Sun Netra Data Plane Software Suite 2.1 Update 1

Reference Manual

Part No. 820-5156-11
April 2010, Revision A

Adobe PostScript

Contents

Figures

Tables

Preface

This reference manual provides detailed information about the various functions and parameters of the application programming interfaces (API). This document is technical, and written for developers who need to know the behavior of the software.

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
- Oracle's Solaris Operating System documentation, which is at:

  `http://docs.sun.com`

Related Documentation

The following table lists the documentation for this product. The online documentation is available at:

http://docs.sun.com/app/docs/prod/netra.dp

Application	Title	Part Number	Format	Location
Operation	*Sun Netra Data Plane Software Suite 2.1 Update 1 User Guide*	820-5154-11	PDF	online
Reference	*Sun Netra Data Plane Software Suite 2.1 Update 1 Reference Manual*	820-5156-11	PDF	online
Last-minute information	*Sun Netra Data Plane Software Suite 2.1 Update 1 Release Notes*	820-5157-11	PDF	online
Documentation Location	*Sun Netra Data Plane Software Suite 2.1 Update 1 Getting Started Guide*	820-5158-11	PDF	online

Documentation, Support, and Training

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

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:

Sun Netra Data Plane Software Suite 2.1 Update 1 Reference Manual,
part number 820-5156-11

CHAPTER 1

Configuration API

This chapter describes the components and functions of the Configuration API. Topics include:

Hardware Architecture API

You can use the hardware architecture API 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 1-1 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: • `TEJA_INTERNAL_BUS` - bus not visible outside its containing architecture • `TEJA_EXPORTED_BUS` - bus made visible outside its containing architecture Example: `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

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.

Syntax

`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

teja_architecture_set_property

Description

Sets the value of the property for the architecture object.

Syntax

`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.

teja_architecture_get_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. The user must allocate a buffer with enough space to hold the value (returned value+1) and call the function again.

Syntax

```
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.

teja_architecture_set_read_only

Description

Prevents modification of given architecture by subsequent processing.

Syntax

```
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.

teja_processor_create

Description

Creates a processor object.

Syntax

```
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.

teja_processor_set_property

Description

Sets the value of the property for the processor object.

Syntax

```
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.

teja_processor_get_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. The user must allocate a buffer with enough space to hold the value (returned value+1) and call the function again.

Syntax

```
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.

teja_processor_add_preprocessor_symbol

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.

Syntax

```
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.

teja_memory_create

Description

Creates a memory object

Syntax

```
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.

teja_memory_set_property

Description

Sets the value of the property for the memory object.

Syntax

```
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.

teja_memory_get_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. The user must allocate a buffer with enough space to hold the value (returned value+1) and call the function again.

Syntax

```
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.

teja_bus_create

Description

Creates a bus object.

Syntax

```
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.

teja_bus_set_property

Description

Sets the value of the property for the bus object.

Syntax

```
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.

teja_bus_get_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. The user must allocate a buffer with enough space to hold the value (returned value+1) and call the function again.

Syntax

```
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.

teja_hardware_object_create

Description

Creates a hardware object.

Syntax

```
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.

teja_hardware_object_set_property

Description

Sets the value of the property for the hardware object.

Syntax

```
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.

teja_hardware_object_get_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. The user must allocate a buffer with enough space to hold the value (returned value+1) and call the function again.

Syntax

```
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.

teja_architecture_connect

Description

Connects an architecture to a bus.

Syntax

```
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.

teja_processor_connect

Description

Connects a processor to a bus.

Syntax

```
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.

teja_memory_connect

Description

Connects a memory to a bus.

Syntax

```
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-nam – 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.

teja_hardware_object_connect

Description

Connects a hardware object to a bus.

Syntax

```
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.

teja_lookup_architecture

Description

Looks up an architecture in the container.

Syntax

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

Parameters

architecture – Container architecture in which the user wants 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.

teja_lookup_processor

Description

Looks up a processor in the container.

Syntax

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

Parameters

architecture – Container architecture in which the user wants 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.

teja_lookup_memory

Description

Looks up a memory in the container.

Syntax

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

Parameters

architecture – Container architecture in which the user wants 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.

teja_lookup_bus

Description

Looks up a bus in the container.

Syntax

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

Parameters

architecture – Container architecture in which the user wants 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.

teja_lookup_hardware_object

Description

Looks up a hardware object in the container.

Syntax

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

Parameters

architecture – Container architecture in which the user wants 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.

teja_port_create

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 17. The port can also be connected internally to objects contained in this architecture. See "teja_architecture_set_port" on page 17.

Syntax

```
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.

teja_architecture_set_port

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.

Syntax

```
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.

teja_architecture_set_port_internal

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.

Syntax

```
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.

teja_processor_set_port

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.

Syntax

```
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.

teja_memory_set_port

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.

Syntax

```
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.

teja_hardware_object_set_port

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.

Syntax

```
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.

teja_bus_set_port

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.

Syntax

```
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.

teja_port_add_property

Description

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

Syntax

```
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.

teja_architecture_get_parent

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.

Syntax

```
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.

teja_processor_get_parent

Description

Returns the architecture containing the specified processor.

Syntax

```
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.

teja_bus_get_parent

Description

Returns the architecture containing the specified processor.

Syntax

```
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.

teja_memory_get_parent

Description

Returns the architecture containing the specified memory.

Syntax

```
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.

teja_hardware_object_get_parent

Description

Returns the architecture containing the specified hardware object.

Syntax

```
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.

teja_architecture_get_processors

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`. The user must deallocate the array by calling `free()` on it.

Syntax

```
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.

teja_architecture_get_memories

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`. The user must deallocate the array by calling `free()` on it.

Syntax

```
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.

teja_architecture_get_hardware_objects

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`. The user must deallocate the array by calling `free()` on it.

Syntax

```
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.

teja_architecture_get_busses

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`. The user must deallocate the array by calling `free()` on it.

Syntax

```
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.

teja_architecture_get_architectures

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`. The user must deallocate the array by calling `free()` on it.

Syntax

```
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.

teja_processor_get_connected_bus

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.

Syntax

```
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`.

teja_memory_get_connected_bus

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.

Syntax

```
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`.

teja_hardware_object_get_connected_bus

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.

Syntax

```
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.

teja_architecture_get_connected_bus

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.

Syntax

```
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`.

teja_bus_get_connected_processors

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`. The user must deallocate the array by calling `free()` on it.

Syntax

```
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.

teja_bus_get_connected_memories

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`. The user must deallocate the array by calling `free()` on it.

Syntax

```
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.

teja_bus_get_connected_hardware_objects

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`. The user must deallocate the array by calling `free()` on it.

Syntax

```
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.

teja_bus_get_connected_architectures

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`. The user must deallocate the array by calling `free()` on it.

Syntax

```
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.

teja_processor_get_busses

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`. The user must deallocate the array by calling `free()` on it.

Syntax

```
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.

teja_memory_get_busses

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`. The user must deallocate the array by calling `free()` on it.

Syntax

```
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.

teja_hardware_object_get_busses

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`. The user must deallocate the array by calling `free()` on it.

Syntax

```
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.

teja_address_space_create

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`).

Syntax

```
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.

teja_address_space_join

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.

Syntax

```
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.

teja_address_range_create_absolute

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.

Syntax

```
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.

teja_address_range_create_aligned

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.

Syntax

```
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.

teja_address_range_create_generic

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.

Syntax

```
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.

teja_address_range_get_lower_bound

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.

Syntax

```
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.

teja_address_range_get_upper_bound

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.

Syntax

```
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.

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 Sun Netra DPS 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 1-2 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`

Description

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

Syntax

```
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.

teja_os_set_property

Description

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

Syntax

```
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.

teja_os_get_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. The user must allocate a buffer with enough space to hold the value (returned value+1) and call the function again.

Syntax

```
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.

teja_process_create

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.

Syntax

```
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.

teja_process_set_property

Description

Sets a process property.

Syntax

```
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.

teja_process_get_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. The user must allocate a buffer with enough space to hold the value (returned value+1) and call the function again.

Syntax

```
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.

teja_processor_add_preprocessor_symbol

See "teja_processor_add_preprocessor_symbol" on page 6

teja_thread_create

Description

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

Syntax

```
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.

teja_thread_set_property

Description

Sets a thread property.

Syntax

```
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.

teja_thread_get_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. The user must allocate a buffer with enough space to hold the value (returned value+1) and call the function again.

Syntax

```
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.

teja_lookup_os

Description

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

Syntax

```
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`.

teja_lookup_process

Description

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

Syntax

```
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`.

teja_lookup_thread

Description

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

Syntax

```
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`.

teja_channel_declare

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.

Syntax

```
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.

teja_channel_set_property

Description

Sets a new value for a channel property.

Syntax

```
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.

teja_channel_get_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. The user must allocate a buffer with enough space to hold the value (returned value+1) and call the function again.

Syntax

```
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.

teja_memory_pool_declare

Description

Creates a new memory pool instance in the software architecture.

Syntax

```
teja_memory_pool_t teja_memory_pool_declare(const char *name,
const char *type, int num-nodes, long 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.

teja_memory_pool_set_property

Description

Sets a new value for a memory pool property.

Syntax

```
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.

teja_memory_pool_get_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. The user must allocate a buffer with enough space to hold the value (returned value+1) and call the function again.

Syntax

```
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.

teja_queue_declare

Description

Creates a new queue instance in the software architecture.

Syntax

```
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.

teja_queue_set_property

Description

Sets a new value for a queue property.

Syntax

```
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.

teja_queue_get_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. The user must allocate a buffer with enough space to hold the value (returned value+1) and call the function again.

Syntax

```
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.

teja_mutex_declare

Description

Creates a new mutex instance in the software architecture.

Syntax

```
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.

teja_mutex_set_property

Description

Sets a new value for a mutex property.

Syntax

```
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.

teja_mutex_get_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. The user must allocate a buffer with enough space to hold the value (returned value+1) and call the function again.

Syntax

```
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.

teja_lookup_channel

Description

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

Syntax

```
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`.

teja_lookup_memory_pool

Description

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

Syntax

```
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`.

teja_lookup_queue

Description

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

Syntax

```
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`.

teja_lookup_mutex

Description

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

Syntax

```
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`.

teja_process_add_symbol*

Description

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

Syntax

```
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.

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 1-3 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

Description

Maps a function to run on a thread.

Syntax

```
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.

teja_map_variable_to_memory

Description

Maps a variable to memory.

Syntax

```
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.

teja_alias_variable

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. The user maps 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.

Syntax

```
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.

teja_map_variables_to_memory

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.

Syntax

```
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.

teja_map_initialization_function_to_process

Description

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

Syntax

```
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.

teja_mapping_set_property

Description

Sets a new value for a mapping.

Syntax

```
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.

Error–Handling API

The error-handling API can be used to provide a user-defined function or behavior when an error occurs 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 1-4 Error-Handling Data Types

`teja_error_handler_t`	Represents a type for the error handler function.

Error–Handling API Functions

`teja_abort`

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*.

Syntax

`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.

teja_register_error_handler

Description

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

Syntax

```
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.

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 Sun Netra DPS 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. The user can replace the

default error handler using the `teja_register_error_handler()` function. For example, the user 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 the user wants to replace the default error handler, the user 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 1-5 CMT-Specific Hardware Architecture Types

Type	Name	Description
Architecture	TEJA_ARCHITECTURE_TYPE_CMT1_CHIP	Architecture type that represents the CMT 1 chip.
	TEJA_ARCHITECTURE_TYPE_CMT2_CHIP	Architecture type that represents the CMT 2 chip.
	TEJA_ARCHITECTURE_TYPE_CMT1_BOARD	Architecture type that represents a board containing the CMT 1 chip.
	TEJA_ARCHITECTURE_TYPE_CMT2_BOARD	Architecture type that represents a board containing the CMT 2 chip.
	TEJA_ARCHITECTURE_TYPE_USER_DEFINED_	Architecture that represents a user-defined architecture that is not known to tejacc.d
Processor	TEJA_PROCESSOR_TYPE_CMT1	Processor type that represents a single strand in the CMT 1 chip.
	TEJA_PROCESSOR_TYPE_CMT2	Processor type that represents a single strand in the CMT 2 chip.
Memory	TEJA_MEMORY_TYPE_CMT1_DRAM	Memory type that represents the DRAM memory for the CMT 1 architecture.
	TEJA_MEMORY_TYPE_CMT2_DRAM	Memory type that represents the DRAM memory for the CMT 2 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 1 architecture.
	TEJA_BUS_TYPE_CMT2_DRAM	Bus type that represents the DRAM bus for the CMT 2 based shared memory stack implementation.
	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 1-6 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.
	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 1-7 CMT-Specific Software Architecture Types

Type	Name	Description
OS	TEJA_OS_TYPE_RAW	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.
	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.
	TEJA_GENERIC_MEMORY_POOL_SHARED_MEMORY_CIRCULAR_BUFFER_OS_BASED	Memory pool type that uses an OS based shared memory circular buffer implementation
	TEJA_GENERIC_MEMORY_POOL_SHARED_MEMORY_CIRCULAR_BUFFER	Memory pool type that uses a non-OS-based shared memory circular buffer 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.

TABLE 1-7 CMT-Specific Software Architecture Types *(Continued)*

Type	Name	Description
Mutex	TEJA_GENERIC_MUTEX_SHARED_MEMORY_OS_BASED	Mutex type that uses OS-based shared memory implementation.
	TEJA_CMT_MUTEX_SPINLOCK	Mutex type that uses spin lock implementation.

CMT–Specific Software Architecture Properties

TABLE 1-8 CMT-Specific Software Architecture Properties

Property	Name	Description
Thread	TEJA_PROPERTY_THREAD_ASSIGN_TO_PROCESSOR	Enables the user 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 2

User API

This chapter describes the User API which consists of functions the user can deploy in the application code. This API consists of three main parts:

Additional information is provided in:

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 *Sun Netra Data Plane Software Suite 2.1 Update 1 User Guide* for an overview of this API.

Late-Binding API Data Types

TABLE 2-1 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 2-2 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 Sun Netra DPS 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`

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.

Syntax

```
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`.

Example

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

teja_mutex_trylock

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.

Syntax

`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`.

Example

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

teja_mutex_unlock

Description

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

Syntax

`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`.

Example

See the examples in "teja_mutex_lock" on page 64 and "teja_mutex_trylock" on page 65.

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

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.

Syntax

`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`.

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

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.

Syntax

```
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.

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

Description

Tests to see if the queue is empty.

Syntax

```
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.

Example

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

teja_queue_get_size

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 computation intensive and not meant for fast path operation.

Syntax

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

Parameters

queue – Queue to obtain size for.

Return Values

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

Example

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

Late-Binding API Memory Pool Functions

teja_memory_pool_get_node

Description

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

Syntax

`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.

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

Description

Frees a node back to a memory pool.

Syntax

```
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`.

Example

See the example in "teja_memory_pool_get_node" on page 70.

teja_memory_pool_get_node_from_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.

Syntax

```
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.

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

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.

Syntax

`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.

Example

See the example in "teja_memory_pool_get_node_from_index" on page 71.

Late-Binding API Channel Functions

teja_channel_is_connection_open

Description

Returns 1 if the connection is open, 0 if the connection is closed.

Syntax

```
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 if the connection is closed.

Example

See the example in "teja_channel_send" on page 74.

teja_channel_make_connection

Description

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

Syntax

```
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.

Example

See the example in "teja_channel_send" on page 74.

teja_channel_break_connection

Description

Breaks an existing connection on the given channel.

Syntax

```
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.

Example

See the example in "teja_channel_send" on page 74.

teja_channel_send

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.

Syntax

```
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.

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 channeling”);
      }
  }
```

See also the example in “`teja_wait`” on page 76, 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 Sun Netra DPS signals and by registered file descriptors.

teja_wait

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. Channels are checked once before starting the timeout wait.

Syntax

`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 `NULL` terminated.

Return Values

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

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.

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.

Note – This function may not be invoked at initialization time.

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 74, which shows how to send data.

Sun Netra DPS Runtime API

The Sun Netra DPS Runtime 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, Sun Netra DPS Runtime APIs are not treated specially by the compiler and are implemented in precompiled libraries. See the *Sun Netra Data Plane Software Suite 2.1 Update 1 User Guide* for an overview of this API.

Sun Netra DPS Runtime API Data Types

TABLE 2-3 Sun Netra DPS Runtime 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 2-3 Sun Netra DPS Runtime 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 2-4 Sun Netra DPS Runtime API Macros

Macros	Description
`TEJA_DEFAULT_STACK_SIZE`	Default stack size for newly started threads (for example, only software architecture threads).
`TEJA_IS_RAW_OS`	Informs the user that running on an OS or in bare hw mode. In NDPS it is always true.

Sun Netra DPS Runtime API Memory Management Functions

The memory management functions offer `malloc` and `free` functionality. These functions are computation 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`

Description

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

Syntax

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

Parameters

ptr – Pointer to buffer to free.

Return Values

void

teja_malloc

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.

Syntax

```
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.

teja_realloc

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.

Syntax

```
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.

Sun Netra DPS Runtime 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. 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 2-5 Sun Netra DPS Runtime 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`

Description

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

Syntax

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

Return Values

`teja_thread_t` – Thread ID of the current thread.

teja_get_thread_name_for_id

Description

Returns the name of the given thread.

Syntax

```
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.

teja_get_id_for_thread_name

Description

Returns the ID of the given thread.

