ChorusOS man pages section 9DKI: Driver to Kernel Interface

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900
U.S.A.

Part No: 806-3342
December 10, 1999

Contents

PREFACE

Overview

A man page is provided for both the naive user, and sophisticated user who is familiar with the ChorusOS™ operating system and is in need of on-line information. A man page is intended to answer concisely the question “What does it do?” The man pages in general comprise a reference manual. They are not intended to be a tutorial.

The following is a list of sections in the ChorusOS man pages and the information it references:

- *Section 1CC: User Utilities; Host and Target Utilities*
- *Section 1M: System Management Utilities*
- *Section 2DL: System Calls; Data Link Services*
- *Section 2K: System Calls; Kernel Services*
- *Section 2MON: System Calls; Monitoring Services*
- *Section 2POSIX: System Calls; POSIX System Calls*
- *Section 2RESTART: System Calls; Hot Restart and Persistent Memory*
- *Section 2SEG: System Calls; Virtual Memory Segment Services*
- *Section 3FTPD: Libraries; FTP Daemon*
- *Section 3M: Libraries; Mathematical Libraries*
- *Section 3POSIX: Libraries; POSIX Library Functions*
- *Section 3RPC: Libraries; RPC Services*
- *Section 3STDC: Libraries; Standard C Library Functions*
- *Section 3TELD: Libraries; Telnet Services*
- *Section 4CC: Files*

- *Section 5FEA: ChorusOS Features and APIs*
- *Section 7P: Protocols*
- *Section 7S: Services*
- *Section 9DDI: Device Driver Interfaces*
- *Section 9DKI: Driver to Kernel Interface*
- *Section 9DRV: Driver Implementations*

ChorusOS man pages are grouped in Reference Manuals, with one reference manual per section.

Below is a generic format for man pages. The man pages of each manual section generally follow this order, but include only needed headings. For example, if there are no bugs to report, there is no BUGS section. See the `intro` pages for more information and detail about each section, and `man`(1) for more information about man pages in general.

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

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

The following special characters are used in this section:

[ ] The option or argument enclosed in these brackets is optional. If the brackets are omitted, the argument must be specified.

. . . Ellipses. Several values may be provided for the previous argument, or the previous argument can be specified multiple times, for example, ‘ "`filename` . . ." .

| Separator. Only one of the arguments separated by this character can be specified at time.

{ } Braces. The options and / or arguments enclosed within braces are

	interdependent, such that everything enclosed must be treated as a unit.
FEATURES	This section provides the list of features which offer an interface. An API may be associated with one or more system features. The interface will be available if one of the associated features has been configured.
DESCRIPTION	This section defines the functionality and behavior of the service. Thus it describes concisely what the command does. It does not discuss OPTIONS or cite EXAMPLES.. Interactive commands, subcommands, requests, macros, functions and such, are described under USAGE.
OPTIONS	This lists the command options with a concise summary of what each option does. The options are listed literally and in the order they appear in the SYNOPSIS section. Possible arguments to options are discussed under the option, and where appropriate, default values are supplied.
OPERANDS	This section lists the command operands and describes how they affect the actions of the command.
OUTPUT	This section describes the output - standard output, standard error, or output files - generated by the command.
RETURN VALUES	If the man page documents functions that return values, this section lists these values and describes the conditions under which they are returned. If a function can return only constant values, such as 0 or –1, these values are listed in tagged paragraphs. Otherwise, a single paragraph describes the return values of each function. Functions declared void do not return values, so they are not discussed in RETURN VALUES.
ERRORS	On failure, most functions place an error code in the global variable `errno` indicating why they failed. This section lists alphabetically all error codes a function can generate and describes the conditions that cause each error. When more than one condition can cause the same error, each condition is described in a separate paragraph under the error code.

USAGE	This section is provided as a guidance on use. This section lists special rules, features and commands that require in-depth explanations. The subsections listed below are used to explain built-in functionality: Commands Modifiers Variables Expressions Input Grammar
EXAMPLES	This section provides examples of usage or of how to use a command or function. Wherever possible a complete example including command line entry and machine response is shown. Whenever an example is given, the prompt is shown as `example%` or if the user must be superuser, `example#`. Examples are followed by explanations, variable substitution rules, or returned values. Most examples illustrate concepts from the SYNOPSIS, DESCRIPTION, OPTIONS and USAGE sections.
ENVIRONMENT VARIABLES	This section lists any environment variables that the command or function affects, followed by a brief description of the effect.
EXIT STATUS	This section lists the values the command returns to the calling program or shell and the conditions that cause these values to be returned. Usually, zero is returned for successful completion and values other than zero for various error conditions.
FILES	This section lists all filenames referred to by the man page, files of interest, and files created or required by commands. Each is followed by a descriptive summary or explanation.
SEE ALSO	This section lists references to other man pages, in-house documentation and outside publications.
DIAGNOSTICS	This section lists diagnostic messages with a brief explanation of the condition causing the error.
WARNINGS	This section lists warnings about special conditions which could seriously affect your working conditions. This is not a list of diagnostics.
NOTES	This section lists additional information that does not belong anywhere else on the page. It takes the form of an aside to the user, covering points of special interest. Critical information is never covered here.

BUGS

This section describes known bugs and wherever possible, suggests workarounds.

CHAPTER

Driver to Kernel Interface

NAME

Intro – driver kernel interface introduction

SYNOPSIS

#include <dki/dki.h>

#include <dki/f_dki.h>

FEATURES

DKI

DESCRIPTION

Provides Driver/Kernel interface services

EXTENDED DESCRIPTION

The Drivers/Kernel Interface or DKI, defines all services provided by the ChorusOS microkernel in order to write driver components.

The DKI is composed of:

- A "Common DKI API", which defines services common to all platforms and processors, and which are usable by all kinds of drivers, notwithstanding the layer level.
- Various "Processor family specific DKI" APIs, which are defined and available only for a given processor family and which should be used only by the lowest-level drivers. That is, drivers for busses and devices which are directly connected to the processor's local bus.

Currently, "Processor family specific DKI" APIs are defined for the following processors:

- PowerPC 60x (603x, 604x, MPC750)
- Intel (ix86 ...)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

Common DKI:

`svDkiOpen`(9DKI), `dtreeNodeRoot`(9DKI), `svDriverRegister`(9DKI), `svDeviceRegister`(9DKI), `svMemAlloc`(9DKI), `svPhysAlloc`(9DKI), `svPhysMap`(9DKI), `svDkiThreadCall`(9DKI), `svTimeoutSet`(9DKI), `usecBusyWait`(9DKI), `loadSwap`(9DKI), `DISABLE_PREEMPT`(9DKI), `imsIntrMask_f`(9DKI)

PowerPC DKI:

`loadSwapEieio_16`(9DKI), `svIntrAttach`(9DKI), `svPhysMap`(9DKI), `dataCacheBlockFlush`(9DKI),

Intel ix86:

svIntrAttach(9DKI), svPhysMap(9DKI), ioLoad(9DKI),

Name	Description
DISABLE_PREEMPT(9DKI)	thread preemption disabling; thread preemption enabling
ENABLE_PREEMPT(9DKI)	See DISABLE_PREEMPT(9DKI)
dataCacheBlockFlush(9DKI)	cache management
dataCacheBlockFlush_powerpc(9DKI)	PowerPC cache management
dataCacheBlockInvalidate(9DKI)	See dataCacheBlockFlush(9DKI)
dataCacheBlockInvalidate_powerpc(9DKI)	See dataCacheBlockFlush_powerpc(9DKI)
dataCacheInvalidate(9DKI)	See dataCacheBlockFlush(9DKI)
dataCacheInvalidate_powerpc(9DKI)	See dataCacheBlockFlush_powerpc(9DKI)
dcacheBlockFlush(9DKI)	See icacheInval(9DKI)
dcacheBlockFlush_usparc(9DKI)	See icacheInval_usparc(9DKI)
dcacheFlush(9DKI)	See icacheInval(9DKI)
dcacheFlush_usparc(9DKI)	See icacheInval_usparc(9DKI)
dcacheLineFlush(9DKI)	See icacheInval(9DKI)
dcacheLineFlush_usparc(9DKI)	See icacheInval_usparc(9DKI)
dtreeNodeAdd(9DKI)	See dtreeNodeRoot(9DKI)
dtreeNodeAlloc(9DKI)	See dtreeNodeRoot(9DKI)
dtreeNodeAttach(9DKI)	See dtreeNodeRoot(9DKI)
dtreeNodeChild(9DKI)	See dtreeNodeRoot(9DKI)
dtreeNodeDetach(9DKI)	See dtreeNodeRoot(9DKI)
dtreeNodeFind(9DKI)	See dtreeNodeRoot(9DKI)

dtreeNodeFree(9DKI)	See dtreeNodeRoot(9DKI)
dtreeNodeParent(9DKI)	See dtreeNodeRoot(9DKI)
dtreeNodePeer(9DKI)	See dtreeNodeRoot(9DKI)
dtreeNodeRoot(9DKI)	device tree operations
dtreePathGet(9DKI)	See dtreeNodeRoot(9DKI)
dtreePathLeng(9DKI)	See dtreeNodeRoot(9DKI)
dtreePropAdd(9DKI)	See dtreeNodeRoot(9DKI)
dtreePropAlloc(9DKI)	See dtreeNodeRoot(9DKI)
dtreePropAttach(9DKI)	See dtreeNodeRoot(9DKI)
dtreePropDetach(9DKI)	See dtreeNodeRoot(9DKI)
dtreePropFind(9DKI)	See dtreeNodeRoot(9DKI)
dtreePropFindNext(9DKI)	See dtreeNodeRoot(9DKI)
dtreePropFree(9DKI)	See dtreeNodeRoot(9DKI)
dtreePropLength(9DKI)	See dtreeNodeRoot(9DKI)
dtreePropName(9DKI)	See dtreeNodeRoot(9DKI)
dtreePropValue(9DKI)	See dtreeNodeRoot(9DKI)
eieio(9DKI)	See loadSwapEieio_16(9DKI)
eieio_powerpc(9DKI)	See loadSwapEieio_16_powerpc(9DKI)
icacheBlockInval(9DKI)	See icacheInval(9DKI)
icacheBlockInval_usparc(9DKI)	See icacheInval_usparc(9DKI)
icacheInval(9DKI)	cache management
icacheInval_usparc(9DKI)	UltraSPARC cache management
icacheLineInval(9DKI)	See icacheInval(9DKI)
icacheLineInval_usparc(9DKI)	See icacheInval_usparc(9DKI)
instCacheBlockInvalidate(9DKI)	See dataCacheBlockFlush(9DKI)

instCacheBlockInvalidate_powerpc(9DKI)	See dataCacheBlockFlush_powerpc(9DKI)
instCacheInvalidate(9DKI)	See dataCacheBlockFlush(9DKI)
instCacheInvalidate_powerpc(9DKI)	See dataCacheBlockFlush_powerpc(9DKI)
ioLoad16(9DKI)	See ioLoad8(9DKI)
ioLoad16_x86(9DKI)	See ioLoad8_x86(9DKI)
ioLoad32(9DKI)	See ioLoad8(9DKI)
ioLoad32_x86(9DKI)	See ioLoad8_x86(9DKI)
ioLoad8(9DKI)	I/O services
ioLoad8_x86(9DKI)	Intel x86 specific I/O services
ioRead16(9DKI)	See ioLoad8(9DKI)
ioRead16_x86(9DKI)	See ioLoad8_x86(9DKI)
ioRead32(9DKI)	See ioLoad8(9DKI)
ioRead32_x86(9DKI)	See ioLoad8_x86(9DKI)
ioRead8(9DKI)	See ioLoad8(9DKI)
ioRead8_x86(9DKI)	See ioLoad8_x86(9DKI)
ioStore16(9DKI)	See ioLoad8(9DKI)
ioStore16_x86(9DKI)	See ioLoad8_x86(9DKI)
ioStore32(9DKI)	See ioLoad8(9DKI)
ioStore32_x86(9DKI)	See ioLoad8_x86(9DKI)
ioStore8(9DKI)	See ioLoad8(9DKI)
ioStore8_x86(9DKI)	See ioLoad8_x86(9DKI)
ioSync_powerpc(9DKI)	See loadSwapEieio_16_powerpc(9DKI)
ioWrite16(9DKI)	See ioLoad8(9DKI)
ioWrite16_x86(9DKI)	See ioLoad8_x86(9DKI)
ioWrite32(9DKI)	See ioLoad8(9DKI)
ioWrite32_x86(9DKI)	See ioLoad8_x86(9DKI)

ioWrite8(9DKI)	See ioLoad8(9DKI)
ioWrite8_x86(9DKI)	See ioLoad8_x86(9DKI)
loadSwapEieio_16(9DKI)	i/o services
loadSwapEieio_16_powerpc(9DKI)	PowerPC specific i/o services
loadSwapEieio_32(9DKI)	See loadSwapEieio_16(9DKI)
loadSwapEieio_32_powerpc(9DKI)	See loadSwapEieio_16_powerpc(9DKI)
loadSwapSync_16_powerpc(9DKI)	See loadSwapEieio_16_powerpc(9DKI)
loadSwapSync_32_powerpc(9DKI)	See loadSwapEieio_16_powerpc(9DKI)
loadSwap_16(9DKI)	specific i/o services
loadSwap_32(9DKI)	See loadSwap_16(9DKI)
loadSwap_64(9DKI)	See loadSwap_16(9DKI)
loadSwap_sync_16_usparc(9DKI)	See load_sync_8_usparc(9DKI)
loadSwap_sync_32_usparc(9DKI)	See load_sync_8_usparc(9DKI)
loadSwap_sync_64_usparc(9DKI)	See load_sync_8_usparc(9DKI)
load_sync_16_usparc(9DKI)	See load_sync_8_usparc(9DKI)
load_sync_32_usparc(9DKI)	See load_sync_8_usparc(9DKI)
load_sync_64_usparc(9DKI)	See load_sync_8_usparc(9DKI)
load_sync_8_usparc(9DKI)	UltraSparc specific i/o services
storeSwapEieio_16(9DKI)	See loadSwapEieio_16(9DKI)
storeSwapEieio_16_powerpc(9DKI)	See loadSwapEieio_16_powerpc(9DKI)
storeSwapEieio_32(9DKI)	See loadSwapEieio_16(9DKI)

storeSwapEieio_32_powerpc(9DKI)	See loadSwapEieio_16_powerpc(9DKI)
storeSwapSync_16_powerpc(9DKI)	See loadSwapEieio_16_powerpc(9DKI)
storeSwapSync_32_powerpc(9DKI)	See loadSwapEieio_16_powerpc(9DKI)
storeSwap_16(9DKI)	See loadSwap_16(9DKI)
storeSwap_32(9DKI)	See loadSwap_16(9DKI)
storeSwap_64(9DKI)	See loadSwap_16(9DKI)
storeSwap_sync_16_usparc(9DKI)	See load_sync_8_usparc(9DKI)
storeSwap_sync_32_usparc(9DKI)	See load_sync_8_usparc(9DKI)
storeSwap_sync_64_usparc(9DKI)	See load_sync_8_usparc(9DKI)
store_sync_16_usparc(9DKI)	See load_sync_8_usparc(9DKI)
store_sync_32_usparc(9DKI)	See load_sync_8_usparc(9DKI)
store_sync_64_usparc(9DKI)	See load_sync_8_usparc(9DKI)
store_sync_8_usparc(9DKI)	See load_sync_8_usparc(9DKI)
svAsyncExcepAttach(9DKI)	asynchronous exceptions management
svAsyncExcepAttach_usparc(9DKI)	UltraSPARC aynchronous exceptions management
svAsyncExcepDetach(9DKI)	See svAsyncExcepAttach(9DKI)
svAsyncExcepDetach_usparc(9DKI)	See svAsyncExcepAttach_usparc(9DKI)
svDeviceAlloc(9DKI)	See svDeviceRegister(9DKI)
svDeviceEntry(9DKI)	See svDeviceRegister(9DKI)
svDeviceEvent(9DKI)	See svDeviceRegister(9DKI)

svDeviceFree(9DKI)	See svDeviceRegister(9DKI)
svDeviceLookup(9DKI)	See svDeviceRegister(9DKI)
svDeviceRegister(9DKI)	device registry operations
svDeviceRelease(9DKI)	See svDeviceRegister(9DKI)
svDeviceUnregister(9DKI)	See svDeviceRegister(9DKI)
svDkiClose(9DKI)	See svDkiOpen(9DKI)
svDkiEvent(9DKI)	See svDkiOpen(9DKI)
svDkiOpen(9DKI)	system event management
svDkiThreadCall(9DKI)	call a routine in the DKI thread context
svDkiThreadTrigger(9DKI)	See svDkiThreadCall(9DKI)
svDriverCap(9DKI)	See svDriverRegister(9DKI)
svDriverEntry(9DKI)	See svDriverRegister(9DKI)
svDriverLookupFirst(9DKI)	See svDriverRegister(9DKI)
svDriverLookupNext(9DKI)	See svDriverRegister(9DKI)
svDriverRegister(9DKI)	driver registry operations
svDriverRelease(9DKI)	See svDriverRegister(9DKI)
svDriverUnregister(9DKI)	See svDriverRegister(9DKI)
svMemAlloc(9DKI)	A general purpose memory allocator
svMemFree(9DKI)	See svMemAlloc(9DKI)
svPhysAlloc(9DKI)	A special purpose physical memory allocator
svPhysFree(9DKI)	See svPhysAlloc(9DKI)
svPhysMap(9DKI)	physical to virtual memory mapping
svPhysMap_powerpc(9DKI)	PowerPC physical to virtual memory mapping
svPhysMap_usparc(9DKI)	UltraSPARC physical to virtual memory mapping

svPhysMap_x86(9DKI)	Intel x86 physical to virtual memory mapping
svPhysUnmap(9DKI)	See svPhysMap(9DKI)
svPhysUnmap_powerpc(9DKI)	See svPhysMap_powerpc(9DKI)
svPhysUnmap_usparc(9DKI)	See svPhysMap_usparc(9DKI)
svPhysUnmap_x86(9DKI)	See svPhysMap_x86(9DKI)
svTimeoutCancel(9DKI)	See svTimeoutSet(9DKI)
svTimeoutGetRes(9DKI)	See svTimeoutSet(9DKI)
svTimeoutSet(9DKI)	timeout operations
swapEieio_16(9DKI)	See loadSwapEieio_16(9DKI)
swapEieio_16_powerpc(9DKI)	See loadSwapEieio_16_powerpc(9DKI)
swapEieio_32(9DKI)	See loadSwapEieio_16(9DKI)
swapEieio_32_powerpc(9DKI)	See loadSwapEieio_16_powerpc(9DKI)
swap_16(9DKI)	See loadSwap_16(9DKI)
swap_32(9DKI)	See loadSwap_16(9DKI)
swap_64(9DKI)	See loadSwap_16(9DKI)
usecBusyWait(9DKI)	precise busy wait service
vmMapToPhys(9DKI)	See svPhysMap(9DKI)
vmMapToPhys_powerpc(9DKI)	See svPhysMap_powerpc(9DKI)
vmMapToPhys_usparc(9DKI)	See svPhysMap_usparc(9DKI)
vmMapToPhys_x86(9DKI)	See svPhysMap_x86(9DKI)

NAME

dataCacheBlockFlush, dataCacheInvalidate, dataCacheBlockInvalidate, instCacheInvalidate, instCacheBlockInvalidate – cache management

FEATURES

DKI

DESCRIPTION

See the architecture specific man pages:

- `dataCacheBlockFlush_powerpc`(9DKI)
- `dataCacheInvalidate_powerpc`(9DKI)
- `dataCacheBlockInvalidate_powerpc`(9DKI)
- `instCacheInvalidate_powerpc`(9DKI)
- `instCacheBlockInvalidate_powerpc`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

dataCacheBlockFlush_powerpc, dataCacheInvalidate_powerpc, dataCacheBlockInvalidate_powerpc, instCacheInvalidate_powerpc, instCacheBlockInvalidate_powerpc – PowerPC cache management

SYNOPSIS

#include <dki/f_dki.h>
void **dataCacheFlush**(void);

void **dataCacheBlockFlush**(VmAddr *addr*, VmSize *size*);

void **dataCacheInvalidate**(void);

void **dataCacheBlockInvalidate**(VmAddr *addr*, VmSize *size*);

void **instCacheInvalidate**(void);

void **instCacheBlockInvalidate**(VmAddr *addr*, VmSize *size*);

FEATURES

DKI

DESCRIPTION

Provides PowerPC cache management services.

EXTENDED DESCRIPTION

The microkernel provides cache management services, mainly to allow host bus drivers to manage memory coherency for DMA purposes.

Typically, PowerPC family processors have separate instruction and data caches which are virtually indexed and physically tagged. However, the PowerPC architecture does not specify the type or existence of a cache. This allows for various different cache types (unified, or no cache at all). In any case, cache management services should behave consistently. They should do nothing if there is no cache, and data or instruction invalidation routines should be equivalent for a unified cache.

Note - The data cache may operate in either write-through or copy-back mode, on a per line basis, depending on the cached memory attributes. The data/instruction cache size and data/instruction line size are processor implementation specific.

The data/instruction cache configuration is available from the `PPC_PROP_CACHE` property attached to the `NODE_CPU` node. (The cpu node may be found in the device tree as a child node of the root node. The cpu node name is `NODE_CPU` .)

The cache configuration property value is the `PpcPropCache` structure.

The *blockNumber* field specifies the number of cache blocks in each data or instruction cache.

The *blockSize* field specifies the cache block size in bytes.

The *blockSizeShift* field specifies the number of bits to shift right/left to divide/multiply by the cache block size (cache block size is always a power of 2).

`dataCacheFlush` globally flushes and invalidates the data cache.

`dataCacheBlockFlush` flushes and invalidates a given range of addresses within the CPU data cache. The range being flushed is specified by the virtual start address (within the current MMU context) and the range size.

`dataCacheInvalidate` globally invalidates the data cache. Note that all blocks in the data cache are marked as invalid without writing back any modified lines to memory. This function does nothing if the data cache is disabled.

`dataCacheBlockInvalidate` invalidates a given range of addresses within the CPU data cache. The range being invalidated is specified by the virtual start address (within the current MMU context) and the range size.

Note that invalidated blocks are not written back to memory. Note also that this service should only be used to invalidate a small memory range; that is, a memory range smaller than the cache size. Otherwise, global invalidation should be used.

`instCacheInvalidate` globally invalidates the instruction cache; that is, all blocks in the instruction cache are marked as invalid. This function does nothing if the instruction cache is disabled.

`instCacheBlockInvalidate` invalidates a given range of addresses within the CPU instruction cache. The range being invalidated is specified by the virtual start address (within the current MMU context) and the range size.

Note that this service should only be used to invalidate a small memory range; that is, a memory range smaller than the cache size. Otherwise, global invalidation should be used.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
dataCacheFlush	+	+	+	-
dataCacheBlockFlush	+	+	+	-
dataCacheInvalidate	+	+	+	-
dataCacheBlockInvalidate	+	+	+	-
instCacheInvalidate	+	+	+	-
instCacheBlockInvalidate	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `svDkiThreadCall`(9DKI)

NAME

dataCacheBlockFlush, dataCacheInvalidate, dataCacheBlockInvalidate, instCacheInvalidate, instCacheBlockInvalidate – cache management

FEATURES

DKI

DESCRIPTION

See the architecture specific man pages:

- `dataCacheBlockFlush_powerpc`(9DKI)
- `dataCacheInvalidate_powerpc`(9DKI)
- `dataCacheBlockInvalidate_powerpc`(9DKI)
- `instCacheInvalidate_powerpc`(9DKI)
- `instCacheBlockInvalidate_powerpc`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

dataCacheBlockFlush_powerpc, dataCacheInvalidate_powerpc, dataCacheBlockInvalidate_powerpc, instCacheInvalidate_powerpc, instCacheBlockInvalidate_powerpc – PowerPC cache management

SYNOPSIS

```
#include <dki/f_dki.h>
void dataCacheFlush(void);

void dataCacheBlockFlush(VmAddr addr, VmSize size);

void dataCacheInvalidate(void);

void dataCacheBlockInvalidate(VmAddr addr, VmSize size);

void instCacheInvalidate(void);

void instCacheBlockInvalidate(VmAddr addr, VmSize size);
```

FEATURES

DKI

DESCRIPTION

Provides PowerPC cache management services.

EXTENDED DESCRIPTION

The microkernel provides cache management services, mainly to allow host bus drivers to manage memory coherency for DMA purposes.

Typically, PowerPC family processors have separate instruction and data caches which are virtually indexed and physically tagged. However, the PowerPC architecture does not specify the type or existence of a cache. This allows for various different cache types (unified, or no cache at all). In any case, cache management services should behave consistently. They should do nothing if there is no cache, and data or instruction invalidation routines should be equivalent for a unified cache.

Note - The data cache may operate in either write-through or copy-back mode, on a per line basis, depending on the cached memory attributes. The data/instruction cache size and data/instruction line size are processor implementation specific.

The data/instruction cache configuration is available from the `PPC_PROP_CACHE` property attached to the `NODE_CPU` node. (The cpu node may be found in the device tree as a child node of the root node. The cpu node name is `NODE_CPU` .)

The cache configuration property value is the `PpcPropCache` structure.

The *blockNumber* field specifies the number of cache blocks in each data or instruction cache.

The *blockSize* field specifies the cache block size in bytes.

The *blockSizeShift* field specifies the number of bits to shift right/left to divide/multiply by the cache block size (cache block size is always a power of 2).

`dataCacheFlush` globally flushes and invalidates the data cache.

`dataCacheBlockFlush` flushes and invalidates a given range of addresses within the CPU data cache. The range being flushed is specified by the virtual start address (within the current MMU context) and the range size.

`dataCacheInvalidate` globally invalidates the data cache. Note that all blocks in the data cache are marked as invalid without writing back any modified lines to memory. This function does nothing if the data cache is disabled.

`dataCacheBlockInvalidate` invalidates a given range of addresses within the CPU data cache. The range being invalidated is specified by the virtual start address (within the current MMU context) and the range size.

Note that invalidated blocks are not written back to memory. Note also that this service should only be used to invalidate a small memory range; that is, a memory range smaller than the cache size. Otherwise, global invalidation should be used.

`instCacheInvalidate` globally invalidates the instruction cache; that is, all blocks in the instruction cache are marked as invalid. This function does nothing if the instruction cache is disabled.

`instCacheBlockInvalidate` invalidates a given range of addresses within the CPU instruction cache. The range being invalidated is specified by the virtual start address (within the current MMU context) and the range size.

Note that this service should only be used to invalidate a small memory range; that is, a memory range smaller than the cache size. Otherwise, global invalidation should be used.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
dataCacheFlush	+	+	+	-
dataCacheBlockFlush	+	+	+	-
dataCacheInvalidate	+	+	+	-
dataCacheBlockInvalidate	+	+	+	-
instCacheInvalidate	+	+	+	-
instCacheBlockInvalidate	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `svDkiThreadCall`(9DKI)

NAME

dataCacheBlockFlush, dataCacheInvalidate, dataCacheBlockInvalidate, instCacheInvalidate, instCacheBlockInvalidate – cache management

FEATURES

DKI

DESCRIPTION

See the architecture specific man pages:

- `dataCacheBlockFlush_powerpc`(9DKI)
- `dataCacheInvalidate_powerpc`(9DKI)
- `dataCacheBlockInvalidate_powerpc`(9DKI)
- `instCacheInvalidate_powerpc`(9DKI)
- `instCacheBlockInvalidate_powerpc`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

dataCacheBlockFlush_powerpc, dataCacheInvalidate_powerpc, dataCacheBlockInvalidate_powerpc, instCacheInvalidate_powerpc, instCacheBlockInvalidate_powerpc – PowerPC cache management

SYNOPSIS

```
#include <dki/f_dki.h>
void dataCacheFlush(void);

void dataCacheBlockFlush(VmAddr addr, VmSize size);

void dataCacheInvalidate(void);

void dataCacheBlockInvalidate(VmAddr addr, VmSize size);

void instCacheInvalidate(void);

void instCacheBlockInvalidate(VmAddr addr, VmSize size);
```

FEATURES

DKI

DESCRIPTION

Provides PowerPC cache management services.

EXTENDED DESCRIPTION

The microkernel provides cache management services, mainly to allow host bus drivers to manage memory coherency for DMA purposes.

Typically, PowerPC family processors have separate instruction and data caches which are virtually indexed and physically tagged. However, the PowerPC architecture does not specify the type or existence of a cache. This allows for various different cache types (unified, or no cache at all). In any case, cache management services should behave consistently. They should do nothing if there is no cache, and data or instruction invalidation routines should be equivalent for a unified cache.

Note - The data cache may operate in either write-through or copy-back mode, on a per line basis, depending on the cached memory attributes. The data/instruction cache size and data/instruction line size are processor implementation specific.

The data/instruction cache configuration is available from the `PPC_PROP_CACHE` property attached to the `NODE_CPU` node. (The cpu node may be found in the device tree as a child node of the root node. The cpu node name is `NODE_CPU` .)

The cache configuration property value is the `PpcPropCache` structure.

The *blockNumber* field specifies the number of cache blocks in each data or instruction cache.

The *blockSize* field specifies the cache block size in bytes.

The *blockSizeShift* field specifies the number of bits to shift right/left to divide/multiply by the cache block size (cache block size is always a power of 2).

`dataCacheFlush` globally flushes and invalidates the data cache.

`dataCacheBlockFlush` flushes and invalidates a given range of addresses within the CPU data cache. The range being flushed is specified by the virtual start address (within the current MMU context) and the range size.

`dataCacheInvalidate` globally invalidates the data cache. Note that all blocks in the data cache are marked as invalid without writing back any modified lines to memory. This function does nothing if the data cache is disabled.

`dataCacheBlockInvalidate` invalidates a given range of addresses within the CPU data cache. The range being invalidated is specified by the virtual start address (within the current MMU context) and the range size.

Note that invalidated blocks are not written back to memory. Note also that this service should only be used to invalidate a small memory range; that is, a memory range smaller than the cache size. Otherwise, global invalidation should be used.

`instCacheInvalidate` globally invalidates the instruction cache; that is, all blocks in the instruction cache are marked as invalid. This function does nothing if the instruction cache is disabled.

`instCacheBlockInvalidate` invalidates a given range of addresses within the CPU instruction cache. The range being invalidated is specified by the virtual start address (within the current MMU context) and the range size.

Note that this service should only be used to invalidate a small memory range; that is, a memory range smaller than the cache size. Otherwise, global invalidation should be used.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
dataCacheFlush	+	+	+	-
dataCacheBlockFlush	+	+	+	-
dataCacheInvalidate	+	+	+	-
dataCacheBlockInvalidate	+	+	+	-
instCacheInvalidate	+	+	+	-
instCacheBlockInvalidate	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `svDkiThreadCall`(9DKI)

NAME

icacheInval, icacheLineInval, icacheBlockInval, dcacheFlush, dcacheLineFlush, dcacheBlockFlush – cache management

FEATURES

DKI

DESCRIPTION

The microkernel provides cache management services, mainly to allow host bus drivers to manage memory coherency for DMA purposes.

Refer to architecture specific man pages:

- `icacheInval_usparc`(9DKI)
- `icacheLineInval_usparc`(9DKI)
- `icacheBlockInval_usparc`(9DKI)
- `dcacheFlush_usparc`(9DKI)
- `dcacheLineFlush_usparc`(9DKI)
- `dcacheBlockFlush_usparc`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

icacheInval_usparc, icacheLineInval_usparc, icacheBlockInval_usparc, dcacheFlush_usparc, dcacheLineFlush_usparc, dcacheBlockFlush_usparc – UltraSPARC cache management

SYNOPSIS

```
#include <dki/f_dki.h>
void icacheInval(void);

void icacheLineInval(VmAddr addr);

void icacheBlockInval(VmAddr addr, VmSize size);

void dcacheFlush(void);

void dcacheLineFlush(VmAddr addr);

void dcacheBlockFlush(VmAddr addr, VmSize size);
```

FEATURES

DKI

DESCRIPTION

The microkernel provides cache management services, mainly to allow host bus drivers to manage memory coherency for DMA purposes.

EXTENDED DESCRIPTION

UltraSPARC family processors have split instruction and data caches which are directly mapped and virtually indexed. The instruction cache is two-way associative, directly mapped. It is composed of two banks with an LRU replacement mechanism. The data cache is one-way associative, directly mapped and operates in the write-through mode.

The data/instruction caches configuration is available as the `"i-cache"` and `"d-cache"` properties attached to the `"cpu"` . The cpu node may be found in the device tree as a child node of the root node. The cpu node name is `"cpu"` .

The cache configuration property value is the `SparcPropCache` structure shown below.

```
typedef struct {

    uint32_f  csize;   /* cache size */
    uint32_f  bsize;   /* cache bank size */
    uint32_f  lsize;   /* cache line size */
    uint32_f  nbanks;  /* number of bunks (csize = bsize * nbanks) */
    uint32_f  type;    /* cache type */

} SparcPropCache;
```

The *csize* field specifies the cache size in bytes.

The *bsize* field specifies the cache bank size in bytes.

The *lsize* field specifies the cache line size in bytes.

The *nbanks* field specifies the number of banks.

The *type* field specifies the cache properties. It is composed of the following bit-fields and flags:

CACHE_TYPE_IDX	Bit-field specifying the cache indexing mode: CACHE_TYPE_IDX_PHYS — physically indexed CACHE_TYPE_IDX_VIRT — virtually indexed
CACHE_TYPE_TAG	Bit-field specifying the cache tagging mode: CACHE_TYPE_TAG_PHYS — physically tagged CACHE_TYPE_TAG_VIRT — virtually tagged
CACHE_TYPE_MODE	Bit-field specifying the data cache mode: CACHE_TYPE_MODE_WT — write-through CACHE_TYPE_MODE_CB — copy-back
CACHE_TYPE_CFS	Flag specifying that the entire cache flush/invalidation is supported.
CACHE_TYPE_LFS	Flag specifying that line cache flush/invalidation is supported.
CACHE_TYPE_MATCH	Flag specifying that the tag match criteria is used for the line flush/invalidation

Note that, typically, a host bus driver does not need to examine the caches' properties because they are already taken into account by the cache management service routines described below.

`icacheInval` invalidates the entire CPU Instruction Cache.

`icacheLineInval` invalidates a given cache line within the CPU Instruction Cache. The cache line being invalidated is specified by a virtual address.

`icacheBlockInval` invalidates a given range within the CPU Instruction Cache. The range being invalidated is specified by the virtual start address and range size.

`dcacheFlush` flushes and invalidates the entire CPU Data Cache.

`dcacheLineFlush` flushes and invalidates a given cache line within the CPU Data Cache. The cache line being flushed and invalidated is specified by a virtual address.

`dcacheLineFlush` flushes and invalidates a given cache line within the CPU Data Cache. The cache line being flushed and invalidated is specified by a virtual address.

ALLOWED CALLING CONTEXTS

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
icacheInval	+	+	+	-
icacheLineInval	+	+	+	-
icacheBlockInval	+	+	+	-
dcacheFlush	+	+	+	-
dcacheLineFlush	+	+	+	-
dcacheBlockFlush	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

icacheInval, icacheLineInval, icacheBlockInval, dcacheFlush, dcacheLineFlush, dcacheBlockFlush – cache management

FEATURES

DKI

DESCRIPTION

The microkernel provides cache management services, mainly to allow host bus drivers to manage memory coherency for DMA purposes.

Refer to architecture specific man pages:

- `icacheInval_usparc`(9DKI)
- `icacheLineInval_usparc`(9DKI)
- `icacheBlockInval_usparc`(9DKI)
- `dcacheFlush_usparc`(9DKI)
- `dcacheLineFlush_usparc`(9DKI)
- `dcacheBlockFlush_usparc`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

icacheInval_usparc, icacheLineInval_usparc, icacheBlockInval_usparc, dcacheFlush_usparc, dcacheLineFlush_usparc, dcacheBlockFlush_usparc – UltraSPARC cache management

SYNOPSIS

```
#include <dki/f_dki.h>
void icacheInval(void);

void icacheLineInval(VmAddr addr);

void icacheBlockInval(VmAddr addr, VmSize size);

void dcacheFlush(void);

void dcacheLineFlush(VmAddr addr);

void dcacheBlockFlush(VmAddr addr, VmSize size);
```

FEATURES

DKI

DESCRIPTION

The microkernel provides cache management services, mainly to allow host bus drivers to manage memory coherency for DMA purposes.

EXTENDED DESCRIPTION

UltraSPARC family processors have split instruction and data caches which are directly mapped and virtually indexed. The instruction cache is two-way associative, directly mapped. It is composed of two banks with an LRU replacement mechanism. The data cache is one-way associative, directly mapped and operates in the write-through mode.

The data/instruction caches configuration is available as the `"i-cache"` and `"d-cache"` properties attached to the `"cpu"` . The cpu node may be found in the device tree as a child node of the root node. The cpu node name is `"cpu"` .

The cache configuration property value is the `SparcPropCache` structure shown below.

```
typedef struct {

    uint32_f  csize;   /* cache size */
    uint32_f  bsize;   /* cache bank size */
    uint32_f  lsize;   /* cache line size */
    uint32_f  nbanks;  /* number of bunks (csize = bsize * nbanks) */
    uint32_f  type;    /* cache type */

} SparcPropCache;
```

The *csize* field specifies the cache size in bytes.

The *bsize* field specifies the cache bank size in bytes.

The *lsize* field specifies the cache line size in bytes.

The *nbanks* field specifies the number of banks.

The *type* field specifies the cache properties. It is composed of the following bit-fields and flags:

CACHE_TYPE_IDX	Bit-field specifying the cache indexing mode: CACHE_TYPE_IDX_PHYS — physically indexed CACHE_TYPE_IDX_VIRT — virtually indexed
CACHE_TYPE_TAG	Bit-field specifying the cache tagging mode: CACHE_TYPE_TAG_PHYS — physically tagged CACHE_TYPE_TAG_VIRT — virtually tagged
CACHE_TYPE_MODE	Bit-field specifying the data cache mode: CACHE_TYPE_MODE_WT — write-through CACHE_TYPE_MODE_CB — copy-back
CACHE_TYPE_CFS	Flag specifying that the entire cache flush/invalidation is supported.
CACHE_TYPE_LFS	Flag specifying that line cache flush/invalidation is supported.
CACHE_TYPE_MATCH	Flag specifying that the tag match criteria is used for the line flush/invalidation

Note that, typically, a host bus driver does not need to examine the caches' properties because they are already taken into account by the cache management service routines described below.

`icacheInval` invalidates the entire CPU Instruction Cache.

`icacheLineInval` invalidates a given cache line within the CPU Instruction Cache. The cache line being invalidated is specified by a virtual address.

`icacheBlockInval` invalidates a given range within the CPU Instruction Cache. The range being invalidated is specified by the virtual start address and range size.

`dcacheFlush` flushes and invalidates the entire CPU Data Cache.

