# Netra™ Data Plane Software Suite 1.1 Reference Manual

Sun Microsystems, Inc.
www.sun.com

Part No. 820-2375-10
July 2007, Revision A

Submit comments about this document at: http://www.sun.com/hwdocs/feedback

Please
Recycle

Adobe PostScript

# Contents

**Preface vii**

**1. User API 1**

Late-Binding API 1

Late-Binding API Data Types 2

Late-Binding API Macros 2

Late-Binding API Mutex Functions 2

Late-Binding API Queue Functions 5

Late-Binding API Memory Pool Functions 9

Late-Binding API Channel Functions 12

Late-Binding API Interruptible Wait 15

NPOS API 18

NPOS API Data Types 18

NPOS API Memory Management Functions 19

NPOS API Thread Functions 21

NPOS API Miscellaneous Functions 24

NPOS API Communication Functions 24

NPOS API Time Functions 40

Miscellaneous Functions 41

Finite State Automata API 42

Finite State Automata API Defines 43
Finite State Automata API Macros 43
FSM Example 46
C Library Support on Bare Hardware 48

**2. Configuration API 51**

Hardware Architecture API 51
Hardware Architecture API Data Types 52
Hardware Architecture API Functions 52
Software Architecture API 85
Software Architecture API Data Types 85
Software Architecture API Functions 86
Map API 100
Map API Data Types 101
Map API Functions 101
Error-Handling API 104
Error-Handling API Data Types 104
Error-Handling API Functions 104
Error Handler Function Prototype 105
CMT-Specific Hardware Architecture Constants 106
CMT-Specific Hardware Architecture Types 107
CMT-Specific Hardware Architecture Properties 107
CMT-Specific Software Architecture Constants 108
CMT-Specific Software Architecture Types 108
CMT-Specific Software Architecture Properties 109

**3. Teja Profiler API 111**

Teja Profiler API Configuration 111
Teja Profiler API 112

Teja Profiler API Data Types 112
Teja Profiler API Functions 112
CMT-Specific Profiler Constants 115
CMT-Specific Profiler Groups 115

# Preface

The *Netra Data Plane Software Suite 1.1 Reference Manual* provides detailed information about the various functions and parameters of the application programming interface (API). This document is highly technical, and written for developers who need to know the behavior of the software.

## How This Document Is Organized

Chapter 1 describes the components and functions of the User API.

Chapter 2 describes the components and functions of the Configuration API.

Chapter 3 describes the components and functions of the Teja Profiler API.

## Using UNIX Commands

This document might not contain information about basic UNIX® commands and procedures such as shutting down the system, booting the system, and configuring devices. Refer to the following for this information:

- Software documentation that you received with your system
- Solaris™ Operating System documentation, which is at:

  `http://docs.sun.com`

# Shell Prompts

| Shell | Prompt |
|---|---|
| C shell | *machine-name*% |
| C shell superuser | *machine-name*# |
| Bourne shell and Korn shell | $ |
| Bourne shell and Korn shell superuser | # |

# Typographic Conventions

| Typeface* | Meaning | Examples |
|---|---|---|
| AaBbCc123 | The names of commands, files, and directories; on-screen computer output | Edit your .login file.<br>Use ls -a to list all files.<br>% You have mail. |
| **AaBbCc123** | What you type, when contrasted with on-screen computer output | % **su**<br>Password: |
| *AaBbCc123* | Book titles, new words or terms, words to be emphasized. Replace command-line variables with real names or values. | Read Chapter 6 in the *User's Guide*.<br>These are called *class* options.<br>You *must* be superuser to do this.<br>To delete a file, type rm *filename*. |

* The settings on your browser might differ from these settings.

# Related Documentation

The documents listed as online are available at:

http://www.sun.com

| Application | Title | Part Number | Format | Location |
|---|---|---|---|---|
| Operation | *Netra Data Plane Software Suite 1.1 User's Guide* | 820-2374-10 | PDF | online |
| Reference | *Netra Data Plane Software Suite 1.1 Reference Manual* | 820-2375-10 | PDF | online |
| Last-minute information | *Netra Data Plane Software Suite 1.1 Release Notes* | 820-2376-10 | PDF | online |
| Documentation Location | *Netra Data Plane Software Suite 1.11.1 Getting Started Guide* | 820-2377-10 | PDF | online |

# Documentation, Support, and Training

| Sun Function | URL |
|---|---|
| Documentation | http://www.sun.com/documentation/ |
| Support | http://www.sun.com/support/ |
| Training | http://www.sun.com/training/ |

# Third-Party Web Sites

Sun is not responsible for the availability of third-party web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Sun is not responsible or liable for any actual or alleged damage or loss caused by or in connection with the use of or reliance on any such content, goods, or services that are available on or through such sites or resources.

# Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. You can submit your comments by going to:

http://www.sun.com/hwdocs/feedback

Please include the title and part number of your document with your feedback:

*Netra Data Plane Software Suite 1.1 Reference Manual*, part number 820-2375-10

CHAPTER 1

# User API

The user application programming interface (API) consists of functions you can deploy in the application code. This API consists of three main parts:


- "Late-Binding API" on page 1
- "NPOS API" on page 18
- "Finite State Automata API" on page 42


Additional information is provided in:


- "C Library Support on Bare Hardware" on page 48


## Late-Binding API

The Late-Binding API provides primitives for the synchronization of distributed threads, communication, and memory allocation. This API is treated specially by the `tejacc` compiler, and is generated dynamically based on contextual information. See the *Netra Data Plane Software Suite 1.1 User's Guide* for an overview of this API.

# Late-Binding API Data Types

**TABLE 1-1** Late-Binding API Data Types

| Data Type | Description |
|---|---|
| `teja_channel_t` | Channel data type |
| `teja_memory_pool_t` | Memory pool data type |
| `teja_mutex_t` | Mutex data type |
| `teja_queue_t` | Queue data type |
| `teja_thread_t` | Thread data type |

# Late-Binding API Macros

**TABLE 1-2** Late-Binding API Macros

| Macros | Description |
|---|---|
| `TEJA_INFINITE_WAIT` | Used to indicate an infinite timeout in `teja_wait()`. |
| `TEJA_IS_RAW_OS` | Defined only on bare hardware systems. Such systems support a subset of the Teja API. |
| `TEJA_NO_EVENT` | Used when sending data on a channel to indicate that event logic can be skipped. |

# Late-Binding API Mutex Functions

## `teja_mutex_lock`

### Function

```
int teja_mutex_lock(teja_mutex_t mutex);
```

### Parameters

*mutex* – Mutex to lock.

### Return Values

`int` – If successful, this function returns `0`. In case of an error, this function returns `-1`.

### Description

Acquires a mutual exclusion lock. If the mutex is already locked, this function does not return until the mutex becomes available and the lock is acquired for the calling thread. Once the lock is held by the thread, what occurs if this function is called a second time is undefined. If an error is returned, the caller can assume the lock was not acquired.

### Example

```
if (teja_mutex_lock (mutex) < 0)
  {
    printf ("Error locking mutex\n");
  }
else
  {
    printf ("Entered critical region\n");
    /* Wait one second */
    teja_wait_time (1, 0);
    printf ("Exiting critical region\n");
    if (teja_mutex_unlock (mutex) < 0)
      {
        printf ("Error unlocking mutex\n");
      }
  }
```

## teja_mutex_trylock

### Function

`int teja_mutex_trylock(teja_mutex_t` *mutex*`);`

### Parameters

*mutex* – Mutex to lock.

### Return Values

`int` – If successful, this function returns `0`. In case of an error, this function returns `-1`.

### Description

Attempts to lock the given mutex without blocking. If the mutex is already locked, this function exits immediately returning `-1`, otherwise the function locks the mutex and returns `0`. Once the lock is held by the thread, what occurs if this function is called a second time is undefined.

### Example

```
if (teja_mutex_trylock (mutex) < 0)
  {
    printf ("Trylock on mutex failed\n");
  }
else
  {
    printf ("Entered critical region\n");
    /* Wait one second */
    teja_wait_time (1, 0);
    printf ("Exiting critical region\n");
    if (teja_mutex_unlock (mutex) < 0)
      {
        printf ("Error unlocking mutex\n");
      }
  }
```

## teja_mutex_unlock

### Function

`int teja_mutex_unlock(teja_mutex_t` *mutex*`);`

### Parameters

*mutex* – Mutex to unlock.

### Return Values

`int` – If successful, this function returns `0`. In case of an error, this function returns `-1`.

### Description

Unlocks the given mutex. If the mutex was not locked by the current thread the result is undefined. Avoid such behavior.

### Example

See the examples in "teja_mutex_lock" on page 2 and "teja_mutex_trylock" on page 3.

## Late-Binding API Queue Functions

The first word of the node that is enqueued is permitted to be overwritten by the queue implementation.

### teja_queue_enqueue

#### Function

```
int teja_queue_enqueue(teja_queue_t queue, void * node);
```

#### Parameters

*queue* – Queue to enqueue to.

*node* – Pointer to node to enqueue.

#### Return Values

`int` – If successful, this function returns `0`. In case of an error, this function returns `-1`.

#### Description

Enqueues a node into a queue. The queue implementation is permitted to overwrite the first word of the node. If `-1` is returned, the queue might be full or some other error has occurred.

### Example

```
void * node;
node = teja_malloc (16);
if (node)
  {
    if (teja_queue_enqueue (queue, node) < 0)
      {
        printf ("Error while attempting to enqueue a node");
      }
  }
```

## teja_queue_dequeue

### Function

`void * teja_queue_dequeue(teja_queue_t` *queue*`);`

### Parameters

*queue* – Queue to dequeue from.

### Return Values

`void *` – `NULL` if the queue was empty or pointer to the dequeued node otherwise.

### Description

Dequeues a pointer to a node from the queue. The first word of the returned node might have been overwritten by the queue implementation.

### Example

```
void * node;
node = teja_queue_dequeue (queue);
if (node)
  {
    printf ("Dequeued node %x\n", node);
  }
else
  {
    printf ("Queue was empty\n");
  }
```

## teja_queue_is_empty

### Function

`int teja_queue_is_empty(teja_queue_t` *queue*`);`

### Parameters

*queue* – Queue to test.

### Return Values

`int` – `0` if the queue is not empty, `1` if the queue is empty.

### Description

Tests to see if the queue is empty.

### Example

```
if (teja_queue_is_empty (queue))
  {
    printf ("Queue is empty\n");
  }
else
  {
    printf ("Queue is not empty\n");
  }
```

## teja_queue_get_size

### Function

`int teja_queue_get_size(teja_queue_t` *queue*`);`

### Parameters

*queue* – Queue to obtain size for.

### Return Values

`int` – `-1` if implementation is not provided for this custom implementation, or the number of elements currently in the queue otherwise.

### Description

Returns the number of elements in the queue. The function returns a value that is a snapshot in time of the depth of the queue. Not all custom implementations support this function. This function is to be used for debug purposes only, because its implementation (when available) is computationally intensive and not meant for fast path operation.

### Example

```
printf ("Queue size is %d\n", teja_queue_get_size (queue));
```

# Late-Binding API Memory Pool Functions

## teja_memory_pool_get_node

### Function

```
void * teja_memory_pool_get_node(teja_memory_pool_t memory-pool);
```

### Parameters

*memory-pool* – Memory pool to allocate from.

### Return Values

`void *` – `NULL` if the memory pool is empty or the pointer to the newly allocated node.

### Description

Returns a pointer to a newly allocated fixed-sized node from the given memory pool.

### Example

```
void * node;
node = teja_memory_pool_get_node (pool);
if (node)
  {
    printf ("Got node %x\n", node);
    if (teja_memory_pool_put_node (pool, node) < 0)
      {
        printf ("Error putting back node %x to the pool\n", node);
      }
    else
      {
        printf ("Node %x was put back to the pool\n", node);
      }
  }
else
  {
    printf ("Pool was empty\n");
  }
```

## teja_memory_pool_put_node

### Function

`int teja_memory_pool_put_node(teja_memory_pool_t` *memory-pool*`, void * ` *node*`);`

### Parameters

*memory-pool* – Memory pool to return the node to.

*node* – Pointer to node to free.

### Return Values

`int` – If successful, this function returns `0`. In case of an error, this function returns `-1`.

### Description

Frees a node back to a memory pool.

### Example

See the example in "teja_memory_pool_get_node" on page 9.

## teja_memory_pool_get_node_from_index

### Function

```
void * teja_memory_pool_get_node_from_index(teja_memory_pool_t memory-pool,
            int index);
```

### Parameters

*memory-pool* – Memory pool from which the node belongs.

*index* – Index of the node.

### Return Values

`void *` – Pointer to the node specified by index.

### Description

Memory pool nodes are contiguous in memory and have a sequential index number. This function returns the node that corresponds to the given index. The effect of this function is not equivalent to a `teja_memory_pool_get_node` call because the node is not actually extracted from the pool. For this reason, the node must be allocated and not free in the memory pool when used by the application. For performance reasons, a range check is not performed, so the index value must be valid or a programming flaw might occur.

### Example

```
void * node;
node = teja_memory_pool_get_node_from_index (pool, 3);
if (teja_memory_pool_get_index_from_node (pool, node) != 3)
  {
    printf ("Impossible!\n");
  }
```

## teja_memory_pool_get_index_from_node

### Function

```
int teja_memory_pool_get_index_from_node(teja_memory_pool_t memory-pool, void
            * node);
```

### Parameters

*memory-pool* – Memory pool from which the node belongs.

### Return Values

`node` – Pointer to a node for which the index is requested.

`int` – Index of the given node.

### Description

Memory pool nodes are contiguous in memory and have a sequential index number. This function returns the index that corresponds to the given node pointer. For performance reasons, a range check is not performed, so the node value must be valid or a programming flaw might occur.

### Example

See the example in "`teja_memory_pool_get_node_from_index`" on page 11.

# Late-Binding API Channel Functions

## `teja_channel_is_connection_open`

### Function

`int teja_channel_is_connection_open(teja_channel_t` *channel*`);`

### Parameters

*channel* – Channel to test.

### Return Values

`int` – `1` if the connection is open, `0` otherwise (or if the channel is connectionless).

### Description

Returns `1` if the connection is open, `0` otherwise (or if the channel is connectionless).

### Example

See the example in "`teja_channel_send`" on page 14.

## teja_channel_make_connection

### Function

```
int teja_channel_make_connection(teja_channel_t channel);
```

### Parameters

*channel* – Channel to operate on.

### Return Values

`int` – `0` if operation was successful, `-1` otherwise.

### Description

Establishes a connection on the given channel (if the channel requires the connection to be established at runtime).

### Example

See the example in "teja_channel_send" on page 14.

## teja_channel_break_connection

### Function

```
int teja_channel_break_connection(teja_channel_t channel);
```

### Parameters

*channel* – Channel to operate on.

### Return Values

`int` – `0` if operation was successful, `-1` otherwise.

### Description

Breaks an existing connection on the given channel.

### Example

See the example in "teja_channel_send" on page 14.

## teja_channel_send

### Function

```
int teja_channel_send(teja_channel_t channel, short int event,
void * message, int message-size);
```

### Parameters

*channel* – Channel to send data on.

*event* – Optional value that is sent on the channel with the data. This value can be used at the receiver to discriminate the data type of the received data. This parameter is optional. Passing the constant `TEJA_NO_EVENT` causes event logic to be skipped in the code generation.