Syntax

```
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.

teja_thread_handle_start

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.

Syntax

```
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. predefined value is `TEJA_DEFAULT_PRIORITY`.

Return Values

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

teja_thread_handle_end

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.

Syntax

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

Return Values

`void`

teja_thread_handle_get_for_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.

Syntax

```
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.

Sun Netra DPS Runtime API Miscellaneous Functions

teja_thread_shutdown

Description

Shuts down the current Sun Netra DPS thread.

Note – This function may not be invoked at initialization time.

Syntax

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

Return Values

`void`

Sun Netra DPS Runtime API Time Functions

teja_get_time

Description

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

Syntax

```
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.

teja_wait_time

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.

Note – This function is implemented as a macro on top of `teja_wait()`. This function may not be invoked at initialization time.

Syntax

```
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.

teja_os_wait

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 to reschedule the thread. Unlike `teja_wait_time()` this function is not implemented on top of `teja_wait()`.

Syntax

`int teja_os_wait(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 contained between 0 and 999999999.

Return Values

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

Miscellaneous Functions

teja_get_argc

Description

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

Syntax

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

Return Values

```
int
```

teja_get_argv

Description

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

Syntax

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

Return Values

```
char **
```

Finite State Automata API

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

These functions are available in different versions:

- Single-context versus multiple-context
- Computed goto versus 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. The user 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.

The user 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 2-6 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

Description

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

Syntax

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

Parameters

name – Name of the state machine.

teja_fsm_begin

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.

Syntax

`#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.

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.

Syntax

```
#define teja_fsm_end()
```

teja_fsm_start

Description

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

Syntax

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

Parameters

name – Name of the state machine.

teja_fsm_state_declare

Description

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

Syntax

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

Parameters

name – Name of the state.

teja_fsm_state_begin

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. The user 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.

Syntax

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

Parameters

name – Name of the state.

teja_fsm_state_end

Description

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

Syntax

```
#define teja_fsm_state_end()
```

teja_fsm_goto_state

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.

Syntax

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

Parameters

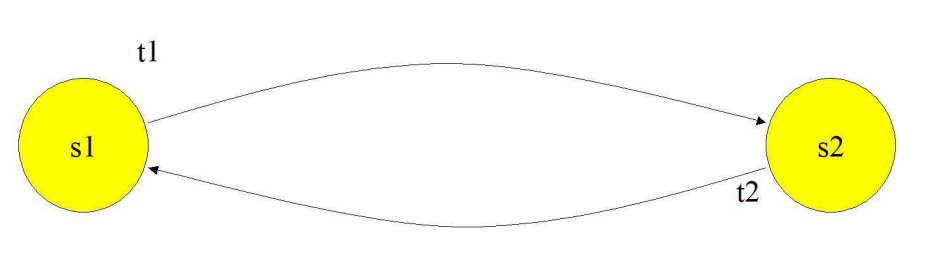
name – Name of the state to jump to.

FSM Example

EXAMPLE 2-1 implements a simple state machine depicted in FIGURE 2-1.

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

FIGURE 2-1 Finite State Machine Example



EXAMPLE 2-1 Finite State Machine Code Example

```
#include <stdio.h>
#if NUM_CONTEXTS == 1
#define TEJA_FSM_SINGLE_CONTEXT
#endif
#include "fsm/teja_fsm.h"
```

EXAMPLE 2-1 Finite State Machine Code Example *(Continued)*

```
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);
    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)
```

EXAMPLE 2-1 Finite State Machine Code Example *(Continued)*

```
{
  int i;
  for (i = 0; i < NUM_CONTEXTS; i++)
    {
      contexts[i].count = 0;
    }
  teja_fsm_start (my_fsm);
}
```

Hardware Specific Miscellaneous Functions

teja_os_get_timer

Description

Returns the value of the clock tick register.

Syntax

```
uint64_t teja_os_get_timer(void);
```

Return Values

`int` – Returns the value of the clock tick register.

C Library Support on Bare Hardware

Sun Netra DPS programs running on bare hardware CMT can use the following standard C library functions:

- `atoi`
- `bcopy`
- `bzero`
- `getchar`
- `memcopy`
- `memmove`
- `memset`
- `printf`
- `putchar`
- `sprintf`
- `strcat`
- `strcmp`
- `strcpy`
- `strlen`
- `strncmp`
- `strncpy`
- `strtok`
- `strtol`
- `strtoul`

CHAPTER 3

Profiler API

This chapter describes the components and functions of the Sun Netra DPS Profiler API. Topics include:

Profiler API Configuration

You can set two properties (TABLE 3-1) 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.

Profiler API Data Types

TABLE 3-2 describes the Profiler API data types.

TABLE 3-2 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 the user in preprocessor define forms.
`teja_profiler_event_t;`	Represents what needs to be measured in a specific group. Group and event combinations make a 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`	Maximum 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:
	`typedef struct teja_profiler_values_t`
	`  uint64_t events`
	`[TEJA_PROFILER_MAX_EVENTS];`

Profiler API Functions

teja_profiler_start

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 the user wants to start profiling more than one group, the user must invoke the same function multiple times.

Syntax

```
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. Up to two different events can be specified at a time.

In case of measuring events inside the CPU group for the UltraSPARC T1 processor, the user can specify only one event. The second *event* is always the number of executed instructions but is not explicitly specified.

In case of measuring events inside DRAM or JBUS group for the UltraSPARC T1 processor or inside any group of events for the UltraSPARC T2 processor, the user can specify two events to be measured at a time. In this case, the *event* argument in the `teja_profiler_start` function call has the following format:

event1 | *event2*

where *event1* and *event2* are events to be measured.

Return Values

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

teja_profiler_stop

Description

Stops collecting profile data for all events in the specified group. This function has empty implementation on some targets.

Syntax

```
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.

teja_profiler_update

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.

Syntax

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

Parameters

group – ID of the group for which the user wants to update profile data.

`...` – List of channels from which to read. The list must be `NULL` terminated.

Return Values

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

teja_profiler_get_values

Description

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

Syntax

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

Parameters

group – ID of the group which the user wants to get the profiler data.

values – User-allocated data structure that will be filled with the profiler data.

Return Values

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

teja_profiler_get_value

Description

Retrieves the value of a given event from a `teja_profiler_values_t` data structure.

Syntax

```
teja_profiler_value_t
teja_profiler_get_value(teja_profiler_values_t *values, int
index);
```

Parameters

values – Data structure that was filled by `teja_profiler_get_values`

index – Index of the event to read (sequential number from 0 up to the maximum number of events specifiable in a group)

Return Values

`teja_profiler_value_t` – Returns the value of the given event.

teja_profiler_dump

Description

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

Syntax

```
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.

Processor Specific Profiler Constants

UltraSPARC T1 Processor–Specific Profiler Groups

TABLE 3-3 lists the specific profiler groups for the UltraSPARC T1 processor.

TABLE 3-3 UltraSPARC T1 Processor – Specific Profiler Groups

Group	Event or Description	Description
`TEJA_PROFILER_CMT_CPU (0x1)`	Captures events related to CPU and caches. The events measured in this group are per strand. The following events are available for this group. The completed instructions count is always an available event for this group.One additional event that can be measured along with the instructions count.	
	`TEJA_PROFILER_CMT_CPU_SB_FULL (0x1)`	Measures the number of store buffer full cycles.
	`TEJA_PROFILER_CMT_CPU_FP_INSTR_CNT (0x2)`	Measures the number of floating point instructions.
	`TEJA_PROFILER_CMT_CPU_IC_MISS (0x4)`	Measures the number of instruction cache misses.

TABLE 3-3 UltraSPARC T1 Processor – Specific Profiler Groups *(Continued)*

Group	Event or Description	Description
	TEJA_PROFILER_CMT_CPU_DC_MISS (0x8)	Measures the number of data cache misses.
	TEJA_PROFILER_CMT_CPU_ITLB_MISS (0x10)	Measures the number of instruction TLB miss traps taken.
	TEJA_PROFILER_CMT_CPU_DTLB_MISS (0x20)	Measures the number of data TLB miss traps taken.
	TEJA_PROFILER_CMT_CPU_L2_IMISS (0x40)	Measures the number of secondary cache (L2) misses due to instruction cache requests.
	TEJA_PROFILER_CMT_CPU_L2_DMISS_LD (0x80)	Measures the number of secondary cache (L2) misses due to data cache load requests.
	TEJA_PROFILER_CMT_CPU_INSTR_COMPLETED (0x100)	Measures the number of completed instructions.
TEJA_PROFILER_CMT_DRAM_CTL0	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.	

TABLE 3-3 UltraSPARC T1 Processor – Specific Profiler Groups *(Continued)*

Group	Event or Description	Description
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.
	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).

UltraSPARC T2 Processor–Specific Profiler Groups

TABLE 3-4 lists the Specific Profiler Groups for the UltraSPARC T2 processor:

TABLE 3-4 UltraSPARC T2 Processor – Specific Profiler Groups

Group	Event or Description	Description
TEJA_PROFILER_CMT_CPU (0x1)	Captures events related to CPU and caches. The events measured in this group are per strand. You can specify up to two independent events that can be concurrently measured. The following events are available for this group.	
	TEJA_PROFILER_CMT2_COMPLETED_BRANCHES	Number of completed branches.
	TEJA_PROFILER_CMT2_TAKEN_BRANCHES	Number of branches taken.
	TEJA_PROFILER_CMT2_FGU_ARITHMATIC_INSTR	Number of floating-point arithmetic instructions executed.
	TEJA_PROFILER_CMT2_LOAD_INSTR	Number of load instructions executed.
	TEJA_PROFILER_CMT2_STORE_INSTR	Number of store instruction executed.
	TEJA_PROFILER_CMT2_SETHI_INSTR	Number of `sethi` instructions executed.
	TEJA_PROFILER_CMT2_OTHER_INSTR	Number of all other instructions executed.
	TEJA_PROFILER_CMT2_ATOMICS	Number of atomic operations executed.
	TEJA_PROFILER_CMT2_ALL_INSTR	Total number of instructions executed.
	TEJA_PROFILER_CMT2_ICACHE_MISSES	Number of instruction cache misses.
	TEJA_PROFILER_CMT2_DCACHE_MISSES	Number of L1 data cache misses.
	TEJA_PROFILER_CMT2_L2_INSTR_MISSES	Number of secondary cache (L2) misses due to instruction cache requests.
	TEJA_PROFILER_CMT2_L2_LOAD_MISSES	Measures the number of secondary cache (L2) misses due to data cache load requests.

TABLE 3-4 UltraSPARC T2 Processor – Specific Profiler Groups *(Continued)*

Group	Event or Description	Description
	TEJA_PROFILER_CMT2_ITLB_REF_L2	For each ITLB miss, counts the number of accesses the ITLB hardware tablewalk makes to L2 when hardware tablewalk is enabled.
	TEJA_PROFILER_CMT2_DTLB_REF_L2	For each DTLB miss, counts the number of accesses the DTLB hardware tablewalk makes to L2 when hardware tablewalk is enabled.
	TEJA_PROFILER_CMT2_ITLB_MISS_L2	For each ITLB miss, counts the number of accesses the ITLB hardware tablewalk makes to L2 which misses in L2 when hardware tablewalk is enabled. Note: Depending on the hardware tablewalk configuration, each ITLB miss might issue from 1 to 4 requests to L2 to search TSB's.
	TEJA_PROFILER_CMT2_DTLB_MISS_L2	For each DTLB miss, counts the number of accesses the DTLB hardware tablewalk makes to L2, which misses in L2 when hardware tablewalk is enabled. Note: Depending on the hardware tablewalk configuration, each DTLB miss may issue from 1 to 4 requests to L2 to search TSB's.
	TEJA_PROFILER_CMT2_STREAM_LD_TO_PCX	Counts the number of SPU load operations to L2.

TABLE 3-4 UltraSPARC T2 Processor – Specific Profiler Groups *(Continued)*

Group	Event or Description	Description
	TEJA_PROFILER_CMT2_STREAM_ST_TO_PCX	Counts the number of SPU store operations to L2.
	TEJA_PROFILER_CMT2_CPU_LD_TO_PCX	Counts the number of CPU loads to L2.
	TEJA_PROFILER_CMT2_CPU_IFETCH_TO_PCX	Counts the number of I-fetches to L2.
	TEJA_PROFILER_CMT2_CPU_ST_TO_PCX	Counts the number of CPU stores to L2.
	TEJA_PROFILER_CMT2_MMU_LD_TO_PCX	Counts the number of MMU loads to L2.
	TEJA_PROFILER_CMT2_DES_3DES_OP	Increments for each CWQ or ASI operation that uses DES/3DES unit.
	TEJA_PROFILER_CMT2_AES_OP	Increments for each CWQ or ASI operation which uses AES unit.
	TEJA_PROFILER_CMT2_RC4_OP	Increments for each CWQ or ASI operation which uses RC4.
	TEJA_PROFILER_CMT2_MD5_SHA1_SHA256_OP	Increments for each CWQ or ASI operation which uses MD5, SHA-1, or SHA-256.
	TEJA_PROFILER_CMT2_MA_OP	Increments for each CWQ or ASI modular arithmetic operation.
	TEJA_PROFILER_CMT2_CRC_TCPIP_CKSUM	Increments for each iSCSI CRC or TCP/IP checksum operation.
	TEJA_PROFILER_CMT2_DES_3DES_BUSY_CYCLE	Increments each cycle when DES/3DES unit is busy.
	TEJA_PROFILER_CMT2_AES_BUSY_CYCLE	Number of busy cycles encountered when attempting to execute the AES operation.

TABLE 3-4 UltraSPARC T2 Processor – Specific Profiler Groups *(Continued)*

Group	Event or Description	Description
	TEJA_PROFILER_CMT2_RC4_BUSY_CYCLE	Number of busy cycles encountered when attempting to execute the RC4 operation.
	TEJA_PROFILER_CMT2_MD5_SHA1_SHA256_BUSY_CYCLE	Number of busy cycles encountered when attempting to execute the MD5_SHA1_SHA256 operation.
	TEJA_PROFILER_CMT2_MA_BUSY	Increments each cycle when modular arithmetic unit is busy.
	TEJA_PROFILER_CMT2_CRC_MPA_CKSUM	Increments each cycle when CRC/MPA/checksum unit is busy.
	TEJA_PROFILER_CMT2_ITLB_MISS	Includes all misses (successful and unsuccessful tablewalks).
	TEJA_PROFILER_CMT2_DTLB_MISS	Includes all misses (successful and unsuccessful tablewalks).
	TEJA_PROFILER_CMT2_TLB_MISS	Counts both ITLB and DTLB misses (successful and unsuccessful tablewalks).
TEJA_PROFILER_CMT_DRAM_CTL0	This group captures events related to DRAM memory read, write, and queues. The events that can be measured are the same as for the UltraSPARC T1 processor (see TABLE 3-3 in "UltraSPARC T1 Processor–Specific Profiler Groups" on page 102). There are different groups for different DRAM controllers.	
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.	

CHAPTER 4

Driver API

This chapter describes the driver application programming interface (API) which consists of the Sun Netra DPS Crypto and Hashing API and Ethernet API. Topics include:

Sun Netra DPS Crypto and Hashing API

Sun Netra DPS Crypto and Hashing API is an interface enables you to access the cryption and hash hardware functions supported by UltraSPARC T2 based platforms.

Note – Sun Netra DPS Cryptography API requires the `SUNWndpsc` Cryptography Driver package.

You do not need to know the details in implementing the crypto and hash APIs when accessing these APIs.

The Sun Netra DPS reference application, IPSec Gateway, is an example of how to use this API. The package `SUNWndpsc` (required export clearance) contains this API.

You must include the following header files under `src/libs/ndps_crypto_api/` in the application:

- `crypt_const.h`
- `ndpscrypto.h`
- `ndpscrypto_impl.h`
- `ndpscrytpo.c` (linked to the `makefile`)

The SPU driver is provided in the binary format located in `SUNWndpsc`: `lib/n2cp/lwrten2cp.o`

You must link the driver into the application.

Sun Netra DPS Crypto and Hash API Function Descriptions

Crypto and Hash Context Setup Part

NDPSCreateCryptoContext

Description

Creates a context for the crypto or hash task to be submitted to the UltraSPARC T2 crypto engine. The caller supplies the cipher, or hash, and the mode, which is the algorithms supported in the UltraSPARC T2 crypto engine. This function allocates the necessary resource to fulfill the crypto or hash task, such as the SPU (Stream Processing Unit) CG devices.

Syntax

```
NDPS_crypto_ctx_t NDPSCreateCryptoContext (
const NDPS_CIPHER cipher, int mode);
```

Parameters

cipher – An algorithm supported in UltraSPARC T2. See `ndpscrypt`. Possible ciphers include AES/DES/3DES/RC4 and MD5/SHA1/SHA256.

mode – The variation for each cipher, such as ECB/CBC/CTR for AES and ECB/CBC/CFB for DES/3DES

Return Values

Returns the opaque handle `NDPS_crypto_ctx_t` to the available hardware CG and SPU devices.

A CG device is used for symmetric key encryption and hashing. You must have `NDPS_crypto_ctx_t` to be able to use other API functions.