`dcacheLineFlush` flushes and invalidates a given cache line within the CPU Data Cache. The cache line being flushed and invalidated is specified by a virtual address.

`dcacheLineFlush` flushes and invalidates a given cache line within the CPU Data Cache. The cache line being flushed and invalidated is specified by a virtual address.

ALLOWED CALLING CONTEXTS

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
icacheInval	+	+	+	-
icacheLineInval	+	+	+	-
icacheBlockInval	+	+	+	-
dcacheFlush	+	+	+	-
dcacheLineFlush	+	+	+	-
dcacheBlockFlush	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

icacheInval, icacheLineInval, icacheBlockInval, dcacheFlush, dcacheLineFlush, dcacheBlockFlush – cache management

FEATURES

DKI

DESCRIPTION

The microkernel provides cache management services, mainly to allow host bus drivers to manage memory coherency for DMA purposes.

Refer to architecture specific man pages:

- `icacheInval_usparc`(9DKI)
- `icacheLineInval_usparc`(9DKI)
- `icacheBlockInval_usparc`(9DKI)
- `dcacheFlush_usparc`(9DKI)
- `dcacheLineFlush_usparc`(9DKI)
- `dcacheBlockFlush_usparc`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

icacheInval_usparc, icacheLineInval_usparc, icacheBlockInval_usparc, dcacheFlush_usparc, dcacheLineFlush_usparc, dcacheBlockFlush_usparc – UltraSPARC cache management

SYNOPSIS

```
#include <dki/f_dki.h>
void icacheInval(void);

void icacheLineInval(VmAddr addr);

void icacheBlockInval(VmAddr addr, VmSize size);

void dcacheFlush(void);

void dcacheLineFlush(VmAddr addr);

void dcacheBlockFlush(VmAddr addr, VmSize size);
```

FEATURES

DKI

DESCRIPTION

The microkernel provides cache management services, mainly to allow host bus drivers to manage memory coherency for DMA purposes.

EXTENDED DESCRIPTION

UltraSPARC family processors have split instruction and data caches which are directly mapped and virtually indexed. The instruction cache is two-way associative, directly mapped. It is composed of two banks with an LRU replacement mechanism. The data cache is one-way associative, directly mapped and operates in the write-through mode.

The data/instruction caches configuration is available as the `"i-cache"` and `"d-cache"` properties attached to the `"cpu"` . The cpu node may be found in the device tree as a child node of the root node. The cpu node name is `"cpu"` .

The cache configuration property value is the `SparcPropCache` structure shown below.

```
typedef struct {

    uint32_f  csize;   /* cache size */
    uint32_f  bsize;   /* cache bank size */
    uint32_f  lsize;   /* cache line size */
    uint32_f  nbanks;  /* number of bunks (csize = bsize * nbanks) */
    uint32_f  type;    /* cache type */

} SparcPropCache;
```

The *csize* field specifies the cache size in bytes.

The *bsize* field specifies the cache bank size in bytes.

The *lsize* field specifies the cache line size in bytes.

The *nbanks* field specifies the number of banks.

The *type* field specifies the cache properties. It is composed of the following bit-fields and flags:

`CACHE_TYPE_IDX`	Bit-field specifying the cache indexing mode: `CACHE_TYPE_IDX_PHYS` — physically indexed `CACHE_TYPE_IDX_VIRT` — virtually indexed
`CACHE_TYPE_TAG`	Bit-field specifying the cache tagging mode: `CACHE_TYPE_TAG_PHYS` — physically tagged `CACHE_TYPE_TAG_VIRT` — virtually tagged
`CACHE_TYPE_MODE`	Bit-field specifying the data cache mode: `CACHE_TYPE_MODE_WT` — write-through `CACHE_TYPE_MODE_CB` — copy-back
`CACHE_TYPE_CFS`	Flag specifying that the entire cache flush/invalidation is supported.
`CACHE_TYPE_LFS`	Flag specifying that line cache flush/invalidation is supported.
`CACHE_TYPE_MATCH`	Flag specifying that the tag match criteria is used for the line flush/invalidation

Note that, typically, a host bus driver does not need to examine the caches' properties because they are already taken into account by the cache management service routines described below.

`icacheInval` invalidates the entire CPU Instruction Cache.

`icacheLineInval` invalidates a given cache line within the CPU Instruction Cache. The cache line being invalidated is specified by a virtual address.

`icacheBlockInval` invalidates a given range within the CPU Instruction Cache. The range being invalidated is specified by the virtual start address and range size.

`dcacheFlush` flushes and invalidates the entire CPU Data Cache.

`dcacheLineFlush` flushes and invalidates a given cache line within the CPU Data Cache. The cache line being flushed and invalidated is specified by a virtual address.

`dcacheLineFlush` flushes and invalidates a given cache line within the CPU Data Cache. The cache line being flushed and invalidated is specified by a virtual address.

ALLOWED CALLING CONTEXTS

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
icacheInval	+	+	+	-
icacheLineInval	+	+	+	-
icacheBlockInval	+	+	+	-
dcacheFlush	+	+	+	-
dcacheLineFlush	+	+	+	-
dcacheBlockFlush	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

DISABLE_PREEMPT, ENABLE_PREEMPT – thread preemption disabling; thread preemption enabling

SYNOPSIS

```
#include <dki/dki.h>

DISABLE_PREEMPT();

ENABLE_PREEMPT();
```

FEATURES

DKI

DESCRIPTION

DKI provides a means for a driver to disable/enable the preemption of the current thread. These services can be used by a driver to prevent the current thread being preempted while interrupts are masked at bus/device level.

Note - These services are implemented as macros.

EXTENDED DESCRIPTION

Thread Preemption Disabling

The `DISABLE_PREEMPT()` macro disables preemption of the thread which is currently executing. This macro increments a per-processor preemption mask count. When this count is not zero, the scheduler is locked. This occurs as when there is a preemption request, the scheduler simply raises a pending preemption flag deferring the real thread preemption until the preemption mask count drops to zero.

Thread Preemption Enabling

The `ENABLE_PREEMPT()` macro enables preemption of the thread which is currently executing and was previously disabled by `DISABLE_PREEMPT()` as outlined above. This macro decrements the preemption mask count and, if it drops to zero, checks whether the pending preemption flag is raised and thus if the current thread should be preempted.

Note - As `DISABLE_PREEMPT()` / `ENABLE_PREEMPT()` rely on the preemption mask count, a driver may issue nested calls to these services.

Allowed Calling Contexts

The following table specifies the allowed calling contexts for each service:

Services	Base level	DKI thread	Interrupt	Blocking
DISABLE_PREEMPT	+	+	+	-
ENABLE_PREEMPT	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `svDkiThreadCall`(9DKI)

NAME

dtreeNodeRoot, dtreeNodeChild, dtreeNodePeer, dtreeNodeParent, dtreeNodeAlloc, dtreeNodeFree, dtreeNodeAttach, dtreeNodeDetach, dtreePropFind, dtreePropFindNext, dtreePropLength, dtreePropValue, dtreePropName, dtreePropAlloc, dtreePropFree, dtreePropAttach, dtreePropDetach, dtreeNodeAdd, dtreeNodeFind, dtreePropAdd, dtreePathLeng, dtreePathGet – device tree operations

SYNOPSIS

```
#include <dki/dki.h>
DevNode dtreeNodeRoot(void);

DevNode dtreeNodeChild(DevNode node);

DevNode dtreeNodePeer(DevNode node);

DevNode dtreeNodeParent(DevNode node);

DevNode dtreeNodeAlloc(void);

void dtreeNodeFree(DevNode );

void dtreeNodeAttach(DevNode pnode, DevNode cnode);

void dtreeNodeDetach(DevNode node);

DevProperty dtreePropFind(DevNode node, char * name);

DevProperty dtreePropFindNext(DevProperty prop, char * name);

unsigned int dtreePropLength(DevProperty prop);

void * dtreePropValue(DevProperty prop);

char * dtreePropName(DevProperty prop);

DevProperty dtreePropAlloc(char * name, int length);

void dtreePropFree(DevProperty prop);

void dtreePropAttach(DevNode node, DevProperty prop);

void dtreePropDetach(DevProperty prop);

DevNode dtreeNodeAdd(DevNode parent, char * name);

DevNode dtreeNodeFind(DevNode parent, char * name);

DevProperty dtreePropAdd(DevNode node, char * name, void * value, unsigned int
length);

DevProperty dtreePathLeng(DevNode node, int length);

DevProperty dtreePathGet(DevNode node, char * buf, void * value);
```

FEATURES

DKI

DESCRIPTION

Provides device tree operations.

EXTENDED DESCRIPTION

The device tree is a data structure which provides a description of the hardware topology and device properties. The hardware topology is specified in terms of parent/child relationships. Device properties associated with each device node in the tree are device specific.

A device property is a *name* / *value* pair. The property *name* is a null terminated ASCII string. The property *value* is a sequence of bytes specified by the *length* / *address* pair.

Note that the property value format is property specific and has to be standardized between the given producer and its consumers. For instance, among all device node properties there are some properties related to the bus resources required/allocated for the device (for instance, interrupt lines, I/O registers, DMA channels).

These types of properties must be standardized in order to be understood by the bus driver as well as by any device drivers connected to the given bus. The device tree data structure may be built either statically or dynamically. In the static case, the device tree is populated by the `booter` . For instance, the `booter` may include a pre-defined sequence of device tree function calls. Another use of the `booter` is to build the device tree from a hardware description provided by firmware.

In the dynamic case, the device tree is populated at system initialization time using an enumeration/probing mechanism. The device tree is populated by propagating from parents to children.

Note that it is possible to combine both methods. In other words, an initial (non complete) device tree may be provided by the `booter` which will later be completed dynamically using an enumeration/probing mechanism. However it is implemented, the device tree structure can be modified (extended/truncated) dynamically at run time using hot-plug insertion/removal (for example, PCMCIA cards). The device tree API is described in detail below.

Device Tree Browsing

This section describes APIs related to device tree browsing. `DevNode` is an abstract type designating a device node object, and is opaque to the driver.

`dtreeNodeRoot` returns the root device node if the device tree is not empty, otherwise NULL is returned.

`dtreeNodeChild` returns the first child node from the list of children. NULL is returned when the list of children is empty.

`dtreeNodePeer` returns the next device node from the sibling list, if any, otherwise NULL is returned. The *node* argument specifies the current device node in the sibling list.

`dtreeNodeParent` returns the parent device node, if any, otherwise NULL is returned. The *node* argument specifies the child device node.

`dtreePathLeng` returns the pathname length of a device node.

`dtreePathGet` returns in *buf* the absolute pathname of a device noe. The trailing path, of the pathname, is the name of the node and is read in a node property. If this property does not exist, the trailing path of the returned pathname is set to '???'.

Device Tree Modification

This section describes APIs related to device tree topology modification.

`dtreeNodeAlloc` allocates a new device node object. A non zero DevNode cookie is returned in case of success, otherwise NULL is returned. The allocated node has neither parent nor child nodes. There are no properties attached to the newly allocated node.

`dtreeNodeFree` releases the memory allocated by the node object and all property objects attached to the node. The *node* argument specifies the node object being released.

`dtreeNodeAttach` adds the device node specified by the *cnode* argument in the child list of the parent node specified by the *pnode* argument.

`dtreeNodeDetach` detaches the node object specified by the *node* argument from its parent (if any). When *node* specifies the root node, the device tree is emptied.

Device Node Properties

This section describes APIs related to the device node properties.

`DevProperty` is an abstract type designating a device property object, and is opaque to the driver.

`dtreePropFind` searches the first property within the properties list of the device node specified by the *node* argument. If the *name* argument is not NULL, `dtreePropFind` returns the first property whose name matches the *name* string. If the *name* argument is NULL, `dtreePropFind` returns the first property from the list regardless of its name.

In case of success, a non zero `DevProperty` cookie is returned, otherwise NULL is returned. Once the first property is found, the `dtreePropFindNext` routine may be used in order to find a subsequent property in the list. `dtreePropFindNext` searches the next property within the properties list. The current position within the properties list is specified by the *prop* argument. If the *name* argument is not `NULL` , `dtreePropFindNext` returns the next property (with respect to the current position) whose name matches the *name* string. If the *name* argument is `NULL` , `dtreePropFindNext` returns the next property from the list (with respect to the current position) regardless of its name.

If successful, a non zero `DevProperty` cookie is returned, otherwise `NULL` is returned. The `dtreePropFind` / `dtreePropFindNext` pair are typically used to iterate through either all device node properties, or a subset of device node properties whose names match a given name. In a case where a user knows that there is only one property with a given name attached to the device node, a single `dtreePropFind` invocation is sufficient.

`dtreePropLength` returns the property value length (in bytes). The property object is specified by the *prop* argument.

`dtreePropValue` returns a pointer to the first byte of the property value. The property object is specified by the *prop* argument. A driver can read and write the property value directly using the returned pointer. Typically, the driver will cast the pointer to a well known type/structure in order to access the property value. Note that the property value must always be presented in the CPU endian format.

`dtreePropName` returns a pointer to an ASCII string which designates the property name. The property object is specified by the prop argument.

`dtreePropAlloc` allocates a new device property object. A non zero `DevProperty` cookie is returned if successful, otherwise `NULL` is returned. The *name* argument specifies the property name. The *length* argument specifies the length of the property value. The property value is undefined. The property object allocated is not attached to any device node. Once the property value is initialized, the property object can be attached to a node (that is, added to the node properties list) using the `dtreePropAttach` function.

`dtreePropFree` releases the memory allocated by the property object. The *prop* parameter specifies the property object being released. Note that the property object must not be attached to any device node.

`dtreePropAttach` adds the property object specified by the *prop* argument to the node properties list. The *node* argument specifies the device node to which the property will be attached.

`dtreePropDetach` detaches the property object from the device node to which it is attached (if any). The *prop* argument specifies the property object.

Note that the device tree structure must only be accessed in the DKI thread context. The DKI thread provides access synchronization to the device tree. In many cases, driver routines which typically examine or modify the device tree are already invoked by the DKI thread. For instance, the driver probing, initialization and shut-down code is executed by the DKI thread. In a case when a driver needs to access the device tree at run time, a driver routine which uses the device tree API must be explicitly called in the DKI thread context. This rule also applies to driver clients; a driver client must always switch to the DKI thread context when accessing the device tree structure.

Device Tree High-Level Services

This section describes high-level APIs built upon other basic services. This API implements certain useful services for building and searching the device tree, in order to avoid implementing this code in all device tree users.

`dtreeNodeAdd` allocates a new device node object, and adds it to the child list of the *parent* device node specified..

The *name* argument specifies the name of the new device node allocated.. This means that a `node` property is allocated and attached to the new node with the value specified by *name* . If successful, the newly allocated node object is returned, otherwise, a NULL pointer is returned.

`dtreeNodeFind` looks for a named node in the child list of a specified device node. The *parent* argument specifies the device node within which the child list should be searched. The *name* argument specifies the value which must match the *node* property value of the node being searched.. If successful, the matching node object is returned, otherwise a NULL pointer is returned.

`dtreePropAdd` allocates a new property, sets its value and attaches it to a given device node. The *node* argument specifies the node to which the new property must be attached. The *name* argument specifies the name of the new property to allocate. The *length* argument specifies the memory size in bytes to be allocated for the new property value. The *value* argument specifies the value to be set for the newly allocated property. If successful, the newly allocated property object is returned, otherwise, a NULL pointer is returned.

Allowed Calling Contexts

The following table specifies the contexts in which a caller may invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
dtreeNodeRoot	-	+	-	-
dtreeNodeChild	-	+	-	-
dtreeNodePeer	-	+	-	-
dtreeNodeParent	-	+	-	-
dtreeNodeAlloc	+	+	-	+
dtreeNodeFree	+	+	-	+
dtreeNodeAttach	-	+	-	-
dtreeNodeDetach	-	+	-	-
dtreePropFind	-	+	-	-
dtreePropFindNext	-	+	-	-
dtreePropLength	-	+	-	-

dtreePropValue	-	+	-	-
dtreePropName	-	+	-	-
dtreePropAlloc	+	+	-	+
dtreePropFree	+	+	-	+
dtreePropAttach	-	+	-	-
dtreePropDetach	-	+	-	-
dtreeNodeAdd	-	+	-	+
dtreeNodeFind	-	+	-	-
dtreePropAdd	-	+	-	+
dtreePathLeng	+	+	_	_
dtreePathGet	+	+	_	_

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

dtreeNodeRoot, dtreeNodeChild, dtreeNodePeer, dtreeNodeParent, dtreeNodeAlloc, dtreeNodeFree, dtreeNodeAttach, dtreeNodeDetach, dtreePropFind, dtreePropFindNext, dtreePropLength, dtreePropValue, dtreePropName, dtreePropAlloc, dtreePropFree, dtreePropAttach, dtreePropDetach, dtreeNodeAdd, dtreeNodeFind, dtreePropAdd, dtreePathLeng, dtreePathGet – device tree operations

SYNOPSIS

```
#include <dki/dki.h>
DevNode dtreeNodeRoot(void);

DevNode dtreeNodeChild(DevNode node);

DevNode dtreeNodePeer(DevNode node);

DevNode dtreeNodeParent(DevNode node);

DevNode dtreeNodeAlloc(void);

void dtreeNodeFree(DevNode );

void dtreeNodeAttach(DevNode pnode, DevNode cnode);

void dtreeNodeDetach(DevNode node);

DevProperty dtreePropFind(DevNode node, char * name);

DevProperty dtreePropFindNext(DevProperty prop, char * name);

unsigned int dtreePropLength(DevProperty prop);

void * dtreePropValue(DevProperty prop);

char * dtreePropName(DevProperty prop);

DevProperty dtreePropAlloc(char * name, int length);

void dtreePropFree(DevProperty prop);

void dtreePropAttach(DevNode node, DevProperty prop);

void dtreePropDetach(DevProperty prop);

DevNode dtreeNodeAdd(DevNode parent, char * name);

DevNode dtreeNodeFind(DevNode parent, char * name);

DevProperty dtreePropAdd(DevNode node, char * name, void * value, unsigned int
length);

DevProperty dtreePathLeng(DevNode node, int length);

DevProperty dtreePathGet(DevNode node, char * buf, void * value);
```

FEATURES

DKI

DESCRIPTION

Provides device tree operations.

EXTENDED DESCRIPTION

The device tree is a data structure which provides a description of the hardware topology and device properties. The hardware topology is specified in terms of parent / child relationships. Device properties associated with each device node in the tree are device specific.

A device property is a *name* / *value* pair. The property *name* is a null terminated ASCII string. The property *value* is a sequence of bytes specified by the *length* / *address* pair.

Note that the property value format is property specific and has to be standardized between the given producer and its consumers. For instance, among all device node properties there are some properties related to the bus resources required / allocated for the device (for instance, interrupt lines, I / O registers, DMA channels).

These types of properties must be standardized in order to be understood by the bus driver as well as by any device drivers connected to the given bus. The device tree data structure may be built either statically or dynamically. In the static case, the device tree is populated by the `booter` . For instance, the `booter` may include a pre-defined sequence of device tree function calls. Another use of the `booter` is to build the device tree from a hardware description provided by firmware.

In the dynamic case, the device tree is populated at system initialization time using an enumeration / probing mechanism. The device tree is populated by propagating from parents to children.

Note that it is possible to combine both methods. In other words, an initial (non complete) device tree may be provided by the `booter` which will later be completed dynamically using an enumeration / probing mechanism. However it is implemented, the device tree structure can be modified (extended / truncated) dynamically at run time using hot-plug insertion / removal (for example, PCMCIA cards). The device tree API is described in detail below.

Device Tree Browsing

This section describes APIs related to device tree browsing. `DevNode` is an abstract type designating a device node object, and is opaque to the driver.

`dtreeNodeRoot` returns the root device node if the device tree is not empty, otherwise NULL is returned.

`dtreeNodeChild` returns the first child node from the list of children. NULL is returned when the list of children is empty.

`dtreeNodePeer` returns the next device node from the sibling list, if any, otherwise NULL is returned. The *node* argument specifies the current device node in the sibling list.

`dtreeNodeParent` returns the parent device node, if any, otherwise NULL is returned. The *node* argument specifies the child device node.

`dtreePathLeng` returns the pathname length of a device node.

`dtreePathGet` returns in *buf* the absolute pathname of a device noe. The trailing path, of the pathname, is the name of the node and is read in a node property. If this property does not exist, the trailing path of the returned pathname is set to '???'.

Device Tree Modification

This section describes APIs related to device tree topology modification.

`dtreeNodeAlloc` allocates a new device node object. A non zero DevNode cookie is returned in case of success, otherwise NULL is returned. The allocated node has neither parent nor child nodes. There are no properties attached to the newly allocated node.

`dtreeNodeFree` releases the memory allocated by the node object and all property objects attached to the node. The *node* argument specifies the node object being released.

`dtreeNodeAttach` adds the device node specified by the *cnode* argument in the child list of the parent node specified by the *pnode* argument.

`dtreeNodeDetach` detaches the node object specified by the *node* argument from its parent (if any). When *node* specifies the root node, the device tree is emptied.

Device Node Properties

This section describes APIs related to the device node properties.

`DevProperty` is an abstract type designating a device property object, and is opaque to the driver.

`dtreePropFind` searches the first property within the properties list of the device node specified by the *node* argument. If the *name* argument is not NULL, `dtreePropFind` returns the first property whose name matches the *name* string. If the *name* argument is NULL, `dtreePropFind` returns the first property from the list regardless of its name.

In case of success, a non zero `DevProperty` cookie is returned, otherwise NULL is returned. Once the first property is found, the `dtreePropFindNext` routine may be used in order to find a subsequent property in the list. `dtreePropFindNext` searches the next property within the properties list. The current position within the properties list is specified by the *prop* argument. If the *name* argument is not `NULL` , `dtreePropFindNext` returns the next property (with respect to the current position) whose name matches the *name* string. If the *name* argument is `NULL` , `dtreePropFindNext` returns the next property from the list (with respect to the current position) regardless of its name.

If successful, a non zero `DevProperty` cookie is returned, otherwise `NULL` is returned. The `dtreePropFind` / `dtreePropFindNext` pair are typically used to iterate through either all device node properties, or a subset of device node properties whose names match a given name. In a case where a user knows that there is only one property with a given name attached to the device node, a single `dtreePropFind` invocation is sufficient.

`dtreePropLength` returns the property value length (in bytes). The property object is specified by the *prop* argument.

`dtreePropValue` returns a pointer to the first byte of the property value. The property object is specified by the *prop* argument. A driver can read and write the property value directly using the returned pointer. Typically, the driver will cast the pointer to a well known type/structure in order to access the property value. Note that the property value must always be presented in the CPU endian format.

`dtreePropName` returns a pointer to an ASCII string which designates the property name. The property object is specified by the prop argument.

`dtreePropAlloc` allocates a new device property object. A non zero `DevProperty` cookie is returned if successful, otherwise `NULL` is returned. The *name* argument specifies the property name. The *length* argument specifies the length of the property value. The property value is undefined. The property object allocated is not attached to any device node. Once the property value is initialized, the property object can be attached to a node (that is, added to the node properties list) using the `dtreePropAttach` function.

`dtreePropFree` releases the memory allocated by the property object. The *prop* parameter specifies the property object being released. Note that the property object must not be attached to any device node.

`dtreePropAttach` adds the property object specified by the *prop* argument to the node properties list. The *node* argument specifies the device node to which the property will be attached.

`dtreePropDetach` detaches the property object from the device node to which it is attached (if any). The *prop* argument specifies the property object.

Note that the device tree structure must only be accessed in the DKI thread context. The DKI thread provides access synchronization to the device tree. In many cases, driver routines which typically examine or modify the device tree are already invoked by the DKI thread. For instance, the driver probing, initialization and shut-down code is executed by the DKI thread. In a case when a driver needs to access the device tree at run time, a driver routine which uses the device tree API must be explicitly called in the DKI thread context. This rule also applies to driver clients; a driver client must always switch to the DKI thread context when accessing the device tree structure.

Device Tree High-Level Services

This section describes high-level APIs built upon other basic services. This API implements certain useful services for building and searching the device tree, in order to avoid implementing this code in all device tree users.

`dtreeNodeAdd` allocates a new device node object, and adds it to the child list of the *parent* device node specified..

The *name* argument specifies the name of the new device node allocated.. This means that a `node` property is allocated and attached to the new node with the value specified by *name* . If successful, the newly allocated node object is returned, otherwise, a NULL pointer is returned.

`dtreeNodeFind` looks for a named node in the child list of a specified device node. The *parent* argument specifies the device node within which the child list should be searched. The *name* argument specifies the value which must match the *node* property value of the node being searched.. If successful, the matching node object is returned, otherwise a NULL pointer is returned.

`dtreePropAdd` allocates a new property, sets its value and attaches it to a given device node. The *node* argument specifies the node to which the new property must be attached. The *name* argument specifies the name of the new property to allocate. The *length* argument specifies the memory size in bytes to be allocated for the new property value. The *value* argument specifies the value to be set for the newly allocated property. If successful, the newly allocated property object is returned, otherwise, a NULL pointer is returned.

Allowed Calling Contexts

The following table specifies the contexts in which a caller may invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
dtreeNodeRoot	-	+	-	-
dtreeNodeChild	-	+	-	-
dtreeNodePeer	-	+	-	-
dtreeNodeParent	-	+	-	-
dtreeNodeAlloc	+	+	-	+
dtreeNodeFree	+	+	-	+
dtreeNodeAttach	-	+	-	-
dtreeNodeDetach	-	+	-	-
dtreePropFind	-	+	-	-
dtreePropFindNext	-	+	-	-
dtreePropLength	-	+	-	-

dtreePropValue	-	+	-	-
dtreePropName	-	+	-	-
dtreePropAlloc	+	+	-	+
dtreePropFree	+	+	-	+
dtreePropAttach	-	+	-	-
dtreePropDetach	-	+	-	-
dtreeNodeAdd	-	+	-	+
dtreeNodeFind	-	+	-	-
dtreePropAdd	-	+	-	+
dtreePathLeng	+	+	_	_
dtreePathGet	+	+	_	_

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

dtreeNodeRoot, dtreeNodeChild, dtreeNodePeer, dtreeNodeParent, dtreeNodeAlloc, dtreeNodeFree, dtreeNodeAttach, dtreeNodeDetach, dtreePropFind, dtreePropFindNext, dtreePropLength, dtreePropValue, dtreePropName, dtreePropAlloc, dtreePropFree, dtreePropAttach, dtreePropDetach, dtreeNodeAdd, dtreeNodeFind, dtreePropAdd, dtreePathLeng, dtreePathGet – device tree operations

SYNOPSIS

```
#include <dki/dki.h>
DevNode dtreeNodeRoot(void);

DevNode dtreeNodeChild(DevNode node);

DevNode dtreeNodePeer(DevNode node);

DevNode dtreeNodeParent(DevNode node);

DevNode dtreeNodeAlloc(void);

void dtreeNodeFree(DevNode );

void dtreeNodeAttach(DevNode pnode, DevNode cnode);

void dtreeNodeDetach(DevNode node);

DevProperty dtreePropFind(DevNode node, char * name);

DevProperty dtreePropFindNext(DevProperty prop, char * name);

unsigned int dtreePropLength(DevProperty prop);

void * dtreePropValue(DevProperty prop);

char * dtreePropName(DevProperty prop);

DevProperty dtreePropAlloc(char * name, int length);

void dtreePropFree(DevProperty prop);

void dtreePropAttach(DevNode node, DevProperty prop);

void dtreePropDetach(DevProperty prop);

DevNode dtreeNodeAdd(DevNode parent, char * name);

DevNode dtreeNodeFind(DevNode parent, char * name);

DevProperty dtreePropAdd(DevNode node, char * name, void * value, unsigned int
length);

DevProperty dtreePathLeng(DevNode node, int length);

DevProperty dtreePathGet(DevNode node, char * buf, void * value);
```

FEATURES

DKI

DESCRIPTION

Provides device tree operations.

EXTENDED DESCRIPTION

The device tree is a data structure which provides a description of the hardware topology and device properties. The hardware topology is specified in terms of parent / child relationships. Device properties associated with each device node in the tree are device specific.

A device property is a *name* / *value* pair. The property *name* is a null terminated ASCII string. The property *value* is a sequence of bytes specified by the *length* / *address* pair.

Note that the property value format is property specific and has to be standardized between the given producer and its consumers. For instance, among all device node properties there are some properties related to the bus resources required / allocated for the device (for instance, interrupt lines, I / O registers, DMA channels).

These types of properties must be standardized in order to be understood by the bus driver as well as by any device drivers connected to the given bus. The device tree data structure may be built either statically or dynamically. In the static case, the device tree is populated by the `booter` . For instance, the `booter` may include a pre-defined sequence of device tree function calls. Another use of the `booter` is to build the device tree from a hardware description provided by firmware.

In the dynamic case, the device tree is populated at system initialization time using an enumeration / probing mechanism. The device tree is populated by propagating from parents to children.

Note that it is possible to combine both methods. In other words, an initial (non complete) device tree may be provided by the `booter` which will later be completed dynamically using an enumeration / probing mechanism. However it is implemented, the device tree structure can be modified (extended / truncated) dynamically at run time using hot-plug insertion / removal (for example, PCMCIA cards). The device tree API is described in detail below.

Device Tree Browsing

This section describes APIs related to device tree browsing. `DevNode` is an abstract type designating a device node object, and is opaque to the driver.

`dtreeNodeRoot` returns the root device node if the device tree is not empty, otherwise NULL is returned.

`dtreeNodeChild` returns the first child node from the list of children. NULL is returned when the list of children is empty.

`dtreeNodePeer` returns the next device node from the sibling list, if any, otherwise NULL is returned. The *node* argument specifies the current device node in the sibling list.

`dtreeNodeParent` returns the parent device node, if any, otherwise NULL is returned. The *node* argument specifies the child device node.

`dtreePathLeng` returns the pathname length of a device node.

`dtreePathGet` returns in *buf* the absolute pathname of a device noe. The trailing path, of the pathname, is the name of the node and is read in a node property. If this property does not exist, the trailing path of the returned pathname is set to '???'.

Device Tree Modification

This section describes APIs related to device tree topology modification.

`dtreeNodeAlloc` allocates a new device node object. A non zero DevNode cookie is returned in case of success, otherwise NULL is returned. The allocated node has neither parent nor child nodes. There are no properties attached to the newly allocated node.

`dtreeNodeFree` releases the memory allocated by the node object and all property objects attached to the node. The *node* argument specifies the node object being released.

`dtreeNodeAttach` adds the device node specified by the *cnode* argument in the child list of the parent node specified by the *pnode* argument.

`dtreeNodeDetach` detaches the node object specified by the *node* argument from its parent (if any). When *node* specifies the root node, the device tree is emptied.

Device Node Properties

This section describes APIs related to the device node properties.

`DevProperty` is an abstract type designating a device property object, and is opaque to the driver.

`dtreePropFind` searches the first property within the properties list of the device node specified by the *node* argument. If the *name* argument is not NULL, `dtreePropFind` returns the first property whose name matches the *name* string. If the *name* argument is NULL, `dtreePropFind` returns the first property from the list regardless of its name.

In case of success, a non zero `DevProperty` cookie is returned, otherwise NULL is returned. Once the first property is found, the `dtreePropFindNext` routine may be used in order to find a subsequent property in the list. `dtreePropFindNext` searches the next property within the properties list. The current position within the properties list is specified by the *prop* argument. If the *name* argument is not `NULL` , `dtreePropFindNext` returns the next property (with respect to the current position) whose name matches the *name* string. If the *name* argument is `NULL` , `dtreePropFindNext` returns the next property from the list (with respect to the current position) regardless of its name.

If successful, a non zero `DevProperty` cookie is returned, otherwise `NULL` is returned. The `dtreePropFind` / `dtreePropFindNext` pair are typically used to iterate through either all device node properties, or a subset of device node properties whose names match a given name. In a case where a user knows that there is only one property with a given name attached to the device node, a single `dtreePropFind` invocation is sufficient.

`dtreePropLength` returns the property value length (in bytes). The property object is specified by the *prop* argument.

`dtreePropValue` returns a pointer to the first byte of the property value. The property object is specified by the *prop* argument. A driver can read and write the property value directly using the returned pointer. Typically, the driver will cast the pointer to a well known type/structure in order to access the property value. Note that the property value must always be presented in the CPU endian format.

`dtreePropName` returns a pointer to an ASCII string which designates the property name. The property object is specified by the prop argument.

`dtreePropAlloc` allocates a new device property object. A non zero `DevProperty` cookie is returned if successful, otherwise `NULL` is returned. The *name* argument specifies the property name. The *length* argument specifies the length of the property value. The property value is undefined. The property object allocated is not attached to any device node. Once the property value is initialized, the property object can be attached to a node (that is, added to the node properties list) using the `dtreePropAttach` function.

`dtreePropFree` releases the memory allocated by the property object. The *prop* parameter specifies the property object being released. Note that the property object must not be attached to any device node.

`dtreePropAttach` adds the property object specified by the *prop* argument to the node properties list. The *node* argument specifies the device node to which the property will be attached.

`dtreePropDetach` detaches the property object from the device node to which it is attached (if any). The *prop* argument specifies the property object.

Note that the device tree structure must only be accessed in the DKI thread context. The DKI thread provides access synchronization to the device tree. In many cases, driver routines which typically examine or modify the device tree are already invoked by the DKI thread. For instance, the driver probing, initialization and shut-down code is executed by the DKI thread. In a case when a driver needs to access the device tree at run time, a driver routine which uses the device tree API must be explicitly called in the DKI thread context. This rule also applies to driver clients; a driver client must always switch to the DKI thread context when accessing the device tree structure.

Device Tree High-Level Services

This section describes high-level APIs built upon other basic services. This API implements certain useful services for building and searching the device tree, in order to avoid implementing this code in all device tree users.

`dtreeNodeAdd` allocates a new device node object, and adds it to the child list of the *parent* device node specified..

The *name* argument specifies the name of the new device node allocated.. This means that a `node` property is allocated and attached to the new node with the value specified by *name* . If successful, the newly allocated node object is returned, otherwise, a NULL pointer is returned.

`dtreeNodeFind` looks for a named node in the child list of a specified device node. The *parent* argument specifies the device node within which the child list should be searched. The *name* argument specifies the value which must match the *node* property value of the node being searched.. If successful, the matching node object is returned, otherwise a NULL pointer is returned.

`dtreePropAdd` allocates a new property, sets its value and attaches it to a given device node. The *node* argument specifies the node to which the new property must be attached. The *name* argument specifies the name of the new property to allocate. The *length* argument specifies the memory size in bytes to be allocated for the new property value. The *value* argument specifies the value to be set for the newly allocated property. If successful, the newly allocated property object is returned, otherwise, a NULL pointer is returned.

Allowed Calling Contexts

The following table specifies the contexts in which a caller may invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
dtreeNodeRoot	-	+	-	-
dtreeNodeChild	-	+	-	-
dtreeNodePeer	-	+	-	-
dtreeNodeParent	-	+	-	-
dtreeNodeAlloc	+	+	-	+
dtreeNodeFree	+	+	-	+
dtreeNodeAttach	-	+	-	-
dtreeNodeDetach	-	+	-	-
dtreePropFind	-	+	-	-
dtreePropFindNext	-	+	-	-
dtreePropLength	-	+	-	-

dtreePropValue	-	+	-	-
dtreePropName	-	+	-	-
dtreePropAlloc	+	+	-	+
dtreePropFree	+	+	-	+
dtreePropAttach	-	+	-	-
dtreePropDetach	-	+	-	-
dtreeNodeAdd	-	+	-	+
dtreeNodeFind	-	+	-	-
dtreePropAdd	-	+	-	+
dtreePathLeng	+	+	_	_
dtreePathGet	+	+	_	_

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

dtreeNodeRoot, dtreeNodeChild, dtreeNodePeer, dtreeNodeParent, dtreeNodeAlloc, dtreeNodeFree, dtreeNodeAttach, dtreeNodeDetach, dtreePropFind, dtreePropFindNext, dtreePropLength, dtreePropValue, dtreePropName, dtreePropAlloc, dtreePropFree, dtreePropAttach, dtreePropDetach, dtreeNodeAdd, dtreeNodeFind, dtreePropAdd, dtreePathLeng, dtreePathGet – device tree operations

SYNOPSIS

```
#include <dki/dki.h>
DevNode dtreeNodeRoot(void);

DevNode dtreeNodeChild(DevNode node);

DevNode dtreeNodePeer(DevNode node);

DevNode dtreeNodeParent(DevNode node);

DevNode dtreeNodeAlloc(void);

void dtreeNodeFree(DevNode );

void dtreeNodeAttach(DevNode pnode, DevNode cnode);

void dtreeNodeDetach(DevNode node);

DevProperty dtreePropFind(DevNode node, char * name);

DevProperty dtreePropFindNext(DevProperty prop, char * name);

unsigned int dtreePropLength(DevProperty prop);

void * dtreePropValue(DevProperty prop);

char * dtreePropName(DevProperty prop);

DevProperty dtreePropAlloc(char * name, int length);

void dtreePropFree(DevProperty prop);

void dtreePropAttach(DevNode node, DevProperty prop);

void dtreePropDetach(DevProperty prop);

DevNode dtreeNodeAdd(DevNode parent, char * name);

DevNode dtreeNodeFind(DevNode parent, char * name);

DevProperty dtreePropAdd(DevNode node, char * name, void * value, unsigned int
length);

DevProperty dtreePathLeng(DevNode node, int length);

DevProperty dtreePathGet(DevNode node, char * buf, void * value);
```

FEATURES

DKI

DESCRIPTION

Provides device tree operations.

EXTENDED DESCRIPTION

The device tree is a data structure which provides a description of the hardware topology and device properties. The hardware topology is specified in terms of parent/child relationships. Device properties associated with each device node in the tree are device specific.

A device property is a *name* / *value* pair. The property *name* is a null terminated ASCII string. The property *value* is a sequence of bytes specified by the *length* / *address* pair.

Note that the property value format is property specific and has to be standardized between the given producer and its consumers. For instance, among all device node properties there are some properties related to the bus resources required/allocated for the device (for instance, interrupt lines, I/O registers, DMA channels).

These types of properties must be standardized in order to be understood by the bus driver as well as by any device drivers connected to the given bus. The device tree data structure may be built either statically or dynamically. In the static case, the device tree is populated by the `booter` . For instance, the `booter` may include a pre-defined sequence of device tree function calls. Another use of the `booter` is to build the device tree from a hardware description provided by firmware.

In the dynamic case, the device tree is populated at system initialization time using an enumeration/probing mechanism. The device tree is populated by propagating from parents to children.

Note that it is possible to combine both methods. In other words, an initial (non complete) device tree may be provided by the `booter` which will later be completed dynamically using an enumeration/probing mechanism. However it is implemented, the device tree structure can be modified (extended/truncated) dynamically at run time using hot-plug insertion/removal (for example, PCMCIA cards). The device tree API is described in detail below.