*message* – Pointer to the data to send.

*message-size* – Size of the message being sent (in bytes).

### Return Values

`int` – Number of bytes sent or `-1` in case of error.

### Description

Sends message-size bytes of data into the channel for the user. This function optionally enables users to send an event value that can be used at the receiver to discriminate the data type of the received data. This functionality is useful if multiple data types are sent. The event logic can be disabled by passing `TEJA_NO_EVENT`. Depending upon the channel implementation, the user might also be signaled at the time the message is sent.

### Example

This example shows how to send data using a channel.

```
#define MY_EVENT 7
if (teja_channel_make_connection(chan) < 0)
  {
    printf ("Error while estabilishing connecting to the channel\n");
  }
if (teja_channel_is_connection_open(chan))
  {
    if (teja_channel_send(chan,7,"hello",5) < 0)
      {
        printf ("Error sending data on the channel\n");
      }
    if (teja_channel_break_connection(chan) < 0)
      {
        printf ("Error while tearing down the connection on the channel\n");
      }
  }
```

See also the example in "teja_wait" on page 15, which shows how to receive data from the channel.

# Late-Binding API Interruptible Wait

The `teja_wait()` call enables users to wait for a timeout to expire or for data to arrive on a list of channels, whichever happens first. This function's semantics are similar to the `select()` call on UNIX (or Linux) systems. For targets on which the `TEJA_IS_RAW_OS` constant is not defined, the `teja_wait()` call can also be interrupted by Teja signals and by registered file descriptors. See "NPOS API Communication Functions" on page 24.

## teja_wait

### Function

`int teja_wait(int` *seconds*`, int` *nanoseconds*`, int` *poll-seconds*`, int` *poll-nanoseconds*`, short int * event, void * buffer, int` *buffer-size*`, ...);`

### Parameters

*seconds* – Number of seconds to wait. Passing `TEJA_INFINITE_WAIT` causes the function to wait indefinitely.

*nanoseconds* – Number of nanoseconds to wait. This value must be from 0 to 999999999.

*poll-seconds* – Number of seconds to wait before polling channels. Passing `TEJA_INFINITE_WAIT` causes the function not to poll channels.

*poll-nanoseconds* – Number of nanoseconds to wait before polling channels. This value must be from 0 to 999999999.

*event* – Pointer to a variable in which the event value is copied. Passing `NULL` causes event logic to be skipped.

*buffer* – Pointer to buffer in which received data is copied.

*buffer-size* – Size of the buffer.

... – List of channels to read from. The list must be `/codeNULL` terminated.

### Return Values

`int` – `-1` if error, `0` if timeout expires, or the number of bytes read from channels and copied into the buffer.

### Description

Waits for a timeout, for data arriving on one of the channels, or for any registered signals or file descriptor to be triggered, whichever happens first. For more information on signal and file descriptor registration, see "NPOS API Communication Functions" on page 24. Channels are checked once before starting the timeout wait.

The *seconds* and *nanoseconds* parameters identify the timeout. If `TEJA_INFINITE_WAIT` is passed to seconds, then no timing logic is generated and the function waits indefinitely until some data arrives on the channels. The *poll-seconds* and *poll-nanoseconds* identify the amount of time to wait between channel polls, while waiting. If `TEJA_INFINITE_WAIT` is passed to *poll-seconds*, then no polling logic is generated.

*event* is an optional parameter. If a non-`NULL` value is passed the event value coming from the sender is copied in the variable pointed by the event parameter. Typically *event* is used to discriminate among a set of possible types for the received data so the *event* can determine what data type to cast the received data to. In case event is not needed (for example, if only one data type is sent on the channel) then the code generator can be instructed to skip event management logic by using `TEJA_NO_EVENT` at the sender and `NULL` at the receiver.

*Buffer* and *buffer-size* identify the buffer in which received data is copied and its size.

The final variable argument list consists of a NULL-terminated channel list. The order in which channels are listed is the same that is used to poll channels. If no channels are listed, then only timing logic is generated.

### Example

This example shows how to receive data from a channel using `teja_wait()`.

```
#define BUF_SIZE 16
#define MY_EVENT 7
#define MY_OTHER_EVENT 8
struct A
{
  short int x;
  short int y;
};
int ret;
short int evt;
char buf[BUF_SIZE];
ret = teja_wait (TEJA_INFINITE_WAIT, 0, 1, 0, &evt, buf, BUF_SIZE, chan, NULL);
if (ret > 0)
  {
    switch (evt)
      {
      case MY_EVENT:
        printf ("%s\n", buf);
        break;
      case MY_OTHER_EVENT:
        printf ("%d,%d\n", ((struct A *)buf)->x, ((struct A *)buf)->y);
        break;
      }
  }
else if (ret == 0)
  {
    printf ("timeout expired\n");
  }
else
  {
    printf ("teja_wait encountered an error\n");
  }
```

See also the example in "teja_channel_send" on page 14, which shows how to send data.

# NPOS API

The NPOS API consists of portable abstractions over various operating system facilities such as threads, nonmemory pool-based memory management, thread management, socket communication, and signal registration and handling. Unlike Late-Binding APIs, NPOS APIs are not treated specially by the compiler and are implemented in precompiled libraries. See the *Netra Data Plane Software Suite 1.1 User's Guide* for an overview of this API.

## NPOS API Data Types

**TABLE 1-3** NPOS API Data Types

| Data Type | Description |
|---|---|
| `int8_t` | 8-bit integer type |
| `int16_t` | 16-bit integer type |
| `int32_t` | 32-bit integer type |
| `int64_t` | 64-bit integer type |
| `teja_fd_handler_t` | fd handler type, used with `teja_register_fd()`. This data type has the following prototype: `int (*` *handler*`) (teja_socket_t fd, void *` *signal-context*`, short int *` *event*`, void *` *msg*`, int` *msg-max-size*) |
| `teja_signal_handler_t` | Signal handler type, used with `teja_register_fd()`. This data type has the following prototype: `int (*` *handler*`) (int sig_code, void *` *signal-context*`, short int *` *event*`, void *` *msg*`, int` *msg-max-size*`)` |
| `teja_sockaddr_t` | Sockaddr type, used with socket API |
| `teja_socket_t` | Socket type, used with socket API |
| `teja_socklen_t` | Socklen type, used with socket API |
| `teja_thread_function_t` | Thread function. Has the following prototype: `void * (*` *function*`) (void *)` |
| `teja_thread_handle_t` | Thread handle type |
| `uint8_t` | 8-bit unsigned integer type |

**TABLE 1-3** NPOS API Data Types *(Continued)*

| Data Type | Description |
|---|---|
| uint16_t | 16-bit unsigned integer type |
| uint32_t | 32-bit unsigned integer type |
| uint64_t | 64-bit unsigned integer type |

**TABLE 1-4** NPOS API Macros

| Macros | Description |
|---|---|
| TEJA_AF_INET | |
| TEJA_DEFAULT_STACK_SIZE | |
| TEJA_FD_SETSIZE | Defined only on bare hardware targets. |
| TEJA_IS_RAW_OS | |
| TEJA_INADDR_ANY | |
| TEJA_SOCK_DGRAM | |
| TEJA_SOCK_STREAM | |

# NPOS API Memory Management Functions

The memory management functions offer `malloc` and `free` functionality. These functions are computationally expensive and only used in initialization code or non-relative critical code. On bare hardware targets the `free()` function is an empty operation, so use `malloc()` only to obtain memory that is not meant to be released. For all other purposes, use the memory pool API.

## teja_free

### Function

```
void teja_free(void * ptr);
```

### Parameters

*ptr* – Pointer to buffer to free.

### Return Values

void

### Description

Frees memory buffer. On bare hardware targets this operation is empty.

## teja_malloc

### Function

```
void * teja_malloc(size_t size);
```

### Parameters

*size* – Size in bytes of memory to allocate.

### Return Values

`void *` – Value to be used as pointer to allocated buffer.

### Description

Allocates memory buffer of specified *size*. On bare hardware targets the `teja_free()` operation is empty, so use `teja_malloc()` only to obtain memory that is not meant to be released. For all other purposes, use the memory pool API.

## teja_realloc

### Function

```
void * teja_realloc(void * ptr, size_t size);
```

### Parameters

*ptr* – Pointer to memory to reallocate.

*size* – Size in bytes of memory to allocate.

### Return Values

`void *` – Pointer to newly allocated memory or `NULL` if the operation failed. In case of failure the original block is left untouched.

### Description

Extends the memory buffer to become as big as the specified size. The new block might be allocated at a new address if there was not enough space for size bytes at the original location.

# NPOS API Thread Functions

This API offers thread management functionality. The `teja_thread_t` type implements thread IDs and the type can be assigned thread identifiers defined in the software architecture. You indicate these thread identifiers as strings in the software architecture using `teja_thread_create()`. In the user application, these identifiers are used as C identifiers (not as strings), which are defined by the compiler.

Two data types can be used to identify threads:

**TABLE 1-5** NPOS API Thread Types

| Data Type | Description |
|---|---|
| `teja_thread_t` | This type is associated only to threads that are defined in the software architecture, and not to dynamic threads, created with `teja_thread_handle_start()`. This data type is an identifier type. |
| `teja_thread_handle_t` | This type is a handle that is associated to every thread in the system, both software architecture threads and dynamic threads. This data type is a handle data structure. |

## `teja_get_thread_id`

### Function

```
teja_thread_t teja_get_thread_id(void);
```

### Return Values

`teja_thread_t` – Thread ID of the current thread.

### Description

Returns the thread ID of the current thread. The thread ID can be compared against thread identifiers defined in the software architecture.

## `teja_get_thread_name_for_id`

### Function

```
char * teja_get_thread_name_for_id(teja_thread_t thread-id);
```

### Parameters

*thread-id* – ID of the thread to operate on.

### Return Values

`char *` – Name of the given thread.

### Description

Returns the name of the given thread.

## teja_get_id_for_thread_name

### Function

`teja_thread_t teja_get_id_for_thread_name(char *` *name*`);`

### Parameters

*name* – Name of the thread to operate on.

### Return Values

`teja_thread_t` – ID of the given thread.

### Description

Returns the ID of the given thread.

## teja_thread_handle_start

### Function

`int teja_thread_handle_start(teja_thread_handle_t *` *thread*`, teja_thread_function_t` *function*`, void *` *arg*`, int` *stack-size*`, int` *priority*`);`

### Parameters

*thread* – Pointer to an uninitialized `TejaThread` instance. Upon successful execution, the thread contains a properly set up `TejaThread` handler.

*function* – Main function of the thread.

*arg* – Argument that is passed to the thread main function.

*stack-size* – Size of the stack for the thread. This functionality is not available on all systems. A predefined value is `TEJA_DEFAULT_STACK_SIZE`.

*priority* – prioRity of the thread. This functionality is not available on all systems. A predefined value is `TEJA_DEFAULT_PRIORITY`.

#### Return Values

`int` – `0` if execution was successful, `-1` if an error occurred.

#### Description

Starts a new thread dynamically, executing the given function. This function is available only on OS-based targets or targets for which the `TEJA_IS_RAW_OS` constant is not defined.

### teja_thread_handle_end

#### Function

```
void teja_thread_handle_end(void);
```

#### Return Values

`void`

#### Description

Ends a thread that was started with `teja_thread_handle_start()`. Do not use this function on threads defined in the software architecture. For software architecture threads use `teja_thread_shutdown()`. This function is available only on OS-based targets or targets for which the `TEJA_IS_RAW_OS` constant is not defined.

### teja_thread_handle_get_for_thread_id

#### Function

`teja_thread_handle_t * teja_thread_handle_get_for_thread_id(int` *thread-id*`);`

#### Parameters

*thread-id* – ID of the thread to operate on.

### Return Values

`teja_thread_handle_t *` – Handle pointer for the given thread ID.

### Description

Returns the thread handle pointer for the given thread ID. This function is available only on OS-based targets or targets for which the `TEJA_IS_RAW_OS` constant is not defined.

# NPOS API Miscellaneous Functions

## teja_thread_shutdown

### Function

```
void teja_thread_shutdown(void);
```

### Return Values

`void`

### Description

Shuts down the current Teja thread.

# NPOS API Communication Functions

The communication API is only available on OS-based targets or targets for which the `TEJA_IS_RAW_OS` constant is not defined. This API offers the following functionality:

- Signal registration and sending
- File descriptor registration
- Abstraction over TCP/IP communication

## Signal Handling

The Signal Handling API provides cross-platform, efficient signal handling.

The API operates on the `teja_thread_handle_t` type and can be applied both to software architecture threads and to dynamic threads created with `teja_thread_handle_start()`.

Teja signals can be registered and associated to handler functions. The signals can be sent to a destination `teja_thread_handle_t` that runs in the same process as the sender. Teja signals are cross-platform and can be used on systems that do not support a UNIX like signaling mechanism (such as Windows). The signals are also more efficient than OS signals and, unlike OS signals, the associated handler is called synchronously.

Any positive integer is a valid Teja signal code that can be passed to the registration function. However, if the signal code is also a valid OS code (such as `SIGUSR1` on UNIX), the signal is also registered using the native OS mechanism. This functionality means the thread reacts to OS signals as well as to Teja signals.

An important feature of Teja signals is that when received, the associated handler is called synchronously, so any function can be safely called from within the handler. This functionality removes the typical limitations of asynchronous handling. Even if the registered signal is a valid OS signal code, when the application receives an actual OS signal, the handler is still called synchronously. If a Teja process running multiple threads receives an OS signal, every one of its threads receives the signal, but only the threads that registered the process using the Teja API react to the signal.

Since Teja signals are handled synchronously, threads can only receive signals and execute the registered handler when the thread is in an interruptible state. The interruptible state is given by the `teja_wait()` function.

A typical, user-defined Teja signal handler reads any data from the relevant source, (for example, a memory-mapped device) and returns the signal to the caller. The caller is always `teja_wait()`. `teja_wait()` in turn exits and returns the data to user program.

The features of the signal API are:

- Registers a Teja software signal with a signal handler.
- Sends a Teja software signal to a thread in the same process.
- Performs interruptible waits. The wait is interrupted by a Teja or OS signal. See "`teja_wait`" on page 15.
- Returns data to `teja_wait()` and to the user application from a signal handler.

## File Descriptors

Registration of file descriptors (files and sockets) has some similarities to registration of signals. This operation registers a file descriptor with the system and associates the file descriptor with a user-defined handler and optionally with a context, which is also a user-defined value, such as a pointer. Whenever data is received on the file descriptor, the system automatically executes the associated handler and passes the handler to the file descriptor.

Just like Teja signal handlers, file descriptor handlers are invoked synchronously, so any function can be safely called from within the handler. This functionality removes the typical limitations of asynchronous handling.

Since file descriptor handlers are called synchronously, threads can only receive file descriptor input and execute the registered handler when the thread is in an interruptible state. The interruptible state is given by the `teja_wait()` function.

A typical file descriptor handler reads the data from the file descriptor and returns the data to `teja_wait()`, which in turn returns the data to the user application.

## teja_fd_isset

### Function

`int teja_fd_isset(teja_socket_t` *socket*`, teja_fd_set_t * ` *fd-set*`);`

### Parameters

*socket* – File descriptor to check.

*fd-set* – File descriptor set to operate on.

