



Netra™ Data Plane Software Suite 2.0 Reference Manual

Sun Microsystems, Inc.
www.sun.com

Part No. 820-3363-10
April 2008, Revision A

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

Copyright 2008 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. All rights reserved.

Sun Microsystems, Inc. has intellectual property rights relating to technology that is described in this document. In particular, and without limitation, these intellectual property rights may include one or more of the U.S. patents listed at <http://www.sun.com/patents> and one or more additional patents or pending patent applications in the U.S. and in other countries.

This document and the product to which it pertains are distributed under licenses restricting their use, copying, distribution, and decompilation. No part of the product or of this document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California. UNIX is a registered trademark in the U.S. and in other countries, exclusively licensed through X/Open Company, Ltd.

Sun, Sun Microsystems, the Sun logo, Java, UltraSPARC, Netra, Sun Fire, OpenBoot, docs.sun.com, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and in other countries.

All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and in other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK and Sun™ Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

U.S. Government Rights—Commercial use. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright 2008 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, Californie 95054, États-Unis. Tous droits réservés.

Sun Microsystems, Inc. possède les droits de propriété intellectuelle relatifs à la technologie décrite dans ce document. En particulier, et sans limitation, ces droits de propriété intellectuelle peuvent inclure un ou plusieurs des brevets américains listés sur le site <http://www.sun.com/patents>, un ou les plusieurs brevets supplémentaires ainsi que les demandes de brevet en attente aux États-Unis et dans d'autres pays.

Ce document et le produit auquel il se rapporte sont protégés par un copyright et distribués sous licences, celles-ci en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a.

Tout logiciel tiers, sa technologie relative aux polices de caractères, comprise, est protégé par un copyright et licencié par des fournisseurs de Sun.

Des parties de ce produit peuvent dériver des systèmes Berkeley BSD licenciés par l'Université de Californie. UNIX est une marque déposée aux États-Unis et dans d'autres pays, licenciée exclusivement par X/Open Company, Ltd.

Sun, Sun Microsystems, le logo Sun, Java, UltraSPARC, Netra, Sun Fire, OpenBoot, docs.sun.com, et Solaris sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux États-Unis et dans d'autres pays.

Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux États-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

L'interface utilisateur graphique OPEN LOOK et Sun™ a été développée par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox dans la recherche et le développement du concept des interfaces utilisateur visuelles ou graphiques pour l'industrie informatique. Sun détient une licence non exclusive de Xerox sur l'interface utilisateur graphique Xerox, cette licence couvrant également les licenciés de Sun implémentant les interfaces utilisateur graphiques OPEN LOOK et se conforment en outre aux licences écrites de Sun.

LA DOCUMENTATION EST FOURNIE "EN L'ÉTAT" ET TOUTES AUTRES CONDITIONS, DÉCLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES DANS LA LIMITE DE LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE À LA QUALITÉ MARCHANDE, À L'APTITUDE À UNE UTILISATION PARTICULIÈRE OU À L'ABSENCE DE CONTREFAÇON.



Contents

Preface ix

1. Configuration API 1

Hardware Architecture API 1

Hardware Architecture API Data Types 2

Hardware Architecture API Functions 3

Software Architecture API 35

Software Architecture API Data Types 35

Software Architecture API Functions 36

Map API 50

Map API Data Types 51

Map API Functions 51

Error – Handling API 54

Error – Handling API Data Types 54

Error – Handling API Functions 54

Error Handler Function Prototype 55

CMT – Specific Hardware Architecture Constants 56

CMT – Specific Hardware Architecture Types 57

CMT – Specific Hardware Architecture Properties 58

CMT – Specific Software Architecture Constants 59

CMT – Specific Software Architecture Types	59
CMT – Specific Software Architecture Properties	60

2. User API 61

Late-Binding API	61
Late-Binding API Data Types	62
Late-Binding API Macros	62
Late-Binding API Mutex Functions	62
Late-Binding API Queue Functions	65
Late-Binding API Memory Pool Functions	68
Late-Binding API Channel Functions	71
Late-Binding API Interruptible Wait	73
Netra DPS Runtime API	77
Netra DPS Runtime API Data Types	77
Netra DPS Runtime API Memory Management Functions	78
Netra DPS Runtime API Thread Functions	80
Netra DPS Runtime API Miscellaneous Functions	83
Netra DPS Runtime API Time Functions	84
Miscellaneous Functions	86
Finite State Automata API	87
Finite State Automata API Defines	88
Finite State Automata API Macros	88
FSM Example	91
Hardware Specific Miscellaneous Functions	93
C Library Support on Bare Hardware	94
 3. Profiler API	 95
Profiler API Configuration	95
Profiler API	96

Profiler API Data Types	96
Profiler API Functions	97
CMT– Specific Profiler Constants	100
CMT– Specific Profiler Groups	100
4. Driver API	103
Netra DPS Crypto and Hashing API	103
Netra DPS Crypto and Hash API Functions	104
Netra DPS Crypto and Hash API Function Descriptions	105
Ethernet API	116
Network Applications	116
Ethernet Device and Device Driver	116
Ethernet API Functions	117
Description of Ethernet API Functions	117
Summary	122
Notes	123
5. Fast Queue API	127
Fast Queue API Introduction	127
Fast Queue API Function Descriptions	128
6. Interprocess Communication API	131
Interprocess Communication API Introduction	131
Common Programming Interfaces	132
IPC Framework Programming Interfaces	135
IPC Programming Interfaces for Solaris Domains	138
User Space	138
Kernel	138

7. Fastpath Manager API	139
Fastpath Manager API Introduction	139
Fastpath Manager API Function Descriptions	140
Index	143

Tables

TABLE 1-1	Hardware Architecture API Data Types	2
TABLE 1-2	Software Architecture API Data Types	35
TABLE 1-3	Map API Data Type	51
TABLE 1-4	Error-Handling Data Types	54
TABLE 1-5	CMT-Specific Hardware Architecture Types	57
TABLE 1-6	CMT-Specific Hardware Architecture Properties	58
TABLE 1-7	CMT-Specific Software Architecture Types	59
TABLE 1-8	CMT-Specific Software Architecture Properties	60
TABLE 2-1	Late-Binding API Data Types	62
TABLE 2-2	Late-Binding API Macros	62
TABLE 2-3	Netra DPS Runtime API Data Types	77
TABLE 2-4	Netra DPS Runtime API Macros	78
TABLE 2-5	Netra DPS Runtime API Thread Types	80
TABLE 2-6	Finite State Automata API Defines	88
TABLE 3-1	Process Properties	95
TABLE 3-2	Profiler API Data Types	96
TABLE 3-3	CMT_Specific Profiler Groups	100
TABLE 4-1	Ethernet API and User Applications	116
TABLE 4-2	Ethernet Devices Supported on Netra DPS Platforms	116
TABLE 4-3	Ethernet API Function Summary	122

Preface

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

How This Document Is Organized

[Chapter 1](#) describes the components and functions of the Configuration API.