Device Tree Browsing

This section describes APIs related to device tree browsing. `DevNode` is an abstract type designating a device node object, and is opaque to the driver.

`dtreeNodeRoot` returns the root device node if the device tree is not empty, otherwise NULL is returned.

`dtreeNodeChild` returns the first child node from the list of children. NULL is returned when the list of children is empty.

`dtreeNodePeer` returns the next device node from the sibling list, if any, otherwise NULL is returned. The *node* argument specifies the current device node in the sibling list.

`dtreeNodeParent` returns the parent device node, if any, otherwise NULL is returned. The *node* argument specifies the child device node.

`dtreePathLeng` returns the pathname length of a device node.

`dtreePathGet` returns in *buf* the absolute pathname of a device noe. The trailing path, of the pathname, is the name of the node and is read in a node property. If this property does not exist, the trailing path of the returned pathname is set to '???'.

Device Tree Modification

This section describes APIs related to device tree topology modification.

`dtreeNodeAlloc` allocates a new device node object. A non zero DevNode cookie is returned in case of success, otherwise NULL is returned. The allocated node has neither parent nor child nodes. There are no properties attached to the newly allocated node.

`dtreeNodeFree` releases the memory allocated by the node object and all property objects attached to the node. The *node* argument specifies the node object being released.

`dtreeNodeAttach` adds the device node specified by the *cnode* argument in the child list of the parent node specified by the *pnode* argument.

`dtreeNodeDetach` detaches the node object specified by the *node* argument from its parent (if any). When *node* specifies the root node, the device tree is emptied.

Device Node Properties

This section describes APIs related to the device node properties.

`DevProperty` is an abstract type designating a device property object, and is opaque to the driver.

`dtreePropFind` searches the first property within the properties list of the device node specified by the *node* argument. If the *name* argument is not NULL, `dtreePropFind` returns the first property whose name matches the *name* string. If the *name* argument is NULL, `dtreePropFind` returns the first property from the list regardless of its name.

In case of success, a non zero `DevProperty` cookie is returned, otherwise NULL is returned. Once the first property is found, the `dtreePropFindNext` routine may be used in order to find a subsequent property in the list. `dtreePropFindNext` searches the next property within the properties list. The current position within the properties list is specified by the *prop* argument. If the *name* argument is not `NULL` , `dtreePropFindNext` returns the next property (with respect to the current position) whose name matches the *name* string. If the *name* argument is `NULL` , `dtreePropFindNext` returns the next property from the list (with respect to the current position) regardless of its name.

If successful, a non zero `DevProperty` cookie is returned, otherwise `NULL` is returned. The `dtreePropFind` / `dtreePropFindNext` pair are typically used to iterate through either all device node properties, or a subset of device node properties whose names match a given name. In a case where a user knows that there is only one property with a given name attached to the device node, a single `dtreePropFind` invocation is sufficient.

`dtreePropLength` returns the property value length (in bytes). The property object is specified by the *prop* argument.

`dtreePropValue` returns a pointer to the first byte of the property value. The property object is specified by the *prop* argument. A driver can read and write the property value directly using the returned pointer. Typically, the driver will cast the pointer to a well known type/structure in order to access the property value. Note that the property value must always be presented in the CPU endian format.

`dtreePropName` returns a pointer to an ASCII string which designates the property name. The property object is specified by the prop argument.

`dtreePropAlloc` allocates a new device property object. A non zero `DevProperty` cookie is returned if successful, otherwise `NULL` is returned. The *name* argument specifies the property name. The *length* argument specifies the length of the property value. The property value is undefined. The property object allocated is not attached to any device node. Once the property value is initialized, the property object can be attached to a node (that is, added to the node properties list) using the `dtreePropAttach` function.

`dtreePropFree` releases the memory allocated by the property object. The *prop* parameter specifies the property object being released. Note that the property object must not be attached to any device node.

`dtreePropAttach` adds the property object specified by the *prop* argument to the node properties list. The *node* argument specifies the device node to which the property will be attached.

`dtreePropDetach` detaches the property object from the device node to which it is attached (if any). The *prop* argument specifies the property object.

Note that the device tree structure must only be accessed in the DKI thread context. The DKI thread provides access synchronization to the device tree. In many cases, driver routines which typically examine or modify the device tree are already invoked by the DKI thread. For instance, the driver probing, initialization and shut-down code is executed by the DKI thread. In a case when a driver needs to access the device tree at run time, a driver routine which uses the device tree API must be explicitly called in the DKI thread context. This rule also applies to driver clients; a driver client must always switch to the DKI thread context when accessing the device tree structure.

Device Tree High-Level Services

This section describes high-level APIs built upon other basic services. This API implements certain useful services for building and searching the device tree, in order to avoid implementing this code in all device tree users.

`dtreeNodeAdd` allocates a new device node object, and adds it to the child list of the *parent* device node specified..

The *name* argument specifies the name of the new device node allocated.. This means that a `node` property is allocated and attached to the new node with the value specified by *name* . If successful, the newly allocated node object is returned, otherwise, a NULL pointer is returned.

`dtreeNodeFind` looks for a named node in the child list of a specified device node. The *parent* argument specifies the device node within which the child list should be searched. The *name* argument specifies the value which must match the *node* property value of the node being searched.. If successful, the matching node object is returned, otherwise a NULL pointer is returned.

`dtreePropAdd` allocates a new property, sets its value and attaches it to a given device node. The *node* argument specifies the node to which the new property must be attached. The *name* argument specifies the name of the new property to allocate. The *length* argument specifies the memory size in bytes to be allocated for the new property value. The *value* argument specifies the value to be set for the newly allocated property. If successful, the newly allocated property object is returned, otherwise, a NULL pointer is returned.

Allowed Calling Contexts

The following table specifies the contexts in which a caller may invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
dtreeNodeRoot	-	+	-	-
dtreeNodeChild	-	+	-	-
dtreeNodePeer	-	+	-	-
dtreeNodeParent	-	+	-	-
dtreeNodeAlloc	+	+	-	+
dtreeNodeFree	+	+	-	+
dtreeNodeAttach	-	+	-	-
dtreeNodeDetach	-	+	-	-
dtreePropFind	-	+	-	-
dtreePropFindNext	-	+	-	-
dtreePropLength	-	+	-	-

dtreePropValue	-	+	-	-
dtreePropName	-	+	-	-
dtreePropAlloc	+	+	-	+
dtreePropFree	+	+	-	+
dtreePropAttach	-	+	-	-
dtreePropDetach	-	+	-	-
dtreeNodeAdd	-	+	-	+
dtreeNodeFind	-	+	-	-
dtreePropAdd	-	+	-	+
dtreePathLeng	+	+	_	_
dtreePathGet	+	+	_	_

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

dtreeNodeRoot, dtreeNodeChild, dtreeNodePeer, dtreeNodeParent, dtreeNodeAlloc, dtreeNodeFree, dtreeNodeAttach, dtreeNodeDetach, dtreePropFind, dtreePropFindNext, dtreePropLength, dtreePropValue, dtreePropName, dtreePropAlloc, dtreePropFree, dtreePropAttach, dtreePropDetach, dtreeNodeAdd, dtreeNodeFind, dtreePropAdd, dtreePathLeng, dtreePathGet – device tree operations

SYNOPSIS

#include <dki/dki.h>

DevNode **dtreeNodeRoot**(void);

DevNode **dtreeNodeChild**(DevNode *node*);

DevNode **dtreeNodePeer**(DevNode *node*);

DevNode **dtreeNodeParent**(DevNode *node*);

DevNode **dtreeNodeAlloc**(void);

void **dtreeNodeFree**(DevNode);

void **dtreeNodeAttach**(DevNode *pnode*, DevNode *cnode*);

void **dtreeNodeDetach**(DevNode *node*);

DevProperty **dtreePropFind**(DevNode *node*, char * *name*);

DevProperty **dtreePropFindNext**(DevProperty *prop*, char * *name*);

unsigned int **dtreePropLength**(DevProperty *prop*);

void * **dtreePropValue**(DevProperty *prop*);

char * **dtreePropName**(DevProperty *prop*);

DevProperty **dtreePropAlloc**(char * *name*, int *length*);

void **dtreePropFree**(DevProperty *prop*);

void **dtreePropAttach**(DevNode *node*, DevProperty *prop*);

void **dtreePropDetach**(DevProperty *prop*);

DevNode **dtreeNodeAdd**(DevNode *parent*, char * *name*);

DevNode **dtreeNodeFind**(DevNode *parent*, char * *name*);

DevProperty **dtreePropAdd**(DevNode *node*, char * *name*, void * *value*, unsigned int *length*);

DevProperty **dtreePathLeng**(DevNode *node*, int *length*);

DevProperty **dtreePathGet**(DevNode *node*, char * *buf*, void * *value*);

FEATURES

DKI

DESCRIPTION

Provides device tree operations.

EXTENDED DESCRIPTION

The device tree is a data structure which provides a description of the hardware topology and device properties. The hardware topology is specified in terms of parent / child relationships. Device properties associated with each device node in the tree are device specific.

A device property is a *name* / *value* pair. The property *name* is a null terminated ASCII string. The property *value* is a sequence of bytes specified by the *length* / *address* pair.

Note that the property value format is property specific and has to be standardized between the given producer and its consumers. For instance, among all device node properties there are some properties related to the bus resources required / allocated for the device (for instance, interrupt lines, I / O registers, DMA channels).

These types of properties must be standardized in order to be understood by the bus driver as well as by any device drivers connected to the given bus. The device tree data structure may be built either statically or dynamically. In the static case, the device tree is populated by the `booter` . For instance, the `booter` may include a pre-defined sequence of device tree function calls. Another use of the `booter` is to build the device tree from a hardware description provided by firmware.

In the dynamic case, the device tree is populated at system initialization time using an enumeration / probing mechanism. The device tree is populated by propagating from parents to children.

Note that it is possible to combine both methods. In other words, an initial (non complete) device tree may be provided by the `booter` which will later be completed dynamically using an enumeration / probing mechanism. However it is implemented, the device tree structure can be modified (extended / truncated) dynamically at run time using hot-plug insertion / removal (for example, PCMCIA cards). The device tree API is described in detail below.

Device Tree Browsing

This section describes APIs related to device tree browsing. `DevNode` is an abstract type designating a device node object, and is opaque to the driver.

`dtreeNodeRoot` returns the root device node if the device tree is not empty, otherwise NULL is returned.

`dtreeNodeChild` returns the first child node from the list of children. NULL is returned when the list of children is empty.

`dtreeNodePeer` returns the next device node from the sibling list, if any, otherwise NULL is returned. The *node* argument specifies the current device node in the sibling list.

`dtreeNodeParent` returns the parent device node, if any, otherwise NULL is returned. The *node* argument specifies the child device node.

`dtreePathLeng` returns the pathname length of a device node.

`dtreePathGet` returns in *buf* the absolute pathname of a device noe. The trailing path, of the pathname, is the name of the node and is read in a node property. If this property does not exist, the trailing path of the returned pathname is set to '???'.

Device Tree Modification

This section describes APIs related to device tree topology modification.

`dtreeNodeAlloc` allocates a new device node object. A non zero DevNode cookie is returned in case of success, otherwise NULL is returned. The allocated node has neither parent nor child nodes. There are no properties attached to the newly allocated node.

`dtreeNodeFree` releases the memory allocated by the node object and all property objects attached to the node. The *node* argument specifies the node object being released.

`dtreeNodeAttach` adds the device node specified by the *cnode* argument in the child list of the parent node specified by the *pnode* argument.

`dtreeNodeDetach` detaches the node object specified by the *node* argument from its parent (if any). When *node* specifies the root node, the device tree is emptied.

Device Node Properties

This section describes APIs related to the device node properties.

`DevProperty` is an abstract type designating a device property object, and is opaque to the driver.

`dtreePropFind` searches the first property within the properties list of the device node specified by the *node* argument. If the *name* argument is not NULL, `dtreePropFind` returns the first property whose name matches the *name* string. If the *name* argument is NULL, `dtreePropFind` returns the first property from the list regardless of its name.

In case of success, a non zero `DevProperty` cookie is returned, otherwise NULL is returned. Once the first property is found, the `dtreePropFindNext` routine may be used in order to find a subsequent property in the list. `dtreePropFindNext` searches the next property within the properties list. The current position within the properties list is specified by the *prop* argument. If the *name* argument is not `NULL` , `dtreePropFindNext` returns the next property (with respect to the current position) whose name matches the *name* string. If the *name* argument is `NULL` , `dtreePropFindNext` returns the next property from the list (with respect to the current position) regardless of its name.

If successful, a non zero `DevProperty` cookie is returned, otherwise `NULL` is returned. The `dtreePropFind` / `dtreePropFindNext` pair are typically used to iterate through either all device node properties, or a subset of device node properties whose names match a given name. In a case where a user knows that there is only one property with a given name attached to the device node, a single `dtreePropFind` invocation is sufficient.

`dtreePropLength` returns the property value length (in bytes). The property object is specified by the *prop* argument.

`dtreePropValue` returns a pointer to the first byte of the property value. The property object is specified by the *prop* argument. A driver can read and write the property value directly using the returned pointer. Typically, the driver will cast the pointer to a well known type/structure in order to access the property value. Note that the property value must always be presented in the CPU endian format.

`dtreePropName` returns a pointer to an ASCII string which designates the property name. The property object is specified by the prop argument.

`dtreePropAlloc` allocates a new device property object. A non zero `DevProperty` cookie is returned if successful, otherwise `NULL` is returned. The *name* argument specifies the property name. The *length* argument specifies the length of the property value. The property value is undefined. The property object allocated is not attached to any device node. Once the property value is initialized, the property object can be attached to a node (that is, added to the node properties list) using the `dtreePropAttach` function.

`dtreePropFree` releases the memory allocated by the property object. The *prop* parameter specifies the property object being released. Note that the property object must not be attached to any device node.

`dtreePropAttach` adds the property object specified by the *prop* argument to the node properties list. The *node* argument specifies the device node to which the property will be attached.

`dtreePropDetach` detaches the property object from the device node to which it is attached (if any). The *prop* argument specifies the property object.

Note that the device tree structure must only be accessed in the DKI thread context. The DKI thread provides access synchronization to the device tree. In many cases, driver routines which typically examine or modify the device tree are already invoked by the DKI thread. For instance, the driver probing, initialization and shut-down code is executed by the DKI thread. In a case when a driver needs to access the device tree at run time, a driver routine which uses the device tree API must be explicitly called in the DKI thread context. This rule also applies to driver clients; a driver client must always switch to the DKI thread context when accessing the device tree structure.

Device Tree High-Level Services

This section describes high-level APIs built upon other basic services. This API implements certain useful services for building and searching the device tree, in order to avoid implementing this code in all device tree users.

`dtreeNodeAdd` allocates a new device node object, and adds it to the child list of the *parent* device node specified..

The *name* argument specifies the name of the new device node allocated.. This means that a `node` property is allocated and attached to the new node with the value specified by *name* . If successful, the newly allocated node object is returned, otherwise, a NULL pointer is returned.

`dtreeNodeFind` looks for a named node in the child list of a specified device node. The *parent* argument specifies the device node within which the child list should be searched. The *name* argument specifies the value which must match the *node* property value of the node being searched.. If successful, the matching node object is returned, otherwise a NULL pointer is returned.

`dtreePropAdd` allocates a new property, sets its value and attaches it to a given device node. The *node* argument specifies the node to which the new property must be attached. The *name* argument specifies the name of the new property to allocate. The *length* argument specifies the memory size in bytes to be allocated for the new property value. The *value* argument specifies the value to be set for the newly allocated property. If successful, the newly allocated property object is returned, otherwise, a NULL pointer is returned.

Allowed Calling Contexts

The following table specifies the contexts in which a caller may invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
dtreeNodeRoot	-	+	-	-
dtreeNodeChild	-	+	-	-
dtreeNodePeer	-	+	-	-
dtreeNodeParent	-	+	-	-
dtreeNodeAlloc	+	+	-	+
dtreeNodeFree	+	+	-	+
dtreeNodeAttach	-	+	-	-
dtreeNodeDetach	-	+	-	-
dtreePropFind	-	+	-	-
dtreePropFindNext	-	+	-	-
dtreePropLength	-	+	-	-

dtreePropValue	-	+	-	-
dtreePropName	-	+	-	-
dtreePropAlloc	+	+	-	+
dtreePropFree	+	+	-	+
dtreePropAttach	-	+	-	-
dtreePropDetach	-	+	-	-
dtreeNodeAdd	-	+	-	+
dtreeNodeFind	-	+	-	-
dtreePropAdd	-	+	-	+
dtreePathLeng	+	+	_	_
dtreePathGet	+	+	_	_

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

dtreeNodeRoot, dtreeNodeChild, dtreeNodePeer, dtreeNodeParent, dtreeNodeAlloc, dtreeNodeFree, dtreeNodeAttach, dtreeNodeDetach, dtreePropFind, dtreePropFindNext, dtreePropLength, dtreePropValue, dtreePropName, dtreePropAlloc, dtreePropFree, dtreePropAttach, dtreePropDetach, dtreeNodeAdd, dtreeNodeFind, dtreePropAdd, dtreePathLeng, dtreePathGet – device tree operations

SYNOPSIS

```
#include <dki/dki.h>
DevNode dtreeNodeRoot(void);

DevNode dtreeNodeChild(DevNode node);

DevNode dtreeNodePeer(DevNode node);

DevNode dtreeNodeParent(DevNode node);

DevNode dtreeNodeAlloc(void);

void dtreeNodeFree(DevNode );

void dtreeNodeAttach(DevNode pnode, DevNode cnode);

void dtreeNodeDetach(DevNode node);

DevProperty dtreePropFind(DevNode node, char * name);

DevProperty dtreePropFindNext(DevProperty prop, char * name);

unsigned int dtreePropLength(DevProperty prop);

void * dtreePropValue(DevProperty prop);

char * dtreePropName(DevProperty prop);

DevProperty dtreePropAlloc(char * name, int length);

void dtreePropFree(DevProperty prop);

void dtreePropAttach(DevNode node, DevProperty prop);

void dtreePropDetach(DevProperty prop);

DevNode dtreeNodeAdd(DevNode parent, char * name);

DevNode dtreeNodeFind(DevNode parent, char * name);

DevProperty dtreePropAdd(DevNode node, char * name, void * value, unsigned int
length);

DevProperty dtreePathLeng(DevNode node, int length);

DevProperty dtreePathGet(DevNode node, char * buf, void * value);
```

FEATURES

DKI

DESCRIPTION

Provides device tree operations.

EXTENDED DESCRIPTION

The device tree is a data structure which provides a description of the hardware topology and device properties. The hardware topology is specified in terms of parent / child relationships. Device properties associated with each device node in the tree are device specific.

A device property is a *name* / *value* pair. The property *name* is a null terminated ASCII string. The property *value* is a sequence of bytes specified by the *length* / *address* pair.

Note that the property value format is property specific and has to be standardized between the given producer and its consumers. For instance, among all device node properties there are some properties related to the bus resources required / allocated for the device (for instance, interrupt lines, I / O registers, DMA channels).

These types of properties must be standardized in order to be understood by the bus driver as well as by any device drivers connected to the given bus. The device tree data structure may be built either statically or dynamically. In the static case, the device tree is populated by the `booter` . For instance, the `booter` may include a pre-defined sequence of device tree function calls. Another use of the `booter` is to build the device tree from a hardware description provided by firmware.

In the dynamic case, the device tree is populated at system initialization time using an enumeration / probing mechanism. The device tree is populated by propagating from parents to children.

Note that it is possible to combine both methods. In other words, an initial (non complete) device tree may be provided by the `booter` which will later be completed dynamically using an enumeration / probing mechanism. However it is implemented, the device tree structure can be modified (extended / truncated) dynamically at run time using hot-plug insertion / removal (for example, PCMCIA cards). The device tree API is described in detail below.

Device Tree Browsing

This section describes APIs related to device tree browsing. `DevNode` is an abstract type designating a device node object, and is opaque to the driver.

`dtreeNodeRoot` returns the root device node if the device tree is not empty, otherwise NULL is returned.

`dtreeNodeChild` returns the first child node from the list of children. NULL is returned when the list of children is empty.

`dtreeNodePeer` returns the next device node from the sibling list, if any, otherwise NULL is returned. The *node* argument specifies the current device node in the sibling list.

`dtreeNodeParent` returns the parent device node, if any, otherwise NULL is returned. The *node* argument specifies the child device node.

`dtreePathLeng` returns the pathname length of a device node.

`dtreePathGet` returns in *buf* the absolute pathname of a device noe. The trailing path, of the pathname, is the name of the node and is read in a node property. If this property does not exist, the trailing path of the returned pathname is set to '???'.

Device Tree Modification

This section describes APIs related to device tree topology modification.

`dtreeNodeAlloc` allocates a new device node object. A non zero DevNode cookie is returned in case of success, otherwise NULL is returned. The allocated node has neither parent nor child nodes. There are no properties attached to the newly allocated node.

`dtreeNodeFree` releases the memory allocated by the node object and all property objects attached to the node. The *node* argument specifies the node object being released.

`dtreeNodeAttach` adds the device node specified by the *cnode* argument in the child list of the parent node specified by the *pnode* argument.

`dtreeNodeDetach` detaches the node object specified by the *node* argument from its parent (if any). When *node* specifies the root node, the device tree is emptied.

Device Node Properties

This section describes APIs related to the device node properties.

`DevProperty` is an abstract type designating a device property object, and is opaque to the driver.

`dtreePropFind` searches the first property within the properties list of the device node specified by the *node* argument. If the *name* argument is not NULL, `dtreePropFind` returns the first property whose name matches the *name* string. If the *name* argument is NULL, `dtreePropFind` returns the first property from the list regardless of its name.

In case of success, a non zero `DevProperty` cookie is returned, otherwise NULL is returned. Once the first property is found, the `dtreePropFindNext` routine may be used in order to find a subsequent property in the list. `dtreePropFindNext` searches the next property within the properties list. The current position within the properties list is specified by the *prop* argument. If the *name* argument is not `NULL` , `dtreePropFindNext` returns the next property (with respect to the current position) whose name matches the *name* string. If the *name* argument is `NULL` , `dtreePropFindNext` returns the next property from the list (with respect to the current position) regardless of its name.

If successful, a non zero `DevProperty` cookie is returned, otherwise `NULL` is returned. The `dtreePropFind` / `dtreePropFindNext` pair are typically used to iterate through either all device node properties, or a subset of device node properties whose names match a given name. In a case where a user knows that there is only one property with a given name attached to the device node, a single `dtreePropFind` invocation is sufficient.

`dtreePropLength` returns the property value length (in bytes). The property object is specified by the *prop* argument.

`dtreePropValue` returns a pointer to the first byte of the property value. The property object is specified by the *prop* argument. A driver can read and write the property value directly using the returned pointer. Typically, the driver will cast the pointer to a well known type/structure in order to access the property value. Note that the property value must always be presented in the CPU endian format.

`dtreePropName` returns a pointer to an ASCII string which designates the property name. The property object is specified by the prop argument.

`dtreePropAlloc` allocates a new device property object. A non zero `DevProperty` cookie is returned if successful, otherwise `NULL` is returned. The *name* argument specifies the property name. The *length* argument specifies the length of the property value. The property value is undefined. The property object allocated is not attached to any device node. Once the property value is initialized, the property object can be attached to a node (that is, added to the node properties list) using the `dtreePropAttach` function.

`dtreePropFree` releases the memory allocated by the property object. The *prop* parameter specifies the property object being released. Note that the property object must not be attached to any device node.

`dtreePropAttach` adds the property object specified by the *prop* argument to the node properties list. The *node* argument specifies the device node to which the property will be attached.

`dtreePropDetach` detaches the property object from the device node to which it is attached (if any). The *prop* argument specifies the property object.

Note that the device tree structure must only be accessed in the DKI thread context. The DKI thread provides access synchronization to the device tree. In many cases, driver routines which typically examine or modify the device tree are already invoked by the DKI thread. For instance, the driver probing, initialization and shut-down code is executed by the DKI thread. In a case when a driver needs to access the device tree at run time, a driver routine which uses the device tree API must be explicitly called in the DKI thread context. This rule also applies to driver clients; a driver client must always switch to the DKI thread context when accessing the device tree structure.

Device Tree High-Level Services

This section describes high-level APIs built upon other basic services. This API implements certain useful services for building and searching the device tree, in order to avoid implementing this code in all device tree users.

`dtreeNodeAdd` allocates a new device node object, and adds it to the child list of the *parent* device node specified..

The *name* argument specifies the name of the new device node allocated.. This means that a `node` property is allocated and attached to the new node with the value specified by *name* . If successful, the newly allocated node object is returned, otherwise, a NULL pointer is returned.

`dtreeNodeFind` looks for a named node in the child list of a specified device node. The *parent* argument specifies the device node within which the child list should be searched. The *name* argument specifies the value which must match the *node* property value of the node being searched.. If successful, the matching node object is returned, otherwise a NULL pointer is returned.

`dtreePropAdd` allocates a new property, sets its value and attaches it to a given device node. The *node* argument specifies the node to which the new property must be attached. The *name* argument specifies the name of the new property to allocate. The *length* argument specifies the memory size in bytes to be allocated for the new property value. The *value* argument specifies the value to be set for the newly allocated property. If successful, the newly allocated property object is returned, otherwise, a NULL pointer is returned.

Allowed Calling Contexts

The following table specifies the contexts in which a caller may invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
dtreeNodeRoot	-	+	-	-
dtreeNodeChild	-	+	-	-
dtreeNodePeer	-	+	-	-
dtreeNodeParent	-	+	-	-
dtreeNodeAlloc	+	+	-	+
dtreeNodeFree	+	+	-	+
dtreeNodeAttach	-	+	-	-
dtreeNodeDetach	-	+	-	-
dtreePropFind	-	+	-	-
dtreePropFindNext	-	+	-	-
dtreePropLength	-	+	-	-

dtreePropValue	-	+	-	-
dtreePropName	-	+	-	-
dtreePropAlloc	+	+	-	+
dtreePropFree	+	+	-	+
dtreePropAttach	-	+	-	-
dtreePropDetach	-	+	-	-
dtreeNodeAdd	-	+	-	+
dtreeNodeFind	-	+	-	-
dtreePropAdd	-	+	-	+
dtreePathLeng	+	+	_	_
dtreePathGet	+	+	_	_

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

dtreeNodeRoot, dtreeNodeChild, dtreeNodePeer, dtreeNodeParent, dtreeNodeAlloc, dtreeNodeFree, dtreeNodeAttach, dtreeNodeDetach, dtreePropFind, dtreePropFindNext, dtreePropLength, dtreePropValue, dtreePropName, dtreePropAlloc, dtreePropFree, dtreePropAttach, dtreePropDetach, dtreeNodeAdd, dtreeNodeFind, dtreePropAdd, dtreePathLeng, dtreePathGet – device tree operations

SYNOPSIS

```
#include <dki/dki.h>
DevNode dtreeNodeRoot(void);

DevNode dtreeNodeChild(DevNode node);

DevNode dtreeNodePeer(DevNode node);

DevNode dtreeNodeParent(DevNode node);

DevNode dtreeNodeAlloc(void);

void dtreeNodeFree(DevNode );

void dtreeNodeAttach(DevNode pnode, DevNode cnode);

void dtreeNodeDetach(DevNode node);

DevProperty dtreePropFind(DevNode node, char * name);

DevProperty dtreePropFindNext(DevProperty prop, char * name);

unsigned int dtreePropLength(DevProperty prop);

void * dtreePropValue(DevProperty prop);

char * dtreePropName(DevProperty prop);

DevProperty dtreePropAlloc(char * name, int length);

void dtreePropFree(DevProperty prop);

void dtreePropAttach(DevNode node, DevProperty prop);

void dtreePropDetach(DevProperty prop);

DevNode dtreeNodeAdd(DevNode parent, char * name);

DevNode dtreeNodeFind(DevNode parent, char * name);

DevProperty dtreePropAdd(DevNode node, char * name, void * value, unsigned int
length);

DevProperty dtreePathLeng(DevNode node, int length);

DevProperty dtreePathGet(DevNode node, char * buf, void * value);
```

FEATURES

DKI

DESCRIPTION

Provides device tree operations.

EXTENDED DESCRIPTION

The device tree is a data structure which provides a description of the hardware topology and device properties. The hardware topology is specified in terms of parent/child relationships. Device properties associated with each device node in the tree are device specific.

A device property is a *name* / *value* pair. The property *name* is a null terminated ASCII string. The property *value* is a sequence of bytes specified by the *length* / *address* pair.

Note that the property value format is property specific and has to be standardized between the given producer and its consumers. For instance, among all device node properties there are some properties related to the bus resources required/allocated for the device (for instance, interrupt lines, I/O registers, DMA channels).

These types of properties must be standardized in order to be understood by the bus driver as well as by any device drivers connected to the given bus. The device tree data structure may be built either statically or dynamically. In the static case, the device tree is populated by the `booter` . For instance, the `booter` may include a pre-defined sequence of device tree function calls. Another use of the `booter` is to build the device tree from a hardware description provided by firmware.

In the dynamic case, the device tree is populated at system initialization time using an enumeration/probing mechanism. The device tree is populated by propagating from parents to children.

Note that it is possible to combine both methods. In other words, an initial (non complete) device tree may be provided by the `booter` which will later be completed dynamically using an enumeration/probing mechanism. However it is implemented, the device tree structure can be modified (extended/truncated) dynamically at run time using hot-plug insertion/removal (for example, PCMCIA cards). The device tree API is described in detail below.

Device Tree Browsing

This section describes APIs related to device tree browsing. `DevNode` is an abstract type designating a device node object, and is opaque to the driver.

`dtreeNodeRoot` returns the root device node if the device tree is not empty, otherwise NULL is returned.

`dtreeNodeChild` returns the first child node from the list of children. NULL is returned when the list of children is empty.

`dtreeNodePeer` returns the next device node from the sibling list, if any, otherwise NULL is returned. The *node* argument specifies the current device node in the sibling list.

`dtreeNodeParent` returns the parent device node, if any, otherwise NULL is returned. The *node* argument specifies the child device node.

`dtreePathLeng` returns the pathname length of a device node.

`dtreePathGet` returns in *buf* the absolute pathname of a device noe. The trailing path, of the pathname, is the name of the node and is read in a node property. If this property does not exist, the trailing path of the returned pathname is set to '???'.

Device Tree Modification

This section describes APIs related to device tree topology modification.

`dtreeNodeAlloc` allocates a new device node object. A non zero DevNode cookie is returned in case of success, otherwise NULL is returned. The allocated node has neither parent nor child nodes. There are no properties attached to the newly allocated node.

`dtreeNodeFree` releases the memory allocated by the node object and all property objects attached to the node. The *node* argument specifies the node object being released.

`dtreeNodeAttach` adds the device node specified by the *cnode* argument in the child list of the parent node specified by the *pnode* argument.

`dtreeNodeDetach` detaches the node object specified by the *node* argument from its parent (if any). When *node* specifies the root node, the device tree is emptied.

Device Node Properties

This section describes APIs related to the device node properties.

`DevProperty` is an abstract type designating a device property object, and is opaque to the driver.

`dtreePropFind` searches the first property within the properties list of the device node specified by the *node* argument. If the *name* argument is not NULL, `dtreePropFind` returns the first property whose name matches the *name* string. If the *name* argument is NULL, `dtreePropFind` returns the first property from the list regardless of its name.

In case of success, a non zero `DevProperty` cookie is returned, otherwise NULL is returned. Once the first property is found, the `dtreePropFindNext` routine may be used in order to find a subsequent property in the list. `dtreePropFindNext` searches the next property within the properties list. The current position within the properties list is specified by the *prop* argument. If the *name* argument is not `NULL` , `dtreePropFindNext` returns the next property (with respect to the current position) whose name matches the *name* string. If the *name* argument is `NULL` , `dtreePropFindNext` returns the next property from the list (with respect to the current position) regardless of its name.

If successful, a non zero `DevProperty` cookie is returned, otherwise `NULL` is returned. The `dtreePropFind` / `dtreePropFindNext` pair are typically used to iterate through either all device node properties, or a subset of device node properties whose names match a given name. In a case where a user knows that there is only one property with a given name attached to the device node, a single `dtreePropFind` invocation is sufficient.

`dtreePropLength` returns the property value length (in bytes). The property object is specified by the *prop* argument.

`dtreePropValue` returns a pointer to the first byte of the property value. The property object is specified by the *prop* argument. A driver can read and write the property value directly using the returned pointer. Typically, the driver will cast the pointer to a well known type/structure in order to access the property value. Note that the property value must always be presented in the CPU endian format.

`dtreePropName` returns a pointer to an ASCII string which designates the property name. The property object is specified by the prop argument.

`dtreePropAlloc` allocates a new device property object. A non zero `DevProperty` cookie is returned if successful, otherwise `NULL` is returned. The *name* argument specifies the property name. The *length* argument specifies the length of the property value. The property value is undefined. The property object allocated is not attached to any device node. Once the property value is initialized, the property object can be attached to a node (that is, added to the node properties list) using the `dtreePropAttach` function.

`dtreePropFree` releases the memory allocated by the property object. The *prop* parameter specifies the property object being released. Note that the property object must not be attached to any device node.

`dtreePropAttach` adds the property object specified by the *prop* argument to the node properties list. The *node* argument specifies the device node to which the property will be attached.

`dtreePropDetach` detaches the property object from the device node to which it is attached (if any). The *prop* argument specifies the property object.

Note that the device tree structure must only be accessed in the DKI thread context. The DKI thread provides access synchronization to the device tree. In many cases, driver routines which typically examine or modify the device tree are already invoked by the DKI thread. For instance, the driver probing, initialization and shut-down code is executed by the DKI thread. In a case when a driver needs to access the device tree at run time, a driver routine which uses the device tree API must be explicitly called in the DKI thread context. This rule also applies to driver clients; a driver client must always switch to the DKI thread context when accessing the device tree structure.

Device Tree High-Level Services

This section describes high-level APIs built upon other basic services. This API implements certain useful services for building and searching the device tree, in order to avoid implementing this code in all device tree users.

`dtreeNodeAdd` allocates a new device node object, and adds it to the child list of the *parent* device node specified..

The *name* argument specifies the name of the new device node allocated.. This means that a `node` property is allocated and attached to the new node with the value specified by *name* . If successful, the newly allocated node object is returned, otherwise, a NULL pointer is returned.

`dtreeNodeFind` looks for a named node in the child list of a specified device node. The *parent* argument specifies the device node within which the child list should be searched. The *name* argument specifies the value which must match the *node* property value of the node being searched.. If successful, the matching node object is returned, otherwise a NULL pointer is returned.

`dtreePropAdd` allocates a new property, sets its value and attaches it to a given device node. The *node* argument specifies the node to which the new property must be attached. The *name* argument specifies the name of the new property to allocate. The *length* argument specifies the memory size in bytes to be allocated for the new property value. The *value* argument specifies the value to be set for the newly allocated property. If successful, the newly allocated property object is returned, otherwise, a NULL pointer is returned.

Allowed Calling Contexts

The following table specifies the contexts in which a caller may invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
dtreeNodeRoot	-	+	-	-
dtreeNodeChild	-	+	-	-
dtreeNodePeer	-	+	-	-
dtreeNodeParent	-	+	-	-
dtreeNodeAlloc	+	+	-	+
dtreeNodeFree	+	+	-	+
dtreeNodeAttach	-	+	-	-
dtreeNodeDetach	-	+	-	-
dtreePropFind	-	+	-	-
dtreePropFindNext	-	+	-	-
dtreePropLength	-	+	-	-

dtreePropValue	-	+	-	-
dtreePropName	-	+	-	-
dtreePropAlloc	+	+	-	+
dtreePropFree	+	+	-	+
dtreePropAttach	-	+	-	-
dtreePropDetach	-	+	-	-
dtreeNodeAdd	-	+	-	+
dtreeNodeFind	-	+	-	-
dtreePropAdd	-	+	-	+
dtreePathLeng	+	+	_	_
dtreePathGet	+	+	_	_

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

dtreeNodeRoot, dtreeNodeChild, dtreeNodePeer, dtreeNodeParent, dtreeNodeAlloc, dtreeNodeFree, dtreeNodeAttach, dtreeNodeDetach, dtreePropFind, dtreePropFindNext, dtreePropLength, dtreePropValue, dtreePropName, dtreePropAlloc, dtreePropFree, dtreePropAttach, dtreePropDetach, dtreeNodeAdd, dtreeNodeFind, dtreePropAdd, dtreePathLeng, dtreePathGet – device tree operations

SYNOPSIS

```
#include <dki/dki.h>
DevNode dtreeNodeRoot(void);

DevNode dtreeNodeChild(DevNode node);

DevNode dtreeNodePeer(DevNode node);

DevNode dtreeNodeParent(DevNode node);

DevNode dtreeNodeAlloc(void);

void dtreeNodeFree(DevNode );

void dtreeNodeAttach(DevNode pnode, DevNode cnode);

void dtreeNodeDetach(DevNode node);

DevProperty dtreePropFind(DevNode node, char * name);

DevProperty dtreePropFindNext(DevProperty prop, char * name);

unsigned int dtreePropLength(DevProperty prop);

void * dtreePropValue(DevProperty prop);

char * dtreePropName(DevProperty prop);

DevProperty dtreePropAlloc(char * name, int length);

void dtreePropFree(DevProperty prop);

void dtreePropAttach(DevNode node, DevProperty prop);

void dtreePropDetach(DevProperty prop);

DevNode dtreeNodeAdd(DevNode parent, char * name);

DevNode dtreeNodeFind(DevNode parent, char * name);

DevProperty dtreePropAdd(DevNode node, char * name, void * value, unsigned int
length);

DevProperty dtreePathLeng(DevNode node, int length);

DevProperty dtreePathGet(DevNode node, char * buf, void * value);
```

FEATURES

DKI

DESCRIPTION

Provides device tree operations.

EXTENDED DESCRIPTION

The device tree is a data structure which provides a description of the hardware topology and device properties. The hardware topology is specified in terms of parent / child relationships. Device properties associated with each device node in the tree are device specific.

A device property is a *name* / *value* pair. The property *name* is a null terminated ASCII string. The property *value* is a sequence of bytes specified by the *length* / *address* pair.

Note that the property value format is property specific and has to be standardized between the given producer and its consumers. For instance, among all device node properties there are some properties related to the bus resources required / allocated for the device (for instance, interrupt lines, I / O registers, DMA channels).

These types of properties must be standardized in order to be understood by the bus driver as well as by any device drivers connected to the given bus. The device tree data structure may be built either statically or dynamically. In the static case, the device tree is populated by the `booter` . For instance, the `booter` may include a pre-defined sequence of device tree function calls. Another use of the `booter` is to build the device tree from a hardware description provided by firmware.

In the dynamic case, the device tree is populated at system initialization time using an enumeration / probing mechanism. The device tree is populated by propagating from parents to children.