### Return Values

`int` – Nonzero value if the bit for socket is set in *fd-set* or `0` otherwise.

### Description

This function is available only on OS-based targets or targets for which the `TEJA_IS_RAW_OS` constant is not defined.

## teja_fd_set

### Function

`void teja_fd_set(teja_socket_t` *socket*`, teja_fd_set_t * ` *fd-set*`);`

### Parameters

*socket* – File descriptor to set.

*fd-set* – File descriptor set to operate on.

### Return Values

`void`

### Description

Sets the bit for the given file descriptor into the file descriptor set. This function is available only on OS-based targets or targets for which the `TEJA_IS_RAW_OS` constant is not defined.

## teja_fd_zero

### Function

```
void teja_fd_zero(teja_fd_set_t * fd-set);
```

### Parameters

*fd-set* – File descriptor set.

### Return Values

`void`

### Description

Zeroes the file descriptor set. This function is available only on OS-based targets or targets for which the `TEJA_IS_RAW_OS` constant is not defined.

## teja_register_fd_handler

### Function

`int teja_register_fd_handler(teja_socket_t` *fd*`, teja_fd_handler_t` *handler*`);`

### Parameters

*fd* – File descriptor to register.

*handler* – Pointer to a user-defined function handler that is executed by the application while in `teja_wait()` whenever data arrives on *fd*. If the application is not in `teja_wait()`, the *fd* data is not lost and is processed as soon as the application enters `teja_wait()`. The handler's *event* and *msg* parameters are return values that can be assigned by the handler. If the handler is `NULL` then the file descriptor is deregistered.

### Return Values

`int` – `0` if registration was successful, `-1` if registration failed.

### Description

Equivalent to `teja_register_fd_handler_with_context()`, passing `NULL` as the `fd_context`. This function is available only on OS-based targets or targets for which the `TEJA_IS_RAW_OS` constant is not defined.

## teja_register_fd_handler_with_context

### Function

```
int teja_register_fd_handler_with_context(teja_socket_t fd, void
* fd-context, teja_fd_handler_t handler);
```

### Parameters

*fd* – File descriptor to register.

*fd-context* – User-provided value (typically a pointer) that is passed to the handler when triggered.

*handler* – Pointer to a user-defined function handler that is executed by the application while in `teja_wait()` whenever data arrives on *fd*. If the application is not in `teja_wait()`, the *fd* data is not lost and is processed as soon as the application enters `teja_wait()`. Upon executing the *handler*, the system passes the user-defined *fd-context* to the handler and is referenced in the body of the handler. The handler's *event* and *msg* parameters are return values that can be assigned by the handler. If the handler is `NULL` then the file descriptor is deregistered.

### Return Values

`int` – `0` if registration was successful, `-1` if registration failed.

### Description

Registers a file descriptor handling routine and does not need to be called again in the file descriptor handler. Teja file descriptor handlers are synchronous, so all functions can be called from within the handler except blocking functions, because the entire thread gets blocked. The handler can read data from the file descriptor using `teja_read_from_socket()`. To deregister a file descriptor handler, pass the *fd* to deregister and a `NULL` handler. This function is available only on OS-based targets or targets for which the `TEJA_IS_RAW_OS` constant is not defined.

## teja_thread_handle_send_signal

### Function

`int teja_thread_handle_send_signal(teja_thread_handle_t *` *destination-thread-id*`, int` *signal-code*`);`

### Parameters

*destination-thread-id* – Destination thread.

*signal-code* – Signal to send.

### Return Values

`int` – `0` if execution was successful, `-1` if an error occurred.

### Description

Ends a Teja software signal to a thread identified by a `teja_thread_handle_t` pointer. If the destination thread is not a valid `teja_thread_handle_t`, the results are undefined. The receiving thread must have registered a signal handler for *signal-code*, using `teja_register_signal_handler()`.

This API element works only among threads in the same process. This function is available only on OS-based targets or targets for which the `TEJA_IS_RAW_OS` constant is not defined.

## teja_register_signal_handler

### Function

`int teja_register_signal_handler(int` *signal-code*`,`
`teja_signal_handler_t` *handler*`);`

### Parameters

*signal-code* – Identifies the code of the signal for which to register a signal handler. Negative numbers are reserved for the system and cannot be used by applications. If running on a UNIX like OS and the value is that of a proper OS signal code (for example, SIGUSR1), the function also registers the given handler function as an OS signal handler, so that the handler function is also invoked in response to OS software signals.

*handler* – Pointer to a user-defined function handler that is executed by the application while in `teja_wait()` whenever a signal is received. If the application is not in `teja_wait()`, the signal is not lost and is processed as soon as the

application enters `teja_wait()`. The handler's *event* and *msg* parameters are return values that can be assigned by the handler. If the handler is `NULL` then the signal is deregistered.

### Return Values

`int` – `0` if registration was successful, `-1` if registration failed.

### Description

Equivalent to `teja_register_signal_handler_with_context()`, passing `NULL` as the *signal-context*. This function is available only on OS-based targets or targets for which the `TEJA_IS_RAW_OS` constant is not defined.

## teja_register_signal_handler_with_context

### Function

```
int teja_register_signal_handler_with_context(int signal-code, void
* context, teja_signal_handler_t handler);
```

### Parameters

*signal-code* – Identifies the code of the signal for which to register a signal handler. Negative numbers are reserved for the system and cannot be used by applications. If running on a UNIX like OS and the value is that of a proper OS signal code (for example, SIGUSR1), the function also registers the given handler function as an OS signal handler, so that the handler function is also invoked in response to OS software signals.

*signal-context* – User-provided value (typically a pointer) that is passed to the handler when the handler is triggered.

*handler* – Pointer to a user-defined function handler that is executed by the application while in `teja_wait()` whenever a signal is received. If the application is not in `teja_wait()`, the signal is not lost and is processed as soon as the application enters `teja_wait()`. Upon executing the handler, the system passes the user-defined *signal-context*, which can therefore be referenced in the body of the handler. The handler's *event* and *msg* parameters are return values that can be assigned by the handler. If the handler is `NULL` then the signal is deregistered.

### Return Values

`int` – `0` if registration was successful, `-1` if registration failed.

### Description

Registers a signal-handling routine. The function does not need to be called again in the signal handler. Teja signal handlers are synchronous so all functions can be called from within the handler except blocking functions, because the entire thread gets blocked. The handler can read data using `teja_read_from_socket()`. If the function is called on a UNIX like OS and is passed a proper OS signal code (for example, `SIGUSR1`), the function also registers the given handler function as an OS signal handler. The handler function is also invoked in response to OS software signals. For example, on UNIX like systems, signals are generated with the `kill()` function. In this situation, the handler function is called synchronously.

To deregister a signal handler, you must pass the signal to deregister and a `NULL` handler. This function is available only on OS-based targets or targets for which the `TEJA_IS_RAW_OS` constant is not defined.

## teja_close_pipe

### Function

`void teja_close_pipe(teja_socket_t` *socket[2]*`);`

### Parameters

*socket[2]* – Array of `teja_socket_t` descriptor for the pipe to close.

### Return Values

`void`

### Description

Closes the given pipe. This function is available only on OS-based targets or targets for which the `TEJA_IS_RAW_OS` constant is not defined.

## teja_open_pipe

### Function

`int teja_open_pipe(teja_socket_t` *socket[2]*`);`

### Parameters

*socket[2]*

### Return Values

`int` – `0` if execution was successful. The value is the file descriptor of the opened socket. `-1` if an error occurred.

### Description

Opens a Teja pipe, creates two `teja_socket_t` descriptors, and stores them in the sock array. The `socket[0]` descriptor is used for reading and the `socket[1]` descriptor is used for writing. This function is available only on OS-based targets or targets for which the `TEJA_IS_RAW_OS` constant is not defined.

## teja_read_from_pipe

### Function

`int teja_read_from_pipe(teja_socket_t` *socket*`, char * ` *buffer*`, int` *buffer-size*`);`

### Parameters

*socket* – Read descriptor of the pipe to read from.

*buffer* – Buffer in which read data is stored.

*buffer-size* – Number of bytes to read from the pipe.

### Return Values

`int` – if execution was successful the number of read bytes is returned (`0` indicates end of file). `-1` if an error occurred.

### Description

Reads data from a Teja pipe. This function is available only on OS-based targets or targets for which the `TEJA_IS_RAW_OS` constant is not defined.

## teja_write_to_pipe

### Function

`int teja_write_to_pipe(teja_socket_t` *socket*`, char * ` *buffer*`, int` *buffer-size*`);`

### Parameters

*socket* – Write descriptor of the pipe to write to.

*buffer* – Buffer in which data to write to the pipe is stored.

*buffer-size* – Number of bytes to write to the pipe.

### Return Values

`int` – If execution was successful the number of written bytes is returned (this value is never greater than *buffer-size*). `-1` if an error occurred.

### Description

Writes data to a Teja pipe. This function is available only on OS-based targets or targets for which the `TEJA_IS_RAW_OS` constant is not defined.

## teja_select

### Function

`int teja_select(int nfds, teja_fd_set_t * read-fds, teja_fd_set_t * write-fds, teja_fd_set_t * exception-fds, int seconds, int nanoseconds);`

### Parameters

*read-fds* – Set of file descriptors to check for change in read status.

*write-fds* – Set of file descriptors to check for change in write status.

*exception-fds* – Set of file descriptors to check for change in exception status.

*seconds* – Seconds before the timeout expires.

*nanoseconds* – Nanoseconds before the timeout expires (must be less than 1000000000).

### Return Values

`int` – Number of descriptor that changed status, `0` if the timeout expired, or `-1` in case of error.

### Description

Waits for a group of file descriptors to change status or for the timeout to expire. This function is available only on OS-based targets or targets for which the `TEJA_IS_RAW_OS` constant is not defined.

## teja_accept

### Function

```
teja_socket_t teja_accept(teja_socket_t socket, teja_sockaddr_t *
address, teja_socklen_t * length);
```

### Parameters

*socket* – Socket that has been created with `teja_open_socket()`.

*address* – Pointer to a structure that contains the address of the connecting program.

*length* – Size of the structure pointed to by the address. This parameter is overwritten with the length of the returned address.

### Return Values

`teja_socket_t` – New connected socket or `-1` in case of error.

### Description

Creates a connected socket starting from the socket parameter. This function is available only on OS-based targets or targets for which the `TEJA_IS_RAW_OS` constant is not defined.

## teja_bind

### Function

```
teja_socket_t teja_bind(teja_socket_t socket, teja_sockaddr_t *
address, teja_socklen_t * length);
```

### Parameters

*socket* – Socket to bind.

*address* – Address to assign to the socket.

*length* – Length of the address.

### Return Values

`teja_socket_t` – `0` if successful, `-1` on error.

### Description

Assigns the *address* to the *socket*. The address is *length* bytes long. This function is available only on OS-based targets or targets for which the `TEJA_IS_RAW_OS` constant is not defined.

## teja_close_socket

### Function

```
void teja_close_socket(teja_socket_t socket);
```

### Parameters

*socket* – Socket to close.

### Return Values

`void`

### Description

Closes a socket. This function is available only on OS-based targets or targets for which the `TEJA_IS_RAW_OS` constant is not defined.

## teja_connect

### Function

```
int teja_connect(teja_socket_t socket, teja_sockaddr_t * address,
teja_socklen_t * length);
```

### Parameters

*socket* – Socket to connect from.

*address* – Address to connect to.

*length* – Length of the address.

### Return Values

`int` – `0` on success, `-1` on error.

### Description

A socket of type TEJA_SOCK_STREAM establishes a connection to the given *address*. With a socket of type TEJA_SOCK_DGRAM, the *address* is the default address to which datagrams are sent and received. This function is available only on OS-based targets or targets for which the TEJA_IS_RAW_OS constant is not defined.

## teja_get_ip_address_by_name_or_address

### Function

```
int teja_get_ip_address_by_name_or_address (char * name, int *
addr);
```

### Parameters

*name* – Name to resolve.

*addr* – Returned address.

### Return Values

int – 0 on success, -1 on error.

### Description

Performs a name to address resolution and returns the address in the *addr* variable. This function is available only on OS-based targets or targets for which the TEJA_IS_RAW_OS constant is not defined.

## teja_get_host_name

### Function

```
int teja_get_host_name (char * name, int len);
```

### Parameters

*name* – User-provided buffer in which the result is stored.

*len* – Length of the user-provided buffer.

### Return Values

int – 0 on success, -1 on error.

### Description

Returns the host name of the system on which the program is running. This function is available only on OS-based targets or targets for which the `TEJA_IS_RAW_OS` constant is not defined.

## teja_htonl

### Function

`uint32_t teja_htonl(uint32_t` *host-val*`);`

### Parameters

*host-val* – Unsigned 32-bit value in host format.

### Return Values

`uint32_t` – Unsigned 32-bit value converted to network format.

### Description

Converts an unsigned long value from host to network format. This function is available only on OS-based targets or targets for which the `TEJA_IS_RAW_OS` constant is not defined.

## teja_htons

### Function

`uint16_t teja_htons(uint16_t` *host-val*`);`

### Parameters

*host-val* – Unsigned 16-bit value in host format.

### Return Values

`uint16_t` – Unsigned 16-bit value converted to network format.

### Description

Converts an unsigned short value from host to network format. This function is available only on OS-based targets or targets for which the `TEJA_IS_RAW_OS` constant is not defined.

## teja_is_data_available_on_socket

### Function

```
int teja_is_data_available_on_socket(teja_socket_t socket);
```

### Parameters

*socket* – Socket to check.

### Return Values

`int` – `1` if data is available, `0` otherwise.

### Description

Checks whether data is available on a socket. This function is available only on OS-based targets or targets for which the `TEJA_IS_RAW_OS` constant is not defined.

## teja_listen

### Function

```
int teja_listen(teja_socket_t socket, int backlog);
```

### Parameters

*socket* – Socket that has previously been opened with `teja_open_socket()`.

*backlog* – Maximum number of entries in the pending connection queue.

### Return Values

`int` – `0` on success, `-1` on error.

### Description

Causes the program to listen for incoming connections on the given socket. This function is available only on OS-based targets or targets for which the `TEJA_IS_RAW_OS` constant is not defined. Variable that contains the r

## teja_open_socket

### Function

```
teja_socket_t teja_open_socket(int type);
```

### Parameters

*type* – Defines the type of socket to open. Can be one of:

- `TEJA_SOCK_STREAM` (for connection-oriented communication)
- `TEJA_SOCK_DGRAM` (for datagram-oriented communication)

### Return Values

`teja_socket_t` – Socket descriptor on success or `-1` on error.

### Description

Opens a new socket of the given *type*. This function is available only on OS-based targets or targets for which the `TEJA_IS_RAW_OS` constant is not defined.

## teja_read_from_socket

### Function

`int teja_read_from_socket(teja_socket_t` *socket*`, void *` *buffer*`, int` *length*`, int` *flags*`);`

### Parameters

*socket* – Socket to read from.

*buffer* – Buffer into which read data is copied.

*length* – Length of the buffer.

*flags* – System-dependent flags.

### Return Values

`int` – Number of bytes received or `-1` in case of error.

### Description

Reads data from a socket and stores the data in the provided buffer. This function is available only on OS-based targets or targets for which the `TEJA_IS_RAW_OS` constant is not defined.