To implement, use the following approach. Since each core owns one SPU, any strands on the same core accesses the same SPU. Therefore, the user's routine to access SPU depends on the strand the user is running. For UltraSPARC T2 platforms, the CG the user is accessing = strand# % 8. If the application has only one strand per core to access the SPU on that core, then no other action is required. If the user has two functions running different strands on the same core and both access the SPU, the user then must either place the mutex around the crypto function by accessing the SPU, or allocate one strand whose task is to access the SPU only. The mutex then handles the callers in round-robin fashion to avoid locking.

NDPSDestroyCryptoContext

Description

Releases the context of accessing the SPU/CG device after finishing the crypto and hash task. The released hardware resources are available for the next caller.

Syntax

```
int NDPSDestroyCryptoContext (NDPS_crypto_ctx_t ctx);
```

Parameters

The opaque `NDPS_crypto_ctx_t` handle is allocated through `NDPSCreateCryptoContext`.

Return Values

`0` – Success

`1` – Failure

Crypto API

NDPSCryptKeyLength

Description

Loads the Key length for the crypto. The key length could be 128-bit, 192-bit, or 256-bit.

Syntax

```
int NDPSCryptKeyLength (NDPS_crypto_ctx_t ctx, int key_len);
```

Parameters

ctx – `NDPS_crypto_ctx_t` handle

key_len – Key length

Return Values

0 – Success

1 – Failure

NDPSCryptKeyLoad

Description

Loads the key for the crypto.

Syntax

```
int NDPSCryptKeyLoad (NDPS_crypto_ctx_t ctx, NDPS_key_t *key);
```

Parameters

ctx – `NDPS_crypto_ctx_t` handle

key – Key

Note – To avoid key copy, the caller must maintain space for its key until it calls `NDPSDestroyContext()`.

Return Values

0 – Success

1 – Failure

NDPSCryptIVLoad

Description

Loads the IV for the crypto.

Syntax

```
int NDPSCryptIVLoad (NDPS_crypto_ctx_t ctx, NDPS_iv_t *iv);
```

Parameters

ctx – `NDPS_crypto_ctx_t` handle

iv – IV

Note – To avoid IV copy, the caller must maintain space for its IV until it calls `NDPSDestroyContext()`.

Return Values

0 – Success

1 – Failure

NDPSCrypt

Description

Submits the crypto task with a single data block to the UltraSPARC T2 crypto device.

Syntax

```
int NDPSCrypt (NDPS_crypto_ctx_t ctx, int encrypt_flag,
uchar_t *outbuf, int *outlen, uchar_t *inbuf, int inlen);
```

Parameters

ctx – `NDPS_crypto_ctx_t` handle

encrypt_flag = 1 – For encryption

encrypt_flag = 0 – For decryption

inbuf – Text to be encrypted or decrypted

inlen – Number of the text in bytes

outbuf – Where the crypted or decrypted data is placed

outlen – Number of the crypted or decrypted data in bytes

Return Values

0 – Success

1 – Failure

NDPSCryptMultiple

Description

Submits the crypto task with chained multiple data blocks to the UltraSPARC T2 crypto device.

Syntax

```
int NDPSCryptMultiple (NDPS_crypto_ctx_t ctx, int encrypt_flag,
int num_blk, uchar_t **outbuf, size_t *outlen, uchar_t **inbuf,
size_t *inlen);
```

Parameters

ctx – `NDPS_crypto_ctx_t` handle

encrypt_flag = 1 – For encryption

encrypt_flag = 0 – For decryption

num_blk – Number of data blocks to be chained

inbuf – Array of the input chained data blocks

inlen – Array of the input lengths of the chained data blocks

outbuf – Array of the chained output data blocks

outlen – Array of the lengths of the chained output data blocks

Return Values

0 – Success

1 – Failure

NDPSCryptAndHashMultiple

Description

Submits the Crypto and Hashing tasks with multiple data blocks to the UltraSPARC T2 Crypto device.

Syntax

```
int NDPSCryptAndHashMultiple(NDPS_crypto_ctx_t ctx, int
encrypt_flag,
int num_blk, char **outbuf, size_t *outlen,
char **inbuf, size_t *inlen, NDPS_crypto_ctx_t
h_ctx, char **h_outbuf, size_t *h_outlen,
char **h_inbuf, size_t *h_inlen)
```

Parameters

ctx – Handler `NDPS_crypto_ctx_t` for Crypto

encrypt_flag = 1 – For encrypt and hash

encrypt_flag = 0 – For unhash and decrypt

num_blk – Number of data block CryptHash pairs to be submitted in one request

outbuf – Array of the output data blocks for Crypto

outlen – Array of the lengths of the output data blocks for Crypto

inbuf – Array of the input data blocks for Crypto

inlen – Array of the lengths of the input data blocks for Crypto

h_ctx – Handler `NDPS_Crypto_ctx_t` for Hash

h_outbuf – Array of the output data blocks for Hash

h_outlen – Array of the lengths of the output data blocks for Hash

h_inbuf – Array of the input data blocks for Hash

h_inlen – Array of the lengths of the input data blocks for Hash

Return Values

0 – Success

1 – Failure

Hash API

NDPSHashLength

Description

Sets the Hash length.

Syntax

```
int NDPSHashLength (NDPS_crypto_ctx_t ctx, int len);
```

Parameters

ctx – `NDPS_crypto_ctx_t` handle

len – Hash length

Return Values

0 – Success

1 – Failure

NDPSHashIVLoad

Description

Loads the Hash IV (initialization vector) load.

Note – To avoid IV copy, the caller must maintain space for its IV until it calls `NDPSDestroyContext()`.

Syntax

```
int NDPSHashIVLoad(NDPS_crypto_ctx_t ctx, NDPS_iv_t *iv);
```

Parameters

ctx – `NDPS_crypto_ctx_t` handle

iv – Hash IV value

Return Values

0 – Success

1 – Failure

NDPSHashIVGet

Description

Acquires the IV (initialization vector) address for the hash.

Syntax

```
int NDPSHashIVGet (NDPS_crypto_ctx_t ctx, NDPS_iv_t **iv);
```

Parameters

ctx – `NDPS_crypto_ctx_t` handle

iv – Pointer to the IV location

Return Values

0 – Success

1 – Failure

NDPSHashDirect

Description

Produces the Hash value from the input data with its length. This Hash function does not overwrite the internal IV, but rather does a complete hash operation and stores the result in the provided `outbuf`.

Syntax

```
int NDPSHashDirect (NDPS_crypto_ctx_t ctx, uchar_t *outbuf,
uchar_t *inbuf, int inlen);
```

Parameters

ctx – `NDPS_crypto_ctx_t` handle

inbuf – Input data to be hashed

inlen – Length of data to be hashed

outbuf – Resulting hash value

Return Values

0 – Success

1 – Failure

NDPSHashDirectMultiple

Description

Submits the Hash task with chained multiple data blocks to the UltraSPARC T2 crypto device.

Syntax

```
int NDPSHashDirectMultiple (NDPS_crypto_ctx_t ctx, int num_blk,
uchar_t **outbuf, size_t *outlen, uchar_t **inbuf,
size_t *inlen);
```

Parameters

ctx – The `NDPS_crypto_ctx_t` handle

num_blk – Number of data blocks to be chained

outbuf – Array of the chained output data blocks

outlen – Array of the lengths of the chained output data blocks

inbuf – Array of the input chained data blocks

inlen – Array of the input lengths of the chained data blocks

Return Values

0 – Success

1 – Failure

Crypto and Hash Combined API

NDPSCryptAndHash

Description

Combines crypto and hash operations in one function call. This API calls SPU in one call to get a performance boost.

Syntax

```
int NDPSCryptAndHash(NDPS_crypto_ctx_t ctx, int encrypt_flag,
char *outbuf, int *outlen, char *inbuf, int inlen,
NDPS_crypto_ctx_t h_ctx,
char *h_outbuf, int h_outlen, char *h_inbuf, int h_inlen);
```

Parameters

ctx – `NDPS_crypto_ctx_t` for crypto handle

encrypt_flag = 1 – For encryption

encrypt_flag = 0 – For decryption

outbuf – Array of the chained output data blocks for crypto

outlen – Array of the lengths of the chained output data blocks for crypto

inbuf – Array of the input chained data blocks for crypto

inlen – Array of the input lengths of the chained data blocks for crypto

h_ctx – `NDPS_crypto_ctx_t` handle for Hash

h_outbuf – Array of the chained output data blocks for Hash

h_outlen – Array of the lengths of the chained output data blocks for Hash

h_inbuf – Array of the input chained data blocks for Hash

h_inlen – Array of the input lengths of the chained data blocks for Hash

Return Values

0 – Success

1 – Failure

Miscellaneous APIs

The following APIs support AES-XCBC-MAC-96.

NDPSAESXCBCMAC96init

Description

Initializes AES-XCBC-MAC-96.

Syntax

```
int NDPSAESXCBCMAC96init();
```

Parameters

None

Return Values

0 – Success

1 – Failure

NDPSAESXCBCMAC96fini

Description

Finalizes AES-XCBC-MAC-96.

Syntax

```
int NDPSAESXCBCMAC96fini();
```

Parameters

None

Return Values

0 – Success

1 – Failure

NDPSAESXCBCMAC96KeyLoad

Description

Loads the initial key for AES-XCBC-MAC-96. which is supplied by the caller.

Syntax

```
int NDPSAESXCBCMAC96KeyLoad (
NDPS_crypto_ctx_t ctx, NDPS_key_t *key);
```

Parameters

ctx – `NDPS_crypto_ctx_t` handle for crypto

key – Key

Return Values

0 – Success

1 – Failure

NDPSAESXCBCMAC96AuthGenerate

Description

Generates the AES-XCBC-MAC-96 authentic value in 96-bit.

Syntax

```
int NDPSAESXCBCMAC96AuthGenerate(NDPS_crypto_ctx_t ctx,
uchar_t *inbuf, int inlen, uchar_t **auth_buf, int *auth_len);
```

Parameters

ctx – `NDPS_crypto_ctx_t` handle for crypto

inbuf – Input data for AES-XCBC-MAC-96

inlen – Input lengths for AES-XCBC-MAC-96

auth_buf – Resulting AES-XCBC-MAC-96 hash value

auth_len – Lengths, in 96-bits

Return Values

0 – Success

1 – Failure

Ethernet API

The Ethernet API is an interface between the user network application and the device drivers. A Sun Netra DPS application developer should be aware of the device features and capabilities but does not need to have the knowledge of the detailed implementation of the device driver. TABLE 4-1 shows the relationship among Ethernet device, device driver, Ethernet API, and the user application.

TABLE 4-1 Ethernet API and User Applications

Network application	For example: RLP, IP packet forwarding, IPSec
Ethernet API	For example: `eth_open`, `eth_close`, `eth_read`
Device driver	For example: `nxge`
Ethernet device	For example: 10Gb Ethernet with NIU, Ophir

Network Applications

Network applications require network hardware resources. The RLP, IP packet forwarding, and IPSec reference applications are all network applications.

Ethernet Device Driver

TABLE 4-2 lists the Ethernet device supported in Sun Netra DPS platforms.

TABLE 4-2 Ethernet Devices Supported on Sun Netra DPS Platforms

Device Driver	Ethernet Device
`nxge`	Sun multithreaded 10Gb Ethernet with NIU

See "Note 11" on page 143 for Ethernet device driver `nxge` tunables.

Ethernet API Functions

The API list of functions include the following:

Description of Ethernet API Functions

eth_pbuf_alloc

Description

Allocates a message block for managing incoming packet data. The allocated entity is returned as a pointer to the buffer block structure (`pbuf_t`). `pbuf_t` is a message block struct (`mblk`) that consists of the all necessary pointers and fields for manipulating the data buffer. See the `mblk_t` in `mblk.h` header file for the details of the message block. Packet data begins at `b_wptr`. The size of the `mblk` must be the size specified as `mblk_size` in the `eth_open`() call. This API is implemented in the user application space. (See "Note 4" on page 139). The device driver calls this function.

Syntax

```
pbuf_t *eth_pbuf_alloc(void *hook, size_t bufsz, uint16_t pool);
```

Parameters

hook – User-provided hook. (See in "Note 1" on page 139)

bufsz – User-provided buffer size to be allocated. (See "Note 2" on page 139.)

pool – DMA channel pool (See "Note 3" on page 139.)

Return Values

On success, returns pointer to `mblk` with `b_rptr` and `b_wptr` pointing to the start of a valid data buffer. An error returns `NULL`.

eth_pbuf_free

Description

Frees a message block allocated by `eth_pbuf_alloc()`. This function is implemented by the user and is called by the device driver.

Syntax

```
void eth_pbuf_free(void *hook, pbuf_t * mblkp, void *arg,
uint16_t pool);
```

Parameters

hook – User-provided hook. (See in "Note 1" on page 139.)

mblkp – Pointer to message block to be freed

arg – Not used (pass in `NULL`)

pool – DMA channel pool. (See "Note 3" on page 139.)

eth_buf_alloc

Description

Allocates a data buffer for storing incoming packet data. The allocated entity is a pointer to the allocated buffer. This function is implemented in the user application space (see "Note 4" on page 139). The device driver calls this function.

Syntax

```
char *eth_buf_alloc(void *hook, size_t bufsz, uint16_t pool);
```

Parameters

hook – User-provided hook. (See "Note 1" on page 139.)

bufsz – User-provided buffer size to be allocated. (See "Note 2" on page 139.)

pool – DMA channel pool (See "Note 3" on page 139.)

Return Values

On success, returns the pointer to a valid data buffer. An error returns `NULL`.

eth_buf_free

Description

Frees a buffer allocated by `eth_buf_alloc()`. This function is implemented in the user application space. (See "Note 4" on page 139.) The device driver calls this function.

Syntax

```
void eth_buf_free(void *hook, char *buf, void *arg, uint16_t pool);
```

Parameters

hook – User-provided hook. (See "Note 1" on page 139.)

buf – Pointer to data buffer to be freed

arg – Not used (pass in `NULL`)

pool – DMA channel pool. (See "Note 3" on page 139.)

eth_open

Description

Probes a network device in the target platform and, if the device is found, this function initializes the network device. On a successful completion, this function returns an opaque handle, which needs to be used in other API calls that is targeted to a specific device. When multiple ports are opened, `eth_open()` *must* be invoked in the increasing order of the port numbers, that is, port0, then port1, and so on, during initialization.

Syntax

```
ihandle_t eth_open(uint16_t vid, uint16_t did, eth_port_t port,
int num_chans, void *txhook, void* rxhook,
size_t mblk_siz, uint_t mpbase);
```

Parameters

vid – Vendor ID of network device

did – Device ID of network device

port – Port number of the Ethernet interface. (See "Note 5" on page 139.)

txhook – Application provided hook to `tx fastq` table. (See "Note 6" on page 140.)

rxhook – Application provided hook to `rx fastq` table (See "Note 7" on page 141.)

mblk_siz – Size of buffer that is returned by `eth_pbuf_alloc()`

mpbase – Base index into the mempool type array used in application (See "Note 8" on page 142.)

Return Values

On success – Returns a valid opaque device handle. (`ihandle_t`)

On error – Returns `INVALID_IHANDLE`

eth_close

Description

Releases the Ethernet interface instance and all resources held by it.

Syntax

```
int eth_close(ihandle_t ihandle);
```

Parameters

ihandle – Opaque handle returned by `eth_open()`.

Return Values

0 – Success

1 – Failure

eth_read

Description

Receives messages from the Ethernet interface instance specified by `ihandle`. This function can be configured to return a chain of packets. The maximum number of packets in the chain is configurable through `ETH_IOC_SET_MAX_PKT_CHAIN`. This function is nonblocking.

Syntax

```
pbuf_t *eth_read(ihandle_t ihandle, eth_chan_t chan_num);
```

Parameters

ihandle – Opaque handle returned by `eth_open()`.

chan_num – DMA channel number (See "Note 9" on page 142.)

Return Values

On success – Returns `mblk` packet chain containing message

On error – Returns `NULL`

eth_write

Description

Sends a message which is specified by the message block structure pointer (`mblk`). This function is nonblocking and can fail if the hardware transmit descriptor ring is full.

Syntax

```
int eth_write(ihandle_t ihandle, eth_chan_t chan_num,
pbuf_t * mblkp);
```

Parameters

ihandle – Opaque handle returned by `eth_open()`.

chan_num – DMA channel number. (See "Note 9" on page 142.)

mblkp – Message block pointer. This pointer can be a chain. The maximum size of chain supported is implementation-specific and can be discovered via `ETH_IOC_GET_MAX_TX_PKT_CHAIN`.

Return Values

0 – Success

1 – Failure

Note – When the `nxge` driver is used, the application is responsible for ensuring that the packet being transmitted is at least 64 bytes in length. The `nxge` hardware does not support padding of Ethernet frames less than 64 bytes.

`eth_ioc`

Description

Catch-all configuration API that can be used to control the device driver attributes.

Syntax

`int eth_ioc(ihandle_t` *ihandle*`, ioc_cmd_t` *cmd*`, void *`*arg*`);`

Parameters

ihandle – Opaque handle returned by `eth_open()`.

cmd – Command to execute (See "Note 10" on page 142.)