Note that it is possible to combine both methods. In other words, an initial (non complete) device tree may be provided by the `booter` which will later be completed dynamically using an enumeration / probing mechanism. However it is implemented, the device tree structure can be modified (extended / truncated) dynamically at run time using hot-plug insertion / removal (for example, PCMCIA cards). The device tree API is described in detail below.

Device Tree Browsing

This section describes APIs related to device tree browsing. `DevNode` is an abstract type designating a device node object, and is opaque to the driver.

`dtreeNodeRoot` returns the root device node if the device tree is not empty, otherwise NULL is returned.

`dtreeNodeChild` returns the first child node from the list of children. NULL is returned when the list of children is empty.

`dtreeNodePeer` returns the next device node from the sibling list, if any, otherwise NULL is returned. The *node* argument specifies the current device node in the sibling list.

`dtreeNodeParent` returns the parent device node, if any, otherwise NULL is returned. The *node* argument specifies the child device node.

`dtreePathLeng` returns the pathname length of a device node.

`dtreePathGet` returns in *buf* the absolute pathname of a device noe. The trailing path, of the pathname, is the name of the node and is read in a node property. If this property does not exist, the trailing path of the returned pathname is set to '???'.

Device Tree Modification

This section describes APIs related to device tree topology modification.

`dtreeNodeAlloc` allocates a new device node object. A non zero DevNode cookie is returned in case of success, otherwise NULL is returned. The allocated node has neither parent nor child nodes. There are no properties attached to the newly allocated node.

`dtreeNodeFree` releases the memory allocated by the node object and all property objects attached to the node. The *node* argument specifies the node object being released.

`dtreeNodeAttach` adds the device node specified by the *cnode* argument in the child list of the parent node specified by the *pnode* argument.

`dtreeNodeDetach` detaches the node object specified by the *node* argument from its parent (if any). When *node* specifies the root node, the device tree is emptied.

Device Node Properties

This section describes APIs related to the device node properties.

`DevProperty` is an abstract type designating a device property object, and is opaque to the driver.

`dtreePropFind` searches the first property within the properties list of the device node specified by the *node* argument. If the *name* argument is not NULL, `dtreePropFind` returns the first property whose name matches the *name* string. If the *name* argument is NULL, `dtreePropFind` returns the first property from the list regardless of its name.

In case of success, a non zero `DevProperty` cookie is returned, otherwise NULL is returned. Once the first property is found, the `dtreePropFindNext` routine may be used in order to find a subsequent property in the list. `dtreePropFindNext` searches the next property within the properties list. The current position within the properties list is specified by the *prop* argument. If the *name* argument is not `NULL` , `dtreePropFindNext` returns the next property (with respect to the current position) whose name matches the *name* string. If the *name* argument is `NULL` , `dtreePropFindNext` returns the next property from the list (with respect to the current position) regardless of its name.

If successful, a non zero `DevProperty` cookie is returned, otherwise `NULL` is returned. The `dtreePropFind` / `dtreePropFindNext` pair are typically used to iterate through either all device node properties, or a subset of device node properties whose names match a given name. In a case where a user knows that there is only one property with a given name attached to the device node, a single `dtreePropFind` invocation is sufficient.

`dtreePropLength` returns the property value length (in bytes). The property object is specified by the *prop* argument.

`dtreePropValue` returns a pointer to the first byte of the property value. The property object is specified by the *prop* argument. A driver can read and write the property value directly using the returned pointer. Typically, the driver will cast the pointer to a well known type/structure in order to access the property value. Note that the property value must always be presented in the CPU endian format.

`dtreePropName` returns a pointer to an ASCII string which designates the property name. The property object is specified by the prop argument.

`dtreePropAlloc` allocates a new device property object. A non zero `DevProperty` cookie is returned if successful, otherwise `NULL` is returned. The *name* argument specifies the property name. The *length* argument specifies the length of the property value. The property value is undefined. The property object allocated is not attached to any device node. Once the property value is initialized, the property object can be attached to a node (that is, added to the node properties list) using the `dtreePropAttach` function.

`dtreePropFree` releases the memory allocated by the property object. The *prop* parameter specifies the property object being released. Note that the property object must not be attached to any device node.

`dtreePropAttach` adds the property object specified by the *prop* argument to the node properties list. The *node* argument specifies the device node to which the property will be attached.

`dtreePropDetach` detaches the property object from the device node to which it is attached (if any). The *prop* argument specifies the property object.

Note that the device tree structure must only be accessed in the DKI thread context. The DKI thread provides access synchronization to the device tree. In many cases, driver routines which typically examine or modify the device tree are already invoked by the DKI thread. For instance, the driver probing, initialization and shut-down code is executed by the DKI thread. In a case when a driver needs to access the device tree at run time, a driver routine which uses the device tree API must be explicitly called in the DKI thread context. This rule also applies to driver clients; a driver client must always switch to the DKI thread context when accessing the device tree structure.

Device Tree High-Level Services

This section describes high-level APIs built upon other basic services. This API implements certain useful services for building and searching the device tree, in order to avoid implementing this code in all device tree users.

`dtreeNodeAdd` allocates a new device node object, and adds it to the child list of the *parent* device node specified..

The *name* argument specifies the name of the new device node allocated.. This means that a `node` property is allocated and attached to the new node with the value specified by *name* . If successful, the newly allocated node object is returned, otherwise, a NULL pointer is returned.

`dtreeNodeFind` looks for a named node in the child list of a specified device node. The *parent* argument specifies the device node within which the child list should be searched. The *name* argument specifies the value which must match the *node* property value of the node being searched.. If successful, the matching node object is returned, otherwise a NULL pointer is returned.

`dtreePropAdd` allocates a new property, sets its value and attaches it to a given device node. The *node* argument specifies the node to which the new property must be attached. The *name* argument specifies the name of the new property to allocate. The *length* argument specifies the memory size in bytes to be allocated for the new property value. The *value* argument specifies the value to be set for the newly allocated property. If successful, the newly allocated property object is returned, otherwise, a NULL pointer is returned.

Allowed Calling Contexts

The following table specifies the contexts in which a caller may invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
dtreeNodeRoot	-	+	-	-
dtreeNodeChild	-	+	-	-
dtreeNodePeer	-	+	-	-
dtreeNodeParent	-	+	-	-
dtreeNodeAlloc	+	+	-	+
dtreeNodeFree	+	+	-	+
dtreeNodeAttach	-	+	-	-
dtreeNodeDetach	-	+	-	-
dtreePropFind	-	+	-	-
dtreePropFindNext	-	+	-	-
dtreePropLength	-	+	-	-

dtreePropValue	-	+	-	-
dtreePropName	-	+	-	-
dtreePropAlloc	+	+	-	+
dtreePropFree	+	+	-	+
dtreePropAttach	-	+	-	-
dtreePropDetach	-	+	-	-
dtreeNodeAdd	-	+	-	+
dtreeNodeFind	-	+	-	-
dtreePropAdd	-	+	-	+
dtreePathLeng	+	+	_	_
dtreePathGet	+	+	_	_

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

dtreeNodeRoot, dtreeNodeChild, dtreeNodePeer, dtreeNodeParent, dtreeNodeAlloc, dtreeNodeFree, dtreeNodeAttach, dtreeNodeDetach, dtreePropFind, dtreePropFindNext, dtreePropLength, dtreePropValue, dtreePropName, dtreePropAlloc, dtreePropFree, dtreePropAttach, dtreePropDetach, dtreeNodeAdd, dtreeNodeFind, dtreePropAdd, dtreePathLeng, dtreePathGet – device tree operations

SYNOPSIS

```
#include <dki/dki.h>
DevNode dtreeNodeRoot(void);

DevNode dtreeNodeChild(DevNode node);

DevNode dtreeNodePeer(DevNode node);

DevNode dtreeNodeParent(DevNode node);

DevNode dtreeNodeAlloc(void);

void dtreeNodeFree(DevNode );

void dtreeNodeAttach(DevNode pnode, DevNode cnode);

void dtreeNodeDetach(DevNode node);

DevProperty dtreePropFind(DevNode node, char * name);

DevProperty dtreePropFindNext(DevProperty prop, char * name);

unsigned int dtreePropLength(DevProperty prop);

void * dtreePropValue(DevProperty prop);

char * dtreePropName(DevProperty prop);

DevProperty dtreePropAlloc(char * name, int length);

void dtreePropFree(DevProperty prop);

void dtreePropAttach(DevNode node, DevProperty prop);

void dtreePropDetach(DevProperty prop);

DevNode dtreeNodeAdd(DevNode parent, char * name);

DevNode dtreeNodeFind(DevNode parent, char * name);

DevProperty dtreePropAdd(DevNode node, char * name, void * value, unsigned int
length);

DevProperty dtreePathLeng(DevNode node, int length);

DevProperty dtreePathGet(DevNode node, char * buf, void * value);
```

FEATURES

DKI

DESCRIPTION

Provides device tree operations.

EXTENDED DESCRIPTION

The device tree is a data structure which provides a description of the hardware topology and device properties. The hardware topology is specified in terms of parent/child relationships. Device properties associated with each device node in the tree are device specific.

A device property is a *name* / *value* pair. The property *name* is a null terminated ASCII string. The property *value* is a sequence of bytes specified by the *length* / *address* pair.

Note that the property value format is property specific and has to be standardized between the given producer and its consumers. For instance, among all device node properties there are some properties related to the bus resources required/allocated for the device (for instance, interrupt lines, I/O registers, DMA channels).

These types of properties must be standardized in order to be understood by the bus driver as well as by any device drivers connected to the given bus. The device tree data structure may be built either statically or dynamically. In the static case, the device tree is populated by the `booter` . For instance, the `booter` may include a pre-defined sequence of device tree function calls. Another use of the `booter` is to build the device tree from a hardware description provided by firmware.

In the dynamic case, the device tree is populated at system initialization time using an enumeration/probing mechanism. The device tree is populated by propagating from parents to children.

Note that it is possible to combine both methods. In other words, an initial (non complete) device tree may be provided by the `booter` which will later be completed dynamically using an enumeration/probing mechanism. However it is implemented, the device tree structure can be modified (extended/truncated) dynamically at run time using hot-plug insertion/removal (for example, PCMCIA cards). The device tree API is described in detail below.

Device Tree Browsing

This section describes APIs related to device tree browsing. `DevNode` is an abstract type designating a device node object, and is opaque to the driver.

`dtreeNodeRoot` returns the root device node if the device tree is not empty, otherwise NULL is returned.

`dtreeNodeChild` returns the first child node from the list of children. NULL is returned when the list of children is empty.

`dtreeNodePeer` returns the next device node from the sibling list, if any, otherwise NULL is returned. The *node* argument specifies the current device node in the sibling list.

`dtreeNodeParent` returns the parent device node, if any, otherwise NULL is returned. The *node* argument specifies the child device node.

`dtreePathLeng` returns the pathname length of a device node.

`dtreePathGet` returns in *buf* the absolute pathname of a device noe. The trailing path, of the pathname, is the name of the node and is read in a node property. If this property does not exist, the trailing path of the returned pathname is set to '???'.

Device Tree Modification

This section describes APIs related to device tree topology modification.

`dtreeNodeAlloc` allocates a new device node object. A non zero DevNode cookie is returned in case of success, otherwise NULL is returned. The allocated node has neither parent nor child nodes. There are no properties attached to the newly allocated node.

`dtreeNodeFree` releases the memory allocated by the node object and all property objects attached to the node. The *node* argument specifies the node object being released.

`dtreeNodeAttach` adds the device node specified by the *cnode* argument in the child list of the parent node specified by the *pnode* argument.

`dtreeNodeDetach` detaches the node object specified by the *node* argument from its parent (if any). When *node* specifies the root node, the device tree is emptied.

Device Node Properties

This section describes APIs related to the device node properties.

`DevProperty` is an abstract type designating a device property object, and is opaque to the driver.

`dtreePropFind` searches the first property within the properties list of the device node specified by the *node* argument. If the *name* argument is not NULL, `dtreePropFind` returns the first property whose name matches the *name* string. If the *name* argument is NULL, `dtreePropFind` returns the first property from the list regardless of its name.

In case of success, a non zero `DevProperty` cookie is returned, otherwise NULL is returned. Once the first property is found, the `dtreePropFindNext` routine may be used in order to find a subsequent property in the list. `dtreePropFindNext` searches the next property within the properties list. The current position within the properties list is specified by the *prop* argument. If the *name* argument is not `NULL` , `dtreePropFindNext` returns the next property (with respect to the current position) whose name matches the *name* string. If the *name* argument is `NULL` , `dtreePropFindNext` returns the next property from the list (with respect to the current position) regardless of its name.

If successful, a non zero `DevProperty` cookie is returned, otherwise `NULL` is returned. The `dtreePropFind` / `dtreePropFindNext` pair are typically used to iterate through either all device node properties, or a subset of device node properties whose names match a given name. In a case where a user knows that there is only one property with a given name attached to the device node, a single `dtreePropFind` invocation is sufficient.

`dtreePropLength` returns the property value length (in bytes). The property object is specified by the *prop* argument.

`dtreePropValue` returns a pointer to the first byte of the property value. The property object is specified by the *prop* argument. A driver can read and write the property value directly using the returned pointer. Typically, the driver will cast the pointer to a well known type/structure in order to access the property value. Note that the property value must always be presented in the CPU endian format.

`dtreePropName` returns a pointer to an ASCII string which designates the property name. The property object is specified by the prop argument.

`dtreePropAlloc` allocates a new device property object. A non zero `DevProperty` cookie is returned if successful, otherwise `NULL` is returned. The *name* argument specifies the property name. The *length* argument specifies the length of the property value. The property value is undefined. The property object allocated is not attached to any device node. Once the property value is initialized, the property object can be attached to a node (that is, added to the node properties list) using the `dtreePropAttach` function.

`dtreePropFree` releases the memory allocated by the property object. The *prop* parameter specifies the property object being released. Note that the property object must not be attached to any device node.

`dtreePropAttach` adds the property object specified by the *prop* argument to the node properties list. The *node* argument specifies the device node to which the property will be attached.

`dtreePropDetach` detaches the property object from the device node to which it is attached (if any). The *prop* argument specifies the property object.

Note that the device tree structure must only be accessed in the DKI thread context. The DKI thread provides access synchronization to the device tree. In many cases, driver routines which typically examine or modify the device tree are already invoked by the DKI thread. For instance, the driver probing, initialization and shut-down code is executed by the DKI thread. In a case when a driver needs to access the device tree at run time, a driver routine which uses the device tree API must be explicitly called in the DKI thread context. This rule also applies to driver clients; a driver client must always switch to the DKI thread context when accessing the device tree structure.

Device Tree High-Level Services

This section describes high-level APIs built upon other basic services. This API implements certain useful services for building and searching the device tree, in order to avoid implementing this code in all device tree users.

`dtreeNodeAdd` allocates a new device node object, and adds it to the child list of the *parent* device node specified..

The *name* argument specifies the name of the new device node allocated.. This means that a `node` property is allocated and attached to the new node with the value specified by *name* . If successful, the newly allocated node object is returned, otherwise, a NULL pointer is returned.

`dtreeNodeFind` looks for a named node in the child list of a specified device node. The *parent* argument specifies the device node within which the child list should be searched. The *name* argument specifies the value which must match the *node* property value of the node being searched.. If successful, the matching node object is returned, otherwise a NULL pointer is returned.

`dtreePropAdd` allocates a new property, sets its value and attaches it to a given device node. The *node* argument specifies the node to which the new property must be attached. The *name* argument specifies the name of the new property to allocate. The *length* argument specifies the memory size in bytes to be allocated for the new property value. The *value* argument specifies the value to be set for the newly allocated property. If successful, the newly allocated property object is returned, otherwise, a NULL pointer is returned.

Allowed Calling Contexts

The following table specifies the contexts in which a caller may invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
dtreeNodeRoot	-	+	-	-
dtreeNodeChild	-	+	-	-
dtreeNodePeer	-	+	-	-
dtreeNodeParent	-	+	-	-
dtreeNodeAlloc	+	+	-	+
dtreeNodeFree	+	+	-	+
dtreeNodeAttach	-	+	-	-
dtreeNodeDetach	-	+	-	-
dtreePropFind	-	+	-	-
dtreePropFindNext	-	+	-	-
dtreePropLength	-	+	-	-

dtreePropValue	-	+	-	-
dtreePropName	-	+	-	-
dtreePropAlloc	+	+	-	+
dtreePropFree	+	+	-	+
dtreePropAttach	-	+	-	-
dtreePropDetach	-	+	-	-
dtreeNodeAdd	-	+	-	+
dtreeNodeFind	-	+	-	-
dtreePropAdd	-	+	-	+
dtreePathLeng	+	+	_	_
dtreePathGet	+	+	_	_

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

dtreeNodeRoot, dtreeNodeChild, dtreeNodePeer, dtreeNodeParent, dtreeNodeAlloc, dtreeNodeFree, dtreeNodeAttach, dtreeNodeDetach, dtreePropFind, dtreePropFindNext, dtreePropLength, dtreePropValue, dtreePropName, dtreePropAlloc, dtreePropFree, dtreePropAttach, dtreePropDetach, dtreeNodeAdd, dtreeNodeFind, dtreePropAdd, dtreePathLeng, dtreePathGet – device tree operations

SYNOPSIS

```
#include <dki/dki.h>
DevNode dtreeNodeRoot(void);

DevNode dtreeNodeChild(DevNode node);

DevNode dtreeNodePeer(DevNode node);

DevNode dtreeNodeParent(DevNode node);

DevNode dtreeNodeAlloc(void);

void dtreeNodeFree(DevNode );

void dtreeNodeAttach(DevNode pnode, DevNode cnode);

void dtreeNodeDetach(DevNode node);

DevProperty dtreePropFind(DevNode node, char * name);

DevProperty dtreePropFindNext(DevProperty prop, char * name);

unsigned int dtreePropLength(DevProperty prop);

void * dtreePropValue(DevProperty prop);

char * dtreePropName(DevProperty prop);

DevProperty dtreePropAlloc(char * name, int length);

void dtreePropFree(DevProperty prop);

void dtreePropAttach(DevNode node, DevProperty prop);

void dtreePropDetach(DevProperty prop);

DevNode dtreeNodeAdd(DevNode parent, char * name);

DevNode dtreeNodeFind(DevNode parent, char * name);

DevProperty dtreePropAdd(DevNode node, char * name, void * value, unsigned int
length);

DevProperty dtreePathLeng(DevNode node, int length);

DevProperty dtreePathGet(DevNode node, char * buf, void * value);
```

FEATURES

DKI

DESCRIPTION

Provides device tree operations.

EXTENDED DESCRIPTION

The device tree is a data structure which provides a description of the hardware topology and device properties. The hardware topology is specified in terms of parent/child relationships. Device properties associated with each device node in the tree are device specific.

A device property is a *name* / *value* pair. The property *name* is a null terminated ASCII string. The property *value* is a sequence of bytes specified by the *length* / *address* pair.

Note that the property value format is property specific and has to be standardized between the given producer and its consumers. For instance, among all device node properties there are some properties related to the bus resources required/allocated for the device (for instance, interrupt lines, I/O registers, DMA channels).

These types of properties must be standardized in order to be understood by the bus driver as well as by any device drivers connected to the given bus. The device tree data structure may be built either statically or dynamically. In the static case, the device tree is populated by the `booter` . For instance, the `booter` may include a pre-defined sequence of device tree function calls. Another use of the `booter` is to build the device tree from a hardware description provided by firmware.

In the dynamic case, the device tree is populated at system initialization time using an enumeration/probing mechanism. The device tree is populated by propagating from parents to children.

Note that it is possible to combine both methods. In other words, an initial (non complete) device tree may be provided by the `booter` which will later be completed dynamically using an enumeration/probing mechanism. However it is implemented, the device tree structure can be modified (extended/truncated) dynamically at run time using hot-plug insertion/removal (for example, PCMCIA cards). The device tree API is described in detail below.

Device Tree Browsing

This section describes APIs related to device tree browsing. `DevNode` is an abstract type designating a device node object, and is opaque to the driver.

`dtreeNodeRoot` returns the root device node if the device tree is not empty, otherwise NULL is returned.

`dtreeNodeChild` returns the first child node from the list of children. NULL is returned when the list of children is empty.

`dtreeNodePeer` returns the next device node from the sibling list, if any, otherwise NULL is returned. The *node* argument specifies the current device node in the sibling list.

`dtreeNodeParent` returns the parent device node, if any, otherwise NULL is returned. The *node* argument specifies the child device node.

`dtreePathLeng` returns the pathname length of a device node.

`dtreePathGet` returns in *buf* the absolute pathname of a device noe. The trailing path, of the pathname, is the name of the node and is read in a node property. If this property does not exist, the trailing path of the returned pathname is set to '???'.

Device Tree Modification

This section describes APIs related to device tree topology modification.

`dtreeNodeAlloc` allocates a new device node object. A non zero DevNode cookie is returned in case of success, otherwise NULL is returned. The allocated node has neither parent nor child nodes. There are no properties attached to the newly allocated node.

`dtreeNodeFree` releases the memory allocated by the node object and all property objects attached to the node. The *node* argument specifies the node object being released.

`dtreeNodeAttach` adds the device node specified by the *cnode* argument in the child list of the parent node specified by the *pnode* argument.

`dtreeNodeDetach` detaches the node object specified by the *node* argument from its parent (if any). When *node* specifies the root node, the device tree is emptied.

Device Node Properties

This section describes APIs related to the device node properties.

`DevProperty` is an abstract type designating a device property object, and is opaque to the driver.

`dtreePropFind` searches the first property within the properties list of the device node specified by the *node* argument. If the *name* argument is not NULL, `dtreePropFind` returns the first property whose name matches the *name* string. If the *name* argument is NULL, `dtreePropFind` returns the first property from the list regardless of its name.

In case of success, a non zero `DevProperty` cookie is returned, otherwise NULL is returned. Once the first property is found, the `dtreePropFindNext` routine may be used in order to find a subsequent property in the list. `dtreePropFindNext` searches the next property within the properties list. The current position within the properties list is specified by the *prop* argument. If the *name* argument is not `NULL` , `dtreePropFindNext` returns the next property (with respect to the current position) whose name matches the *name* string. If the *name* argument is `NULL` , `dtreePropFindNext` returns the next property from the list (with respect to the current position) regardless of its name.

If successful, a non zero `DevProperty` cookie is returned, otherwise `NULL` is returned. The `dtreePropFind` / `dtreePropFindNext` pair are typically used to iterate through either all device node properties, or a subset of device node properties whose names match a given name. In a case where a user knows that there is only one property with a given name attached to the device node, a single `dtreePropFind` invocation is sufficient.

`dtreePropLength` returns the property value length (in bytes). The property object is specified by the *prop* argument.

`dtreePropValue` returns a pointer to the first byte of the property value. The property object is specified by the *prop* argument. A driver can read and write the property value directly using the returned pointer. Typically, the driver will cast the pointer to a well known type/structure in order to access the property value. Note that the property value must always be presented in the CPU endian format.

`dtreePropName` returns a pointer to an ASCII string which designates the property name. The property object is specified by the prop argument.

`dtreePropAlloc` allocates a new device property object. A non zero `DevProperty` cookie is returned if successful, otherwise `NULL` is returned. The *name* argument specifies the property name. The *length* argument specifies the length of the property value. The property value is undefined. The property object allocated is not attached to any device node. Once the property value is initialized, the property object can be attached to a node (that is, added to the node properties list) using the `dtreePropAttach` function.

`dtreePropFree` releases the memory allocated by the property object. The *prop* parameter specifies the property object being released. Note that the property object must not be attached to any device node.

`dtreePropAttach` adds the property object specified by the *prop* argument to the node properties list. The *node* argument specifies the device node to which the property will be attached.

`dtreePropDetach` detaches the property object from the device node to which it is attached (if any). The *prop* argument specifies the property object.

Note that the device tree structure must only be accessed in the DKI thread context. The DKI thread provides access synchronization to the device tree. In many cases, driver routines which typically examine or modify the device tree are already invoked by the DKI thread. For instance, the driver probing, initialization and shut-down code is executed by the DKI thread. In a case when a driver needs to access the device tree at run time, a driver routine which uses the device tree API must be explicitly called in the DKI thread context. This rule also applies to driver clients; a driver client must always switch to the DKI thread context when accessing the device tree structure.

Device Tree High-Level Services

This section describes high-level APIs built upon other basic services. This API implements certain useful services for building and searching the device tree, in order to avoid implementing this code in all device tree users.

`dtreeNodeAdd` allocates a new device node object, and adds it to the child list of the *parent* device node specified..

The *name* argument specifies the name of the new device node allocated.. This means that a `node` property is allocated and attached to the new node with the value specified by *name* . If successful, the newly allocated node object is returned, otherwise, a NULL pointer is returned.

`dtreeNodeFind` looks for a named node in the child list of a specified device node. The *parent* argument specifies the device node within which the child list should be searched. The *name* argument specifies the value which must match the *node* property value of the node being searched.. If successful, the matching node object is returned, otherwise a NULL pointer is returned.

`dtreePropAdd` allocates a new property, sets its value and attaches it to a given device node. The *node* argument specifies the node to which the new property must be attached. The *name* argument specifies the name of the new property to allocate. The *length* argument specifies the memory size in bytes to be allocated for the new property value. The *value* argument specifies the value to be set for the newly allocated property. If successful, the newly allocated property object is returned, otherwise, a NULL pointer is returned.

Allowed Calling Contexts

The following table specifies the contexts in which a caller may invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
dtreeNodeRoot	-	+	-	-
dtreeNodeChild	-	+	-	-
dtreeNodePeer	-	+	-	-
dtreeNodeParent	-	+	-	-
dtreeNodeAlloc	+	+	-	+
dtreeNodeFree	+	+	-	+
dtreeNodeAttach	-	+	-	-
dtreeNodeDetach	-	+	-	-
dtreePropFind	-	+	-	-
dtreePropFindNext	-	+	-	-
dtreePropLength	-	+	-	-

dtreePropValue	-	+	-	-
dtreePropName	-	+	-	-
dtreePropAlloc	+	+	-	+
dtreePropFree	+	+	-	+
dtreePropAttach	-	+	-	-
dtreePropDetach	-	+	-	-
dtreeNodeAdd	-	+	-	+
dtreeNodeFind	-	+	-	-
dtreePropAdd	-	+	-	+
dtreePathLeng	+	+	_	_
dtreePathGet	+	+	_	_

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

dtreeNodeRoot, dtreeNodeChild, dtreeNodePeer, dtreeNodeParent, dtreeNodeAlloc, dtreeNodeFree, dtreeNodeAttach, dtreeNodeDetach, dtreePropFind, dtreePropFindNext, dtreePropLength, dtreePropValue, dtreePropName, dtreePropAlloc, dtreePropFree, dtreePropAttach, dtreePropDetach, dtreeNodeAdd, dtreeNodeFind, dtreePropAdd, dtreePathLeng, dtreePathGet – device tree operations

SYNOPSIS

```
#include <dki/dki.h>
DevNode dtreeNodeRoot(void);

DevNode dtreeNodeChild(DevNode node);

DevNode dtreeNodePeer(DevNode node);

DevNode dtreeNodeParent(DevNode node);

DevNode dtreeNodeAlloc(void);

void dtreeNodeFree(DevNode );

void dtreeNodeAttach(DevNode pnode, DevNode cnode);

void dtreeNodeDetach(DevNode node);

DevProperty dtreePropFind(DevNode node, char * name);

DevProperty dtreePropFindNext(DevProperty prop, char * name);

unsigned int dtreePropLength(DevProperty prop);

void * dtreePropValue(DevProperty prop);

char * dtreePropName(DevProperty prop);

DevProperty dtreePropAlloc(char * name, int length);

void dtreePropFree(DevProperty prop);

void dtreePropAttach(DevNode node, DevProperty prop);

void dtreePropDetach(DevProperty prop);

DevNode dtreeNodeAdd(DevNode parent, char * name);

DevNode dtreeNodeFind(DevNode parent, char * name);

DevProperty dtreePropAdd(DevNode node, char * name, void * value, unsigned int
length);

DevProperty dtreePathLeng(DevNode node, int length);

DevProperty dtreePathGet(DevNode node, char * buf, void * value);
```

FEATURES

DKI

DESCRIPTION

Provides device tree operations.

EXTENDED DESCRIPTION

The device tree is a data structure which provides a description of the hardware topology and device properties. The hardware topology is specified in terms of parent/child relationships. Device properties associated with each device node in the tree are device specific.

A device property is a *name* / *value* pair. The property *name* is a null terminated ASCII string. The property *value* is a sequence of bytes specified by the *length* / *address* pair.

Note that the property value format is property specific and has to be standardized between the given producer and its consumers. For instance, among all device node properties there are some properties related to the bus resources required/allocated for the device (for instance, interrupt lines, I/O registers, DMA channels).

These types of properties must be standardized in order to be understood by the bus driver as well as by any device drivers connected to the given bus. The device tree data structure may be built either statically or dynamically. In the static case, the device tree is populated by the `booter` . For instance, the `booter` may include a pre-defined sequence of device tree function calls. Another use of the `booter` is to build the device tree from a hardware description provided by firmware.

In the dynamic case, the device tree is populated at system initialization time using an enumeration/probing mechanism. The device tree is populated by propagating from parents to children.

Note that it is possible to combine both methods. In other words, an initial (non complete) device tree may be provided by the `booter` which will later be completed dynamically using an enumeration/probing mechanism. However it is implemented, the device tree structure can be modified (extended/truncated) dynamically at run time using hot-plug insertion/removal (for example, PCMCIA cards). The device tree API is described in detail below.

Device Tree Browsing

This section describes APIs related to device tree browsing. `DevNode` is an abstract type designating a device node object, and is opaque to the driver.

`dtreeNodeRoot` returns the root device node if the device tree is not empty, otherwise NULL is returned.

`dtreeNodeChild` returns the first child node from the list of children. NULL is returned when the list of children is empty.

`dtreeNodePeer` returns the next device node from the sibling list, if any, otherwise NULL is returned. The *node* argument specifies the current device node in the sibling list.

`dtreeNodeParent` returns the parent device node, if any, otherwise NULL is returned. The *node* argument specifies the child device node.

`dtreePathLeng` returns the pathname length of a device node.

`dtreePathGet` returns in *buf* the absolute pathname of a device noe. The trailing path, of the pathname, is the name of the node and is read in a node property. If this property does not exist, the trailing path of the returned pathname is set to '???'.

Device Tree Modification

This section describes APIs related to device tree topology modification.

`dtreeNodeAlloc` allocates a new device node object. A non zero DevNode cookie is returned in case of success, otherwise NULL is returned. The allocated node has neither parent nor child nodes. There are no properties attached to the newly allocated node.

`dtreeNodeFree` releases the memory allocated by the node object and all property objects attached to the node. The *node* argument specifies the node object being released.

`dtreeNodeAttach` adds the device node specified by the *cnode* argument in the child list of the parent node specified by the *pnode* argument.

`dtreeNodeDetach` detaches the node object specified by the *node* argument from its parent (if any). When *node* specifies the root node, the device tree is emptied.

Device Node Properties

This section describes APIs related to the device node properties.

`DevProperty` is an abstract type designating a device property object, and is opaque to the driver.

`dtreePropFind` searches the first property within the properties list of the device node specified by the *node* argument. If the *name* argument is not NULL, `dtreePropFind` returns the first property whose name matches the *name* string. If the *name* argument is NULL, `dtreePropFind` returns the first property from the list regardless of its name.

In case of success, a non zero `DevProperty` cookie is returned, otherwise NULL is returned. Once the first property is found, the `dtreePropFindNext` routine may be used in order to find a subsequent property in the list. `dtreePropFindNext` searches the next property within the properties list. The current position within the properties list is specified by the *prop* argument. If the *name* argument is not `NULL` , `dtreePropFindNext` returns the next property (with respect to the current position) whose name matches the *name* string. If the *name* argument is `NULL` , `dtreePropFindNext` returns the next property from the list (with respect to the current position) regardless of its name.

If successful, a non zero `DevProperty` cookie is returned, otherwise `NULL` is returned. The `dtreePropFind` / `dtreePropFindNext` pair are typically used to iterate through either all device node properties, or a subset of device node properties whose names match a given name. In a case where a user knows that there is only one property with a given name attached to the device node, a single `dtreePropFind` invocation is sufficient.

`dtreePropLength` returns the property value length (in bytes). The property object is specified by the *prop* argument.

`dtreePropValue` returns a pointer to the first byte of the property value. The property object is specified by the *prop* argument. A driver can read and write the property value directly using the returned pointer. Typically, the driver will cast the pointer to a well known type/structure in order to access the property value. Note that the property value must always be presented in the CPU endian format.

`dtreePropName` returns a pointer to an ASCII string which designates the property name. The property object is specified by the prop argument.

`dtreePropAlloc` allocates a new device property object. A non zero `DevProperty` cookie is returned if successful, otherwise `NULL` is returned. The *name* argument specifies the property name. The *length* argument specifies the length of the property value. The property value is undefined. The property object allocated is not attached to any device node. Once the property value is initialized, the property object can be attached to a node (that is, added to the node properties list) using the `dtreePropAttach` function.

`dtreePropFree` releases the memory allocated by the property object. The *prop* parameter specifies the property object being released. Note that the property object must not be attached to any device node.

`dtreePropAttach` adds the property object specified by the *prop* argument to the node properties list. The *node* argument specifies the device node to which the property will be attached.

`dtreePropDetach` detaches the property object from the device node to which it is attached (if any). The *prop* argument specifies the property object.

Note that the device tree structure must only be accessed in the DKI thread context. The DKI thread provides access synchronization to the device tree. In many cases, driver routines which typically examine or modify the device tree are already invoked by the DKI thread. For instance, the driver probing, initialization and shut-down code is executed by the DKI thread. In a case when a driver needs to access the device tree at run time, a driver routine which uses the device tree API must be explicitly called in the DKI thread context. This rule also applies to driver clients; a driver client must always switch to the DKI thread context when accessing the device tree structure.

Device Tree High-Level Services

This section describes high-level APIs built upon other basic services. This API implements certain useful services for building and searching the device tree, in order to avoid implementing this code in all device tree users.

`dtreeNodeAdd` allocates a new device node object, and adds it to the child list of the *parent* device node specified..

The *name* argument specifies the name of the new device node allocated.. This means that a `node` property is allocated and attached to the new node with the value specified by *name* . If successful, the newly allocated node object is returned, otherwise, a NULL pointer is returned.

`dtreeNodeFind` looks for a named node in the child list of a specified device node. The *parent* argument specifies the device node within which the child list should be searched. The *name* argument specifies the value which must match the *node* property value of the node being searched.. If successful, the matching node object is returned, otherwise a NULL pointer is returned.

`dtreePropAdd` allocates a new property, sets its value and attaches it to a given device node. The *node* argument specifies the node to which the new property must be attached. The *name* argument specifies the name of the new property to allocate. The *length* argument specifies the memory size in bytes to be allocated for the new property value. The *value* argument specifies the value to be set for the newly allocated property. If successful, the newly allocated property object is returned, otherwise, a NULL pointer is returned.

Allowed Calling Contexts

The following table specifies the contexts in which a caller may invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
dtreeNodeRoot	-	+	-	-
dtreeNodeChild	-	+	-	-
dtreeNodePeer	-	+	-	-
dtreeNodeParent	-	+	-	-
dtreeNodeAlloc	+	+	-	+
dtreeNodeFree	+	+	-	+
dtreeNodeAttach	-	+	-	-
dtreeNodeDetach	-	+	-	-
dtreePropFind	-	+	-	-
dtreePropFindNext	-	+	-	-
dtreePropLength	-	+	-	-

dtreePropValue	-	+	-	-
dtreePropName	-	+	-	-
dtreePropAlloc	+	+	-	+
dtreePropFree	+	+	-	+
dtreePropAttach	-	+	-	-
dtreePropDetach	-	+	-	-
dtreeNodeAdd	-	+	-	+
dtreeNodeFind	-	+	-	-
dtreePropAdd	-	+	-	+
dtreePathLeng	+	+	_	_
dtreePathGet	+	+	_	_

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

dtreeNodeRoot, dtreeNodeChild, dtreeNodePeer, dtreeNodeParent, dtreeNodeAlloc, dtreeNodeFree, dtreeNodeAttach, dtreeNodeDetach, dtreePropFind, dtreePropFindNext, dtreePropLength, dtreePropValue, dtreePropName, dtreePropAlloc, dtreePropFree, dtreePropAttach, dtreePropDetach, dtreeNodeAdd, dtreeNodeFind, dtreePropAdd, dtreePathLeng, dtreePathGet – device tree operations

SYNOPSIS

```
#include <dki/dki.h>
DevNode dtreeNodeRoot(void);

DevNode dtreeNodeChild(DevNode node);

DevNode dtreeNodePeer(DevNode node);

DevNode dtreeNodeParent(DevNode node);

DevNode dtreeNodeAlloc(void);

void dtreeNodeFree(DevNode );

void dtreeNodeAttach(DevNode pnode, DevNode cnode);

void dtreeNodeDetach(DevNode node);

DevProperty dtreePropFind(DevNode node, char * name);

DevProperty dtreePropFindNext(DevProperty prop, char * name);

unsigned int dtreePropLength(DevProperty prop);

void * dtreePropValue(DevProperty prop);

char * dtreePropName(DevProperty prop);

DevProperty dtreePropAlloc(char * name, int length);

void dtreePropFree(DevProperty prop);

void dtreePropAttach(DevNode node, DevProperty prop);

void dtreePropDetach(DevProperty prop);

DevNode dtreeNodeAdd(DevNode parent, char * name);

DevNode dtreeNodeFind(DevNode parent, char * name);

DevProperty dtreePropAdd(DevNode node, char * name, void * value, unsigned int
length);

DevProperty dtreePathLeng(DevNode node, int length);

DevProperty dtreePathGet(DevNode node, char * buf, void * value);
```

FEATURES

DKI

DESCRIPTION

Provides device tree operations.

EXTENDED DESCRIPTION

The device tree is a data structure which provides a description of the hardware topology and device properties. The hardware topology is specified in terms of parent/child relationships. Device properties associated with each device node in the tree are device specific.

A device property is a *name* / *value* pair. The property *name* is a null terminated ASCII string. The property *value* is a sequence of bytes specified by the *length* / *address* pair.

Note that the property value format is property specific and has to be standardized between the given producer and its consumers. For instance, among all device node properties there are some properties related to the bus resources required/allocated for the device (for instance, interrupt lines, I/O registers, DMA channels).

These types of properties must be standardized in order to be understood by the bus driver as well as by any device drivers connected to the given bus. The device tree data structure may be built either statically or dynamically. In the static case, the device tree is populated by the `booter` . For instance, the `booter` may include a pre-defined sequence of device tree function calls. Another use of the `booter` is to build the device tree from a hardware description provided by firmware.