[Chapter 2](#) describes the components and functions of the User API.

[Chapter 3](#) describes the components and functions of the Profiler API.

[Chapter 4](#) describes the Netra DPS Crypto and Hashing API and Ethernet API.

[Chapter 5](#) describes the Fast Queue API functions.

[Chapter 6](#) describes the Interprocess Communication (IPC) API.

[Chapter 7](#) describes the Fastpath Manager API.

Using UNIX Commands

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

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

<http://docs.sun.com>

Shell Prompts

Shell	Prompt
C shell	<i>machine-name%</i>
C shell superuser	<i>machine-name#</i>
Bourne shell and Korn shell	\$
Bourne shell and Korn shell superuser	#

Typographic Conventions

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

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

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	<i>Netra Data Plane Software Suite 2.0 User's Guide</i>	820-3362-10	PDF	online
Reference	<i>Netra Data Plane Software Suite 2.0 Reference Manual</i>	820-3363-10	PDF	online
Last-minute information	<i>Netra Data Plane Software Suite 2.0 Release Notes</i>	820-3364-10	PDF	online
Documentation Location	<i>Netra Data Plane Software Suite 2.0 Getting Started Guide</i>	820-3365-10	PDF	online

Documentation, Support, and Training

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

Third-Party Web Sites

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

Sun Welcomes Your Comments

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

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

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

Netra Data Plane Software Suite 2.0 Reference Manual, part number 820-3363-10

Configuration API

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

- [“Hardware Architecture API” on page 1](#)
- [“Software Architecture API” on page 35](#)
- [“Map API” on page 50](#)
- [“Error – Handling API” on page 54](#)
- [“CMT – Specific Hardware Architecture Constants” on page 56](#)
- [“CMT – Specific Software Architecture Constants” on page 59](#)

Hardware Architecture API

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

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

Hardware Architecture API Data Types

The hardware architecture definitions use the following data types.

TABLE 1-1 Hardware Architecture API Data Types

Data Type	Description
<code>teja_architecture_t</code>	Hardware architecture. An architecture might contain processors, memories, buses, hardware objects, and other architectures.
<code>teja_processor_t</code>	Processor. A processor is a target for an OS (<code>teja_os_t</code>).
<code>teja_memory_t</code>	Memory. A memory is a target for mapping variables declared in user-application source code.
<code>teja_bus_t</code>	Bus connecting objects with each other.
<code>teja_bus_visibility_t</code>	Buses have two types of visibility: <ul style="list-style-type: none">• <code>TEJA_INTERNAL_BUS</code> - bus not visible outside its containing architecture• <code>TEJA_EXPORTED_BUS</code> - bus made visible outside its containing architecture Example: <pre>typedef enum {TEJA_INTERNAL_BUS, TEJA_EXPORTED_BUS} teja_bus_visibility_t;</pre>
<code>teja_hardware_object_t</code>	Generic hardware module that is not a processor, a memory, or a bus.
<code>teja_port_t</code>	Hardware port.
<code>teja_address_space_t</code>	Address space. An address space is used as context for allocating address ranges.
<code>teja_address_range_t</code>	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.

Function

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

Parameters

container – Container for the new architecture.

name – Name of the new architecture.

type – Type of the architecture.

Return Values

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

teja_architecture_set_property

Description

Sets the value of the property for the architecture object.

Function

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

Parameters

arch – Architecture object.

property-name – Name of the property.

value – Value of the property.

Return Values

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

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

Function

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

Parameters

arch – Architecture object.

property-name – Name of the property.

value – Returned value of the property.

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

Return Values

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

`teja_architecture_set_read_only`

Description

Prevents modification of given architecture by subsequent processing.

Function

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

Parameters

arch – Architecture object.

Return Values

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

`teja_processor_create`

Description

Creates a processor object.

Function

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

Parameters

container – Container of the new processor.

name – Name of the new processor.

type – Type of the new processor.

Return Values

`teja_processor_t` – Returns newly created processor object.

`teja_processor_set_property`

Description

Sets the value of the property for the processor object.

Function

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

Parameters

processor – Processor object.

property-name – Name of the property.

value – Value of the property.

Return Values

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

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

Function

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

Parameters

processor – Processor object.

property-name – Name of the property.

value – Value of the property.

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

Return Values

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

`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.

Function

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

Parameters

processor – Processor instance to which the symbol is added.

symbol – Name of the symbol.

value – Value of the symbol.

Return Values

int – Returns TEJA_SUCCESS or error code for failure.

`teja_memory_create`

Description

Creates a memory object

Function

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

Parameters

container – Container of the new memory.

name – Name of the new memory.

type – Type of the new memory.

Return Values

teja_memory_t – Returns the newly created memory object.

`teja_memory_set_property`

Description

Sets the value of the property for the memory object.

Function

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

Parameters

memory – Memory object.

property-name – Name of the property.

value – Value of the property.

Return Values

int – Returns TEJA_SUCCESS or error code for failure.

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

Function

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

Parameters

memory – Memory object.

property-name – Name of the property.

value – Returned value of the property.

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

Return Values

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

teja_bus_create

Description

Creates a bus object.

Function

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

Parameters

container – Container of the new bus.

name – Name of the new bus.

type – Type of the new bus.

Return Values

teja_bus_t – Returns newly created bus object.

teja_bus_set_property

Description

Sets the value of the property for the bus object.

Function

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

Parameters

bus – Bus object.

property-name – Name of the property.

value – Value of the property.

Return Values

int – Returns TEJA_SUCCESS or error code for failure.

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

Function

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

Parameters

bus – Bus object.

property-name – Name of the property.

value – Returned value of the property.

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

Return Values

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

teja_hardware_object_create

Description

Creates a hardware object.

Function

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

Parameters

container – Container of the new hardware object.

name – Name of the new hardware object.

type – Type of the new hardware object.

Return Values

`teja_hardware_object_t` – Returns newly created hardware object.

`teja_hardware_object_set_property`

Description

Sets the value of the property for the hardware object.

Function

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

Parameters

hardware-object – Hardware object.

property-name – Name of the property.

value – Value of the property.

Return Values

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

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

Function

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

Parameters

hardware-object – Hardware object.

property-name – Name of the property.

value – Returned value of the property.

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

Return Values

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

teja_architecture_connect

Description

Connects an architecture to a bus.