**arg* – Argument passed to command

Return Values

0 – Success

1 – Failure

`eth_ioc` Command and Arguments

This section specifies the content that needs to be passed into the `arg` parameter of the IOC command.

`ETH_IOC_GET_MAC_ADDR`

Description

Obtain MAC ID of the ethernet port from the MAC address hardware registers.

Arguments

```
(struct ether_addr *)arg

struct  ether_addr {

        ether_addr_t    ether_addr_octet;

};
```

typedef uchar_t ether_addr_t[ETHERADDRL]; /* 6 octets */

ETH_IOC_SET_MAC_ADDR

Description

Set MAC ID into the MAC address hardware registers of the ethernet port.

Arguments

(struct ether_addr *)arg

ETH_IOC_CHK_LINK

Description

Check the ethernet link status. When link status is changed, display a message on the console showing the new Ethernet status.

Arguments

None

ETH_IOC_GET_LINK

Description

Obtain the current Ethernet link status and return it to the `link_status_ioc_t` structure.

Arguments

```
(link_status_ioc_t *)arg

typedef struct _link_status_ioc_s {

        int status; /* 0: Link Down 1: Link Up */

        int speed; /* Link speed in Mbps 10/100/1000/10000 */
```

```
        int duplex; /* 1: Half 2: Full */
} link_status_ioc_t;
```

ETH_IOC_SET_PROMISC

Description

Set a promiscous mode to the Ethernet port.

Arguments

```
(promisc_mode_ioc_t *)arg
typedef struct _promisc_mode_ioc_s {
        promisc_mode_t  mode; /* Promiscuous mode */
       boolean_t       enable; /* B_TRUE: enable B_FALSE: disable */
} promisc_mode_ioc_t;
typedef enum {
        PROMISC_ALL = 0, /* Accepts all valid frames */
        PROMISC_GRP, /* Accepts all valid multicast frames */
        PROMISC_VIRT /* Enable MAC Address Filtering */
} promisc_mode_t;
```

ETH_IOC_SET_MAX_FRAME_SIZE

Description

Set the maximum allowable frame size that the MAC will receive.

Arguments

```
(*(int *)arg)
```

Note – User can declare an interger variable named `maxfrmsz` and pass in the address of the variable as the third argument when calling `eth_ioc()`.

ETH_IOC_ADD_MCAST_ADDR

Description

Add a multicast address entry into the hardware multicast hash table.

Arguments

```
(struct ether_addr *)arg
```

Note – See argument of `ETH_IOC_GET_MAC_ADDR`.

ETH_IOC_DEL_MCAST_ADDR

Description

Delete a multicast address entry from the hardware multicast hash table.

Arguments

(struct ether_addr *)arg

ETH_IOC_SHOW_MCAST_ADDR

Description

Diaplay the hardware multicast hash table content to the console.

Arguments

None

ETH_IOC_SET_ADDR_FILTER

Description

Set up the address filter and the address filter mask hardware registers.

Arguments

```
(addr_filter_ioc_t *)arg
typedef struct _addr_filter_ioc_s {
        struct ether_addr       addr_filter; /* Address Filter */
        uint16_t        mask_0_15_bits_a0; /* Bit mask octet4-5 */
```

```
        uint8_t         mask_0_7_nib_a1_a2; /* Nibble mask octet0-3*/
} addr_filter_ioc_t;
```

ETH_IOC_GET_STATS

Description

Obtain statistics of `igress` and `egress` packet count of the Ethernet port.

Arguments

```
(nxge_eth_kstat_t *)arg
typedef struct nxge_eth_kstat {
        uint64_t ipackets; /* Ingress packets count */
        uint64_t opackets; /* Egress packets count */
        uint64_t rbytes; /* Ingress packets byte count */
        uint64_t obytes; /* Egress packets byte count */
} nxge_eth_kstat_t;
```

ETH_IOC_SHOW_STATS

Description

Display Ethernet port statistical information to the console.

Arguments

None

ETH_IOC_SET_MAC_TBL

Description

Set up the hardware MAC table.

Arguments

```
(mac_tbl_ioc_t *)arg
typedef struct _mac_tbl_ioc_s {
        mac_addr_type_t         addr_type; /* MAC address type */
```

```
        struct ether_addr       mac_addr; /* MAC ID */
        uint8_t                 rdc_grp_id; /* RDC Group ID (0~7)*/
        boolean_t               enable; /* enable/disable */
        boolean_t               priority; /* High/Low Priority */
} mac_tbl_ioc_t;
typedef enum {
        MAC_ADDR_SELF = 0, /* MAC ID of this interface */
        MAC_ADDR_ALT0, /* Alternate MAC ID #0 */
        MAC_ADDR_ALT1, /* Alternate MAC ID #1 */
        MAC_ADDR_ALT2, /* Alternate MAC ID #2 */
        MAC_ADDR_ALT3, /* Alternate MAC ID #3 */
        MAC_ADDR_ALT4, /* Alternate MAC ID #4 */
        MAC_ADDR_ALT5, /* Alternate MAC ID #5 */
        MAC_ADDR_ALT6, /* Alternate MAC ID #6 */
        MAC_ADDR_ALT7, /* Alternate MAC ID #7 */
        MAC_ADDR_ALT8, /* Alternate MAC ID #8 */
        MAC_ADDR_ALT9, /* Alternate MAC ID #9 */
        MAC_ADDR_ALT10, /* Alternate MAC ID #10 */
        MAC_ADDR_ALT11, /* Alternate MAC ID #11 */
        MAC_ADDR_ALT12, /* Alternate MAC ID #12 */
        MAC_ADDR_ALT13, /* Alternate MAC ID #13 */
        MAC_ADDR_ALT14, /* Alternate MAC ID #14 */
        MAC_ADDR_ALT15, /* Alternate MAC ID #15 */
        MAC_ADDR_RSVD_MULTICAST, /* Reserved Multicast */
        MAC_ADDR_FILTER, /* Address Filter */
        MAC_ADDR_FLOW_CTL /* Flow Control Address */
} mac_addr_type_t;
```

ETH_IOC_SHOW_MAC_TBL

Description

Display the hardware MAC table content to the console.

Arguments

None

ETH_IOC_SET_VLAN_TBL

Description

Set up the hardware VLAN table.

Arguments

```
(vlan_tbl_ioc_t *)arg
typedef struct _vlan_tbl_ioc_s {
        uint16_t        vlan_id; /* VLAN LD (0 ~ 4095) */
       uint8_t        rdc_grp_id; /* RX DMA Channel Group ID (0~7)*/
        boolean_t       enable; /* Enable/Disable */
        boolean_t       priority; /* High/Low priority */
} vlan_tbl_ioc_t;
```

ETH_IOC_SET_RDC_GRP

Description

Set up the hardware RDC (Receive DMA Channel) group table.

Arguments

```
(rdc_tbl_ioc_t *)arg
typedef struct _rdc_tbl_ioc_s {
        uint8_t                 rdc_grp_id; /* RDC Group ID (0~7)*/
       uint8_t               rdc[NXGE_NUM_CHANLS]; /* RDC (0~15) */
} rdc_tbl_ioc_t;
```

ETH_IOC_SHOW_RDC_GRPS

Description

Display the hardware RDC group table content to the console.

Arguments

None

ETH_IOC_BIND_RDC_GRP

Description

Bind a default RDC group number to a port.

Arguments

```
(uint8_t *)arg /* RDC Group number */
```

ETH_IOC_GET_PORTINFO

Description

Obtain the port information, and return it to `portinfo_ioc_t`.

Arguments

```
(portinfo_ioc_t *)arg
typedef struct _portinfo_ioc_s {
        struct ether_addr               mac_addr; /* MAC ID */
       struct _link_status_ioc_s      link_status; /* Link status */
        int             rx_min_frame_sz; /* Rx Minimum Frame Size */
        int             tx_min_frame_sz; /* Tx Minimum Frame Size */
        int             max_frame_sz; /* Maximum Frame Size */
        boolean_t       tx_en; /* Transmit Enable/Disable */
        boolean_t       rx_en; /* Receive Enable/Disable */
        boolean_t       addr_filter_en; /* Addr Filtering Ena/Dis */
        boolean_t       hash_filter_en; /* Hash Filtering Ena/Dis */
        boolean_t       promisc_all; /* Promiscuous All Ena/Dis */
        boolean_t       promisc_grp; /* Promiscuous Group Ena/Dis */
```

```
        boolean_t       rx_strip_crc; /* Rx Strip CRC Ena/Dis */
        boolean_t       tx_gen_crc; /* Tx CRC generation Ena/Dis */
        boolean_t       rx_err_chk; /* Rx Error Check Ena/Dis */
} portinfo_ioc_t;
```

ETH_IOC_SHOW_PORTINFO

Description

Display the port information to the console.

Arguments

None

ETH_IOC_SET_CLASSIFY

Description

Set classification rules for incoming traffic to the receive port.

Arguments

```
(classify_ioc_t *)arg
typedef struct classify_ioc_s {
          uint_t opcode; /* 1:Add an entry 2:Invalidate an entry */
          uint_t action; /* 1:Accept on match 2:Discard on match*/
          flow_spec_t flow_spec; /* Flow specification */
} classify_ioc_t;
typedef struct flow_spec_s {
        uint_t fs_type;                 /* Flow Spec Type */
        uint_t index;                   /* Index of flow entry */
        uint_t channel; /* RDC to be selected when match */
        union {
                flow_spec_ipv4_t ip4;   /* IPv4 flow spec */
                flow_spec_ipv6_t ip6;   /* IPv6 flow spec */
```

```
                flow_spec_l2_t   l2;    /* L2 flow spec */
                uint8_t hd[64];         /* Hex data */
        } ue, um;
} flow_spec_t;
typedef struct flow_spec_ipv4_s {
        uint8_t protocol;
        uint8_t tos;
        union {
                port_t tcp;
                port_t udp;
                spi_port_t spi;
        } port;
        uint32_t src;
        uint32_t dst;
} flow_spec_ipv4_t;
typedef struct flow_spec_ipv6_s {
        uint8_t protocol;
        uint8_t tos;
        union {
                port_t tcp;
                port_t udp;
                spi_port_t spi;
        } port;
        struct ip6_addr src;
        struct ip6_addr dst;
} flow_spec_ipv6_t;
typedef struct flow_spec_l2_s {
        // uint32_t l2_rule;    /* l2 classification types */ XXXX
```

```
        uint8_t dst[6];         /* MAC address */
        uint8_t src[6];         /* MAC address */
        uint16_t type;          /* Ether type */
        uint16_t vlantag;       /* VLANID|CFI|PRI */
} flow_spec_l2_t;
```

ETH_IOC_CHK_ERRS

Description

Check errors on the Ethernet port.

Arguments

None

Ethernet API Function Summary

TABLE 4-3 lists a summary of the Ethernet API functions.

TABLE 4-3 Ethernet API Function Summary

API	Purpose	Implemented by	Called by
eth_pbuf_alloc	Allocate message block	User application	Device driver
eth_pbuf_free	Free message block	User application	Device driver
eth_buf_alloc	Allocate buffer	User application	Device driver
eth_buf_free	Free buffer	User application	Device driver
eth_open	Find and init device	Device driver	User application
eth_close	Free up device resource	Device driver	User application
eth_read	Poll for received packet	Device driver	User application
eth_write	Send a packet	Device driver	User application
eth_ioc	Device Control	Device driver	User application

Notes

Note 1

This is a catch-all argument for the user application. This argument can be used for any purpose. If not used, pass in `NULL`.

Note 2

The size value should be large enough to hold an Ethernet packet.

Note 3

When using multiple memory pools (one for each DMA channel), *pool* indicates the ID of the memory pool, which is normally indexed by the DMA channel number. When a single memory pool is used, always pass in a zero.

Note 4

The user application has the best knowledge and control of how system memory is utilized. The device driver calls this function in the Packet Read routine.

Note 5

The port number of the device can be determined using the `-v` option during boot. For example, `boot net:,my_binary_file -v`.

Part of the console output is similar to the following:

```
NIU : SUNW,niumx
netdev[0]: VendorId 0x108e DevId 0xabce
netdev[0]: Subsystem VendorId 0x0 SubsystemId 0x0
netdev[0]: RevisionId 0x0
netdev[0]: PhyType gsd
netdev[0]: Compatible SUNW,niusl
netdev[0]: cfg_addr 0x0 pio_addr 0x8100000000
netdev[0]: mac_addr 0x0:14:4f:8c:4:3e
netdev[1]: VendorId 0x108e DevId 0xabce
netdev[1]: Subsystem VendorId 0x0 SubsystemId 0x0
netdev[1]: RevisionId 0x0
netdev[1]: PhyType gsd
netdev[1]: Compatible SUNW,niusl
netdev[1]: cfg_addr 0x0 pio_addr 0x8102000000
netdev[1]: mac_addr 0x0:14:4f:8c:4:3f
```

This output indicates that there are two NIU ports. The numbers inside the `netdev[]` (0 and 1) are the port numbers used in `eth_open` calls.

Note 6

For an application that needs to forward packets (for example, RLP, IP packet forwarding, and IPSec applications), the application must pass in the pointer to the transmit `fastq` allocated by the application.

If `nxge_multi_qs` is enabled (set to 1), the driver expects the application to pass in the entire free queue array because the driver needs to determine the index of the free queue element to be freed based on the forwarding port number and channel number.

Example:

```
fastq_t
txfreeq_dram[NUM_PORTS+START_PORT][NUM_CHANS][NUM_PORTS];

fastq_t
rxfreeq_dram[NUM_PORTS+START_PORT][NUM_CHANS][NUM_PORTS];

eth_open(vid, did, port, NUM_CHANS, (void *)txfreeq_dram, (void
*)rxfreeq_dram, RX_BUF_SIZE, poolidx);
```

If `nxge_multi_qs` is disabled (set to 0), the driver expects the application to pass in the starting element of free queue array indexed by the port. The driver needs to determine only the `freeq` array element to be freed based on the channel number. The port number in this mode is static.

Example:

```
fastq_t  txfreeq_dram[NUM_PORTS+START_PORT][NUM_CHANS];

fastq_t  rxfreeq_dram[NUM_PORTS+START_PORT][NUM_CHANS];

eth_open(vid, did, port, NUM_CHANS, (void
*)&txfreeq_dram[port], (void *)&rxfreeq_dram[port], RX_BUF_SIZE,
poolidx);
```

Note 7

For an application that must forward packets (for example, RLP, IP packet forwarding, and IPSec applications), the application must pass in the pointer to the receive `fastq` allocated by the application.

If `nxge_multi_qs` is enabled (set to 1), the driver expects the application to pass in the entire free queue array because the driver needs to determine the index of the free queue element to be freed based on the forwarding port number and channel number.

Example:

```
fastq_t
txfreeq_dram[NUM_PORTS+START_PORT][NUM_CHANS][NUM_PORTS];

fastq_t
rxfreeq_dram[NUM_PORTS+START_PORT][NUM_CHANS][NUM_PORTS];

eth_open(vid, did, port, NUM_CHANS, (void *)txfreeq_dram, (void
*)rxfreeq_dram, RX_BUF_SIZE, poolidx);
```

If `nxge_multi_qs` is disabled (set to 0), the driver expects the application to pass in the starting element of free queue array indexed by the port. The driver needs to determine only the `freeq` array element to be freed based on the channel number. The port number in this mode is static.

Example:

```
fastq_t  txfreeq_dram[NUM_PORTS+START_PORT][NUM_CHANS];

fastq_t  rxfreeq_dram[NUM_PORTS+START_PORT][NUM_CHANS];

eth_open(vid, did, port, NUM_CHANS, (void
*)&txfreeq_dram[port], (void *)&rxfreeq_dram[port], RX_BUF_SIZE,
poolidx);
```

Note 8

When the application is using multiple memory pools, this is the index to the first memory pool used by the device. For example, if the device to be opened has eight memory pools (one for each DMA channel) and the memory pool ID are identified from 0 to 7, then the base index is 0.

Note 9

This is the DMA channel number of the DMA channel receiving the packet. In Sun multithreaded 10GbE with NIU, up to 16 DMA channels can be used. The number of DMA channels to be used is specified when calling `eth_open()`.

Note 10

In the Sun Netra DPS 2.1 Update 1 release, the following control commands are implemented:

- `ETH_IOC_GET_MAC_ADDR` – Get MAC ID of the interface
- `ETH_IOC_SET_MAC_ADDR` – Set MAC ID to the interface
- `ETH_IOC_CHK_LINK` – Check link status
- `ETH_IOC_GET_LINK` – Acquire Ethernet link status
- `ETH_IOC_SET_PROMISC` – Set promiscuous mode
- `ETH_IOC_SET_MAX_FRAME_SIZE` – Set maximum receive frame size
- `ETH_IOC_ADD_MCAST_ADDR` – Add multicast address
- `ETH_IOC_DEL_MCAST_ADDR` – Delete multicast address
- `ETH_IOC_SHOW_MCAST_ADDRS` – Display multicast addresses
- `ETH_IOC_SET_ADDR_FILTER` – Set address filter and filter mask
- `ETH_IOC_GET_STATS` – Get network statistics
- `ETH_IOC_SHOW_STATS` – Display network statistics
- `ETH_IOC_SET_MAC_TBK` – Setup the hardware MAC table content
- `ETH_IOC_SHOW_MAC_TBL` – Display the hardware MAC table content
- `ETH_IOC_SET_VLAN_TBL` – Setup the hardware VLAN table content
- `ETH_IOC_SET_RDC_GRP` – Setup the hardware RDC table content
- `ETH_IOC_SHOW_RDC_GRPS` – Display the hardware RDC table content
- `ETH_IOC_BIND_RDC_GRP` – Bind a default RDC group to a port
- `ETH_IOC_GET_PORTINFO` – Get Ethernet port information
- `ETH_IOC_SHOW_PORTINFO` – Display Ethernet port information

- ETH_IOC_SET_CLASSIFY – Setup receive classification rules
- ETH_IOC_CHK_ERRS – Check device errors

Refer to the reference application (for example, IP packet forwarding) for usage of these commands.