In the dynamic case, the device tree is populated at system initialization time using an enumeration/probing mechanism. The device tree is populated by propagating from parents to children.

Note that it is possible to combine both methods. In other words, an initial (non complete) device tree may be provided by the `booter` which will later be completed dynamically using an enumeration/probing mechanism. However it is implemented, the device tree structure can be modified (extended/truncated) dynamically at run time using hot-plug insertion/removal (for example, PCMCIA cards). The device tree API is described in detail below.

Device Tree Browsing

This section describes APIs related to device tree browsing. `DevNode` is an abstract type designating a device node object, and is opaque to the driver.

`dtreeNodeRoot` returns the root device node if the device tree is not empty, otherwise NULL is returned.

`dtreeNodeChild` returns the first child node from the list of children. NULL is returned when the list of children is empty.

`dtreeNodePeer` returns the next device node from the sibling list, if any, otherwise NULL is returned. The *node* argument specifies the current device node in the sibling list.

`dtreeNodeParent` returns the parent device node, if any, otherwise NULL is returned. The *node* argument specifies the child device node.

`dtreePathLeng` returns the pathname length of a device node.

`dtreePathGet` returns in *buf* the absolute pathname of a device noe. The trailing path, of the pathname, is the name of the node and is read in a node property. If this property does not exist, the trailing path of the returned pathname is set to '???'.

Device Tree Modification

This section describes APIs related to device tree topology modification.

`dtreeNodeAlloc` allocates a new device node object. A non zero DevNode cookie is returned in case of success, otherwise NULL is returned. The allocated node has neither parent nor child nodes. There are no properties attached to the newly allocated node.

`dtreeNodeFree` releases the memory allocated by the node object and all property objects attached to the node. The *node* argument specifies the node object being released.

`dtreeNodeAttach` adds the device node specified by the *cnode* argument in the child list of the parent node specified by the *pnode* argument.

`dtreeNodeDetach` detaches the node object specified by the *node* argument from its parent (if any). When *node* specifies the root node, the device tree is emptied.

Device Node Properties

This section describes APIs related to the device node properties.

`DevProperty` is an abstract type designating a device property object, and is opaque to the driver.

`dtreePropFind` searches the first property within the properties list of the device node specified by the *node* argument. If the *name* argument is not NULL, `dtreePropFind` returns the first property whose name matches the *name* string. If the *name* argument is NULL, `dtreePropFind` returns the first property from the list regardless of its name.

In case of success, a non zero `DevProperty` cookie is returned, otherwise NULL is returned. Once the first property is found, the `dtreePropFindNext` routine may be used in order to find a subsequent property in the list. `dtreePropFindNext` searches the next property within the properties list. The current position within the properties list is specified by the *prop* argument. If the *name* argument is not `NULL` , `dtreePropFindNext` returns the next property (with respect to the current position) whose name matches the *name* string. If the *name* argument is `NULL` , `dtreePropFindNext` returns the next property from the list (with respect to the current position) regardless of its name.

If successful, a non zero `DevProperty` cookie is returned, otherwise `NULL` is returned. The `dtreePropFind` / `dtreePropFindNext` pair are typically used to iterate through either all device node properties, or a subset of device node properties whose names match a given name. In a case where a user knows that there is only one property with a given name attached to the device node, a single `dtreePropFind` invocation is sufficient.

`dtreePropLength` returns the property value length (in bytes). The property object is specified by the *prop* argument.

`dtreePropValue` returns a pointer to the first byte of the property value. The property object is specified by the *prop* argument. A driver can read and write the property value directly using the returned pointer. Typically, the driver will cast the pointer to a well known type/structure in order to access the property value. Note that the property value must always be presented in the CPU endian format.

`dtreePropName` returns a pointer to an ASCII string which designates the property name. The property object is specified by the prop argument.

`dtreePropAlloc` allocates a new device property object. A non zero `DevProperty` cookie is returned if successful, otherwise `NULL` is returned. The *name* argument specifies the property name. The *length* argument specifies the length of the property value. The property value is undefined. The property object allocated is not attached to any device node. Once the property value is initialized, the property object can be attached to a node (that is, added to the node properties list) using the `dtreePropAttach` function.

`dtreePropFree` releases the memory allocated by the property object. The *prop* parameter specifies the property object being released. Note that the property object must not be attached to any device node.

`dtreePropAttach` adds the property object specified by the *prop* argument to the node properties list. The *node* argument specifies the device node to which the property will be attached.

`dtreePropDetach` detaches the property object from the device node to which it is attached (if any). The *prop* argument specifies the property object.

Note that the device tree structure must only be accessed in the DKI thread context. The DKI thread provides access synchronization to the device tree. In many cases, driver routines which typically examine or modify the device tree are already invoked by the DKI thread. For instance, the driver probing, initialization and shut-down code is executed by the DKI thread. In a case when a driver needs to access the device tree at run time, a driver routine which uses the device tree API must be explicitly called in the DKI thread context. This rule also applies to driver clients; a driver client must always switch to the DKI thread context when accessing the device tree structure.

Device Tree High-Level Services

This section describes high-level APIs built upon other basic services. This API implements certain useful services for building and searching the device tree, in order to avoid implementing this code in all device tree users.

`dtreeNodeAdd` allocates a new device node object, and adds it to the child list of the *parent* device node specified..

The *name* argument specifies the name of the new device node allocated.. This means that a `node` property is allocated and attached to the new node with the value specified by *name* . If successful, the newly allocated node object is returned, otherwise, a NULL pointer is returned.

`dtreeNodeFind` looks for a named node in the child list of a specified device node. The *parent* argument specifies the device node within which the child list should be searched. The *name* argument specifies the value which must match the *node* property value of the node being searched.. If successful, the matching node object is returned, otherwise a NULL pointer is returned.

`dtreePropAdd` allocates a new property, sets its value and attaches it to a given device node. The *node* argument specifies the node to which the new property must be attached. The *name* argument specifies the name of the new property to allocate. The *length* argument specifies the memory size in bytes to be allocated for the new property value. The *value* argument specifies the value to be set for the newly allocated property. If successful, the newly allocated property object is returned, otherwise, a NULL pointer is returned.

Allowed Calling Contexts

The following table specifies the contexts in which a caller may invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
dtreeNodeRoot	-	+	-	-
dtreeNodeChild	-	+	-	-
dtreeNodePeer	-	+	-	-
dtreeNodeParent	-	+	-	-
dtreeNodeAlloc	+	+	-	+
dtreeNodeFree	+	+	-	+
dtreeNodeAttach	-	+	-	-
dtreeNodeDetach	-	+	-	-
dtreePropFind	-	+	-	-
dtreePropFindNext	-	+	-	-
dtreePropLength	-	+	-	-

dtreePropValue	-	+	-	-
dtreePropName	-	+	-	-
dtreePropAlloc	+	+	-	+
dtreePropFree	+	+	-	+
dtreePropAttach	-	+	-	-
dtreePropDetach	-	+	-	-
dtreeNodeAdd	-	+	-	+
dtreeNodeFind	-	+	-	-
dtreePropAdd	-	+	-	+
dtreePathLeng	+	+	_	_
dtreePathGet	+	+	_	_

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

dtreeNodeRoot, dtreeNodeChild, dtreeNodePeer, dtreeNodeParent, dtreeNodeAlloc, dtreeNodeFree, dtreeNodeAttach, dtreeNodeDetach, dtreePropFind, dtreePropFindNext, dtreePropLength, dtreePropValue, dtreePropName, dtreePropAlloc, dtreePropFree, dtreePropAttach, dtreePropDetach, dtreeNodeAdd, dtreeNodeFind, dtreePropAdd, dtreePathLeng, dtreePathGet – device tree operations

SYNOPSIS

```
#include <dki/dki.h>
DevNode dtreeNodeRoot(void);

DevNode dtreeNodeChild(DevNode node);

DevNode dtreeNodePeer(DevNode node);

DevNode dtreeNodeParent(DevNode node);

DevNode dtreeNodeAlloc(void);

void dtreeNodeFree(DevNode );

void dtreeNodeAttach(DevNode pnode, DevNode cnode);

void dtreeNodeDetach(DevNode node);

DevProperty dtreePropFind(DevNode node, char * name);

DevProperty dtreePropFindNext(DevProperty prop, char * name);

unsigned int dtreePropLength(DevProperty prop);

void * dtreePropValue(DevProperty prop);

char * dtreePropName(DevProperty prop);

DevProperty dtreePropAlloc(char * name, int length);

void dtreePropFree(DevProperty prop);

void dtreePropAttach(DevNode node, DevProperty prop);

void dtreePropDetach(DevProperty prop);

DevNode dtreeNodeAdd(DevNode parent, char * name);

DevNode dtreeNodeFind(DevNode parent, char * name);

DevProperty dtreePropAdd(DevNode node, char * name, void * value, unsigned int
length);

DevProperty dtreePathLeng(DevNode node, int length);

DevProperty dtreePathGet(DevNode node, char * buf, void * value);
```

FEATURES

DKI

DESCRIPTION

Provides device tree operations.

EXTENDED DESCRIPTION

The device tree is a data structure which provides a description of the hardware topology and device properties. The hardware topology is specified in terms of parent/child relationships. Device properties associated with each device node in the tree are device specific.

A device property is a *name* / *value* pair. The property *name* is a null terminated ASCII string. The property *value* is a sequence of bytes specified by the *length* / *address* pair.

Note that the property value format is property specific and has to be standardized between the given producer and its consumers. For instance, among all device node properties there are some properties related to the bus resources required/allocated for the device (for instance, interrupt lines, I/O registers, DMA channels).

These types of properties must be standardized in order to be understood by the bus driver as well as by any device drivers connected to the given bus. The device tree data structure may be built either statically or dynamically. In the static case, the device tree is populated by the `booter` . For instance, the `booter` may include a pre-defined sequence of device tree function calls. Another use of the `booter` is to build the device tree from a hardware description provided by firmware.

In the dynamic case, the device tree is populated at system initialization time using an enumeration/probing mechanism. The device tree is populated by propagating from parents to children.

Note that it is possible to combine both methods. In other words, an initial (non complete) device tree may be provided by the `booter` which will later be completed dynamically using an enumeration/probing mechanism. However it is implemented, the device tree structure can be modified (extended/truncated) dynamically at run time using hot-plug insertion/removal (for example, PCMCIA cards). The device tree API is described in detail below.

Device Tree Browsing

This section describes APIs related to device tree browsing. `DevNode` is an abstract type designating a device node object, and is opaque to the driver.

`dtreeNodeRoot` returns the root device node if the device tree is not empty, otherwise NULL is returned.

`dtreeNodeChild` returns the first child node from the list of children. NULL is returned when the list of children is empty.

`dtreeNodePeer` returns the next device node from the sibling list, if any, otherwise NULL is returned. The *node* argument specifies the current device node in the sibling list.

`dtreeNodeParent` returns the parent device node, if any, otherwise NULL is returned. The *node* argument specifies the child device node.

`dtreePathLeng` returns the pathname length of a device node.

`dtreePathGet` returns in *buf* the absolute pathname of a device noe. The trailing path, of the pathname, is the name of the node and is read in a node property. If this property does not exist, the trailing path of the returned pathname is set to '???'.

Device Tree Modification

This section describes APIs related to device tree topology modification.

`dtreeNodeAlloc` allocates a new device node object. A non zero DevNode cookie is returned in case of success, otherwise NULL is returned. The allocated node has neither parent nor child nodes. There are no properties attached to the newly allocated node.

`dtreeNodeFree` releases the memory allocated by the node object and all property objects attached to the node. The *node* argument specifies the node object being released.

`dtreeNodeAttach` adds the device node specified by the *cnode* argument in the child list of the parent node specified by the *pnode* argument.

`dtreeNodeDetach` detaches the node object specified by the *node* argument from its parent (if any). When *node* specifies the root node, the device tree is emptied.

Device Node Properties

This section describes APIs related to the device node properties.

`DevProperty` is an abstract type designating a device property object, and is opaque to the driver.

`dtreePropFind` searches the first property within the properties list of the device node specified by the *node* argument. If the *name* argument is not NULL, `dtreePropFind` returns the first property whose name matches the *name* string. If the *name* argument is NULL, `dtreePropFind` returns the first property from the list regardless of its name.

In case of success, a non zero `DevProperty` cookie is returned, otherwise NULL is returned. Once the first property is found, the `dtreePropFindNext` routine may be used in order to find a subsequent property in the list. `dtreePropFindNext` searches the next property within the properties list. The current position within the properties list is specified by the *prop* argument. If the *name* argument is not `NULL` , `dtreePropFindNext` returns the next property (with respect to the current position) whose name matches the *name* string. If the *name* argument is `NULL` , `dtreePropFindNext` returns the next property from the list (with respect to the current position) regardless of its name.

If successful, a non zero `DevProperty` cookie is returned, otherwise `NULL` is returned. The `dtreePropFind` / `dtreePropFindNext` pair are typically used to iterate through either all device node properties, or a subset of device node properties whose names match a given name. In a case where a user knows that there is only one property with a given name attached to the device node, a single `dtreePropFind` invocation is sufficient.

`dtreePropLength` returns the property value length (in bytes). The property object is specified by the *prop* argument.

`dtreePropValue` returns a pointer to the first byte of the property value. The property object is specified by the *prop* argument. A driver can read and write the property value directly using the returned pointer. Typically, the driver will cast the pointer to a well known type/structure in order to access the property value. Note that the property value must always be presented in the CPU endian format.

`dtreePropName` returns a pointer to an ASCII string which designates the property name. The property object is specified by the prop argument.

`dtreePropAlloc` allocates a new device property object. A non zero `DevProperty` cookie is returned if successful, otherwise `NULL` is returned. The *name* argument specifies the property name. The *length* argument specifies the length of the property value. The property value is undefined. The property object allocated is not attached to any device node. Once the property value is initialized, the property object can be attached to a node (that is, added to the node properties list) using the `dtreePropAttach` function.

`dtreePropFree` releases the memory allocated by the property object. The *prop* parameter specifies the property object being released. Note that the property object must not be attached to any device node.

`dtreePropAttach` adds the property object specified by the *prop* argument to the node properties list. The *node* argument specifies the device node to which the property will be attached.

`dtreePropDetach` detaches the property object from the device node to which it is attached (if any). The *prop* argument specifies the property object.

Note that the device tree structure must only be accessed in the DKI thread context. The DKI thread provides access synchronization to the device tree. In many cases, driver routines which typically examine or modify the device tree are already invoked by the DKI thread. For instance, the driver probing, initialization and shut-down code is executed by the DKI thread. In a case when a driver needs to access the device tree at run time, a driver routine which uses the device tree API must be explicitly called in the DKI thread context. This rule also applies to driver clients; a driver client must always switch to the DKI thread context when accessing the device tree structure.

Device Tree High-Level Services

This section describes high-level APIs built upon other basic services. This API implements certain useful services for building and searching the device tree, in order to avoid implementing this code in all device tree users.

`dtreeNodeAdd` allocates a new device node object, and adds it to the child list of the *parent* device node specified..

The *name* argument specifies the name of the new device node allocated.. This means that a `node` property is allocated and attached to the new node with the value specified by *name* . If successful, the newly allocated node object is returned, otherwise, a NULL pointer is returned.

`dtreeNodeFind` looks for a named node in the child list of a specified device node. The *parent* argument specifies the device node within which the child list should be searched. The *name* argument specifies the value which must match the *node* property value of the node being searched.. If successful, the matching node object is returned, otherwise a NULL pointer is returned.

`dtreePropAdd` allocates a new property, sets its value and attaches it to a given device node. The *node* argument specifies the node to which the new property must be attached. The *name* argument specifies the name of the new property to allocate. The *length* argument specifies the memory size in bytes to be allocated for the new property value. The *value* argument specifies the value to be set for the newly allocated property. If successful, the newly allocated property object is returned, otherwise, a NULL pointer is returned.

Allowed Calling Contexts

The following table specifies the contexts in which a caller may invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
dtreeNodeRoot	-	+	-	-
dtreeNodeChild	-	+	-	-
dtreeNodePeer	-	+	-	-
dtreeNodeParent	-	+	-	-
dtreeNodeAlloc	+	+	-	+
dtreeNodeFree	+	+	-	+
dtreeNodeAttach	-	+	-	-
dtreeNodeDetach	-	+	-	-
dtreePropFind	-	+	-	-
dtreePropFindNext	-	+	-	-
dtreePropLength	-	+	-	-

dtreePropValue	-	+	-	-
dtreePropName	-	+	-	-
dtreePropAlloc	+	+	-	+
dtreePropFree	+	+	-	+
dtreePropAttach	-	+	-	-
dtreePropDetach	-	+	-	-
dtreeNodeAdd	-	+	-	+
dtreeNodeFind	-	+	-	-
dtreePropAdd	-	+	-	+
dtreePathLeng	+	+	_	_
dtreePathGet	+	+	_	_

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

dtreeNodeRoot, dtreeNodeChild, dtreeNodePeer, dtreeNodeParent, dtreeNodeAlloc, dtreeNodeFree, dtreeNodeAttach, dtreeNodeDetach, dtreePropFind, dtreePropFindNext, dtreePropLength, dtreePropValue, dtreePropName, dtreePropAlloc, dtreePropFree, dtreePropAttach, dtreePropDetach, dtreeNodeAdd, dtreeNodeFind, dtreePropAdd, dtreePathLeng, dtreePathGet – device tree operations

SYNOPSIS

```
#include <dki/dki.h>
DevNode dtreeNodeRoot(void);

DevNode dtreeNodeChild(DevNode node);

DevNode dtreeNodePeer(DevNode node);

DevNode dtreeNodeParent(DevNode node);

DevNode dtreeNodeAlloc(void);

void dtreeNodeFree(DevNode );

void dtreeNodeAttach(DevNode pnode, DevNode cnode);

void dtreeNodeDetach(DevNode node);

DevProperty dtreePropFind(DevNode node, char * name);

DevProperty dtreePropFindNext(DevProperty prop, char * name);

unsigned int dtreePropLength(DevProperty prop);

void * dtreePropValue(DevProperty prop);

char * dtreePropName(DevProperty prop);

DevProperty dtreePropAlloc(char * name, int length);

void dtreePropFree(DevProperty prop);

void dtreePropAttach(DevNode node, DevProperty prop);

void dtreePropDetach(DevProperty prop);

DevNode dtreeNodeAdd(DevNode parent, char * name);

DevNode dtreeNodeFind(DevNode parent, char * name);

DevProperty dtreePropAdd(DevNode node, char * name, void * value, unsigned int
length);

DevProperty dtreePathLeng(DevNode node, int length);

DevProperty dtreePathGet(DevNode node, char * buf, void * value);
```

FEATURES

DKI

DESCRIPTION

Provides device tree operations.

EXTENDED DESCRIPTION

The device tree is a data structure which provides a description of the hardware topology and device properties. The hardware topology is specified in terms of parent/child relationships. Device properties associated with each device node in the tree are device specific.

A device property is a *name* / *value* pair. The property *name* is a null terminated ASCII string. The property *value* is a sequence of bytes specified by the *length* / *address* pair.

Note that the property value format is property specific and has to be standardized between the given producer and its consumers. For instance, among all device node properties there are some properties related to the bus resources required/allocated for the device (for instance, interrupt lines, I/O registers, DMA channels).

These types of properties must be standardized in order to be understood by the bus driver as well as by any device drivers connected to the given bus. The device tree data structure may be built either statically or dynamically. In the static case, the device tree is populated by the `booter` . For instance, the `booter` may include a pre-defined sequence of device tree function calls. Another use of the `booter` is to build the device tree from a hardware description provided by firmware.

In the dynamic case, the device tree is populated at system initialization time using an enumeration/probing mechanism. The device tree is populated by propagating from parents to children.

Note that it is possible to combine both methods. In other words, an initial (non complete) device tree may be provided by the `booter` which will later be completed dynamically using an enumeration/probing mechanism. However it is implemented, the device tree structure can be modified (extended/truncated) dynamically at run time using hot-plug insertion/removal (for example, PCMCIA cards). The device tree API is described in detail below.

Device Tree Browsing

This section describes APIs related to device tree browsing. `DevNode` is an abstract type designating a device node object, and is opaque to the driver.

`dtreeNodeRoot` returns the root device node if the device tree is not empty, otherwise NULL is returned.

`dtreeNodeChild` returns the first child node from the list of children. NULL is returned when the list of children is empty.

`dtreeNodePeer` returns the next device node from the sibling list, if any, otherwise NULL is returned. The *node* argument specifies the current device node in the sibling list.

`dtreeNodeParent` returns the parent device node, if any, otherwise NULL is returned. The *node* argument specifies the child device node.

`dtreePathLeng` returns the pathname length of a device node.

`dtreePathGet` returns in *buf* the absolute pathname of a device noe. The trailing path, of the pathname, is the name of the node and is read in a node property. If this property does not exist, the trailing path of the returned pathname is set to '???'.

Device Tree Modification

This section describes APIs related to device tree topology modification.

`dtreeNodeAlloc` allocates a new device node object. A non zero DevNode cookie is returned in case of success, otherwise NULL is returned. The allocated node has neither parent nor child nodes. There are no properties attached to the newly allocated node.

`dtreeNodeFree` releases the memory allocated by the node object and all property objects attached to the node. The *node* argument specifies the node object being released.

`dtreeNodeAttach` adds the device node specified by the *cnode* argument in the child list of the parent node specified by the *pnode* argument.

`dtreeNodeDetach` detaches the node object specified by the *node* argument from its parent (if any). When *node* specifies the root node, the device tree is emptied.

Device Node Properties

This section describes APIs related to the device node properties.

`DevProperty` is an abstract type designating a device property object, and is opaque to the driver.

`dtreePropFind` searches the first property within the properties list of the device node specified by the *node* argument. If the *name* argument is not NULL, `dtreePropFind` returns the first property whose name matches the *name* string. If the *name* argument is NULL, `dtreePropFind` returns the first property from the list regardless of its name.

In case of success, a non zero `DevProperty` cookie is returned, otherwise NULL is returned. Once the first property is found, the `dtreePropFindNext` routine may be used in order to find a subsequent property in the list. `dtreePropFindNext` searches the next property within the properties list. The current position within the properties list is specified by the *prop* argument. If the *name* argument is not `NULL` , `dtreePropFindNext` returns the next property (with respect to the current position) whose name matches the *name* string. If the *name* argument is `NULL` , `dtreePropFindNext` returns the next property from the list (with respect to the current position) regardless of its name.

If successful, a non zero `DevProperty` cookie is returned, otherwise `NULL` is returned. The `dtreePropFind` / `dtreePropFindNext` pair are typically used to iterate through either all device node properties, or a subset of device node properties whose names match a given name. In a case where a user knows that there is only one property with a given name attached to the device node, a single `dtreePropFind` invocation is sufficient.

`dtreePropLength` returns the property value length (in bytes). The property object is specified by the *prop* argument.

`dtreePropValue` returns a pointer to the first byte of the property value. The property object is specified by the *prop* argument. A driver can read and write the property value directly using the returned pointer. Typically, the driver will cast the pointer to a well known type/structure in order to access the property value. Note that the property value must always be presented in the CPU endian format.

`dtreePropName` returns a pointer to an ASCII string which designates the property name. The property object is specified by the prop argument.

`dtreePropAlloc` allocates a new device property object. A non zero `DevProperty` cookie is returned if successful, otherwise `NULL` is returned. The *name* argument specifies the property name. The *length* argument specifies the length of the property value. The property value is undefined. The property object allocated is not attached to any device node. Once the property value is initialized, the property object can be attached to a node (that is, added to the node properties list) using the `dtreePropAttach` function.

`dtreePropFree` releases the memory allocated by the property object. The *prop* parameter specifies the property object being released. Note that the property object must not be attached to any device node.

`dtreePropAttach` adds the property object specified by the *prop* argument to the node properties list. The *node* argument specifies the device node to which the property will be attached.

`dtreePropDetach` detaches the property object from the device node to which it is attached (if any). The *prop* argument specifies the property object.

Note that the device tree structure must only be accessed in the DKI thread context. The DKI thread provides access synchronization to the device tree. In many cases, driver routines which typically examine or modify the device tree are already invoked by the DKI thread. For instance, the driver probing, initialization and shut-down code is executed by the DKI thread. In a case when a driver needs to access the device tree at run time, a driver routine which uses the device tree API must be explicitly called in the DKI thread context. This rule also applies to driver clients; a driver client must always switch to the DKI thread context when accessing the device tree structure.

Device Tree High-Level Services

This section describes high-level APIs built upon other basic services. This API implements certain useful services for building and searching the device tree, in order to avoid implementing this code in all device tree users.

`dtreeNodeAdd` allocates a new device node object, and adds it to the child list of the *parent* device node specified..

The *name* argument specifies the name of the new device node allocated.. This means that a `node` property is allocated and attached to the new node with the value specified by *name* . If successful, the newly allocated node object is returned, otherwise, a NULL pointer is returned.

`dtreeNodeFind` looks for a named node in the child list of a specified device node. The *parent* argument specifies the device node within which the child list should be searched. The *name* argument specifies the value which must match the *node* property value of the node being searched.. If successful, the matching node object is returned, otherwise a NULL pointer is returned.

`dtreePropAdd` allocates a new property, sets its value and attaches it to a given device node. The *node* argument specifies the node to which the new property must be attached. The *name* argument specifies the name of the new property to allocate. The *length* argument specifies the memory size in bytes to be allocated for the new property value. The *value* argument specifies the value to be set for the newly allocated property. If successful, the newly allocated property object is returned, otherwise, a NULL pointer is returned.

Allowed Calling Contexts

The following table specifies the contexts in which a caller may invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
dtreeNodeRoot	-	+	-	-
dtreeNodeChild	-	+	-	-
dtreeNodePeer	-	+	-	-
dtreeNodeParent	-	+	-	-
dtreeNodeAlloc	+	+	-	+
dtreeNodeFree	+	+	-	+
dtreeNodeAttach	-	+	-	-
dtreeNodeDetach	-	+	-	-
dtreePropFind	-	+	-	-
dtreePropFindNext	-	+	-	-
dtreePropLength	-	+	-	-

dtreePropValue	-	+	-	-
dtreePropName	-	+	-	-
dtreePropAlloc	+	+	-	+
dtreePropFree	+	+	-	+
dtreePropAttach	-	+	-	-
dtreePropDetach	-	+	-	-
dtreeNodeAdd	-	+	-	+
dtreeNodeFind	-	+	-	-
dtreePropAdd	-	+	-	+
dtreePathLeng	+	+	_	_
dtreePathGet	+	+	_	_

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

dtreeNodeRoot, dtreeNodeChild, dtreeNodePeer, dtreeNodeParent, dtreeNodeAlloc, dtreeNodeFree, dtreeNodeAttach, dtreeNodeDetach, dtreePropFind, dtreePropFindNext, dtreePropLength, dtreePropValue, dtreePropName, dtreePropAlloc, dtreePropFree, dtreePropAttach, dtreePropDetach, dtreeNodeAdd, dtreeNodeFind, dtreePropAdd, dtreePathLeng, dtreePathGet – device tree operations

SYNOPSIS

```
#include <dki/dki.h>
DevNode dtreeNodeRoot(void);

DevNode dtreeNodeChild(DevNode node);

DevNode dtreeNodePeer(DevNode node);

DevNode dtreeNodeParent(DevNode node);

DevNode dtreeNodeAlloc(void);

void dtreeNodeFree(DevNode );

void dtreeNodeAttach(DevNode pnode, DevNode cnode);

void dtreeNodeDetach(DevNode node);

DevProperty dtreePropFind(DevNode node, char * name);

DevProperty dtreePropFindNext(DevProperty prop, char * name);

unsigned int dtreePropLength(DevProperty prop);

void * dtreePropValue(DevProperty prop);

char * dtreePropName(DevProperty prop);

DevProperty dtreePropAlloc(char * name, int length);

void dtreePropFree(DevProperty prop);

void dtreePropAttach(DevNode node, DevProperty prop);

void dtreePropDetach(DevProperty prop);

DevNode dtreeNodeAdd(DevNode parent, char * name);

DevNode dtreeNodeFind(DevNode parent, char * name);

DevProperty dtreePropAdd(DevNode node, char * name, void * value, unsigned int
length);

DevProperty dtreePathLeng(DevNode node, int length);

DevProperty dtreePathGet(DevNode node, char * buf, void * value);
```

FEATURES

DKI

DESCRIPTION

Provides device tree operations.

EXTENDED DESCRIPTION

The device tree is a data structure which provides a description of the hardware topology and device properties. The hardware topology is specified in terms of parent/child relationships. Device properties associated with each device node in the tree are device specific.

A device property is a *name* / *value* pair. The property *name* is a null terminated ASCII string. The property *value* is a sequence of bytes specified by the *length* / *address* pair.

Note that the property value format is property specific and has to be standardized between the given producer and its consumers. For instance, among all device node properties there are some properties related to the bus resources required/allocated for the device (for instance, interrupt lines, I/O registers, DMA channels).

These types of properties must be standardized in order to be understood by the bus driver as well as by any device drivers connected to the given bus. The device tree data structure may be built either statically or dynamically. In the static case, the device tree is populated by the `booter` . For instance, the `booter` may include a pre-defined sequence of device tree function calls. Another use of the `booter` is to build the device tree from a hardware description provided by firmware.

In the dynamic case, the device tree is populated at system initialization time using an enumeration/probing mechanism. The device tree is populated by propagating from parents to children.

Note that it is possible to combine both methods. In other words, an initial (non complete) device tree may be provided by the `booter` which will later be completed dynamically using an enumeration/probing mechanism. However it is implemented, the device tree structure can be modified (extended/truncated) dynamically at run time using hot-plug insertion/removal (for example, PCMCIA cards). The device tree API is described in detail below.

Device Tree Browsing

This section describes APIs related to device tree browsing. `DevNode` is an abstract type designating a device node object, and is opaque to the driver.

`dtreeNodeRoot` returns the root device node if the device tree is not empty, otherwise NULL is returned.

`dtreeNodeChild` returns the first child node from the list of children. NULL is returned when the list of children is empty.

`dtreeNodePeer` returns the next device node from the sibling list, if any, otherwise NULL is returned. The *node* argument specifies the current device node in the sibling list.

`dtreeNodeParent` returns the parent device node, if any, otherwise NULL is returned. The *node* argument specifies the child device node.

`dtreePathLeng` returns the pathname length of a device node.

`dtreePathGet` returns in *buf* the absolute pathname of a device noe. The trailing path, of the pathname, is the name of the node and is read in a node property. If this property does not exist, the trailing path of the returned pathname is set to '???'.

Device Tree Modification

This section describes APIs related to device tree topology modification.

`dtreeNodeAlloc` allocates a new device node object. A non zero DevNode cookie is returned in case of success, otherwise NULL is returned. The allocated node has neither parent nor child nodes. There are no properties attached to the newly allocated node.

`dtreeNodeFree` releases the memory allocated by the node object and all property objects attached to the node. The *node* argument specifies the node object being released.

`dtreeNodeAttach` adds the device node specified by the *cnode* argument in the child list of the parent node specified by the *pnode* argument.

`dtreeNodeDetach` detaches the node object specified by the *node* argument from its parent (if any). When *node* specifies the root node, the device tree is emptied.

Device Node Properties

This section describes APIs related to the device node properties.

`DevProperty` is an abstract type designating a device property object, and is opaque to the driver.

`dtreePropFind` searches the first property within the properties list of the device node specified by the *node* argument. If the *name* argument is not NULL, `dtreePropFind` returns the first property whose name matches the *name* string. If the *name* argument is NULL, `dtreePropFind` returns the first property from the list regardless of its name.

In case of success, a non zero `DevProperty` cookie is returned, otherwise NULL is returned. Once the first property is found, the `dtreePropFindNext` routine may be used in order to find a subsequent property in the list. `dtreePropFindNext` searches the next property within the properties list. The current position within the properties list is specified by the *prop* argument. If the *name* argument is not `NULL` , `dtreePropFindNext` returns the next property (with respect to the current position) whose name matches the *name* string. If the *name* argument is `NULL` , `dtreePropFindNext` returns the next property from the list (with respect to the current position) regardless of its name.

If successful, a non zero `DevProperty` cookie is returned, otherwise `NULL` is returned. The `dtreePropFind` / `dtreePropFindNext` pair are typically used to iterate through either all device node properties, or a subset of device node properties whose names match a given name. In a case where a user knows that there is only one property with a given name attached to the device node, a single `dtreePropFind` invocation is sufficient.

`dtreePropLength` returns the property value length (in bytes). The property object is specified by the *prop* argument.

`dtreePropValue` returns a pointer to the first byte of the property value. The property object is specified by the *prop* argument. A driver can read and write the property value directly using the returned pointer. Typically, the driver will cast the pointer to a well known type/structure in order to access the property value. Note that the property value must always be presented in the CPU endian format.

`dtreePropName` returns a pointer to an ASCII string which designates the property name. The property object is specified by the prop argument.

`dtreePropAlloc` allocates a new device property object. A non zero `DevProperty` cookie is returned if successful, otherwise `NULL` is returned. The *name* argument specifies the property name. The *length* argument specifies the length of the property value. The property value is undefined. The property object allocated is not attached to any device node. Once the property value is initialized, the property object can be attached to a node (that is, added to the node properties list) using the `dtreePropAttach` function.

`dtreePropFree` releases the memory allocated by the property object. The *prop* parameter specifies the property object being released. Note that the property object must not be attached to any device node.

`dtreePropAttach` adds the property object specified by the *prop* argument to the node properties list. The *node* argument specifies the device node to which the property will be attached.

`dtreePropDetach` detaches the property object from the device node to which it is attached (if any). The *prop* argument specifies the property object.

Note that the device tree structure must only be accessed in the DKI thread context. The DKI thread provides access synchronization to the device tree. In many cases, driver routines which typically examine or modify the device tree are already invoked by the DKI thread. For instance, the driver probing, initialization and shut-down code is executed by the DKI thread. In a case when a driver needs to access the device tree at run time, a driver routine which uses the device tree API must be explicitly called in the DKI thread context. This rule also applies to driver clients; a driver client must always switch to the DKI thread context when accessing the device tree structure.

Device Tree High-Level Services

This section describes high-level APIs built upon other basic services. This API implements certain useful services for building and searching the device tree, in order to avoid implementing this code in all device tree users.

`dtreeNodeAdd` allocates a new device node object, and adds it to the child list of the *parent* device node specified..

The *name* argument specifies the name of the new device node allocated.. This means that a `node` property is allocated and attached to the new node with the value specified by *name* . If successful, the newly allocated node object is returned, otherwise, a NULL pointer is returned.

`dtreeNodeFind` looks for a named node in the child list of a specified device node. The *parent* argument specifies the device node within which the child list should be searched. The *name* argument specifies the value which must match the *node* property value of the node being searched.. If successful, the matching node object is returned, otherwise a NULL pointer is returned.

`dtreePropAdd` allocates a new property, sets its value and attaches it to a given device node. The *node* argument specifies the node to which the new property must be attached. The *name* argument specifies the name of the new property to allocate. The *length* argument specifies the memory size in bytes to be allocated for the new property value. The *value* argument specifies the value to be set for the newly allocated property. If successful, the newly allocated property object is returned, otherwise, a NULL pointer is returned.

Allowed Calling Contexts

The following table specifies the contexts in which a caller may invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
dtreeNodeRoot	-	+	-	-
dtreeNodeChild	-	+	-	-
dtreeNodePeer	-	+	-	-
dtreeNodeParent	-	+	-	-
dtreeNodeAlloc	+	+	-	+
dtreeNodeFree	+	+	-	+
dtreeNodeAttach	-	+	-	-
dtreeNodeDetach	-	+	-	-
dtreePropFind	-	+	-	-
dtreePropFindNext	-	+	-	-
dtreePropLength	-	+	-	-

dtreePropValue	-	+	-	-
dtreePropName	-	+	-	-
dtreePropAlloc	+	+	-	+
dtreePropFree	+	+	-	+
dtreePropAttach	-	+	-	-
dtreePropDetach	-	+	-	-
dtreeNodeAdd	-	+	-	+
dtreeNodeFind	-	+	-	-
dtreePropAdd	-	+	-	+
dtreePathLeng	+	+	_	_
dtreePathGet	+	+	_	_

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

dtreeNodeRoot, dtreeNodeChild, dtreeNodePeer, dtreeNodeParent, dtreeNodeAlloc, dtreeNodeFree, dtreeNodeAttach, dtreeNodeDetach, dtreePropFind, dtreePropFindNext, dtreePropLength, dtreePropValue, dtreePropName, dtreePropAlloc, dtreePropFree, dtreePropAttach, dtreePropDetach, dtreeNodeAdd, dtreeNodeFind, dtreePropAdd, dtreePathLeng, dtreePathGet – device tree operations

SYNOPSIS

```
#include <dki/dki.h>
DevNode dtreeNodeRoot(void);

DevNode dtreeNodeChild(DevNode node);

DevNode dtreeNodePeer(DevNode node);

DevNode dtreeNodeParent(DevNode node);

DevNode dtreeNodeAlloc(void);

void dtreeNodeFree(DevNode );

void dtreeNodeAttach(DevNode pnode, DevNode cnode);

void dtreeNodeDetach(DevNode node);

DevProperty dtreePropFind(DevNode node, char * name);

DevProperty dtreePropFindNext(DevProperty prop, char * name);

unsigned int dtreePropLength(DevProperty prop);

void * dtreePropValue(DevProperty prop);

char * dtreePropName(DevProperty prop);

DevProperty dtreePropAlloc(char * name, int length);

void dtreePropFree(DevProperty prop);

void dtreePropAttach(DevNode node, DevProperty prop);

void dtreePropDetach(DevProperty prop);

DevNode dtreeNodeAdd(DevNode parent, char * name);

DevNode dtreeNodeFind(DevNode parent, char * name);

DevProperty dtreePropAdd(DevNode node, char * name, void * value, unsigned int
length);

DevProperty dtreePathLeng(DevNode node, int length);

DevProperty dtreePathGet(DevNode node, char * buf, void * value);
```

FEATURES

DKI

DESCRIPTION

Provides device tree operations.

EXTENDED DESCRIPTION

The device tree is a data structure which provides a description of the hardware topology and device properties. The hardware topology is specified in terms of parent / child relationships. Device properties associated with each device node in the tree are device specific.

A device property is a *name* / *value* pair. The property *name* is a null terminated ASCII string. The property *value* is a sequence of bytes specified by the *length* / *address* pair.

Note that the property value format is property specific and has to be standardized between the given producer and its consumers. For instance, among all device node properties there are some properties related to the bus resources required / allocated for the device (for instance, interrupt lines, I / O registers, DMA channels).

These types of properties must be standardized in order to be understood by the bus driver as well as by any device drivers connected to the given bus. The device tree data structure may be built either statically or dynamically. In the static case, the device tree is populated by the `booter` . For instance, the `booter` may include a pre-defined sequence of device tree function calls. Another use of the `booter` is to build the device tree from a hardware description provided by firmware.

