of the file. This function should be used only to retrieve a file being read. For example, a program might use `elf_rawfile()` to retrieve the bytes for an archive member.

A program may not close or disable (see `elf_cntl(3ELF)`) the file descriptor associated with *elf* before the initial call to `elf_rawfile()` , because `elf_rawfile()` might have to read the data from the file if it does not already have the original bytes in memory. Generally, this function is more efficient for unknown file types than for object files. The library implicitly translates object files in memory, while it leaves unknown files unmodified. Thus, asking for the uninterpreted image of an object file may create a duplicate copy in memory.

`elf_rawdata()` is a related function, providing access to sections within a file. See `elf_getdata(3ELF)`.

If *ptr* is non-null, the library also stores the file's size, in bytes, in the location to which *ptr* points. If no data are present, *elf* is null, or an error occurs, the return value is a null pointer, with `0` stored through *ptr*, if *ptr* is non-null.

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

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

See Also `elf(3ELF)`, `elf32_getehdr(3ELF)`, `elf_begin(3ELF)`, `elf_cntl(3ELF)`, `elf_getdata(3ELF)`, `elf_getident(3ELF)`, `elf_kind(3ELF)`, `libelf(3LIB)`, `attributes(5)`

Notes A program that uses `elf_rawfile()` and that also interprets the same file as an object file potentially has two copies of the bytes in memory. If such a program requests the raw image first, before it asks for translated information (through such functions as `elf32_getehdr()`, `elf_getdata()`, and so on), the library “freezes" its original memory copy for the raw image. It then uses this frozen copy as the source for creating translated objects, without reading the file again. Consequently, the application should view the raw file image returned by `elf_rawfile()` as a read-only buffer, unless it wants to alter its own view of data subsequently translated. In any case, the application may alter the translated objects without changing bytes visible in the raw image.

Multiple calls to `elf_rawfile()` with the same ELF descriptor return the same value; the library does not create duplicate copies of the file.

Name elf_strptr – make a string pointer

Synopsis

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

char *elf_strptr(Elf *elf, size_t section, size_t offset);
```

Description The `elf_strptr()` function converts a string section *offset* to a string pointer. *elf* identifies the file in which the string section resides, and *section* identifies the section table index for the strings. `elf_strptr()` normally returns a pointer to a string, but it returns a null pointer when *elf* is null, *section* is invalid or is not a section of type `SHT_STRTAB`, the section data cannot be obtained, *offset* is invalid, or an error occurs.

Examples **EXAMPLE 1** A sample program of calling `elf_strptr()` function.

A prototype for retrieving section names appears below. The file header specifies the section name string table in the `e_shstrndx` member. The following code loops through the sections, printing their names.

```
/* handle the error */
if ((ehdr = elf32_getehdr(elf)) == 0) {
    return;
}
ndx = ehdr->e_shstrndx;
scn = 0;
while ((scn = elf_nextscn(elf, scn)) != 0) {
    char    *name = 0;
    if ((shdr = elf32_getshdr(scn)) != 0)
                name = elf_strptr(elf, ndx, (size_t)shdr->sh_name);
    printf("'%s'\n", name? name: "(null)");
}
```

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

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

See Also `elf(3ELF)`, `elf32_getshdr(3ELF)`, `elf32_xlatetof(3ELF)`, `elf_getdata(3ELF)`, `libelf(3LIB)`, `attributes(5)`

Notes A program may call `elf_getdata()` to retrieve an entire string table section. For some applications, that would be both more efficient and more convenient than using `elf_strptr()`.

Name elf_update – update an ELF descriptor

Synopsis

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include &lt;libelf.h&gt;

off_t elf_update(Elf *elf, Elf_Cmd cmd);
```

Description The `elf_update()` function causes the library to examine the information associated with an ELF descriptor, *elf*, and to recalculate the structural data needed to generate the file's image.

The *cmd* argument can have the following values:

`ELF_C_NULL` This value tells `elf_update()` to recalculate various values, updating only the ELF descriptor's memory structures. Any modified structures are flagged with the `ELF_F_DIRTY` bit. A program thus can update the structural information and then reexamine them without changing the file associated with the ELF descriptor. Because this does not change the file, the ELF descriptor may allow reading, writing, or both reading and writing (see `elf_begin`(3ELF)).

`ELF_C_WRITE` If *cmd* has this value, `elf_update()` duplicates its `ELF_C_NULL` actions and also writes any "dirty" information associated with the ELF descriptor to the file. That is, when a program has used `elf_getdata`(3ELF) or the `elf_flagdata`(3ELF) facilities to supply new (or update existing) information for an ELF descriptor, those data will be examined, coordinated, translated if necessary (see `elf32_xlatetof`(3ELF)), and written to the file. When portions of the file are written, any `ELF_F_DIRTY` bits are reset, indicating those items no longer need to be written to the file (see `elf_flagdata`(3ELF)). The sections' data are written in the order of their section header entries, and the section header table is written to the end of the file. When the ELF descriptor was created with `elf_begin()`, it must have allowed writing the file. That is, the `elf_begin()` command must have been either `ELF_C_RDWR` or `ELF_C_WRITE`.

If `elf_update()` succeeds, it returns the total size of the file image (not the memory image), in bytes. Otherwise an error occurred, and the function returns `−1`.

When updating the internal structures, `elf_update()` sets some members itself. Members listed below are the application's responsibility and retain the values given by the program.

The following table shows ELF Header members:

Member	Notes
e_ident[EI_DATA]	Library controls other `e_ident` values

e_type	
e_machine	
e_version	
e_entry	
e_phoff	Only when ELF_F_LAYOUT asserted
e_shoff	Only when ELF_F_LAYOUT asserted
e_flags	
e_shstrndx	

The following table shows the Program Header members:

Member	Notes
p_type	The application controls all
p_offset	program header entries
p_vaddr	
p_paddr	
p_filesz	
p_memsz	
p_flags	
p_align	

The following table shows the Section Header members:

Member	Notes
sh_name	
sh_type	
sh_flags	
sh_addr	

sh_offset	Only when ELF_F_LAYOUT asserted
sh_size	Only when ELF_F_LAYOUT asserted
sh_link	
sh_info	
sh_addralign	Only when ELF_F_LAYOUT asserted
sh_entsize	

The following table shows the Data Descriptor members:

Member	Notes
d_buf	
d_type	
d_size	
d_off	Only when ELF_F_LAYOUT asserted
d_align	
d_version	

Note that the program is responsible for two particularly important members (among others) in the ELF header. The `e_version` member controls the version of data structures written to the file. If the version is `EV_NONE`, the library uses its own internal version. The `e_ident[EI_DATA]` entry controls the data encoding used in the file. As a special case, the value may be `ELFDATANONE` to request the native data encoding for the host machine. An error occurs in this case if the native encoding doesn't match a file encoding known by the library.

Further note that the program is responsible for the `sh_entsize` section header member. Although the library sets it for sections with known types, it cannot reliably know the correct value for all sections. Consequently, the library relies on the program to provide the values for unknown section types. If the entry size is unknown or not applicable, the value should be set to `0`.

When deciding how to build the output file, `elf_update()` obeys the alignments of individual data buffers to create output sections. A section's most strictly aligned data buffer controls the section's alignment. The library also inserts padding between buffers, as necessary, to ensure the proper alignment of each buffer.

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

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

See Also elf(3ELF), elf32_fsize(3ELF), elf32_getehdr(3ELF), elf32_getshdr(3ELF), elf32_xlatetof(3ELF), elf_begin(3ELF), elf_flagdata(3ELF), elf_getdata(3ELF), libelf(3LIB), attributes(5)

Notes As mentioned above, the ELF_C_WRITE command translates data as necessary, before writing them to the file. This translation is *not* always transparent to the application program. If a program has obtained pointers to data associated with a file (for example, see elf32_getehdr(3ELF) and elf_getdata(3ELF)), the program should reestablish the pointers after calling elf_update().

Name elf_version – coordinate ELF library and application versions

Synopsis

```
cc [ flag ... ] file ... -lelf [ library ... ]
#include <libelf.h>

unsigned elf_version(unsigned ver);
```

Description As `elf(3ELF)` explains, the program, the library, and an object file have independent notions of the latest ELF version. `elf_version()` lets a program query the ELF library's *internal version*. It further lets the program specify what memory types it uses by giving its own *working version*, `ver`, to the library. Every program that uses the ELF library must coordinate versions as described below.

The header `<libelf.h>` supplies the version to the program with the macro `EV_CURRENT`. If the library's internal version (the highest version known to the library) is lower than that known by the program itself, the library may lack semantic knowledge assumed by the program. Accordingly, `elf_version()` will not accept a working version unknown to the library.

Passing *ver* equal to `EV_NONE` causes `elf_version()` to return the library's internal version, without altering the working version. If `ver` is a version known to the library, `elf_version()` returns the previous (or initial) working version number. Otherwise, the working version remains unchanged and `elf_version()` returns `EV_NONE`.

Examples **EXAMPLE 1** A sample display of using the `elf_version()` function.

The following excerpt from an application program protects itself from using an older library:

```
if (elf_version(EV_CURRENT) == EV_NONE) {
    /* library out of date */
    /* recover from error */
}
```

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

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

See Also `elf(3ELF)`, `elf32_xlatetof(3ELF)`, `elf_begin(3ELF)`, `libelf(3LIB)`, `attributes(5)`

Notes The working version should be the same for all operations on a particular ELF descriptor. Changing the version between operations on a descriptor will probably not give the expected results.

Name FCOE_CreatePort – create an FCoE port

Synopsis

```
cc [ flag... ] file... -lfcoe [ library... ]
#include <libfcoe.h>

int FCOE_CreatePort(const char *macLinkName, int portType,
     struct fcoe_port_wwn pwwn, struct fcoe_port_wwn nwwn,
     int promiscuous);
```

Parameters

macLinkName The name of the MAC link on which to create the FCoE port.

portType This parameter should always be `FCOE_PORTTYPE_TARGET`.

pwwn The Port WorldWideName to be used for the FCoE port. Fill the structure with zeros to let the `fcoe` driver generate a valid Port WWN from the MAC address of the underlying NIC hardware.

nwwn The Node WorldWideName to be used for the FCoE port. Fill the structure with zeros to let the `fcoe` driver generate a valid Node WWN from the MAC address of the underlying NIC hardware.

promiscuous A non-zero value to enable promiscuous mode on the underlying NIC hardware. A value of 0 indicates use of the multiple unicast address feature of the underlying NIC hardware.

Description The `FCOE_CreatePort()` function creates an FCoE port over the specified MAC link.

Return Values The following values are returned:

`FCOE_STATUS_ERROR_BUSY`
The `fcoe` driver is busy and cannot complete the operation.

`FCOE_STATUS_ERROR_ALREADY`
An existing FCoE port was found over the specified MAC link.

`FCOE_STATUS_ERROR_OPEN_DEV`
Failed to open `fcoe` device.

`FCOE_STATUS_ERROR_WWN_SAME`
The specified Port WWN is same as the specified Node WWN.

`FCOE_STATUS_ERROR_MAC_LEN`
MAC link name exceeds the maximum length.

`FCOE_STATUS_ERROR_PWWN_CONFLICTED`
The specified Port WWN is already in use.

`FCOE_STATUS_ERROR_NWWN_CONFLICTED`
The specified Node WWN is already in use.

`FCOE_STATUS_ERROR_NEED_JUMBO_FRAME`
The MTU size of the specified MAC link needs to be increased to 2500 or above.

`FCOE_STATUS_ERROR_OPEN_MAC`
Failed to open the specified MAC link.

`FCOE_STATUS_ERROR_CREATE_PORT`
Failed to create FCoE port on the specified MAC link.

`FCOE_STATUS_OK`
The API call was successful.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libfcoe(3LIB)`, `attributes(5)`

Name FCOE_DeletePort – delete an FCoE port

Synopsis
```
cc [ flag... ] file... -lfcoe [ library... ]
#include <libfcoe.h>

int FCOE_DeletePort(const char *macLinkName);
```

Parameters *macLinkName* The name of the MAC link from which to delete the FCoE port.

Description The `FCOE_DeletePort()` function deletes an FCoE port from the specified MAC link.

Return Values The following values are returned:

`FCOE_STATUS_ERROR_BUSY`
The `fcoe` driver is busy and cannot complete the operation.

`FCOE_STATUS_ERROR_MAC_LEN`
The MAC link name exceeds the maximum length.

`FCOE_STATUS_MAC_NOT_FOUND`
The FCoE port was not found on the specified MAC link.

`FCOE_STATUS_OK`
The API call was successful.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libfcoe(3LIB)`, `attributes(5)`

Name FCOE_GetPortList – get a list of FCoE ports

Synopsis
```
cc [ flag... ] file... -lfcoe [ library... ]
#include <libfcoe.h>

int FCOE_GetPortList(unsigned int *port_num,
     struct fcoe_port_attr **portlist);
```

Parameters

port_num A pointer to an integer that, on successful return, contains the number of FCoE ports in the system.

portlist A pointer to a pointer to an `fcoe_port_attr` structure that, on successful return, contains a list of the FCoE ports in the system.

Description The `FCOE_GetPortList()` function retrieves a list of FCoE ports. When the caller is finished using the list, it must free the memory used by the list by calling `free`(3C).

Return Values The following values are returned:

`FCOE_STATUS_ERROR_BUSY`
The `fcoe` driver is busy and cannot complete the operation.

`FCOE_STATUS_ERROR_INVAL_ARG`
The value specified for *port_num* or *portlist* was not valid.

`FCOE_STATUS_ERROR_OPEN_DEV`
Failed to open `fcoe` device.

`FCOE_STATUS_OK`
The API call was successful.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `free`(3C), `libfcoe`(3LIB), `attributes`(5)

Name fmev_shdl_init, fmev_shdl_fini, fmev_shdl_subscribe, fmev_shdl_unsubscribe, fmev_shdl_getauthority, fmev_errno, fmev_strerror, fmev_attr_list, fmev_class, fmev_timespec, fmev_time_sec, fmev_time_nsec, fmev_localtime, fmev_hold, fmev_hrtime, fmev_rele, fmev_dup, fmev_ev2shdl, fmev_shdl_alloc, fmev_shdl_zalloc, fmev_shdl_free, fmev_shdl_strdup, fmev_shdl_strfree, fmev_shdl_nvl2str, fmev_shdlctl_serialize, fmev_shdlctl_thrattr, fmev_shdlctl_sigmask, fmev_shdlctl_thrsetup, fmev_shdlctl_thrcreate – subscription to fault management events from an external process

Synopsis

```
cc [ flag... ] file... –L/usr/lib/fm –lfmevent –lnvpair [ library... ]
#include <fm/libfmevent.h>
#include <libnvpair.h>

typedef enum fmev_err_t;
extern fmev_err_t fmev_errno;
const char *fmev_strerror(fmev_err_t err);

typedef struct fmev_shdl *fmev_shdl_t;

typedef void fmev_cbfunc_t(fmev_t, const char *, nvlist_t *, void *);

fmev_shdl_t fmev_shdl_init(uint32_t api_version,
     void *(*alloc)(size_t), void *(*zalloc)(size_t),
     void (*free)(void *, size_t));

fmev_err_t fmev_shdl_fini(fmev_shdl_t hdl);

fmev_err_t fmev_shdl_subscribe(fmev_shdl_t hdl, const char *classpat,
     fmev_cbfunc_t callback, void *cookie);

fmev_err_t fmev_shdl_unsubscribe(fmev_shdl_t hdl,
     const char *classpat);

fmev_err_t fmev_shdl_getauthority(fmev_shdl_t hdl, nvlist_t **authp);

fmev_err_t fmev_shdlctl_serialize(fmev_shdl_t hdl);

fmev_err_t fmev_shdlctl_thrattr(fmev_shdl_t hdl, pthread_attr_t *attr);

fmev_err_t fmev_shdlctl_sigmask(fmev_shdl_t hdl, sigset_t *set);

fmev_err_t fmev_shdlctl_thrsetup(fmev_shdl_t hdl,
     door_xcreate_thrsetup_func_t *setupfunc, void *cookie);

fmev_err_t fmev_shdlctl_thrcreate(fmev_shdl_t hdl,
     door_xcreate_server_func_t *createfunc, void *cookie);

typedef struct fmev *fmev_t;

nvlist_t *fmev_attr_list(fmev_t ev);

const char *fmev_class(fmev_t ev);

fmev_err_t fmev_timespec(fmev_t ev, struct timespec *res);

uint64_t fmev_time_sec(fmev_t ev);
```

```
uint64_t fmev_time_nsec(fmev_t ev);

struct tm *fmev_localtime(fmev_t ev, struct tm *res);

hrtime_t fmev_hrtime(fmev_t ev);

void fmev_hold(fmev_t ev);

void fmev_rele(fmev_t ev);

fmev_t fmev_dup(fmev_t ev);

fmev_shdl_t fmev_ev2shdl(fmev_t ev);

void *fmev_shdl_alloc(fmev_shdl_t hdl, size_t sz);

void *fmev_shdl_zalloc(fmev_shdl_t hdl, size_t sz);

void fmev_shdl_free(fmev_shdl_t hdl, void *buf, size_t sz);

char *fmev_shdl_strdup(fmev_shdl_t hdl, char *str);

void fmev_shdl_strfree(fmev_shdl_t hdl, char *str);

char *fmev_shdl_nvl2str(fmev_shdl_t hdl, nvlist_t *fmri);
```

Description

The Solaris fault management daemon (fmd) is the central point in Solaris for fault management. It receives observations from various sources and delivers them to subscribing diagnosis engines; if those diagnosis engines diagnose a problem, the fault manager publishes additional protocol events to track the problem lifecycle from initial diagnosis through repair and final problem resolution. The event protocol is specified in the Sun Fault Management Event Protocol Specification. The interfaces described here allow an external process to subscribe to protocol events. See the Fault Management Daemon Programmer's Reference Guide for additional information on fmd.

The fmd module API (not a Committed interface) allows plugin modules to load within the fmd process, subscribe to events of interest, and participate in various diagnosis and response activities. Of those modules, some are notification agents and will subscribe to events describing diagnoses and their subsequent lifecycle and render these to console/syslog (for the `syslog-msgs` agent) and via SNMP trap and browsable MIB (for the `snmp-trapgen` module and the corresponding dlmod for the SNMP daemon). It has not been possible to subscribe to protocol events outside of the context of an fmd plugin. The `libfmevent` interface provides this external subscription mechanism. External subscribers may receive protocol events as fmd modules do, but they cannot participate in other aspects of the fmd module API such as diagnosis. External subscribers are therefore suitable as notification agents and for transporting fault management events.

Fault Management Protocol Events

This protocol is defined in the Sun Fault Management Event Protocol Specification. Note that while the API described on this manual page are Committed, the protocol events themselves (in class names and all event payload) are not Committed along with this API. The protocol specification document describes the commitment level of individual event classes and their payload content. In broad terms, the list.* events are Committed in most of their content and semantics while events of other classes are generally Uncommitted with a few exceptions.

All protocol events include an identifying class string, with the hierarchies defined in the protocol document and individual events registered in the Events Registry. The `libfmevent` mechanism will permit subscription to events with Category 1 class of “list” and “swevent”, that is, to classes matching patterns “list.*” and “swevent.*”.

All protocol events consist of a number of (name, datatype, value) tuples (“nvpairs”). Depending on the event class various nvpairs are required and have well-defined meanings. In Solaris fmd protocol events are represented as name-value lists using the `libnvpair(3LIB)` interfaces.

API Overview

The API is simple to use in the common case (see Examples), but provides substantial control to cater for more-complex scenarios.

We obtain an opaque subscription handle using `fmev_shdl_init()`, quoting the ABI version and optionally nominating `alloc()`, `zalloc()` and `free()` functions (the defaults use the `umem` family). More than one handle may be opened if desired. Each handle opened establishes a communication channel with fmd, the implementation of which is opaque to the `libfmevent` mechanism.

On a handle we may establish one or more subscriptions using `fmev_shdl_subscribe()`. Events of interest are specified using a simple wildcarded pattern which is matched against the event class of incoming events. For each match that is made a callback is performed to a function we associate with the subscription, passing a nominated cookie to that function. Subscriptions may be dropped using `fmev_shdl_unsubscribe()` quoting exactly the same class or class pattern as was used to establish the subscription.

Each call to `fmev_shdl_subscribe()` creates a single thread dedicated to serving callback requests arising from this subscription.

An event callback handler has as arguments an opaque event handle, the event class, the event nvlist, and the cookie it was registered with in `fmev_shdl_subscribe()`. The timestamp for when the event was generated (not when it was received) is available as a `struct timespec` with `fmev_timespec()`, or more directly with `fmev_time_sec()` and `fmev_time_nsec()`; an event handle and `struct tm` can also be passed to `fmev_localtime()` to fill the `struct tm`. A high-resolution timestamp for an event may be retrieved using `fmev_hrtime()`; this value has the semantics described in `gethrtime(3C)`.

The event handle, class string pointer, and `nvlist_t` pointer passed as arguments to a callback are valid for the duration of the callback. If the application wants to continue to process the event beyond the duration of the callback then it can hold the event with `fmev_hold()`, and later release it with `fmev_rele()`. When the reference count drops to zero the event is freed.

Error Handling

In `<libfmevent.h>` an enumeration `fmev_err_t` of error types is defined. To render an error message string from an `fmev_err_t` use `fmev_strerror()`. An `fmev_errno` is defined which returns the error number for the last failed `libfmevent` API call made by the current thread. You may not assign to `fmev_errno`.

If a function returns type `fmev_err_t`, then success is indicated by `FMEV_SUCCESS` (or `FMEV_OK` as an alias); on failure a `FMEVERR_*` value is returned (see `<fm/libfmevent.h>`).

If a function returns a pointer type then failure is indicated by a `NULL` return, and `fmev_errno` will record the error type.

Subscription Handles

A subscription handle is required in order to establish and manage subscriptions. This handle represents the abstract communication mechanism between the application and the fault management daemon running in the current zone.

A subscription handle is represented by the opaque `fmev_shdl_t` datatype. A handle is initialized with `fmev_shdl_init()` and quoted to subsequent API members.