Note 11

TABLE 4-4 lists the Ethernet device driver nxge tunables.

TABLE 4-4 Ethernet Device Driver nxge Tunables

Driver Tunable	Description
extern uint_t nxge_max_dmas	max_dmas increases the maximum DMAs passed to the eth_open() API. The default is 8, however, it can be set up to use more than 8 channels per port. For example, it can be set up for all 16 channels on one port.
extern uint_t nxge_tx_kicks	tx_kicks specifies the number of packets the driver tx is going to store before it actually notifies the hardware to send packets out.
extern uint_t nxge_rxoff_var	rxoff_var enables spreading of ingress packets to several cache line-aligned start addresses for each DMA: 64 | 128.
extern uint_t nxge_rxoff_var_n2_2	rxoff_var_n2_2 enables the UltraSPARC T2 2.2 behavior which allows two more start offsets to be used by ingress packets: 320 | 384.
extern uint_t nxge_flow_cfg	flow_cfg enables different flow/classification policies.
extern uint_t nxge_rx_pref	rx_pref enables prefetch instructions on the rx packet data buffer so that subsequent stages can have the data ready in cache.
extern uint_t nxge_kstat_on	kstat_on flag is 1 by default. If off will not update the ipackets/opackets counters in the driver. It can be turned off by setting it to 0.
extern uint_t nxge_prmsc_all	prmsc_all flag is 1 by default. If set to 1, it turns on promiscuous mode, meaning that packets that do not match port mac-id would be received on default DMA.
extern uint_t nxge_prmsc_mgrp	prmsc_mgrp flag is 1 by default; it enables receive of multicast packets; if set to 0, it turns off multicast packets.
extern uint_t nxge_prmsc_virt	prmsc_virt flag is 0 by default. If set to 1, it turns on virtual promiscuous mode, which is a special promiscuous mode to spread packets to all available DMAs for that port. When this flag is enabled, it takes precedence over the other two promiscuous mode flags: "prmsc_all" and "prmsc_mgrp".

TABLE 4-4 Ethernet Device Driver nxge Tunables

Driver Tunable	Description
extern uint_t nxge_dma_sprd	This flag should always be left to the default value (0) so that equal among of DMA channels will be sub-divided among ports.
extern uint_t nxge_atomic_on	When this flag is set, the driver enables atomic operation for packet reference count. This flag is set to 0 by default.
extern uint_t nxge_multi_qs	nxge_multi_qs flag enables an efficient free queue buffer management mechanism for handling dynamic forwarding from one input port to a variable output port. This flag is set to 0 by default.
extern uint_t nxge_max_frame_sz	nxge_max_frame_sz flag sets the maximum Ethernet frame size the MAC can transmit and receive. Default value is 1518.
extern uint_t nxge_tx_chain	nxge_tx_chain flag, when set, enables the Tx chaining functionality of the driver's transmit function. When this flag is disabled, the application needs to assure that it does not utilize Tx packet chaining.
extern uint_t nxge_niu0_artm	This tunable is set to inform the driver whether the NIU port0 of the CP3260 blade is connected to the backplane switch or to the ARTM module. If this flag is set to 1, the driver is informed that NIU port0 is connected to the ARTM's XAUI. If this flag is set to 0, the driver is informed that NIU port0 is connected to the backplane switch.
extern uint_t nxge_niu0_artm	This tunable is set to inform the driver whether the NIU port1 of the CP3260 blade is connected to the backplane switch or to the ARTM module. If this flag is set to 1, the driver is informed that NIU port1 is connected to the ARTM's XAUI. If this flag is set to 0, the driver is informed that NIU port0 is connected to the backplane switch.

Note 12

This note describes how to enable the hardware checksum offload features on the Sun multithreaded 10Gb/NIU Ethernet hardware using the nxge driver.

The following mblk fields are used:

```
unsigned char b_ick_flag; /* H/W checksum enable flag : TX */
unsigned char *b_ick_start; /* Pointer to start offset : TX/RX */
unsigned char *b_ick_stuff; /* Pointer to stuff offset : TX */
```

If (b_ick_flag and NXGE_TX_CKENB), then the hardware is programmed to compute hardware checksum. It is expected that the ick_start/stuff point to the L4 payload start/stuff offsets, respectively. Also, the udp/tcp header checksum field must be filled with the pseudo header checksum value. The hardware will use this field for computing the full checksum.

On rx, if (b_ick_flag and NXGE_RX_CKERR), the hardware detected a checksum error in the ingress packet.

vnet Driver API

The vnet driver API is an interface between the user application and the Sun Netra DPS vnet driver. A Sun Netra DPS application developer must be aware of the vnet driver capabilities and features, but she or he does not need to know the detailed implementation of the device driver.

vnet API Functions

- vnet_pbuf_alloc
- vnet_buf_alloc
- vnet_pbuf_free
- vnet_buf_free
- vnet_eth_open
- vnet_eth_read
- vnet_eth_write
- vnet_eth_ioc
- vnet_eth_get_mac_addr
- vnet_eth_flush
- vnet_set_rxburst
- vnet_get_rxburst

Description of vnet API functions

vnet_pbuf_alloc

Description

External function to be implemented by the user and called by the library to allocate a message block for incoming packets. The allocated entity is returned as a pointer to the buffer block structure (`pbuf_t`). `pbuf_t` is a message block structure (`mblk_t`) that contains the necessary pointers and fields for manipulating the data buffer. See `mblk_t` in the `mblk.h` header file for the details about the message block.

Packet data begins at `b_wptr`. The library assumes that the `mblk` returned by this function will point to a valid buffer. The library will start writing packet data at the byte pointed by `b_wptr`. This function is implemented in the user application space. The device driver calls this function.

Syntax

```
pbuf_t *vnet_pbuf_alloc(void *hook, size_t bufsz, uint16_t pool);
```

Parameters

hook – currently unused by the `vnet` device driver; the driver will always pass NULL for this value.

bufsz - buffer size to be allocated

pool - pool id from which buffer is to be allocated

Return Values

On success, returns a `mblk` with `b_rptr` and `b_wptr` pointing to start of a valid data buffer. On error, returns NULL.

vnet_buf_alloc

Description

Allocates a data buffer for storing incoming packet data. The allocated entity is a pointer to the allocated buffer. This function is implemented in the user application and called by the device driver.

Syntax

```
unsigned char *vnet_buf_alloc(void *hook, size_t bufsz, uint16_t
pool);
```

Parameters

hook – Currently unused by the `vnet` device driver. The driver will always pass NULL for this value.

bufsz – Buffer size to be allocated

pool – Pool id from which buffer is to be allocated

Return Values

On success, returns the pointer to a valid data buffer. On error, returns NULL.

vnet_pbuf_free

Description

Frees a message block allocated by `vnet_pbuf_alloc()`. This function is implemented by the user and called by the driver.

Syntax

`void vnet_pbuf_free(void *`*hook*`, pbuf_t *`*mblkp*`, void *`*arg*`, uint16_t` *pool*`);`

Parameters

hook – currently unused by the `vnet` device driver; the driver will always pass NULL for this value.

mblkp - pointer to the message block to be freed

arg - currently ususued; the driver will pass NULL for this value

pool - pool id to which the buffer must be freed

vnet_buf_free

Description

Frees a buffer that is allocated using `vnet_buf_alloc()`. This is implemented by the user and called by the device driver.

Syntax

`int vnet_buf_free(void *`*hook*`, unsigned char *`*buf*`, void *`*arg*`,` `uint16_t` *pool*`);`

Parameters

hook – currently unused by the `vnet` device driver; the driver will always pass NULL for this value.

buf - pointer to the data buffer to be freed

arg - currently unused by the `vnet` device driver; the driver will always pass NULL for this value

pool - pool id to which the buffer must be freed

Return Values

On success, returns 0. On error, returns -1.

vnet_eth_open

Description

Probes a virtual network device in the target platform and if the device is found, the function initializes the virtual network device. On successful completion, this function returns an opaque handle that needs to be used in other API function calls that are targeted to a specific virtual network device.

Syntax

`void *vnet_eth_open(uint16_t` *vid*`, uint16_t` *did*`, int` *port*`, uint_t` *mpbase*`, void *`*receive_packet_queue*`, void *`*transmit_free_queue*`);`

Parameters

`vid` – vedor ID (see `vnet_ethapi.h`)

`did` – device ID (see `vnet_ethapi.h`)

port – virtual network device instance number; this is the value shown in the output of `ldm-list-bindings -e` *ndps-ldom* under the `DEVICE` column. For example, if output shows `network@4`, then user must pass 4 for the port number.

mpbase – base index into the mempool type array used in the application

receive_packet_queue – `fastq` into which ingress packets are queued by the device driver for processing by the application

transmit_free_queue – `fastq` into which packets whose transmission is complete are queued by the device driver for processing(freeing) by the application

Return Values

On success, returns a valid opaque handle. On failure, returns NULL.

vnet_eth_read

Description

Receives messages from the virtual network device instance specified by `ihandle`. This function can block in some circumstances (see description of `vnet_set_rxburst`).

Syntax

```
int64_t vnet_eth_read(void *ihandle, int chan);
```

Parameters

ihandle – opaque handle which was returned by `vnet_eth_open()`

chan – relative index into memory pool array into which packet must be freed after transmission completes

Return Values

On success, a value that indicates the number of frames received. On failure, returns -1.

vnet_eth_write

Description

Sends a message that is specified by the message block structure pointer. This function is non-blocking.

Syntax

```
int vnet_eth_write(void *ihandle, int chan, pbuf_t *txmp);
```

Parameters

ihandle – opaque handle which was returned by `vnet_eth_open()`

chan – relative index into memory pool array into which packet must be freed after transmission completes

txmp – message block pointer

Return Values

On success, returns 0. On failure, returns -1.

vnet_eth_ioc

Description

A catch-all configuration function that is used to configure and read device driver attributes.

Syntax

```
int vnet_eth_ioc(void *ihandle, ioc_cmd_t cmd, void *arg);
```

Parameters

ihandle – opaque handle which was returned by `vnet_eth_open()`

cmd – command to execute

arg – argument passed to the command

Return Values

On succes, returns 0. On error, returns -1.

vnet_eth_ioc Commands and Arguments

The `vnet` device driver uses the same commands as the Sun Netra DPS Ethernet device drivers like `nxge` and `ipge`. Some of the commands are not supported by the `vnet` device driver. For some of the commands, the argument type is also different. The following section outlines the commands and their arguments.

ETH_IOC_GET_MAC_ADDR

Description

Obtains the MAC address of the `vnet` port.

Arguments

Pointer to an array of `ETHERADDRL` bytes.

ETH_IOC_CHK_LINK

Description

Checks the `vnet` device link status. Display the status of the links on the console.

Arguments

None.

ETH_IOC_GET_LINK

Description

Obtains the vnet device link status, and returns it in the argument passed.

Arguments

```
(vnet_link_status_ioc_t *)arg

typedef struct {
       int             ldc_num;
       eth_event_t     status;
       int             prv_unused;

} vnet_link_status_ele_t;
```

where,

ldc_num - LDC channel number

status - link status in terms of a eth_event_t value; please see ethapi.h

```
typedef struct {
       int     instance;
       int     ele_cnt;
       int     error;
       char    status_ele[1];

} vnet_link_status_ioc_t;
```

where,

instance – virtual network device instance; this is the value displayed in the output of `ldm list-bindings -e` *ndps-ldom* under the DEVICE column. For example, if output shows network@4, then user must pass 4 for the instance number.

ele_cnt – number of vnet_link_status_ele_t elements for which user has allocated memory starting from status_ele

error – indicates any error encountered in the device driver while executing the command; currently only one error code is supported; if this value is 1 upon return from the ioctl call, then it implies that memory starting from `status_ele` is insufficient to store the status of all LDC channels in a `vnet` device; if this value is 0 upon return from the ioc call, then it means the call was successful

status_ele – beginning of the memory region where all `vnet_link_status_ele_t` for each LDC channel of a `vnet` device is stored. The user needs to allocate this memory before the call. This memory must accomodate at least `ele_cnt` elements.

ETH_IOC_ADD_MCAST_ADDR

Description

Adds a multicast address into the multicast table of the `vnet` device and the connected virtual switch.

Arguments

Pointer to a byte array of length `ETHERADDRL` that contains the multicast address.

ETH_IOC_DEL_MCAST_ADDR

Description

Deletes a multicast address from the multicast table for the `vnet` device and the connected virtual switch.

Arguments

Pointer to a byte array of length `ETHERADDRL` that contains the multicast address.

ETH_IOC_SHOW_MCAST_ADDRS

Description

Displays or obtains a copy of the `vnet` device multicast table.

Arguments

```
(vnet_mc_table_t *)arg

typedef struct {
       int             ele_cnt;
       int             error;
       boolean_t       display;
       unsigned char   pad[4];
```

```
        char            data[1];
} vnet_mc_table_t;
```

where,

ele_cnt – number of MAC addresses for which user has allocated space in memory region starting at data

error – indicates any error encountered in the library while executing the `ioctl`. Currently, only one error code is supported. If this value is 1 on return from the `ioctl` call, memory starting from data is insufficient to store all the MAC addresses in the multicast table. If this value is 0 on return, then the call was succesful and all entries were copied.

display – If set to `B_TRUE` by the user, the driver will print the multicast table on the console.

data – beginning of the memory region where all MAC addresses are stored. The user needs to allocate this memory before the call; this memory must accomodate at least `ele_cnt` elements

ETH_IOC_GET_STATS

Description

Obtain statistics about a `vnet` instance.

Arguments

```
(vnet_eth_kstat_t *)arg
```

```
typedef struct {
        /* Link Input/Output stats */
        uint64_t ipackets;        /* # rx packets */
        uint64_t ierrors;         /* # rx error */
        uint64_t opackets;        /* # tx packets */
        uint64_t oerrors;         /* # tx error */
        /* MIB II variables */
        uint64_t rbytes;          /* # bytes received */
        uint64_t obytes;          /* # bytes transmitted */
        uint32_t multircv;        /* # multicast packets received */
        uint32_t multixmt;        /* # multicast packets for xmit */
        uint32_t brdcstrcv;       /* # broadcast packets received */
        uint32_t brdcstxmt;       /* # broadcast packets for xmit */
        uint32_t norcvbuf;        /* # rcv packets discarded */
        uint32_t noxmtbuf;        /* # xmit packets discarded */
        /* Tx Statistics */
```

```
        uint32_t tx_no_desc;     /* # out of transmit descriptors */
        uint32_t tx_qfull;      /* pkts dropped due to qfull in vsw */
        uint32_t tx_pri_fail;    /* # tx priority packet failures */
        uint64_t tx_pri_packets; /* # priority packets transmitted */
        uint64_t tx_pri_bytes;   /* # priority bytes transmitted */
        /* Rx Statistics */
        uint32_t rx_allocb_fail; /* # rx buf allocb() failures */
        uint32_t rx_vio_allocb_fail; /* # vio_allocb() failures */
        uint32_t rx_lost_pkts;   /* # rx lost packets */
        uint32_t rx_pri_fail;    /* # rx priority packet failures */
        uint64_t rx_pri_packets; /* # priority packets received */
        uint64_t rx_pri_bytes;   /* # priority bytes received */
        /* Callback statistics */
        uint32_t callbacks;              /* # callbacks */
        uint32_t dring_data_acks;       /* # dring data acks recvd  */
        uint32_t dring_stopped_acks;    /* # dring stopped acks recvd
        */
        uint32_t dring_data_msgs;        /* # dring data msgs sent */

} vnet_eth_stats_t;

typedef struct {
        int                     ldc_num;
        unsigned char           pad[4];
        vnet_eth_stats_t        stats;

} vnet_eth_kstat_ele_t;
```

where,

ldc_num – LDC channel number

stats – statistics for the LDC channel

```
typedef struct {
        int             instance;
        int             ele_cnt;
        int             error;
        unsigned char   pad[4];
        char            stats_ele[1];

} vnet_eth_kstat_t;
```

where,

instance – virtual network device instance. This is the value displayed in the output of `ldm list_bindings -e` *ndps-ldom* under the `DEVICE` column. For example, if output shows `network@4`, then user must pass 4 for the instance number

ele_cnt – number of `vnet_eth_kstat_ele_t` elements for which user has allocated memory starting from `stats_ele`

error – indicates any error encountered in the library while executing the `ioctl`. Currently, only one error code is supported. If this value is 1 upon return from the `ioctl` call, then it implies that memory starting from `status_ele` is insufficient to store the status of all LDC channels in a `vnet` device. If this value is 0 upon return from the `ioctl` call, then it means the call was successful

stats_ele – beginning of the memory region where all `vnet_eth_kstat_ele_t` for each LDC channel of a `vnet` device is stored. The user needs to allocate this memory before the call. This memory must accomodate at least `ele_cnt` elements.