In the dynamic case, the device tree is populated at system initialization time using an enumeration / probing mechanism. The device tree is populated by propagating from parents to children.

Note that it is possible to combine both methods. In other words, an initial (non complete) device tree may be provided by the `booter` which will later be completed dynamically using an enumeration / probing mechanism. However it is implemented, the device tree structure can be modified (extended / truncated) dynamically at run time using hot-plug insertion / removal (for example, PCMCIA cards). The device tree API is described in detail below.

Device Tree Browsing

This section describes APIs related to device tree browsing. `DevNode` is an abstract type designating a device node object, and is opaque to the driver.

`dtreeNodeRoot` returns the root device node if the device tree is not empty, otherwise NULL is returned.

`dtreeNodeChild` returns the first child node from the list of children. NULL is returned when the list of children is empty.

`dtreeNodePeer` returns the next device node from the sibling list, if any, otherwise NULL is returned. The *node* argument specifies the current device node in the sibling list.

`dtreeNodeParent` returns the parent device node, if any, otherwise NULL is returned. The *node* argument specifies the child device node.

`dtreePathLeng` returns the pathname length of a device node.

`dtreePathGet` returns in *buf* the absolute pathname of a device noe. The trailing path, of the pathname, is the name of the node and is read in a node property. If this property does not exist, the trailing path of the returned pathname is set to '???'.

Device Tree Modification

This section describes APIs related to device tree topology modification.

`dtreeNodeAlloc` allocates a new device node object. A non zero DevNode cookie is returned in case of success, otherwise NULL is returned. The allocated node has neither parent nor child nodes. There are no properties attached to the newly allocated node.

`dtreeNodeFree` releases the memory allocated by the node object and all property objects attached to the node. The *node* argument specifies the node object being released.

`dtreeNodeAttach` adds the device node specified by the *cnode* argument in the child list of the parent node specified by the *pnode* argument.

`dtreeNodeDetach` detaches the node object specified by the *node* argument from its parent (if any). When *node* specifies the root node, the device tree is emptied.

Device Node Properties

This section describes APIs related to the device node properties.

`DevProperty` is an abstract type designating a device property object, and is opaque to the driver.

`dtreePropFind` searches the first property within the properties list of the device node specified by the *node* argument. If the *name* argument is not NULL, `dtreePropFind` returns the first property whose name matches the *name* string. If the *name* argument is NULL, `dtreePropFind` returns the first property from the list regardless of its name.

In case of success, a non zero `DevProperty` cookie is returned, otherwise NULL is returned. Once the first property is found, the `dtreePropFindNext` routine may be used in order to find a subsequent property in the list. `dtreePropFindNext` searches the next property within the properties list. The current position within the properties list is specified by the *prop* argument. If the *name* argument is not `NULL` , `dtreePropFindNext` returns the next property (with respect to the current position) whose name matches the *name* string. If the *name* argument is `NULL` , `dtreePropFindNext` returns the next property from the list (with respect to the current position) regardless of its name.

If successful, a non zero `DevProperty` cookie is returned, otherwise `NULL` is returned. The `dtreePropFind` / `dtreePropFindNext` pair are typically used to iterate through either all device node properties, or a subset of device node properties whose names match a given name. In a case where a user knows that there is only one property with a given name attached to the device node, a single `dtreePropFind` invocation is sufficient.

`dtreePropLength` returns the property value length (in bytes). The property object is specified by the *prop* argument.

`dtreePropValue` returns a pointer to the first byte of the property value. The property object is specified by the *prop* argument. A driver can read and write the property value directly using the returned pointer. Typically, the driver will cast the pointer to a well known type/structure in order to access the property value. Note that the property value must always be presented in the CPU endian format.

`dtreePropName` returns a pointer to an ASCII string which designates the property name. The property object is specified by the prop argument.

`dtreePropAlloc` allocates a new device property object. A non zero `DevProperty` cookie is returned if successful, otherwise `NULL` is returned. The *name* argument specifies the property name. The *length* argument specifies the length of the property value. The property value is undefined. The property object allocated is not attached to any device node. Once the property value is initialized, the property object can be attached to a node (that is, added to the node properties list) using the `dtreePropAttach` function.

`dtreePropFree` releases the memory allocated by the property object. The *prop* parameter specifies the property object being released. Note that the property object must not be attached to any device node.

`dtreePropAttach` adds the property object specified by the *prop* argument to the node properties list. The *node* argument specifies the device node to which the property will be attached.

`dtreePropDetach` detaches the property object from the device node to which it is attached (if any). The *prop* argument specifies the property object.

Note that the device tree structure must only be accessed in the DKI thread context. The DKI thread provides access synchronization to the device tree. In many cases, driver routines which typically examine or modify the device tree are already invoked by the DKI thread. For instance, the driver probing, initialization and shut-down code is executed by the DKI thread. In a case when a driver needs to access the device tree at run time, a driver routine which uses the device tree API must be explicitly called in the DKI thread context. This rule also applies to driver clients; a driver client must always switch to the DKI thread context when accessing the device tree structure.

Device Tree High-Level Services

This section describes high-level APIs built upon other basic services. This API implements certain useful services for building and searching the device tree, in order to avoid implementing this code in all device tree users.

`dtreeNodeAdd` allocates a new device node object, and adds it to the child list of the *parent* device node specified..

The *name* argument specifies the name of the new device node allocated.. This means that a `node` property is allocated and attached to the new node with the value specified by *name* . If successful, the newly allocated node object is returned, otherwise, a NULL pointer is returned.

`dtreeNodeFind` looks for a named node in the child list of a specified device node. The *parent* argument specifies the device node within which the child list should be searched. The *name* argument specifies the value which must match the *node* property value of the node being searched.. If successful, the matching node object is returned, otherwise a NULL pointer is returned.

`dtreePropAdd` allocates a new property, sets its value and attaches it to a given device node. The *node* argument specifies the node to which the new property must be attached. The *name* argument specifies the name of the new property to allocate. The *length* argument specifies the memory size in bytes to be allocated for the new property value. The *value* argument specifies the value to be set for the newly allocated property. If successful, the newly allocated property object is returned, otherwise, a NULL pointer is returned.

Allowed Calling Contexts

The following table specifies the contexts in which a caller may invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
dtreeNodeRoot	-	+	-	-
dtreeNodeChild	-	+	-	-
dtreeNodePeer	-	+	-	-
dtreeNodeParent	-	+	-	-
dtreeNodeAlloc	+	+	-	+
dtreeNodeFree	+	+	-	+
dtreeNodeAttach	-	+	-	-
dtreeNodeDetach	-	+	-	-
dtreePropFind	-	+	-	-
dtreePropFindNext	-	+	-	-
dtreePropLength	-	+	-	-

dtreePropValue	-	+	-	-
dtreePropName	-	+	-	-
dtreePropAlloc	+	+	-	+
dtreePropFree	+	+	-	+
dtreePropAttach	-	+	-	-
dtreePropDetach	-	+	-	-
dtreeNodeAdd	-	+	-	+
dtreeNodeFind	-	+	-	-
dtreePropAdd	-	+	-	+
dtreePathLeng	+	+	_	_
dtreePathGet	+	+	_	_

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

dtreeNodeRoot, dtreeNodeChild, dtreeNodePeer, dtreeNodeParent, dtreeNodeAlloc, dtreeNodeFree, dtreeNodeAttach, dtreeNodeDetach, dtreePropFind, dtreePropFindNext, dtreePropLength, dtreePropValue, dtreePropName, dtreePropAlloc, dtreePropFree, dtreePropAttach, dtreePropDetach, dtreeNodeAdd, dtreeNodeFind, dtreePropAdd, dtreePathLeng, dtreePathGet – device tree operations

SYNOPSIS

```
#include <dki/dki.h>
DevNode dtreeNodeRoot(void);

DevNode dtreeNodeChild(DevNode node);

DevNode dtreeNodePeer(DevNode node);

DevNode dtreeNodeParent(DevNode node);

DevNode dtreeNodeAlloc(void);

void dtreeNodeFree(DevNode );

void dtreeNodeAttach(DevNode pnode, DevNode cnode);

void dtreeNodeDetach(DevNode node);

DevProperty dtreePropFind(DevNode node, char * name);

DevProperty dtreePropFindNext(DevProperty prop, char * name);

unsigned int dtreePropLength(DevProperty prop);

void * dtreePropValue(DevProperty prop);

char * dtreePropName(DevProperty prop);

DevProperty dtreePropAlloc(char * name, int length);

void dtreePropFree(DevProperty prop);

void dtreePropAttach(DevNode node, DevProperty prop);

void dtreePropDetach(DevProperty prop);

DevNode dtreeNodeAdd(DevNode parent, char * name);

DevNode dtreeNodeFind(DevNode parent, char * name);

DevProperty dtreePropAdd(DevNode node, char * name, void * value, unsigned int
length);

DevProperty dtreePathLeng(DevNode node, int length);

DevProperty dtreePathGet(DevNode node, char * buf, void * value);
```

FEATURES

DKI

DESCRIPTION

Provides device tree operations.

EXTENDED DESCRIPTION

The device tree is a data structure which provides a description of the hardware topology and device properties. The hardware topology is specified in terms of parent / child relationships. Device properties associated with each device node in the tree are device specific.

A device property is a *name* / *value* pair. The property *name* is a null terminated ASCII string. The property *value* is a sequence of bytes specified by the *length* / *address* pair.

Note that the property value format is property specific and has to be standardized between the given producer and its consumers. For instance, among all device node properties there are some properties related to the bus resources required / allocated for the device (for instance, interrupt lines, I / O registers, DMA channels).

These types of properties must be standardized in order to be understood by the bus driver as well as by any device drivers connected to the given bus. The device tree data structure may be built either statically or dynamically. In the static case, the device tree is populated by the `booter` . For instance, the `booter` may include a pre-defined sequence of device tree function calls. Another use of the `booter` is to build the device tree from a hardware description provided by firmware.

In the dynamic case, the device tree is populated at system initialization time using an enumeration / probing mechanism. The device tree is populated by propagating from parents to children.

Note that it is possible to combine both methods. In other words, an initial (non complete) device tree may be provided by the `booter` which will later be completed dynamically using an enumeration / probing mechanism. However it is implemented, the device tree structure can be modified (extended / truncated) dynamically at run time using hot-plug insertion / removal (for example, PCMCIA cards). The device tree API is described in detail below.

Device Tree Browsing

This section describes APIs related to device tree browsing. `DevNode` is an abstract type designating a device node object, and is opaque to the driver.

`dtreeNodeRoot` returns the root device node if the device tree is not empty, otherwise NULL is returned.

`dtreeNodeChild` returns the first child node from the list of children. NULL is returned when the list of children is empty.

`dtreeNodePeer` returns the next device node from the sibling list, if any, otherwise NULL is returned. The *node* argument specifies the current device node in the sibling list.

`dtreeNodeParent` returns the parent device node, if any, otherwise NULL is returned. The *node* argument specifies the child device node.

`dtreePathLeng` returns the pathname length of a device node.

`dtreePathGet` returns in *buf* the absolute pathname of a device noe. The trailing path, of the pathname, is the name of the node and is read in a node property. If this property does not exist, the trailing path of the returned pathname is set to '???'.

Device Tree Modification

This section describes APIs related to device tree topology modification.

`dtreeNodeAlloc` allocates a new device node object. A non zero DevNode cookie is returned in case of success, otherwise NULL is returned. The allocated node has neither parent nor child nodes. There are no properties attached to the newly allocated node.

`dtreeNodeFree` releases the memory allocated by the node object and all property objects attached to the node. The *node* argument specifies the node object being released.

`dtreeNodeAttach` adds the device node specified by the *cnode* argument in the child list of the parent node specified by the *pnode* argument.

`dtreeNodeDetach` detaches the node object specified by the *node* argument from its parent (if any). When *node* specifies the root node, the device tree is emptied.

Device Node Properties

This section describes APIs related to the device node properties.

`DevProperty` is an abstract type designating a device property object, and is opaque to the driver.

`dtreePropFind` searches the first property within the properties list of the device node specified by the *node* argument. If the *name* argument is not NULL, `dtreePropFind` returns the first property whose name matches the *name* string. If the *name* argument is NULL, `dtreePropFind` returns the first property from the list regardless of its name.

In case of success, a non zero `DevProperty` cookie is returned, otherwise NULL is returned. Once the first property is found, the `dtreePropFindNext` routine may be used in order to find a subsequent property in the list. `dtreePropFindNext` searches the next property within the properties list. The current position within the properties list is specified by the *prop* argument. If the *name* argument is not `NULL` , `dtreePropFindNext` returns the next property (with respect to the current position) whose name matches the *name* string. If the *name* argument is `NULL` , `dtreePropFindNext` returns the next property from the list (with respect to the current position) regardless of its name.

If successful, a non zero `DevProperty` cookie is returned, otherwise `NULL` is returned. The `dtreePropFind` / `dtreePropFindNext` pair are typically used to iterate through either all device node properties, or a subset of device node properties whose names match a given name. In a case where a user knows that there is only one property with a given name attached to the device node, a single `dtreePropFind` invocation is sufficient.

`dtreePropLength` returns the property value length (in bytes). The property object is specified by the *prop* argument.

`dtreePropValue` returns a pointer to the first byte of the property value. The property object is specified by the *prop* argument. A driver can read and write the property value directly using the returned pointer. Typically, the driver will cast the pointer to a well known type/structure in order to access the property value. Note that the property value must always be presented in the CPU endian format.

`dtreePropName` returns a pointer to an ASCII string which designates the property name. The property object is specified by the prop argument.

`dtreePropAlloc` allocates a new device property object. A non zero `DevProperty` cookie is returned if successful, otherwise `NULL` is returned. The *name* argument specifies the property name. The *length* argument specifies the length of the property value. The property value is undefined. The property object allocated is not attached to any device node. Once the property value is initialized, the property object can be attached to a node (that is, added to the node properties list) using the `dtreePropAttach` function.

`dtreePropFree` releases the memory allocated by the property object. The *prop* parameter specifies the property object being released. Note that the property object must not be attached to any device node.

`dtreePropAttach` adds the property object specified by the *prop* argument to the node properties list. The *node* argument specifies the device node to which the property will be attached.

`dtreePropDetach` detaches the property object from the device node to which it is attached (if any). The *prop* argument specifies the property object.

Note that the device tree structure must only be accessed in the DKI thread context. The DKI thread provides access synchronization to the device tree. In many cases, driver routines which typically examine or modify the device tree are already invoked by the DKI thread. For instance, the driver probing, initialization and shut-down code is executed by the DKI thread. In a case when a driver needs to access the device tree at run time, a driver routine which uses the device tree API must be explicitly called in the DKI thread context. This rule also applies to driver clients; a driver client must always switch to the DKI thread context when accessing the device tree structure.

Device Tree High-Level Services

This section describes high-level APIs built upon other basic services. This API implements certain useful services for building and searching the device tree, in order to avoid implementing this code in all device tree users.

`dtreeNodeAdd` allocates a new device node object, and adds it to the child list of the *parent* device node specified..

The *name* argument specifies the name of the new device node allocated.. This means that a `node` property is allocated and attached to the new node with the value specified by *name* . If successful, the newly allocated node object is returned, otherwise, a NULL pointer is returned.

`dtreeNodeFind` looks for a named node in the child list of a specified device node. The *parent* argument specifies the device node within which the child list should be searched. The *name* argument specifies the value which must match the *node* property value of the node being searched.. If successful, the matching node object is returned, otherwise a NULL pointer is returned.

`dtreePropAdd` allocates a new property, sets its value and attaches it to a given device node. The *node* argument specifies the node to which the new property must be attached. The *name* argument specifies the name of the new property to allocate. The *length* argument specifies the memory size in bytes to be allocated for the new property value. The *value* argument specifies the value to be set for the newly allocated property. If successful, the newly allocated property object is returned, otherwise, a NULL pointer is returned.

Allowed Calling Contexts

The following table specifies the contexts in which a caller may invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
dtreeNodeRoot	-	+	-	-
dtreeNodeChild	-	+	-	-
dtreeNodePeer	-	+	-	-
dtreeNodeParent	-	+	-	-
dtreeNodeAlloc	+	+	-	+
dtreeNodeFree	+	+	-	+
dtreeNodeAttach	-	+	-	-
dtreeNodeDetach	-	+	-	-
dtreePropFind	-	+	-	-
dtreePropFindNext	-	+	-	-
dtreePropLength	-	+	-	-

dtreePropValue	-	+	-	-
dtreePropName	-	+	-	-
dtreePropAlloc	+	+	-	+
dtreePropFree	+	+	-	+
dtreePropAttach	-	+	-	-
dtreePropDetach	-	+	-	-
dtreeNodeAdd	-	+	-	+
dtreeNodeFind	-	+	-	-
dtreePropAdd	-	+	-	+
dtreePathLeng	+	+	_	_
dtreePathGet	+	+	_	_

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

dtreeNodeRoot, dtreeNodeChild, dtreeNodePeer, dtreeNodeParent, dtreeNodeAlloc, dtreeNodeFree, dtreeNodeAttach, dtreeNodeDetach, dtreePropFind, dtreePropFindNext, dtreePropLength, dtreePropValue, dtreePropName, dtreePropAlloc, dtreePropFree, dtreePropAttach, dtreePropDetach, dtreeNodeAdd, dtreeNodeFind, dtreePropAdd, dtreePathLeng, dtreePathGet – device tree operations

SYNOPSIS

```
#include <dki/dki.h>
DevNode dtreeNodeRoot(void);

DevNode dtreeNodeChild(DevNode node);

DevNode dtreeNodePeer(DevNode node);

DevNode dtreeNodeParent(DevNode node);

DevNode dtreeNodeAlloc(void);

void dtreeNodeFree(DevNode );

void dtreeNodeAttach(DevNode pnode, DevNode cnode);

void dtreeNodeDetach(DevNode node);

DevProperty dtreePropFind(DevNode node, char * name);

DevProperty dtreePropFindNext(DevProperty prop, char * name);

unsigned int dtreePropLength(DevProperty prop);

void * dtreePropValue(DevProperty prop);

char * dtreePropName(DevProperty prop);

DevProperty dtreePropAlloc(char * name, int length);

void dtreePropFree(DevProperty prop);

void dtreePropAttach(DevNode node, DevProperty prop);

void dtreePropDetach(DevProperty prop);

DevNode dtreeNodeAdd(DevNode parent, char * name);

DevNode dtreeNodeFind(DevNode parent, char * name);

DevProperty dtreePropAdd(DevNode node, char * name, void * value, unsigned int
length);

DevProperty dtreePathLeng(DevNode node, int length);

DevProperty dtreePathGet(DevNode node, char * buf, void * value);
```

FEATURES

DKI

DESCRIPTION

Provides device tree operations.

EXTENDED DESCRIPTION

The device tree is a data structure which provides a description of the hardware topology and device properties. The hardware topology is specified in terms of parent/child relationships. Device properties associated with each device node in the tree are device specific.

A device property is a *name* / *value* pair. The property *name* is a null terminated ASCII string. The property *value* is a sequence of bytes specified by the *length* / *address* pair.

Note that the property value format is property specific and has to be standardized between the given producer and its consumers. For instance, among all device node properties there are some properties related to the bus resources required/allocated for the device (for instance, interrupt lines, I/O registers, DMA channels).

These types of properties must be standardized in order to be understood by the bus driver as well as by any device drivers connected to the given bus. The device tree data structure may be built either statically or dynamically. In the static case, the device tree is populated by the `booter` . For instance, the `booter` may include a pre-defined sequence of device tree function calls. Another use of the `booter` is to build the device tree from a hardware description provided by firmware.

In the dynamic case, the device tree is populated at system initialization time using an enumeration/probing mechanism. The device tree is populated by propagating from parents to children.

Note that it is possible to combine both methods. In other words, an initial (non complete) device tree may be provided by the `booter` which will later be completed dynamically using an enumeration/probing mechanism. However it is implemented, the device tree structure can be modified (extended/truncated) dynamically at run time using hot-plug insertion/removal (for example, PCMCIA cards). The device tree API is described in detail below.

Device Tree Browsing

This section describes APIs related to device tree browsing. `DevNode` is an abstract type designating a device node object, and is opaque to the driver.

`dtreeNodeRoot` returns the root device node if the device tree is not empty, otherwise NULL is returned.

`dtreeNodeChild` returns the first child node from the list of children. NULL is returned when the list of children is empty.

`dtreeNodePeer` returns the next device node from the sibling list, if any, otherwise NULL is returned. The *node* argument specifies the current device node in the sibling list.

`dtreeNodeParent` returns the parent device node, if any, otherwise NULL is returned. The *node* argument specifies the child device node.

`dtreePathLeng` returns the pathname length of a device node.

`dtreePathGet` returns in *buf* the absolute pathname of a device noe. The trailing path, of the pathname, is the name of the node and is read in a node property. If this property does not exist, the trailing path of the returned pathname is set to '???'.

Device Tree Modification

This section describes APIs related to device tree topology modification.

`dtreeNodeAlloc` allocates a new device node object. A non zero DevNode cookie is returned in case of success, otherwise NULL is returned. The allocated node has neither parent nor child nodes. There are no properties attached to the newly allocated node.

`dtreeNodeFree` releases the memory allocated by the node object and all property objects attached to the node. The *node* argument specifies the node object being released.

`dtreeNodeAttach` adds the device node specified by the *cnode* argument in the child list of the parent node specified by the *pnode* argument.

`dtreeNodeDetach` detaches the node object specified by the *node* argument from its parent (if any). When *node* specifies the root node, the device tree is emptied.

Device Node Properties

This section describes APIs related to the device node properties.

`DevProperty` is an abstract type designating a device property object, and is opaque to the driver.

`dtreePropFind` searches the first property within the properties list of the device node specified by the *node* argument. If the *name* argument is not NULL, `dtreePropFind` returns the first property whose name matches the *name* string. If the *name* argument is NULL, `dtreePropFind` returns the first property from the list regardless of its name.

In case of success, a non zero `DevProperty` cookie is returned, otherwise NULL is returned. Once the first property is found, the `dtreePropFindNext` routine may be used in order to find a subsequent property in the list. `dtreePropFindNext` searches the next property within the properties list. The current position within the properties list is specified by the *prop* argument. If the *name* argument is not `NULL` , `dtreePropFindNext` returns the next property (with respect to the current position) whose name matches the *name* string. If the *name* argument is `NULL` , `dtreePropFindNext` returns the next property from the list (with respect to the current position) regardless of its name.

If successful, a non zero `DevProperty` cookie is returned, otherwise `NULL` is returned. The `dtreePropFind` / `dtreePropFindNext` pair are typically used to iterate through either all device node properties, or a subset of device node properties whose names match a given name. In a case where a user knows that there is only one property with a given name attached to the device node, a single `dtreePropFind` invocation is sufficient.

`dtreePropLength` returns the property value length (in bytes). The property object is specified by the *prop* argument.

`dtreePropValue` returns a pointer to the first byte of the property value. The property object is specified by the *prop* argument. A driver can read and write the property value directly using the returned pointer. Typically, the driver will cast the pointer to a well known type/structure in order to access the property value. Note that the property value must always be presented in the CPU endian format.

`dtreePropName` returns a pointer to an ASCII string which designates the property name. The property object is specified by the prop argument.

`dtreePropAlloc` allocates a new device property object. A non zero `DevProperty` cookie is returned if successful, otherwise `NULL` is returned. The *name* argument specifies the property name. The *length* argument specifies the length of the property value. The property value is undefined. The property object allocated is not attached to any device node. Once the property value is initialized, the property object can be attached to a node (that is, added to the node properties list) using the `dtreePropAttach` function.

`dtreePropFree` releases the memory allocated by the property object. The *prop* parameter specifies the property object being released. Note that the property object must not be attached to any device node.

`dtreePropAttach` adds the property object specified by the *prop* argument to the node properties list. The *node* argument specifies the device node to which the property will be attached.

`dtreePropDetach` detaches the property object from the device node to which it is attached (if any). The *prop* argument specifies the property object.

Note that the device tree structure must only be accessed in the DKI thread context. The DKI thread provides access synchronization to the device tree. In many cases, driver routines which typically examine or modify the device tree are already invoked by the DKI thread. For instance, the driver probing, initialization and shut-down code is executed by the DKI thread. In a case when a driver needs to access the device tree at run time, a driver routine which uses the device tree API must be explicitly called in the DKI thread context. This rule also applies to driver clients; a driver client must always switch to the DKI thread context when accessing the device tree structure.

Device Tree High-Level Services

This section describes high-level APIs built upon other basic services. This API implements certain useful services for building and searching the device tree, in order to avoid implementing this code in all device tree users.

`dtreeNodeAdd` allocates a new device node object, and adds it to the child list of the *parent* device node specified..

The *name* argument specifies the name of the new device node allocated.. This means that a `node` property is allocated and attached to the new node with the value specified by *name* . If successful, the newly allocated node object is returned, otherwise, a NULL pointer is returned.

`dtreeNodeFind` looks for a named node in the child list of a specified device node. The *parent* argument specifies the device node within which the child list should be searched. The *name* argument specifies the value which must match the *node* property value of the node being searched.. If successful, the matching node object is returned, otherwise a NULL pointer is returned.

`dtreePropAdd` allocates a new property, sets its value and attaches it to a given device node. The *node* argument specifies the node to which the new property must be attached. The *name* argument specifies the name of the new property to allocate. The *length* argument specifies the memory size in bytes to be allocated for the new property value. The *value* argument specifies the value to be set for the newly allocated property. If successful, the newly allocated property object is returned, otherwise, a NULL pointer is returned.

Allowed Calling Contexts

The following table specifies the contexts in which a caller may invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
dtreeNodeRoot	-	+	-	-
dtreeNodeChild	-	+	-	-
dtreeNodePeer	-	+	-	-
dtreeNodeParent	-	+	-	-
dtreeNodeAlloc	+	+	-	+
dtreeNodeFree	+	+	-	+
dtreeNodeAttach	-	+	-	-
dtreeNodeDetach	-	+	-	-
dtreePropFind	-	+	-	-
dtreePropFindNext	-	+	-	-
dtreePropLength	-	+	-	-

dtreePropValue	-	+	-	-
dtreePropName	-	+	-	-
dtreePropAlloc	+	+	-	+
dtreePropFree	+	+	-	+
dtreePropAttach	-	+	-	-
dtreePropDetach	-	+	-	-
dtreeNodeAdd	-	+	-	+
dtreeNodeFind	-	+	-	-
dtreePropAdd	-	+	-	+
dtreePathLeng	+	+	_	_
dtreePathGet	+	+	_	_

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

loadSwapEieio_16, storeSwapEieio_16, swapEieio_16, loadSwapEieio_32, storeSwapEieio_32, swapEieio_32, eieio – i/o services

FEATURES

DKI

DESCRIPTION

Provides access to DKI i/o routines, optimized to facilitate byte swapping and synchronization of input/output.

See the architecture specific man pages:

- `loadSwapEieio_16_powerpc`(9DKI)
- `storeSwapEieio_16_powerpc`(9DKI)
- `swapEieio_16_powerpc`(9DKI)
- `loadSwapEieio_32_powerpc`(9DKI)
- `storeSwapEieio_32_powerpc`(9DKI)
- `swapEieio_32_powerpc`(9DKI)
- `eieio`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

loadSwapEieio_16_powerpc, loadSwapSync_16_powerpc, storeSwapEieio_16_powerpc, storeSwapSync_16_powerpc, swapEieio_16_powerpc, loadSwapEieio_32_powerpc, loadSwapSync_32_powerpc, storeSwapEieio_32_powerpc, storeSwapSync_32_powerpc, swapEieio_32_powerpc, eieio_powerpc, ioSync_powerpc – PowerPC specific i/o services

SYNOPSIS

```
#include <dki/dki.h>
uint16_f loadSwapEieio_16(uint16_f * addr);

unit16_f loadSwapSync_16(uint16_f * addr);

void storeSwapEieio_16(uint16_f * addr, uint16_f value);

void storeSwapSync_16(uint16_f * addr, uint16_f value);

void swapEieio_16(uint16_f * addr);

uint32_f loadSwapEieio_32(uint32_f * addr);

unit32_f loadSwapSync_32(uint32_f * addr);

void storeSwapEieio_32(uint32_f * addr, uint32_f value);

void storeSwapSync_32(uint32_f * addr, uint32_f value);

void swapEieio_32 (uint32_f * addr);

void ioSync(void);

void eieio(void);
```

FEATURES

DKI

DESCRIPTION

Provides access to PowerPC DKI I/O routines, optimized to facilitate byte swapping and synchronization of input/output.

EXTENDED DESCRIPTION

The PowerPC DKI provides specific I/O routines to handle byte swapping and synchronization of I/O. These services are based on PowerPC specific instructions to store/load with byte swapping, and to enforce in-order execution of I/O.

Typically, these services are intended to be used by a host bus driver to handle different byte ordering between the PowerPC bus and the host bus, or to enforce sequential execution of I/O. PowerPC specific I/O services are defined below as sets of routines where the _xx suffix indicates the bit length of the data on which the service applies. This suffix may take one of the following values:

_16	for 16-bit data
_32	for 32-bit data

`loadSwapEieio_xx` loads data from a given address and returns the corresponding byte swapped value. A PowerPC `eieio` instruction is issued after the data is read, to enforce execution of subsequent I/O sequentially with respect to the current load. The *addr* argument specifies the address to read from.

`loadSwapSync_xx` behaves like `loadSwapEieio_xx` but issues a PowerPC `isync` instruction after that data is read. This ensures that all possible exceptions are taken into account before returning.

`storeSwapEieio_xx` stores, at a given address, the *value* byte swapped. A PowerPC `eieio` instruction is issued after the data is written, to enforce execution of subsequent I/O sequentially with respect to the current store. The *addr* argument specifies the address to write to.

`storeSwapSync_xx` behaves like `storeSwapEieio_xx` but issues a PowerPC `isync` instruction after that data is read. This ensures that all possible exceptions are taken into account before returning.

`swapEieio_xx` swaps in place the bytes of the data stored at a given address. A PowerPC `eieio` instruction is issued between reading the data and re-writing the byte-swapped data, to enforce execution in order. The *addr* argument specifies the address of the data to byte-swap.

`ioSync` enforces context synchronization, completion of all instructions, and completion of all exceptions before returning.

`eieio` enforces execution in order of I/O and is mapped to the corresponding PowerPC assembler instruction. This routine is intended for use in managing shared data structures, in doing memory-mapped I/O, and in preventing load/store combining operations in main memory. Refer to PowerPC programming manuals for more details.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
loadSwapEieio_xx	+	+	+	-
loadSwapSync_xx	+	+	+	-
storeSwapEieio_xx	+	+	+	-
storeSwapSync_xx	+	+	+	-
swapEieio_xx	+	+	+	-
ioSync	+	+	+	-
eieio	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`loadSwap_16`(9DKI) , `svDkiThreadCall`(9DKI)

NAME

DISABLE_PREEMPT, ENABLE_PREEMPT – thread preemption disabling; thread preemption enabling

SYNOPSIS

```
#include <dki/dki.h>

DISABLE_PREEMPT();

ENABLE_PREEMPT();
```

FEATURES

DKI

DESCRIPTION

DKI provides a means for a driver to disable/enable the preemption of the current thread. These services can be used by a driver to prevent the current thread being preempted while interrupts are masked at bus/device level.

Note - These services are implemented as macros.

EXTENDED DESCRIPTION

Thread Preemption Disabling

The `DISABLE_PREEMPT()` macro disables preemption of the thread which is currently executing. This macro increments a per-processor preemption mask count. When this count is not zero, the scheduler is locked. This occurs as when there is a preemption request, the scheduler simply raises a pending preemption flag deferring the real thread preemption until the preemption mask count drops to zero.

Thread Preemption Enabling

The `ENABLE_PREEMPT()` macro enables preemption of the thread which is currently executing and was previously disabled by `DISABLE_PREEMPT()` as outlined above. This macro decrements the preemption mask count and, if it drops to zero, checks whether the pending preemption flag is raised and thus if the current thread should be preempted.

Note - As `DISABLE_PREEMPT()` / `ENABLE_PREEMPT()` rely on the preemption mask count, a driver may issue nested calls to these services.

Allowed Calling Contexts

The following table specifies the allowed calling contexts for each service:

Services	Base level	DKI thread	Interrupt	Blocking
DISABLE_PREEMPT	+	+	+	-
ENABLE_PREEMPT	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `svDkiThreadCall`(9DKI)

NAME

icacheInval, icacheLineInval, icacheBlockInval, dcacheFlush, dcacheLineFlush, dcacheBlockFlush – cache management

FEATURES

DKI

DESCRIPTION

The microkernel provides cache management services, mainly to allow host bus drivers to manage memory coherency for DMA purposes.

Refer to architecture specific man pages:

- `icacheInval_usparc`(9DKI)
- `icacheLineInval_usparc`(9DKI)
- `icacheBlockInval_usparc`(9DKI)
- `dcacheFlush_usparc`(9DKI)
- `dcacheLineFlush_usparc`(9DKI)
- `dcacheBlockFlush_usparc`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

icacheInval_usparc, icacheLineInval_usparc, icacheBlockInval_usparc, dcacheFlush_usparc, dcacheLineFlush_usparc, dcacheBlockFlush_usparc – UltraSPARC cache management

SYNOPSIS

```
#include <dki/f_dki.h>
void icacheInval(void);

void icacheLineInval(VmAddr addr);

void icacheBlockInval(VmAddr addr, VmSize size);

void dcacheFlush(void);

void dcacheLineFlush(VmAddr addr);

void dcacheBlockFlush(VmAddr addr, VmSize size);
```

FEATURES

DKI

DESCRIPTION

The microkernel provides cache management services, mainly to allow host bus drivers to manage memory coherency for DMA purposes.

EXTENDED DESCRIPTION

UltraSPARC family processors have split instruction and data caches which are directly mapped and virtually indexed. The instruction cache is two-way associative, directly mapped. It is composed of two banks with an LRU replacement mechanism. The data cache is one-way associative, directly mapped and operates in the write-through mode.

The data/instruction caches configuration is available as the `"i-cache"` and `"d-cache"` properties attached to the `"cpu"` . The cpu node may be found in the device tree as a child node of the root node. The cpu node name is `"cpu"` .

The cache configuration property value is the `SparcPropCache` structure shown below.

```
typedef struct {

    uint32_f  csize;   /* cache size */
    uint32_f  bsize;   /* cache bank size */
    uint32_f  lsize;   /* cache line size */
    uint32_f  nbanks;  /* number of bunks (csize = bsize * nbanks) */
    uint32_f  type;    /* cache type */

} SparcPropCache;
```

The *csize* field specifies the cache size in bytes.

The *bsize* field specifies the cache bank size in bytes.

The *lsize* field specifies the cache line size in bytes.

The *nbanks* field specifies the number of banks.

The *type* field specifies the cache properties. It is composed of the following bit-fields and flags:

CACHE_TYPE_IDX	Bit-field specifying the cache indexing mode: CACHE_TYPE_IDX_PHYS — physically indexed CACHE_TYPE_IDX_VIRT — virtually indexed
CACHE_TYPE_TAG	Bit-field specifying the cache tagging mode: CACHE_TYPE_TAG_PHYS — physically tagged CACHE_TYPE_TAG_VIRT — virtually tagged
CACHE_TYPE_MODE	Bit-field specifying the data cache mode: CACHE_TYPE_MODE_WT — write-through CACHE_TYPE_MODE_CB — copy-back
CACHE_TYPE_CFS	Flag specifying that the entire cache flush/invalidation is supported.
CACHE_TYPE_LFS	Flag specifying that line cache flush/invalidation is supported.
CACHE_TYPE_MATCH	Flag specifying that the tag match criteria is used for the line flush/invalidation

Note that, typically, a host bus driver does not need to examine the caches' properties because they are already taken into account by the cache management service routines described below.

`icacheInval` invalidates the entire CPU Instruction Cache.

`icacheLineInval` invalidates a given cache line within the CPU Instruction Cache. The cache line being invalidated is specified by a virtual address.

`icacheBlockInval` invalidates a given range within the CPU Instruction Cache. The range being invalidated is specified by the virtual start address and range size.

`dcacheFlush` flushes and invalidates the entire CPU Data Cache.

`dcacheLineFlush` flushes and invalidates a given cache line within the CPU Data Cache. The cache line being flushed and invalidated is specified by a virtual address.

`dcacheLineFlush` flushes and invalidates a given cache line within the CPU Data Cache. The cache line being flushed and invalidated is specified by a virtual address.

ALLOWED CALLING CONTEXTS

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
icacheInval	+	+	+	-
icacheLineInval	+	+	+	-
icacheBlockInval	+	+	+	-
dcacheFlush	+	+	+	-
dcacheLineFlush	+	+	+	-
dcacheBlockFlush	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

icacheInval, icacheLineInval, icacheBlockInval, dcacheFlush, dcacheLineFlush, dcacheBlockFlush – cache management

FEATURES

DKI

DESCRIPTION

The microkernel provides cache management services, mainly to allow host bus drivers to manage memory coherency for DMA purposes.

Refer to architecture specific man pages:

- `icacheInval_usparc`(9DKI)
- `icacheLineInval_usparc`(9DKI)
- `icacheBlockInval_usparc`(9DKI)
- `dcacheFlush_usparc`(9DKI)
- `dcacheLineFlush_usparc`(9DKI)
- `dcacheBlockFlush_usparc`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

icacheInval_usparc, icacheLineInval_usparc, icacheBlockInval_usparc, dcacheFlush_usparc, dcacheLineFlush_usparc, dcacheBlockFlush_usparc – UltraSPARC cache management

SYNOPSIS

```
#include <dki/f_dki.h>
void icacheInval(void);

void icacheLineInval(VmAddr addr);

void icacheBlockInval(VmAddr addr, VmSize size);

void dcacheFlush(void);

void dcacheLineFlush(VmAddr addr);

void dcacheBlockFlush(VmAddr addr, VmSize size);
```

FEATURES

DKI

DESCRIPTION

The microkernel provides cache management services, mainly to allow host bus drivers to manage memory coherency for DMA purposes.

EXTENDED DESCRIPTION

UltraSPARC family processors have split instruction and data caches which are directly mapped and virtually indexed. The instruction cache is two-way associative, directly mapped. It is composed of two banks with an LRU replacement mechanism. The data cache is one-way associative, directly mapped and operates in the write-through mode.

The data/instruction caches configuration is available as the `"i-cache"` and `"d-cache"` properties attached to the `"cpu"` . The cpu node may be found in the device tree as a child node of the root node. The cpu node name is `"cpu"` .

The cache configuration property value is the `SparcPropCache` structure shown below.

```
typedef struct {

    uint32_f  csize;   /* cache size */
    uint32_f  bsize;   /* cache bank size */
    uint32_f  lsize;   /* cache line size */
    uint32_f  nbanks;  /* number of bunks (csize = bsize * nbanks) */
    uint32_f  type;    /* cache type */

} SparcPropCache;
```

The *csize* field specifies the cache size in bytes.