## teja_write_to_socket

### Function

```
int teja_write_to_socket(teja_socket_t socket, void * buffer, int
length, int flags);
```

### Parameters

*socket* – Socket to write to.

*buffer* – Buffer to write to the socket.

*length* – Length of the buffer.

*flags* – System-dependent flags.

### Return Values

Number of bytes sent or `-1` in case of error.

### Description

Writes data to a socket. This function is available only on OS-based targets or targets for which the `TEJA_IS_RAW_OS` constant is not defined.

# NPOS API Time Functions

## teja_get_time

### Function

```
int teja_get_time(int * seconds, int * nanoseconds);
```

### Parameters

*seconds* – User-provided variable that contains the current seconds after the call.

*nanoseconds* – User-provided variable that contains the current nanoseconds after the call.

### Return Values

`int` – `0` on success, `-1` on error.

### Description

Returns the current time in *seconds* and *nanoseconds*. The precision depends on the granularity of the underlying system clock.

## teja_wait_time

### Function

```
int teja_wait_time(int seconds, int nanoseconds);
```

### Parameters

*seconds* – Number of seconds to wait. Passing `TEJA_INFINITE_WAIT` causes the function to wait indefinitely

*nanoseconds* – Number of nanoseconds to wait. This value must be between 0 and 999999999.

### Return Values

`int` – `0` on success, `-1` on error.

### Description

Causes the current thread to sleep the specified time. The actual sleep time varies, depending upon the granularity of the underlying system clock and the system overhead involved in rescheduling the thread.

# Miscellaneous Functions

## teja_get_argc

### Function

```
int teja_get_argc(void);
```

### Return Values

`int`

### Description

Returns the number of arguments passed to the program on the command line.

`teja_get_argv`

### Function

```
char ** teja_get_argv(void);
```

### Return Values

```
char **
```

### Description

Returns an array of strings containing the arguments passed to the program on the command line.

# Finite State Automata API

This macro-based API can be used to implement efficient state machine logic within a Teja application. States are computational elements and transitions are program flow elements that connect states.

These functions are available in different versions:

- Single-context vs. multiple-context
- Computed goto vs. function pointer

State machines come with a user-defined context. The first field of the context must be a `void *` pointer and is reserved for the system. You can freely add other fields.

The multiple-context version of the API invokes a user-provided scheduler to switch in a new context at the end of each transition. This is an efficient way to implement parallel execution on single-threaded systems. For example, while a context waits, the state machine could switch in a new context and continue computation, thus increasing the CPU utilization.

The single-context version of the API uses a simple pointer scheduler and does not perform any switching. This version is useful on architectures that support multithreading in hardware.

You might choose an implementation based on computed gotos, versus function pointers. Computed gotos might perform faster, but not all target compilers support them.

---

**Note –** State machines need to be declared outside of functions.

---

# Finite State Automata API Defines

**TABLE 1-6** Finite State Automata API Defines

| Macros | Description |
|---|---|
| TEJA_FSM_SINGLE_CONTEXT | If defined the single-context version of the API is used, otherwise the multi-context version of the API is used. |
| TEJA_FSM_COMPUTED_GOTO | If defined the computed goto optimized version is used, otherwise the regular function pointer based version is used. Computed goto might perform faster, but is not available on all target compilers. |
| TEJA_FSM_CONTEXT | Pointer to the current context. In case of single-context implementation, this value never changes. In case of multiple-context implementation, this value is updated by the system. |

# Finite State Automata API Macros

## teja_fsm_declare

### Function

```
#define teja_fsm_declare(name)
```

### Parameters

*name* – Name of the state machine.

### Description

Declares a state machine with the given name. This function must be used in the global scope outside functions.

## teja_fsm_begin

### Function

`#define teja_fsm_begin(`*name initial-state-name context-scheduler context-iterator*`)`

### Parameters

*name* – Name of the state machine.

*initial-state-name* – Name of the initial state.

*context-scheduler* – If using single-context mode, this is the pointer to the context. If using multi-context mode this is the name of a user-defined function (of signature `void * f (void)`) returning the next context.

*context-iterator* – Name of a user-defined function (of signature `void * f (void)`) that returns a pointer to the next context until there are no more contexts, in which case the function returns `NULL`. The system uses this function to iterate over the contexts in the beginning in order to initialize them. This function is not used in single-context mode.

### Description

Starts the definition of a state machine of the given name. This function must be used after `teja_fsm_declare` and must be used in the global scope outside functions. No semi-colon (;) is required at the end of this call.

## teja_fsm_end

### Function

`#define teja_fsm_end()`

### Description

Ends the definition of a state machine. This function must be used after `teja_fsm_begin` and must be used in the global scope outside functions. No semi-colon (;) is required at the end of this call.

## teja_fsm_start

### Function

`#define teja_fsm_start(`*name*`)`

#### Parameters

*name* – Name of the state machine.

#### Description

Starts execution of a state machine with the given name. This function must be used inside a function.

### teja_fsm_state_declare

#### Function

```
#define teja_fsm_state_declare(name)
```

#### Parameters

*name* – Name of the state.

#### Description

Declares a state with the given name. This function must be used inside a state machine declaration, immediately after `teja_fsm_begin`.

### teja_fsm_state_begin

#### Function

```
#define teja_fsm_state_begin(name)
```

#### Parameters

*name* – Name of the state.

#### Description

Starts the definition of a state of the given name. This function must be used inside a state machine after all `teja_fsm_declare` calls. You can add regular C code immediately after this macro up to the `teja_fsm_state_end` macro. No semi-colon (;) is required at the end of this call.

### teja_fsm_state_end

#### Function

```
#define teja_fsm_state_end()
```

### Description

Ends the definition of a state. This function must be used after `teja_fsm_state_begin`. You can add regular C code immediately before this macro. No semi-colon (;) is required at the end of this call.

## `teja_fsm_goto_state`

### Function

```
#define teja_fsm_goto_state(name)
```

### Parameters

*name* – Name of the state to jump to.

### Description

Performs a jump to the given state. This function must be used inside a state definition (that is between `teja_fsm_state_begin` and `teja_fsm_state_end`). This macro can be invoked No semi-colon (;) is required at the end of this call.

# FSM Example

CODE EXAMPLE 1-1 implements a simple state machine depicted in FIGURE 1-1.

- t1 – Thread 1
- t2 – Thread 2
- s1 – State 1
- s2 – State 2

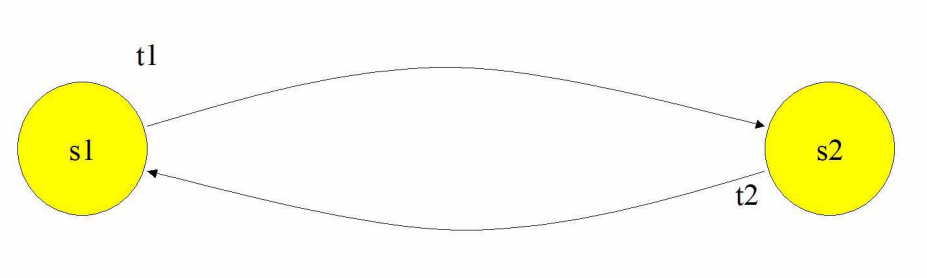


**FIGURE 1-1** Finite State Machine Example

**CODE EXAMPLE 1-1** Finite State Machine Code Example

```
#include <stdio.h>
#if NUM_CONTEXTS == 1
#define TEJA_FSM_SINGLE_CONTEXT
#endif
#include "fsm/teja_fsm.h"
typedef struct Context
{
  void * state;
  int count;
} Context;
static Context contexts[NUM_CONTEXTS];
#if NUM_CONTEXTS == 1
#define ctx_scheduler() (&contexts[0])
#else
void *
ctx_scheduler (void)
{
  static int i = -1;
  i = (i + 1) % NUM_CONTEXTS;
  return &contexts[i];
}
#endif
void *
ctx_iterator (void)
{
  static int i = -1;
  void * cur_context = 0;
  i = (i + 1);
  if (i < NUM_CONTEXTS)
    cur_context = &contexts[i];
  else
    i = -1;
  return cur_context;
}
teja_fsm_declare (my_fsm);
teja_fsm_begin (my_fsm, s2, ctx_scheduler, ctx_iterator)
  teja_fsm_state_declare (s1);
  teja_fsm_state_declare (s2);
  teja_fsm_state_begin (s1)
    printf ("t1\n");
    ((Context *) TEJA_FSM_CONTEXT)->count++;
    teja_fsm_goto_state (s2);
  teja_fsm_state_end ()
  teja_fsm_state_begin (s2)
    printf ("t2: %d\n", contexts[0].count);
```

**CODE EXAMPLE 1-1** Finite State Machine Code Example *(Continued) (Continued)*

```
    if (((Context *) TEJA_FSM_CONTEXT)->count == 100)
     teja_thread_shutdown();
    teja_fsm_goto_state (s1);
  teja_fsm_state_end()
teja_fsm_end()
void
fsm_main (void)
{
  int i;
  for (i = 0; i < NUM_CONTEXTS; i++)
    {
      contexts[i].count = 0;
    }
  teja_fsm_start (my_fsm);
}
```

# C Library Support on Bare Hardware

Teja programs running on bare hardware CMT can use the following standard C library functions:

- `atoi`
- `bcopy`
- `bzero`
- `getchar`
- `memcopy`
- `memmove`
- `memset`
- `printf` (Floating-point values, width, length, and precision are not currently supported in the format string.)
- `putchar`
- `sprintf` (Floating-point values, width, length, and precision are not currently supported in the format string.)
- `strcat`
- `strcmp`
- `strcpy`
- `strlen`
- `strncmp`

- strncpy
- strtok
- strtol
- strtoul

CHAPTER 2

# Configuration API

This chapter describes the components and functions of the Configuration API. The chapter includes the following topics:


- "Hardware Architecture API" on page 51
- "Software Architecture API" on page 85
- "Map API" on page 100
- "Error-Handling API" on page 104
- "CMT-Specific Hardware Architecture Constants" on page 106
- "CMT-Specific Software Architecture Constants" on page 108


## Hardware Architecture API

The Hardware Architecture API is used to describe the target hardware architecture of the application.

The file `teja_hardware_architecture.h` file contains the declaration of the data types and API functions.

# Hardware Architecture API Data Types

The hardware architecture definitions use the following data types.

**TABLE 2-1** Hardware Architecture API Data Types

| Data Type | Description |
|---|---|
| `teja_architecture_t` | Hardware architecture. An architecture might contain processors, memories, buses, hardware objects, and other architectures. |
| `teja_processor_t` | Processor. A processor is a target for an OS (`teja_os_t`). |
| `teja_memory_t` | Memory. A memory is a target for mapping variables declared in user-application source code. |
| `teja_bus_t` | Bus connecting objects with each other. |
| `teja_bus_visibility_t` | Buses have two types of visibility:<br>• `TEJA_INTERNAL_BUS` - bus not visible outside its containing architecture<br>• `TEJA_EXPORTED_BUS` - bus made visible outside its containing architecture<br>Example:<br>`typedef enum {TEJA_INTERNAL_BUS, TEJA_EXPORTED_BUS} teja_bus_visibility_t;` |
| `teja_hardware_object_t` | Generic hardware module that is not a processor, a memory, or a bus. |
| `teja_port_t` | Hardware port. |
| `teja_address_space_t` | Address space. An address space is used as context for allocating address ranges. |
| `teja_address_range_t` | Address range. An address range is a (lo, hi) range obtained from an address space. |

# Hardware Architecture API Functions

## `teja_architecture_create`

### Function

`teja_architecture_t teja_architecture_create(teja_architecture_t` *`container`*`, const char *` *`name`*`, const char *` *`type`*`);`

### Parameters

*container* – Container for the new architecture.

*name* – Name of the new architecture.

*type* – Type of the architecture.

### Return Values

`teja_architecture_t` – value that can be used as handle for the new architecture

### Description

Creates a new architecture with the specified name. The new architecture is contained in the container architecture. The top-level architecture is created by passing `NULL` as value for container. Legal values for the type parameter are found in the chip support package (CSP) specific properties, characterized by the `TEJA_ARCHITECTURE_` prefix. Most of the types result in a read-only, preconfigured architecture. To create custom architectures, use the `TEJA_ARCHITECTURE_USER_DEFINED` value for type.

## teja_architecture_set_property

### Function

`int teja_architecture_set_property(teja_architecture_t` *arch*`, const char * `*property-name*`, const char * `*value*`);`

### Parameters

*arch* – Architecture object.

*property-name* – Name of the property.

*value* – Value of the property.

### Return Values

`int` – Returns `TEJA_SUCCESS` or error code for failure.

### Description

Sets the value of the property for the architecture object.

## teja_architecture_get_property

### Function

```
int teja_architecture_get_property(teja_architecture_t arch,
const char * property-name, char * value, int buf-size);
```

### Parameters

*arch* – Architecture object.

*property-name* – Name of the property.

*value* – Returned value of the property.

*buf-size* – Size of the character buffer that was passed for the return value.

### Return Values

`int` – Returns the length of the current value of the property.

### Description

Returns the value of the property for the architecture object. If the returned value+1 is greater than the size of the passed buffer, the returned value is truncated. You must allocate a buffer with enough space to hold the value (returned value+1) and call the function again.

## teja_architecture_set_read_only

### Function

```
int teja_architecture_set_read_only(teja_architecture_t arch);
```

### Parameters

*arch* – Architecture object.

### Return Values

`int` – Returns `TEJA_SUCCESS` or error code for failure.

### Description

Prevents modification of given architecture by subsequent processing.

## teja_processor_create

### Function

```
teja_processor_t teja_processor_create(teja_architecture_t
container, const char * name, const char * type);
```

### Parameters

*container* – Container of the new processor.

*name* – Name of the new processor.

*type* – Type of the new processor.

### Return Values

`teja_processor_t` – Returns newly created processor object.

### Description

Creates a processor object.

## teja_processor_set_property

### Function

```
int teja_processor_set_property(teja_processor_t processor, const
char * property-name, const char * value);
```

### Parameters

*processor* – Processor object.

*property-name* – Name of the property.

*value* – Value of the property.

### Return Values

`int` – Returns `TEJA_SUCCESS` or error code for failure.

### Description

Sets the value of the property for the processor object.

## teja_processor_get_property

### Function

```
int teja_processor_get_property(teja_processor_t processor, const
char * property-name, char * value, int buf-size);
```

### Parameters

*processor* – Processor object.

*property-name* – Name of the property.

*value* – Value of the property.

*buf-size* – Size of the character buffer that was passed for the return value.

### Return Values

`int` – Returns the length of the current value of the property.

### Description

Returns the value of the property for the processor object. If the returned value+1 is greater than the size of the passed buffer, the returned value is truncated. You must allocate a buffer with enough space to hold the value (returned value+1) and call the function again.

## teja_processor_add_preprocessor_symbol

### Function

```
int teja_processor_add_preprocessor_symbol(teja_processor_t
processor, const char * symbol, const char * value);
```

### Parameters

*processor* – Processor instance to which the symbol is added.

*symbol* – Name of the symbol.

*value* – Value of the symbol.

### Return Values

`int` – Returns `TEJA_SUCCESS` or error code for failure.

### Description

Adds a preprocessor symbol to the processor. All of the processes running on the processor have the same symbol defined. The function adds convenience when passing values from the hardware architecture code into the user code.