`ETH_IOC_SHOW_STATS`

Description

Display the `vnet` device statistics on the console.

Arguments

None.

`ETH_IOC_SET_MAC_ADDR`

`ETH_IOC_SET_PROMISC`

`ETH_IOC_SET_MAX_FRAME_SIZE`

`ETH_IOC_SET_ADDR_FILTER`

`ETH_IOC_SET_MAC_TBL`

`ETH_IOC_SET_VLAN_TBL`

`ETH_IOC_SET_RDC_GRP`

`ETH_IOC_SHOW_RDC_GRPS`

`ETH_IOC_BIND_RDC_GRP`

`ETH_IOC_GET_PORTINFO`

ETH_IOC_SHOW_PORTINFO

ETH_IOC_SET_CLASSIFY

ETH_IOC_CHK_ERRS

Description

Not supported by the vnet device driver.

vnet_eth_get_mac_addr

Description

Obtains the MAC address of a vnet device.

Syntax

```
int vnet_eth_get_mac_addr(void *ihandle, unsigned char *addr);
```

Parameters

ihandle – opaque handle returned by vnet_eth_open()

addr – pointer to a user space buffer of length ETHERADDRL bytes where device MAC address is copied by the driver

Return Values

On success, returns 0. On failure, returns -1.

vnet_eth_flush

Description

This function accomplishes two important tasks:

1. Flush packets that are pending transmission in the vnet transmit buffers and free the associated transmit resources
2. Functionality needed to re-initialize the vnet device in case it has been reset.

Given the nature of the VIO protocol that is used by vnet drivers, it is possible that when an application is transmitting packets in bursts, at the end of a burst, some packets are left in the transmit buffers of the vnet device waiting for the peer vnet device to consume them. In such a scenario, this function must be called to signal the peer vnet device that there are pending packets for it to consume. When an

application is transmitting packets continuously by calling `vnet_eth_write()`, the same functionality is achieved by `vnet_eth_write()`. When an application has stopped transmitting packets using `vnet_eth_write()`, it must call `vnet_eth_flush()` until it returns 0 to ensure that there are no more pending transmits.

It is also necessary that the application call this function in order to re-initialize the `vnet` device, in case the device is reset, when the application is not transmitting packets using `vnet_eth_write()`.

If the application is sending packets by calling `vnet_eth_write()`, the signalling necessary for re-initializing the device is achieved in `vnet_eth_write()`. If not, the application must call this function in a timely manner.

Syntax

```
int vnet_eth_flush(void *ihandle, int chan);
```

Parameters

ihandle – opaque handle returned by `vnet_eth_open()`

chan – relative index into memory pool array into which packet must be freed after transmission completes

Return Values

If all pending transmissions are completed, returns 0. If more packets are pending transmission, returns 1.

vnet_set_rxburst

Description

A `vnet` device can have several LDC channels to other `vnet` devices and also to virtual switches.

`vnet_eth_read()` reads packets from the ingress FIFOs of each of these LDC channels in a round robin manner. Each `vnet` device has an associated Receive Burst Size value that determines the budget available for each LDC channel in units of packets, in one call of `vnet_eth_read()` by the application .

A non-zero Receive Burst Size value implies that atmost Receive Burst Size packets are read from a LDC channel. If more packets than Receive Burst Size are available to be processed, the `vnet_eth_read()` function will stop processing that LDC channel and will proceed to process another LDC channel.

If Receive Burst Size value is non-zero and packets available are fewer than the value, then those packets are processed and `vnet_eth_read()` will continue to process another LDC channel.

If the Receive Burst Size is zero, then it implies that the `vnet_eth_read()` function will process the LDC channel until there are no more packets to be processed. That is, this represents an infinite burst size value.

When there are more than one LDC channels in a `vnet` and if the Receive Burst Size value for the `vnet` device is zero, then there is a possibility that an LDC channel that has heavy traffic arriving on it can starve the other LDC channels. In the same scenario, if a LDC channel is receiving continuous traffic, `vnet_eth_read()` function can be blocked processing that LDC channel.

This function is used by the user application to set the Receive Burst Size of a `vnet` device to a desired value.

The default value for Receive Burst Size is zero.

Syntax

```
int vnet_set_rxburst(void *ihandle, unsigned int burst_size);
```

Parameters

ihandle – opaque handle returned from `vnet_eth_open()`

burst_size – burst size value for the `vnet` device

Return Values

On success, returns 0. On failure, returns -1.

vnet_get_rxburst

Description

This function is used by the user application to read the current setting of the Receive Burst Size value.

Syntax

```
unsigned int vnet_get_rxburst(void *ihandle);
```

Parameters

ihandle – opaque value returned by `vnet_eth_open()`

Return Values

Returns the current Receive Burst Size value for the vnet device.

Note – A vnet device does not have a notion of DMA channels. Nevertheless, some functions in the vnet Driver API must be passed a parameter called chan. This value is used by the vnet driver in conjunction with the mpbase value passed in the vnet_eth_open() to determine the pool ID parameter that needs to be passed to functions like vnet_pbuf_alloc(). The driver calculates the pool ID as pool_id = mpbase + chan.

---

vnet Device Driver Tunables

Description

```
extern uint_t vnet_rxoff_var
```

Enables the start addresses of packets received over a vnet device to be offset by cache aligned addresses. Offsets are 64B, 128B, or 192B. Default value is 1 (turned on).

```
extern uint_t vnet_rxoff_var_n2_2
```

vnet_rxoff_var_n2_2 enables N2 2.2 behaviour for start address offsets of packets received over a vnet device. Offsets are 64B, 128B, 320B, and 384B. Default value is 0 (turned off).

Notes

Note 1

When both vnet_rxoff_var and vnet_rxoff_var_n2_2 are set to 0, a constant offset of 128B is used for packets received over all vnet devices being used.

Note 2

The actual offset in a received packet is the pre-determined packet offset plus 6B. So actual offsets when vnet_rxoff_var is enabled are 70B, 134B, and 198B. The actual offsets when vnet_rxoff_var_n2_2 is enabled are 70B, 134B, 326B, and 392B.

Note 3

The offset for received packets are determined per-vnet device. So, for a given vnet device, the offset is constant for all packets received. When multiple vnet devices are used, each vnet device can have different offsets. When the number of vnet devices used are more than the set of offset values, the offsets can be the same for some of the vnet devices.

Note 4

This note explains the buffer management in the Sun Netra DPS vnet driver. In this note, all references to *chained packet* imply a packet that is created by linking several mblks using their b_cont field and each mblk points to a data buffer.

Buffer Allocation

The vnet driver will always call vnet_pbuf_alloc() to allocate buffers for packets or frames. It does not support buffer reuse from transmit free queue. All mblks allocated by the vnet driver have their inport field set to INPORT_TYPE_VNET (see vnet_ethapi.h). This field allows applications to identify buffers that are allocated by the vnet driver.

Buffer Freeing When transmit_free_queue *Is NULL*

If the user application passes a NULL in the transmit_free_queue argument of vnet_eth_open(), then the vnet driver calls vnet_pbuf_free() to free the buffers that have completed transmission. If the user application has transmitted a chained packet, then the vnet driver will walk the chain and free the individual mblks in the chain.

Buffer Freeing When transmit_free_queue *Is Not NULL*

If user application passes a non-NULL valid fast queue in the transmit_free_queue argument of vnet_eth_open(), then the vnet driver en-queues all frames whose transmission is complete into the transmit_free_queue. The user application must de-queue such packets from this fast queue for further processing.

If user application has transmitted a chained packet, then the head mblk of the chain is en-queued by the driver into the transmit_free_queue. The user is responsible for freeing the individual mblks in this chain by walking the chain.

If the user application transmits an untagged frame that is destined to another vnet device that uses VLAN Tagging, then the vnet driver may prepend an mblk to the frame during the process of tagging. This means that, in this scenario, the vnet driver creates a chain of mblks to complete the transmission. When the transmission is complete, the chained frame is en-queued into the transmit_free_queue. So the user application must always check for a chain before freeing the mblk it dequeues from the transmit_free_queue.

Note 5

This note gives an example of using the receive packet queue and the transmit free queue in vnet driver.

Example:

```
fastq_t rxfq_dram[MAX_VNET_DEVS], tx_freeq_dram[MAX_VNET_DEVS];

vnet_eth_open(VNET_VID, VNET_DID, port, poolid, rxfq_dram[port],
tx_freeq_dram[port]);
```

Note 6

This note explains VLAN support for vnet devices in Sun Netra DPS.

The Sun Netra DPS vnet driver supports VLANs. A Sun Netra DPS vnet interface can be assigned a VLAN by using the respective ldm command. Either Port VLANs or VLAN Tagging can be used for Sun Netra DPS vnet interfaces.

When a Sun Netra DPS vnet interface is connected to a Linux vnet interfaces, then VLAN is not supported. This is because Linux vnet driver does not support VLANs. Hence, the Sun Netra DPS vnet interface initialization is known to fail when they are connected to Linux vnet interfaces that have VLANs enabled.

When a Sun Netra DPS vnet interface is connected to a Solaris OS vnet interface, VLAN is supported. The Sun Netra DPS vnet driver will support the following functionality:

- When Sun Netra DPS vnet interface uses VLAN Tagging and Solaris OS vnet interface uses Port VLAN, Sun Netra DPS vnet driver will untag frames before sending it to the Solaris OS vnet interface
- When Sun Netra DPS vnet interface uses Port VLAN and Solaris OS vnet interface uses VLAN Tagging, Sun Netra DPS vnet driver will tag the frames before sending it to the Solaris OS vnet interface

CHAPTER 5

Fast Queue API

This chapter describes the Fast Queue API functions. Topics include:

Fast Queue API Introduction

The Fast Queue API provides a facility where threads can exchange or pass data in a first-in-first-out (FIFO) order. This API provides routines for creating and using fast queues, a fast communication mechanism between two or more threads. The fast queues are based on circular buffers, which offer an advantage over `teja` queues that are for one consumer and one producer. The fast queues are poll-driven and are more efficient for high packet rates. The fast queue API is not thread safe and needs to be protected with locks when it is used by more than one consumer or producer.

The API functions are defined in `fastq.h`, located in the `src/dev/net/include` directory of the `SUNWndps` package.

Fast Queue API Function Descriptions

fastq_create

Description

Creates a new instance of the fast queue. The function yields the CPU for a few cycles by executing a long latency instruction when the queue is full.

Syntax

```
fastq_t  fastq_create(size_t size)
```

Parameters

size – Size of the queue. Required size is the power of 2.

Return Values

`fastq_t` – Returns a fast queue instance or `NULL` if it fails.

fastq_enqueue

Description

Enqueues a node pointer into the fast queue. If `-1` is returned, the queue is either full or some other error has occurred.

Syntax

```
fastq_enqueue(queue, node)
```

Parameters

queue – Queue to enqueue to.

node – Pointer to the node to enqueue.

Return Values

`int` – Returns `0` if successful or `-1` if it fails.

fastq_dequeue

Description

Dequeues a pointer to a node from the queue. The function yields the CPU for a few cycles by executing a long latency instruction when the queue is empty.

Syntax

`fastq_dequeue(`*queue*`)`

Parameters

queue – Queue selected for dequeue.

Return Values

`void *` – Returns a pointer to the dequeued node or `NULL` if the queue is empty.

fastq_enqueue_noyield

Description

Enqueues a node pointer into the fast queue. If `-1` is returned, the queue is either full or some other error has occurred.

Syntax

`fastq_enqueue_noyield(`*queue*`,` *node*`)`

Parameters

queue – Queue selected for dequeue.

node – Pointer to the node to enqueue.

Return Values

int – Returns 0 if successful or -1 if it fails.

fastq_dequeue_noyield

Description

Dequeues a pointer to a node from the queue.

Syntax

`fastq_dequeue_noyield(`*queue*`)`

Parameters

queue – Queue selected for dequeue.

Return Values

void * – Returns a pointer to the dequeued node or NULL if the queue is empty.

fastq_get_size

Description

Returns a value that is the depth of the queue at a given point of time.

Syntax

`fastq_get_size(`*queue*`)`

Parameters

queue – Queue requested to obtain size.

Return Values

`int` – Returns the number of elements in the queue or 0 if empty.

fastq_is_empty

Description

Checks if the queue is empty.

Syntax

`fastq_is_empty(`*queue*`)`

Parameters

queue – Queue selected to test.

Return Values

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

fastq_is_full

Description

Checks if the queue is full.

Syntax

`fastq_is_full(`*queue*`)`

Parameters

queue – Queue selected to test.

Return Values

`void *` – Returns `1` if the queue is full or `0` if the queue is not full.

CHAPTER 6

Interprocess Communication API

This chapter describes the Interprocess Communication (IPC) API. Topics include:

Interprocess Communication API Introduction

The Interprocess Communication (IPC) mechanism provides a means to communicate between processes that run in a domain under the Sun Netra DPS runtime environment and processes in a domain with a control plane operating system.

Common Programming Interfaces

The API described in this section is available on all operating environments that support IPC communications with a `LWRTE` domain. The `tnipc.h` header located in the `src/common/include` directory of the `SUNWndps` package defines the interface and must be included in source files using the API. The header file defines a number of IPC protocol types. User-defined protocols must not be in conflict with these predefined types.

ipc_connect

Description

Registers a consumer with an IPC channel. The opaque handle that is returned by a successful call to this function must be passed to access the channel using any of the other interface functions.

Syntax

```
ipc_handle_t
ipc_connect(uint16_t channel, uint16_t ipc_proto)
```

Parameters

channel – ID of channel

ipc_proto – Protocol type of IPC messages that are expected

Return Values

NULL in case of `failure`.

IPC handle otherwise. This handle must be passed to the `tx`/`rx`/free functions.

ipc_register_callbacks

Description

Registers callback functions for the consumer of an IPC channel. When a message is received by the IPC framework, this function strips the IPC header from the message and calls the `rx_hdlr` function with the content of the message.

Syntax

```
int
ipc_register_callbacks ipc_hdl,
event_handler_ft  evt_hdler,
rx_handler_ft ipc_hdler,
caddr_t arg){
```

Parameters

ipc_hdl – Handle for IPC channel, obtained from `ipc_register_callbacks()`.

evt_hdlr – Function to handle link events.

rx_hdlr – Function to handle received messages.

arg – Opaque argument that the framework will pass back to the handler functions.

Return Values

`IPC_SUCCESS`

`EFAULT` – Invalid handle

ipc_tx

Description

Transmits messages over IPC. The message is described by the `mblk` passed to the function. To make the function as efficient as possible, the function makes some nonstandard assumptions about the messages:

- There are 8 bytes of headroom in the data buffer before the messages.
- Messages are contained in a single data buffer.

As the memory containing the message is not freed inside the function, the caller must deal with memory management accordingly.

Syntax

```
int
ipc_tx(mblk_t *mp, ipc_handle_t ipc_hdl)
```

Parameters

mp – Pointer to message block describing the messages.

ipc_hdl – Handle for IPC channel, obtained from `ipc_connect()`.

Return Values

`IPC_SUCCESS`

`EIO` – The write to the underlying media failed.

ipc_rx

Description

At this time, the only way to receive messages is through the callback function. In `LWRTE`, the callback function is called when the polling context finds a message on the channel. In Solaris user space, the callback is hidden in the framework, making the message available to be read by the `read()` system call.

Syntax

`mblk_t *ipc_rx(ipc_handle_t` *ipc_hdl*`)`

Parameters

ipc_hdl – Handle for IPC channel, obtained from `ipc_connect()`.

ipc_free

Description

The IPC framework allocates memory for messages that are received using its available memory pools. The consumer of an IPC message must call this function to return the memory to that pool.

Syntax

```
void
ipc_free(mblk_t *mp, ipc_handle_t ipc_hdl)
```

Parameters

mp – Pointer to message block describing message to be freed.

ipc_hdl – Handle for IPC channel, obtained from `ipc_connect()`.

IPC Framework Programming Interfaces

In the Sun Netra DPS runtime environment domain, the interfaces described in the section "Common Programming Interfaces" on page 170 are used to communicate with other domains using IPC. Before this infrastructure can be utilized, it must be initialized. Once it is initialized, because there are no interrupts, the developer must ensure that every channel is polled periodically. This section describes the API for these tasks.

To use this function, the `lwrte_ipc_if.h` header file must be included where needed. The file is located in the `lib/ipc/include` directory of the `SUNWndps` package.

tnipc_init

Description

This function must be called in the initialization routine. This function must be called after the Oracle VX Server (`ldom`) software framework has been initialized, that is, `mach_descrip_init()`, `lwrte_cnex_init()` and `lwrte_init_ldc()` must be called first.

Syntax

```
int
tnipc_init()
```

Return Values

`0` – Success

`EFAULT` – Too many channels in machine description

`ENOENT` – Global configuration channel not defined

tnipc_poll

Description

To receive messages or event notifications for any IPC channel, this function must be called periodically. For example, this function may be called as part of the main loop in the statistics thread. When a message is received, this function ensures that the callback function registered for the channel and IPC type is called.