The *bsize* field specifies the cache bank size in bytes.

The *lsize* field specifies the cache line size in bytes.

The *nbanks* field specifies the number of banks.

The *type* field specifies the cache properties. It is composed of the following bit-fields and flags:

CACHE_TYPE_IDX	Bit-field specifying the cache indexing mode: CACHE_TYPE_IDX_PHYS — physically indexed CACHE_TYPE_IDX_VIRT — virtually indexed
CACHE_TYPE_TAG	Bit-field specifying the cache tagging mode: CACHE_TYPE_TAG_PHYS — physically tagged CACHE_TYPE_TAG_VIRT — virtually tagged
CACHE_TYPE_MODE	Bit-field specifying the data cache mode: CACHE_TYPE_MODE_WT — write-through CACHE_TYPE_MODE_CB — copy-back
CACHE_TYPE_CFS	Flag specifying that the entire cache flush/invalidation is supported.
CACHE_TYPE_LFS	Flag specifying that line cache flush/invalidation is supported.
CACHE_TYPE_MATCH	Flag specifying that the tag match criteria is used for the line flush/invalidation

Note that, typically, a host bus driver does not need to examine the caches' properties because they are already taken into account by the cache management service routines described below.

`icacheInval` invalidates the entire CPU Instruction Cache.

`icacheLineInval` invalidates a given cache line within the CPU Instruction Cache. The cache line being invalidated is specified by a virtual address.

`icacheBlockInval` invalidates a given range within the CPU Instruction Cache. The range being invalidated is specified by the virtual start address and range size.

`dcacheFlush` flushes and invalidates the entire CPU Data Cache.

`dcacheLineFlush` flushes and invalidates a given cache line within the CPU Data Cache. The cache line being flushed and invalidated is specified by a virtual address.

`dcacheLineFlush` flushes and invalidates a given cache line within the CPU Data Cache. The cache line being flushed and invalidated is specified by a virtual address.

ALLOWED CALLING CONTEXTS

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
icacheInval	+	+	+	-
icacheLineInval	+	+	+	-
icacheBlockInval	+	+	+	-
dcacheFlush	+	+	+	-
dcacheLineFlush	+	+	+	-
dcacheBlockFlush	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

icacheInval, icacheLineInval, icacheBlockInval, dcacheFlush, dcacheLineFlush, dcacheBlockFlush – cache management

FEATURES

DKI

DESCRIPTION

The microkernel provides cache management services, mainly to allow host bus drivers to manage memory coherency for DMA purposes.

Refer to architecture specific man pages:

- `icacheInval_usparc`(9DKI)
- `icacheLineInval_usparc`(9DKI)
- `icacheBlockInval_usparc`(9DKI)
- `dcacheFlush_usparc`(9DKI)
- `dcacheLineFlush_usparc`(9DKI)
- `dcacheBlockFlush_usparc`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

icacheInval_usparc, icacheLineInval_usparc, icacheBlockInval_usparc, dcacheFlush_usparc, dcacheLineFlush_usparc, dcacheBlockFlush_usparc – UltraSPARC cache management

SYNOPSIS

```
#include <dki/f_dki.h>
void icacheInval(void);

void icacheLineInval(VmAddr addr);

void icacheBlockInval(VmAddr addr, VmSize size);

void dcacheFlush(void);

void dcacheLineFlush(VmAddr addr);

void dcacheBlockFlush(VmAddr addr, VmSize size);
```

FEATURES

DKI

DESCRIPTION

The microkernel provides cache management services, mainly to allow host bus drivers to manage memory coherency for DMA purposes.

EXTENDED DESCRIPTION

UltraSPARC family processors have split instruction and data caches which are directly mapped and virtually indexed. The instruction cache is two-way associative, directly mapped. It is composed of two banks with an LRU replacement mechanism. The data cache is one-way associative, directly mapped and operates in the write-through mode.

The data/instruction caches configuration is available as the `"i-cache"` and `"d-cache"` properties attached to the `"cpu"` . The cpu node may be found in the device tree as a child node of the root node. The cpu node name is `"cpu"` .

The cache configuration property value is the `SparcPropCache` structure shown below.

```
typedef struct {

    uint32_f  csize;   /* cache size */
    uint32_f  bsize;   /* cache bank size */
    uint32_f  lsize;   /* cache line size */
    uint32_f  nbanks;  /* number of bunks (csize = bsize * nbanks) */
    uint32_f  type;    /* cache type */

} SparcPropCache;
```

The *csize* field specifies the cache size in bytes.

The *bsize* field specifies the cache bank size in bytes.

The *lsize* field specifies the cache line size in bytes.

The *nbanks* field specifies the number of banks.

The *type* field specifies the cache properties. It is composed of the following bit-fields and flags:

CACHE_TYPE_IDX	Bit-field specifying the cache indexing mode: CACHE_TYPE_IDX_PHYS — physically indexed CACHE_TYPE_IDX_VIRT — virtually indexed
CACHE_TYPE_TAG	Bit-field specifying the cache tagging mode: CACHE_TYPE_TAG_PHYS — physically tagged CACHE_TYPE_TAG_VIRT — virtually tagged
CACHE_TYPE_MODE	Bit-field specifying the data cache mode: CACHE_TYPE_MODE_WT — write-through CACHE_TYPE_MODE_CB — copy-back
CACHE_TYPE_CFS	Flag specifying that the entire cache flush/invalidation is supported.
CACHE_TYPE_LFS	Flag specifying that line cache flush/invalidation is supported.
CACHE_TYPE_MATCH	Flag specifying that the tag match criteria is used for the line flush/invalidation

Note that, typically, a host bus driver does not need to examine the caches' properties because they are already taken into account by the cache management service routines described below.

`icacheInval` invalidates the entire CPU Instruction Cache.

`icacheLineInval` invalidates a given cache line within the CPU Instruction Cache. The cache line being invalidated is specified by a virtual address.

`icacheBlockInval` invalidates a given range within the CPU Instruction Cache. The range being invalidated is specified by the virtual start address and range size.

`dcacheFlush` flushes and invalidates the entire CPU Data Cache.

`dcacheLineFlush` flushes and invalidates a given cache line within the CPU Data Cache. The cache line being flushed and invalidated is specified by a virtual address.

`dcacheLineFlush` flushes and invalidates a given cache line within the CPU Data Cache. The cache line being flushed and invalidated is specified by a virtual address.

ALLOWED CALLING CONTEXTS

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
icacheInval	+	+	+	-
icacheLineInval	+	+	+	-
icacheBlockInval	+	+	+	-
dcacheFlush	+	+	+	-
dcacheLineFlush	+	+	+	-
dcacheBlockFlush	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

imsIntrMask_f, imsIntrUnmask_f – global interrupts masking

SYNOPSIS

```
#include <dki/dki.h>
void imsIntrMask_f (void);

void imsIntrUnmask_f (void);
```

FEATURES

DKI

DESCRIPTION

Provides global interrupts masking operations.

EXTENDED DESCRIPTION

Some of the Interrupt Management Service (IMS) routines are part of the DKI in order to provide drivers with global interrupts masking services. These services can be useful for a driver to protect a critical section from interrupts, if necessary.

`imsIntrMask_f` masks all maskable interrupts at processor level, and increments the `imsIntrMaskCount_f` kernel variable.

`imsIntrUnmask_f` decrements the `imsIntrMaskCount_f` kernel variable, and unmasks interrupts at processor level, if `imsIntrMaskCount_f` becomes equal to zero. Note that, as they rely on a kernel interrupt mask count, a driver may issue nested calls to these services.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service.

Services	Base level	DKI thread	Interrupt	Blocking
imsIntrMask_f	+	+	+	-
imsIntrUnmask_f	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

imsIntrMask_f, imsIntrUnmask_f – global interrupts masking

SYNOPSIS

```
#include <dki/dki.h>
void imsIntrMask_f (void);

void imsIntrUnmask_f (void);
```

FEATURES

DKI

DESCRIPTION

Provides global interrupts masking operations.

EXTENDED DESCRIPTION

Some of the Interrupt Management Service (IMS) routines are part of the DKI in order to provide drivers with global interrupts masking services. These services can be useful for a driver to protect a critical section from interrupts, if necessary.

`imsIntrMask_f` masks all maskable interrupts at processor level, and increments the `imsIntrMaskCount_f` kernel variable.

`imsIntrUnmask_f` decrements the `imsIntrMaskCount_f` kernel variable, and unmasks interrupts at processor level, if `imsIntrMaskCount_f` becomes equal to zero. Note that, as they rely on a kernel interrupt mask count, a driver may issue nested calls to these services.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service.

Services	Base level	DKI thread	Interrupt	Blocking
imsIntrMask_f	+	+	+	-
imsIntrUnmask_f	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

dataCacheBlockFlush, dataCacheInvalidate, dataCacheBlockInvalidate, instCacheInvalidate, instCacheBlockInvalidate – cache management

FEATURES

DKI

DESCRIPTION

See the architecture specific man pages:

- `dataCacheBlockFlush_powerpc`(9DKI)
- `dataCacheInvalidate_powerpc`(9DKI)
- `dataCacheBlockInvalidate_powerpc`(9DKI)
- `instCacheInvalidate_powerpc`(9DKI)
- `instCacheBlockInvalidate_powerpc`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

dataCacheBlockFlush_powerpc, dataCacheInvalidate_powerpc, dataCacheBlockInvalidate_powerpc, instCacheInvalidate_powerpc, instCacheBlockInvalidate_powerpc – PowerPC cache management

SYNOPSIS

```
#include <dki/f_dki.h>
void dataCacheFlush(void);

void dataCacheBlockFlush(VmAddr addr, VmSize size);

void dataCacheInvalidate(void);

void dataCacheBlockInvalidate(VmAddr addr, VmSize size);

void instCacheInvalidate(void);

void instCacheBlockInvalidate(VmAddr addr, VmSize size);
```

FEATURES

DKI

DESCRIPTION

Provides PowerPC cache management services.

EXTENDED DESCRIPTION

The microkernel provides cache management services, mainly to allow host bus drivers to manage memory coherency for DMA purposes.

Typically, PowerPC family processors have separate instruction and data caches which are virtually indexed and physically tagged. However, the PowerPC architecture does not specify the type or existence of a cache. This allows for various different cache types (unified, or no cache at all). In any case, cache management services should behave consistently. They should do nothing if there is no cache, and data or instruction invalidation routines should be equivalent for a unified cache.

Note - The data cache may operate in either write-through or copy-back mode, on a per line basis, depending on the cached memory attributes. The data/instruction cache size and data/instruction line size are processor implementation specific.

The data/instruction cache configuration is available from the `PPC_PROP_CACHE` property attached to the `NODE_CPU` node. (The cpu node may be found in the device tree as a child node of the root node. The cpu node name is `NODE_CPU` .)

The cache configuration property value is the `PpcPropCache` structure.

The *blockNumber* field specifies the number of cache blocks in each data or instruction cache.

The *blockSize* field specifies the cache block size in bytes.

The *blockSizeShift* field specifies the number of bits to shift right/left to divide/multiply by the cache block size (cache block size is always a power of 2).

`dataCacheFlush` globally flushes and invalidates the data cache.

`dataCacheBlockFlush` flushes and invalidates a given range of addresses within the CPU data cache. The range being flushed is specified by the virtual start address (within the current MMU context) and the range size.

`dataCacheInvalidate` globally invalidates the data cache. Note that all blocks in the data cache are marked as invalid without writing back any modified lines to memory. This function does nothing if the data cache is disabled.

`dataCacheBlockInvalidate` invalidates a given range of addresses within the CPU data cache. The range being invalidated is specified by the virtual start address (within the current MMU context) and the range size.

Note that invalidated blocks are not written back to memory. Note also that this service should only be used to invalidate a small memory range; that is, a memory range smaller than the cache size. Otherwise, global invalidation should be used.

`instCacheInvalidate` globally invalidates the instruction cache; that is, all blocks in the instruction cache are marked as invalid. This function does nothing if the instruction cache is disabled.

`instCacheBlockInvalidate` invalidates a given range of addresses within the CPU instruction cache. The range being invalidated is specified by the virtual start address (within the current MMU context) and the range size.

Note that this service should only be used to invalidate a small memory range; that is, a memory range smaller than the cache size. Otherwise, global invalidation should be used.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
dataCacheFlush	+	+	+	-
dataCacheBlockFlush	+	+	+	-
dataCacheInvalidate	+	+	+	-
dataCacheBlockInvalidate	+	+	+	-
instCacheInvalidate	+	+	+	-
instCacheBlockInvalidate	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `svDkiThreadCall`(9DKI)

NAME

dataCacheBlockFlush, dataCacheInvalidate, dataCacheBlockInvalidate, instCacheInvalidate, instCacheBlockInvalidate – cache management

FEATURES

DKI

DESCRIPTION

See the architecture specific man pages:

- `dataCacheBlockFlush_powerpc`(9DKI)
- `dataCacheInvalidate_powerpc`(9DKI)
- `dataCacheBlockInvalidate_powerpc`(9DKI)
- `instCacheInvalidate_powerpc`(9DKI)
- `instCacheBlockInvalidate_powerpc`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

dataCacheBlockFlush_powerpc, dataCacheInvalidate_powerpc, dataCacheBlockInvalidate_powerpc, instCacheInvalidate_powerpc, instCacheBlockInvalidate_powerpc – PowerPC cache management

SYNOPSIS

```
#include <dki/f_dki.h>
void dataCacheFlush(void);

void dataCacheBlockFlush(VmAddr addr, VmSize size);

void dataCacheInvalidate(void);

void dataCacheBlockInvalidate(VmAddr addr, VmSize size);

void instCacheInvalidate(void);

void instCacheBlockInvalidate(VmAddr addr, VmSize size);
```

FEATURES

DKI

DESCRIPTION

Provides PowerPC cache management services.

EXTENDED DESCRIPTION

The microkernel provides cache management services, mainly to allow host bus drivers to manage memory coherency for DMA purposes.

Typically, PowerPC family processors have separate instruction and data caches which are virtually indexed and physically tagged. However, the PowerPC architecture does not specify the type or existence of a cache. This allows for various different cache types (unified, or no cache at all). In any case, cache management services should behave consistently. They should do nothing if there is no cache, and data or instruction invalidation routines should be equivalent for a unified cache.

Note - The data cache may operate in either write-through or copy-back mode, on a per line basis, depending on the cached memory attributes. The data/instruction cache size and data/instruction line size are processor implementation specific.

The data/instruction cache configuration is available from the `PPC_PROP_CACHE` property attached to the `NODE_CPU` node. (The cpu node may be found in the device tree as a child node of the root node. The cpu node name is `NODE_CPU` .)

The cache configuration property value is the `PpcPropCache` structure.

The *blockNumber* field specifies the number of cache blocks in each data or instruction cache.

The *blockSize* field specifies the cache block size in bytes.

The *blockSizeShift* field specifies the number of bits to shift right/left to divide/multiply by the cache block size (cache block size is always a power of 2).

`dataCacheFlush` globally flushes and invalidates the data cache.

`dataCacheBlockFlush` flushes and invalidates a given range of addresses within the CPU data cache. The range being flushed is specified by the virtual start address (within the current MMU context) and the range size.

`dataCacheInvalidate` globally invalidates the data cache. Note that all blocks in the data cache are marked as invalid without writing back any modified lines to memory. This function does nothing if the data cache is disabled.

`dataCacheBlockInvalidate` invalidates a given range of addresses within the CPU data cache. The range being invalidated is specified by the virtual start address (within the current MMU context) and the range size.

Note that invalidated blocks are not written back to memory. Note also that this service should only be used to invalidate a small memory range; that is, a memory range smaller than the cache size. Otherwise, global invalidation should be used.

`instCacheInvalidate` globally invalidates the instruction cache; that is, all blocks in the instruction cache are marked as invalid. This function does nothing if the instruction cache is disabled.

`instCacheBlockInvalidate` invalidates a given range of addresses within the CPU instruction cache. The range being invalidated is specified by the virtual start address (within the current MMU context) and the range size.

Note that this service should only be used to invalidate a small memory range; that is, a memory range smaller than the cache size. Otherwise, global invalidation should be used.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
dataCacheFlush	+	+	+	-
dataCacheBlockFlush	+	+	+	-
dataCacheInvalidate	+	+	+	-
dataCacheBlockInvalidate	+	+	+	-
instCacheInvalidate	+	+	+	-
instCacheBlockInvalidate	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `svDkiThreadCall`(9DKI)

NAME

ioLoad8, ioStore8, ioRead8, ioWrite8, ioLoad16, ioStore16, ioRead16, ioWrite16, ioLoad32, ioStore32, ioRead32, ioWrite32 – I/O services

FEATURES

DKI

DESCRIPTION

The DKI provides specific I/O routines which can be used by a host bus driver to implement bus I/O operations.

See the architecture specific man pages:

- `ioLoad8_x86`(9DKI)
- `ioStore8_x86`(9DKI)
- `ioRead8_x86`(9DKI)
- `ioWrite8_x86`(9DKI)
- `ioLoad16_x86`(9DKI)
- `ioStore16_x86`(9DKI)
- `ioRead16_x86`(9DKI)
- `ioWrite16_x86`(9DKI)
- `ioLoad32_x86`(9DKI)
- `ioStore32_x86`(9DKI)
- `ioRead32_x86`(9DKI)
- `ioWrite32_x86`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ioLoad8_x86, ioStore8_x86, ioRead8_x86, ioWrite8_x86, ioLoad16_x86, ioStore16_x86, ioRead16_x86, ioWrite16_x86, ioLoad32_x86, ioStore32_x86, ioRead32_x86, ioWrite32_x86 – Intel x86 specific I/O services

SYNOPSIS

```
#include <dki/dki.h>
uint8_f ioLoad8(uint32_f base, uint32_f offset);

void ioStore8(uint32_f base, uint32_f offset, uint8_f value);

void ioRead8(uint32_f base, uint32_f offset, uint8_f * addr, uint32_f count);

void ioWrite8(uint32_f base, uint32_f offset, uint8_f * addr, uint32_f count);

uint16_f ioLoad16(uint32_f base, uint32_f offset);

void ioStore16(uint32_f base, uint32_f offset, uint16_f value);

void ioRead16(uint32_f base, uint32_f offset, uint16_f * addr, uint32_f count);

void ioWrite16(uint32_f base, uint32_f offset, uint16_f * addr, uint32_f count);

uint32_f ioLoad32(uint32_f base, uint32_f offset);

void ioStore32(uint32_f base, uint32_f offset, uint32_f value);

void ioRead32(uint32_f base, uint32_f offset, uint32_f * addr, uint32_f count);

void ioWrite32(uint32_f base, uint32_f offset, uint32_f * addr, uint32_f count);
```

FEATURES

DKI

EXTENDED DESCRIPTION

The DKI provides specific I/O routines which can be used by a host bus driver to implement bus I/O operations.

Specific I/O services are defined below as sets of routines where the **xx** suffix indicates the bit length of the data to which the service applies. This suffix may take one of the following values:

- 8 for 8-bit data
- 16 for 16-bit data
- 32 for 32-bit data

`ioLoadxx`

The `ioLoadxx` loads data from a given I/O address and returns the read value. The read location is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address.

`ioStorexx`

The `ioStorexx` stores a specified value at a given I/O address. The I/O address written is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *value* argument specifies the value to store (write).

`ioReadxx`

The `ioReadxx` reads data from a given I/O address *count* times and stores values at the memory location pointed to by *addr* .

The read location is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *addr* argument specifies the location where read values will be stored. The *count* argument specifies the desired number of reads.

`ioWritexx`

The `ioWritexx` reads *count* data from the memory location at *addr* and writes them at the I/O address specified by *base* + *offset* .

The read location is specified by the parameter *addr* . The I/O location written is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *count* argument specifies the desired number of writes to perform.

Specific I/O Allowed Calling Contexts

The table below specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
ioLoadxx	+	+	+	-
ioStorexx	+	+	+	-
ioReadxx	+	+	+	-
ioWritexx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

ioLoad8, ioStore8, ioRead8, ioWrite8, ioLoad16, ioStore16, ioRead16, ioWrite16, ioLoad32, ioStore32, ioRead32, ioWrite32 – I/O services

FEATURES

DKI

DESCRIPTION

The DKI provides specific I/O routines which can be used by a host bus driver to implement bus I/O operations.

See the architecture specific man pages:

- `ioLoad8_x86`(9DKI)
- `ioStore8_x86`(9DKI)
- `ioRead8_x86`(9DKI)
- `ioWrite8_x86`(9DKI)
- `ioLoad16_x86`(9DKI)
- `ioStore16_x86`(9DKI)
- `ioRead16_x86`(9DKI)
- `ioWrite16_x86`(9DKI)
- `ioLoad32_x86`(9DKI)
- `ioStore32_x86`(9DKI)
- `ioRead32_x86`(9DKI)
- `ioWrite32_x86`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ioLoad8_x86, ioStore8_x86, ioRead8_x86, ioWrite8_x86, ioLoad16_x86, ioStore16_x86, ioRead16_x86, ioWrite16_x86, ioLoad32_x86, ioStore32_x86, ioRead32_x86, ioWrite32_x86 – Intel x86 specific I/O services

SYNOPSIS

```
#include <dki/dki.h>
uint8_f ioLoad8(uint32_f base, uint32_f offset);

void ioStore8(uint32_f base, uint32_f offset, uint8_f value);

void ioRead8(uint32_f base, uint32_f offset, uint8_f * addr, uint32_f count);

void ioWrite8(uint32_f base, uint32_f offset, uint8_f * addr, uint32_f count);

uint16_f ioLoad16(uint32_f base, uint32_f offset);

void ioStore16(uint32_f base, uint32_f offset, uint16_f value);

void ioRead16(uint32_f base, uint32_f offset, uint16_f * addr, uint32_f count);

void ioWrite16(uint32_f base, uint32_f offset, uint16_f * addr, uint32_f count);

uint32_f ioLoad32(uint32_f base, uint32_f offset);

void ioStore32(uint32_f base, uint32_f offset, uint32_f value);

void ioRead32(uint32_f base, uint32_f offset, uint32_f * addr, uint32_f count);

void ioWrite32(uint32_f base, uint32_f offset, uint32_f * addr, uint32_f count);
```

FEATURES

DKI

EXTENDED DESCRIPTION

The DKI provides specific I/O routines which can be used by a host bus driver to implement bus I/O operations.

Specific I/O services are defined below as sets of routines where the **xx** suffix indicates the bit length of the data to which the service applies. This suffix may take one of the following values:

- 8 for 8-bit data
- 16 for 16-bit data
- 32 for 32-bit data

`ioLoadxx`

The `ioLoadxx` loads data from a given I/O address and returns the read value. The read location is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address.

`ioStorexx`

The `ioStorexx` stores a specified value at a given I/O address. The I/O address written is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *value* argument specifies the value to store (write).

`ioReadxx`

The `ioReadxx` reads data from a given I/O address *count* times and stores values at the memory location pointed to by *addr* .

The read location is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *addr* argument specifies the location where read values will be stored. The *count* argument specifies the desired number of reads.

`ioWritexx`

The `ioWritexx` reads *count* data from the memory location at *addr* and writes them at the I/O address specified by *base* + *offset* .

The read location is specified by the parameter *addr* . The I/O location written is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *count* argument specifies the desired number of writes to perform.

Specific I/O Allowed Calling Contexts

The table below specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
ioLoadxx	+	+	+	-
ioStorexx	+	+	+	-
ioReadxx	+	+	+	-
ioWritexx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

ioLoad8, ioStore8, ioRead8, ioWrite8, ioLoad16, ioStore16, ioRead16, ioWrite16, ioLoad32, ioStore32, ioRead32, ioWrite32 – I/O services

FEATURES

DKI

DESCRIPTION

The DKI provides specific I/O routines which can be used by a host bus driver to implement bus I/O operations.

See the architecture specific man pages:

- `ioLoad8_x86`(9DKI)
- `ioStore8_x86`(9DKI)
- `ioRead8_x86`(9DKI)
- `ioWrite8_x86`(9DKI)
- `ioLoad16_x86`(9DKI)
- `ioStore16_x86`(9DKI)
- `ioRead16_x86`(9DKI)
- `ioWrite16_x86`(9DKI)
- `ioLoad32_x86`(9DKI)
- `ioStore32_x86`(9DKI)
- `ioRead32_x86`(9DKI)
- `ioWrite32_x86`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ioLoad8_x86, ioStore8_x86, ioRead8_x86, ioWrite8_x86, ioLoad16_x86, ioStore16_x86, ioRead16_x86, ioWrite16_x86, ioLoad32_x86, ioStore32_x86, ioRead32_x86, ioWrite32_x86 – Intel x86 specific I/O services

SYNOPSIS

```
#include <dki/dki.h>
uint8_f ioLoad8(uint32_f base, uint32_f offset);

void ioStore8(uint32_f base, uint32_f offset, uint8_f value);

void ioRead8(uint32_f base, uint32_f offset, uint8_f * addr, uint32_f count);

void ioWrite8(uint32_f base, uint32_f offset, uint8_f * addr, uint32_f count);

uint16_f ioLoad16(uint32_f base, uint32_f offset);

void ioStore16(uint32_f base, uint32_f offset, uint16_f value);

void ioRead16(uint32_f base, uint32_f offset, uint16_f * addr, uint32_f count);

void ioWrite16(uint32_f base, uint32_f offset, uint16_f * addr, uint32_f count);

uint32_f ioLoad32(uint32_f base, uint32_f offset);

void ioStore32(uint32_f base, uint32_f offset, uint32_f value);

void ioRead32(uint32_f base, uint32_f offset, uint32_f * addr, uint32_f count);

void ioWrite32(uint32_f base, uint32_f offset, uint32_f * addr, uint32_f count);
```

FEATURES

DKI

EXTENDED DESCRIPTION

The DKI provides specific I/O routines which can be used by a host bus driver to implement bus I/O operations.

Specific I/O services are defined below as sets of routines where the **xx** suffix indicates the bit length of the data to which the service applies. This suffix may take one of the following values:

- 8 for 8-bit data
- 16 for 16-bit data
- 32 for 32-bit data

`ioLoadxx`

The `ioLoadxx` loads data from a given I/O address and returns the read value. The read location is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address.

`ioStorexx`

The `ioStorexx` stores a specified value at a given I/O address. The I/O address written is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *value* argument specifies the value to store (write).

`ioReadxx`

The `ioReadxx` reads data from a given I/O address *count* times and stores values at the memory location pointed to by *addr* .

The read location is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *addr* argument specifies the location where read values will be stored. The *count* argument specifies the desired number of reads.

`ioWritexx`

The `ioWritexx` reads *count* data from the memory location at *addr* and writes them at the I/O address specified by *base* + *offset* .

The read location is specified by the parameter *addr* . The I/O location written is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *count* argument specifies the desired number of writes to perform.

Specific I/O Allowed Calling Contexts

The table below specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
ioLoadxx	+	+	+	-
ioStorexx	+	+	+	-
ioReadxx	+	+	+	-
ioWritexx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

ioLoad8, ioStore8, ioRead8, ioWrite8, ioLoad16, ioStore16, ioRead16, ioWrite16, ioLoad32, ioStore32, ioRead32, ioWrite32 – I/O services

FEATURES

DKI

DESCRIPTION

The DKI provides specific I/O routines which can be used by a host bus driver to implement bus I/O operations.

See the architecture specific man pages:

- `ioLoad8_x86`(9DKI)
- `ioStore8_x86`(9DKI)
- `ioRead8_x86`(9DKI)
- `ioWrite8_x86`(9DKI)
- `ioLoad16_x86`(9DKI)
- `ioStore16_x86`(9DKI)
- `ioRead16_x86`(9DKI)
- `ioWrite16_x86`(9DKI)
- `ioLoad32_x86`(9DKI)
- `ioStore32_x86`(9DKI)
- `ioRead32_x86`(9DKI)
- `ioWrite32_x86`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ioLoad8_x86, ioStore8_x86, ioRead8_x86, ioWrite8_x86, ioLoad16_x86, ioStore16_x86, ioRead16_x86, ioWrite16_x86, ioLoad32_x86, ioStore32_x86, ioRead32_x86, ioWrite32_x86 – Intel x86 specific I/O services

SYNOPSIS

```
#include <dki/dki.h>
uint8_f ioLoad8(uint32_f base, uint32_f offset);

void ioStore8(uint32_f base, uint32_f offset, uint8_f value);

void ioRead8(uint32_f base, uint32_f offset, uint8_f * addr, uint32_f count);

void ioWrite8(uint32_f base, uint32_f offset, uint8_f * addr, uint32_f count);

uint16_f ioLoad16(uint32_f base, uint32_f offset);

void ioStore16(uint32_f base, uint32_f offset, uint16_f value);

void ioRead16(uint32_f base, uint32_f offset, uint16_f * addr, uint32_f count);

void ioWrite16(uint32_f base, uint32_f offset, uint16_f * addr, uint32_f count);

uint32_f ioLoad32(uint32_f base, uint32_f offset);

void ioStore32(uint32_f base, uint32_f offset, uint32_f value);

void ioRead32(uint32_f base, uint32_f offset, uint32_f * addr, uint32_f count);

void ioWrite32(uint32_f base, uint32_f offset, uint32_f * addr, uint32_f count);
```

FEATURES

DKI

EXTENDED DESCRIPTION

The DKI provides specific I/O routines which can be used by a host bus driver to implement bus I/O operations.

Specific I/O services are defined below as sets of routines where the **xx** suffix indicates the bit length of the data to which the service applies. This suffix may take one of the following values:

- 8 for 8-bit data
- 16 for 16-bit data
- 32 for 32-bit data

`ioLoadxx`

The `ioLoadxx` loads data from a given I/O address and returns the read value. The read location is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address.

`ioStorexx`

The `ioStorexx` stores a specified value at a given I/O address. The I/O address written is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *value* argument specifies the value to store (write).

`ioReadxx`

The `ioReadxx` reads data from a given I/O address *count* times and stores values at the memory location pointed to by *addr* .

The read location is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *addr* argument specifies the location where read values will be stored. The *count* argument specifies the desired number of reads.

`ioWritexx`

The `ioWritexx` reads *count* data from the memory location at *addr* and writes them at the I/O address specified by *base* + *offset* .

The read location is specified by the parameter *addr* . The I/O location written is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *count* argument specifies the desired number of writes to perform.

Specific I/O Allowed Calling Contexts

The table below specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
ioLoadxx	+	+	+	-
ioStorexx	+	+	+	-
ioReadxx	+	+	+	-
ioWritexx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

ioLoad8, ioStore8, ioRead8, ioWrite8, ioLoad16, ioStore16, ioRead16, ioWrite16, ioLoad32, ioStore32, ioRead32, ioWrite32 – I/O services

FEATURES

DKI

DESCRIPTION

The DKI provides specific I/O routines which can be used by a host bus driver to implement bus I/O operations.

See the architecture specific man pages:

- `ioLoad8_x86`(9DKI)
- `ioStore8_x86`(9DKI)
- `ioRead8_x86`(9DKI)
- `ioWrite8_x86`(9DKI)
- `ioLoad16_x86`(9DKI)
- `ioStore16_x86`(9DKI)
- `ioRead16_x86`(9DKI)
- `ioWrite16_x86`(9DKI)
- `ioLoad32_x86`(9DKI)
- `ioStore32_x86`(9DKI)
- `ioRead32_x86`(9DKI)
- `ioWrite32_x86`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ioLoad8_x86, ioStore8_x86, ioRead8_x86, ioWrite8_x86, ioLoad16_x86, ioStore16_x86, ioRead16_x86, ioWrite16_x86, ioLoad32_x86, ioStore32_x86, ioRead32_x86, ioWrite32_x86 – Intel x86 specific I/O services

SYNOPSIS

```
#include <dki/dki.h>
uint8_f ioLoad8(uint32_f base, uint32_f offset);

void ioStore8(uint32_f base, uint32_f offset, uint8_f value);

void ioRead8(uint32_f base, uint32_f offset, uint8_f * addr, uint32_f count);

void ioWrite8(uint32_f base, uint32_f offset, uint8_f * addr, uint32_f count);

uint16_f ioLoad16(uint32_f base, uint32_f offset);

void ioStore16(uint32_f base, uint32_f offset, uint16_f value);

void ioRead16(uint32_f base, uint32_f offset, uint16_f * addr, uint32_f count);

void ioWrite16(uint32_f base, uint32_f offset, uint16_f * addr, uint32_f count);

uint32_f ioLoad32(uint32_f base, uint32_f offset);

void ioStore32(uint32_f base, uint32_f offset, uint32_f value);

void ioRead32(uint32_f base, uint32_f offset, uint32_f * addr, uint32_f count);

void ioWrite32(uint32_f base, uint32_f offset, uint32_f * addr, uint32_f count);
```

FEATURES

DKI

EXTENDED DESCRIPTION

The DKI provides specific I/O routines which can be used by a host bus driver to implement bus I/O operations.

Specific I/O services are defined below as sets of routines where the **xx** suffix indicates the bit length of the data to which the service applies. This suffix may take one of the following values:

- 8 for 8-bit data
- 16 for 16-bit data
- 32 for 32-bit data

`ioLoadxx`

The `ioLoadxx` loads data from a given I/O address and returns the read value. The read location is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address.

`ioStorexx`

The `ioStorexx` stores a specified value at a given I/O address. The I/O address written is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *value* argument specifies the value to store (write).

`ioReadxx`

The `ioReadxx` reads data from a given I/O address *count* times and stores values at the memory location pointed to by *addr* .

The read location is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *addr* argument specifies the location where read values will be stored. The *count* argument specifies the desired number of reads.

`ioWritexx`

The `ioWritexx` reads *count* data from the memory location at *addr* and writes them at the I/O address specified by *base* + *offset* .

The read location is specified by the parameter *addr* . The I/O location written is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *count* argument specifies the desired number of writes to perform.

Specific I/O Allowed Calling Contexts

The table below specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
ioLoadxx	+	+	+	-
ioStorexx	+	+	+	-
ioReadxx	+	+	+	-
ioWritexx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

ioLoad8, ioStore8, ioRead8, ioWrite8, ioLoad16, ioStore16, ioRead16, ioWrite16, ioLoad32, ioStore32, ioRead32, ioWrite32 – I/O services

FEATURES

DKI

DESCRIPTION

The DKI provides specific I/O routines which can be used by a host bus driver to implement bus I/O operations.

See the architecture specific man pages:

- `ioLoad8_x86`(9DKI)
- `ioStore8_x86`(9DKI)
- `ioRead8_x86`(9DKI)
- `ioWrite8_x86`(9DKI)
- `ioLoad16_x86`(9DKI)
- `ioStore16_x86`(9DKI)
- `ioRead16_x86`(9DKI)
- `ioWrite16_x86`(9DKI)
- `ioLoad32_x86`(9DKI)
- `ioStore32_x86`(9DKI)
- `ioRead32_x86`(9DKI)
- `ioWrite32_x86`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ioLoad8_x86, ioStore8_x86, ioRead8_x86, ioWrite8_x86, ioLoad16_x86, ioStore16_x86, ioRead16_x86, ioWrite16_x86, ioLoad32_x86, ioStore32_x86, ioRead32_x86, ioWrite32_x86 – Intel x86 specific I/O services

SYNOPSIS

```
#include <dki/dki.h>
uint8_f ioLoad8(uint32_f base, uint32_f offset);

void ioStore8(uint32_f base, uint32_f offset, uint8_f value);

void ioRead8(uint32_f base, uint32_f offset, uint8_f * addr, uint32_f count);

void ioWrite8(uint32_f base, uint32_f offset, uint8_f * addr, uint32_f count);

uint16_f ioLoad16(uint32_f base, uint32_f offset);

void ioStore16(uint32_f base, uint32_f offset, uint16_f value);

void ioRead16(uint32_f base, uint32_f offset, uint16_f * addr, uint32_f count);

void ioWrite16(uint32_f base, uint32_f offset, uint16_f * addr, uint32_f count);

uint32_f ioLoad32(uint32_f base, uint32_f offset);

void ioStore32(uint32_f base, uint32_f offset, uint32_f value);

void ioRead32(uint32_f base, uint32_f offset, uint32_f * addr, uint32_f count);

void ioWrite32(uint32_f base, uint32_f offset, uint32_f * addr, uint32_f count);
```

FEATURES

DKI

EXTENDED DESCRIPTION

The DKI provides specific I/O routines which can be used by a host bus driver to implement bus I/O operations.

Specific I/O services are defined below as sets of routines where the **xx** suffix indicates the bit length of the data to which the service applies. This suffix may take one of the following values:

- 8 for 8-bit data
- 16 for 16-bit data
- 32 for 32-bit data

`ioLoadxx`

The `ioLoadxx` loads data from a given I/O address and returns the read value. The read location is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address.

`ioStorexx`

The `ioStorexx` stores a specified value at a given I/O address. The I/O address written is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *value* argument specifies the value to store (write).

`ioReadxx`

The `ioReadxx` reads data from a given I/O address *count* times and stores values at the memory location pointed to by *addr* .

The read location is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *addr* argument specifies the location where read values will be stored. The *count* argument specifies the desired number of reads.

`ioWritexx`

The `ioWritexx` reads *count* data from the memory location at *addr* and writes them at the I/O address specified by *base* + *offset* .

The read location is specified by the parameter *addr* . The I/O location written is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *count* argument specifies the desired number of writes to perform.

Specific I/O Allowed Calling Contexts

The table below specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
ioLoadxx	+	+	+	-
ioStorexx	+	+	+	-
ioReadxx	+	+	+	-
ioWritexx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

ioLoad8, ioStore8, ioRead8, ioWrite8, ioLoad16, ioStore16, ioRead16, ioWrite16, ioLoad32, ioStore32, ioRead32, ioWrite32 – I/O services

FEATURES

DKI

DESCRIPTION

The DKI provides specific I/O routines which can be used by a host bus driver to implement bus I/O operations.

See the architecture specific man pages:

- `ioLoad8_x86`(9DKI)
- `ioStore8_x86`(9DKI)
- `ioRead8_x86`(9DKI)
- `ioWrite8_x86`(9DKI)
- `ioLoad16_x86`(9DKI)
- `ioStore16_x86`(9DKI)
- `ioRead16_x86`(9DKI)
- `ioWrite16_x86`(9DKI)
- `ioLoad32_x86`(9DKI)
- `ioStore32_x86`(9DKI)
- `ioRead32_x86`(9DKI)
- `ioWrite32_x86`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ioLoad8_x86, ioStore8_x86, ioRead8_x86, ioWrite8_x86, ioLoad16_x86, ioStore16_x86, ioRead16_x86, ioWrite16_x86, ioLoad32_x86, ioStore32_x86, ioRead32_x86, ioWrite32_x86 – Intel x86 specific I/O services

SYNOPSIS

```
#include <dki/dki.h>
uint8_f ioLoad8(uint32_f base, uint32_f offset);

void ioStore8(uint32_f base, uint32_f offset, uint8_f value);

void ioRead8(uint32_f base, uint32_f offset, uint8_f * addr, uint32_f count);

void ioWrite8(uint32_f base, uint32_f offset, uint8_f * addr, uint32_f count);

uint16_f ioLoad16(uint32_f base, uint32_f offset);

void ioStore16(uint32_f base, uint32_f offset, uint16_f value);

void ioRead16(uint32_f base, uint32_f offset, uint16_f * addr, uint32_f count);

void ioWrite16(uint32_f base, uint32_f offset, uint16_f * addr, uint32_f count);

uint32_f ioLoad32(uint32_f base, uint32_f offset);

void ioStore32(uint32_f base, uint32_f offset, uint32_f value);

void ioRead32(uint32_f base, uint32_f offset, uint32_f * addr, uint32_f count);

void ioWrite32(uint32_f base, uint32_f offset, uint32_f * addr, uint32_f count);
```

FEATURES

DKI

EXTENDED DESCRIPTION

The DKI provides specific I/O routines which can be used by a host bus driver to implement bus I/O operations.