Function

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

Parameters

architecture – Architecture object that needs to be connected.

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

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

Return Values

int – Returns TEJA_SUCCESS or error code for failure.

teja_processor_connect

Description

Connects a processor to a bus.

Function

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

Parameters

processor – Processor object that needs to be connected.

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

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

Return Values

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

`teja_memory_connect`

Description

Connects a memory to a bus.

Function

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

Parameters

memory – Memory object that needs to be connected.

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

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

Return Values

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

`teja_hardware_object_connect`

Description

Connects a hardware object to a bus.

Function

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

Parameters

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

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

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

Return Values

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

`teja_lookup_architecture`

Description

Looks up an architecture in the container.

Function

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

Parameters

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

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

Return Values

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

`teja_lookup_processor`

Description

Looks up a processor in the container.

Function

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

Parameters

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

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

Return Values

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

teja_lookup_memory

Description

Looks up a memory in the container.

Function

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

Parameters

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

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

Return Values

teja_memory_t – The found memory or NULL if not found.

teja_lookup_bus

Description

Looks up a bus in the container.

Function

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

Parameters

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

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

Return Values

teja_bus_t – The found bus or NULL if not found.

teja_lookup_hardware_object

Description

Looks up a hardware object in the container.

Function

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

Parameters

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

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

Return Values

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

`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](#).

Function

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

Parameters

arch – Container architecture in which the port is created.

port-name – Name of the port.

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

Return Values

teja_port_t – Returns the newly created port.

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.

Function

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

Parameters

arch – Architecture containing the port.

port-name – Name of the architecture port.

value – Value to be assigned externally to the port.

Return Values

int – Returns TEJA_SUCCESS or error code for failure.

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.

Function

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

Parameters

arch – Architecture containing the port.

port-name – Name of the architecture port.

value – Value to be assigned internally to the port.

Return Values

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

`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.

Function

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

Parameters

proc – Processor containing the port.

port-name – Name of the port.

value – Value to be assigned to the port.

Return Values

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

`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.

Function

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

Parameters

memory – Memory containing the port.

port-name – Name of the port.

value – Value to be assigned to the port.

Return Values

int – Returns TEJA_SUCCESS or error code for failure.

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.

Function

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

Parameters

hardware-object – Hardware object containing the port.

port-name – Name of the port.

value – Value to be assigned to the port.

Return Values

int – Returns TEJA_SUCCESS or error code for failure.

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.

Function

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

Parameters

bus – Bus containing the port.

port-name – Name of the port.

value – Value to be assigned to the port.

Return Values

int – Returns TEJA_SUCCESS or error code for failure.

teja_port_add_property

Description

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

Function

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

Parameters

port – Port to which the property is added.

property-name – Name of the new property.

value – Value of the new property.

description – Description associated to the property.

Return Values

int – Returns TEJA_SUCCESS or error code for failure.

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.

Function

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

Parameters

architecture – An architecture.

Return Values

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

teja_processor_get_parent

Description

Returns the architecture containing the specified processor.

Function

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

Parameters

processor – A processor.

Return Values

teja_architecture_t – The architecture containing the processor.

teja_bus_get_parent

Description

Returns the architecture containing the specified processor.

Function

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

Parameters

bus – A bus.

Return Values

teja_architecture_t – The architecture containing the bus.

teja_memory_get_parent

Description

Returns the architecture containing the specified memory.

Function

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

Parameters

memory – A memory.

Return Values

teja_architecture_t – The architecture containing the memory.

teja_hardware_object_get_parent

Description

Returns the architecture containing the specified hardware object.

Function

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

Parameters

hardware-object – A hardware object.

Return Values

`teja_architecture_t` – The architecture containing the hardware object.

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

Function

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

Parameters

arch – An architecture.

Return Values

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

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

Function

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

Parameters

arch – An architecture.

Return Values

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

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

Function

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

Parameters

arch – An architecture.

Return Values

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

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

Function

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

Parameters

arch – An architecture.

Return Values

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

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

Function

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

Parameters

arch – An architecture.

Return Values

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

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.

Function

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

Parameters

processor – A processor.

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

Return Values

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

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.

Function

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

Parameters

memory – A memory.

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

Return Values

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

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.

Function

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

Parameters

hardware-object – An hardware object.

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

Return Values

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

`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.

Function

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

Parameters

architecture – An architecture.

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

Return Values

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

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

Function

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

Parameters

bus – A bus.

Return Values

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

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

Function

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

Parameters

bus – A bus.

Return Values

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

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

Function

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

Parameters

bus – A bus.

Return Values

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

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

Function

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

Parameters

bus – A bus.

Return Values

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

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

Function

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

Parameters

processor – A processor.

Return Values

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

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

Function

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

Parameters

memory – A memory.

Return Values

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

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

Function

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

Parameters

hardware-object – A hardware object.

Return Values

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

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').

Function

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

Parameters

arch – An architecture.

name – Name of an address space to be created.

base – Base address for the space.

high – Highest address in the space.

Return Values

teja_address_space_t – The newly created address space.

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.

Function

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

Parameters

s1 – An address space.

s2 – An address space.

Return Values

`int` – TEJA_SUCCESS or error code for failure.

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.

Function

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

Parameters

space – An address space.

sym – A symbol to be associated to the range.

base – Base address for the range.

size – Size of the range.

Return Values

`teja_address_range_t` – The newly created address range.

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.

Function

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

Parameters

space – An address space.

sym – A symbol to be associated to the range.

alignment – An alignment constraint.

size – Size of the range.

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

Return Values

`teja_address_range_t` – The newly created address range.

`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.

Function

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

Parameters

space – An address space.

sym – A symbol to be associated to the range.

size – Size of the range.

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

Return Values

`teja_address_range_t` – The newly created address range.

`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.

Function

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

Parameters

range – An address range.

buf – An array of characters.

buf-size – Size of the array of characters.

Return Values

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

`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.

Function

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

Parameters

range – An address range.

buf – An array of characters.

buf-size – Size of the array of characters.

Return Values

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

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 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

<code>teja_os_t</code>	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.
<code>teja_process_t</code>	Process. One or more processes run on an OS.
<code>teja_thread_t</code>	Thread. One or more threads run in a process.
<code>teja_channel_t</code>	Channel. Channels provide the communication facility to send structured data between two or more threads.
<code>teja_memory_pool_t</code>	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.
<code>teja_queue_t</code>	Queue. A queue provides a facility to pass structured data between two or more threads.
<code>teja_mutex_t</code>	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.

Function

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

Parameters

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

name – Name of the OS instance.

type – Type of the OS.