Syntax

```
int
tnipc_poll()
```

Return Values

This function always returns 0.

Polling through the `tnipc_poll()` API is adequate for most IPC channels carrying low bandwidth control traffic. For higher throughput channels, the polling can be moved to a separate strand, using the following API functions:

- `tnipc_register_local_poll`
- `tnipc_local_poll`
- `tnipc_unregister_local_poll`

tnipc_register_local_poll

Description

Removes the channel identified by the handle passed to the function from the pool of channels polled by the `tnipc_poll()` function. This function returns an opaque handle that must be passed to the `tnipc_local_poll()` function.

Syntax

```
ipc_poll_handle_t
tnipc_register_local_poll(ipc_handle_t ipc_hdl)
```

Parameter

ipc_hdl – Channel handle obtained from the `ipc_connect()` API call.

Return Values

NULL – Invalid input

Opaque handle to be passed to the `tnipc_local_poll()` call.

tnipc_local_poll

Description

Works the same way as `tnipc_poll()`, except that only the channel identified by the handle is polled. If there is data on the channel, the `rx` callback will be called.

Syntax

```
int
tnipc_local_poll(ipc_poll_handle_t poll_hdl)
```

Parameter

poll_hdl – The handle obtained from the `tnipc_register_local_poll()` API call

Return Values

This function always returns 0.

`tnipc_unregister_local_poll`

Description

Reverses the effect of the `tnipc_register_local_poll()` call and places the channel identified by the handle back into the common pool polled by `tnipc_poll()`.

Syntax

```
int
tnipc_unregister_local_poll(ipc_poll_handle_t poll_hdl)
```

Parameter

poll_hdl – Handle obtained from the `tnipc_register_local_poll()` API call

Return Values

This function always returns 0.

IPC Programming Interfaces for Solaris Domains

In the Solaris OS, the IPC mechanism can be used either from user space or from kernel space.

User Space

To use an IPC channel from the Solaris user space, the character-driver interfaces are used. A program opens the `tnsm` device (`/devices/pseudo/tnsm@0:tnsm`), issues an `ioctl()` call to connect the device to a particular channel, and then uses `read()` and `write()` calls to send and receive messages. To use the `ioctl()` interface, the `tnsm.h` header file must be included. This file is located in the directory `src/solaris/include/sys` in the `SUNWndps` package.

Before any of the interfaces can be used, the `tnsm` driver must be installed and loaded. This is done using the `pkgadd` system administration command to install the `SUNWndpsd` package on the Solaris domains that use IPC for communication.

The `open()`, `close()`, `read()`, and `write()` interfaces are described in their respective man pages.

The `open()` call on the `tnsm` driver creates a new instance for the specific client program. Before the `read()` and `write()` calls can be used, the `TNIPC_IOC_CH_CONNECT` `ioctl` must be called. The arguments for this `ioctl` are the channel ID and IPC type to be used for messages by this instance.

Kernel

In the kernel, the interfaces described in "Common Programming Interfaces" on page 170 are used.

CHAPTER 7

Fastpath Manager API

This chapter describes the Fastpath Manager API. Topics include:

Fastpath Manager API Introduction

The Fastpath Manager API provides a means to register tasks that must be run periodically. For example, the user can use this API to check whether a link is up or to perform other health checks. The Fastpath Manager is included as part of the command-line interface (`cli`) library. The user can find the header file declaring the API (`lwrte_fastpath_mgr`.h) in the `include` directory for that library.

The Fastpath Manager is run on a dedicated thread, and in a logical domain environment usually polls the IPC channels, in particular the global control channel. The granularity of the interval length for checking tasks that can be registered with the framework is one millisecond. However, the user must be aware that there is no pre-emption, so the actual granularity depends on the length and number of tasks that are registered.

Fastpath Manager API Function Descriptions

fastpath_mgr_init

Description

Initialization for the Fastpath Manager framework. This function must be called in the `init` routine of applications to use the framework.

Syntax

```
void fastpath_mgr_init()
```

Parameters

None

Return Values

None

fastpath_mgr_process

Description

Implements the periodic execution of scheduled tasks. This function must run on its own strand.

Syntax

`void fastpath_mgr_process(boolean_t` *poll_ipc*`)`

Parameters

poll_ipc – Indication whether the IPC channels are polled in this thread.

Return Values

None

fastpath_mgr_register_event_handler

Description

This interface is used to register functions that periodically check for a condition.

Syntax

```
fastpath_mgr_handle_t

fastpath_mgr_register_event_handler(status_check_ft check_fun,
event_cb_ft event_cb, void *args,
int interval);
```

Parameters

check_fun – Function that performs check. Must return 0 if the check passed.

event_cb – Optional handler for events. If present, this function is called when the *check_fun* returns a value other than 0. That value is passed to the event_cb to identify the event.

args – Argument passed to checking and callback functions.

interval – Frequency of call to *check_fun* (in milliseconds).

Return Values

NULL in case of error.

Handle in case of success. This handle is needed to unregister the task.

fastpath_mgr_unregister_event_handler

Description

Unregisters functions from the Fastpath Manager.

Note – This function may be called by the event callback registered through `fastpath_mgr_register_event_handler()`, but must not be called by the checking function registered in that call.

Syntax

```
int fastpath_mgr_unregister_event_handler(fastpath_mgr_handle_t
hdl)
```

Parameters

hdl – Handle obtained from `fastpath_mgr_register_event_handler`().

Return Values

0 – Success

-1 – Failure

`fastpath_mgr_check`

Description

Runs all check functions whose interval have expired. This function is an alternative entry point into the Fastpath Manager that checks whether there are any registered functions that should be run at the time of the call. If this entry point is used, the user must make ensure that this function is run with a sufficient frequency.

Syntax

`void fastpath_mgr_check(boolean_t` *poll_ipc*)

Parameters

poll_ipc – Indication whether `ipc_poll`() is called in the function.

Return Values

None

CHAPTER 8

Access Control List Library API

This chapter describes the Access Control List (ACL) library API. Topics include:

Access Control List Library API Introduction

The Access Control List (ACL) library for Sun Netra DPS classifies IPv4 packets using a set of rules.

The classification can be done using the source/destination addresses and ports, as well as the protocol and the priority of the packet.

The algorithms are used in the library trade memory for speed. The rules are preprocessed to achieve high lookup rate while using a lot of memory.

Algorithms

The ACL library uses various algorithms to classify the packets.

Hybrid Algorithm

This algorithm finds the Longest Matching Prefixes of the source and destination addresses and searches for the highest priority rule among all those rules matching the particular prefix pair. The Longest Prefix Match algorithm can use either Binary Search on Prefix Lengths (BSPL) or TRIE lookup (see "TRIE Algorithm" on page 182). The Longest Prefix Match algorithm provides a set of interface that can be used independently from the interface provided in the ACL library.

This algorithm is well suited for rulesets with a large number of rules (millions) where only a few rules (dozens) remain after the prefix lookups. The data structures can be updated quickly, enabling the addition or removal of thousands of rules each second. The initial rule insertion is even faster, that is, millions of rules can be added in a few seconds.

Binary Search on Prefix Lengths

Binary Search on Prefix Lengths (BSPL) finds the longest matching prefix of an address by doing binary search on prefix length. That is, starting the search in the hash table containing median length prefixes and continuing in a hash table with longer prefixes if a match is found, shorter prefixes otherwise.

TRIE Algorithm

The TRIE (retrieval) algorithm uses a three-level prefix tree to find the longest matching prefix of an address.

Swapping

The ACL functions use a pointer to a data structure that contains all data necessary to change the ruleset or match packets against them. This poiner enables changing the rulesets without disturbing the packet classification, by having two datasets and

using one of them to classify packets while applying changes to the other. Once the changes are made the datasets can be swapped without affecting lookup performance. That is, no locks are necessary.

Remapping

The ACL data structures can be copied to a new buffer or remapped to a new address without breaking the lookup algorithm. The ACL data structures enables preparing them in a domain and using them in another. That is, rule management and packet classification can be performed in separate domains if required.

Data Types

Data types consist of packet and rule types.

Packet Type

The packets used by the ACL library are standard TCP/UDP over IPv4 packets.

Rule Type

A rule consists of six fields to match against TCP/IP packets:

- Source address prefix
- Destination address prefix
- Source port range
- Destination port range
- Type of service mask
- IP protocol mask

The rule also contains a classification value (color) that is returned by the lookup algorithm when the packet matches the rule. The rules are ordered by the color in ascending order: the lookup returns the color of the lowest-color matching rule.

ACL Library API Function Descriptions

acl_init

Description

Initialization routine for the ACL. Based on the selected algorithm, this routine fills up the given buffer with data necessary to insert and remove rules and lookup packets. That is, the caller must allocate a buffer and pass it to `acl_init` and subsequent `acl_*` calls.

Error code is written in the provided variable.

Syntax

```
void *acl_init(void *buf, size_t size, int alg, int *error);
```

Parameters

buf – Pointer to the buffer to be filled with initialized data

size – Size of the buffer

alg – Algorithm selector (see "Algorithms" on page 182), in short:
`ACL_ALG_HYBRID_BSPL` – Hybrid algorithm, LPM is using BSPL
`ACL_ALG_HYBRID_TRIE` – Hybrid algorithm, LPM is using TRIE
`ACL_ALG_HICUT` – HiCut

error – Pointer to a variable where the error code is to be written

Return Values

On success, it returns a pointer to the initialized ACL data or NULL in case of error. The error code is returned in case of error.

acl_insert

Description

Inserts rules. Takes the rules from the given array and inserts them into the pre-initialized data structures. Then performs the necessary preprocessing and optimization, leaving the dataset ready to be used for packet lookup.

Syntax

```
int acl_insert(void *acl, rule_t *rule, int num);
```

Parameters

acl – Pointer to the initialized ACL data

rule – Pointer to the first rule in the array

num – Number of rules in the rule array

Return Values

On success, it returns zero, or an error code in case of error.

acl_remove

Description

Removes rules. Takes rules from the given array and removes each of them from the data structures.

Syntax

```
int acl_remove(void *acl, rule_t *rule, int num);
```

Parameters

acl – Pointer to the initialized ACL data

rule – Pointer to the first rule in the array

num – Number of rules in the rule array

Return Values

On success, it returns zero, or an error code in case of error.

acl_lookup

Description

Lookup packet. Matches the packet against the rules and returns the classification value.

Syntax

```
color_t acl_lookup(void *acl, packet_t *packet);
```

Parameters

acl – Pointer to the initialized ACL data

packet – Pointer to the packet to be processed

Return Values

Returns the color of the lowest-color matching rule or the default color value if none of the rules matches the packet.

acl_list

Description

Lists the current ruleset. Copies rules into the provided array.

Syntax

```
int acl_list(void *buf, rule_t *rule, int num);
```

Parameters

buf – Pointer to the initialized ACL data

rule – Array of rules to copy to

num – Maximum number of rules to copy

Return Values

On success, returns the number of rules. If there are more rules to list than the provided array can store, then return (`-num`).

Error Codes

`ACL_INIT_OK` – Initialization was successful
`ACL_INIT_FAILED` – Initialization has failed
`ACL_INIT_UNKNOWN_ALG` – Invalid algorithm was passed
`ACL_INIT_MEMORY_ERROR` – Buffer size is too small
`ACL_INVALID_MAGIC` – Corrupted data in ACL buffer

LPM - Trie API Function Descriptions

trie_create

Description

Initializes a three level 16-8-8 stride trie data structure using the provided buffer.

Syntax

```
void *trie_create(void *buf);
```

Parameters

buf – Pointer to the trie data to be initialized

Return Values

On success, returns the pointer to the trie data structure. Return NULL on failure.

trie_get_buf

Description

Get the pointer to the buffer provided to trie_create.

Syntax

```
void *trie_get_buf(void *trie);
```

Parameters

trie – Pointer to the trie data structure.

Return Values

Returns the pointer to the trie buffer.

trie_add_prefix

Description

Add prefix to trie.

Syntax

```
int trie_add_prefix(void *trie, prefix_t *prefix, prefix_id_t value);
```

Parameters

trie – Pointer to the trie data structure.

prefix – Pointer to the prefix to be added to trie.

value– Value to be returned when prefix matches.

Return Values

On success, return zero. Otherwise, return non-zero value.

trie_remove_prefix

Description

Remove prefix from trie.

Syntax

```
int trie_remove_prefix(void *trie, prefix_t *prefix);
```

Parameters

trie – Pointer to the trie data structure.

prefix – Pointer to the prefix to be removed from trie.

Return Values

On success, return zero. Otherwise, return non-zero value.

trie_lookup

Description

Find the longest matching prefix for the given address using trie algorithm and return the prefix ID.

Syntax

```
prefix_id_t trie_lookup(void *trie, address_t address);
```

Parameters

trie – Pointer to the trie data structure.

address – IPv4 address to be looked up.

Return Values

Prefix ID of the longest matching prefix

LPM - BSPL API Function Descriptions

bspl_create

Description

Initializes a BSPL table using the provided buffer.

Syntax

`void *bspl_create(void *`*hybrid*);

Parameters

hybrid – Pointer to the hybrid data structure (used by acl_malloc). If input is NULL, then teja_malloc will be used to allocate BSPL table.

Return Values

On success, returns the pointer to the initialized BSPL table. Return NULL on failure.

bspl_destroy

Description

Free up resources associated with the BSPL table.

Syntax

`void bspl_destory(void *`*bspl*);

Parameters

bspl – Pointer to the BSPL table.

Return Values

None.

bspl_add_prefix

Description

Add prefix to bspl.

Syntax

`int bspl_add_prefix(void *`*bspl*`, prefix_t *`*prefix*`, prefix_id_t` *value*);

Parameters

bspl – Pointer to the bspl image.

prefix – Pointer to the prefix to be added to BSPL table.

value– Value to be returned when prefix matches.

Return Values

On success, return zero. Otherwise, return non-zero value.

bspl_add_markers

Description

Add markers and BMPs to guide the binary search algorithm. It must be called before the newly added prefixes can be used.

Syntax

`void bspl_add_markers(void *`*bspl*);

Parameters

bspl– Pointer to the BSPL table.

Return Values

None.

bspl_remove_prefix

Description

Remove prefix from *bspl*.

Syntax

```
int bspl_remove_prefix(void *bspl, prefix_t *prefix);
```

Parameters

bspl– Pointer to the BSPL table.

prefix – Pointer to the prefix to be removed from BSPL table.

Return Values

On success, return zero. Otherwise, return non-zero value.

bspl_lookup

Description

Find the longest matching prefix for the given address using BSPL algorithm and return the prefix ID.

Syntax

```
prefix_id_t bspl_lookup(void *bspl, address_t address);
```

Parameters

bspl– Pointer to the BSPL table.

address – IP address to be looked up.

Return Values

Prefix ID of the longest matching prefix

bspl6_create

Description

Initializes a BSPL table for IPv6 using the provided buffer.

Syntax

```
void *bspl6_create(void *hybrid);
```

Parameters

hybrid – Pointer to the hybrid data structure (used by acl_malloc). If input is NULL, then teja_malloc will be used to allocate BSPL table.

Return Values

On success, returns the pointer to the initialized BSPL table. Return NULL on failure.

bspl6_destroy

Description

Free up resources associated with the IPv6 BSPL table.

Syntax

```
void bspl6_destory(void *bspl);
```

Parameters

bspl – Pointer to the BSPL table.

Return Values

None.

bspl6_add_prefix

Description

Add IPv6 prefix to bspl.

Syntax

```
int bspl6_add_prefix(void *bspl, prefix6_t *prefix, prefix_id_t value);
```

Parameters

bspl – Pointer to the BSPL image.

prefix – Pointer to the prefix to be added to BSPL table.

value– Value to be returned when prefix matches.

Return Values

On success, return zero. Otherwise, return non-zero value.

bspl6_add_markers

Description

Add markers and BMPs to guide the binary search algorithm for IPv6. It must be called before the newly added IPv6 prefixes can be used.

Syntax

`void bspl6_add_markers(void *`*bspl*);

Parameters

bspl – Pointer to the BSPL table.

Return Values

None.

bspl6_remove_prefix

Description

remove IPv6 prefix from bspl.

Syntax

`int bspl_remove_prefix(void *`*bspl*, `prefix6_t *`*prefix*);

Parameters

bspl– Pointer to the BSPL table.

prefix – Pointer to the prefix to be removed from BSPL table.

Return Values

On success, return zero. Otherwise, return non-zero value.

bspl6_lookup

Description

Find the longest matching prefix for the given IPv6 address using BSPL algorithm and return the prefix ID.

Syntax

`prefix_id_t bspl6_lookup(void *`*bspl*`, address6_t` *address*);

Parameters

bspl– Pointer to the BSPL table.

address – IPv6 address to be looked up.

Return Values

Prefix ID of the longest matching prefix

CHAPTER 9

malloc Library for Slow Path

This chapter describes the memory allocation (malloc) library API. Topics include:

malloc Library API Introduction

All applications need memory. Teja APIs such as teja_memory_pool_get_node(), which can be used in Sun Netra DPS, enables you to get a block of memory. Using teja APIs ensures high performance, but there is an overhead to the application writer finding the optimum memory pool for the required memory size.