## teja_memory_create

### Function

```
teja_memory_t teja_memory_create(teja_architecture_t container,
const char * name, const char * type);
```

### Parameters

*container* – Container of the new memory.

*name* – Name of the new memory.

*type* – Type of the new memory.

### Return Values

`teja_memory_t` – Returns the newly created memory object.

### Description

Creates a memory object

## teja_memory_set_property

### Function

```
int teja_memory_set_property(teja_memory_t memory, const char *
property-name, const char * value);
```

### Parameters

*memory* – Memory object.

*property-name* – Name of the property.

*value* – Value of the property.

### Return Values

`int` – Returns `TEJA_SUCCESS` or error code for failure.

### Description

Sets the value of the property for the memory object.

## teja_memory_get_property

### Function

```
int teja_memory_get_property(teja_memory_t memory, const char *
property-name, char * value, int buf-size);
```

### Parameters

*memory* – Memory object.

*property-name* – Name of the property.

*value* – Returned value of the property.

*buf-size* – Size of the character buffer that was passed for the return value.

### Return Values

`int` – Returns the length of the current value of the property.

### Description

Returns the value of the property for the memory object. If the returned value+1 is greater than the size of the passed buffer, the returned value is truncated. You must allocate a buffer with enough space to hold the value (returned value+1) and call the function again.

## teja_bus_create

### Function

```
teja_bus_t teja_bus_create(teja_architecture_t container, const
char * name, const char * type, teja_bus_visibility_t v);
```

### Parameters

*container* – Container of the new bus.

*name* – Name of the new bus.

*type* – Type of the new bus.

### Return Values

`teja_bus_t` – Returns newly created bus object.

### Description

Creates a bus object.

## teja_bus_set_property

### Function

`int teja_bus_set_property(teja_bus_t` *bus*`, const char *` *property-name*`, const char *` *value*`);`

### Parameters

*bus* – Bus object.

*property-name* – Name of the property.

*value* – Value of the property.

### Return Values

`int` – Returns `TEJA_SUCCESS` or error code for failure.

### Description

Sets the value of the property for the bus object.

## teja_bus_get_property

### Function

`int teja_bus_get_property(teja_bus_t` *bus*`, const char *` *property-name*`, char *` *value*`, int` *buf-size*`);`

### Parameters

*bus* – Bus object.

*property-name* – Name of the property.

*value* – Returned value of the property.

*buf-size* – Size of the character buffer that was passed for the return value.

### Return Values

`int` – Returns the length of the current value of the property.

### Description

Returns the value of the property for the bus object. If the returned value+1 is greater than the size of the passed buffer, the returned value is truncated. You must allocate a buffer with enough space to hold the value (returned value+1) and call the function again.

## teja_hardware_object_create

### Function

```
teja_hardware_object_t
teja_hardware_object_create(teja_architecture_t container, const
char * name, const char * type);
```

### Parameters

*container* – Container of the new hardware object.

*name* – Name of the new hardware object.

*type* – Type of the new hardware object.

### Return Values

`teja_hardware_object_t` – Returns newly created hardware object.

### Description

Creates a hardware object.

## teja_hardware_object_set_property

### Function

```
int teja_hardware_object_set_property(teja_hardware_object_t
hardware-object, const char * property-name, const char * value);
```

### Parameters

*hardware-object* – Hardware object.

*property-name* – Name of the property.

*value* – Value of the property.

### Return Values

`int` – Returns `TEJA_SUCCESS` or error code for failure.

### Description

Sets the value of the property for the hardware object.

## teja_hardware_object_get_property

### Function

```
int teja_hardware_object_get_property(teja_hardware_object_t
hardware-object, const char * property-name, char * value, int buf-size);
```

### Parameters

*hardware-object* – Hardware object.

*property-name* – Name of the property.

*value* – Returned value of the property.

*buf-size* – Size of the character buffer that was passed for the return value.

### Return Values

`int` – Returns the length of the current value of the property.

### Description

Returns the value of the property for the hardware_object. If the returned value+1 is greater than the size of the passed buffer, the returned value is truncated. You must allocate a buffer with enough space to hold the value (returned value+1) and call the function again.

## teja_architecture_connect

### Function

```
int teja_architecture_connect(teja_architecture_t architecture,
const char * bus-name, teja_bus_t bus);
```

### Parameters

*architecture* – Architecture object that needs to be connected.

*bus-name* – Name of the bus inside the architecture that is connected to the bus.

*bus* – Bus object that needs to be connected to the architecture.

#### Return Values

`int` – Returns `TEJA_SUCCESS` or error code for failure.

#### Description

Connects an architecture to a bus.

### teja_processor_connect

#### Function

`int teja_processor_connect(teja_processor_t` *processor*`, const char * bus-name, teja_bus_t bus);`

#### Parameters

*processor* – Processor object that needs to be connected.

*bus-name* – Name of the bus inside the processor that is connected to the bus.

*bus* – Bus object that needs to be connected to the processor.

#### Return Values

`int` – Returns `TEJA_SUCCESS` or error code for failure.

#### Description

Connects a processor to a bus.

### teja_memory_connect

#### Function

`int teja_memory_connect(teja_memory_t` *memory*`, const char * bus-name, teja_bus_t bus);`

#### Parameters

*memory* – Memory object that needs to be connected.

*bus-name* – Name of the bus inside the memory that is connected to the bus.

*bus* – Bus object that needs to be connected to the memory.

#### Return Values

`int` – Returns `TEJA_SUCCESS` or error code for failure.

#### Description

Connects a memory to a bus.

### teja_hardware_object_connect

#### Function

`int teja_hardware_object_connect(teja_hardware_object_t` *hardware-object*`, const char *` *bus-name*`, teja_bus_t` *bus*`);`

#### Parameters

*hardware-object* – Hardware object that needs to be connected.

*bus-name* – Name of the bus inside the hardware object that is connected to the bus.

*bus* – Bus object that needs to be connected to the hardware object.

#### Return Values

`int` – Returns `TEJA_SUCCESS` or error code for failure.

#### Description

Connects a hardware object to a bus.

### teja_lookup_architecture

#### Function

`teja_architecture_t teja_lookup_architecture(teja_architecture_t` *architecture*`, const char *` *architecture-name*`);`

#### Parameters

*architecture* – Container architecture in which you want to look up the architecture.

*architecture-name* – Name of the architecture to look up in the container.

### Return Values

`teja_architecture_t` – The found architecture or `NULL` if not found.

### Description

Looks up an architecture in the container.

## teja_lookup_processor

### Function

`teja_processor_t teja_lookup_processor(teja_architecture_t` *architecture*`, const char *` *processor-name*`);`

### Parameters

*architecture* – Container architecture in which you want to look up the processor.

*processor-name* – Name of the processor to look up in the container.

### Return Values

`teja_processor_t` – The found processor or `NULL` if not found.

### Description

Looks up a processor in the container.

## teja_lookup_memory

### Function

`teja_memory_t teja_lookup_memory(teja_architecture_t` *architecture*`, const char *` *memory-name*`);`

### Parameters

*architecture* – Container architecture in which you want to look up the memory.

*memory-name* – Name of the memory to look up in the container.

### Return Values

`teja_memory_t` – The found memory or `NULL` if not found.

### Description

Looks up a memory in the container.

## teja_lookup_bus

### Function

```
teja_bus_t teja_lookup_bus(teja_architecture_t architecture, const
char * bus-name);
```

### Parameters

*architecture* – Container architecture in which you want to look up the bus.

*bus-name* – Name of the bus to look up in the container.

### Return Values

`teja_bus_t` – The found bus or `NULL` if not found.

### Description

Looks up a bus in the container.

## teja_lookup_hardware_object

### Function

```
teja_hardware_object_t
teja_lookup_hardware_object(teja_architecture_t architecture, const
char * hardware-object-name);
```

### Parameters

*architecture* – Container architecture in which you want to look up the hardware object.

*hardware-object-name* – Name of the hardware object to look up in the container.

### Return Values

`teja_hardware_object_t` – The found hardware object or `NULL` if not found.

### Description

Looks up a hardware object in the container.

## teja_port_create

### Function

```
teja_port_t teja_port_create(teja_architecture_t arch, const char
* port-name, const char * dir);
```

### Parameters

*arch* – Container architecture in which the port is created.

*port-name* – Name of the port.

*dir* – Direction of the port. Legal values are `IN` and `OUT`.

### Return Values

`teja_port_t` – Returns the newly created port.

### Description

Creates a port in an hardware architecture. The port can be connected externally to ports of objects in the containing architecture, or ports of the containing architecture itself. See "teja_architecture_set_port_internal" on page 67. The port can also be connected internally to objects contained in this architecture. See "teja_architecture_set_port" on page 66.

## teja_architecture_set_port

### Function

```
int teja_architecture_set_port(teja_architecture_t arch, const
char * port-name, const char * value);
```

### Parameters

*arch* – Architecture containing the port.

*port-name* – Name of the architecture port.

*value* – Value to be assigned externally to the port.

### Return Values

`int` – Returns `TEJA_SUCCESS` or error code for failure.

### Description

Assigns a value externally to an architecture port. If a port of another object in the architecture containing *arch* is assigned with the same value, the two ports are connected. Also, if ports belonging to the architecture containing *arch* are assigned internally with the same value, the two ports are connected. The *value* represents the name of a wire connecting the ports.

## teja_architecture_set_port_internal

### Function

`int teja_architecture_set_port_internal(teja_architecture_t` *arch*`, const char * port-name, const char * value);`

### Parameters

*arch* – Architecture containing the port.

*port-name* – Name of the architecture port.

*value* – Value to be assigned internally to the port.

### Return Values

`int` – Returns `TEJA_SUCCESS` or error code for failure.

### Description

Assigns a value internally to an architecture port. If a port of another object contained in *arch* is assigned with the same value, the two ports are connected. The *value* represents the name of a wire connecting the ports.

## teja_processor_set_port

### Function

`int teja_processor_set_port(teja_processor_t` *proc*`, const char *` *port-name*`, const char * value);`

### Parameters

*proc* – Processor containing the port.

*port-name* – Name of the port.

*value* – Value to be assigned to the port.

### Return Values

`int` – Returns `TEJA_SUCCESS` or error code for failure.

### Description

Assigns a value to a processor port. If a port of another object in the architecture containing the processor is assigned with the same value, the two ports are connected. Also, if ports belonging to the architecture containing the processor are assigned internally with the same value, the two ports are connected. The *value* represents the name of a wire connecting the ports.

## teja_memory_set_port

### Function

`int teja_memory_set_port(teja_memory_t` *memory*`, const char *` *port-name*`, const char *` *value*`);`

### Parameters

*memory* – Memory containing the port.

*port-name* – Name of the port.

*value* – Value to be assigned to the port.

### Return Values

`int` – Returns `TEJA_SUCCESS` or error code for failure.

### Description

Assigns a value to a memory port. If a port of another object in the architecture containing the memory is assigned with the same value, the two ports are connected. Also, if ports belonging to the architecture containing the memory are assigned internally with the same value, the two ports are connected. The *value* represents the name of a wire connecting the ports.

## teja_hardware_object_set_port

### Function

`int teja_hardware_object_set_port(teja_hardware_object_t` *hardware-object*`, const char *` *port-name*`, const char *` *value*`);`

### Parameters

*hardware-object* – Hardware object containing the port.

*port-name* – Name of the port.

*value* – Value to be assigned to the port.

### Return Values

`int` – Returns `TEJA_SUCCESS` or error code for failure.

### Description

Assigns a value to a hardware object port. If a port of another object in the architecture containing the hardware object is assigned with the same value, the two ports are connected. Also, if ports belonging to the architecture containing the hardware object are assigned internally with the same value, the two ports are connected. The *value* represents the name of a wire connecting the ports.

## teja_bus_set_port

### Function

`int teja_bus_set_port(teja_bus_t` *bus*`, const char *` *port-name*`, const char *` *value*`);`

### Parameters

*bus* – Bus containing the port.

*port-name* – Name of the port.

*value* – Value to be assigned to the port.

### Return Values

`int` – Returns `TEJA_SUCCESS` or error code for failure.

### Description

Assigns a value to a bus port. If a port of another object in the architecture containing the bus is assigned with the same value, the two ports are connected. Also, if ports belonging to the architecture containing the bus are assigned internally with the same value, the two ports are connected. The *value* represents the name of a wire connecting the ports.

## teja_port_add_property

### Function

```
int teja_port_add_property(teja_port_t port, const char * property-
name, const char * value, const char * description);
```

### Parameters

*port* – Port to which the property is added.

*property-name* – Name of the new property.

*value* – Value of the new property.

*description* – Description associated to the property.

### Return Values

`int` – Returns `TEJA_SUCCESS` or error code for failure.

### Description

Associates a new key=value pair to a port. This allows association of target-specific properties to ports.

## teja_architecture_get_parent

### Function

```
teja_architecture_t
teja_architecture_get_parent(teja_architecture_t architecture);
```

### Parameters

*architecture* – An architecture.

### Return Values

`teja_architecture_t` – The architecture containing the one passed as parameter, or `NULL` if such architecture is the top-level one.

### Description

Returns the parent architecture for the specified architecture. If the specified architecture has no parent (for example, the top level architecture), `NULL` is returned.

## teja_processor_get_parent

### Function

`teja_architecture_t teja_processor_get_parent(teja_processor_t` *processor*`);`

### Parameters

*processor* – A processor.

### Return Values

`teja_architecture_t` – The architecture containing the processor.

### Description

Returns the architecture containing the specified processor.

## teja_bus_get_parent

### Function

`teja_architecture_t teja_bus_get_parent(teja_bus_t` *bus*`);`

### Parameters

*bus* – A bus.

### Return Values

`teja_architecture_t` – The architecture containing the bus.

### Description

Returns the architecture containing the specified processor.

## teja_memory_get_parent

### Function

`teja_architecture_t teja_memory_get_parent(teja_memory_t` *memory*`);`

### Parameters

*memory* – A memory.

### Return Values

`teja_architecture_t` – The architecture containing the memory.

### Description

Returns the architecture containing the specified memory.

## teja_hardware_object_get_parent

### Function

```
teja_architecture_t
teja_hardware_object_get_parent(teja_hardware_object_t hardware-object);
```

### Parameters

*hardware-object* – A hardware object.

### Return Values

`teja_architecture_t` – The architecture containing the hardware object.

### Description

Returns the architecture containing the specified hardware object.

## teja_architecture_get_processors

### Function

```
teja_processor_t *
teja_architecture_get_processors(teja_architecture_t arch);
```

### Parameters

*arch* – An architecture.

### Return Values

`teja_processor_t *` – The null-terminated array of processors contained in the architecture.

### Description

Returns an array of processors contained in the architecture. If the architecture contains *N* processors, the array contains *N*+1 entries, with entry *N* set to `NULL`. You must deallocate the array by calling `free()` on it.

## teja_architecture_get_memories

### Function

```
teja_memory_t *
teja_architecture_get_memories(teja_architecture_t arch);
```

### Parameters

*arch* – An architecture.

### Return Values

`teja_memory_t *` – The null-terminated array of memories contained in the architecture.

### Description

Returns an array of memories contained in the architecture. If the architecture contains *N* memories, the array contains *N*+1 entries, with entry *N* set to `NULL`. You must deallocate the array by calling `free()` on it.