To simplify usage of the API, subscription attributes for all subscriptions established on a handle are a property of the handle itself ; they cannot be varied per-subscription. In such use cases multiple handles will need to be used.

libfmevent ABI version

The first argument to `fmev_shdl_init()` indicates the `libfmevent` ABI version with which the handle is being opened. Specify either `LIBFMEVENT_VERSION_LATEST` to indicate the most recent version available at compile time or `LIBFMEVENT_VERSION_1` (_2, etc. as the interface evolves) for an explicit choice.

Interfaces present in an earlier version of the interface will continue to be present with the same or compatible semantics in all subsequent versions. When additional interfaces and functionality are introduced the ABI version will be incremented. When an ABI version is chosen in `fmev_shdl_init()`, only interfaces introduced in or before that version will be available to the application via that handle. Attempts to use later API members will fail with `FMEVERR_VERSION_MISMATCH`.

This manual page describes `LIBFMEVENT_VERSION_1`.

Privileges

The `libfmevent` API is not least-privilege aware; you need to have all privileges to call `fmev_shdl_init()`. Once a handle has been initialized with `fmev_shdl_init()` a process can drop privileges down to the basic set and continue to use `fmev_shdl_subscribe()` and other `libfmevent` interfaces on that handle.

Underlying Event Transport

The implementation of the event transport by which events are published from the fault manager and multiplexed out to `libfmevent` consumers is strictly private. It is subject to change at any time, and you should not encode any dependency on the underlying mechanism into your application. Use only the API described on this manual page and in `<libfmevent.h>`.

The underlying transport mechanism is guaranteed to have the property that a subscriber may attach to it even before the fault manager is running. If the fault manager starts first then any events published before the first consumer subscribes will wait in the transport until a consumer appears.

The underlying transport will also have some maximum depth to the queue of events pending delivery. This may be hit if there are no consumers, or if consumers are not processing events quickly enough. In practice the rate of events is small. When this maximum depth is reached additional events will be dropped.

The underlying transport has no concept of priority delivery; all events are treated equally.

Subscription Handle Initialization

Obtain a new subscription handle with `fmev_shdl_init()`. The first argument is the `libfmevent` ABI version to be used (see above). The remaining three arguments should be all `NULL` to leave the library to use its default allocator functions (the `libumem` family), or all non-`NULL` to appoint wrappers to custom allocation functions if required.

`FMEVERR_VERSION_MISMATCH`
The library does not support the version requested.

`FMEVERR_ALLOC`
An error occurred in trying to allocate data structures.

`FMEVERR_API`
The `alloc()`, `zalloc()`, or `free()` arguments must either be all `NULL` or all non-`NULL`.

`FMEVERR_NOPRIV`
Insufficient privilege to perform operation. In version 1 root privilege is required.

`FMEVERR_INTERNAL`
Internal library error.

Fault Manager Authority Information

Once a subscription handle has been initialized, authority information for the fault manager to which the client is connected may be retrieved with `fmev_shdl_getauthority()`. The caller is responsible for freeing the returned nvlist using `nvlist_free`(3NVPAIR).

Subscription Handle Finalization

Close a subscription handle with `fmev_shdl_fini()`. This call must not be performed from within the context of an event callback handler, else it will fail with `FMEVERR_API`.

The `fmev_shdl_fini()` call will remove all active subscriptions on the handle and free resources used in managing the handle.

`FMEVERR_API`
May not be called from event delivery context for a subscription on the same handle.

Subscribing To Events

To establish a new subscription on a handle, use `fmev_shdl_subscribe()`. Besides the handle argument you provide the class or class pattern to subscribe to (the latter permitting simple wildcarding using '*'), a callback function pointer for a function to be called for all matching events, and a cookie to pass to that callback function.

The class pattern must match events per the fault management protocol specification, such as “list.suspect” or “list.*”. Patterns that do not map onto existing events will not be rejected - they just won't result in any callbacks.

A callback function has type fmev_cbfunc_t. The first argument is an opaque event handle for use in event access functions described below. The second argument is the event class string, and the third argument is the event nvlist; these could be retrieved using fmev_class() and fmev_attr_list() on the event handle, but they are supplied as arguments for convenience. The final argument is the cookie requested when the subscription was established in fmev_shdl_subscribe().

Each call to fmev_shdl_subscribe() opens a new door into the process that the kernel uses for event delivery. Each subscription therefore uses one file descriptor in the process.

See below for more detail on event callback context.

FMEVERR_API
: Class pattern is NULL or callback function is NULL.

FMEVERR_BADCLASS
: Class pattern is the empty string, or exceeds the maximum length of FMEV_MAX_CLASS.

FMEVERR_ALLOC
: An attempt to fmev_shdl_zalloc() additional memory failed.

FMEVERR_DUPLICATE
: Duplicate subscription request. Only one subscription for a given class pattern may exist on a handle.

FMEVERR_MAX_SUBSCRIBERS
: A system-imposed limit on the maximum number of subscribers to the underlying transport mechanism has been reached.

FMEVERR_INTERNAL
: An unknown error occurred in trying to establish the subscription.

Unsubscribing

An unsubscribe request using fmev_shdl_unsubscribe() must exactly match a previous subscription request or it will fail with FMEVERR_NOMATCH. The request stops further callbacks for this subscription, waits for any existing active callbacks to complete, and drops the subscription.

Do not call fmev_shdl_unsubscribe from event callback context, else it will fail with FMEVERR_API.

FMEVERR_API
: A NULL pattern was specified, or the call was attempted from callback context.

FMEVERR_NOMATCH
: The pattern provided does not match any open subscription. The pattern must be an exact match.

FMEVERR_BADCLASS
: The class pattern is the empty string or exceeds FMEV_MAX_CLASS.

Event Callback Context

Event callback context is defined as the duration of a callback event, from the moment we enter the registered callback function to the moment it returns. There are a few restrictions on actions that may be performed from callback context:

- You can perform long-running actions, but this thread will not be available to service other event deliveries until you return.
- You must not cause the current thread to exit.
- You must not call either `fmev_shdl_unsubscribe()` or `fmev_shdl_fini()` for the subscription handle on which this callback has been made.
- You can invoke `fork()`, `popen()`, etc.

Event Handles

A callback receives an `fmev_t` as a handle on the associated event. The callback may use the access functions described below to retrieve various event attributes.

By default, an event handle `fmev_t` is valid for the duration of the callback context. You cannot access the event outside of callback context.

If you need to continue to work with an event beyond the initial callback context in which it is received, you may place a "hold" on the event with `fmev_hold()`. When finished with the event, release it with `fmev_rele()`. These calls increment and decrement a reference count on the event; when it drops to zero the event is freed. On initial entry to a callback the reference count is 1, and this is always decremented when the callback returns.

An alternative to `fmev_hold()` is `fmev_dup()`, which duplicates the event and returns a new event handle with a reference count of 1. When `fmev_rele()` is applied to the new handle and reduces the reference count to 0, the event is freed. The advantage of `fmev_dup()` is that it allocates new memory to hold the event rather than continuing to hold a buffer provided by the underlying delivery mechanism. If your operation is going to be long-running, you may want to use `fmev_dup()` to avoid starving the underlying mechanism of event buffers.

Given an `fmev_t`, a callback function can use `fmev_ev2shdl()` to retrieve the subscription handle on which the subscription was made that resulted in this event delivery.

The `fmev_hold()` and `fmev_rele()` functions always succeed.

The `fmev_dup()` function may fail and return `NULL` with `fmev_errno` of:

`FMEVERR_API` — A `NULL` event handle was passed.

`FMEVERR_ALLOC` — The `fmev_shdl_alloc()` call failed.

Event Class

A delivery callback already receives the event class as an argument, so `fmev_class()` will only be of use outside of callback context (that is, for an event that was held or duped in callback context and is now being processed in an asynchronous handler). This is a convenience function that returns the same result as accessing the event attributes with `fmev_attr_list()` and using `nvlist_lookup_string(3NVPAIR)` to lookup a string member of name "class".

The string returned by `fmev_class()` is valid for as long as the event handle itself.

The fmev_class() function may fail and return NULL with fmev_errno of:

FMEVERR_API — A NULL event handle was passed.

FMEVERR_MALFORMED_EVENT — The event appears corrupted.

Event Attribute List All events are defined as a series of (name, type) pairs. An instance of an event is therefore a series of tuples (name, type, value). Allowed types are defined in the protocol specification. In Solaris, and in libfmevent, an event is represented as an nvlist_t using the libnvpair(3LIB) library.

The nvlist of event attributes can be accessed using fmev_attr_list(). The resulting nvlist_t pointer is valid for the same duration as the underlying event handle. Do not use nvlist_free() to free the nvlist. You may then lookup members, iterate over members, and so on using the libnvpair interfaces.

The fmev_attr_list() function may fail and return NULL with fmev_errno of:

FMEVERR_API — A NULL event handle was passed.

FMEVERR_MALFORMED_EVENT — The event appears corrupted.

Event Timestamp These functions refer to the time at which the event was originally produced, not the time at which it was forwarded to libfmevent or delivered to the callback.

Use fmev_timespec() to fill a struct timespec with the event time in seconds since the Epoch (tv_sec, signed integer) and nanoseconds past that second (tv_nsec, a signed long). This call can fail and return FMEVERR_OVERFLOW if the seconds value will not fit in a signed 32-bit integer (as used in struct timespec tv_sec).

You can use fmev_time_sec() and fmev_time_nsec() to retrieve the same second and nanosecond values as uint64_t quantities.

The fmev_localtime function takes an event handle and a struct tm pointer and fills that structure according to the timestamp. The result is suitable for use with strftime(3C). This call will return NULL and fmev_errno of FMEVERR_OVERFLOW under the same conditions as above.

FMEVERR_OVERFLOW — The fmev_timespec() function cannot fit the seconds value into the signed long integer tv_sec member of a struct timespec.

String Functions A string can be duplicated using fmev_shdl_strdup(); this will allocate memory for the copy using the allocator nominated in fmev_shdl_init(). The caller is responsible for freeing the buffer using fmev_shdl_strfree(); the caller can modify the duplicated string but must not change the string length.

An FMRI retrieved from a received event as an nvlist_t may be rendered as a string using fmev_shdl_nvl2str(). The nvlist must be a legal FMRI (recognized class, version and payload), or NULL is returned with fmev_errno() of FMEVERR_INVALIDARG. The formatted string is rendered into a buffer allocated using the memory allocation functions nominated in fmev_shdl_init(), and the caller is responsible for freeing that buffer using fmev_shdl_strfree().

Memory Allocation The fmev_shdl_alloc(), fmev_shdl_zalloc(), and fmev_shdl_free() functions allocate and free memory using the choices made for the given handle when it was initialized, typically the libumem(3LIB) family if all were specified NULL.

Subscription Handle Control The fmev_shdlctl_*() interfaces offer control over various properties of the subscription handle, allowing fine-tuning for particular applications. In the common case the default handle properties will suffice.

These properties apply to the handle and uniformly to all subscriptions made on that handle. The properties may only be changed when there are no subscriptions in place on the handle, otherwise FMEVERR_BUSY is returned.

Event delivery is performed through invocations of a private door. A new door is opened for each fmev_shdl_subscribe() call. These invocations occur in the context of a single private thread associated with the door for a subscription. Many of the fmev_shdlctl_*() interfaces are concerned with controlling various aspects of this delivery thread.

If you have applied fmev_shdlctl_thrcreate(), “custom thread creation semantics” apply on the handle; otherwise “default thread creation semantics” are in force. Some fmev_shdlctl_*() interfaces apply only to default thread creation semantics.

The fmev_shdlctl_serialize() control requests that all deliveries on a handle, regardless of which subscription request they are for, be serialized - no concurrent deliveries on this handle. Without this control applied deliveries arising from each subscription established with fmev_shdl_subscribe() are individually single-threaded, but if multiple subscriptions have been established then deliveries arising from separate subscriptions may be concurrent. This control applies to both custom and default thread creation semantics.

The fmev_shdlctl_thrattr() control applies only to default thread creation semantics. Threads that are created to service subscriptions will be created with pthread_create(3C) using the pthread_attr_t provided by this interface. The attribute structure is not copied and so must persist for as long as it is in force on the handle.

The default thread attributes are also the minimum requirement: threads must be created PTHREAD_CREATE_DETACHED and PTHREAD_SCOPE_SYSTEM. A NULL pointer for the pthread_attr_t will reinstate these default attributes.

The fmev_shdlctl_sigmask() control applies only to default thread creation semantics. Threads that are created to service subscriptions will be created with the requested signal set masked - a pthread_sigmask(3C) request to SIG_SETMASK to this mask prior to pthread_create(). The default is to mask all signals except SIGABRT.

See door_xcreate(3C) for a detailed description of thread setup and creation functions for door server threads.

The fmev_shdlctl_thrsetup() function runs in the context of the newly-created thread before it binds to the door created to service the subscription. It is therefore a suitable place to perform any thread-specific operations the application may require. This control applies to both custom and default thread creation semantics.

Using fmev_shdlctl_thrcreate() forfeits the default thread creation semantics described above. The function appointed is responsible for all of the tasks required of a door_xcreate_server_func_t in door_xcreate().

The fmev_shdlctl_*() functions may fail and return NULL with fmev_errno of:

FMEVERR_BUSY Subscriptions are in place on this handle.

Examples

EXAMPLE 1 Subscription example

The following example subscribes to list.suspect events and prints out a simple message for each one that is received. It foregoes most error checking for the sake of clarity.

```
#include <fm/libfmevent.h>
#include <libnvpair.h>

/*
 * Callback to receive list.suspect events
 */
void
mycb(fmev_t ev, const char *class, nvlist_t *attr, void *cookie)
{
        struct tm tm;
        char buf[64];
        char *evcode;

        if (strcmp(class, "list.suspect") != 0)
                return; /* only happens if this code has a bug! */

        (void) strftime(buf, sizeof (buf), NULL,
            fmev_localtime(ev, &tm));

        (void) nvlist_lookup_string(attr, "code", &evcode);

        (void) fprintf(stderr, "Event class %s published at %s, "
            "event code %s\n", class, buf, evcode);
}

int
main(int argc, char *argv[])
```

EXAMPLE 1 Subscription example *(Continued)*

```
{
        fmev_shdl_t hdl;
        sigset_t set;

        hdl = fmev_shdl_init(LIBFMEVENT_VERSION_LATEST,
            NULL, NULL, NULL);

        (void) fmev_shdl_subscribe(hdl, "list.suspect", mycb, NULL);

        /* Wait here until signalled with SIGTERM to finish */
        (void) sigemptyset(&set);
        (void) sigaddset(&set, SIGTERM);
        (void) sigwait(&set);

        /* fmev_shdl_fini would do this for us if we skipped it */
        (void) fmev_shdl_unsubscribe(hdl, "list.suspect");

        (void) fmev_shdl_fini(hdl);

        return (0);
}
```

Attributes

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

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

See Also

`door_xcreate(3C)`, `gethrtime(3C)`, `libnvpair(3LIB)`, `libumem(3LIB)`, `nvlist_lookup_string(3NVPAIR)`, `pthread_create(3C)`, `pthread_sigmask(3C)`, `strftime(3C)`, `attributes(5)`, `privileges(5)`

Name fstyp_get_attr – get file system attributes

Synopsis

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

int fstyp_get_attr(fstyp_handle_t handle, nvlist_t **attrp);
```

Parameters

handle Opaque handle returned by `fstyp_ident`(3FSTYP).

attrp Address to which the name-pair list is returned.

Description The `fstyp_get_attr()` function returns a name-value pair list of various attributes for an identified file system. This function can be called only after a successful call to `fstyp_ident()`.

Each file system has its own set of attributes. The following attributes are generic and are returned when appropriate for a particular file system type:

`gen_clean (DATA_TYPE_BOOLEAN_VALUE)` Attribute for which `true` and `false` values are allowed. A `false` value is returned if the file system is damaged or if the file system is not cleanly unmounted. In the latter case, `fsck`(1M) is required before the file system can be mounted.

`gen_guid (DATA_TYPE_STRING)` Globally unique string identifier used to establish the identity of the file system.

`gen_version (DATA_TYPE_STRING)` String that specifes the file system version.

`gen_volume_label (DATA_TYPE_STRING)` Human-readable volume label string used to describe and/or identify the file system.

Attribute names associated with specific file systems should not start with `gen_`.

Return Values The `fstyp_get_attr()` function returns `0` on success and an error value on failure. See `fstyp_strerror`(3FSTYP).

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

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

See Also fstyp_ident(3FSTYP), fstyp_mod_init(3FSTYP), fstyp_strerror(3FSTYP), libfstyp(3LIB), attributes(5)

Name fstyp_ident – identify file system attributes

Synopsis

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

int fstyp_ident(fstyp_handle_t handle, const char *fstyp,
     const char **ident);
```

Parameters

handle Opaque handle returned by `fstyp_init`(3FSTYP).

fstype Opaque argument that specifies the file system type to be identified.

ident File system type returned if identification succeeds.

Description The `fstyp_ident()` function attempts to identify a file system associated with the *handle*. If the function succeeds, the file system name is returned in the *ident* pointer.

If *fstype* is `NULL`, the `fstyp_ident()` function tries all available identification modules. If *fstype* is other than `NULL`, `fstyp_ident()` tries only the module for the file system type which is specified.

Return Values The `fstyp_ident()` function returns `0` on success and an error value on failure. See `fstyp_strerror`(3FSTYP).

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

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

See Also `fstyp_init`(3FSTYP), `fstyp_mod_init`(3FSTYP), `fstyp_strerror`(3FSTYP), `libfstyp`(3LIB), `attributes`(5)

Name fstyp_init, fstyp_fini – initialize and finalize libfstyp handle

Synopsis

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

int fstyp_init(int fd, off64_t **offset, char *module_dir,
     fstyp_handle_t *handle);

void fstyp_fini(fstyp_handle_t handle);
```

Parameters

fd	Open file descriptor of a block or a raw device that contains the file system to be identified.
offset	Offset from the beginning of the device where the file system is located.
module_dir	Optional location of the `libfstyp` modules.
handle	Opaque handle to be used with `libfstyp` functions.

Description The `fstyp_init()` function returns a *handle* associated with the specified parameters. This *handle* should be used with all other `libfstyp` functions.

If *module_dir* is `NULL`, `fstyp_init()` looks for modules in the default location: `/usr/lib/fs` subdirectories. The `fstyp_init()` function locates `libfstyp` modules, but might defer loading the modules until the subsequent `fstyp_ident()` call.

If *module_dir* is other than `NULL`, the `fstyp_init()` function locates a module in the directory that is specified. If no module is found, `fstyp_init` fails with `FSTYP_ERR_MOD_NOT_FOUND`.

Modules that do not support non-zero offset can fail `fstyp_init()` with `FSTYP_ERR_OFFSET`.

The `fstyp_fini()` function releases all resources associated with a handle and invalidates the handle.

Return Values The `fstyp_init()` function returns `0` on success and an error value on failure. See `fstyp_strerror`(3FSTYP).

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

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

See Also `fstyp_ident`(3FSTYP), `fstyp_mod_init`(3FSTYP), `fstyp_strerror`(3FSTYP), `libfstyp`(3LIB), `attributes`(5)

Name fstyp_mod_init, fstyp_mod_fini, fstyp_mod_ident, fstyp_mod_get_attr, fstyp_mod_dump – libfstyp module interface

Synopsis

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

int fstyp_mod_init(int fd, off64_t **offset, fstyp_mod_handle_t *handle);

void fstyp_mod_fini(fstyp_mod_handle_t handle);

int fstyp_mod_ident(fstyp_mod_handle_t handle);

int fstyp_mod_get_attr(fstyp_mod_handle_t handle, nvlist_t **attr);

int fstyp_mod_dump(fstyp_mod_handle_t handle, FILE *fout, FILE *ferr);
```

Parameters

fd Open file descriptor of a block or a raw device that contains the file system to be identified.

offset Offset from the beginning of the device where the file system is located.

handle Opaque handle that the module returns in `fstyp_mod_init()` and is used with other module functions.

fout Output stream.

ferr Error stream.

Description A `libfstyp` module implements heuristics required to identify a file system type. The modules are shared objects loaded by `libfstyp`. The `libfstyp` modules are located in `/usr/lib/fs` subdirectories. A subdirectory name defines the name of the file system.

Each module exports the `fstyp_mod_init()`, `fstyp_mod_fini()`, `fstyp_mod_ident()`, and `fstyp_mod_get_attr()` functions. All of these functions map directly to the respective `libfstyp` interfaces.

The `fstyp_mod_dump()` function is optional. It can be used to output unformatted information about the file system. This function is used by the `fstyp(1M)` command when the `-v` option is specified. The `fstyp_mod_dump()` function is not recommended and should be used only in legacy modules.

Files

`/usr/lib/fs/`	Default module directory.
`/usr/lib/fs/`*fstype*`/fstyp.so.1`	Default path to a `libfstyp` module for an *fstype* file system.