Return Values

teja_os_t – Returns an object that represents the OS instance.

teja_os_set_property

Description

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

Function

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

Parameters

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

property-name – Name of the property.

value – Value of the property to be set.

Return Values

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

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

Function

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

Parameters

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

property-name – Name of the property.

value – Value that is read and returned.

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

Return Values

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

`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.

Function

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

Parameters

container – OS instance where the process is created.

name – Name of the instance.

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

Return Values

teja_process_t – Returns created process instance.

teja_process_set_property

Description

Sets a process property.

Function

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

Parameters

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

property-name – Name of the property.

value – Value of the property.

Return Values

int – Returns TEJA_SUCCESS or error code for failure.

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

Function

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

Parameters

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

property-name – Name of the property.

value – Returned value of the property.

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

Return Values

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

`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.

Function

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

Parameters

container – Process instance where the thread is created.

name – Name of the thread.

Return Values

`teja_thread_t` – Returns thread instance.

`teja_thread_set_property`

Description

Sets a thread property.

Function

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

Parameters

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

property-name – Name of the property.

value – Value of the property.

Return Values

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

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

Function

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

Parameters

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

property-name – Name of the property.

value – Returned value of the property.

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

Return Values

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

teja_lookup_os

Description

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

Function

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

Parameters

os-name – Name of the OS.

Return Values

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

`teja_lookup_process`

Description

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

Function

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

Parameters

process-name – Name of the process.

Return Values

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

`teja_lookup_thread`

Description

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

Function

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

Parameters

thread-name – Name of the thread.

Return Values

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

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.

Function

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

Parameters

name – Name of the channel.

type – Type of the channel.

producers – NULL-terminated list of producer thread instances.

consumers – NULL-terminated list of consumer thread instances.

Return Values

teja_channel_t – Returns the created channel instance.

teja_channel_set_property

Description

Sets a new value for a channel property.

Function

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

Parameters

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

property-name – Name of the property.

value – Value of the property.

Return Values

int – Returns TEJA_SUCCESS or error code for failure.

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

Function

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

Parameters

channel – Channel instance for which the property is read.

property-name – Name of the property.

value – Returned value of the property.

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

Return Values

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

teja_memory_pool_declare

Description

Creates a new memory pool instance in the software architecture.

Function

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

Parameters

name – Name of the memory pool.

type – Type of the memory pool.

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

node-size – Size in bytes for each node.

getters – NULL-terminated list of getter threads.

setters – NULL-terminated list of setter threads.

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

Return Values

`teja_memory_pool_t` – Returns the memory pool instance.

`teja_memory_pool_set_property`

Description

Sets a new value for a memory pool property.

Function

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

Parameters

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

property-name – Name of the property.

value – Value of the property.

Return Values

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

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

Function

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

Parameters

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

property_name – Name of the property.

value – Value of the property.

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

Return Values

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

teja_queue_declare

Description

Creates a new queue instance in the software architecture.

Function

```
teja_queue_t teja_queue_declare(const char *name,  
    const char *type, teja_thread_t *enqueueers, teja_thread_t *dequeueers);
```

Parameters

name – Name of the queue.

type – Type of the queue.

enqueueers – NULL-terminated list of enqueueers threads.

dequeueers – NULL-terminated list of dequeueers threads.

Return Values

teja_queue_t – Returns queue instance.

teja_queue_set_property

Description

Sets a new value for a queue property.

Function

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

Parameters

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

property-name – Name of the property.

value – Value of the property.

Return Values

int – Returns TEJA_SUCCESS or error code for failure.

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

Function

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

Parameters

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

property-name – Name of the property.

value – Returned value of the property.

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

Return Values

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

teja_mutex_declare

Description

Creates a new mutex instance in the software architecture.

Function

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

Parameters

name – Name of the mutex.

type – Type of the mutex.

users – NULL-terminated list of user-threads.

Return Values

teja_mutex_t – Returns a new mutex instance.

teja_mutex_set_property

Description

Sets a new value for a mutex property.

Function

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

Parameters

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

property-name – Name of the property.

value – Value of the property.

Return Values

int – Returns TEJA_SUCCESS or error code for failure.

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

Function

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

Parameters

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

property-name – Name of the property.

value – Returned value of the property.

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

Return Values

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

teja_lookup_channel

Description

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

Function

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

Parameters

channel-name – Name of the channel.

Return Values

teja_channel_t – Returns the found channel instance or NULL.

teja_lookup_memory_pool

Description

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

Function

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

Parameters

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

Return Values

teja_memory_pool_t – Returns the found memory pool instance or NULL.

teja_lookup_queue

Description

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

Function

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

Parameters

queue-name – Name of the queue.

Return Values

teja_queue_t – Returns the found queue instance or NULL.

teja_lookup_mutex

Description

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

Function

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

Parameters

mutex-name – Name of the mutex.

Return Values

teja_mutex_t – Returns the found mutex instance or NULL.

`teja_process_add_symbol`

Description

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

Function

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

Parameters

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

symbol – String that represents the symbol.

value – String that represents the value for the symbol.

Return Values

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

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

<code>teja_mapping_t</code>	Every mapping returns a handle of type <code>teja_mapping_t</code> .
-----------------------------	--

Map API Functions

`teja_map_function_to_thread`

Description

Maps a function to run on a thread.

Function

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

Parameters

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

thread-name – Name of the thread.

Return Values

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

`teja_map_variable_to_memory`

Description

Maps a variable to memory.

Function

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

Parameters

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

memory-name – Name of the memory bank.

Return Values

teja_mapping_t – Returns a handle that represents this mapping.

`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. You map any one of these variables to a memory bank using `teja_map_variable_to_memory`. The remaining variables are mapped to that variable using this function.

Function

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

Parameters

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

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

Return Values

teja_mapping_t – Returns a handle that represents this mapping.

`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.

Function

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


Parameters

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

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

Return Values

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

`teja_map_initialization_function_to_process`

Description

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

Function

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

Parameters

function – Name of the function.

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

Return Values

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

`teja_mapping_set_property`

Description

Sets a new value for a mapping.

Function

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

Parameters

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

property-name – Name of the property.

value – Value of the property.

Return Values

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

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

<code>teja_error_handler_t</code>	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*.

Function

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

Parameters

error-code – Error code.

error-message – Error message.

Return Values

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

teja_register_error_handler

Description

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

Function

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

Parameters

handler – Error handler function.

Return Values

`teja_error_handler_t` – The previously registered error handler.

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

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

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

CMT – Specific Hardware Architecture Constants

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

CMT – Specific Hardware Architecture Types

TABLE 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 <code>true</code> , makes location 0 non writable. The default value is <code>true</code> .
	TEJA_PROPERTY_MEMORY_IS_OS_BASED	When set to <code>true</code> , marks the memory OS-based. The default value is <code>false</code> .
	TEJA_PROPERTY_MEMORY_NO_ADDRESS_CONVERSION	When set to <code>true</code> , hal conversion is necessary. The default value is <code>true</code> .