## teja_architecture_get_hardware_objects

### Function

```
teja_hardware_object_t *
teja_architecture_get_hardware_objects(teja_architecture_t
arch);
```

### Parameters

*arch* – An architecture.

### Return Values

`teja_hardware_object_t *` – The null-terminated array of hardware objects contained in the architecture.

### Description

Returns an array of hardware objects contained in the architecture. If the architecture contains *N* objects, the array contains *N*+1 entries, with entry *N* set to `NULL`. You must deallocate the array by calling `free()` on it.

## teja_architecture_get_busses

### Function

```
teja_bus_t * teja_architecture_get_busses(teja_architecture_t arch);
```

### Parameters

*arch* – An architecture.

### Return Values

`teja_bus_t *` – The null-terminated array of buses contained in the architecture.

### Description

Returns an array of buses contained in the architecture. If the architecture contains *N* buses, the array contains *N*+1 entries, with entry *N* set to `NULL`. You must deallocate the array by calling `free()` on it.

## teja_architecture_get_architectures

### Function

```
teja_architecture_t *
teja_architecture_get_architectures(teja_architecture_t arch);
```

### Parameters

*arch* – An architecture.

### Return Values

`teja_architecture_t *` – The null-terminated array of architectures contained in the architecture.

### Description

Returns an array of architectures contained in the architecture. If the architecture contains *N* architectures, the array contains *N*+1 entries, with entry *N* set to `NULL`. You must deallocate the array by calling `free()` on it..

## teja_processor_get_connected_bus

### Function

`teja_bus_t teja_processor_get_connected_bus(teja_processor_t` *processor*`, const char *` *internal-bus-name*`);`

### Parameters

*processor* – A processor.

*internal-bus-name* – Name of a bus internal to the processor.

### Return Values

`teja_bus_t` – The bus connected to the specified internal processor bus, or `NULL`.

### Description

Returns the bus connected to the specified internal processor, or `NULL`. For example, a bus b contained in the same architecture as a processor and connected to such processor, actually connects to an bus contained inside the processor. Given the processor and the name of the bus contained in it (*internal-bus-name)*, this function returns b (`teja_bus_t`). If no bus is connected to the specified internal bus, `NULL` is returned.

## teja_memory_get_connected_bus

### Function

`teja_bus_t teja_memory_get_connected_bus(teja_memory_t` *memory*`, const char *` *internal-bus-name*`);`

### Parameters

*memory* – A memory.

*internal-bus-name* – Name to the bus internal to the memory.

### Return Values

`teja_bus_t` – The bus connected to the specified internal memory bus, or `NULL`.

### Description

Returns the bus connected to the specified memory, or `NULL`. For example, a bus b contained in the same architecture as a processor and connected to such memory, actually connects to an bus contained inside the memory. Given the memory and the name of the bus contained in it (*internal-bus-name)*, this function returns b (`teja_bus_t`). If no bus is connected to the specified internal bus, `NULL` is returned.

## teja_hardware_object_get_connected_bus

### Function

```
teja_bus_t
teja_hardware_object_get_connected_bus(teja_hardware_object_t
hardware-object, const char * internal-bus-name);
```

### Parameters

*hardware-object* – An hardware object.

*internal-bus-name* – Name of a bus internal to the hardware object.

### Return Values

`teja_bus_t` – The bus connected to the specified internal hardware object bus, or `NULL`.

### Description

Returns the bus connected to the specified hardware object, or `NULL`. For example, a bus b contained in the same architecture as a hardware object and connected to such hardware object , actually connects to an bus contained inside the hardware object. Given the hardware object and the name of the bus contained in it (*internal-bus-name)*, this function returns b (`teja_bus_t`). If no bus is connected to the specified internal bus, `NULL` is returned.

## teja_architecture_get_connected_bus

### Function

```
teja_bus_t
teja_architecture_get_connected_bus(teja_architecture_t
architecture, const char * internal-bus-name);
```

### Parameters

*architecture* – An architecture.

*internal-bus-name* – Name of a bus internal to the architecture.

### Return Values

`teja_bus_t` – The bus connected to the specified internal architecture bus, or `NULL`.

### Description

Returns the bus connected to the specified architecture bus, or `NULL`. For example, consider an architecture arch1 contained in an architecture arch2, a bus b1 contained in arch1, and a bus b2 contained in arch2. If b1 and b2 are connected, calling this function with arch2 as first parameter (*architecture*) and the name of b2 as the second (*internal-bus-name*) returns b1. If no bus is connected to b2, `NULL` is returned.

## teja_bus_get_connected_processors

### Function

```
teja_processor_t * teja_bus_get_connected_processors(teja_bus_t
bus);
```

### Parameters

*bus* – A bus.

### Return Values

`teja_processor_t *` – A `NULL`-terminated array of processors connected to the bus.

### Description

Returns an array of processors connected to the specified bus. If *N* processors are connected to the bus, the array contains *N*+1 entries, with entry *N* set to `NULL`. You must deallocate the array by calling `free()` on it.

## teja_bus_get_connected_memories

### Function

```
teja_memory_t * teja_bus_get_connected_memories(teja_bus_t bus);
```

### Parameters

*bus* – A bus.

### Return Values

`teja_memory_t *` – A `NULL`-terminated array of memories connected to the bus.

### Description

Returns an array of memories connected to the specified bus. If *N* memories are connected to the bus, the array contains *N*+1 entries, with entry *N* set to `NULL`. You must deallocate the array by calling `free()` on it.

## teja_bus_get_connected_hardware_objects

### Function

```
teja_hardware_object_t *
teja_bus_get_connected_hardware_objects(teja_bus_t bus);
```

### Parameters

*bus* – A bus.

### Return Values

`teja_hardware_object_t *` – A `NULL`-terminated array of hardware objects connected to the bus.

### Description

Returns an array of hardware objects connected to the specified bus. If *N* hardware objects are connected to the bus, the array contains *N*+1 entries, with entry *N* set to `NULL`. You must deallocate the array by calling `free()` on it.

## teja_bus_get_connected_architectures

### Function

```
teja_architecture_t *
teja_bus_get_connected_architectures(teja_bus_t bus);
```

### Parameters

*bus* – A bus.

### Return Values

`teja_architecture_t *` – A `NULL`-terminated array of architectures connected to the bus.

### Description

Returns an array of architectures connected to the specified bus. If *N* architectures are connected to the bus, the array contains *N*+1 entries, with entry *N* set to `NULL`. You must deallocate the array by calling `free()` on it.

## teja_processor_get_busses

### Function

`teja_bus_t * teja_processor_get_busses(teja_processor_t` *processor*`);`

### Parameters

*processor* – A processor.

### Return Values

`teja_bus_t *` – A `NULL`-terminated array of buses contained in the processor.

### Description

Returns an array of buses contained the specified processor. If the processor contains *N* buses, the array contains *N*+1 entries, with entry *N* set to `NULL`. You must deallocate the array by calling `free()` on it.

## teja_memory_get_busses

### Function

```
teja_bus_t * teja_memory_get_busses(teja_memory_t memory);
```

### Parameters

*memory* – A memory.

### Return Values

`teja_bus_t *` – A `NULL`-terminated array of buses contained in the memory.

### Description

Returns an array of buses contained the specified memory. If the memory contains *N* buses, the array contains *N*+1 entries, with entry *N* set to `NULL`. You must deallocate the array by calling `free()` on it.

## teja_hardware_object_get_busses

### Function

```
teja_bus_t *
teja_hardware_object_get_busses(teja_hardware_object_t hardware-
object);
```

### Parameters

*hardware-object* – A hardware object.

### Return Values

`teja_bus_t *` – A `NULL`-terminated array of buses contained in the hardware object.

### Description

Returns an array of busses contained the specified hardware object. If the object contains *N* busses, the array contains *N*+1 entries, with entry *N* set to `NULL`. You must deallocate the array by calling `free()` on it.

## teja_address_space_create

### Function

```
teja_address_space_t
teja_address_space_create(teja_architecture_t arch, const char *
name, const char * base, const char * high);
```

### Parameters

*arch* – An architecture.

*name* – Name of an address space to be created.

*base* – Base address for the space.

*high* – Highest address in the space.

### Return Values

`teja_address_space_t` – The newly created address space.

### Description

Allocates an address space with the specified name, base, and high address, and associated to the specified architecture. Requests for address ranges with various constraints are performed against an address space. At compile time all the request are resolved into actual address ranges within the space. Base and high address are specified as strings containing the hexadecimal address representation (for example, `'0x100000000'`).

## teja_address_space_join

### Function

```
int teja_address_space_join(teja_address_space_t s1,
teja_address_space_t s2);
```

### Parameters

*s1* – An address space.

*s2* – An address space.

### Return Values

`int` – `TEJA_SUCCESS` or error code for failure.

### Description

Joins two address spaces. Address range requests performed against the two spaces is resolved as if the two addresses had been issued against a single space. The base-high range of the resulting address space is the union of the two original ranges.

## teja_address_range_create_absolute

### Function

```
teja_address_range_t
teja_address_range_create_absolute(teja_address_space_t space,
const char * sym, const char * base, const char * size,);
```

### Parameters

*space* – An address space.

*sym* – A symbol to be associated to the range.

*base* – Base address for the range.

*size* – Size of the range.

### Return Values

`teja_address_range_t` – The newly created address range.

### Description

Creates an address range with the specified absolute address and size. The symbol has to be unique with respect to address ranges created against the same address space.

## teja_address_range_create_aligned

### Function

```
teja_address_range_t
teja_address_range_create_aligned(teja_address_space_t space,
const char * sym, const char * alignment, const char * size, const
char * minaddr);
```

### Parameters

*space* – An address space.

*sym* – A symbol to be associated to the range.

*alignment* – An alignment constraint.

*size* – Size of the range.

*minaddr* – A lower bound to the base address for the range.

### Return Values

`teja_address_range_t` – The newly created address range.

### Description

Creates an address range with the specified size. When address range resolution is performed, this range is assigned a base address that is multiple of alignment, but not smaller than *minaddr*. The symbol has to be unique with respect to address ranges created against the same address space.

## teja_address_range_create_generic

### Function

```
teja_address_range_t
teja_address_range_create_generic(teja_address_space_t space,
const char * sym, const char * size, const char * minaddr);
```

### Parameters

*space* – An address space.

*sym* – A symbol to be associated to the range.

*size* – Size of the range.

*minaddr* – A lower bound to the base address for the range.

### Return Values

`teja_address_range_t` – The newly created address range.

### Description

Creates an address range with the specified size. When address range resolution is performed, this range is assigned a base address higher than *minaddr*. The symbol has to be unique with respect to address ranges created against the same address space.

## teja_address_range_get_lower_bound

### Function

```
char * teja_address_range_get_lower_bound(teja_address_range_t
range, char * buf, int buf-size);
```

### Parameters

*range* – An address range.

*buf* – An array of characters.

*buf-size* – Size of the array of characters.

### Return Values

`char *` – The array of characters filled with the handle, or `NULL` in case of error.

### Description

This function returns a handle for the lower bound of the address range. The handle can be set as value for any property. When address resolution is performed, `tejacc` replaces such value with the actual lower bound address assigned to the range. If the array passed as buffer to store the handle is not large enough, `NULL` is returned.

## teja_address_range_get_upper_bound

### Function

```
char * teja_address_range_get_upper_bound(teja_address_range_t
range, char * buf, int buf-size);
```

### Parameters

*range* – An address range.

*buf* – An array of characters.

*buf-size* – Size of the array of characters.

### Return Values

`char *` – The array of characters filled with the handle, or `NULL` in case of error.

### Description

Returns a handle for the upper bound of the address range. The handle can be set as value for any property. When address resolution is performed, `tejacc` replaces the value with the actual upper bound address assigned to the range. If the array passed as a buffer to store the handle is not large enough, `NULL` is returned.

# Software Architecture API

The Software Architecture API is used to describe the threads, processes, and OS composing the software part of the application, as well as Teja mutexes, queues, memory pools, and channels used by the various threads.

The `teja_software_architecture.h` file contains the declaration of the API functions.

## Software Architecture API Data Types

The following data types are used in the software architecture definitions.

**TABLE 2-2** Software Architecture API Data Types

| | |
|---|---|
| `teja_os_t` | OS. An OS runs on one or more processors. These are the different OS types that are supported for different targets. Refer to the chip support package documentation for which operating systems are supported for that particular chip support package. |
| `teja_process_t` | Process. One or more processes run on an OS. |
| `teja_thread_t` | Thread. One or more threads run in a process. |
| `teja_channel_t` | Channel. Channels provide the communication facility to send structured data between two or more threads. |
| `teja_memory_pool_t` | Memory pool. A memory pool is a pool of same-sized nodes that are pre-allocated. The memory pool provides an efficient mechanism for memory allocation and deallocation at runtime without the cost of dynamic memory allocation. |
| `teja_queue_t` | Queue. A queue provides a facility to pass structured data between two or more threads. |
| `teja_mutex_t` | Mutex. A mutex provides a synchronization facility between two or more threads. |

# Software Architecture API Functions

## teja_os_create

### Function

```
teja_os_t teja_os_create(const char ** processor-names, const char
* name, const char * type);
```

### Parameters

*processor-names* – `NULL`-terminated array of processor names on which the OS is running.

*name* – Name of the OS instance.

*type* – Type of the OS.

### Return Values

`teja_os_t` – Returns an object that represents the OS instance.

### Description

Creates an OS instance. Return value can be used to set or get properties of the OS.

## teja_os_set_property

### Function

```
int teja_os_set_property(teja_os_t os, const char * property-name,
const char * value);
```

### Parameters

*os* – OS instance for which the property is being set.

*property-name* – Name of the property.

*value* – Value of the property to be set.

### Return Values

`int` – Returns `TEJA_SUCCESS` or error code for failure.

### Description

Sets the specified value of the property for the OS instance.

## teja_os_get_property

### Function

```
int teja_os_get_property(teja_os_t os, const char * property-name,
const char * value, int buf-size);
```

### Parameters

*os* – OS instance for which the property is being read.

*property-name* – Name of the property.

*value* – Value that is read and returned.

*buf-size* – Size of the character buffer that was passed for the return value.

### Return Values

`int` – Returns the length of the current value of the property.

### Description

Returns the current value of the OS property. If the returned value+1 is greater than the size of the passed buffer, the returned value is truncated. You must allocate a buffer with enough space to hold the value (returned value+1) and call the function again.

## teja_process_create

### Function

```
teja_process_t teja_process_create(teja_os_t container, const char
* name, const char ** srcset);
```

### Parameters

*container* – OS instance where the process is created.

*name* – Name of the instance.

*srcset* – `NULL`-terminated list of one or more source sets that are part of the process.

### Return Values

`teja_process_t` – Returns created process instance.

### Description

Creates a process instance. One or more processes can be created per OS. The source files listed in the source sets comprise the sources for the process.

## teja_process_set_property

### Function

`int teja_process_set_property(teja_process_t` *process*`, const char *` *property-name*`, const char *` *value*`);`

### Parameters

*process* – Process instance for which the property is being set.

*property-name* – Name of the property.

*value* – Value of the property.

### Return Values

`int` – Returns `TEJA_SUCCESS` or error code for failure.

### Description

Sets a process property.