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

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

See Also fstyp(1M), fstyp_strerror(3FSTYP), libfstyp(3LIB), attributes(5)

Name fstyp_strerror – get error message string

Synopsis

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

const char *fstyp_strerror(fstyp_handle_t handle, int error);
```

Parameters

handle Opaque handle returned by `fstyp_init(3FSTYP)`. This argument is optional and can be `0`.

error Error value returned by a `libfstyp` function.

Description The `fstyp_strerror()` function maps the error value to an error message string and returns a pointer to that string. The returned string should not be overwritten.

The following error values are defined:

`FSTYP_ERR_NO_MATCH`	No file system match.
`FSTYP_ERR_MULT_MATCH`	Multiple file system matches.
`FSTYP_ERR_HANDLE`	Invalid handle.
`FSTYP_ERR_OFFSET`	Supplied offset is invalid or unsupported by the module.
`FSTYP_ERR_NO_PARTITION`	Specified partition not found.
`FSTYP_ERR_NOP`	No such operation.
`FSTYP_ERR_DEV_OPEN`	Device cannot be opened.
`FSTYP_ERR_IO`	I/O error.
`FSTYP_ERR_NOMEM`	Out of memory.
`FSTYP_ERR_MOD_NOT_FOUND`	Requested file system module not found.
`FSTYP_ERR_MOD_DIR_OPEN`	Directory cannot be opened.
`FSTYP_ERR_MOD_OPEN`	Module cannot be opened.
`FSTYP_ERR_MOD_INVALID`	Invalid module version.
`FSTYP_ERR_NAME_TOO_LONG`	File system name length exceeds system limit.

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

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

See Also fstyp_init(3FSTYP), libfstyp(3LIB), attributes(5)

Name gelf, gelf_checksum, gelf_fsize, gelf_getcap, gelf_getclass, gelf_getdyn, gelf_getehdr, gelf_getmove, gelf_getphdr, gelf_getrel, gelf_getrela, gelf_getshdr, gelf_getsym, gelf_getsyminfo, gelf_getsymshndx, gelf_newehdr, gelf_newphdr, gelf_update_cap, gelf_update_dyn, gelf_update_ehdr, gelf_update_getmove, gelf_update_move, gelf_update_phdr, gelf_update_rel, gelf_update_rela, gelf_update_shdr, gelf_update_sym, gelf_update_symshndx, gelf_update_syminfo, gelf_xlatetof, gelf_xlatetom – generic class-independent ELF interface

Synopsis

```
cc [ flag... ] file... −lelf [ library... ]
#include <gelf.h>

long gelf_checksum(Elf *elf);

size_t gelf_fsize(Elf *elf, Elf_Type type, size_t cnt, unsigned ver);

int gelf_getcap(Elf_Data *src, int ndx, GElf_Cap *dst);

int gelf_getclass(Elf *elf);

GElf_Dyn *gelf_getdyn(Elf_Data *src, int ndx, GElf_Dyn *dst);

GElf_Ehdr *gelf_getehdr(Elf *elf, GElf_Ehdr *dst);

GElf_Move *gelf_getmove(Elf_Data *src, int ndx, GElf_Move *dst);

GElf_Phdr *gelf_getphdr(Elf *elf, int ndx, GElf_Phdr *dst);

GElf_Rel *gelf_getrel(Elf_Data *src, int ndx, GElf_Rel *dst);

GElf_Rela *gelf_getrela(Elf_Data *src, int ndx, GElf_Rela *dst);

GElf_Shdr *gelf_getshdr(Elf_Scn *scn, GElf_Shdr *dst);

GElf_Sym *gelf_getsym(Elf_Data *src, int ndx, GElf_Sym *dst);

GElf_Syminfo *gelf_getsyminfo(Elf_Data *src, int ndx, GElf_Syminfo *dst);

GElf_Sym *gelf_getsymshndx(Elf_Data *symsrc, Elf_Data *shndxsrc,
     int ndx, GElf_Sym *symdst, Elf32_Word *shndxdst);

unsigned long gelf_newehdr(Elf *elf, int class);

unsigned long gelf_newphdr(Elf *elf, size_t phnum);

int gelf_update_cap(Elf_Data *dst, int ndx, GElf_Cap *src);

int gelf_update_dyn(Elf_Data *dst, int ndx, GElf_Dyn *src);

int gelf_update_ehdr(Elf *elf, GElf_Ehdr *src);

int gelf_update_move(Elf_Data *dst, int ndx, GElf_Move *src);

int gelf_update_phdr(Elf *elf, int ndx, GElf_Phdr *src);

int gelf_update_rel(Elf_Data *dst, int ndx, GElf_Rel *src);

int gelf_update_rela(Elf_Data *dst, int ndx, GElf_Rela *src);

int gelf_update_shdr(Elf_Scn *dst, GElf_Shdr *src);
```

```
int gelf_update_sym(Elf_Data *dst, int ndx, GElf_Sym *src);

int gelf_update_syminfo(Elf_Data *dst, int ndx, GElf_Syminfo *src);

int gelf_update_symshndx(Elf_Data *symdst, Elf_Data *shndxdst, int ndx,
     GElf_Sym *symsrc, Elf32_Word shndxsrc);

Elf_Data *gelf_xlatetof(Elf *elf, Elf_Data *dst, const Elf_Data *src,
     unsigned encode);

Elf_Data *gelf_xlatetom(Elf *elf, Elf_Data *dst, const Elf_Data *src,
     unsigned encode);
```

Description

GElf is a generic, ELF class-independent API for manipulating ELF object files. GElf provides a single, common interface for handling 32–bit and 64–bit ELF format object files. GElf is a translation layer between the application and the class-dependent parts of the ELF library. Thus, the application can use GElf, which in turn, will call the corresponding elf32_ or elf64_ functions on behalf of the application. The data structures returned are all large enough to hold 32–bit and 64–bit data.

GElf provides a simple, class-independent layer of indirection over the class-dependent ELF32 and ELF64 API's. GElf is stateless, and may be used along side the ELF32 and ELF64 API's.

GElf always returns a copy of the underlying ELF32 or ELF64 structure, and therefore the programming practice of using the address of an ELF header as the base offset for the ELF's mapping into memory should be avoided. Also, data accessed by type-casting the Elf_Data buffer to a class-dependent type and treating it like an array, for example, a symbol table, will not work under GElf, and the gelf_get functions must be used instead. See the EXAMPLE section.

Programs that create or modify ELF files using libelf(3LIB) need to perform an extra step when using GElf. Modifications to GElf values must be explicitly flushed to the underlying ELF32 or ELF64 structures by way of the gelf_update_ interfaces. Use of elf_update or elf_flagelf and the like remains the same.

The sizes of versioning structures remain the same between ELF32 and ELF64. The GElf API only defines types for versioning, rather than a functional API. The processing of versioning information will stay the same in the GElf environment as it was in the class-dependent ELF environment.

List of Functions

gelf_checksum()	An analog to elf32_checksum(3ELF) and elf64_checksum(3ELF).
gelf_fsize()	An analog to elf32_fsize(3ELF) and elf64_fsize(3ELF).
gelf_getcap()	Retrieves the Elf32_Cap or Elf64_Cap information from the capability table at the given index. dst points to the location where the GElf_Cap capability entry is stored.
gelf_getclass()	Returns one of the constants ELFCLASS32, ELFCLASS64 or ELFCLASSNONE.

`gelf_getdyn()`	Retrieves the `Elf32_Dyn` or `Elf64_Dyn` information from the dynamic table at the given index. `dst` points to the location where the `GElf_Dyn` dynamic entry is stored.
`gelf_getehdr()`	An analog to `elf32_getehdr`(3ELF) and `elf64_getehdr`(3ELF). `dst` points to the location where the `GElf_Ehdr` header is stored.
`gelf_getmove()`	Retrieves the `Elf32_Move` or `Elf64_Move` information from the move table at the given index. `dst` points to the location where the `GElf_Move` move entry is stored.
`gelf_getphdr()`	An analog to`elf32_getphdr`(3ELF) and `elf64_getphdr`(3ELF). `dst` points to the location where the `GElf_Phdr` program header is stored.
`gelf_getrel()`	Retrieves the `Elf32_Rel` or `Elf64_Rel` information from the relocation table at the given index. `dst` points to the location where the `GElf_Rel` relocation entry is stored.
`gelf_getrela()`	Retrieves the `Elf32_Rela` or `Elf64_Rela` information from the relocation table at the given index. `dst` points to the location where the `GElf_Rela` relocation entry is stored.
`gelf_getshdr()`	An analog to `elf32_getshdr`(3ELF) and `elf64_getshdr`(3ELF). `dst` points to the location where the `GElf_Shdr` section header is stored.
`gelf_getsym()`	Retrieves the `Elf32_Sym` or `Elf64_Sym` information from the symbol table at the given index. `dst` points to the location where the `GElf_Sym` symbol entry is stored.
`gelf_getsyminfo()`	Retrieves the `Elf32_Syminfo` or `Elf64_Syminfo` information from the relocation table at the given index. `dst` points to the location where the `GElf_Syminfo` symbol information entry is stored.
`gelf_getsymshndx()`	Provides an extension to `gelf_getsym()` that retrieves the `Elf32_Sym` or `Elf64_Sym` information, and the section index from the symbol table at the given index *ndx*. The symbols section index is typically recorded in the `st_shndx` field of the symbols structure. However, a file that requires ELF Extended Sections may record an `st_shndx` of `SHN_XINDEX` indicating that the section index must be obtained from an associated `SHT_SYMTAB_SHNDX` section entry. If *xshndx* and *shndxdata* are non-null, the value recorded at index *ndx* of

	the `SHT_SYMTAB_SHNDX` table pointed to by *shndxdata* is returned in *xshndx*. See USAGE.
`gelf_newehdr()`	An analog to `elf32_newehdr`(3ELF) and `elf64_newehdr`(3ELF).
`gelf_newphdr()`	An analog to `elf32_newphdr`(3ELF) and `elf64_newphdr`(3ELF).
`gelf_update_cap()`	Copies the `GElf_Cap` information back into the underlying `Elf32_Cap` or `Elf64_Cap` structure at the given index.
`gelf_update_dyn()`	Copies the `GElf_Dyn` information back into the underlying `Elf32_Dyn` or `Elf64_Dyn` structure at the given index.
`gelf_update_ehdr()`	Copies the contents of the `GElf_Ehdr` `ELF` header to the underlying `Elf32_Ehdr` or `Elf64_Ehdr` structure.
`gelf_update_move()`	Copies the `GElf_Move` information back into the underlying `Elf32_Move` or `Elf64_Move` structure at the given index.
`gelf_update_phdr()`	Copies of the contents of `GElf_Phdr` program header to underlying the `Elf32_Phdr` or `Elf64_Phdr` structure.
`gelf_update_rel()`	Copies the `GElf_Rel` information back into the underlying `Elf32_Rel` or `Elf64_Rel` structure at the given index.
`gelf_update_rela()`	Copies the `GElf_Rela` information back into the underlying `Elf32_Rela` or `Elf64_Rela` structure at the given index.
`gelf_update_shdr()`	Copies of the contents of `GElf_Shdr` section header to underlying the `Elf32_Shdr` or `Elf64_Shdr` structure.
`gelf_update_sym()`	Copies the `GElf_Sym` information back into the underlying `Elf32_Sym` or `Elf64_Sym` structure at the given index.
`gelf_update_syminfo()`	Copies the `GElf_Syminfo` information back into the underlying `Elf32_Syminfo` or `Elf64_Syminfo` structure at the given index.
`gelf_update_symshndx()`	Provides an extension to `gelf_update_sym()` that copies the `GElf_Sym` information back into the `Elf32_Sym` or `Elf64_Sym` structure at the given index *ndx*, and copies the extended *xshndx* section index into the `Elf32_Word` at the given index *ndx* in the buffer described by `shndxdata`. See USAGE.
`gelf_xlatetof()`	An analog to `elf32_xlatetof`(3ELF) and `elf64_xlatetof`(3ELF)
`gelf_xlatetom()`	An analog to `elf32_xlatetom`(3ELF) and `elf64_xlatetom`(3ELF)

Return Values Upon failure, all `GElf` functions return `0` and set `elf_errno`. See `elf_errno`(3ELF)

Examples **EXAMPLE 1** Printing the ELF Symbol Table

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <libelf.h>
#include <gelf.h>

void
main(int argc, char **argv)
{
    Elf         *elf;
    Elf_Scn     *scn = NULL;
    GElf_Shdr   shdr;
    Elf_Data    *data;
    int         fd, ii, count;

    elf_version(EV_CURRENT);

    fd = open(argv[1], O_RDONLY);
    elf = elf_begin(fd, ELF_C_READ, NULL);

    while ((scn = elf_nextscn(elf, scn)) != NULL) {
        gelf_getshdr(scn, &shdr);
        if (shdr.sh_type == SHT_SYMTAB) {
            /* found a symbol table, go print it. */
            break;
        }
    }

    data = elf_getdata(scn, NULL);
    count = shdr.sh_size / shdr.sh_entsize;

    /* print the symbol names */
    for (ii = 0; ii < count; ++ii) {
        GElf_Sym sym;
        gelf_getsym(data, ii, &sym);
        printf("%s\n", elf_strptr(elf, shdr.sh_link, sym.st_name));
    }
    elf_end(elf);
    close(fd);
}
```

Usage ELF Extended Sections are employed to allow an ELF file to contain more than 0xff00 (SHN_LORESERVE) section. See the *Linker and Libraries Guide* for more information.

Files

/lib/libelf.so.1	shared object
/lib/64/libelf.so.1	64–bit shared object

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

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

See Also elf(3ELF), elf32_checksum(3ELF), elf32_fsize(3ELF), elf32_getehdr(3ELF), elf32_newehdr(3ELF), elf32_getphdr(3ELF), elf32_newphdr(3ELF), elf32_getshdr(3ELF), elf32_xlatetof(3ELF), elf32_xlatetom(3ELF), elf_errno(3ELF), libelf(3LIB), attributes(5)

Linker and Libraries Guide

Name generic_events – generic performance counter events

Description The Solaris cpc(3CPC) subsystem implements a number of predefined, generic performance counter events. Each generic event maps onto a single platform specific event and one or more optional attributes. Each hardware platform only need support a subset of the total set of generic events.

The defined generic events are:

`PAPI_br_cn`	Conditional branch instructions
`PAPI_br_ins`	Branch instructions
`PAPI_br_msp`	Conditional branch instructions mispredicted
`PAPI_br_ntk`	Conditional branch instructions not taken
`PAPI_br_prc`	Conditional branch instructions correctly predicted
`PAPI_br_tkn`	Conditional branch instructions taken
`PAPI_br_ucn`	Unconditional branch instructions
`PAPI_bru_idl`	Cycles branch units are idle
`PAPI_btac_m`	Branch target address cache misses
`PAPI_ca_cln`	Requests for exclusive access to clean cache line
`PAPI_ca_inv`	Requests for cache invalidation
`PAPI_ca_itv`	Requests for cache line intervention
`PAPI_ca_shr`	Request for exclusive access to shared cache line
`PAPI_ca_snp`	Request for cache snoop
`PAPI_csr_fal`	Failed conditional store instructions
`PAPI_csr_suc`	Successful conditional store instructions
`PAPI_csr_tot`	Total conditional store instructions
`PAPI_fad_ins`	Floating point add instructions
`PAPI_fdv_ins`	Floating point divide instructions
`PAPI_fma_ins`	Floating point multiply and add instructions
`PAPI_fml_ins`	Floating point multiply instructions
`PAPI_fnv_ins`	Floating point inverse instructions
`PAPI_fp_ins`	Floating point instructions
`PAPI_fp_ops`	Floating point operations

PAPI_fp_stal	Cycles the floating point unit stalled
PAPI_fpu_idl	Cycles the floating point units are idle
PAPI_fsq_ins	Floating point sqrt instructions
PAPI_ful_ccy	Cycles with maximum instructions completed
PAPI_ful_icy	Cycles with maximum instruction issue
PAPI_fxu_idl	Cycles when units are idle
PAPI_hw_int	Hardware interrupts
PAPI_int_ins	Integer instructions
PAPI_tot_cyc	Total cycles
PAPI_tot_iis	Instructions issued
PAPI_tot_ins	Instructions completed
PAPI_vec_ins	VectorSIMD instructions
PAPI_l1_dca	Level 1 data cache accesses
PAPI_l1_dch	Level 1 data cache hits
PAPI_l1_dcm	Level 1 data cache misses
PAPI_l1_dcr	Level 1 data cache reads
PAPI_l1_dcw	Level 1 data cache writes
PAPI_l1_ica	Level 1 instruction cache accesses
PAPI_l1_ich	Level 1 instruction cache hits
PAPI_l1_icm	Level 1 instruction cache misses
PAPI_l1_icr	Level 1 instruction cache reads
PAPI_l1_icw	Level 1 instruction cache writes
PAPI_l1_ldm	Level 1 cache load misses
PAPI_l1_stm	Level 1 cache store misses
PAPI_l1_tca	Level 1 cache accesses
PAPI_l1_tch	Level 1 cache hits
PAPI_l1_tcm	Level 1 cache misses
PAPI_l1_tcr	Level 1 cache reads
PAPI_l1_tcw	Level 1 cache writes

PAPI_l2_dca	Level 2 data cache accesses
PAPI_l2_dch	Level 2 data cache hits
PAPI_l2_dcm	Level 2 data cache misses
PAPI_l2_dcr	Level 2 data cache reads
PAPI_l2_dcw	Level 2 data cache writes
PAPI_l2_ica	Level 2 instruction cache accesses
PAPI_l2_ich	Level 2 instruction cache hits
PAPI_l2_icm	Level 2 instruction cache misses
PAPI_l2_icr	Level 2 instruction cache reads
PAPI_l2_icw	Level 2 instruction cache writes
PAPI_l2_ldm	Level 2 cache load misses
PAPI_l2_stm	Level 2 cache store misses
PAPI_l2_tca	Level 2 cache accesses
PAPI_l2_tch	Level 2 cache hits
PAPI_l2_tcm	Level 2 cache misses
PAPI_l2_tcr	Level 2 cache reads
PAPI_l2_tcw	Level 2 cache writes
PAPI_l3_dca	Level 3 data cache accesses
PAPI_l3_dch	Level 3 data cache hits
PAPI_l3_dcm	Level 3 data cache misses
PAPI_l3_dcr	Level 3 data cache reads
PAPI_l3_dcw	Level 3 data cache writes
PAPI_l3_ica	Level 3 instruction cache accesses
PAPI_l3_ich	Level 3 instruction cache hits
PAPI_l3_icm	Level 3 instruction cache misses
PAPI_l3_icr	Level 3 instruction cache reads
PAPI_l3_icw	Level 3 instruction cache writes
PAPI_l3_ldm	Level 3 cache load misses
PAPI_l3_stm	Level 3 cache store misses

PAPI_l3_tca	Level 3 cache accesses
PAPI_l3_tch	Level 3 cache hits
PAPI_l3_tcm	Level 3 cache misses
PAPI_l3_tcr	Level 3 cache reads
PAPI_l3_tcw	Level 3 cache writes
PAPI_ld_ins	Load Instructions
PAPI_lst_ins	Loadstore Instructions
PAPI_lsu_idl	Cycles load store units are idle
PAPI_mem_rcy	Cycles stalled waiting for memory reads
PAPI_mem_scy	Cycles stalled waiting for memory accesses
PAPI_mem_wcy	Cycles stalled waiting for memory writes
PAPI_prf_dm	Data prefetch cache misses
PAPI_res_stl	Cycles stalled on any resource
PAPI_sr_ins	Store Instructions
PAPI_stl_ccy	Cycles with no instructions completed
PAPI_syc_ins	Synchronization instructions completed
PAPI_tlb_dm	Data TLB misses
PAPI_tlb_im	Instruction TLB misses
PAPI_tlb_sd	TLB shootdowns
PAPI_tlb_tl	Total TLB misses

The tables below define mappings of generic events to platform events and any associated attribute for all supported platforms.

Intel Core2 Processors

Generic Event	Event Code/Unit Mask	Platform Event
PAPI_tot_cyc	0x3c/0x00	cpu_clk_unhalted.thread_p/core
PAPI_tot_ins	0xc0/0x00	inst_retired.any_p
PAPI_br_ins	0xc4/0x0c	br_inst_retired.taken
PAPI_br_msp	0xc5/0x00	br_inst_retired.mispred
PAPI_br_ntk	0xc4/0x03	br_inst_retired.pred_not_taken |pred_taken

Generic Event	Event Code/Unit Mask	Platform Event
PAPI_br_prc	0xc4/0x05	br_inst_retired.pred_not_taken | pred_taken
PAPI_hw_int	0xc8/0x00	hw_int_rvc
PAPI_tot_iis	0xaa/0x01	macro_insts.decoded
PAPI_l1_dca	0x43/0x01	l1d_all_ref
PAPI_l1_icm	0x81/0x00	l1i_misses
PAPI_l1_icr	0x80/0x00	l1i_reads
PAPI_l1_tcw	0x41/0x0f	l1d_cache_st.mesi
PAPI_l2_stm	0x2a/0x41	l2_st.self.i_state
PAPI_l2_tca	0x2e/0x4f	l2_rqsts.self.demand.mesi
PAPI_l2_tch	0x2e/0x4e	l2_rqsts.mes
PAPI_l2_tcm	0x2e/0x41	l2_rqsts.self.demand.i_state
PAPI_l2_tcw	0x2a/0x4f	l2_st.self.mesi
PAPI_ld_ins	0xc0/0x01	inst_retired.loads
PAPI_lst_ins	0xc0/0x03	inst_retired.loads | stores
PAPI_sr_ins	0xc0/0x02	inst_retired.stores
PAPI_tlb_dm	0x08/0x01	dtlb_misses.any
PAPI_tlb_im	0x82/0x12	itlb.small_miss | large_miss
PAPI_tlb_tl	0x0c/0x03	page_walks
PAPI_l1_dcm	0xcb/0x01	mem_load_retired.l1d_miss

Fixed-function counters do not require Event Code and Unit Mask. The generic event to fixed-function counter event mappings available are:

Generic Event	Platform Fixed-function Event
PAPI_tot_ins	instr_retired.any
PAPI_tot_cyc	cpu_clk_unhalted.core/thread

Intel Processor 5500 Family (Core i7)

Generic Event	Event Code/Unit Mask	Platform Event
PAPI_tot_cyc	0x3c/0x00	cpu_clk_unhalted.thread_p

Generic Event	Event Code/Unit Mask	Platform Event
PAPI_tot_ins	0xc0/0x00	inst_retired.any_p
PAPI_br_cn	0xc4/0x01	br_inst_retired.conditional
PAPI_hw_int	0x1d/0x01	hw_int.rcx
PAPI_tot_iis	0x17/0x01	inst_queue_writes
PAPI_l1_dca	0x43/0x01	l1d_all_ref.any
PAPI_l1_dcm	0x24/0x03	l2_rqsts.loads | rfos
PAPI_l1_dcr	0x40/0x0f	l1d_cache_ld.mesi
PAPI_l1_dcw	0x41/0x0f	l1d_cache_st.mesi
PAPI_l1_ica	0x80/0x03	l1i.reads
PAPI_l1_ich	0x80/0x01	l1i.hits
PAPI_l1_icm	0x80/0x02	l1i.misses
PAPI_l1_icr	0x80/0x03	l1i.reads
PAPI_l1_ldm	0x24/0x33	l2_rqsts.loads | ifetches
PAPI_l1_tcm	0x24/0xff	l2_rqsts.references
PAPI_l2_ldm	0x24/0x02	l2_rqsts.ld_miss
PAPI_l2_stm	0x24/0x08	l2_rqsts.rfo_miss
PAPI_l2_tca	0x24/0x3f	l2_rqsts.loads|rfos|ifetches
PAPI_l2_tch	0x24/0x15	l2_rqsts.ld_hit,rfo_hit|ifetch_hit
PAPI_l2_tcm	0x24/0x2a	l2_rqsts.ld_miss,rfo_miss|ifetch_miss
PAPI_l2_tcr	0x24/0x33	l2_rqsts.loads|ifetches
PAPI_l2_tcw	0x24/0x0c	l2_rqsts.rfos
PAPI_l3_tca	0x2e/0x4f	l3_lat_cache.reference
PAPI_l3_tcm	0x2e/0x41	l3_lat_cache.misses
PAPI_ld_ins	0x0b/0x01	mem_inst_retired.loads
PAPI_lst_ins	0x0b/0x03	mem_inst_retired.loads|stores
PAPI_prf_dm	0x26/0xf0	l2_data_rqsts.prefetch.mesi
PAPI_sr_ins	0x0b/0x02	mem_inst_retired.stores
PAPI_tlb_dm	0x49/0x01	dtlb_misses.any

Generic Event	Event Code/Unit Mask	Platform Event
PAPI_tlb_im	0x85/0x01	itlb_misses.any

For fixed-function counter mappings refer to the Intel Core2 listing above.