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	This is the only type of OS that is supported for CMT CSP.
Channel	TEJA_GENERIC_CHANNEL_SHARED_MEMORY_OS_BASED	Channel type that uses OS-based shared memory implementation.
	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 you to assign a thread to a specific processor (hardware thread). Specify the processor using a fully qualified name from the hardware architecture. The default value for this property is <code>NULL</code> 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.

User API

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

- [“Late-Binding API” on page 61](#)
- [“Netra DPS Runtime API” on page 77](#)
- [“Finite State Automata API” on page 87](#)

Additional information is provided in:

- [“C Library Support on Bare Hardware” on page 94](#)

Late-Binding API

The Late-Binding API provides primitives for the synchronization of distributed threads, communication, and memory allocation. This API is treated specially by the `tejacc` compiler, and is generated dynamically based on contextual information. See the *Netra Data Plane Software Suite 2.0 User's Guide* for an overview of this API.

Late-Binding API Data Types

TABLE 2-1 Late-Binding API Data Types

Data Type	Description
<code>teja_channel_t</code>	Channel data type
<code>teja_memory_pool_t</code>	Memory pool data type
<code>teja_mutex_t</code>	Mutex data type
<code>teja_queue_t</code>	Queue data type
<code>teja_thread_t</code>	Thread data type

Late-Binding API Macros

TABLE 2-2 Late-Binding API Macros

Macros	Description
<code>TEJA_INFINITE_WAIT</code>	Used to indicate an infinite timeout in <code>teja_wait()</code> .
<code>TEJA_IS_RAW_OS</code>	Defined only on bare hardware systems. Such systems support a subset of the Netra DPS API.
<code>TEJA_NO_EVENT</code>	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.

Function

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

Parameters

mutex – Mutex to lock.

Return Values

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

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.

Function

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

Parameters

mutex – Mutex to lock.

Return Values

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

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.

Function

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

Parameters

mutex – Mutex to unlock.

Return Values

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

Example

See the examples in [“teja_mutex_lock” on page 62](#) and [“teja_mutex_trylock” on page 63](#).

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.

Function

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

Parameters

queue – Queue to enqueue to.

node – Pointer to node to enqueue.

Return Values

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

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.

Function

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

Parameters

queue – Queue to dequeue from.

Return Values

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

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.

Function

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

Parameters

queue – Queue to test.

Return Values

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

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.

Function

```
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.

Function

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

Parameters

memory-pool – Memory pool to allocate from.

Return Values

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

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.

Function

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

Parameters

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

node – Pointer to node to free.

Return Values

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

Example

See the example in [“teja_memory_pool_get_node” on page 68](#).

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.

Function

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

Parameters

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

index – Index of the node.

Return Values

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

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.

Function

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

Parameters

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

Return Values

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

`int` – Index of the given node.

Example

See the example in [“teja_memory_pool_get_node_from_index”](#) on page 69.

Late-Binding API Channel Functions

`teja_channel_is_connection_open`

Description

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

Function

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

Parameters

channel – Channel to test.

Return Values

int – 1 if the connection is open, 0 if the connection is closed.

Example

See the example in [“teja_channel_send” on page 72](#).

`teja_channel_make_connection`

Description

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

Function

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

Parameters

channel – Channel to operate on.

Return Values

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

Example

See the example in [“teja_channel_send” on page 72](#).

teja_channel_break_connection

Description

Breaks an existing connection on the given channel.

Function

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

Parameters

channel – Channel to operate on.

Return Values

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

Example

See the example in [“teja_channel_send” on page 72](#).

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.

Function

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

Parameters

channel – Channel to send data on.

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

message – Pointer to the data to send.

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

Return Values

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

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 74](#), 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 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.

Function

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

Parameters

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

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

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

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

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

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

buffer-size – Size of the buffer.

... – List of channels to read from. The list must be `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 72](#), which shows how to send data.

Netra DPS Runtime API

The 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, Netra DPS Runtime APIs are not treated specially by the compiler and are implemented in precompiled libraries. See the *Netra Data Plane Software Suite 2.0 User's Guide* for an overview of this API.

Netra DPS Runtime API Data Types

TABLE 2-3 Netra DPS Runtime API Data Types

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

TABLE 2-3 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 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 you that running on an OS or in bare hw mode. In NDPS it is always true.

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.

Function

```
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.

Function

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

Parameters

size – Size in bytes of memory to allocate.

Return Values

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

`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.

Function

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

Parameters

ptr – Pointer to memory to reallocate.

size – Size in bytes of memory to allocate.

Return Values

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

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 Netra DPS Runtime API Thread Types

Data Type	Description
<code>teja_thread_t</code>	This type is associated only to threads that are defined in the software architecture, and not to dynamic threads, created with <code>teja_thread_handle_start()</code> . This data type is an identifier type.
<code>teja_thread_handle_t</code>	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.

Function

```
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.

Function

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

Parameters

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

Return Values

char * – Name of the given thread.

teja_get_id_for_thread_name

Description

Returns the ID of the given thread.

Function

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

Parameters

name – Name of the thread to operate on.

Return Values

teja_thread_t – ID of the given thread.

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.

Function

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

Parameters

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

function – Main function of the thread.

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

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

priority – Priority of the thread. This functionality is not available on all systems. 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.

Function

```
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.

Function

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

Parameters

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

Return Values

teja_thread_handle_t * – Handle pointer for the given thread ID.

Netra DPS Runtime API Miscellaneous Functions

teja_thread_shutdown

Description

Shuts down the current Netra DPS thread.

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

Function

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

Return Values

void

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.

Function

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

Parameters

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

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

Return Values

int – 0 on success, -1 on error.

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.

Function

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

Parameters

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

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

Return Values

int – 0 on success, -1 on error.

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()`.

Function

```
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.

Function

```
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.

Function

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

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

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

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

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

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

Finite State Automata API Defines

TABLE 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.

Function

```
#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.

Function

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

Parameters

name – Name of the state machine.

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

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

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

```
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 semicolon (;) is required at the end of this call.

Function

```
#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.

Function

```
#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`.

Function

```
#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. You can add regular C code immediately after this macro up to the `teja_fsm_state_end` macro. No semi-colon (;) is required at the end of this call.

Function

```
#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`. You can add regular C code immediately before this macro. No semi-colon (;) is required at the end of this call.