## teja_process_get_property

### Function

`int teja_process_get_property(teja_process_t` *process*`, const char *` *property-name*`, const char *` *value*`, int` *buf-size*`);`

### Parameters

*process* – Process instance for which the property is being read.

*property-name* – Name of the property.

*value* – Returned value of the property.

*buf-size* – Size of the character buffer that was passed for the return value.

### Return Values

`int` – Returns the length of the current value of the property.

### Description

Returns the current value of the process property. If the returned value+1 is greater than the size of the passed buffer, the returned value is truncated. You must allocate a buffer with enough space to hold the value (returned value+1) and call the function again.

## teja_processor_add_preprocessor_symbol

See "teja_processor_add_preprocessor_symbol" on page 56

## teja_thread_create

### Function

`teja_thread_t teja_thread_create(teja_process_t` *container*`, const char * name);`

### Parameters

*container* – Process instance where the thread is created.

*name* – Name of the thread.

### Return Values

`teja_thread_t` – Returns thread instance.

### Description

Creates a thread instance. One or more threads can be created per process.

## teja_thread_set_property

### Function

`int teja_thread_set_property(teja_thread_t` *thread*`, const char *` *property-name*`, const char *` *value*`);`

### Parameters

*thread* – Thread instance for which the property is being set.

*property-name* – Name of the property.

*value* – Value of the property.

#### Return Values

`int` – Returns `TEJA_SUCCESS` or error code for failure.

#### Description

Sets a thread property.

### teja_thread_get_property

#### Function

```
int teja_thread_get_property(teja_thread_t thread, const char *
property-name, const char * value, int buf-size);
```

#### Parameters

*thread* – Thread instance for which the property is being read.

*property-name* – Name of the property.

*value* – Returned value of the property.

*buf-size* – Size of the character buffer that was passed for the return value.

#### Return Values

`int` – Returns the length of the current value of the property.

#### Description

Returns the current value of a thread property. If the returned value+1 is greater than the size of the passed buffer, the returned value is truncated. You must allocate a buffer with enough space to hold the value (returned value+1) and call the function again.

### teja_lookup_os

#### Function

```
teja_os_t teja_lookup_os(const char * os-name);
```

### Parameters

*os-name* – Name of the OS.

### Return Values

`teja_os_t` – Returns the found os instance or `NULL`.

### Description

Looks up an OS from its name in the software architecture.

## teja_lookup_process

### Function

`teja_process_t teja_lookup_process(const char *` *process-name*`);`

### Parameters

*process-name* – Name of the process.

### Return Values

`teja_process_t` – Returns the found process instance or `NULL`.

### Description

Looks up a process from its name in the software architecture.

## teja_lookup_thread

### Function

`teja_thread_t teja_lookup_thread(const char *` *thread-name*`);`

### Parameters

*thread-name* – Name of the thread.

### Return Values

`teja_thread_t` – Returns the found thread instance or `NULL`.

### Description

Looks up a thread from its name in the software architecture.

## teja_channel_declare

### Function

```
teja_channel_t teja_channel_declare(const char * name, const
char * type, teja_thread_t * producers, teja_thread_t * consumers);
```

### Parameters

*name* – Name of the channel.

*type* – Type of the channel.

*producers* – `NULL`-terminated list of producer thread instances.

*consumers* – `NULL`-terminated list of consumer thread instances.

### Return Values

`teja_channel_t` – Returns the created channel instance.

### Description

Creates a new channel instance in the software architecture. The instance is accessed in the user-application code as a C preprocessor symbol with the same name as specified in this function.

## teja_channel_set_property

### Function

```
int teja_channel_set_property(teja_channel_t channel, const char
* property-name, const char * value);
```

### Parameters

*channel* – Channel instance for which the property is being set.

*property-name* – Name of the property.

*value* – Value of the property.

### Return Values

`int` – Returns `TEJA_SUCCESS` or error code for failure.

### Description

Sets a new value for a channel property.

## teja_channel_get_property

### Function

```
int teja_channel_get_property(teja_channel_t channel, const char
* property-name, const char * value, int buf-size);
```

### Parameters

*channel* – Channel instance for which the property is read.

*property-name* – Name of the property.

*value* – Returned value of the property.

*buf-size* – Size of the character buffer that was passed for the return value.

### Return Values

`int` – Returns the length of the current value of the property.

### Description

Returns the current value of a channel property. If the returned value+1 is greater than the size of the passed buffer, the returned value is truncated. You must allocate a buffer with enough space to hold the value (returned value+1) and call the function again.

## teja_memory_pool_declare

### Function

```
teja_memory_pool_t teja_memory_pool_declare(const char * name,
const char * type, int num-nodes, int node-size, teja_thread_t *
getters, teja_thread_t * setters, const char * memory-bank);
```

### Parameters

*name* – Name of the memory pool.

*type* – Type of the memory pool.

*num-nodes* – Number of nodes to allocate in the memory pool.

*node-size* – Size in bytes for each node.

*getters* – `NULL`-terminated list of getter threads.

*setters* – `NULL`-terminated list of setter threads.

*memory-bank* – Name of the memory bank (in the hardware architecture) from which the memory is allocated.

### Return Values

`teja_memory_pool_t` – Returns the memory pool instance.

### Description

Creates a new memory pool instance in the software architecture.

## teja_memory_pool_set_property

### Function

`int teja_memory_pool_set_property(teja_memory_pool_t` *memory-pool*`, const char *` *property-name*`, const char *` *value*`);`

### Parameters

*memory-pool* – Memory pool instance for which the property is being set.

*property-name* – Name of the property.

*value* – Value of the property.

### Return Values

`int` – Returns `TEJA_SUCCESS` or error code for failure.

### Description

Sets a new value for a memory pool property.

## teja_memory_pool_get_property

### Function

`int teja_memory_pool_get_property(teja_memory_pool_t` *memory-pool*`, const char *` *property-name*`, const char *` *value*`, int` *buf-size*`);`

### Parameters

*memory-pool* – Memory pool instance for which the property is being read.

*property_name* – Name of the property.

*value* – Value of the property.

*buf-size* – Size of the character buffer that was passed for the return value.

### Return Values

`int` – Returns the length of the current value of the property.

### Description

Returns the current value of a memory pool property. If the returned value+1 is greater than the size of the passed buffer, the returned value is truncated. You must allocate a buffer with enough space to hold the value (returned value+1) and call the function again.

## teja_queue_declare

### Function

`teja_queue_t teja_queue_declare(const char * name, const char * type, teja_thread_t * enqueuers, teja_thread_t * dequeuers);`

### Parameters

*name* – Name of the queue.

*type* – Type of the queue.

*enqueuers* – `NULL`-terminated list of enqueuers threads.

*dequeuers* – `NULL`-terminated list of dequeuers threads.

### Return Values

`teja_queue_t` – Returns queue instance.

### Description

Creates a new queue instance in the software architecture.

## teja_queue_set_property

### Function

`int teja_queue_set_property(teja_queue_t queue, const char * property-name, const char * value);`

### Parameters

*queue* – Queue instance for which the property is being set.

*property-name* – Name of the property.

*value* – Value of the property.

### Return Values

`int` – Returns `TEJA_SUCCESS` or error code for failure.

### Description

Sets a new value for a queue property.

## teja_queue_get_property

### Function

```
int teja_queue_get_property(teja_queue_t queue, const char *
property-name, const char * value, int buf-size);
```

### Parameters

*queue* – Queue instance for which the property is being read.

*property-name* – Name of the property.

*value* – Returned value of the property.

*buf-size* – Size of the character buffer that was passed for the return value.

### Return Values

`int` – Returns the length of the current value of the property.

### Description

Returns current value of a queue property. If the returned value+1 is greater than the size of the passed buffer, the returned value is truncated. You must allocate a buffer with enough space to hold the value (returned value+1) and call the function again.

## teja_mutex_declare

### Function

```
teja_mutex_t teja_mutex_declare(const char * name, const char *
type, teja_thread_t * users);
```

### Parameters

*name* – Name of the mutex.

*type* – Type of the mutex.

*users* – `NULL`-terminated list of user-threads.

### Return Values

`teja_mutex_t` – Returns a new mutex instance.

### Description

Creates a new mutex instance in the software architecture.

## teja_mutex_set_property

### Function

`int teja_mutex_set_property(teja_mutex_t` *mutex*`, const char *` *property-name*`, const char *` *value*`);`

### Parameters

*mutex* – Mutex instance for which the property is being set.

*property-name* – Name of the property.

*value* – Value of the property.

### Return Values

`int` – Returns `TEJA_SUCCESS` or error code for failure.

### Description

Sets a new value for a mutex property.

## teja_mutex_get_property

### Function

`int teja_mutex_get_property(teja_mutex_t` *mutex*`, const char *` *property-name*`, const char *` *value*`, int` *buf-size*`);`

### Parameters

*mutex* – Mutex instance for which the property is being read.

*property-name* – Name of the property.

*value* – Returned value of the property.

*buf-size* – Size of the character buffer that was passed for the return value.

### Return Values

`int` – Returns the length of the current value of the property.

### Description

Returns a current value of a mutex property. If the returned value+1 is greater than the size of the passed buffer, the returned value is truncated. You must allocate a buffer with enough space to hold the value (returned value+1) and call the function again.

## teja_lookup_channel

### Function

`teja_channel_t teja_lookup_channel(const char *` *channel-name*`);`

### Parameters

*channel-name* – Name of the channel.

### Return Values

`teja_channel_t` – Returns the found channel instance or `NULL`.

### Description

Looks up the channel instance in the software architecture using the channel name as a key.

## teja_lookup_memory_pool

### Function

`teja_memory_pool_t teja_lookup_memory_pool(const char *` *memory-pool-name*`);`

### Parameters

*memory-pool-name* – Name of the memory pool.

### Return Values

`teja_memory_pool_t` – Returns the found memory pool instance or `NULL`.

### Description

Looks up the memory pool instance in the software architecture using the memory pool name as a key.

## teja_lookup_queue

### Function

```
teja_queue_t teja_lookup_queue(const char * queue-name);
```

### Parameters

*queue-name* – Name of the queue.

### Return Values

`teja_queue_t` – Returns the found queue instance or `NULL`.

### Description

Looks up the queue instance in the software architecture using the queue name as a key.

## teja_lookup_mutex

### Function

```
teja_mutex_t teja_lookup_mutex(const char * mutex-name);
```

### Parameters

*mutex-name* – Name of the mutex.

### Return Values

`teja_mutex_t` – Returns the found mutex instance or `NULL`.

### Description

Looks up the mutex instance in the software architecture using the mutex name as a key.

## teja_process_add_symbol

### Function

```
int teja_process_add_symbol(teja_process_t process, const char *
symbol, const char * value);
```

### Parameters

*process* – Process instance for which the symbol is being defined.

*symbol* – String that represents the symbol.

*value* – String that represents the value for the symbol.

### Return Values

`int` – Returns `TEJA_SUCCESS` or error code for failure.

### Description

Adds a definition of symbol in the process same as passing `-D` option on the command line.

# Map API

The Map API is used to describe the mapping between user-application source objects (functions and variables) and hardware architecture or software architecture.

The `teja_mapping.h` file contains the declaration of the Map data types and API functions.

# Map API Data Types

The following data type is used in the map definitions.

**TABLE 2-3** Map API Data Type

| | |
|---|---|
| `teja_mapping_t` | Every mapping returns a handle of type `teja_mapping_t`. |

# Map API Functions

## `teja_map_function_to_thread`

### Function

`teja_mapping_t teja_map_function_to_thread(const char *` *function-name*`, const char *` *thread-name*`);`

### Parameters

*function-name* – Name of the function from the source files.

*thread-name* – Name of the thread.

### Return Values

`teja_mapping_t` – Returns a handle that represents this mapping.

### Description

Maps a function to run on a thread.

## `teja_map_variable_to_memory`

### Function

`teja_mapping_t teja_map_variable_to_memory(const char *` *var-name*`, const char *` *memory-name*`);`

### Parameters

*var-name* – Name of the variable from the source files.

*memory-name* – Name of the memory bank.

### Return Values

`teja_mapping_t` – Returns a handle that represents this mapping.

### Description

Maps a variable to memory.

## teja_alias_variable

### Function

`teja_mapping_t teja_alias_variable(const char * var-name, const char * target-var-name);`

### Parameters

*var-name* – Name of the variable from the source files.

*target-var-name* – Name of the variable that the var_name maps to.

### Return Values

`teja_mapping_t` – Returns a handle that represents this mapping.

### Description

Creates an alias for a variable. This function helps in mapping two or more variables from different sources to the same location in memory. You map any one of these variables to a memory bank using `teja_map_variable_to_memory`. The remaining variables are mapped to that variable using this function.

## teja_map_variables_to_memory

### Function

`teja_mapping_t * teja_map_variables_to_memory(const char * var-regexp, const char * memory-name);`

### Parameters

*var-regexp* – Regular expression that results in one or more variables from the source files.

*memory-name* – Name of the memory bank to map.

### Return Values

`teja_mapping_t *` – Returns an array of handles that represents this mapping.

### Description

Maps one or more variables to memory using a regular expression. A regular expression can result in one or more variables from the source files. All the resultant variables are mapped to the memory bank.

## teja_map_initialization_function_to_process

### Function

`teja_mapping_t teja_map_initialization_function_to_process(const char *` *function*`, const char *` *process*`);`

### Parameters

*function* – Name of the function.

*process* – Name of the process as defined in the software architecture.

### Return Values

`teja_mapping_t` – Returns a handle that represents this mapping.

### Description

Maps an initialization function to the process. This function is executed before any thread starts execution.

## teja_mapping_set_property

### Function

`int teja_mapping_set_property(teja_mapping_t` *mapping-handle*`, const char *` *property-name*`, const char *` *value*`);`

### Parameters

*mapping-handle* – Mapping handle for which the property is being set.

*property-name* – Name of the property.

*value* – Value of the property.

### Return Values

`int` – Returns `TEJA_SUCCESS` or error code for failure.

### Description

Sets a new value for a mapping.

# Error-Handling API

The Error-Handling API can be used to provide a user-defined function or behavior when an error occurrs in the configuration API. The Error-Handling API is not available for the User API. The `teja_error.h` file contains the declaration of the error-handling data types and API functions.

## Error-Handling API Data Types

The following data type is used in the error-handling definitions.

**TABLE 2-4** Error-Handling Data Types

| | |
|---|---|
| `teja_error_handler_` | Represents a type for the error handler function. |

## Error-Handling API Functions

### `teja_abort`

#### Function

`void teja_abort(int` *error-code*`, const char *` *error-message*`,);`

#### Parameters

*error-code* – Error code.

*error-message* – Error message.

### Return Values

`void` – Aborts the execution of the hardware architecture, software architecture, or mapping definition.

### Description

When this function is called, the control is transferred to the caller of the library entry point function, which reports the error appropriately. When executed from the command line, `tejacc` terminates returning the provided *error-code*.

## `teja_register_error_handler`

### Function

```
teja_error_handler_t
teja_register_error_handler(teja_error_handler_t handler);
```

### Parameters

*handler* – Error handler function.

### Return Values

`teja_error_handler_t` – The previously registered error handler.

### Description

Enables you to implement a custom behavior in case of errors. When an error is encountered during the execution of a Teja hardware architecture, software architecture, or mapping API function, the registered error handler function is called.