Intel Atom Processors

Generic Event	Event Code/Unit Mask	Platform Event
PAPI_br_ins	0xc4/0x00	br_inst_retired.any
PAPI_br_msp	0xc5/0x00	br_inst_retired.mispred
PAPI_br_ntk	0xc4/0x03	br_inst_retired.pred_not_taken |mispred_not_taken
PAPI_br_prc	0xc4/0x05	br_inst_retired.pred_not_taken |pred_taken
PAPI_hw_int	0xc8/0x00	hw_int_rcv
PAPI_tot_iis	0xaa/0x03	macro_insts.all_decoded
PAPI_l1_dca	0x40/0x23	l1d_cache.l1|st
PAPI_l2_stm	0x2a/0x41	l2_st.self.i_state
PAPI_l2_tca	0x2e/0x4f	longest_lat_cache.reference
PAPI_l2_tch	0x2e/0x4e	l2_rqsts.mes
PAPI_l2_tcm	0x2e/0x41	longest_lat_cache.miss
PAPI_l2_tcw	0x2a/0x4f	l2_st.self.mesi
PAPI_tlb_dm	0x08/0x07	data_tlb_misses.dtlb.miss
PAPI_tlb_im	0x82/0x02	itlb.misses

For fixed-function counter mappings refer to the Intel Core2 listing above.

AMD Opteron Family 0xF Processor

Generic Event	Platform Event	Unit Mask
PAPI_br_ins	FR_retired_branches_w_excp_intr	0x0
PAPI_br_msp	FR_retired_branches_mispred	0x0
PAPI_br_tkn	FR_retired_taken_branches	0x0
PAPI_fp_ops	FP_dispatched_fpu_ops	0x3
PAPI_fad_ins	FP_dispatched_fpu_ops	0x1

Generic Event	Platform Event	Unit Mask
PAPI_fml_ins	FP_dispatched_fpu_ops	0x2
PAPI_fpu_idl	FP_cycles_no_fpu_ops_retired	0x0
PAPI_tot_cyc	BU_cpu_clk_unhalted	0x0
PAPI_tot_ins	FR_retired_x86_instr_w_excp_intr	0x0
PAPI_l1_dca	DC_access	0x0
PAPI_l1_dcm	DC_miss	0x0
PAPI_l1_ldm	DC_refill_from_L2	0xe
PAPI_l1_stm	DC_refill_from_L2	0x10
PAPI_l1_ica	IC_fetch	0x0
PAPI_l1_icm	IC_miss	0x0
PAPI_l1_icr	IC_fetch	0x0
PAPI_l2_dch	DC_refill_from_L2	0x1e
PAPI_l2_dcm	DC_refill_from_system	0x1e
PAPI_l2_dcr	DC_refill_from_L2	0xe
PAPI_l2_dcw	DC_refill_from_L2	0x10
PAPI_l2_ich	IC_refill_from_L2	0x0
PAPI_l2_icm	IC_refill_from_system	0x0
PAPI_l2_ldm	DC_refill_from_system	0xe
PAPI_l2_stm	DC_refill_from_system	0x10
PAPI_res_stl	FR_dispatch_stalls	0x0
PAPI_stl_icy	FR_nothing_to_dispatch	0x0
PAPI_hw_int	FR_taken_hardware_intrs	0x0
PAPI_tlb_dm	DC_dtlb_L1_miss_L2_miss	0x0
PAPI_tlb_im	IC_itlb_L1_miss_L2_miss	0x0
PAPI_fp_ins	FR_retired_fpu_instr	0xd
PAPI_vec_ins	FR_retired_fpu_instr	0x4

AMD Opteron Family 0x10 Processors

Generic Event	Platform Event	Event Mask
PAPI_br_ins	FR_retired_branches_w_excp_intr	0x0
PAPI_br_msp	FR_retired_branches_mispred	0x0
PAPI_br_tkn	FR_retired_taken_branches	0x0
PAPI_fp_ops	FP_dispatched_fpu_ops	0x3
PAPI_fad_ins	FP_dispatched_fpu_ops	0x1
PAPI_fml_ins	FP_dispatched_fpu_ops	0x2
PAPI_fpu_idl	FP_cycles_no_fpu_ops_retired	0x0
PAPI_tot_cyc	BU_cpu_clk_unhalted	0x0
PAPI_tot_ins	FR_retired_x86_instr_w_excp_intr	0x0
PAPI_l1_dca	DC_access	0x0
PAPI_l1_dcm	DC_miss	0x0
PAPI_l1_ldm	DC_refill_from_L2	0xe
PAPI_l1_stm	DC_refill_from_L2	0x10
PAPI_l1_ica	IC_fetch	0x0
PAPI_l1_icm	IC_miss	0x0
PAPI_l1_icr	IC_fetch	0x0
PAPI_l2_dch	DC_refill_from_L2	0x1e
PAPI_l2_dcm	DC_refill_from_system	0x1e
PAPI_l2_dcr	DC_refill_from_L2	0xe
PAPI_l2_dcw	DC_refill_from_L2	0x10
PAPI_l2_ich	IC_refill_from_L2	0x0
PAPI_l2_icm	IC_refill_from_system	0x0
PAPI_l2_ldm	DC_refill_from_system	0xe
PAPI_l2_stm	DC_refill_from_system	0x10
PAPI_res_stl	FR_dispatch_stalls	0x0
PAPI_stl_icy	FR_nothing_to_dispatch	0x0
PAPI_hw_int	FR_taken_hardware_intrs	0x0
PAPI_tlb_dm	DC_dtlb_L1_miss_L2_miss	0x7

Generic Event	Platform Event	Event Mask
PAPI_tlb_im	IC_itlb_L1_miss_L2_miss	0x3
PAPI_fp_ins	FR_retired_fpu_instr	0xd
PAPI_vec_ins	FR_retired_fpu_instr	0x4
PAPI_l3_dcr	L3_read_req	0xf1
PAPI_l3_icr	L3_read_req	0xf2
PAPI_l3_tcr	L3_read_req	0xf7
PAPI_l3_stm	L3_miss	0xf4
PAPI_l3_ldm	L3_miss	0xf3
PAPI_l3_tcm	L3_miss	0xf7

Intel Pentium IV Processor

Generic Event	Platform Event	Event Mask
PAPI_br_msp	branch_retired	0xa
PAPI_br_ins	branch_retired	0xf
PAPI_br_tkn	branch_retired	0xc
PAPI_br_ntk	branch_retired	0x3
PAPI_br_prc	branch_retired	0x5
PAPI_tot_ins	instr_retired	0x3
PAPI_tot_cyc	global_power_events	0x1
PAPI_tlb_dm	page_walk_type	0x1
PAPI_tlb_im	page_walk_type	0x2
PAPI_tlb_tm	page_walk_type	0x3
PAPI_l2_ldm	BSQ_cache_reference	0x100
PAPI_l2_stm	BSQ_cache_reference	0x400
PAPI_l2_tcm	BSQ_cache_reference	0x500

Intel Pentium Pro/II/III Processor

Generic Event	Platform Event	Event Mask
PAPI_ca_shr	l2_ifetch	0xf
PAPI_ca_cln	bus_tran_rfo	0x0

Generic Event	Platform Event	Event Mask
PAPI_ca_itv	bus_tran_inval	0x0
PAPI_tlb_im	itlb_miss	0x0
PAPI_btac_m	btb_misses	0x0
PAPI_hw_int	hw_int_rx	0x0
PAPI_br_cn	br_inst_retired	0x0
PAPI_br_tkn	br_taken_retired	0x0
PAPI_br_msp	br_miss_pred_taken_ret	0x0
PAPI_br_ins	br_inst_retired	0x0
PAPI_res_stl	resource_stalls	0x0
PAPI_tot_iis	inst_decoder	0x0
PAPI_tot_ins	inst_retired	0x0
PAPI_tot_cyc	cpu_clk_unhalted	0x0
PAPI_l1_dcm	dcu_lines_in	0x0
PAPI_l1_icm	l2_ifetch	0xf
PAPI_l1_tcm	l2_rqsts	0xf
PAPI_l1_dca	data_mem_refs	0x0
PAPI_l1_ldm	l2_ld	0xf
PAPI_l1_stm	l2_st	0xf
PAPI_l2_icm	bus_tran_ifetch	0x0
PAPI_l2_dcr	l2_ld	0xf
PAPI_l2_dcw	l2_st	0xf
PAPI_l2_tcm	l2_lines_in	0x0
PAPI_l2_tca	l2_rqsts	0xf
PAPI_l2_tcw	l2_st	0xf
PAPI_l2_stm	l2_m_lines_inm	0x0
PAPI_fp_ins	flops	0x0
PAPI_fp_ops	flops	0x0
PAPI_fml_ins	mul	0x0

Generic Event	Platform Event	Event Mask
PAPI_fdv_ins	div	0x0

Niagara T1 Processor

Generic Event	Platform Event
PAPI_tot_cyc	Cycle_cnt
PAPI_l2_icm	L2_imiss
PAPI_l2_ldm	L2_dmiss_ld
PAPI_fp_ins	FP_instr_cnt
PAPI_fp_ops	FP_instr_cnt
PAPI_l1_icm	IC_miss
PAPI_l1_dcm	DC_miss
PAPI_tlb_im	ITLB_miss
PAPI_tlb_dm	DTLB_miss

Niagara T2/T2+/T3 Processor

Generic Event	Platform Event
PAPI_tot_ins	Instr_cnt
PAPI_fp_ins	Instr_FGU_arithmetic
PAPI_fp_ops	Instr_FGU_arithmetic
PAPI_l1_dcm	DC_miss
PAPI_l1_icm	IC_miss
PAPI_l2_icm	L2_imiss
PAPI_l2_ldm	L2_dmiss_ld
PAPI_tlb_dm	DTLB_miss
PAPI_tlb_im	ITLB_miss
PAPI_tlb_tm	TLB_miss
PAPI_br_tkn	Br_taken
PAPI_br_ins	Br_completed
PAPI_ld_ins	Instr_ld
PAPI_sr_ins	Instr_st

SPARC64 VI/VII Processor

Generic Event	Platform Event
PAPI_tot_cyc	cycle_counts
PAPI_tot_ins	instruction_counts
PAPI_br_tkn	branch_instructions
PAPI_fp_ops	floating_instructions
PAPI_fma_ins	impdep2_instructions
PAPI_l1_dcm	op_r_iu_req_mi_go
PAPI_l1_icm	if_r_iu_req_mi_go
PAPI_tlb_dm	trap_DMMU_miss
PAPI_tlb_im	trap_IMMU_miss

SPARC T4 Processor

Generic Event	Platform Event
PAPI_br_cn	Branches
PAPI_br_ins	Br_taken
PAPI_br_msp	Br_mispred
PAPI_btac_m	BTC_miss
PAPI_fp_ins	Instr_FGU_crypto
PAPI_tot_ins	Instr_all
PAPI_l1_dcm	DC_miss
PAPI_l1_icm	IC_miss
PAPI_ld_ins	Instr_ld
PAPI_sr_ins	Instr_st
PAPI_tlb_im	ITLB_miss
PAPI_tlb_dm	DTLB_miss_asynch

SPARC M5/T5 Processor

Generic Event	Platform Event
PAPI_br_cn	Branches
PAPI_br_ins	Br_taken
PAPI_br_msp	Br_mispred

Generic Event	Platform Event
PAPI_btac_m	BTC_miss
PAPI_fp_ops	Instr_FGU_crypto
PAPI_fp_ins	Instr_FGU_crypto
PAPI_tot_ins	Instr_all
PAPI_l1_dcm	DC_miss
PAPI_l1_icm	IC_miss
PAPI_ld_ins	Instr_ld
PAPI_sr_ins	Instr_st
PAPI_tlb_im	ITLB_miss
PAPI_tlb_dm	DTLB_miss_asynch

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Volatile

See Also cpc(3CPC), attributes(5)

Notes Generic names prefixed with "PAPI_" are taken from the University of Tennessee's PAPI project, http://icl.cs.utk.edu/papi.

Name ld_support, ld_atexit, ld_atexit64, ld_file, ld_file64, ld_input_done, ld_input_section, ld_input_section64, ld_open, ld_open64, ld_section, ld_section64, ld_start, ld_start64, ld_version – link-editor support functions

Synopsis

```
void ld_atexit(int status);

void ld_atexit64(int status);

void ld_file(const char *name, const Elf_Kind kind, int flags,
     Elf *elf);

void ld_file64(const char *name, const Elf_Kind kind, int flags,
     Elf *elf);

void ld_input_done(uint_t *flags);

void ld_input_section(const char *name, Elf32_Shdr **shdr,
     Elf32_Word sndx, Elf_Data *data, Elf *elf, uint_t *flags);

void ld_input_section64(const char *name, Elf64_Shdr **shdr,
     Elf64_Word sndx, Elf_Data *data, Elf *elf, uint_t *flags);

void ld_open(const char **pname, const char **fname, int *fd,
     int flags, Elf **elf, Elf *ref, size_t off, Elf_kind kind);

void ld_open64(const char **pname, const char **fname, int *fd,
     int flags, Elf **elf, Elf *ref, size_t off, Elf_kind kind);

void ld_section(const char *name, Elf32_Shdr shdr, Elf32_Word sndx,
     Elf_Data *data, Elf *elf);

void ld_section64(const char *name, Elf64_Shdr shdr, Elf64_Word sndx,
     Elf_Data *data, Elf *elf);

void ld_start(const char *name, const Elf32_Half type,
     const char *caller);

void ld_start64(const char *name, const Elf64_Half type,
     const char *caller);

void ld_version(uint_t version);
```

Description A link-editor support library is a user-created shared object offering one or more of these interfaces. These interfaces are called by the link-editor `ld(1)` at various stages of the link-editing process. See the *Linker and Libraries Guide* for a full description of the link-editor support mechanism.

See Also `ld(1)`

Linker and Libraries Guide

Name md4, MD4Init, MD4Update, MD4Final – MD4 digest functions

Synopsis

```
cc [ flag ... ] file ... -lmd [ library ... ]
#include <md4.h>

void MD4Init(MD4_CTX *context);

void MD4Update(MD4_CTX *context, unsigned char *input,
     unsigned int inlen);

void MD4Final(unsigned char *output, MD4_CTX *context);
```

Description The `MD4` functions implement the `MD4` message-digest algorithm. The algorithm takes as input a message of arbitrary length and produces a "fingerprint" or "message digest" as output. The `MD4` message-digest algorithm is intended for digital signature applications in which large files are "compressed" in a secure manner before being encrypted with a private (secret) key under a public-key cryptosystem such as RSA.

`MD4Init()`, `MD4Update()`, `MD4Final()`

The `MD4Init()`, `MD4Update()`, and `MD4Final()` functions allow an `MD4` digest to be computed over multiple message blocks. Between blocks, the state of the `MD4` computation is held in an `MD4` context structure allocated by the caller. A complete digest computation consists of calls to `MD4` functions in the following order: one call to `MD4Init()`, one or more calls to `MD4Update()`, and one call to `MD4Final()`.

The `MD4Init()` function initializes the `MD4` context structure pointed to by *context*.

The `MD4Update()` function computes a partial `MD4` digest on the *inlen*-byte message block pointed to by *input*, and updates the `MD4` context structure pointed to by *context* accordingly.

The `MD4Final()` function generates the final `MD4` digest, using the `MD4` context structure pointed to by *context*. The `MD4` digest is written to output. After a call to `MD4Final()`, the state of the context structure is undefined. It must be reinitialized with `MD4Init()` before it can be used again.

Return Values These functions do not return a value.

Security The `MD4` digest algorithm is not currently considered cryptographically secure. It is included in `libmd(3LIB)` for use by legacy protocols and systems only. It should not be used by new systems or protocols.

Examples **EXAMPLE 1** Authenticate a message found in multiple buffers

The following is a sample function that must authenticate a message that is found in multiple buffers. The calling function provides an authentication buffer that will contain the result of the `MD4` digest.

```
#include <sys/types.h>
#include <sys/uio.h>
#include <md4.h>
```

EXAMPLE 1 Authenticate a message found in multiple buffers *(Continued)*

```
int
AuthenticateMsg(unsigned char *auth_buffer, struct iovec
                *messageIov, unsigned int num_buffers)
{
    MD4_CTX ctx;
    unsigned int i;

    MD4Init(&ctx);

    for(i=0; i<num_buffers; i++)
    {
         MD4Update(&ctx, messageIov->iov_base,
                   messageIov->iov_len);
         messageIov += sizeof(struct iovec);
    }

    MD4Final(auth_buffer, &ctx);

    return 0;
}
```

Attributes

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

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

See Also

libmd(3LIB)

RFC 1320

Name md5, md5_calc, MD5Init, MD5Update, MD5Final – MD5 digest functions

Synopsis

```
cc [ flag ... ] file ... -lmd5 [ library ... ]
#include <md5.h>

void md5_calc(unsigned char *output, unsigned char *input,
     unsigned int inlen);

void MD5Init(MD5_CTX *context);

void MD5Update(MD5_CTX *context, unsigned char *input,
     unsigned int inlen);

void MD5Final(unsigned char *output, MD5_CTX *context);
```

Description These functions implement the MD5 message-digest algorithm, which takes as input a message of arbitrary length and produces as output a 128-bit "fingerprint" or "message digest" of the input. It is intended for digital signature applications, where large file must be "compressed" in a secure manner before being encrypted with a private (secret) key under a public-key cryptosystem such as RSA.

`md5_calc()` The `md5_calc()` function computes an MD5 digest on a single message block. The *inlen*-byte block is pointed to by *input*, and the 16-byte MD5 digest is written to *output*.

`MD5Init()`, `MD5Update()`, `MD5Final()` The `MD5Init()`, `MD5Update()`, and `MD5Final()` functions allow an MD5 digest to be computed over multiple message blocks; between blocks, the state of the MD5 computation is held in an MD5 context structure, allocated by the caller. A complete digest computation consists of one call to `MD5Init()`, one or more calls to `MD5Update()`, and one call to `MD5Final()`, in that order.

The `MD5Init()` function initializes the MD5 context structure pointed to by *context*.

The `MD5Update()` function computes a partial MD5 digest on the *inlen*-byte message block pointed to by *input*, and updates the MD5 context structure pointed to by *context* accordingly.

The `MD5Final()` function generates the final MD5 digest, using the MD5 context structure pointed to by *context*; the 16-byte MD5 digest is written to *output*. After calling `MD5Final()`, the state of the context structure is undefined; it must be reinitialized with `MD5Init()` before being used again.

Return Values These functions do not return a value.

Examples **EXAMPLE 1** Authenticate a message found in multiple buffers

The following is a sample function that must authenticate a message that is found in multiple buffers. The calling function provides an authentication buffer that will contain the result of the MD5 digest.

```
#include <sys/types.h>
#include <sys/uio.h>
#include <md5.h>
```

EXAMPLE 1 Authenticate a message found in multiple buffers *(Continued)*

```
int
AuthenticateMsg(unsigned char *auth_buffer, struct iovec
                *messageIov, unsigned int num_buffers)
{
    MD5_CTX md5_context;
    unsigned int i;

    MD5Init(&md5_context);

    for(i=0; i<num_buffers; i++)
    {
         MD5Update(&md5_context, messageIov->iov_base,
                   messageIov->iov_len);
         messageIov += sizeof(struct iovec);
    }

    MD5Final(auth_buffer, &md5_context);

    return 0;
}
```

EXAMPLE 2 Use `md5_calc()` to generate the MD5 digest

Since the buffer to be computed is contiguous, the `md5_calc()` function can be used to generate the MD5 digest.

```
int AuthenticateMsg(unsigned char *auth_buffer, unsigned
                    char *buffer, unsigned int length)
{
    md5_calc(buffer, auth_buffer, length);

    return (0);
}
```

Attributes

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

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

See Also

`libmd5(3LIB)`

Rivest, R., The MD5 Message-Digest Algorithm, RFC 1321, April 1992.

Name nlist – get entries from name list

Synopsis

```
cc [ flag... ] file ... -lelf [ library ... ]
#include <nlist.h>

int nlist(const char *filename, struct nlist *nl);
```

Description `nlist()` examines the name list in the executable file whose name is pointed to by *filename*, and selectively extracts a list of values and puts them in the array of `nlist()` structures pointed to by `nl`. The name list `nl` consists of an array of structures containing names of variables, types, and values. The list is terminated with a null name, that is, a null string is in the name position of the structure. Each variable name is looked up in the name list of the file. If the name is found, the type, value, storage class, and section number of the name are inserted in the other fields. The type field may be set to 0 if the file was not compiled with the `-g` option to `cc`.