Function

```
#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.

Function

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

Parameters

name – Name of the state to jump to.

FSM Example

[CODE 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

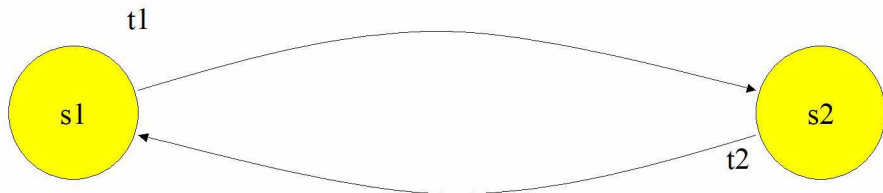


FIGURE 2-1 Finite State Machine Example

CODE 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"
```

CODE 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)
```



```
{
    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.

Function

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

Return Values

int – Returns the value of the clock tick register.

C Library Support on Bare Hardware

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

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

Profiler API

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

- [“Profiler API Configuration” on page 95](#)
- [“Profiler API” on page 96](#)
- [“CMT– Specific Profiler Constants” on page 100](#)



Profiler API Configuration

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

TABLE 3-1 Process Properties

Property	Description
<code>profiler_log_table_size</code>	Sets the total number of profile records in the log. The default value is 1024.
<code>profiler_user_data_size</code>	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

Profiler API Data Types

TABLE 3-2 Profiler API Data Types

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

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

Function

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

Parameters

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

event – Events of the group as a bit mask.

Return Values

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

teja_profiler_stop

Description

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

Function

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

Parameters

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

Return Values

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

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.

Function

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

Parameters

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

... – 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.

Function

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

Parameters

group – ID of the group which you want 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.

Function

```
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.

Function

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

Parameters

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

Return Values

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

CMT– Specific Profiler Constants

CMT– Specific Profiler Groups

TABLE 3-3 CMT_Specific Profiler Groups

Event or Group	Event or Description	Description
TEJA_PROFILER_CMT_CPU (0x1)	Captures events related to CPU and cache. The events measured in this group are per CPU strand. The following events are available for this group. The completed instructions count is always an available event for this group. There is additionally one more event that can be measured along with instructions count	
TEJA_PROFILER_CMT_CPU_SB_FULL (0x1)		Measures number of store buffer full cycles.
TEJA_PROFILER_CMT_CPU_FP_INSTR_CNT (0x2)		Measures number of floating point instructions.
TEJA_PROFILER_CMT_CPU_IC_MISS (0x4)		Measures number of instruction cache misses.
TEJA_PROFILER_CMT_CPU_DC_MISS (0x8)		Measures number of data cache misses.
TEJA_PROFILER_CMT_CPU_ITLB_MISS (0x10)		Measures number of instruction TLB miss traps taken.
TEJA_PROFILER_CMT_CPU_DTLB_MISS (0x20)		Measures number of data TLB miss traps taken.
TEJA_PROFILER_CMT_CPU_L2_IMISS (0x40)		Measures number of secondary cache (L2) misses due to instruction cache requests.
TEJA_PROFILER_CMT_CPU_L2_DMISS_LD (0x80)		Measures number of secondary cache (L2) misses due to data cache load requests.

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

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

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

Event or Group	Event or Description	Description
	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)

Driver API

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

- [“Netra DPS Crypto and Hashing API” on page 103](#)
- [“Ethernet API” on page 116](#)

Netra DPS Crypto and Hashing API

Netra DPS Crypto and Hashing API is an interface that allows Netra DPS developers to access the crypton and hash hardware functions supported by UltraSPARC T2 based platforms.

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

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

The 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 need to include the following header files under `src/libs/ndps_crypto_api/` in your application:

- `crypt_const.h`
- `ndpscrypto.h`
- `ndpscrypto_impl.h`
- `ndpscrypto.c` is linked into your Makefile.

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

You need to link the driver into your application.

Netra DPS Crypto and Hash API Functions

API functions are detailed in [“Netra DPS Crypto and Hash API Function Descriptions” on page 105](#). The API functions include the following:

- [“Crypto and Hash Context Setup Part” on page 105](#):

- `NDPSCreateCryptoContext`
- `NDPSTDestroyCryptoContext`

- [“Crypto API” on page 106](#)

- `NDPSCryptKeyLength`
- `NDPSCryptKeyLoad`
- `NDPSCryptIVLoad`
- `NDPSCrypt`
- `NDPSCryptMultiple`

- [“Hash API” on page 110](#):

- `NDPSHashLength`
- `NDPSHashIVLoad`
- `NDPSHashIVGet`
- `NDPSHashDirect`
- `NDPSHashDirectMultiple`

- [“Crypto and Hash Combined API” on page 113](#)

- `NDPSCryptAndHash`

- [“Miscellaneous APIs” on page 114](#)

AES-XCBC-MAC-96 support:

- `NDPSAESXCBCMAC96init`
- `NDPSAESXCBCMAC96fini`
- `NDPSAESXCBCMAC96KeyLoad`
- `NDPSAESXCBCMAC96AuthGenerate`

Netra DPS Crypto and Hash API Function Descriptions

Crypto and Hash Context Setup Part

NDPSCreateCryptoContext

Description

NDPSCreateCryptoContext 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.

Function

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

Parameters

cipher – An algorithm supported in UltraSPARC T2. See `ndpscrypt.h`, among them are 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. The user needs to have `NDPS_crypto_ctx_t` to be able to use other API functions.

Note – To implement, use the following approach: since each core owns one SPU, any strands on the same core accesses the same SPU, therefore, your routine to access SPU depends on the strand you are running. For UltraSPARC T2 platforms, the CG you are accessing = strand# % 8. If your application has only one strand per core to access the SPU on that core, then no other action is required. If you have two functions running different strands on the same core and both access the SPU. You then you need to either place the mutex around the crypto function by accessing the SPU, or you can allocate one strand whose task is to access the SPU only. It then handles your 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.

Function

```
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.

Function

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

Parameters

ctx – The NDPS_crypto_ctx_t handle

key_len – The key length

Return Values

0 – Success

1 – Failure

NDPSCryptKeyLoad

Description

Loads the Key for the crypto.

Function

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

Parameters

ctx – The NDPS_crypto_ctx_t handle

key – The key

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

Return Values

0 – Success

1 – Failure

NDPSCryptIVLoad

Description

Loads the IV for the crypto.

Function

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

Parameters

ctx – The NDPS_crypto_ctx_t handle

iv – The iv

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

Return Values

0 – Success

1 – Failure

NDPSCrypt

Description

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

Function

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

Parameters

ctx – The NDPS_crypto_ctx_t handle

encrypt_flag = 1 – For encrypt

encrypt_flag = 0 – For decrypt

inbuf – The 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.