Slow path requires various sizes of memory, but not high performance. The malloc/free implementation in this library can be used in slow path.

You also can use this library to use the LDC and IPC libraries, which require a malloc/free implementation.

Compiling Sun Netra DPS Application with `malloc` Library

The `malloc` library has two components:

- Declaring memory pools
- Including `malloc` definition

▼ Declare Memory Pools

In the software architecture of the application, you must declare the memory pools for `malloc` library.

1. **Add the `include` path `/opt/SUNWndps/src/libs/malloc/include` to `tajacc` flags and to `CFLAGS`.**

```
TEJACC_FLAGS+= /opt/SUNWndps/src/libs/malloc/include
CFLAGS+= /opt/SUNWndps/src/libs/malloc/include
```

2. **To carry out the following steps, add the following `makefile` target and call it at the beginning:**

```
all: init $(APPHWARCH_LIB) $(APPSWARCH_LIB) $(APPMAP_LIB) app
init:
# cp -f /opt/SUNWndps/src/libs/malloc/malloc_mem_pool.c
src/config/malloc_mem_pool.c touch netra_dps_malloc_init.c
```

3. **Copy `/opt/SUNWndps/src/libs/malloc/malloc_mem_pool.c` to the application directory.**

   For example, create `src/config/malloc_mem_pool.c` and compile it along with the software architecture file.

```
APPSWARCH_C = src/config/swarch.c src/config/malloc_mem_pool.c
```

4. **Create an empty file, `netra_dps_malloc_init.c`.**

Declare the memory pools needed to call `create_malloc_mem_pools()`.

▼ Include `malloc` Definition

1. **Add the** `netra_dps_malloc_init.c` **and** `/opt/SUNWndps/src/libs/malloc/netra_dps_malloc.c` **files to the list of C files that are passed to** `tejacc`**.**

```
C += netra_dps_malloc_init.c
C += /opt/SUNWndps/src/libs/malloc/netra_dps_malloc.c
```

2. **Call** `netra_dps_malloc_init()` **in the application initialization, to initialize the memory pools for malloc.**

`malloc` Configuration File

The user must create the `malloc.conf` configuration file to create the memory pools of the desired size and node count. In this file, the user needs to enter the memory pool node size, followed by the total number of nodes of that size:

```
# node_size     total_nodes
64              10000
128             10000
256             10000
```

For example, the first entry above in `malloc.conf` is set to create 10000 nodes of size 64 byte, and so on.

If the application does not have its own `malloc.conf` file, then it picks the configuration file from the `malloc` library that is in: `/opt/SUNWndps/src/libs/malloc/malloc.conf`

malloc Library APIs

create_malloc_mem_pools

Description

Declares the memory pools as specified in the configuration file (`malloc.conf`) and generates the `netra_dps_malloc_init.c`.

Syntax

```
int

create_malloc_mem_pools(teja_thread_t threads[], const char *mem_bank);
```

Parameters

threads – NULL terminated list of all the threads, from where the application is going to call `malloc/free`.

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

Return Values

0 – Success

-1 – Error

netra_dps_malloc_init

Description

Initializes the `malloc` memory pool data structures.

Syntax

```
void

netra_dps_malloc_init(void);
```

malloc

Description

Allocates and returns the memory of size equal to or greater than the requested size.

Syntax

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

Parameters

size – Required memory size.

Return Values

NULL on error, otherwise, the allocated memory.

free

Description

Frees the requested memory location.

Syntax

```
void
free(void *mem);
```

Parameters

mem – Memory location to be freed.

CHAPTER 10

Transparent Interprocess Communication API

This chapter describes the Transparent Interprocess Communication (TIPC) API. Topics include:

Transparent Interprocess Communication API Introduction

The Transparent Interprocess Communication (TIPC) protocol is for clustered computer environments. TIPC enables developers to create applications that can communicate quickly and reliably with other applications regardless of their location within the cluster.

TIPC Ethernet Bearer API for Sun Sun Netra DPS

The TIPC Ethernet Bearer API can be used to develop Sun Netra DPS TIPC application with TIPC Ethernet bearer support.

tipc_eth_get_mac

Description

Must be implemented by the application developer. The function is used inside the TIPC stack. The function must copy the MAC address of the given Ethernet or `vnet` port number in the MAC argument.

The sample implementation of this function is found in the TIPC reference application: `/opt/SUNWndps/src/apps/tipc/src/app/tipc_eth.c`

Syntax

```
int

tipc_eth_get_mac(tipc_dev_type dev_type, int port, uchar_t *mac);
```

Parameters

dev_type - `DEV_TYPE_ETHERNET` for ethernet port.
`DEV_TYPE_VNET` for `vnet` port.

port – Ethernet or `vnet` port number.

mac – Buffer to copy the MAC address of the Ethernet or `vnet` port.

Return Values

0 – Success.

-1 – Error.

tipc_eth_get_fastq

Description

Returns the TIPC stack receive and transmit fast queue handles for the given Ethernet or `vnet` port and DMA channel number.

Syntax

```
fastq_t

tipc_eth_get_fastq(tipc_dev_type dev_type, int port, int chan,
tipc_fastq type);
```

Parameters

dev_type - `DEV_TYPE_ETHERNET` for ethernet port.
`DEV_TYPE_VNET` for `vnet` port.

port – Ethernet/vnet port number.

chan – DMA channel number. Must be 0 for `vnet` port

type – `TIPC_FASTQ_RX` for receive fast queue.
`TIPC_FASTQ_TX` for transmit fast queue.

Return Values

Fast queue handle – Success.

NULL – On invalid Ethernet or `vnet` port number.

tipc_eth_pbuf_alloc

Description

Allocates a message block for outgoing packet data. This function must be implemented by application developer. The function should call the Ethernet `alloc` API (`eth_pbuf_alloc`).

The sample implementation of this function is found in the TIPC reference application: `/opt/SUNWndps/src/apps/tipc/src/app/tipc_util.c`

Syntax

```
pbuf_t *

tipc_eth_pbuf_alloc(void *hook, size_t size, uint16_t mpool_id);
```

Parameters

hook – User-provided hook. (See in "Note 1" on page 139.)

size – User-provided buffer size to be allocated. (See "Note 2" on page 139.)

mpool_id – DMA channel pool. (See "Note 3" on page 139.)

Return Values

Pointer to `mblk` with `b_rptr` and `b_wptr` pointing to the start of a valid data buffer – Success.

NULL – Error.

tipc_eth_pbuf_free

Description

Free the message block allocated by `eth_pbuf_alloc`. This function must be implemented by application developer. The function should call the Ethernet `free` API (`eth_pbuf_free`).

The sample implementation of this function is found in the TIPC reference application: `/opt/SUNWndps/src/apps/tipc/src/app/tipc_util.c`

Syntax

```
void

tipc_eth_pbuf_free(void *hook, pbuf_t *mp, void *arg, uint16_t mpool_id);
```

Parameters

hook – User-provided hook. (See in "Note 1" on page 139.)

mp – Pointer to message block to be freed.

arg – Not used (pass in NULL).

mpool_id – DMA channel pool. (See "Note 3" on page 139.)

Return Values

None.

TIPC Entry Point APIs for Sun Netra DPS

The following function prototypes can be found in `netra_dps/tipc.h` in `/opt/SUNWndps/lib/tipc/include/`.

tipc_init

Description

Must be called from the application initialization function.

Syntax

```
int
tipc_init(int tipc_strandid, int numport, int numchan, int startport);
```

Parameters

tipc_strandid – The thread number that runs the TIPC stack.

numport – Number of Ethernet ports the TIPC stack to support.

numchan – Number of DMA channels per port.

startport – First Ethernet port number from which TIPC stack to support.

Return Values

0 – Success.

-1 – Failure.

tipc_init_mempool

Description

Initializes the memory pool ID used by the Ethernet driver for given port and DMA channel.

Syntax

```
void
```

`tipc_eth_init_mempool(int` *port*, `int` *chan*, `uint16_t` *mpool_id*);

Parameters

port – Port number of the Ethernet interface.

chan – DMA channel number.

mpool_id – Memory pool ID used by Ethernet driver for the given DMA channel.

Return Values

None.

tipc_pbuf_alloc

Description

Must be implemented by the application developer. The function is used by the TIPC stack to allocate `pbuf_t` buffers. The buffer returned by the function should be 8-byte aligned.

The sample implementation of this function is found in the TIPC reference application: `/opt/SUNWndps/src/apps/tipc/src/app/tipc_util.c`

Syntax

```
pbuf_t *
tipc_pbuf_alloc(size_t size);
```

Parameters

size – Size of the buffer.

Return Values

Buffer address – Success.

NULL – Failure.

tipc_pbuf_free

Description

Must be implemented by the application developer. The function is used by the TIPC stack to free `pbuf_t` buffers.

The sample implementation of this function is found in the TIPC reference application: `/opt/SUNWndps/src/apps/tipc/src/app/tipc_util.c`

Syntax

```
void

tipc_pbuf_free(pbuf_t *mp);
```

Parameters

mp – Buffer obtained through `tipc_pbuf_alloc`.

Return Values

None.

tipc_process

Description

Must be polled regularly at least in an interval of 1 ms. This function provides the TIPC stack functionality. This function can be called from a dedicated strand or it can be put in the management strand. This function must not be called from the strand where a TIPC application is running, otherwise, the `fastq` called will cause a dead-lock.

Syntax

```
int

tipc_process(void *arg);
```

Parameters

arg – Set to NULL.

Return Values

0 – Return value.

tipc_vnet_config_register

Description

This function is used to register fast queues for sending and receiving `tn-tipc-config` tool packets between the Sun Netra DPS application and the Sun Netra DPS TIPC stack when Linux `tn-tipc-config` tool is used. This function is also used to register a packet buffer pool with the TIPC stack. This pool is used by the TIPC stack for allocating `tn-tipc-config` tool packets.

Syntax

```
int

tipc_vnet_config_register(void *txfq, void *rxfq, int poolid)
```

Parameters

txfq – Fast queue on which configuration packets transmitted by the TIPC stack are queued. The application dequeues packets from this fast queue and transmits them over a `vnet` interface.

rxfq – Fast queue on which application enqueues configuration packets that are received on a `vnet` interface. The TIPC stack dequeues from this fast queue for processing.

poolid – Relative index for the memory pool from which packets are allocated.

Return Values

On success, returns 0. On failure, returns a value less than 0.

tipc_cfgsrv_pbuf_alloc

Description

Must be implemented by the application developer. The function is used by the TIPC stack to allocate message blocks for communicating with Linux `tn-tipc-config` tool.

Syntax

```
pbuf_t *

tipc_cfgsrv_pbuf_alloc(void *hook, size_t bufsz, uint16_t pool)
```

Parameters

hook – Unused. TIPC stack always passes a NULL.

bufsz – Size of the buffer requested.

pool – The relative memory pool index that was provided by the application using `tipc_vnet_config_register` described above.

Return Values

NULL on failure. A non-NULL value on success.

tipc_cfgsrv_pbuf_free

Description

Must be implemented by the application developer. The function is used by the TIPC stack to free message blocks that were allocated for communicating with Linux `tn-tipc-config` tool.

Syntax

```
void

tipc_cfgsrv_pbuf_free(void *hook, pbuf_t *mblkp, void *arg,
uint16_t pool)
```

Parameters

hook – Unused. TIPC stack always passes a NULL.

mblkp – Message block that needs to be freed.

arg – Unused. TIPC stack always passes a NULL.

pool – The relative memory pool index that was provided by the application using `tipc_vnet_config_register` described above.

Return Values

None.

TIPC Socket APIs for Sun Netra DPS

All the Socket APIs adhere to the Solaris standard, but all blocking APIs have been modified to poll instead. The following functions have the same functionality as the Solaris socket API:

The following functions have the same functionality as the Solaris socket API:

- `shutdown()`
- `bind()`
- `getpeername()`
- `getsockname()`
- `getsockopt()`
- `setsockopt()`
- `close()`
- `connect()`
- `listen()`

Note – `poll()` is not supported in Sun Netra DPS.

The following Socket API functions differ from the Solaris standard in functionality.

`socket`

Description

Creates TIPC socket. The handle that is returned by a successful call to this function must be passed to access the socket using any of the other interface functions.

Syntax

```
int
socket(int family, int type, int protocol)
```

Parameters

family – The family must be `AF_TIPC` to create the TIPC socket.

type – The following socket types are supported:

```
SOCK_DGRAM
SOCK_STREAM
SOCK_RDM
SOCK_SEQPACKET
```

protocol – Currently not supported, should be set to 0.

Return Values

New socket handle – Success.

Error number – Error values similar to Solaris socket `accept()` API.

accept

Description

Nonblocking call that extracts the first connection request on the queue of pending connections, creates a new socket with the same properties of socket, and allocates a new socket handle for the socket.

Syntax

`int`

`tipc_accept(int` *socket_handle*, `struct sockaddr` **addrp*, `int` **addrlenp*)

Parameters

socket_handle – Handle returned by `socket()` API.

addrp – Result parameter that is filled in with the address of the connecting entity. This should be of type `struct sockaddr_tipc`.

addrlenp – Address length of the connecting entity.

Return Values

New socket handle – Success.

Error number – Error values similar to Solaris socket `accept()` API.

recv

Description

Nonblocking implementation of `recv()`. This function receives a maximum of length bytes of data from the given socket handle. The length of the message is returned. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket from which the message is received.

Syntax

```
int
recv(int socket_handle, char *buffer, int length, int flags)
```

Parameters

socket_handle – Handle returned by `socket`() API.

buffer – Buffer to which the data to be copied.

length – Number of bytes to be read.

flags – Must be 0.

Return Values

New socket handle – Success.

Error number – Error values similar to Solaris socket `accept()` API.

recvfrom

Description

Nonblocking implementation of `recvfrom()`. This function receives a maximum of *length* bytes of message from the given socket handle. The length of the message is returned. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket from which the message is received. The *from* parameter is filled with the address of source from which the message is received.

Syntax

```
int
```

`recvfrom(int` *socket_handle*, `char` **buffer*, `int` *length*, `int` *flag*, `struct sockaddr` **from*, int **fromlen*)

Parameters

socket_handle – Handle returned by socket() API.

buffer – Buffer to which the data to be copied.

length – Number of bytes to be read.

flag – Must be 0.

from – Source address of the message received. This should be of type `struct sockaddr_tipc`.

fromlen – Length of the source address.

Return Values

New socket handle – Success.

Error number – Error values similar to Solaris socket `accept()` API.

send

Description

Nonblocking implementation of `send()`. This function sends a maximum of *length* bytes of the message to the socket.

Syntax

```
int
```

`send(int` *socket_handle*, `const char` **dataptr*, `int` *size*, `int` *flags*)

Parameters

socket_handle – Handle returned by `socket`() API.

dataptr – Buffer to which the data to be copied

size – Number of bytes to be send.

flags – Must be 0.

Return Values

New socket handle – Success.

Error number – Error values similar to Solaris socket `accept()` API.

sendto

Description

Nonblocking implementation of `sendto()`. This function sends a maximum of *length* bytes of message to the socket.

Syntax

```
int
```

`sendto(int` *socket_handle*, `const char` **dataptr*, `int` *size*, `int` *flag*, `const struct sockaddr` **to*, `int` *tolen*)

Parameters

socket_handle – Handle returned by `socket`() API.

dataptr – Buffer to which the data is to be copied.

size – Number of bytes to be send.

flag – Must be 0.

to – Destination address to which data is to be sent. This parameter should be of type `struct sockaddr_tipc`.

tolen – Length of the *to* address.

Return Values

New socket handle – Success.

Error number – Error values similar to Solaris socket `accept()` API.

TIPC Tunables

TABLE 4-5 lists the TIPC Stack tunables.

TABLE 10-1 TIPC Stack Tunables

TIPC Tunable	Description
`tipc_config_thru_vnet`	Enables or disables the capability to configure Sun Netra DPS TIPC stack using Linux `tn-tipc-config tool` that uses `vnet`. When set to 1, it implies that Sun Netra DPS TIPC stack can be configured using `tn-tipc-config` tool that uses `vnet` for command or data exchange. When set to 0, it implies that Sun Netra DPS TIPC stack cannot be configured using `tn-tipc-config` tool that uses `vnet` for command or data exchange.

▼ To Configure the TIPC Stack With the Linux `tn-tipc-config` Tool

1. **Set the TIPC stack tunable `tipc_config_thru_vnet` to 1 before calling `tipc_init`.**
2. **Create two fast queues: one for dequeueing TIPC configuration packets from the stack and one to enqueue TIPC configuration packets into the stack.**
3. **Register the fast queues with the Sun Netra DPS TIPC stack using `tipc_vnet_config_register`.**
4. **Add the Ethernet multicast address, `33:33:00:00:00:01`, to the multicast table of the `vnet` in the Oracle VX Server for SPARC software that will be used for `tn-tipc-config`.**

   The `vnet` `ioctl` `ETH_IOC_ADD_MCAST_ADDR` must be used.
5. **Enqueue all Ethernet frames that have an Ethernet type `0x3c21` in to the TIPC stack using the fast queue created in Step 2.**
6. **Dequeue packets from the fast queue created in Step 2, and transmit the packet over the `vnet` used for `tn-tipc-config`.**

Index

M

N

P

S

T

U