`nlist()` will always return the information for an external symbol of a given name if the name exists in the file. If an external symbol does not exist, and there is more than one symbol with the specified name in the file (such as static symbols defined in separate files), the values returned will be for the last occurrence of that name in the file. If the name is not found, all fields in the structure except `n_name` are set to 0.

This function is useful for examining the system name list kept in the file `/dev/ksyms`. In this way programs can obtain system addresses that are up to date.

Return Values All value entries are set to 0 if the file cannot be read or if it does not contain a valid name list.

`nlist()` returns 0 on success, −1 on error.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `elf(3ELF)`, `kvm_nlist(3KVM)`, `kvm_open(3KVM)`, `libelf(3LIB)`, `a.out(4)`, `attributes(5)`, `ksyms(7D)`, `mem(7D)`

Name NOTE, _NOTE – annotate source code with info for tools

Synopsis

```
#include <note.h>

NOTE(NoteInfo)

#include<sys/note.h>

_NOTE(NoteInfo)
```

Description These macros are used to embed information for tools in program source. A use of one of these macros is called an "annotation". A tool may define a set of such annotations which can then be used to provide the tool with information that would otherwise be unavailable from the source code.

Annotations should, in general, provide documentation useful to the human reader. If information is of no use to a human trying to understand the code but is necessary for proper operation of a tool, use another mechanism for conveying that information to the tool (one which does not involve adding to the source code), so as not to detract from the readability of the source. The following is an example of an annotation which provides information of use to a tool and to the human reader (in this case, which data are protected by a particular lock, an annotation defined by the static lock analysis tool `lock_lint`).

```
NOTE(MUTEX_PROTECTS_DATA(foo_lock, foo_list Foo))
```

Such annotations do not represent executable code; they are neither statements nor declarations. They should not be followed by a semicolon. If a compiler or tool that analyzes C source does not understand this annotation scheme, then the tool will ignore the annotations. (For such tools, `NOTE(`*x*`)` expands to nothing.)

Annotations may only be placed at particular places in the source.

These places are where the following C constructs would be allowed:

- a top-level declaration (that is, a declaration not within a function or other construct)
- a declaration or statement within a block (including the block which defines a function)
- a member of a `struct` or `union`.

Annotations are not allowed in any other place. For example, the following are illegal:

```
x = y + NOTE(...) z ;
typedef NOTE(...) unsigned int uint ;
```

While `NOTE` and `_NOTE` may be used in the places described above, a particular type of annotation may only be allowed in a subset of those places. For example, a particular annotation may not be allowed inside a `struct` or `union` definition.

NOTE vs _NOTE Ordinarily, `NOTE` should be used rather than `_NOTE`, since use of `_NOTE` technically makes a program non-portable. However, it may be inconvenient to use `NOTE` for this purpose in existing code if `NOTE` is already heavily used for another purpose. In this case one should use a different macro and write a header file similar to `/usr/include/note.h` which maps that macro to `_NOTE` in the same manner. For example, the following makes `FOO` such a macro:

```
#ifndef _FOO_H
#define _FOO_H
#define FOO _NOTE
#include <sys/note.h>
#endif
```

Public header files which span projects should use `_NOTE` rather than `NOTE`, since `NOTE` may already be used by a program which needs to include such a header file.

***NoteInfo* Argument** The actual *NoteInfo* used in an annotation should be specified by a tool that deals with program source (see the documentation for the tool to determine which annotations, if any, it understands).

NoteInfo must have one of the following forms:

NoteName
NoteName(*Args*)

where *NoteName* is simply an identifier which indicates the type of annotation, and *Args* is something defined by the tool that specifies the particular *NoteName*. The general restrictions on *Args* are that it be compatible with an ANSI C tokenizer and that unquoted parentheses be balanced (so that the end of the annotation can be determined without intimate knowledge of any particular annotation).

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also

`note(4)`, `attributes(5)`

Name pctx_capture, pctx_create, pctx_run, pctx_release – process context library

Synopsis

```
cc [ flag... ] file... –lpctx [ library... ]
#include <libpctx.h>

typedef void (pctx_errfn_t)(const char *fn, const char *fmt, va_list ap);

pctx_t *pctx_create(const char *filename, char *const *argv, void *arg,
     int verbose, pctx_errfn_t *errfn);

pctx_t *pctx_capture(pid_t pid, void *arg, int verbose,
     pctx_errfn_t *errfn);

int pctx_run(pctx_t *pctx, uint_t sample, uint_t nsamples,
     int (*tick)(pctx *, pid_t, id_t, void *));

void pctx_release(pctx_t *pctx);
```

Description This family of functions allows a controlling process (the process that invokes them) to create or capture controlled processes. The functions allow the occurrence of various events of interest in the controlled process to cause the controlled process to be stopped, and to cause callback routines to be invoked in the controlling process.

`pctx_create()` and `pctx_capture()` There are two ways a process can be acquired by the process context functions. First, a named application can be invoked with the usual *argv*[] array using `pctx_create()`, which forks the caller and `exec`s the application in the child. Alternatively, an existing process can be captured by its process ID using `pctx_capture()`.

Both functions accept a pointer to an opaque handle, *arg*; this is saved and treated as a caller-private handle that is passed to the other functions in the library. Both functions accept a pointer to a `printf`(3C)-like error routine *errfn*; a default version is provided if `NULL` is specified.

A freshly-created process is created stopped; similarly, a process that has been successfully captured is stopped by the act of capturing it, thereby allowing the caller to specify the handlers that should be called when various events occur in the controlled process. The set of handlers is listed on the `pctx_set_events`(3CPC) manual page.

`pctx_run()` Once the callback handlers have been set with `pctx_set_events()`, the application can be set running using `pctx_run()`. This function starts the event handling loop; it returns only when either the process has exited, the number of time samples has expired, or an error has occurred (for example, if the controlling process is not privileged, and the controlled process has `exec`-ed a setuid program).

Every *sample* milliseconds the process is stopped and the *tick*( ) routine is called so that, for example, the performance counters can be sampled by the caller. No periodic sampling is performed if *sample* is 0.

`pctx_release()` Once `pctx_run()` has returned, the process can be released and the underlying storage freed using `pctx_release()`. Releasing the process will either allow the controlled process to continue (in the case of an existing captured process and its children) or kill the process (if it and its children were created using `pctx_create()`).

Return Values

Upon successful completion, `pctx_capture()` and `pctx_create()` return a valid handle. Otherwise, the functions print a diagnostic message and return `NULL`.

Upon successful completion, `pctx_run()` returns `0` with the controlled process either stopped or exited (if the controlled process has invoked `exit(2)`.) If an error has occurred (for example, if the controlled process has `exec`–ed a set-ID executable, if certain callbacks have returned error indications, or if the process was unable to respond to `proc(4)` requests) an error message is printed and the function returns −1.

Usage

Within an event handler in the controlling process, the controlled process can be made to perform various system calls on its behalf. No system calls are directly supported in this version of the API, though system calls are executed by the `cpc_pctx` family of interfaces in `libcpc` such as `cpc_pctx_bind_event(3CPC)`. A specially created agent LWP is used to execute these system calls in the controlled process. See `proc(4)` for more details.

While executing the event handler functions, the library arranges for the signals `SIGTERM`, `SIGQUIT`, `SIGABRT`, and `SIGINT` to be blocked to reduce the likelihood of a keyboard signal killing the controlling process prematurely, thereby leaving the controlled process permanently stopped while the agent LWP is still alive inside the controlled process.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe

See Also

`fork(2)`, `cpc(3CPC)`, `pctx_set_events(3CPC)`, `libpctx(3LIB)`, `proc(4)`, `attributes(5)`

Name pctx_set_events – associate callbacks with process events

Synopsis

```
cc [ flag... ] file... −lpctx [ library... ]
#include <libpctx.h>

typedef    enum {
        PCTX_NULL_EVENT = 0,
        PCTX_SYSC_EXEC_EVENT,
        PCTX_SYSC_FORK_EVENT,
        PCTX_SYSC_EXIT_EVENT,
        PCTX_SYSC_LWP_CREATE_EVENT,
        PCTX_INIT_LWP_EVENT,
        PCTX_FINI_LWP_EVENT,
        PCTX_SYSC_LWP_EXIT_EVENT
} pctx_event_t;

typedef int pctx_sysc_execfn_t(pctx_t *pctx, pid_t pid, id_t lwpid,
     char *cmd, void *arg);

typedef void pctx_sysc_forkfn_t(pctx_t *pctx,
     pid_t pid, id_t lwpid, pid_t child, void *arg);

typedef void pctx_sysc_exitfn_t(pctx_t *pctx, pid_t pid, id_t lwpid,
     void *arg);

typedef int pctx_sysc_lwp_createfn_t(pctx_t *pctx, pid_t pid, id_t lwpid,
     void *arg);

typedef int pctx_init_lwpfn_t(pctx_t *pctx, pid_t pid, id_t lwpid,
     void *arg);

typedef int pctx_fini_lwpfn_t(pctx_t *pctx, pid_t pid, id_t lwpid,
     void *arg);

typedef int pctx_sysc_lwp_exitfn_t(pctx_t *pctx, pid_t pid, id_t lwpid,
     void *arg);

int pctx_set_events(pctx_t *pctx...);
```

Description The `pctx_set_events()` function allows the caller (the controlling process) to express interest in various events in the controlled process. See `pctx_capture(3CPC)` for information about how the controlling process is able to create, capture and manipulate the controlled process.

The `pctx_set_events()` function takes a `pctx_t` handle, followed by a variable length list of pairs of `pctx_event_t` tags and their corresponding handlers, terminated by a `PCTX_NULL_EVENT` tag.

Most of the events correspond closely to various classes of system calls, though two additional pseudo-events (*init_lwp* and *fini_lwp*) are provided to allow callers to perform various housekeeping tasks. The *init_lwp* handler is called as soon as the library identifies a new LWP, while *fini_lwp* is called just before the LWP disappears. Thus the classic “hello world” program would see an *init_lwp* event, a *fini_lwp* event and (process) *exit* event, in that order. The table below displays the interactions between the states of the controlled process and the handlers executed by users of the library.

System Calls and pctx Handlers

System call	Handler	Comments
exec,execve	*fini_lwp*	Invoked serially on all lwps in the process.
	exec	Only invoked if the `exec()` system call succeeded.
	init_lwp	If the exec succeeds, only invoked on lwp 1. If the exec fails, invoked serially on all lwps in the process.
`fork`, `vfork`, `fork1`	*fork*	Only invoked if the `fork()` system call succeeded.
exit	*fini_lwp*	Invoked on all lwps in the process.
	exit	Invoked on the exiting lwp.

Each of the handlers is passed the caller's opaque handle, a `pctx_t` handle, the pid, and lwpid of the process and lwp generating the event. The *lwp_exit*, and (process) `exit` events are delivered *before* the underlying system calls begin, while the `exec`, *fork*, and *lwp_create* events are only delivered after the relevant system calls complete successfully. The `exec` handler is passed a string that describes the command being executed. Catching the *fork* event causes the calling process to `fork(2)`, then capture the child of the controlled process using `pctx_capture()` before handing control to the *fork* handler. The process is released on return from the handler.

Return Values

Upon successful completion, `pctx_set_events()` returns 0. Otherwise, the function returns −1.

Examples

EXAMPLE 1 HandleExec example.

This example captures an existing process whose process identifier is *pid*, and arranges to call the *HandleExec* routine when the process performs an `exec(2)`.

```
static void
HandleExec(pctx_t *pctx, pid_t pid, id_t lwpid, char *cmd, void *arg)
{
     (void) printf("pid %d execed '%s'\n", (int)pid, cmd);
}
int
main()
{
     ...
     pctx = pctx_capture(pid, NULL, 1, NULL);
     (void) pctx_set_events(pctx,
```

EXAMPLE 1 HandleExec example. *(Continued)*

```
            PCTX_SYSC_EXEC_EVENT, HandleExec,
            ...
            PCTX_NULL_EVENT);
     (void) pctx_run(pctx, 0, 0, NULL);
     pctx_release(pctx);
}
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe

See Also

exec(2), exit(2), fork(2), vfork(2), fork1(2), cpc(3CPC), libpctx(3LIB), proc(4), attributes(5)

Name queue, SLIST_HEAD, SLIST_HEAD_INITIALIZER, SLIST_ENTRY, SLIST_INIT, SLIST_INSERT_AFTER, SLIST_INSERT_HEAD, SLIST_REMOVE_HEAD, SLIST_REMOVE, SLIST_FOREACH, SLIST_EMPTY, SLIST_FIRST, SLIST_NEXT, SIMPLEQ_HEAD, SIMPLEQ_HEAD_INITIALIZER, SIMPLEQ_ENTRY, SIMPLEQ_INIT, SIMPLEQ_INSERT_HEAD, SIMPLEQ_INSERT_TAIL, SIMPLEQ_INSERT_AFTER, SIMPLEQ_REMOVE_HEAD, SIMPLEQ_REMOVE, SIMPLEQ_FOREACH, SIMPLEQ_EMPTY, SIMPLEQ_FIRST, SIMPLEQ_NEXT, STAILQ_HEAD, STAILQ_HEAD_INITIALIZER, STAILQ_ENTRY, STAILQ_INIT, STAILQ_INSERT_HEAD, STAILQ_INSERT_TAIL, STAILQ_INSERT_AFTER, STAILQ_REMOVE_HEAD, STAILQ_REMOVE, STAILQ_FOREACH, STAILQ_EMPTY, STAILQ_FIRST, STAILQ_NEXT, STAILQ_CONCAT, LIST_HEAD, LIST_HEAD_INITIALIZER, LIST_ENTRY, LIST_INIT, LIST_INSERT_AFTER, LIST_INSERT_BEFORE, LIST_INSERT_HEAD, LIST_REMOVE, LIST_FOREACH, LIST_EMPTY, LIST_FIRST, LIST_NEXT, TAILQ_HEAD, TAILQ_HEAD_INITIALIZER, TAILQ_ENTRY, TAILQ_INIT, TAILQ_INSERT_HEAD, TAILQ_INSERT_TAIL, TAILQ_INSERT_AFTER, TAILQ_INSERT_BEFORE, TAILQ_REMOVE, TAILQ_FOREACH, TAILQ_FOREACH_REVERSE, TAILQ_EMPTY, TAILQ_FIRST, TAILQ_NEXT, TAILQ_LAST, TAILQ_PREV, TAILQ_CONCAT, CIRCLEQ_HEAD, CIRCLEQ_HEAD_INITIALIZER, CIRCLEQ_ENTRY, CIRCLEQ_INIT, CIRCLEQ_INSERT_AFTER, CIRCLEQ_INSERT_BEFORE, CIRCLEQ_INSERT_HEAD, CIRCLEQ_INSERT_TAIL, CIRCLEQ_REMOVE, CIRCLEQ_FOREACH, CIRCLEQ_FOREACH_REVERSE, CIRCLEQ_EMPTY, CIRCLEQ_FIRST, CIRCLEQ_LAST, CIRCLEQ_NEXT, CIRCLEQ_PREV, CIRCLEQ_LOOP_NEXT, CIRCLEQ_LOOP_PREV – implementations of singly-linked lists, simple queues, lists, tail queues, and circular queues

Synopsis

```
#include <sys/queue.h>

SLIST_HEAD(HEADNAME, TYPE);

SLIST_HEAD_INITIALIZER(head);

SLIST_ENTRY(TYPE);

SLIST_INIT(SLIST_HEAD *head)

SLIST_INSERT_AFTER(TYPE *listelm, TYPE *elm, SLIST_ENTRY NAME);

SLIST_INSERT_HEAD(SLIST_HEAD *head, TYPE *elm, SLIST_ENTRY NAME)

SLIST_REMOVE_HEAD(SLIST_HEAD *head, SLIST_ENTRY NAME);

SLIST_REMOVE(SLIST_HEAD *head, TYPE *elm, TYPE, SLIST_ENTRY NAME);

SLIST_FOREACH(TYPE *var, SLIST_HEAD *head, SLIST_ENTRY NAME);

int SLIST_EMPTY(SLIST_HEAD *head);

TYPE *SLIST_FIRST(SLIST_HEAD *head);

TYPE *SLIST_NEXT(TYPE *elm, SLIST_ENTRY NAME);
```

```
SIMPLEQ_HEAD(HEADNAME, TYPE);

SIMPLEQ_HEAD_INITIALIZER(head);

SIMPLEQ_ENTRY(TYPE);

SIMPLEQ_INIT(SIMPLEQ_HEAD *head);

SIMPLEQ_INSERT_HEAD(SIMPLEQ_HEAD *head, TYPE *elm, SIMPLEQ_ENTRY NAME);

SIMPLEQ_INSERT_TAIL(SIMPLEQ_HEAD *head, TYPE *elm, SIMPLEQ_ENTRY NAME);

SIMPLEQ_INSERT_AFTER(SIMPLEQ_HEAD *head, TYPE *listelm, TYPE *elm,
     SIMPLEQ_ENTRY NAME);

SIMPLEQ_REMOVE_HEAD(SIMPLEQ_HEAD *head, SIMPLEQ_ENTRY NAME);

SIMPLEQ_REMOVE(SIMPLEQ_HEAD *head, TYPE *elm, TYPE, SIMPLEQ_ENTRY NAME);

SIMPLEQ_FOREACH(TYPE *var, SIMPLEQ_HEAD *head, SIMPLEQ_ENTRY NAME);

int SIMPLEQ_EMPTY(SIMPLEQ_HEAD *head)

TYPE *SIMPLEQ_FIRST(SIMPLEQ_HEAD *head);

TYPE *SIMPLEQ_NEXT(TYPE *elm, SIMPLEQ_ENTRY NAME);

STAILQ_HEAD(HEADNAME, TYPE);

STAILQ_HEAD_INITIALIZER(head);

STAILQ_ENTRY(TYPE);

STAILQ_INIT(STAILQ_HEAD *head);

STAILQ_INSERT_HEAD(STAILQ_HEAD *head, TYPE *elm, STAILQ_ENTRY NAME);

STAILQ_INSERT_TAIL(STAILQ_HEAD *head, TYPE *elm, STAILQ_ENTRY NAME);

STAILQ_INSERT_AFTER(STAILQ_HEAD *head, TYPE *listelm, TYPE *elm,
     STAILQ_ENTRY NAME);

STAILQ_REMOVE_HEAD(STAILQ_HEAD *head, STAILQ_ENTRY NAME);

STAILQ_REMOVE(STAILQ_HEAD *head, TYPE *elm, TYPE, STAILQ_ENTRY NAME);

STAILQ_FOREACH(TYPE *var, STAILQ_HEAD *head, STAILQ_ENTRY NAME);

int STAILQ_EMPTY(STAILQ_HEAD *head);

TYPE *STAILQ_FIRST(STAILQ_HEAD *head);

TYPE *STAILQ_NEXT(TYPE *elm, STAILQ_ENTRY NAME);

STAILQ_CONCAT(STAILQ_HEAD *head1, STAILQ_HEAD *head2);

LIST_HEAD(HEADNAME, TYPE);

LIST_HEAD_INITIALIZER(head);

LIST_ENTRY(TYPE);
```

```
LIST_INIT(LIST_HEAD *head);

LIST_INSERT_AFTER(TYPE *listelm, TYPE *elm, LIST_ENTRY NAME);

LIST_INSERT_BEFORE(TYPE *listelm, TYPE *elm, LIST_ENTRY NAME);

LIST_INSERT_HEAD(LIST_HEAD *head, TYPE *elm, LIST_ENTRY NAME);

LIST_REMOVE(TYPE *elm, LIST_ENTRY NAME);

LIST_FOREACH(TYPE *var, LIST_HEAD *head, LIST_ENTRY NAME);

int LIST_EMPTY(LIST_HEAD *head);

TYPE *LIST_FIRST(LIST_HEAD *head);

TYPE *LIST_NEXT(TYPE *elm, LIST_ENTRY NAME);

TAILQ_HEAD(HEADNAME, TYPE);

TAILQ_HEAD_INITIALIZER(head);

TAILQ_ENTRY(TYPE);

TAILQ_INIT(TAILQ_HEAD *head);

TAILQ_INSERT_HEAD(TAILQ_HEAD *head, TYPE *elm, TAILQ_ENTRY NAME);

TAILQ_INSERT_TAIL(TAILQ_HEAD *head, TYPE *elm, TAILQ_ENTRY NAME)

TAILQ_INSERT_AFTER(TAILQ_HEAD *head, TYPE *listelm, TYPE *elm,
     TAILQ_ENTRY NAME);

TAILQ_INSERT_BEFORE(TYPE *listelm, TYPE *elm, TAILQ_ENTRY NAME);

TAILQ_REMOVE(TAILQ_HEAD *head, TYPE *elm, TAILQ_ENTRY NAME);

TAILQ_FOREACH(TYPE *var, TAILQ_HEAD *head, TAILQ_ENTRY NAME);

TAILQ_FOREACH_REVERSE(TYPE *var, TAILQ_HEAD *head, HEADNAME,
     TAILQ_ENTRY NAME);

int TAILQ_EMPTY(TAILQ_HEAD *head);

TYPE *TAILQ_FIRST(TAILQ_HEAD *head);

TYPE *TAILQ_NEXT(TYPE *elm, TAILQ_ENTRY NAME);

TYPE *TAILQ_LAST(TAILQ_HEAD *head, HEADNAME);

TYPE *TAILQ_PREV(TYPE *elm, HEADNAME, TAILQ_ENTRY NAME);

TAILQ_CONCAT(TAILQ_HEAD *head1, TAILQ_HEAD *head2, TAILQ_ENTRY NAME);

CIRCLEQ_HEAD(HEADNAME, TYPE);

CIRCLEQ_HEAD_INITIALIZER(head);

CIRCLEQ_ENTRY(TYPE);