Function

```
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 – The NDPS_crypto_ctx_t handle
encrypt_flag = 1 – For encrypt
encrypt_flag = 0 – For decrypt
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

Hash API

NDPSHashLength

Description

Sets the Hash length.

Function

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

Parameters

ctx – The NDPS_crypto_ctx_t handle

len – The 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()

Function

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

Parameters

ctx – The NDPS_crypto_ctx_t handle

iv – The hash IV value

Return Values

0 – Success

1 – Failure

NDPSTHashIVGet

Description

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

Function

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

Parameters

ctx – The NDPS_crypto_ctx_t handle

iv – The pointer to the IV location

Return Values

0 – Success

1 – Failure

NDPSTHashDirect

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.

Function

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

Parameters

ctx – The NDPS_crypto_ctx_t handle

inbuf – The input data to be hashed

inlen – The length of data to be hashed

outbuf – The resulting hash value

Return Values

0 – Success

1 – Failure

NDPSTHashDirectMultiple

Description

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

Function

```
int NDPSTHashDirectMultiple (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. It calls SPU in one call to get a performance boost.

Function

```
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 – The `NDPS_crypto_ctx_t` for crypto; handle

encrypt_flag = 1 – For encrypt

encrypt_flag = 0 – For decrypt

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 – The `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.

Function

```
int NDPSAESXCBCMAC96init();
```

Parameters

None

Return Values

0 – Success

1 – Failure

NDPSAESXCBCMAC96fini

Description

Finalizes AES-XCBC-MAC-96.

Function

```
int NDPSAESXCBCMAC96fini();
```

Parameters

None

Return Values

0 – Success

1 – Failure

NDPSAESXCBCMAC96KeyLoad

Description

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

Function

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

Parameters

ctx – The NDPS_crypto_ctx_t handle for crypto

key – The key

Return Values

0 – Success

1 – Failure

NDPSAESXCBCMAC96AuthGenerate

Description

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

Function

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

Parameters

ctx – The NDPS_crypto_ctx_t handle for crypto

inbuf – The input data for AES-XCBC-MAC-96

inlen – The input lengths for AES-XCBC-MAC-96

auth_buf – The 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 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: <code>eth_open</code> , <code>eth_close</code> , <code>eth_read</code>
Device Driver	For example: <code>nxge</code> , <code>ipge</code>
Ethernet Device	For example: Sun multithreaded 10GbE with NIU, Ophir

Network Applications

Network Applications are any applications that requires network hardware resources. The RLP, IP packet forwarding, and IPSec reference applications are all network applications.

Ethernet Device and Device Driver

[TABLE 4-2](#) lists the Ethernet devices supported in Netra DPS platforms.

TABLE 4-2 Ethernet Devices Supported on Netra DPS Platforms

Device Driver	Ethernet Device
<code>ipge</code>	Intel Ophir chip (82571) Ethernet
<code>nxge</code>	Sun Neptune/NIU Ethernet

Ethernet API Functions

The API list of functions include the following:

- [“eth_pbuf_alloc” on page 117](#)
- [“eth_pbuf_free” on page 118](#)
- [“eth_buf_alloc” on page 118](#)
- [“eth_buf_free” on page 119](#)
- [“eth_open” on page 119](#)
- [“eth_close” on page 120](#)
- [“eth_read” on page 120](#)
- [“eth_write” on page 121](#)
- [“eth_ioc” on page 122](#)

Description of Ethernet API Functions

`eth_pbuf_alloc`

Description

This function is used to allocate 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`) consists of the all necessary pointers and fields for manipulating the data buffer. See `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 in [“Notes” on page 123](#)). The device driver calls this function.

Function

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

Parameters

hook – User provided hook. (See Note 1 in [“Notes” on page 123](#))

bufisz – User provided buffer size to be allocated. (See Note 2 in [“Notes” on page 123](#).)

pool – DMA channel pool (See Note 3 in [“Notes” on page 123](#).)

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

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

Function

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

Parameters

hook – User provided hook. (See Note 1 in [“Notes” on page 123.](#))

mblkp – Pointer to message block to be freed

arg – Not used (pass in `NULL`)

pool – DMA channel pool. (See Note 3 in [“Notes” on page 123.](#))

`eth_buf_alloc`

Description

This function is used to allocate 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 in [“Notes” on page 123.](#)). The device driver calls this function.

Function

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

Parameters

hook – User provided hook. (See Note 1 in [“Notes” on page 123.](#))

bufsz – User provided buffer size to be allocated. (See Note 2 in [“Notes” on page 123.](#))

pool – DMA channel pool (See Note 3 in [“Notes” on page 123.](#))

Return Values

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

eth_buf_free

Description

This function is used to free a buffer allocated by `eth_buf_alloc()`. This function is implemented in the user application space. (See Note 4 in [“Notes” on page 123.](#)) The device driver calls this function.

Function

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

Parameters

hook – User provided hook. (See Note 1 in [“Notes” on page 123.](#))

buf – Pointer to data buffer to be freed

arg – Not used (pass in NULL)

pool – DMA channel pool (See Note 3 in [“Notes” on page 123.](#))

eth_open

Description

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

Function

```
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 in [“Notes” on page 123.](#))

txhook – Application provided hook to tx fastq table. (See Note 6 in “Notes” on [page 123](#).)

rxhook – Application provided hook to rx fastq table (See Note 7 in “Notes” on [page 123](#).)

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 in “Notes” on [page 123](#).)

Return Values

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

On error – Returns `INVALID_IHANDLE`

`eth_close`

Description

This function is used to release the ethernet interface instance and all resources held by it.

Function

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

Parameters

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

Return Values

0 – Success

1 – Failure

`eth_read`

Description

This function is used to receive messages from the ethernet interface instance specified by `ihandle`. It 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 non-blocking.

Function

```
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 in [“Notes” on page 123.](#))

Return Values

On success – Returns `mblk` packet chain containing message

On error – Returns `NULL`

`eth_write`

Description

This function is used to send a message which is specified by the message block structure pointer (`mblk`). This function is non-blocking and can fail if the hardware transmit descriptor ring is full.

Function

```
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 in [“Notes” on page 123.](#))

mblkp – Message block pointer. This 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

eth_ioc

Description

This function is a catch all configuration API that can be used to control the device driver attributes.