# Error Handler Function Prototype

```
int fn(int error_code, const char * error_msg);
```

The handler function is called with an error code and message, and returns a value. The value returned by the handler is in turn returned by the Teja API function that encountered the error. The default error handler does not return a value, but invokes `teja_abort()`, with the effect of transferring the control immediately to the caller

of the library entry point function. You can replace the default error handler using the `teja_register_error_handler()` function. For example, you can replace the default handler with one that just returns an error code as follows:

```
#define ERR_SHOULD_RETURN_NULL (TEJA_ERROR_CREATE_FAILED |
                                TEJA_ERROR_LOOKUP_FAILED)
int my_error_handler(int code, const char* msg) {
  if (code & ERR_SHOULD_RETURN_NULL) {
    /* code is an error during creation or lookup,
     * should return NULL rather than error code
     */
    return (int)NULL;
  else
    return code;
}

void entry_fn(void) {
  teja_register_error_handler(my_error_handler);
  ...
```

**Note –** The software architecture, hardware architecture, and mapping have three independent error handlers, so if you want to replace the default error handler, you must register the new one in each entry point function.

# CMT-Specific Hardware Architecture Constants

The `include/csp/sun/teja_cmt.h` file lists all the hardware object types and properties that are supported by CMP CSP.

# CMT-Specific Hardware Architecture Types

**TABLE 2-5** CMT-Specific Hardware Architecture Types

| Type | Name | Description |
|---|---|---|
| Architecture | TEJA_ARCHITECTURE_TYPE_CMT1_CHIP | Architecture type that represents the CMT chip. |
| | TEJA_ARCHITECTURE_TYPE_CMT1_BOARD | Architecture type that represents a board containing the CMT chip. |
| | TEJA_ARCHITECTURE_TYPE_USER_DEFINED | Architecture that represents a user-defined architecture that is not known to tejacc. |
| Processor | TEJA_PROCESSOR_TYPE_CMT1 | Processor type that represents a single strand in the CMT chip. |
| Memory | TEJA_MEMORY_TYPE_CMT1_DRAM | Memory type that represents the DRAM memory for the CMT architecture. |
| | TEJA_MEMORY_TYPE_OS_BASED | Memory type that represents OS-based memory in the architecture. |
| Bus | TEJA_BUS_TYPE_CMT1_DRAM | Bus type that represents the DRAM bus for the CMT architecture. |
| | TEJA_BUS_TYPE_OS_BASED | Bus type that represents a bus that connects OS-based memory to other objects. |
| | TEJA_BUS_TYPE_PCI | Bus type that represents PCI bus in the architecture. |

# CMT-Specific Hardware Architecture Properties

**TABLE 2-6** CMT-Specific Hardware Architecture Properties

| Property | Name | Description |
|---|---|---|
| Architecture | TEJA_PROPERTY_BSP_PATH | Sets the path to the board support package (BSP) located on the host machine. There is no default value set. |
| Processor | TEJA_PROPERTY_CLOCK_FREQUENCY | Sets the clock frequency of the processor. There is no default value set. |
| Memory | TEJA_PROPERTY_MEMORY_SIZE | Sets the size of the memory in bytes. The default value is 256. |

**TABLE 2-6** CMT-Specific Hardware Architecture Properties

| Property | Name | Description |
|---|---|---|
| | TEJA_PROPERTY_MEMORY_OFFSET | Sets the offset from where the memory is available for the user application. The default value is 0. |
| | TEJA_PROPERTY_MEMORY_PHYSICAL_ADDRESS | Sets the actual physical base address that is used to access the memory. The default value is 0. |
| | TEJA_PROPERTY_MEMORY_BIT_ALIGNMENT | Sets the alignment of the memory in bits. The default value is 32. |
| | TEJA_PROPERTY_MEMORY_RESERVE_WORD_0 | When set to `true`, makes location 0 non writable. The default value is `true`. |
| | TEJA_PROPERTY_MEMORY_IS_OS_BASED | When set to `true`, marks the memory OS-based. The default value is `false`. |
| | TEJA_PROPERTY_MEMORY_NO_ADDRESS_CONVERSION | When set to `true`, hal conversion is necessary. The default value is `true`. |

# CMT-Specific Software Architecture Constants

## CMT-Specific Software Architecture Types

**TABLE 2-7** CMT-Specific Software Architecture Types

| Type | Name | Description |
|---|---|---|
| OS | TEJA_OS_TYPE_RAW | This is the only type of OS that is supported for CMT CSP. |
| Channel | TEJA_GENERIC_CHANNEL_SHARED_MEMORY_OS_BASED | Channel type that uses OS-based shared memory implementation. |

**TABLE 2-7** CMT-Specific Software Architecture Types *(Continued)*

| Type | Name | Description |
|---|---|---|
| | TEJA_GENERIC_CHANNEL_SHARED_MEMORY | Channel type that uses non-OS-based shared memory implementation. |
| Memory pool | TEJA_GENERIC_MEMORY_POOL_SHARED_MEMORY_OS_BASED | Memory Pool type that uses OS-based shared memory implementation. |
| | TEJA_GENERIC_MEMORY_POOL_SHARED_MEMORY | Memory Pool type that uses non-OS-based shared memory implementation. |
| Queue | TEJA_GENERIC_QUEUE_SHARED_MEMORY_OS_BASED | Queue type that uses OS-based shared memory implementation. |
| | TEJA_GENERIC_QUEUE_SHARED_MEMORY | Queue type that uses non-OS-based shared memory implementation. |
| Mutex | TEJA_GENERIC_MUTEX_SHARED_MEMORY_OS_BASED | Mutex type that uses OS-based shared memory implementation. |
| | TEJA_CMT1_MUTEX_SPINLOCK | Mutex type that uses spin lock implementation. |

# CMT-Specific Software Architecture Properties

**TABLE 2-8** CMT-Specific Software Architecture Properties

| Property | Name | Description |
|---|---|---|
| Thread | TEJA_PROPERTY_THREAD_ASSIGN_TO_PROCESSOR | Enables you to assign a thread to a specific processor (hardware thread). Specify the processor using a fully qualified name from the hardware architecture. The default value for this property is NULL so the thread is not assigned to any specific processor by default. |
| Channel | TEJA_PROPERTY_CHANNEL_BUFFER_SIZE | Sets the buffer size for the circular buffer size. The default value is 1024. |
| Memory Pool | TEJA_PROPERTY_MEMORY_POOL_ALIGNMENT | Sets the alignment for the memory pool nodes. The default value is 32. |

CHAPTER 3

# Teja Profiler API

This chapter describes the components and functions of the Teja Profiler API. Topics in this chapter include:


- "Teja Profiler API Configuration" on page 111
- "Teja Profiler API" on page 112
- "CMT-Specific Profiler Constants" on page 115


## Teja Profiler API Configuration

You can set two properties for a process in the software architecture. These properties are configured per process and applied to all threads of that process.

**TABLE 3-1** Process Properties

| Property | Description |
|---|---|
| `profiler_log_table_size` | Sets the total number of profile records in the log. The default value is 1024. |
| `profiler_user_data_size` | Represents the maximum number of user-data in 64-bit words that user wants to log along with the profile record. The default value is 0. |

# Teja Profiler API

## Teja Profiler API Data Types

**TABLE 3-2** Teja Profiler API Data Types

| Data Type | Description |
|---|---|
| `teja_profiler_group_t;` | Represents a group of events. For example, events regarding instructions and cache hit or miss in one group, while memory related events can be in another group. Groups are target-specific and available to you in preprocessor define forms. |
| `teja_profiler_event_t;` | Represents what needs to be measured in a specific group. Group and event combinations make an unique event. Each bit in the 64-bit value represents a different event so more that one event can be specified using an event mask. |
| `teja_profiler_value_t;` | Type for the value of the event. This is the type for the actual value that is being measured. |
| `TEJA_PROFILER_MAX_EVENTS` | Max number of events that can be measured per group. This value is target-dependent. |
| `teja_profiler_values_t;` | Type for the values of the events. The events array contains the values of the events in the same group. For example:<br>`typedef struct teja_profiler_values_t`<br>`  uint64_t events`<br>`[TEJA_PROFILER_MAX_EVENTS];` |

## Teja Profiler API Functions

### `teja_profiler_start`

#### Function

`int teja_profiler_start(const teja_profiler_group_t` *group*`, const teja_profiler_event_t` *event*`);`

#### Parameters

*group* – ID of the group for to start collecting profiler data.

*event* – Events of the group as a bit mask.

#### Return Values

`int` – `0` for success and `-1` for error.

#### Description

Starts collecting profile data for the specified events in the specified group. More than one event can be specified as a bit mask. Only one group is allowed. If you want to start profiling more than one group, you must invoke the same function multiple times.

### teja_profiler_stop

#### Function

```
int teja_profiler_stop(const teja_profiler_group_t group);
```

#### Parameters

*group* – ID of the group to stop collecting profiler data.

#### Return Values

`int` – `0` for success and `-1` for error.

#### Description

Stops collecting profile data for all events in the specified group.

### teja_profiler_update

#### Function

```
int teja_profiler_update(const teja_profiler_group_t group,
...);
```

#### Parameters

*group* – ID of the group for which you want to update profile data.

### Return Values

`int` – `0` for success and `-1` for error.

### Description

Takes a snapshot of the current profiling data and saves the snapshot in the log. All the events that were specified for the group with the `teja_profiler_start` are updated. User-defined data that needs to be logged with the profiler log entry can be specified using variable arguments. The maximum number of arguments is specified in the software architecture using the process property.

## teja_profiler_get_value

### Function

```
int teja_profiler_get_value(const teja_profiler_group_t group,
teja_profiler_values_t * values);
```

### Parameters

*group* – ID of the group for which you want to get the profiler data.

*values* – User-allocated data structure that is filled with the profiler data.

### Return Values

`int` – Returns overflow information or `-1` for error.

### Description

Takes a snapshot of the current profiling data and returns the values. All the events that were specified for the group with the `teja_profiler_start` are returned.

## teja_profiler_dump

### Function

```
int teja_profiler_dump(teja_thread_t thread);
```

### Parameters

*thread* – Thread identifier for which the profiler dump is requested.

### Return Values

`int` – Returns `0` for success and `-1` for error.

### Description

Dumps the profile data the `stdout`. The profiler data represents the profiler records that are collected so far for the thread identifier.

# CMT-Specific Profiler Constants

## CMT-Specific Profiler Groups

**TABLE 3-3** CMT_Specific Profiler Groups

| Event or Group | Event or Description | Description |
|---|---|---|
| `TEJA_PROFILER_CMT_CPU (0x1)` | Captures events related to CPU and cache. The events measured in this group are per CPU strand. The following events are available for this group. The completed instructions count is always an available event for this group. There is additionally one more event that can be measured along with instructions count | |
| | `TEJA_PROFILER_CMT_CPU_SB_FULL (0x1)` | Measures number of store buffer full cycles. |
| | `TEJA_PROFILER_CMT_CPU_FP_INSTR_CNT (0x2)` | Measures number of floating point instructions. |
| | `TEJA_PROFILER_CMT_CPU_IC_MISS (0x4)` | Measures number of instruction cache misses. |
| | `TEJA_PROFILER_CMT_CPU_DC_MISS (0x8)` | Measures number of data cache misses. |
| | `TEJA_PROFILER_CMT_CPU_ITLB_MISS (0x10)` | Measures number of instruction TLB miss traps taken. |
| | `TEJA_PROFILER_CMT_CPU_DTLB_MISS (0x20)` | Measures number of data TLB miss traps taken. |
| | `TEJA_PROFILER_CMT_CPU_L2_IMISS (0x40)` | Measures number of secondary cache (L2) misses due to instruction cache requests. |

**TABLE 3-3** CMT_Specific Profiler Groups *(Continued)*

| Event or Group | Event or Description | Description |
|---|---|---|
| | TEJA_PROFILER_CMT_CPU_L2_DMISS_LD (0x80) | Measures number of secondary cache (L2) misses due to data cache load requests. |
| | TEJA_PROFILER_CMT_CPU_INSTR_COMPLETED (0x100) | Measures number of completed instructions. |
| TEJA_PROFILER_CMT_DRAM_CTL0 | This group captures events related to DRAM memory read, write, and queues. There are different groups for different DRAM controllers. The following events can be measured in this group. | |
| | TEJA_PROFILER_CMT_DRAM_MEM_READS (0x1) | Read transactions. |
| | TEJA_PROFILER_CMT_DRAM_MEM_WRITES (0x2) | Write transactions. |
| | TEJA_PROFILER_CMT_DRAM_MEM_READ_WRITE (0x4) | Read + write transactions. |
| | TEJA_PROFILER_CMT_DRAM_BANK_BUSY_STALLS (0x8) | Bank busy stalls. |
| | TEJA_PROFILER_CMT_DRAM_RD_QUEUE_LATENCY (0x10) | Read queue latency. |
| | TEJA_PROFILER_CMT_DRAM_WR_QUEUE_LATENCY (0x20) | Write queue latency. |
| | TEJA_PROFILER_CMT_DRAM_RW_QUEUE_LATENCY (0x40) | Read + write queue latency. |
| | TEJA_PROFILER_CMT_DRAM_WR_BUF_HITS (0x80) | Write-back buffer hits. |
| TEJA_PROFILER_CMT_DRAM_CTL1 | Measures same events as DRAM controller 0, but for DRAM controller 1. | |
| TEJA_PROFILER_CMT_DRAM_CTL2 | Measures same events as DRAM controller 0, but for DRAM controller 2. | |
| TEJA_PROFILER_CMT_DRAM_CTL3 | Measures same events as DRAM controller 0, but for DRAM controller 3. | |
| TEJA_PROFILER_CMT_JBUS | This group captures events related to JBus read, write, and cycles. Following events can be measured for this group. | |
| | TEJA_PROFILER_CMT_JBUS_CYCLES (0x1) | JBus cycles |
| | TEJA_PROFILER_CMT_JBUS_DMA_READS (0x2) | DMA read transactions (Inbound) |
| | TEJA_PROFILER_CMT_JBUS_DMA_READ_LATENCY (0x4) | Total DMA read latency |

**TABLE 3-3** CMT_Specific Profiler Groups *(Continued)*

| Event or Group | Event or Description | Description |
|---|---|---|
| | TEJA_PROFILER_CMT_JBUS_DMA_WRITES (0x8) | DMA write transactions |
| | TEJA_PROFILER_CMT_JBUS_DMA_WRITE8 (0x10) | DMA WR8 subtransactions |
| | TEJA_PROFILER_CMT_JBUS_ORDERING_WAITS (0x20) | Ordering waits |
| | TEJA_PROFILER_CMT_JBUS_PIO_READS (0x40) | PIO read transactions (outbound) |
| | TEJA_PROFILER_CMT_JBUS_PIO_READ_LATENCY (0x80) | Total PIO read latency |
| | TEJA_PROFILER_CMT_JBUS_AOK_DOK_OFF_CYCLES (0x100) | AOK_OFF or DOK_OFF seen (cycles) |
| | TEJA_PROFILER_CMT_JBUS_AOK_OFF_CYCLES (0x200) | AOK_OFF seen (cycles) |
| | TEJA_PROFILER_CMT_JBUS_DOK_OFF_CYCLES (0x400) | DOK_OFF seen (cycles) |