CIRCLEQ_INIT(CIRCLEQ_HEAD *head);
```

```
CIRCLEQ_INSERT_AFTER(CIRCLEQ_HEAD *head, TYPE *listelm, TYPE *elm,
     CIRCLEQ_ENTRY NAME);

CIRCLEQ_INSERT_BEFORE(CIRCLEQ_HEAD *head, TYPE *listelm, TYPE *elm,
     CIRCLEQ_ENTRY NAME);

CIRCLEQ_INSERT_HEAD(CIRCLEQ_HEAD *head, TYPE *elm, CIRCLEQ_ENTRY NAME);

CIRCLEQ_INSERT_TAIL(CIRCLEQ_HEAD *head, TYPE *elm, CIRCLEQ_ENTRY NAME);

CIRCLEQ_REMOVE(CIRCLEQ_HEAD *head, TYPE *elm, CIRCLEQ_ENTRY NAME);

CIRCLEQ_FOREACH(TYPE *var, CIRCLEQ_HEAD *head, CIRCLEQ_ENTRY NAME);

CIRCLEQ_FOREACH_REVERSE(TYPE *var, CIRCLEQ_HEAD *head,
     CIRCLEQ_ENTRY NAME);

int CIRCLEQ_EMPTY(CIRCLEQ_HEAD *head);

TYPE *CIRCLEQ_FIRST(CIRCLEQ_HEAD *head);

TYPE *CIRCLEQ_LAST(CIRCLEQ_HEAD *head);

TYPE *CIRCLEQ_NEXT(TYPE *elm, CIRCLEQ_ENTRY NAME);

TYPE *CIRCLEQ_PREV(TYPE *elm, CIRCLEQ_ENTRY NAME);

TYPE *CIRCLEQ_LOOP_NEXT(CIRCLEQ_HEAD *head, TYPE *elm, CIRCLEQ_ENTRY NAME);

TYPE *CIRCLEQ_LOOP_PREV(CIRCLEQ_HEAD *head, TYPE *elm, CIRCLEQ_ENTRY NAME);
```

Description

These macros define and operate on five types of data structures: singly- linked lists, simple queues, lists, tail queues, and circular queues. All five structures support the following functionality:

1. Insertion of a new entry at the head of the list.
2. Insertion of a new entry before or after any element in the list.
3. Removal of any entry in the list.
4. Forward traversal through the list.

Singly-linked lists are the simplest of the five data structures and support only the above functionality. Singly-linked lists are ideal for applications with large datasets and few or no removals, or for implementing a LIFO queue.

1. Entries can be added at the end of a list.
2. They may be concatenated.

However:

1. Entries may not be added before any element in the list.
2. All list insertions and removals must specify the head of the list.
3. Each head entry requires two pointers rather than one.

Simple queues are ideal for applications with large datasets and few or no removals, or for implementing a FIFO queue.

All doubly linked types of data structures (lists, tail queues, and circle queues) additionally allow:

1. Insertion of a new entry before any element in the list.
2. O(1) removal of any entry in the list.

However:

1. Each element requires two pointers rather than one.
2. Code size and execution time of operations (except for removal) is about twice that of the singly-linked data structures

Linked lists are the simplest of the doubly linked data structures and support only the above functionality over singly-linked lists.

Tail queues add the following functionality:

1. Entries can be added at the end of a list.
2. They may be concatenated.

However:

1. All list insertions and removals, except insertion before another element, must specify the head of the list.
2. Each head entry requires two pointers rather than one.
3. Code size is about 15% greater and operations run about 20% slower than lists.

Circular queues add the following functionality:

1. Entries can be added at the end of a list.
2. They may be traversed backwards, from tail to head.

However:

1. All list insertions and removals must specify the head of the list.
2. Each head entry requires two pointers rather than one.
3. The termination condition for traversal is more complex.
4. Code size is about 40% greater and operations run about 45% slower than lists.

In the macro definitions, TYPE is the name of a user defined structure, that must contain a field of type `LIST_ENTRY`, `SIMPLEQ_ENTRY`, `SLIST_ENTRY`, `TAILQ_ENTRY`, or `CIRCLEQ_ENTRY`, named *NAME*. The argument *HEADNAME* is the name of a user defined structure that must be declared using the macros `LIST_HEAD()`, `SIMPLEQ_HEAD()`, `SLIST_HEAD()`, `TAILQ_HEAD()`, or `CIRCLEQ_HEAD()`. See the examples below for further explanation of how these macros are used.

Summary of Operations

The following table summarizes the supported macros for each type of data structure.

	SLIST	LIST	SIMPLEQ	STAILQ	TAILQ	CIRCLEQ
_EMPTY	+	+	+	+	+	+
_FIRST	+	+	+	+	+	+
_FOREACH	+	+	+	+	+	+
_FOREACH_REVERSE	-	-	-	-	+	+
_INSERT_AFTER	+	+	+	+	+	+
_INSERT_BEFORE	-	+	-	-	+	+
_INSERT_HEAD	+	+	+	+	+	+
_INSERT_TAIL	-	-	+	+	+	+
_LAST	-	-	-	-	+	+
_LOOP_NEXT	-	-	-	-	-	+
_LOOP_PREV	-	-	-	-	-	+
_NEXT	+	+	+	+	+	+
_PREV	-	-	-	-	+	+
_REMOVE	+	+	+	+	+	+
_REMOVE_HEAD	+	-	+	+	-	-
_CONCAT	-	-	-	+	+	-

Singly-linked Lists

A singly-linked list is headed by a structure defined by the `SLIST_HEAD()` macro. This structure contains a single pointer to the first element on the list. The elements are singly linked for minimum space and pointer manipulation overhead at the expense of O(n) removal for arbitrary elements. New elements can be added to the list after an existing element or at the head of the list. An `SLIST_HEAD` structure is declared as follows:

`SLIST_HEAD(`*HEADNAME*`,` *TYPE*`)` *head*`;`

where *HEADNAME* is the name of the structure to be defined, and *TYPE* is the type of the elements to be linked into the list. A pointer to the head of the list can later be declared as:

`struct` *HEADNAME *headp*`;`

The names *head* and *headp* are user selectable.

The macro `SLIST_HEAD_INITIALIZER()` evaluates to an initializer for the list head

The macro `SLIST_EMPTY()` evaluates to true if there are no elements in the list.

The macro `SLIST_ENTRY()` declares a structure that connects the elements in the list.

The macro `SLIST_FIRST()` returns the first element in the list or `NULL` if the list is empty.

The macro `SLIST_FOREACH()` traverses the list referenced by head in the forward direction, assigning each element in turn to *var*.

The macro `SLIST_INIT()` initializes the list referenced by *head*.

The macro `SLIST_INSERT_HEAD()` inserts the new element *elm* at the head of the list.

The macro `SLIST_INSERT_AFTER()` inserts the new element *elm* after the element *listelm*.

The macro `SLIST_NEXT()` returns the next element in the list.

The macro `SLIST_REMOVE()` removes the element *elm* from the list.

The macro `SLIST_REMOVE_HEAD()` removes the first element from the head of the list. For optimum efficiency, elements being removed from the head of the list should explicitly use this macro instead of the generic `SLIST_REMOVE()` macro.

Singly-linked List Example

```
SLIST_HEAD(slisthead, entry) head =
    SLIST_HEAD_INITIALIZER(head);
struct slisthead *headp;                /* Singly-linked List head. */
struct entry {
        ...
        SLIST_ENTRY(entry) entries;     /* Singly-linked List. */
        ...
} *n1, *n2, *n3, *np;

SLIST_INIT(&head);                      /* Initialize the list. */

n1 = malloc(sizeof(struct entry));      /* Insert at the head. */
SLIST_INSERT_HEAD(&head, n1, entries);

n2 = malloc(sizeof(struct entry));      /* Insert after. */
SLIST_INSERT_AFTER(n1, n2, entries);

SLIST_REMOVE(&head, n2, entry, entries);/* Deletion. */
free(n2);

n3 = SLIST_FIRST(&head);
SLIST_REMOVE_HEAD(&head, entries);      /* Deletion from the head. */
free(n3);
                                        /* Forward traversal. */
SLIST_FOREACH(np, &head, entries)
        np-> ...

while (!SLIST_EMPTY(&head)) {           /* List Deletion. */
        n1 = SLIST_FIRST(&head);
        SLIST_REMOVE_HEAD(&head, entries);
        free(n1);
}
```

Simple Queues

A simple queue is headed by a structure defined by the `SIMPLEQ_HEAD()` macro. This structure contains a pair of pointers, one to the first element in the simple queue and the other to the last element in the simple queue. The elements are singly linked for minimum space and pointer manipulation overhead at the expense of O(n) removal for arbitrary elements. New elements can be added to the queue after an existing element, at the head of the queue, or at the end of the queue. A `SIMPLEQ_HEAD` structure is declared as follows:

`SIMPLEQ_HEAD(`*HEADNAME*`,` *TYPE*`)` *head*`;`

where *HEADNAME* is the name of the structure to be defined, and *TYPE* is the type of the elements to be linked into the simple queue. A pointer to the head of the simple queue can later be declared as:

`struct HEADNAME *`*headp*`;`

The names *head* and *headp* are user selectable.

The macro `SIMPLEQ_ENTRY()` declares a structure that connects the elements in the simple queue.

The macro `SIMPLEQ_HEAD_INITIALIZER()` provides a value which can be used to initialize a simple queue head at compile time, and is used at the point that the simple queue head variable is declared, like:

`struct` *HEADNAME* *head* `= SIMPLEQ_HEAD_INITIALIZER(`*head*`);`

The macro `SIMPLEQ_INIT()` initializes the simple queue referenced by *head*.

The macro `SIMPLEQ_INSERT_HEAD()` inserts the new element elm at the head of the simple queue.

The macro `SIMPLEQ_INSERT_TAIL()` inserts the new element elm at the end of the simple queue.

The macro `SIMPLEQ_INSERT_AFTER()` inserts the new element *elm* after the element *listelm*.

The macro `SIMPLEQ_REMOVE()` removes *elm* from the simple queue.

The macro `SIMPLEQ_REMOVE_HEAD()` removes the first element from the head of the simple queue. For optimum efficiency, elements being removed from the head of the queue should explicitly use this macro instead of the generic `SIMPLQ_REMOVE()` macro.

The macro `SIMPLEQ_EMPTY()` return true if the simple queue head has no elements.

The macro `SIMPLEQ_FIRST()` returns the first element of the simple queue head.

The macro `SIMPLEQ_FOREACH()` traverses the tail queue referenced by head in the forward direction, assigning each element in turn to *var*.

The macro `SIMPLEQ_NEXT()` returns the element after the element *elm*.

The macros prefixed with "`STAILQ_`" (`STAILQ_HEAD()`, `STAILQ_HEAD_INITIALIZER()`, `STAILQ_ENTRY()`, `STAILQ_INIT()`, `STAILQ_INSERT_HEAD()`, `STAILQ_INSERT_TAIL()`, `STAILQ_INSERT_AFTER()`, `STAILQ_REMOVE_HEAD()`, `STAILQ_REMOVE()`, `STAILQ_FOREACH()`, `STAILQ_EMPTY()`, `STAILQ_FIRST()`, and `STAILQ_NEXT()`) are functionally identical to these simple queue functions, and are provided for compatibility with FreeBSD.

Simple Queue Example

```
SIMPLEQ_HEAD(simplehead, entry) head;
struct simplehead *headp;               /* Simple queue head. */
struct entry {
        ...
        SIMPLEQ_ENTRY(entry) entries;   /* Simple queue. */
        ...
} *n1, *n2, *np;

SIMPLEQ_INIT(&head);                    /* Initialize the queue. */

n1 = malloc(sizeof(struct entry));      /* Insert at the head. */
SIMPLEQ_INSERT_HEAD(&head, n1, entries);

n1 = malloc(sizeof(struct entry));      /* Insert at the tail. */
SIMPLEQ_INSERT_TAIL(&head, n1, entries);

n2 = malloc(sizeof(struct entry));      /* Insert after. */
SIMPLEQ_INSERT_AFTER(&head, n1, n2, entries);
                                        /* Forward traversal. */
SIMPLEQ_FOREACH(np, &head, entries)
        np-> ...
                                        /* Delete. */
while (SIMPLEQ_FIRST(&head) != NULL)
        SIMPLEQ_REMOVE_HEAD(&head, entries);
if (SIMPLEQ_EMPTY(&head))               /* Test for emptiness. */
        printf("nothing to do\n");
```

Lists A list is headed by a structure defined by the `LIST_HEAD()` macro. This structure contains a single pointer to the first element on the list. The elements are doubly linked so that an arbitrary element can be removed without traversing the list. New elements can be added to the list after an existing element, before an existing element, or at the head of the list. A `LIST_HEAD` structure is declared as follows:

`LIST_HEAD(`*HEADNAME*`,` *TYPE*`)` *head*`;`

where *HEADNAME* is the name of the structure to be defined, and *TYPE* is the type of the elements to be linked into the list. A pointer to the head of the list can later be declared as:

`struct` *HEADNAME* **headp*`;`

The names *head* and *headp* are user selectable.

The macro `LIST_ENTRY()` declares a structure that connects the elements in the list.

The macro `LIST_HEAD_INITIALIZER()` provides a value which can be used to initialize a list head at compile time, and is used at the point that the list head variable is declared, like:

`struct` *HEADNAME* *head* `= LIST_HEAD_INITIALIZER(`*head*`);`

The macro `LIST_INIT()` initializes the list referenced by *head*.

The macro `LIST_INSERT_HEAD()` inserts the new element *elm* at the head of the list.

The macro `LIST_INSERT_AFTER()` inserts the new element *elm* after the element *listelm*.

The macro `LIST_INSERT_BEFORE()` inserts the new element *elm* before the element *listelm*.

The macro `LIST_REMOVE()` removes the element *elm* from the list.

The macro `LIST_EMPTY()` returns `true` if the list head has no elements.

The macro `LIST_FIRST()` returns the first element of the list head.

The macro `LIST_FOREACH()` traverses the list referenced by *head* in the forward direction, assigning each element in turn to *var*.

The macro `LIST_NEXT()` returns the element after the element *elm*.

List Example

```
LIST_HEAD(listhead, entry) head;
struct listhead *headp;         /* List head. */
struct entry {
        ...
        LIST_ENTRY(entry) entries;      /* List. */
        ...
} *n1, *n2, *np;

LIST_INIT(&head);                       /* Initialize the list. */

n1 = malloc(sizeof(struct entry));      /* Insert at the head. */
LIST_INSERT_HEAD(&head, n1, entries);

n2 = malloc(sizeof(struct entry));      /* Insert after. */
LIST_INSERT_AFTER(n1, n2, entries);

n2 = malloc(sizeof(struct entry));      /* Insert before. */
LIST_INSERT_BEFORE(n1, n2, entries);
                                        /* Forward traversal. */
LIST_FOREACH(np, &head, entries)
        np-> ...
                                        /* Delete. */
while (LIST_FIRST(&head) != NULL)
        LIST_REMOVE(LIST_FIRST(&head), entries);
if (LIST_EMPTY(&head))                  /* Test for emptiness. */
        printf("nothing to do\n");
```

Tail Queues

A tail queue is headed by a structure defined by the `TAILQ_HEAD()` macro. This structure contains a pair of pointers, one to the first element in the tail queue and the other to the last element in the tail queue. The elements are doubly linked so that an arbitrary element can be removed without traversing the tail queue. New elements can be added to the queue after an existing element, before an existing element, at the head of the queue, or at the end the queue. A `TAILQ_HEAD` structure is declared as follows:

TAILQ_HEAD(*HEADNAME*, *TYPE*) *head*;

where *HEADNAME* is the name of the structure to be defined, and *TYPE* is the type of the elements to be linked into the tail queue. A pointer to the head of the tail queue can later be declared as:

```
struct HEADNAME *headp;
```

The names *head* and *headp* are user selectable.

The macro TAILQ_ENTRY() declares a structure that connects the elements in the tail queue.

The macro TAILQ_HEAD_INITIALIZER() provides a value which can be used to initialize a tail queue head at compile time, and is used at the point that the tail queue head variable is declared, like:

struct *HEADNAME* *head* = TAILQ_HEAD_INITIALIZER(*head*);

The macro TAILQ_INIT() initializes the tail queue referenced by *head*.

The macro TAILQ_INSERT_HEAD() inserts the new element *elm* at the head of the tail queue.

The macro TAILQ_INSERT_TAIL() inserts the new element *elm* at the end of the tail queue.

The macro TAILQ_INSERT_AFTER() inserts the new element *elm* after the element *listelm*.

The macro TAILQ_INSERT_BEFORE() inserts the new element *elm* before the element *listelm*.

The macro TAILQ_REMOVE() removes the element *elm* from the tail queue.

The macro TAILQ_EMPTY() return true if the tail queue head has no elements.

The macro TAILQ_FIRST() returns the first element of the tail queue head.

The macro TAILQ_FOREACH() traverses the tail queue referenced by *head* in the forward direction, assigning each element in turn to *var*.

The macro TAILQ_FOREACH_REVERSE() traverses the tail queue referenced by *head* in the reverse direction, assigning each element in turn to *var*.

The macro TAILQ_NEXT() returns the element after the element *elm*.

The macro TAILQ_CONCAT() concatenates the tail queue headed by *head2* onto the end of the one headed by *head1* removing all entries from the former.

Tail Queue Example

```
TAILQ_HEAD(tailhead, entry) head;
struct tailhead *headp;         /* Tail queue head. */
struct entry {
        ...
        TAILQ_ENTRY(entry) entries;     /* Tail queue. */
        ...
```

```
} *n1, *n2, *np;

TAILQ_INIT(&head);                      /* Initialize the queue. */

n1 = malloc(sizeof(struct entry));      /* Insert at the head. */
TAILQ_INSERT_HEAD(&head, n1, entries);

n1 = malloc(sizeof(struct entry));      /* Insert at the tail. */
TAILQ_INSERT_TAIL(&head, n1, entries);

n2 = malloc(sizeof(struct entry));      /* Insert after. */
TAILQ_INSERT_AFTER(&head, n1, n2, entries);

n2 = malloc(sizeof(struct entry));      /* Insert before. */
TAILQ_INSERT_BEFORE(n1, n2, entries);
                                        /* Forward traversal. */
TAILQ_FOREACH(np, &head, entries)
        np-> ...
                                        /* Reverse traversal. */
TAILQ_FOREACH_REVERSE(np, &head, tailhead, entries)
        np-> ...
                                        /* Delete. */
while (TAILQ_FIRST(&head) != NULL)
        TAILQ_REMOVE(&head, TAILQ_FIRST(&head), entries);
if (TAILQ_EMPTY(&head))                 /* Test for emptiness. */
        printf("nothing to do\n");
```

Circular Queues

A circular queue is headed by a structure defined by the `CIRCLEQ_HEAD()` macro. This structure contains a pair of pointers, one to the first element in the circular queue and the other to the last element in the circular queue. The elements are doubly linked so that an arbitrary element can be removed without traversing the queue. New elements can be added to the queue after an existing element, before an existing element, at the head of the queue, or at the end of the queue. A `CIRCLEQ_HEAD` structure is declared as follows:

`CIRCLEQ_HEAD(`*HEADNAME*`,` *TYPE*`)` *head*`;`

where *HEADNAME* is the name of the structure to be defined, and *TYPE* is the type of the elements to be linked into the circular queue. A pointer to the head of the circular queue can later be declared as:

`struct` *HEADNAME* **headp*`;`

The names *head* and *headp* are user selectable.

The macro `CIRCLEQ_ENTRY()` declares a structure that connects the elements in the circular queue.

The macro `CIRCLEQ_HEAD_INITIALIZER()` provides a value which can be used to initialize a circular queue head at compile time, and is used at the point that the circular queue head variable is declared, like:

```
struct HEADNAME() head() = CIRCLEQ_HEAD_INITIALIZER(head());
```

The macro `CIRCLEQ_INIT()` initializes the circular queue referenced by *head*.

The macro `CIRCLEQ_INSERT_HEAD()` inserts the new element *elm* at the head of the circular queue.

The macro `CIRCLEQ_INSERT_TAIL()` inserts the new element *elm* at the end of the circular queue.

The macro `CIRCLEQ_INSERT_AFTER()` inserts the new element *elm* after the element *listelm*.

The macro `CIRCLEQ_INSERT_BEFORE()` inserts the new element *elm* before the element *listelm*.

The macro `CIRCLEQ_REMOVE()` removes the element *elm* from the circular queue.

The macro `CIRCLEQ_EMPTY()` return true if the circular queue head has no elements.

The macro `CIRCLEQ_FIRST()` returns the first element of the circular queue head.

The macro `CIRCLEQ_FOREACH()` traverses the circle queue referenced by head in the forward direction, assigning each element in turn to *var*. Each element is assigned exactly once.

The macro `CIRCLEQ_FOREACH_REVERSE()` traverses the circle queue referenced by head in the reverse direction, assigning each element in turn to *var*. Each element is assigned exactly once.

The macro `CIRCLEQ_LAST()` returns the last element of the circular queue head.

The macro `CIRCLEQ_NEXT()` returns the element after the element *elm*.

The macro `CIRCLEQ_PREV()` returns the element before the element *elm*.

The macro `CIRCLEQ_LOOP_NEXT()` returns the element after the element *elm*. If elm was the last element in the queue, the first element is returned.

The macro `CIRCLEQ_LOOP_PREV()` returns the element before the element *elm*. If elm was the first element in the queue, the last element is returned.

Circular Queue Example

```
CIRCLEQ_HEAD(circleq, entry) head;
struct circleq *headp;                  /* Circular queue head. */
struct entry {
        ...
        CIRCLEQ_ENTRY(entry) entries;   /* Circular queue. */
        ...
} *n1, *n2, *np;