Function

```
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 in [“Notes” on page 123.](#))

**arg* – Argument passed to command

Return Values

0 – Success

1 – Failure

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
<code>eth_pbuf_alloc</code>	Allocate message block	User application	Device driver
<code>eth_pbuf_free</code>	Free message block	User application	Device driver
<code>eth_buf_alloc</code>	Allocate buffer	User application	Device driver
<code>eth_buf_free</code>	Free buffer	User application	Device driver
<code>eth_open</code>	Find and init device	Device driver	User application
<code>eth_close</code>	Free up device resource	Device driver	User application
<code>eth_read</code>	Poll for received packet	Device driver	User application
<code>eth_write</code>	Send a packet	Device driver	User application
<code>eth_ioc</code>	Device Control	Device driver	User application

Notes

Note 1

This is a “catch all” argument for the user application. It 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,niux
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,nius1
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,nius1
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 number 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 needs to pass in the pointer to the transmit `fastq` allocated by the application.

Note 7

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

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 Ophir (`ipge`) only supports one DMA channel. Therefore, if the application uses `ipge`, it should always pass the number zero into this parameter. The number of DMA channels to be used is specified when calling `eth_open()`.

Note 10

In Netra DPS 2.0, the following control commands are implemented:

- `ETH_IOC_GET_LINK` – Acquire ethernet link status
- `ETH_IOC_SET_CLASSIFY` – Set TCAM classification attributes
- `ETH_IOC_GET_STATS` – Get network statistics
- `ETH_IOC_CHK_ERRS` – Check device errors

Refer to the reference application (for example, RLP, IP packet forwarding, and IPSec) for their usage.

Fast Queue API

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

- [“Fast Queue API Introduction” on page 127](#)
- [“Fast Queue API Function Descriptions” on page 128](#)

Fast Queue API Introduction

The Fast Queue API provide 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 and is 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 need to be protected with locks when they are 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

This function creates a new instance of the fast queue.

Function

```
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

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

Function

```
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

This function dequeues a pointer to a node from the queue.

Function

`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_get_size`

Description

This function returns a value which is the depth of the queue at a given point of time.

Function

`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

This function checks if the queue is empty.

Function

`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

This function checks if the queue is full.

Function

`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.

Interprocess Communication API

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

- [“Interprocess Communication API Introduction” on page 131](#)
- [“Common Programming Interfaces” on page 132](#)
- [“IPC Framework Programming Interfaces” on page 135](#)
- [“IPC Programming Interfaces for Solaris Domains” on page 138](#)

Interprocess Communication API Introduction

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

This chapter describes the APIs available for such communications. It is divided into the common API that is available in all operating environments, the API needed to manage IPC communications in the Netra DPS runtime, and the API for use by Solaris processes.

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

This function 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.

Function

```
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 needs to be passed to the `tx/rx/free` functions.

`ipc_register_callbacks`

Description

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

Function

```
int
ipc_register_callbacks ipc_hdl,
                      event_handler_ft evt_hdlr,
                      rx_handler_ft ipc_hdlr,
                      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

This function 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.

Function

```
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, it makes the message available to be read by the `read()` system call.

Function

```
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.

Function

```
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 Netra DPS runtime environment domain, the interfaces described in the section “[Common Programming Interfaces](#)” on page 132 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 programmer must ensure that every channel is polled periodically. This section describes the API for these tasks.

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

`tnipc_init`

Description

This function must be called in the initialization routine. This function must be called after the LDom framework has been initialized, that is, `mach_descrip_init()`, `lwrtipc_init()` and `lwrtipc_init_ldc()` must be called first.

Function

```
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, it may be called as part of the main loop in the statistics thread. When a message is received, this ensures that the callback function registered for the channel and IPC type is called.

Function

```
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

This function 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.

Function

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

Parameter

ipc_hdl – The 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

This function 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.

Function

```
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

This function 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()`.

Function

```
int
tnipc_unregister_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.

IPC Programming Interfaces for Solaris Domains

In the Solaris Operating Environment, 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, which is located in the directory `src/solaris/include/sys` in the `SUNWndps` package, must be included.

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 132](#) are used.

Fastpath Manager API

This chapter describes the Fastpath Manager API. Topics include:

- [“Fastpath Manager API Introduction” on page 139](#)
- [“Fastpath Manager API Function Descriptions” on page 140](#)

Fastpath Manager API Introduction

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

The fastpath manager is run on a dedicated thread, and in an LDom 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, you must be aware that there is no pre-emption, so the actual granularity is dependent 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.

Function

```
void fastpath_mgr_init()
```

Parameters

None

Return Values

None

`fastpath_mgr_process`

Description

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

Function

```
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.

Function

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

Unregister 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.

Function

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

Parameters

hdl – The handle obtained from `fastpath_mgr_register_event_handler()`.

Return Values

0 – Success

-1 – Failure

`fastpath_mgr_check`

Description

This function 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 sure that it is run with a sufficient frequency.

Function

```
void fastpath_mgr_check(boolean_t poll_ipc)
```

Parameters

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

Return Values

None

Index

C

- C library support on bare hardware, 94
- CMT- specific profiler groups, 100
- CMT-specific hardware architecture
 - description, 56
 - properties, 58
 - types, 57
- CMT-specific software architecture
 - properties, 60
 - types, 59
- Configuration API, 1
- Crypto and Hash Combined API, 113
- Crypto and Hash Context Setup Part, 105

D

- driver API
 - ethernet API, 116
 - miscellaneous API, 114
 - Netra DPS Crypto and Hashing API, 103

E

- error-handling API
 - data types, 54
 - description, 54
 - functions, 54
- ethernet API
 - API descriptions, 105, 117
 - functions, 117
- ethernet device and device driver, 116

F

- fast queue API, 127

- fastpath manager API, 139

- finite state automata API

- defines, 88
 - description, 87
 - macros, 88

H

- hardware architecture API

- data types, 2
 - description, 1
 - functions, 3

- Hash API, 110

I

- interprocess communication (IPC) API, 131

- IPC API

- common programming interfaces, 132
 - framework programming interfaces, 135
 - programming interfaces for Solaris
 - domains, 138
 - Solaris domain, 138
 - user space, Solaris domain, 138

K

- kernel, Solaris domain, 138

L

- late-binding API

- channel functions, 71
 - data types, 62
 - description, 61
 - interruptible wait, 73

- macros, 62
- memory pool functions, 68
- mutex functions, 62
- queue functions, 65

M

- map API
 - data types, 51
 - description, 50
 - functions, 51

N

- Netra DPS Crypto and Hashing API
 - description, 103
- Netra DPS Runtime API
 - data types, 77
 - description, 77
 - memory management functions, 78
 - miscellaneous functions, 83
 - thread functions, 80

P

- profiler API
 - configuration, 95
 - data types, 96
 - functions, 97
 - profiler constants, 100

S

- software architecture API
 - data types, 35
 - description, 35
 - functions, 36
- Solaris domain and IPC, 138

U

- user API, 61
 - hardware specific miscellaneous functions, 93