CIRCLEQ_INIT(&head);                    /* Initialize circular queue. */
```

```
n1 = malloc(sizeof(struct entry));      /* Insert at the head. */
CIRCLEQ_INSERT_HEAD(&head, n1, entries);

n1 = malloc(sizeof(struct entry));      /* Insert at the tail. */
CIRCLEQ_INSERT_TAIL(&head, n1, entries);

n2 = malloc(sizeof(struct entry));      /* Insert after. */
CIRCLEQ_INSERT_AFTER(&head, n1, n2, entries);

n2 = malloc(sizeof(struct entry));      /* Insert before. */
CIRCLEQ_INSERT_BEFORE(&head, n1, n2, entries);
                                        /* Forward traversal. */
CIRCLEQ_FOREACH(np, &head, entries)
        np-> ...
                                        /* Reverse traversal. */
CIRCLEQ_FOREACH_REVERSE(np, &head, entries)
        np-> ...
                                        /* Delete. */
while (CIRCLEQ_FIRST(&head) != (void *)&head)
        CIRCLEQ_REMOVE(&head, CIRCLEQ_FIRST(&head), entries);
if (CIRCLEQ_EMPTY(&head))               /* Test for emptiness. */
        printf("nothing to do\n");
```

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

See Also attributes(5), queue(9F)

Notes Some of these macros or functions perform no error checking, and invalid usage leads to undefined behavior. In the case of macros or functions that expect their arguments to be elements that are present in the list or queue, passing an element that is not present is invalid.

The queue functions first appeared in 4.4BSD. The SIMPLEQ functions first appeared in NetBSD 1.2. The SLIST and STAILQ functions first appeared in FreeBSD 2.1.5. The CIRCLEQ_LOOP functions first appeared in NetBSD 4.0.

Name read_vtoc, write_vtoc – read and write a disk's VTOC

Synopsis
```
cc [ flag ... ] file ... -ladm [ library ... ]
#include <sys/vtoc.h>

int read_vtoc(int fd, struct vtoc *vtoc);

int write_vtoc(int fd, struct vtoc *vtoc);

int read_extvtoc(int fd, struct extvtoc *extvtoc);

int write_extvtoc(int fd, struct extvtoc *extvtoc);
```

Description The `read_vtoc()` and `read_extvtoc()` functions return the VTOC (volume table of contents) structure that is stored on the disk associated with the open file descriptor *fd*. On disks larger than 1 TB `read_extvtoc()` must be used.

The `write_vtoc()` and `write_extvtoc()` function stores the VTOC structure on the disk associated with the open file descriptor *fd*. On disks larger then 1TB `write_extvtoc()` function must be used.

The *fd* argument refers to any slice on a raw disk.

Return Values Upon successful completion, `read_vtoc()` and `read_extvtoc()` return a positive integer indicating the slice index associated with the open file descriptor. Otherwise, they return a negative integer indicating one of the following errors:

`VT_EIO` An I/O error occurred.

`VT_ENOTSUP` This operation is not supported on this disk.

`VT_ERROR` An unknown error occurred.

`VT_OVERFLOW` The caller attempted an operation that is illegal on the disk and may overflow the fields in the data structure.

Upon successful completion, `write_vtoc()` and `write_extvtoc()` return 0. Otherwise, they return a negative integer indicating one of the following errors:

`VT_EINVAL` The VTOC contains an incorrect field.

`VT_EIO` An I/O error occurred.

`VT_ENOTSUP` This operation is not supported on this disk.

`VT_ERROR` An unknown error occurred.

`VT_OVERFLOW` The caller attempted an operation that is illegal on the disk and may overflow the fields in the data structure.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also fmthard(1M), format(1M), prtvtoc(1M), ioctl(2), efi_alloc_and_init(3EXT), attributes(5), dkio(7I)

Bugs The write_vtoc() function cannot write a VTOC on an unlabeled disk. Use format(1M) for this purpose.

Name rtld_audit, la_activity, la_i86_pltenter, la_objsearch, la_objopen, la_objfilter, la_pltexit, la_pltexit64, la_preinit, la_sparcv8_pltenter, la_sparcv9_pltenter, la_amd64_pltenter, la_symbind32, la_symbind64, la_version – runtime linker auditing functions

Synopsis

```
void la_activity(uintptr_t *cookie, uint_t flag);

uintptr_t la_i86_pltenter(Elf32_Sym *sym, uint_t ndx, uintptr_t *refcook,
     uintptr_t *defcook, La_i86_regs *regs, uint_t *flags);

char *la_objsearch(const char *name, uintptr_t *cookie, uint_t flag);

uint_t la_objopen(Link_map  *lmp, Lmid_t lmid, uintptr_t *cookie);

int la_objfilter(uintptr_t  *fltrcook, uintptr_t *fltecook,
     uint_t *flags);

uintptr_t la_pltexit(Elf32_Sym *sym, uint_t ndx, uintptr_t *refcook,
     uintptr_t *defcook, uintptr_t retval);

uintptr_t la_pltexit64(Elf64_Sym *sym, uint_t ndx, uintptr_t *refcook,
     uintptr_t *defcook, uintptr_t retval, const char *sym_name);

void la_preinit(uintptr_t *cookie);

uintptr_t la_sparcv8_pltenter(Elf32_Sym *sym, uint_t ndx,
     uintptr_t *refcook, uintptr_t *defcook, La_amd64_regs *regs,
     uint_t *flags);

uintptr_t la_sparcv9_pltenter(Elf64_Sym *sym, uint_t ndx,
     uintptr_t *refcook, uintptr_t *defcook, La_sparcv8_regs *regs,
     uint_t *flags, const char *sym_name);

uintptr_t la_amd64_pltenter(Elf32_Sym *sym, uint_t ndx,
     uintptr_t *refcook, uintptr_t *defcook, La_sparcv8_regs *regs,
     uint_t *flags, const char *sym_name);

uintptr_t la_symbind32(Elf32_Sym *sym, uint_t ndx, uintptr_t *refcook,
     uintptr_t *defcook, uint_t *flags);

uintptr_t la_symbind64(Elf64_Sym *sym, uint_t ndx,
     uintptr_t *refcook,uintptr_t *defcook, uint_t *flags,
     const char *sym_name);

uint_t la_version(uint_t version);
```

Description A runtime linker auditing library is a user-created shared object offering one or more of these interfaces. The runtime linker `ld.so.1(1)`, calls these interfaces during process execution. See the *Linker and Libraries Guide* for a full description of the link auditing mechanism.

See Also `ld.so.1(1)`

Linker and Libraries Guide

Name rtld_db, rd_delete, rd_errstr, rd_event_addr, rd_event_enable, rd_event_getmsg, rd_init, rd_loadobj_iter, rd_log, rd_new, rd_objpad_enable, rd_plt_resolution, rd_reset – runtime linker debugging functions

Synopsis

```
cc [ flag ... ] file ... -lrtld_db [ library ... ]
#include <proc_service.h>
#include <rtld_db.h>

void rd_delete(struct rd_agent *rdap);

char *rd_errstr(rd_err_e rderr);

rd_err_e rd_event_addr(rd_agent *rdap, rd_notify_t *notify);

rd_err_e rd_event_enable(struct rd_agent *rdap, int onoff);

rd_err_e rd_event_getmsg(struct rd_agent *rdap,
     rd_event_msg_t *msg);

rd_err_e rd_init(int version);

typedef int rl_iter_f(const rd_loadobj_t *, void *);

rd_err_e rd_loadobj_iter(rd_agent_t *rap, rl_iter_f *cb,
     void *clnt_data);

void rd_log(const int onoff);

rd_agent_t *rd_new(struct ps_prochandle *php);

rd_err_e rd_objpad_enable(struct rd_agent *rdap, size_t padsize);

rd_err_e rd_plt_resolution(rd_agent *rdap, paddr_t pc,
     lwpid_t lwpid, paddr_t plt_base, rd_plt_info_t *rpi);

rd_err_e rd_reset(struct rd_agent *rdap);
```

Description The `librtld_db` library provides support for monitoring and manipulating runtime linking aspects of a program. There are at least two processes involved, the controlling process and one or more target processes. The controlling process is the `librtld_db` client that links with `librtld_db` and uses `librtld_db` to inspect or modify runtime linking aspects of one or more target processes. See the *Linker and Libraries Guide* for a full description of the runtime linker debugger interface mechanism.

Usage To use `librtld_db`, applications need to implement the interfaces documented in `ps_pread(3PROC)` and `proc_service(3PROC)`.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also ld.so.1(1), libc_db(3LIB), librtld_db(3LIB), proc_service(3PROC), ps_pread(3PROC), attributes(5)

Linker and Libraries Guide

Name sendfile – send files over sockets or copy files to files

Synopsis

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

ssize_t sendfile(int out_fd, int in_fd, off_t *off, size_t len);
```

Description The `sendfile()` function copies data from *in_fd* to *out_fd* starting at offset *off* and of length *len* bytes. The *in_fd* argument should be a file descriptor to a regular file opened for reading. See `open(2)`. The *out_fd* argument should be a file descriptor to a regular file opened for writing or to a connected `AF_INET` or `AF_INET6` socket of `SOCK_STREAM` type. See `socket(3SOCKET)`. The *off* argument is a pointer to a variable holding the input file pointer position from which the data will be read. After `sendfile()` has completed, the variable will be set to the offset of the byte following the last byte that was read. The `sendfile()` function does not modify the current file pointer of *in_fd*, but does modify the file pointer for *out_fd* if it is a regular file.

The `sendfile()` function can also be used to send buffers by pointing *in_fd* to `SFV_FD_SELF`.

Return Values Upon successful completion, `sendfile()` returns the total number of bytes written to *out_fd* and also updates the offset to point to the byte that follows the last byte read. Otherwise, it returns –1, and `errno` is set to indicate the error.

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

`EAFNOSUPPORT` The implementation does not support the specified address family for socket.

`EAGAIN` Mandatory file or record locking is set on either the file descriptor or output file descriptor if it points at regular files. `O_NDELAY` or `O_NONBLOCK` is set, and there is a blocking record lock. An attempt has been made to write to a stream that cannot accept data with the `O_NDELAY` or the `O_NONBLOCK` flag set.

`EBADF` The *out_fd* or *in_fd* argument is either not a valid file descriptor, *out_fd* is not opened for writing. or *in_fd* is not opened for reading.

`EINVAL` The offset cannot be represented by the `off_t` structure, or the length is negative when cast to `ssize_t`.

`EIO` An I/O error occurred while accessing the file system.

`ENOMEM` There is insufficient memory available. The offset parameter is updated by the amount of data transferred so that the call may be retried.

`ENOTCONN` The socket is not connected.

`EOPNOTSUPP` The socket type is not supported.

`EPIPE` The *out_fd* argument is no longer connected to the peer endpoint.

EINTR — A signal was caught during the write operation and no data was transferred.

Usage

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

Examples

EXAMPLE 1 Sending a Buffer Over a Socket

The following example demonstrates how to send the buffer *buf* over a socket. At the end, it prints the number of bytes transferred over the socket from the buffer. It assumes that *addr* will be filled up appropriately, depending upon where to send the buffer.

```
int tfd;
off_t baddr;
struct sockaddr_in sin;
char buf[64 * 1024];
in_addr_t addr;
size_t len;

tfd = socket(AF_INET, SOCK_STREAM, 0);
if (tfd == -1) {
    perror("socket");
    exit(1);
}

sin.sin_family = AF_INET;
sin.sin_addr.s_addr = addr;    /* Fill in the  appropriate address. */
sin.sin_port = htons(2345);
if (connect(tfd, (struct sockaddr *)&sin, sizeof(sin))<0) {
    perror("connect");
    exit(1);
}

baddr = (off_t)buf;
len = sizeof(buf);
while (len > 0) {
    ssize_t res;
    res = sendfile(tfd, SFV_FD_SELF, &baddr, len);
    if (res == -1)
            if (errno != EINTR) {
                    perror("sendfile");
                    exit(1);
            } else continue;
    len -= res;
}
```

EXAMPLE 2 Transferring Files to Sockets

The following program demonstrates a transfer of files to sockets:

EXAMPLE 2 Transferring Files to Sockets *(Continued)*

```
int ffd, tfd;
off_t off;
struct sockaddr_in sin;
in_addr_t  addr;
int len;
struct stat stat_buf;
ssize_t len;

ffd = open("file", O_RDONLY);
if (ffd == -1) {
    perror("open");
    exit(1);
}

tfd = socket(AF_INET, SOCK_STREAM, 0);
if (tfd == -1) {
    perror("socket");
    exit(1);
}

sin.sin_family = AF_INET;
sin.sin_addr = addr;    /* Fill in the  appropriate address. */
sin.sin_port = htons(2345);
if (connect(tfd, (struct sockaddr *) &sin, sizeof(sin)) <0) {
    perror("connect");
    exit(1);
}

if (fstat(ffd, &stat_buf) == -1) {
    perror("fstat");
    exit(1);
}

len = stat_buf.st_size;
while (len > 0) {
    ssize_t res;
    res = sendfile(tfd, ffd, &off, len);
    if (res == -1)
            if (errno != EINTR) {
                    perror("sendfile");
                    exit(1);
            } else continue;
    len -= res;
}
```

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

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

See Also open(2), libsendfile(3LIB), sendfilev(3EXT), socket(3SOCKET), attributes(5), lf64(5)

Name sendfilev – send a file

Synopsis

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

ssize_t sendfilev(int fildes, const struct sendfilevec *vec,
     int sfvcnt, size_t *xferred);
```

Parameters The `sendfilev()` function supports the following parameters:

fildes A file descriptor to a regular file or to a `AF_NCA`, `AF_INET`, or `AF_INET6` family type `SOCK_STREAM` socket that is open for writing. For `AF_NCA`, the protocol type should be zero.

vec An array of `SENDFILEVEC_T`, as defined in the `sendfilevec` structure above.

sfvcnt The number of members in *vec*.

xferred The total number of bytes written to `out_fd`.

Description The `sendfilev()` function attempts to write data from the *sfvcnt* buffers specified by the members of *vec* array: `vec[0], vec[1], ... , vec[sfvcnt−1]`. The *fildes* argument is a file descriptor to a regular file or to an `AF_NCA`, `AF_INET`, or `AF_INET6` family type `SOCK_STREAM` socket that is open for writing.

This function is analogous to `writev(2)`, but can read from both buffers and file descriptors. Unlike `writev()`, in the case of multiple writers to a file the effect of `sendfilev()` is not necessarily atomic; the writes may be interleaved. Application-specific synchronization methods must be employed if this causes problems.

The following is the `sendfilevec` structure:

```
typedef struct sendfilevec {
        int     sfv_fd;         /* input fd */
        uint_t  sfv_flag;       /* Flags. see below */
        off_t   sfv_off;        /* offset to start reading from */
        size_t  sfv_len;        /* amount of data */
} sendfilevec_t;

#define SFV_FD_SELF     (-2)
```

To send a file, open the file for reading and point `sfv_fd` to the file descriptor returned as a result. See `open(2)`. `sfv_off` should contain the offset within the file. `sfv_len` should have the length of the file to be transferred.

The *xferred* argument is updated to record the total number of bytes written to `out_fd`.

The `sfv_flag` field is reserved and should be set to zero.

To send data directly from the address space of the process, set `sfv_fd` to `SFV_FD_SELF`. `sfv_off` should point to the data, with `sfv_len` containing the length of the buffer.

Return Values

Upon successful completion, the `sendfilev()` function returns total number of bytes written to `out_fd`. Otherwise, it returns `-1`, and `errno` is set to indicate the error. The *xferred* argument contains the amount of data successfuly transferred, which can be used to discover the error vector.

Errors

`EACCES` — The process does not have appropriate privileges or one of the files pointed by `sfv_fd` does not have appropriate permissions.

`EAFNOSUPPORT` — The implementation does not support the specified address family for socket.

`EAGAIN` — Mandatory file or record locking is set on either the file descriptor or output file descriptor if it points at regular files. `O_NDELAY` or `O_NONBLOCK` is set, and there is a blocking record lock. An attempt has been made to write to a stream that cannot accept data with the `O_NDELAY` or the `O_NONBLOCK` flag set.

`EBADF` — The *fildes* argument is not a valid descriptor open for writing or an `sfv_fd` is invalid or not open for reading.

`EFAULT` — The *vec* argument points to an illegal address.

The *xferred* argument points to an illegal address.

`EINTR` — A signal was caught during the write operation and no data was transferred.

`EINVAL` — The *sfvcnt* argument was less than or equal to `0`. One of the `sfv_len` values in *vec* array was less than or equal to `0`, or greater than the file size. An `sfv_fd` is not seekable.

Fewer bytes were transferred than were requested.

`EIO` — An I/O error occurred while accessing the file system.

`ENOMEM` — There is insufficient memory available. The offset parameter is updated by the amount of data transferred so that the call may be retried.

`EPIPE` — The *fildes* argument is a socket that has been shut down for writing.

`EPROTOTYPE` — The socket type is not supported.

Usage

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

Examples

The following example sends 2 vectors, one of HEADER data and a file of length 100 over `sockfd`. `sockfd` is in a connected state, that is, `socket()`, `accept()`, and `bind()` operation are complete.

```
#include <sys/sendfile.h>
.
.
.
int
main (int argc, char *argv[]){
  int sockfd;
  ssize_t ret;
  size_t xfer;
  struct sendfilevec vec[2];
    .
    .
    .
  vec[0].sfv_fd = SFV_FD_SELF;
  vec[0].sfv_flag = 0;
  vec[0].sfv_off = "HEADER_DATA";
  vec[0].sfv_len = strlen("HEADER_DATA");
  vec[1].sfv_fd = open("input_file",.... );
  vec[1].sfv_flag = 0;
  vec[1].sfv_off = 0;
  vec[1].sfv_len = 100;

  ret = sendfilev(sockfd, vec, 2, &xfer);
.
.
.
}
```

Attributes

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

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

See Also

open(2), writev(2), libsendfile(3LIB), sendfile(3EXT), socket(3SOCKET), attributes(5)

Name sha1, SHA1Init, SHA1Update, SHA1Final – SHA1 digest functions

Synopsis

```
cc [ flag ... ] file ... -lmd [ library ... ]
#include <sha1.h>

void SHA1Init(SHA1_CTX *context);

void SHA1Update(SHA1_CTX *context, unsigned char *input,
     unsigned int inlen);

void SHA1Final(unsigned char *output, SHA1_CTX *context);
```

Description The SHA1 functions implement the SHA1 message-digest algorithm. The algorithm takes as input a message of arbitrary length and produces a 160-bit "fingerprint" or "message digest" as output. The SHA1 message-digest algorithm is intended for digital signature applications in which large files are "compressed" in a secure manner before being encrypted with a private (secret) key under a public-key cryptosystem such as RSA.

`SHA1Init()`, `SHA1Update()`, `SHA1Final()`

The `SHA1Init()`, `SHA1Update()`, and `SHA1Final()` functions allow a SHA1 digest to be computed over multiple message blocks. Between blocks, the state of the SHA1 computation is held in an SHA1 context structure allocated by the caller. A complete digest computation consists of calls to SHA1 functions in the following order: one call to `SHA1Init()`, one or more calls to `SHA1Update()`, and one call to `SHA1Final()`.

The `SHA1Init()` function initializes the SHA1 context structure pointed to by *context*.

The `SHA1Update()` function computes a partial SHA1 digest on the *inlen*-byte message block pointed to by *input*, and updates the SHA1 context structure pointed to by *context* accordingly.

The `SHA1Final()` function generates the final SHA1 digest, using the SHA1 context structure pointed to by *context*. The 16-bit SHA1 digest is written to output. After a call to `SHA1Final()`, the state of the context structure is undefined. It must be reinitialized with `SHA1Init()` before it can be used again.

Security The SHA1 algorithm is also believed to have some weaknesses. Migration to one of the SHA2 algorithms–including SHA224, SHA256, SHA386 or SHA512–is highly recommended when compatibility with data formats and on wire protocols is permitted.

Return Values These functions do not return a value.

Examples **EXAMPLE 1** Authenticate a message found in multiple buffers

The following is a sample function that authenticates a message found in multiple buffers. The calling function provides an authentication buffer to contain the result of the SHA1 digest.

```
#include <sys/types.h>
#include <sys/uio.h>
#include <sha1.h>

int
AuthenticateMsg(unsigned char *auth_buffer, struct iovec
                *messageIov, unsigned int num_buffers)
{
    SHA1_CTX sha1_context;
    unsigned int i;

    SHA1Init(&sha1_context);

    for(i=0; i<num_buffers; i++)
    {
         SHA1Update(&sha1_context, messageIov->iov_base,
                   messageIov->iov_len);
         messageIov += sizeof(struct iovec);
    }

    SHA1Final(auth_buffer, &sha1_context);

    return 0;
}
```

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

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

See Also sha2(3EXT), libmd(3LIB)

RFC 1374

Name sha2, SHA2Init, SHA2Update, SHA2Final, SHA224Init, SHA224Update, SHA224Final, SHA256Init, SHA256Update, SHA256Final, SHA384Init, SHA384Update, SHA384Final, SHA512Init, SHA512Update, SHA512Final – SHA2 digest functions

Synopsis

```
cc [ flag ... ] file ... -lmd [ library ... ]
#include <sha2.h>

void SHA2Init(uint64_t mech, SHA2_CTX *context);

void SHA2Update(SHA2_CTX *context, unsigned char *input,
     unsigned int inlen);

void SHA2Final(unsigned char *output, SHA2_CTX *context);

void SHA224Init(SHA224_CTX *context);

void SHA224Update(SHA224_CTX *context, unsigned char *input,
     unsigned int inlen);

void SHA224Final(unsigned char *output, SHA224_CTX *context);

void SHA256Init(SHA256_CTX *context);

void SHA256Update(SHA256_CTX *context, unsigned char *input,
     unsigned int inlen);

void SHA256Final(unsigned char *output, SHA256_CTX *context);

void SHA384Init(SHA384_CTX *context);

void SHA384Update(SHA384_CTX *context, unsigned char *input,
     unsigned int inlen);

void SHA384Final(unsigned char *output, SHA384_CTX *context);

void SHA512Init(SHA512_CTX *context);

void SHA512Update(SHA512_CTX *context, unsigned char *input,
     unsigned int inlen);

void SHA512Final(unsigned char *output, SHA512_CTX *context);
```

Description The `SHA2Init()`, `SHA2Update()`, `SHA2Final()` functions implement the `SHA224`, `SHA256`, `SHA384` and `SHA512` message-digest algorithms. The algorithms take as input a message of arbitrary length and produces a 200-bit "fingerprint" or "message digest" as output. The `SHA2` message-digest algorithms are intended for digital signature applications in which large files are "compressed" in a secure manner before being encrypted with a private (secret) key under a public-key cryptosystem such as RSA.

`SHA2Init()`, `SHA2Update()`, `SHA2Final()`

The `SHA2Init()`, `SHA2Update()`, and `SHA2Final()` functions allow an `SHA2` digest to be computed over multiple message blocks. Between blocks, the state of the `SHA2` computation is held in an `SHA2` context structure allocated by the caller. A complete digest computation consists of calls to `SHA2` functions in the following order: one call to `SHA2Init()`, one or more calls to `SHA2Update()`, and one call to `SHA2Final()`.

The `SHA2Init()` function initializes the SHA2 context structure pointed to by *context*. The *mech* argument is one of `SHA224`, `SHA256`, `SHA512`, and `SHA384`.

The `SHA2Update()` function computes a partial `SHA2` digest on the *inlen*-byte message block pointed to by *input*, and updates the `SHA2` context structure pointed to by *context* accordingly.

The `SHA2Final()` function generates the final `SHA2Final` digest, using the `SHA2` context structure pointed to by *context*. The `SHA2` digest is written to output. After a call to `SHA2Final()`, the state of the context structure is undefined. It must be reinitialized with `SHA2Init()` before it can be used again.

`SHA224Init()`, `SHA224Update()`, `SHA224Final()`, `SHA256Init()`, `SHA256Update()`, `SHA256Final()`, `SHA384Init()`, `SHA384Update()`, `SHA384Final()`, `SHA512Init()`, `SHA512Update()`, `SHA512Final()`

Alternative APIs exist as named above. The `Update()` and `Final()` sets of functions operate exactly as the previously described `SHA2Update()` and `SHA2Final()` functions. The `SHA224Init()`, `SHA256Init()`, `SHA384Init()`, and `SHA512Init()` functions do not take the *mech* argument as it is implicit in the function names.

Return Values These functions do not return a value.

Examples **EXAMPLE 1** Authenticate a message found in multiple buffers

The following is a sample function that authenticates a message found in multiple buffers. The calling function provides an authentication buffer to contain the result of the `SHA2` digest.

```
#include <sys/types.h>
#include <sys/uio.h>
#include <sha2.h>

int
AuthenticateMsg(unsigned char *auth_buffer, struct iovec
                *messageIov, unsigned int num_buffers)
{
    SHA2_CTX sha2_context;
    unsigned int i;

    SHA2Init(SHA384, &sha2_context);

    for(i=0; i<num_buffers; i++)
    {
         SHA2Update(&sha2_context, messageIov->iov_base,
                    messageIov->iov_len);
         messageIov += sizeof(struct iovec);
    }

    SHA2Final(auth_buffer, &sha2_context);
```

EXAMPLE 1 Authenticate a message found in multiple buffers *(Continued)*

```
    return 0;
}
```

EXAMPLE 2 Authenticate a message found in multiple buffers

The following is a sample function that authenticates a message found in multiple buffers. The calling function provides an authentication buffer that will contain the result of the SHA384 digest, using alternative interfaces.

```
int
AuthenticateMsg(unsigned char *auth_buffer, struct iovec
                *messageIov, unsigned int num_buffers)
{
    SHA384_CTX ctx;
    unsigned int i;

    SHA384Init(&ctx);

    for(i=0, i<num_buffers; i++
    {
         SHA384Update(&ctx, messageIov->iov_base,
                   messageIov->iov_len);
         messageIov += sizeof(struct iovec);
    }

    SHA384Final(auth_buffer, &ctx);

    return 0;
}
```

Attributes

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

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

See Also

libmd(3LIB)

FIPS 180–2

Name stdarg – handle variable argument list

Synopsis

```
#include <stdarg.h>
va_list pvar;

void va_start(va_list pvar, void name);

(type *) va_arg(va_list pvar, type);

void va_copy(va_list dest, va_list src);

void va_end(va_list pvar);
```

Description This set of macros allows portable procedures that accept variable numbers of arguments of variable types to be written. Routines that have variable argument lists (such as `printf`) but do not use *stdarg* are inherently non-portable, as different machines use different argument-passing conventions.

`va_list` is a type defined for the variable used to traverse the list.

The `va_start` macro is invoked before any access to the unnamed arguments and initializes *pvar* for subsequent use by `va_arg()` and `va_end()`. The parameter *name* is the identifier of the rightmost parameter in the variable parameter list in the function definition (the one just before the `, ...`). If this parameter is declared with the `register` storage class or with a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions, the behavior is undefined.

The parameter *name* is required under strict ANSI C compilation. In other compilation modes, *name* need not be supplied and the second parameter to the `va_start()` macro can be left empty (for example, `va_start(pvar, );`). This allows for routines that contain no parameters before the `...` in the variable parameter list.

The `va_arg()` macro expands to an expression that has the type and value of the next argument in the call. The parameter `pvar` should have been previously initialized by `va_start()`. Each invocation of `va_arg()` modifies `pvar` so that the values of successive arguments are returned in turn. The parameter `type` is the type name of the next argument to be returned. The type name must be specified in such a way so that the type of a pointer to an object that has the specified type can be obtained simply by postfixing a * to *type*. If there is no actual next argument, or if *type* is not compatible with the type of the actual next argument (as promoted according to the default argument promotions), the behavior is undefined.

The `va_copy()` macro saves the state represented by the `va_list`*src* in the `va_list` *dest*. The `va_list` passed as *dest* should not be initialized by a previous call to `va_start()`, and must be passed to `va_end()` before being reused as a parameter to `va_start()` or as the *dest* parameter of a subsequent call to `va_copy()`. The behavior is undefined should any of these restrictions not be met.

The `va_end()` macro is used to clean up.

Multiple traversals, each bracketed by `va_start()` and `va_end()`, are possible.

Examples

EXAMPLE 1 A sample program.

This example gathers into an array a list of arguments that are pointers to strings (but not more than `MAXARGS` arguments) with function `f1`, then passes the array as a single argument to function `f2`. The number of pointers is specified by the first argument to `f1`.

```
#include <stdarg.h>
#define MAXARGS    31
void f1(int n_ptrs, ...)
{
     va_list ap;
     char *array[MAXARGS];
     int ptr_no = 0;

     if (n_ptrs > MAXARGS)
          n_ptrs = MAXARGS;
     va_start(ap, n_ptrs);
     while (ptr_no < n_ptrs)
          array[ptr_no++] = va_arg(ap, char*);
     va_end(ap);
     f2(n_ptrs, array);
}
```

Each call to `f1` shall have visible the definition of the function or a declaration such as

```
void f1(int, ...)
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
Standard	See `standards(5)`.

See Also

`vprintf(3C)`, `attributes(5)`, `standards(5)`

Notes

It is the responsibility of the calling routine to specify in some manner how many arguments there are, since it is not always possible to determine the number of arguments from the stack frame. For example, *execl* is passed a zero pointer to signal the end of the list. The *printf* function can determine the number of arguments by the format. It is non-portable to specify a second argument of `char`, `short`, or `float` to `va_arg()`, because arguments seen by the called function are not `char`, `short`, or `float`. C converts `char` and `short` arguments to `int` and converts `float` arguments to `double` before passing them to a function.

Name SUNW_C_GetMechSession, SUNW_C_KeyToObject – PKCS#11 Cryptographic Framework functions

Synopsis

```
cc [ flag ... ] file... -lpkcs11 [ library... ]
#include <security/cryptoki.h>
#include <security/pkcs11.h>

CK_RV SUNW_C_GetMechSession(CK_MECHANISM_TYPE mech,
     CK_SESSION_HANDLE_PTR hSession);

CK_RV SUNW_C_KeyToObject(CK_SESSION_HANDLE hSession,
     CK_MECHANISM_TYPE mech, const void *rawkey, size_t rawkey_len,
     CK_OBJECT_HANDLE_PTR obj);
```

Description These functions implement the RSA PKCS#11 v2.20 specification by using plug-ins to provide the slots.

The `SUNW_C_GetMechSession()` function initializes the PKCS#11 cryptographic framework and performs all necessary calls to Standard PKCS#11 functions (see `libpkcs11`(3LIB)) to create a session capable of providing operations on the requested mechanism. It is not neccessary to call `C_Initalize()` or `C_GetSlotList()` before the first call to `SUNW_C_GetMechSession()`.

If the `SUNW_C_GetMechSession()` function is called multiple times, it will return a new session each time without re-initalizing the framework. If it is unable to return a new session, `CKR_SESSION_COUNT` is returned.

The `C_CloseSession()` function should be called to release the session when it is no longer required.

The `SUNW_C_KeyToObject()` function creates a key object for the specified mechanism from the *rawkey* data. The object should be destroyed with `C_DestroyObject()` when it is no longer required.

Return Values The `SUNW_C_GetMechSession()` function returns the following values:

`CKR_OK`	The function completed successfully.
`CKR_SESSION_COUNT`	No sessions are available.
`CKR_ARGUMENTS_BAD`	A null pointer was passed for the return session handle.
`CKR_MECHANISM_INVALID`	The requested mechanism is invalid or no available plug-in provider supports it.
`CKR_FUNCTION_FAILED`	The function failed.
`CKR_GENERAL_ERROR`	A general error occurred.

The `SUNW_C_KeyToObject()` function returns the following values:

`CKR_OK`	The function completed successfully.

`CKR_ARGUMENTS_BAD` A null pointer was passed for the session handle or the key material.

`CKR_MECHANISM_INVALID` The requested mechanism is invalid or no available plug-in provider supports it.

`CKR_FUNCTION_FAILED` The function failed.

`CKR_GENERAL_ERROR` A general error occurred.

The return values of each of the implemented functions are defined and listed in the RSA PKCS#11 v2.20 specification. See http://www.rsasecurity.com.

Usage These functions are not part of the RSA PKCS#11 v2.20 specification. They are not likely to exist on non-Solaris systems. They are provided as a convenience to application programmers. Use of these functions will make the application non-portable to other systems.

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

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

See Also `libpkcs11`(3LIB), `attributes`(5)

http://www.rsasecurity.com

Name tsalarm_get, tsalarm_set – get or set alarm relays

Synopsis

```
cc [ flag... ] file... -ltsalarm [ library... ]
#include <tsalarm.h>

int tsalarm_get(uint32_t alarm_type, uint32_t *alarm_state);

int tsalarm_set(uint32_t alarm_type, uint32_t alarm_state);
```

Parameters

alarm_type

The alarm type whose state is retrieved or set. Valid settings are:

`TSALARM_CRITICAL`	critical
`TSALARM_MAJOR`	major
`TSALARM_MINOR`	minor
`TSALARM_USER`	user

alarm_state

The state of the alarm. Valid settings are:

`TSALARM_STATE_ON`	The alarm state needs to be changed to “on”, or is returned as “on”.
`TSALARM_STATE_OFF`	The alarm state needs to be changed to “off”, or is returned as “off”.
`TSALARM_STATE_UNKNOWN`	The alarm state is returned as unknown.

Description The TSALARM interface provides functions through which alarm relays can be controlled. The set of functions and data structures of this interface are defined in the `<tsalarm.h>` header.

There are four alarm relays that are controlled by ILOM. Each alarm can be set to “on” or “off” by using `tsalarm` interfaces provided from the host. The four alarms are labeled as `critical`, `major`, `minor`, and `user`. The user alarm is set by a user application depending on system condition. LEDs in front of the box provide a visual indication of the four alarms. The number of alarms and their meanings and labels can vary across platforms.

The `tsalarm_get()` function gets the state of *alarm_type* and returnsit in *alarm_state*. If successful, the function returns 0.

The `tsalarm_set()` function sets the state of *alarm_type* to the value in *alarm_state*. If successful, the function returns 0.

The following structures are defined in `<tsalarm.h>`:

```
typedef struct tsalarm_req {
     uint32_t     alarm_id;
```

```
     uint32_t     alarm_action;
} tsalarm_req_t;

typedef struct tsalarm_resp {
     uint32_t     status;
     uint32_t     alarm_id;
     uint32_t     alarm_state;
} tsalarm_resp_t;
```

Return Values

The `tsalarm_get()` and `tsalarm_set()` functions return the following values:

`TSALARM_CHANNEL_INIT_FAILURE`	Channel initialization failed.
`TSALARM_COMM_FAILURE`	Channel communication failed.
`TSALARM_NULL_REQ_DATA`	Allocating memory for request data failed.
`TSALARM_SUCCESS`	Successful completion.
`TSALARM_UNBOUND_PACKET_RECVD`	An incorrect packet was received.

The `tsalarm_get()` function returns the following value:

`TSALARM_GET_ERROR`	An error occurred while getting the alarm state.

The `tsalarm_set()` function returns the following value:

`TSALARM_SET_ERROR`	An error occurred while setting the alarm state.

Examples

EXAMPLE 1 Get and set an alarm state.

The following example demonstrates how to get and set an alarm state.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <tsalarm.h>

void help(char *name) {
   printf("Syntax:  %s [get <type> | set <type> <state>]\n\n", name);
   printf("         type  = { critical, major, minor, user }\n");
   printf("         state = { on, off }\n\n");

   exit(0);
}

int main(int argc, char **argv) {

   uint32_t alarm_type, alarm_state;

   if (argc < 3)
```

EXAMPLE 1 Get and set an alarm state. *(Continued)*

```
        help(argv[0]);

    if (strncmp(argv[2], "critical", 1) == 0)
        alarm_type = TSALARM_CRITICAL;
    else if (strncmp(argv[2], "major", 2) == 0)
        alarm_type = TSALARM_MAJOR;
    else if (strncmp(argv[2], "minor", 2) == 0)
        alarm_type = TSALARM_MINOR;
    else if (strncmp(argv[2], "user", 1) == 0)
        alarm_type = TSALARM_USER;
    else
        help(argv[0]);

    if (strncmp(argv[1], "get", 1) == 0) {
        tsalarm_get(alarm_type, &alarm_state);
        printf("alarm = %d\tstate = %d\n", alarm_type, alarm_state);
    }
    else if (strncmp(argv[1], "set", 1) == 0) {
        if (strncmp(argv[3], "on", 2) == 0)
            alarm_state = TSALARM_STATE_ON;
        else if (strncmp(argv[3], "off", 2) == 0)
            alarm_state = TSALARM_STATE_OFF;
        else
            help(argv[0]);

        tsalarm_set(alarm_type, alarm_state);
    }
    else {
        help(argv[0]);
    }

    return 0;
}
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Uncommitted
MT-Level	Safe

See Also

`libtsalarm(3LIB)`, `attributes(5)`

Name v12n, v12n_capabilities, v12n_domain_roles, v12n_domain_name, v12n_domain_uuid, v12n_ctrl_domain, v12n_chassis_serialno – return virtualization environment domain parameters

Synopsis

```
cc [ flag... ] file... -lv12n [ library... ]
#include <libv12n.h>

int v12n_capabilities();

int v12n_domain_roles();

int v12n_domain_uuid(uuid_t uuid);

size_t v12n_domain_name(char *buf, size_t buflen);

size_t v12n_ctrl_domain(char *buf, size_t buflen);

size_t v12n_chassis_serialno(char *buf, size_t buflen);
```

Description The `v12n_capabilities()` function returns the virtualization capabilities mask of the current domain. The virtualization capabilities bit mask consists of the following values:

`V12N_CAP_SUPPORTED`	Virtualization is supported on this domain.
`V12N_CAP_ENABLED`	Virtualization is enabled on this domain.
`V12N_CAP_IMPL_LDOMS`	Logical Domains is the supported virtualization implementation.

The `v12n_domain_roles()` function returns the virtualization domain role mask. The virtualization domain role mask consists of the following values:

`V12N_ROLE_CONTROL`	If the virtualization implementation is Logical Domains, and this bit is one, the current domain is a control domain. If this bit is zero, the current domain is a guest domain.
`V12N_ROLE_IO`	Current domain is an I/O domain.
`V12N_ROLE_SERVICE`	Current domain is a service domain.
`V12N_ROLE_ROOT`	Current domain is an root I/O domain.

The `v12n_domain_uuid()` function stores the universally unique identifier (UUID) for the current virtualization domain in the `uuid` argument. See the `libuuid`(3LIB) manual page.

The `v12n_domain_name()` function stores the name of the current virtualization domain in the location specified by *buf*. *buflen* specifies the size in bytes of the buffer. If the buffer is too small to hold the complete null-terminated name, the first *buflen* bytes of the name are stored in the buffer. A buffer of size `V12N_NAME_MAX` is sufficient to hold any domain name. If *buf* is NULL or *buflen* is 0, the name is not copied into the buffer.

The `v12n_ctrl_domain()` function stores the control domain or dom0 network node name of the current domain in the location specified by *buf*. Note that a domain's control domain is volatile during a domain migration. The information returned by this function might be stale if the domain was in the process of migrating. *buflen* specifies the size in bytes of the buffer. If the buffer is too small to hold the complete null-terminated name, the first *buflen* bytes of the name are stored in the buffer. A buffer of size `V12N_NAME_MAX` is sufficient to hold the control domain node name string. If *buf* is NULL or *buflen* is 0, the name is not copied into the buffer.

The `v12n_chassis_serialno()` function stores the chassis serial number of the platform on which the current domain is running in the location specified by *buf*. Note that the chassis serial number is volatile during a domain migration. The information returned by this function might be stale if the domain was in the process of migrating. *buflen* specifies the size in bytes of the buffer. If the buffer is too small to hold the complete null-terminated name, the first *buflen* bytes of the name are stored in the buffer. A buffer of size `V12N_NAME_MAX` is sufficient to hold any chassis serial number string. If *buf* is NULL or *buflen* is 0, the name is not copied into the buffer.

Return Values

On successful completion, the `v12n_capabilties()` and `v12n_domain_roles()` functions return a non-negative bit mask. Otherwise, the `v12n_domain_roles()` function returns -1 and sets `errno` to indicate the error.

On successful completion, the `v12n_domain_uuid()` function returns 0. Otherwise, the `v12n_domain_uuid()` function returns -1 and sets `errno` to indicate the error.

On successful completion, the `v12n_domain_name()`, `v12n_ctrl_domain()`, and `v12n_chassis_serialno()` functions return the buffer size required to hold the full non-terminated string. Otherwise, these functions return -1 and set `errno` to indicate the error.

Errors

The `v12n_domain_roles()` function fails with `EPERM` when the calling process has an ID other than the privileged user.

The `v12n_domain_name()` function will fail if:

`EPERM` The calling process has an ID other than the privileged user.

`ENOTSUP` Virtualization is not supported or enabled on this domain.

`EFAULT` *buf* points to an illegal address.

`ENOENT` The `sun4v` machine description is inaccessible or has no `uuid` node.

The `v12n_domain_uuid()` function will fail if:

`EPERM` The calling process has an ID other than the privileged user.

`ENOTSUP` Virtualization is not supported or enabled on this domain.

`EFAULT` *buf* points to an illegal address.

`ENOENT` The `sun4v` machine description is inaccessible or has no `uuid` node.

The `v12n_ctrl_domain()` function will fail if:

`EPERM` The calling process has an ID other than the privileged user.

`ENOTSUP` Virtualization is not supported or enabled on this domain.

`EFAULT` *buf* points to an illegal address.

`ETIME` The domain service on the control domain did not respond within the timeout value.

The `v12n_chassis_serialno()` function will fail if:

`EPERM` The calling process has an ID other than the privileged user.

`ENOTSUP` Virtualization is not supported or enabled on this domain.

`EFAULT` *buf* points to an illegal address.

`ETIME` The domain service on the control domain did not respond within the timeout value.

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

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

See Also `virtinfo(1M)`, `libuuid(3LIB)`, `libv12n(3LIB)`, `attributes(5)`

Name varargs – handle variable argument list

Synopsis

```
#include <varargs.h>
va_alist
va_dcl
va_list pvar;

void va_start(va_listpvar);

type va_arg(va_list pvar, type);

void va_end(va_list pvar);
```

Description This set of macros allows portable procedures that accept variable argument lists to be written. Routines that have variable argument lists (such as `printf`(3C)) but do not use `varargs` are inherently non-portable, as different machines use different argument-passing conventions.

`va_alist` is used as the parameter list in a function header.

`va_dcl` is a declaration for `va_alist`. No semicolon should follow `va_dcl`.

`va_list` is a type defined for the variable used to traverse the list.

`va_start` is called to initialize `pvar` to the beginning of the list.

`va_arg` will return the next argument in the list pointed to by `pvar`. `type` is the type the argument is expected to be. Different types can be mixed, but it is up to the routine to know what type of argument is expected, as it cannot be determined at runtime.

`va_end` is used to clean up.

Multiple traversals, each bracketed by `va_start` and `va_end`, are possible.

Examples **EXAMPLE 1** A sample program.

This example is a possible implementation of `execl` (see `exec`(2)).

```
#include <unistd.h>
#include <varargs.h>
#define MAXARGS    100
/*    execl is called by
        execl(file, arg1, arg2, ..., NULL);
*/
execl(va_alist)
va_dcl
{
    va_list ap;
    char *file;
    char *args[MAXARGS];        /* assumed big enough*/
    int argno = 0;
```

EXAMPLE 1 A sample program. *(Continued)*

```
    va_start(ap);
    file = va_arg(ap, char *);
    while ((args[argno++] = va_arg(ap, char *)) != 0)
        ;
    va_end(ap);
    return execv(file, args);
}
```

See Also

exec(2), printf(3C), vprintf(3C), stdarg(3EXT)

Notes

It is up to the calling routine to specify in some manner how many arguments there are, since it is not always possible to determine the number of arguments from the stack frame. For example, execl is passed a zero pointer to signal the end of the list. printf can tell how many arguments are there by the format.

It is non-portable to specify a second argument of char, short, or float to va_arg, since arguments seen by the called function are not char, short, or float. C converts char and short arguments to int and converts float arguments to double before passing them to a function.

stdarg is the preferred interface.