Specific I/O services are defined below as sets of routines where the **xx** suffix indicates the bit length of the data to which the service applies. This suffix may take one of the following values:

- 8 for 8-bit data
- 16 for 16-bit data
- 32 for 32-bit data

`ioLoadxx`

The `ioLoadxx` loads data from a given I/O address and returns the read value. The read location is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address.

`ioStorexx`

The `ioStorexx` stores a specified value at a given I/O address. The I/O address written is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *value* argument specifies the value to store (write).

`ioReadxx`

The `ioReadxx` reads data from a given I/O address *count* times and stores values at the memory location pointed to by *addr* .

The read location is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *addr* argument specifies the location where read values will be stored. The *count* argument specifies the desired number of reads.

`ioWritexx`

The `ioWritexx` reads *count* data from the memory location at *addr* and writes them at the I/O address specified by *base* + *offset* .

The read location is specified by the parameter *addr* . The I/O location written is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *count* argument specifies the desired number of writes to perform.

Specific I/O Allowed Calling Contexts

The table below specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
ioLoadxx	+	+	+	-
ioStorexx	+	+	+	-
ioReadxx	+	+	+	-
ioWritexx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

ioLoad8, ioStore8, ioRead8, ioWrite8, ioLoad16, ioStore16, ioRead16, ioWrite16, ioLoad32, ioStore32, ioRead32, ioWrite32 – I/O services

FEATURES

DKI

DESCRIPTION

The DKI provides specific I/O routines which can be used by a host bus driver to implement bus I/O operations.

See the architecture specific man pages:

- `ioLoad8_x86`(9DKI)
- `ioStore8_x86`(9DKI)
- `ioRead8_x86`(9DKI)
- `ioWrite8_x86`(9DKI)
- `ioLoad16_x86`(9DKI)
- `ioStore16_x86`(9DKI)
- `ioRead16_x86`(9DKI)
- `ioWrite16_x86`(9DKI)
- `ioLoad32_x86`(9DKI)
- `ioStore32_x86`(9DKI)
- `ioRead32_x86`(9DKI)
- `ioWrite32_x86`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ioLoad8_x86, ioStore8_x86, ioRead8_x86, ioWrite8_x86, ioLoad16_x86, ioStore16_x86, ioRead16_x86, ioWrite16_x86, ioLoad32_x86, ioStore32_x86, ioRead32_x86, ioWrite32_x86 – Intel x86 specific I/O services

SYNOPSIS

```
#include <dki/dki.h>
uint8_f ioLoad8(uint32_f base, uint32_f offset);

void ioStore8(uint32_f base, uint32_f offset, uint8_f value);

void ioRead8(uint32_f base, uint32_f offset, uint8_f * addr, uint32_f count);

void ioWrite8(uint32_f base, uint32_f offset, uint8_f * addr, uint32_f count);

uint16_f ioLoad16(uint32_f base, uint32_f offset);

void ioStore16(uint32_f base, uint32_f offset, uint16_f value);

void ioRead16(uint32_f base, uint32_f offset, uint16_f * addr, uint32_f count);

void ioWrite16(uint32_f base, uint32_f offset, uint16_f * addr, uint32_f count);

uint32_f ioLoad32(uint32_f base, uint32_f offset);

void ioStore32(uint32_f base, uint32_f offset, uint32_f value);

void ioRead32(uint32_f base, uint32_f offset, uint32_f * addr, uint32_f count);

void ioWrite32(uint32_f base, uint32_f offset, uint32_f * addr, uint32_f count);
```

FEATURES

DKI

EXTENDED DESCRIPTION

The DKI provides specific I/O routines which can be used by a host bus driver to implement bus I/O operations.

Specific I/O services are defined below as sets of routines where the **xx** suffix indicates the bit length of the data to which the service applies. This suffix may take one of the following values:

- 8 for 8-bit data
- 16 for 16-bit data
- 32 for 32-bit data

`ioLoadxx`

The `ioLoadxx` loads data from a given I/O address and returns the read value. The read location is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address.

`ioStorexx`

The `ioStorexx` stores a specified value at a given I/O address. The I/O address written is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *value* argument specifies the value to store (write).

`ioReadxx`

The `ioReadxx` reads data from a given I/O address *count* times and stores values at the memory location pointed to by *addr* .

The read location is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *addr* argument specifies the location where read values will be stored. The *count* argument specifies the desired number of reads.

`ioWritexx`

The `ioWritexx` reads *count* data from the memory location at *addr* and writes them at the I/O address specified by *base* + *offset* .

The read location is specified by the parameter *addr* . The I/O location written is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *count* argument specifies the desired number of writes to perform.

Specific I/O Allowed Calling Contexts

The table below specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
ioLoadxx	+	+	+	-
ioStorexx	+	+	+	-
ioReadxx	+	+	+	-
ioWritexx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

ioLoad8, ioStore8, ioRead8, ioWrite8, ioLoad16, ioStore16, ioRead16, ioWrite16, ioLoad32, ioStore32, ioRead32, ioWrite32 – I/O services

FEATURES

DKI

DESCRIPTION

The DKI provides specific I/O routines which can be used by a host bus driver to implement bus I/O operations.

See the architecture specific man pages:

- `ioLoad8_x86`(9DKI)
- `ioStore8_x86`(9DKI)
- `ioRead8_x86`(9DKI)
- `ioWrite8_x86`(9DKI)
- `ioLoad16_x86`(9DKI)
- `ioStore16_x86`(9DKI)
- `ioRead16_x86`(9DKI)
- `ioWrite16_x86`(9DKI)
- `ioLoad32_x86`(9DKI)
- `ioStore32_x86`(9DKI)
- `ioRead32_x86`(9DKI)
- `ioWrite32_x86`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ioLoad8_x86, ioStore8_x86, ioRead8_x86, ioWrite8_x86, ioLoad16_x86, ioStore16_x86, ioRead16_x86, ioWrite16_x86, ioLoad32_x86, ioStore32_x86, ioRead32_x86, ioWrite32_x86 – Intel x86 specific I/O services

SYNOPSIS

```
#include <dki/dki.h>
uint8_f ioLoad8(uint32_f base, uint32_f offset);

void ioStore8(uint32_f base, uint32_f offset, uint8_f value);

void ioRead8(uint32_f base, uint32_f offset, uint8_f * addr, uint32_f count);

void ioWrite8(uint32_f base, uint32_f offset, uint8_f * addr, uint32_f count);

uint16_f ioLoad16(uint32_f base, uint32_f offset);

void ioStore16(uint32_f base, uint32_f offset, uint16_f value);

void ioRead16(uint32_f base, uint32_f offset, uint16_f * addr, uint32_f count);

void ioWrite16(uint32_f base, uint32_f offset, uint16_f * addr, uint32_f count);

uint32_f ioLoad32(uint32_f base, uint32_f offset);

void ioStore32(uint32_f base, uint32_f offset, uint32_f value);

void ioRead32(uint32_f base, uint32_f offset, uint32_f * addr, uint32_f count);

void ioWrite32(uint32_f base, uint32_f offset, uint32_f * addr, uint32_f count);
```

FEATURES

DKI

EXTENDED DESCRIPTION

The DKI provides specific I/O routines which can be used by a host bus driver to implement bus I/O operations.

Specific I/O services are defined below as sets of routines where the **xx** suffix indicates the bit length of the data to which the service applies. This suffix may take one of the following values:

- 8 for 8-bit data
- 16 for 16-bit data
- 32 for 32-bit data

`ioLoadxx`

The `ioLoadxx` loads data from a given I/O address and returns the read value. The read location is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address.

`ioStorexx`

The `ioStorexx` stores a specified value at a given I/O address. The I/O address written is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *value* argument specifies the value to store (write).

`ioReadxx`

The `ioReadxx` reads data from a given I/O address *count* times and stores values at the memory location pointed to by *addr* .

The read location is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *addr* argument specifies the location where read values will be stored. The *count* argument specifies the desired number of reads.

`ioWritexx`

The `ioWritexx` reads *count* data from the memory location at *addr* and writes them at the I/O address specified by *base* + *offset* .

The read location is specified by the parameter *addr* . The I/O location written is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *count* argument specifies the desired number of writes to perform.

Specific I/O Allowed Calling Contexts

The table below specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
ioLoadxx	+	+	+	-
ioStorexx	+	+	+	-
ioReadxx	+	+	+	-
ioWritexx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

ioLoad8, ioStore8, ioRead8, ioWrite8, ioLoad16, ioStore16, ioRead16, ioWrite16, ioLoad32, ioStore32, ioRead32, ioWrite32 – I/O services

FEATURES

DKI

DESCRIPTION

The DKI provides specific I/O routines which can be used by a host bus driver to implement bus I/O operations.

See the architecture specific man pages:

- `ioLoad8_x86`(9DKI)
- `ioStore8_x86`(9DKI)
- `ioRead8_x86`(9DKI)
- `ioWrite8_x86`(9DKI)
- `ioLoad16_x86`(9DKI)
- `ioStore16_x86`(9DKI)
- `ioRead16_x86`(9DKI)
- `ioWrite16_x86`(9DKI)
- `ioLoad32_x86`(9DKI)
- `ioStore32_x86`(9DKI)
- `ioRead32_x86`(9DKI)
- `ioWrite32_x86`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ioLoad8_x86, ioStore8_x86, ioRead8_x86, ioWrite8_x86, ioLoad16_x86, ioStore16_x86, ioRead16_x86, ioWrite16_x86, ioLoad32_x86, ioStore32_x86, ioRead32_x86, ioWrite32_x86 – Intel x86 specific I/O services

SYNOPSIS

```
#include <dki/dki.h>
uint8_f ioLoad8(uint32_f base, uint32_f offset);

void ioStore8(uint32_f base, uint32_f offset, uint8_f value);

void ioRead8(uint32_f base, uint32_f offset, uint8_f * addr, uint32_f count);

void ioWrite8(uint32_f base, uint32_f offset, uint8_f * addr, uint32_f count);

uint16_f ioLoad16(uint32_f base, uint32_f offset);

void ioStore16(uint32_f base, uint32_f offset, uint16_f value);

void ioRead16(uint32_f base, uint32_f offset, uint16_f * addr, uint32_f count);

void ioWrite16(uint32_f base, uint32_f offset, uint16_f * addr, uint32_f count);

uint32_f ioLoad32(uint32_f base, uint32_f offset);

void ioStore32(uint32_f base, uint32_f offset, uint32_f value);

void ioRead32(uint32_f base, uint32_f offset, uint32_f * addr, uint32_f count);

void ioWrite32(uint32_f base, uint32_f offset, uint32_f * addr, uint32_f count);
```

FEATURES

DKI

EXTENDED DESCRIPTION

The DKI provides specific I/O routines which can be used by a host bus driver to implement bus I/O operations.

Specific I/O services are defined below as sets of routines where the **xx** suffix indicates the bit length of the data to which the service applies. This suffix may take one of the following values:

- 8 for 8-bit data
- 16 for 16-bit data
- 32 for 32-bit data

`ioLoadxx`

The `ioLoadxx` loads data from a given I/O address and returns the read value. The read location is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address.

`ioStorexx`

The `ioStorexx` stores a specified value at a given I/O address. The I/O address written is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *value* argument specifies the value to store (write).

`ioReadxx`

The `ioReadxx` reads data from a given I/O address *count* times and stores values at the memory location pointed to by *addr* .

The read location is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *addr* argument specifies the location where read values will be stored. The *count* argument specifies the desired number of reads.

`ioWritexx`

The `ioWritexx` reads *count* data from the memory location at *addr* and writes them at the I/O address specified by *base* + *offset* .

The read location is specified by the parameter *addr* . The I/O location written is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *count* argument specifies the desired number of writes to perform.

Specific I/O Allowed Calling Contexts

The table below specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
ioLoadxx	+	+	+	-
ioStorexx	+	+	+	-
ioReadxx	+	+	+	-
ioWritexx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

ioLoad8, ioStore8, ioRead8, ioWrite8, ioLoad16, ioStore16, ioRead16, ioWrite16, ioLoad32, ioStore32, ioRead32, ioWrite32 – I/O services

FEATURES

DKI

DESCRIPTION

The DKI provides specific I/O routines which can be used by a host bus driver to implement bus I/O operations.

See the architecture specific man pages:

- `ioLoad8_x86`(9DKI)
- `ioStore8_x86`(9DKI)
- `ioRead8_x86`(9DKI)
- `ioWrite8_x86`(9DKI)
- `ioLoad16_x86`(9DKI)
- `ioStore16_x86`(9DKI)
- `ioRead16_x86`(9DKI)
- `ioWrite16_x86`(9DKI)
- `ioLoad32_x86`(9DKI)
- `ioStore32_x86`(9DKI)
- `ioRead32_x86`(9DKI)
- `ioWrite32_x86`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ioLoad8_x86, ioStore8_x86, ioRead8_x86, ioWrite8_x86, ioLoad16_x86, ioStore16_x86, ioRead16_x86, ioWrite16_x86, ioLoad32_x86, ioStore32_x86, ioRead32_x86, ioWrite32_x86 – Intel x86 specific I/O services

SYNOPSIS

```
#include <dki/dki.h>
uint8_f ioLoad8(uint32_f base, uint32_f offset);

void ioStore8(uint32_f base, uint32_f offset, uint8_f value);

void ioRead8(uint32_f base, uint32_f offset, uint8_f * addr, uint32_f count);

void ioWrite8(uint32_f base, uint32_f offset, uint8_f * addr, uint32_f count);

uint16_f ioLoad16(uint32_f base, uint32_f offset);

void ioStore16(uint32_f base, uint32_f offset, uint16_f value);

void ioRead16(uint32_f base, uint32_f offset, uint16_f * addr, uint32_f count);

void ioWrite16(uint32_f base, uint32_f offset, uint16_f * addr, uint32_f count);

uint32_f ioLoad32(uint32_f base, uint32_f offset);

void ioStore32(uint32_f base, uint32_f offset, uint32_f value);

void ioRead32(uint32_f base, uint32_f offset, uint32_f * addr, uint32_f count);

void ioWrite32(uint32_f base, uint32_f offset, uint32_f * addr, uint32_f count);
```

FEATURES

DKI

EXTENDED DESCRIPTION

The DKI provides specific I/O routines which can be used by a host bus driver to implement bus I/O operations.

Specific I/O services are defined below as sets of routines where the **xx** suffix indicates the bit length of the data to which the service applies. This suffix may take one of the following values:

- 8 for 8-bit data
- 16 for 16-bit data
- 32 for 32-bit data

`ioLoadxx`

The `ioLoadxx` loads data from a given I/O address and returns the read value. The read location is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address.

`ioStorexx`

The `ioStorexx` stores a specified value at a given I/O address. The I/O address written is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *value* argument specifies the value to store (write).

`ioReadxx`

The `ioReadxx` reads data from a given I/O address *count* times and stores values at the memory location pointed to by *addr* .

The read location is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *addr* argument specifies the location where read values will be stored. The *count* argument specifies the desired number of reads.

`ioWritexx`

The `ioWritexx` reads *count* data from the memory location at *addr* and writes them at the I/O address specified by *base* + *offset* .

The read location is specified by the parameter *addr* . The I/O location written is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *count* argument specifies the desired number of writes to perform.

Specific I/O Allowed Calling Contexts

The table below specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
ioLoadxx	+	+	+	-
ioStorexx	+	+	+	-
ioReadxx	+	+	+	-
ioWritexx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

ioLoad8, ioStore8, ioRead8, ioWrite8, ioLoad16, ioStore16, ioRead16, ioWrite16, ioLoad32, ioStore32, ioRead32, ioWrite32 – I/O services

FEATURES

DKI

DESCRIPTION

The DKI provides specific I/O routines which can be used by a host bus driver to implement bus I/O operations.

See the architecture specific man pages:

- `ioLoad8_x86`(9DKI)
- `ioStore8_x86`(9DKI)
- `ioRead8_x86`(9DKI)
- `ioWrite8_x86`(9DKI)
- `ioLoad16_x86`(9DKI)
- `ioStore16_x86`(9DKI)
- `ioRead16_x86`(9DKI)
- `ioWrite16_x86`(9DKI)
- `ioLoad32_x86`(9DKI)
- `ioStore32_x86`(9DKI)
- `ioRead32_x86`(9DKI)
- `ioWrite32_x86`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ioLoad8_x86, ioStore8_x86, ioRead8_x86, ioWrite8_x86, ioLoad16_x86, ioStore16_x86, ioRead16_x86, ioWrite16_x86, ioLoad32_x86, ioStore32_x86, ioRead32_x86, ioWrite32_x86 – Intel x86 specific I/O services

SYNOPSIS

```
#include <dki/dki.h>
uint8_f ioLoad8(uint32_f base, uint32_f offset);

void ioStore8(uint32_f base, uint32_f offset, uint8_f value);

void ioRead8(uint32_f base, uint32_f offset, uint8_f * addr, uint32_f count);

void ioWrite8(uint32_f base, uint32_f offset, uint8_f * addr, uint32_f count);

uint16_f ioLoad16(uint32_f base, uint32_f offset);

void ioStore16(uint32_f base, uint32_f offset, uint16_f value);

void ioRead16(uint32_f base, uint32_f offset, uint16_f * addr, uint32_f count);

void ioWrite16(uint32_f base, uint32_f offset, uint16_f * addr, uint32_f count);

uint32_f ioLoad32(uint32_f base, uint32_f offset);

void ioStore32(uint32_f base, uint32_f offset, uint32_f value);

void ioRead32(uint32_f base, uint32_f offset, uint32_f * addr, uint32_f count);

void ioWrite32(uint32_f base, uint32_f offset, uint32_f * addr, uint32_f count);
```

FEATURES

DKI

EXTENDED DESCRIPTION

The DKI provides specific I/O routines which can be used by a host bus driver to implement bus I/O operations.

Specific I/O services are defined below as sets of routines where the **xx** suffix indicates the bit length of the data to which the service applies. This suffix may take one of the following values:

- 8 for 8-bit data
- 16 for 16-bit data
- 32 for 32-bit data

`ioLoadxx`

The `ioLoadxx` loads data from a given I/O address and returns the read value. The read location is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address.

`ioStorexx`

The `ioStorexx` stores a specified value at a given I/O address. The I/O address written is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *value* argument specifies the value to store (write).

`ioReadxx`

The `ioReadxx` reads data from a given I/O address *count* times and stores values at the memory location pointed to by *addr* .

The read location is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *addr* argument specifies the location where read values will be stored. The *count* argument specifies the desired number of reads.

`ioWritexx`

The `ioWritexx` reads *count* data from the memory location at *addr* and writes them at the I/O address specified by *base* + *offset* .

The read location is specified by the parameter *addr* . The I/O location written is composed of a base address and an offset from this base. The *base* argument specifies the base address. The *offset* argument specifies the offset from the base address. The *count* argument specifies the desired number of writes to perform.

Specific I/O Allowed Calling Contexts

The table below specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
ioLoadxx	+	+	+	-
ioStorexx	+	+	+	-
ioReadxx	+	+	+	-
ioWritexx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

loadSwap_16, storeSwap_16, swap_16, loadSwap_32, storeSwap_32, swap_32, loadSwap_64, storeSwap_64, swap_64 – specific i/o services

SYNOPSIS

```
#include <dki/dki.h>
uint16_f loadSwap_16(uint16_f * addr);

void storeSwap_16(uint16_f * addr, uint16_f value);

void swap_16(uint16_f * addr);

uint32_f loadSwap_32(uint32_f * addr);

void storeSwap_32(uint32_f * addr, uint32_f value);

void swap_32(uint32_f * addr);

uint64_f loadSwap_64(uint64_f * addr);

void storeSwap_64(uint64_f * addr, uint64_f value);

void swap_64(uint64_f * addr);
```

FEATURES

DKI

DESCRIPTION

This function provides DKI routines to handle different byte ordering between the processor bus and the host bus.

EXTENDED DESCRIPTION

Typically, these services are intended to be used by a host bus driver to handle different byte ordering between the processor bus and the host bus. Specific I/O services are defined below as sets of routines where the _xx suffix indicates the bit length of the data on which the service applies. This suffix may take one of the following values:

_16 for 16-bit data

_32 for 32-bit data

_64 for 64-bit data

`loadSwap_xx` loads data from a given address and returns the corresponding byte swapped value. The *addr* argument specifies the address to read from.

`storeSwap_xx` stores into a given address the *value* byte swapped. The *addr* argument specifies the address to write to.

`swap_xx` swap in place the bytes of the data stored at a given address. The *addr* argument specifies the address of the data to byte-swap.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
loadSwap_xx	+	+	+	-

storeSwap_xx	+	+	+	-
swap_xx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

loadSwap_16, storeSwap_16, swap_16, loadSwap_32, storeSwap_32, swap_32, loadSwap_64, storeSwap_64, swap_64 – specific i/o services

SYNOPSIS

```
#include <dki/dki.h>
uint16_f loadSwap_16(uint16_f * addr);

void storeSwap_16(uint16_f * addr, uint16_f value);

void swap_16(uint16_f * addr);

uint32_f loadSwap_32(uint32_f * addr);

void storeSwap_32(uint32_f * addr, uint32_f value);

void swap_32(uint32_f * addr);

uint64_f loadSwap_64(uint64_f * addr);

void storeSwap_64(uint64_f * addr, uint64_f value);

void swap_64(uint64_f * addr);
```

FEATURES

DKI

DESCRIPTION

This function provides DKI routines to handle different byte ordering between the processor bus and the host bus.

EXTENDED DESCRIPTION

Typically, these services are intended to be used by a host bus driver to handle different byte ordering between the processor bus and the host bus. Specific I/O services are defined below as sets of routines where the _xx suffix indicates the bit length of the data on which the service applies. This suffix may take one of the following values:

_16 for 16-bit data

_32 for 32-bit data

_64 for 64-bit data

loadSwap_xx loads data from a given address and returns the corresponding byte swapped value. The *addr* argument specifies the address to read from.

storeSwap_xx stores into a given address the *value* byte swapped. The *addr* argument specifies the address to write to.

swap_xx swap in place the bytes of the data stored at a given address. The *addr* argument specifies the address of the data to byte-swap.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
loadSwap_xx	+	+	+	-

storeSwap_xx	+	+	+	-
swap_xx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

loadSwap_16, storeSwap_16, swap_16, loadSwap_32, storeSwap_32, swap_32, loadSwap_64, storeSwap_64, swap_64 – specific i/o services

SYNOPSIS

```
#include <dki/dki.h>
uint16_f loadSwap_16(uint16_f * addr);

void storeSwap_16(uint16_f * addr, uint16_f value);

void swap_16(uint16_f * addr);

uint32_f loadSwap_32(uint32_f * addr);

void storeSwap_32(uint32_f * addr, uint32_f value);

void swap_32(uint32_f * addr);

uint64_f loadSwap_64(uint64_f * addr);

void storeSwap_64(uint64_f * addr, uint64_f value);

void swap_64(uint64_f * addr);
```

FEATURES

DKI

DESCRIPTION

This function provides DKI routines to handle different byte ordering between the processor bus and the host bus.

EXTENDED DESCRIPTION

Typically, these services are intended to be used by a host bus driver to handle different byte ordering between the processor bus and the host bus. Specific I/O services are defined below as sets of routines where the _xx suffix indicates the bit length of the data on which the service applies. This suffix may take one of the following values:

_16 for 16-bit data

_32 for 32-bit data

_64 for 64-bit data

loadSwap_xx loads data from a given address and returns the corresponding byte swapped value. The *addr* argument specifies the address to read from.

storeSwap_xx stores into a given address the *value* byte swapped. The *addr* argument specifies the address to write to.

swap_xx swap in place the bytes of the data stored at a given address. The *addr* argument specifies the address of the data to byte-swap.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
loadSwap_xx	+	+	+	-

storeSwap_xx	+	+	+	-
swap_xx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

loadSwapEieio_16, storeSwapEieio_16, swapEieio_16, loadSwapEieio_32, storeSwapEieio_32, swapEieio_32, eieio – i/o services

FEATURES

DKI

DESCRIPTION

Provides access to DKI i/o routines, optimized to facilitate byte swapping and synchronization of input/output.

See the architecture specific man pages:

- `loadSwapEieio_16_powerpc`(9DKI)
- `storeSwapEieio_16_powerpc`(9DKI)
- `swapEieio_16_powerpc`(9DKI)
- `loadSwapEieio_32_powerpc`(9DKI)
- `storeSwapEieio_32_powerpc`(9DKI)
- `swapEieio_32_powerpc`(9DKI)
- `eieio`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

loadSwapEieio_16_powerpc, loadSwapSync_16_powerpc, storeSwapEieio_16_powerpc, storeSwapSync_16_powerpc, swapEieio_16_powerpc, loadSwapEieio_32_powerpc, loadSwapSync_32_powerpc, storeSwapEieio_32_powerpc, storeSwapSync_32_powerpc, swapEieio_32_powerpc, eieio_powerpc, ioSync_powerpc – PowerPC specific i/o services

SYNOPSIS

```
#include <dki/dki.h>
uint16_f loadSwapEieio_16(uint16_f * addr);

unit16_f loadSwapSync_16(uint16_f * addr);

void storeSwapEieio_16(uint16_f * addr, uint16_f value);

void storeSwapSync_16(uint16_f * addr, uint16_f value);

void swapEieio_16(uint16_f * addr);

uint32_f loadSwapEieio_32(uint32_f * addr);

unit32_f loadSwapSync_32(uint32_f * addr);

void storeSwapEieio_32(uint32_f * addr, uint32_f value);

void storeSwapSync_32(uint32_f * addr, uint32_f value);

void swapEieio_32 (uint32_f * addr);

void ioSync(void);

void eieio(void);
```

FEATURES

DKI

DESCRIPTION

Provides access to PowerPC DKI I/O routines, optimized to facilitate byte swapping and synchronization of input/output.

EXTENDED DESCRIPTION

The PowerPC DKI provides specific I/O routines to handle byte swapping and synchronization of I/O. These services are based on PowerPC specific instructions to store/load with byte swapping, and to enforce in-order execution of I/O.

Typically, these services are intended to be used by a host bus driver to handle different byte ordering between the PowerPC bus and the host bus, or to enforce sequential execution of I/O. PowerPC specific I/O services are defined below as sets of routines where the _xx suffix indicates the bit length of the data on which the service applies. This suffix may take one of the following values:

_16	for 16-bit data
_32	for 32-bit data

`loadSwapEieio_xx` loads data from a given address and returns the corresponding byte swapped value. A PowerPC `eieio` instruction is issued after the data is read, to enforce execution of subsequent I/O sequentially with respect to the current load. The *addr* argument specifies the address to read from.

`loadSwapSync_xx` behaves like `loadSwapEieio_xx` but issues a PowerPC `isync` instruction after that data is read. This ensures that all possible exceptions are taken into account before returning.

`storeSwapEieio_xx` stores, at a given address, the *value* byte swapped. A PowerPC `eieio` instruction is issued after the data is written, to enforce execution of subsequent I/O sequentially with respect to the current store. The *addr* argument specifies the address to write to.

`storeSwapSync_xx` behaves like `storeSwapEieio_xx` but issues a PowerPC `isync` instruction after that data is read. This ensures that all possible exceptions are taken into account before returning.

`swapEieio_xx` swaps in place the bytes of the data stored at a given address. A PowerPC `eieio` instruction is issued between reading the data and re-writing the byte-swapped data, to enforce execution in order. The *addr* argument specifies the address of the data to byte-swap.

`ioSync` enforces context synchronization, completion of all instructions, and completion of all exceptions before returning.

`eieio` enforces execution in order of I/O and is mapped to the corresponding PowerPC assembler instruction. This routine is intended for use in managing shared data structures, in doing memory-mapped I/O, and in preventing load/store combining operations in main memory. Refer to PowerPC programming manuals for more details.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
loadSwapEieio_xx	+	+	+	-
loadSwapSync_xx	+	+	+	-
storeSwapEieio_xx	+	+	+	-
storeSwapSync_xx	+	+	+	-
swapEieio_xx	+	+	+	-
ioSync	+	+	+	-
eieio	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`loadSwap_16`(9DKI) , `svDkiThreadCall`(9DKI)

NAME

loadSwapEieio_16, storeSwapEieio_16, swapEieio_16, loadSwapEieio_32, storeSwapEieio_32, swapEieio_32, eieio – i/o services

FEATURES

DKI

DESCRIPTION

Provides access to DKI i/o routines, optimized to facilitate byte swapping and synchronization of input/output.

See the architecture specific man pages:

- `loadSwapEieio_16_powerpc`(9DKI)
- `storeSwapEieio_16_powerpc`(9DKI)
- `swapEieio_16_powerpc`(9DKI)
- `loadSwapEieio_32_powerpc`(9DKI)
- `storeSwapEieio_32_powerpc`(9DKI)
- `swapEieio_32_powerpc`(9DKI)
- `eieio`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

loadSwapEieio_16_powerpc, loadSwapSync_16_powerpc, storeSwapEieio_16_powerpc, storeSwapSync_16_powerpc, swapEieio_16_powerpc, loadSwapEieio_32_powerpc, loadSwapSync_32_powerpc, storeSwapEieio_32_powerpc, storeSwapSync_32_powerpc, swapEieio_32_powerpc, eieio_powerpc, ioSync_powerpc – PowerPC specific i/o services

SYNOPSIS

```
#include <dki/dki.h>
uint16_f loadSwapEieio_16(uint16_f * addr);

unit16_f loadSwapSync_16(uint16_f * addr);

void storeSwapEieio_16(uint16_f * addr, uint16_f value);

void storeSwapSync_16(uint16_f * addr, uint16_f value);

void swapEieio_16(uint16_f * addr);

uint32_f loadSwapEieio_32(uint32_f * addr);

unit32_f loadSwapSync_32(uint32_f * addr);

void storeSwapEieio_32(uint32_f * addr, uint32_f value);

void storeSwapSync_32(uint32_f * addr, uint32_f value);

void swapEieio_32 (uint32_f * addr);

void ioSync(void);

void eieio(void);
```

FEATURES

DKI

DESCRIPTION

Provides access to PowerPC DKI I/O routines, optimized to facilitate byte swapping and synchronization of input/output.

EXTENDED DESCRIPTION

The PowerPC DKI provides specific I/O routines to handle byte swapping and synchronization of I/O. These services are based on PowerPC specific instructions to store/load with byte swapping, and to enforce in-order execution of I/O.

Typically, these services are intended to be used by a host bus driver to handle different byte ordering between the PowerPC bus and the host bus, or to enforce sequential execution of I/O. PowerPC specific I/O services are defined below as sets of routines where the _xx suffix indicates the bit length of the data on which the service applies. This suffix may take one of the following values:

_16	for 16-bit data
_32	for 32-bit data

`loadSwapEieio_xx` loads data from a given address and returns the corresponding byte swapped value. A PowerPC `eieio` instruction is issued after the data is read, to enforce execution of subsequent I/O sequentially with respect to the current load. The *addr* argument specifies the address to read from.

`loadSwapSync_xx` behaves like `loadSwapEieio_xx` but issues a PowerPC `isync` instruction after that data is read. This ensures that all possible exceptions are taken into account before returning.

`storeSwapEieio_xx` stores, at a given address, the *value* byte swapped. A PowerPC `eieio` instruction is issued after the data is written, to enforce execution of subsequent I/O sequentially with respect to the current store. The *addr* argument specifies the address to write to.

`storeSwapSync_xx` behaves like `storeSwapEieio_xx` but issues a PowerPC `isync` instruction after that data is read. This ensures that all possible exceptions are taken into account before returning.

`swapEieio_xx` swaps in place the bytes of the data stored at a given address. A PowerPC `eieio` instruction is issued between reading the data and re-writing the byte-swapped data, to enforce execution in order. The *addr* argument specifies the address of the data to byte-swap.

`ioSync` enforces context synchronization, completion of all instructions, and completion of all exceptions before returning.

`eieio` enforces execution in order of I/O and is mapped to the corresponding PowerPC assembler instruction. This routine is intended for use in managing shared data structures, in doing memory-mapped I/O, and in preventing load/store combining operations in main memory. Refer to PowerPC programming manuals for more details.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
loadSwapEieio_xx	+	+	+	-
loadSwapSync_xx	+	+	+	-
storeSwapEieio_xx	+	+	+	-
storeSwapSync_xx	+	+	+	-
swapEieio_xx	+	+	+	-
ioSync	+	+	+	-
eieio	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`loadSwap_16`(9DKI) , `svDkiThreadCall`(9DKI)

NAME

load_sync_8_usparc, store_sync_8_usparc, loadSwap_sync_16_usparc, storeSwap_sync_16_usparc, load_sync_16_usparc, store_sync_16_usparc, loadSwap_sync_32_usparc, storeSwap_sync_32_usparc, load_sync_32_usparc, store_sync_32_usparc, loadSwap_sync_64_usparc, storeSwap_sync_64_usparc, load_sync_64_usparc, store_sync_64_usparc – UltraSparc specific i/o services

SYNOPSIS

```
#include <dki/f_dki.h>
uint8_f load_sync_8(uint8_f * addr);

void store_sync_8(uint8_f * addr, uint8_f value);

uint16_f loadSwap_sync_16(uint16_f * addr);

void storeSwap_sync_16(uint16_f * addr, uint16_f value);

uint16_f load_sync_16(uint16_f * addr);

void store_sync_16(uint16_f * addr, uint16_f value);

uint32_f loadSwap_sync_32(uint32_f * addr);

void storeSwap_sync_32(uint32_f * addr, uint32_f value);

uint32_f load_sync_32(uint32_f * addr);

void store_sync_32(uint32_f * addr, uint32_f value);

uint64_f loadSwap_sync_64(uint64_f * addr);

void storeSwap_sync_64(uint64_f * addr, uint64_f value);

uint64_f load_sync_64(uint64_f * addr);

void store_sync_64(uint64_f * addr, uint64_f value);
```

FEATURES

DKI

DESCRIPTION

The UltraSPARC DKI provides specific I/O routines optimized to handle byte swapping. These services are analogous to the generic ones except the `membar #Sync` instruction is issued each time the data is read/written from/to the memory. Note that this allows recovery from an asynchronous exception caused by these types of service routines.

Typically, these services are intended to be used by a host bus driver to handle different byte ordering between the processor bus and the host bus.

In addtion, the UltraSPARC DKI provides specific synchronized I/O routines. These routines do not perform the byte swapping but they are safe and allow recovery from an asynchronous exception caused by this type of service routine.

Specific I/O services are defined below as sets of routines where the `_xx` suffix indicates the bit length of the data on which the service applies. This suffix may take one of the following values:

- _8 for 8-bit data
- _16 for 16-bit data
- _32 for 32-bit data
- _64 for 64-bit data

Note that the _8 suffix is only applied to the load_sync_xx and store_sync_xx routines.

loadSwap_sync_xx loadSwap_sync_xx loads data from a given address and returns the corresponding byte swapped value. The membar #Sync instruction is issued once the data has been loaded.

The *addr* argument specifies the address from which to read.

storeSwap_sync_xx storeSwap_sync_xx stores into a given address the *value* byte swapped. The membar #Sync instruction is issued once the data is stored.

The *addr* argument specifies the address to which to write.

load_sync_xx load_sync_xx loads data from a given address. The membar #Sync instruction is issued once the data is loaded.

The *addr* argument specifies the address from which to read.

storeSwap_sync_xx storeSwap_sync_xx stores into a given address the *value* . The membar #Sync instruction is issued once the data is stored.

The *addr* argument specifies the address to which to write.

ALLOWED CALLING CONTEXTS

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
loadSwap_sync_xx	+	+	+	-
storeSwap_sync_xx	+	+	+	-
load_sync_xx	+	+	+	-
store_sync_xx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `svAsyncExcepAttach_usparc>`(9DKI) ,
`svAsyncExcepDetach_usparc>`(9DKI)

NAME

load_sync_8_usparc, store_sync_8_usparc, loadSwap_sync_16_usparc, storeSwap_sync_16_usparc, load_sync_16_usparc, store_sync_16_usparc, loadSwap_sync_32_usparc, storeSwap_sync_32_usparc, load_sync_32_usparc, store_sync_32_usparc, loadSwap_sync_64_usparc, storeSwap_sync_64_usparc, load_sync_64_usparc, store_sync_64_usparc – UltraSparc specific i/o services

SYNOPSIS

```
#include <dki/f_dki.h>
uint8_f load_sync_8(uint8_f * addr);

void store_sync_8(uint8_f * addr, uint8_f value);

uint16_f loadSwap_sync_16(uint16_f * addr);

void storeSwap_sync_16(uint16_f * addr, uint16_f value);

uint16_f load_sync_16(uint16_f * addr);

void store_sync_16(uint16_f * addr, uint16_f value);

uint32_f loadSwap_sync_32(uint32_f * addr);

void storeSwap_sync_32(uint32_f * addr, uint32_f value);

uint32_f load_sync_32(uint32_f * addr);

void store_sync_32(uint32_f * addr, uint32_f value);

uint64_f loadSwap_sync_64(uint64_f * addr);

void storeSwap_sync_64(uint64_f * addr, uint64_f value);

uint64_f load_sync_64(uint64_f * addr);

void store_sync_64(uint64_f * addr, uint64_f value);
```

FEATURES

DKI

DESCRIPTION

The UltraSPARC DKI provides specific I/O routines optimized to handle byte swapping. These services are analogous to the generic ones except the `membar #Sync` instruction is issued each time the data is read/written from/to the memory. Note that this allows recovery from an asynchronous exception caused by these types of service routines.

Typically, these services are intended to be used by a host bus driver to handle different byte ordering between the processor bus and the host bus.

In addtion, the UltraSPARC DKI provides specific synchronized I/O routines. These routines do not perform the byte swapping but they are safe and allow recovery from an asynchronous exception caused by this type of service routine.

Specific I/O services are defined below as sets of routines where the `_xx` suffix indicates the bit length of the data on which the service applies. This suffix may take one of the following values:

- _8 for 8-bit data
- _16 for 16-bit data
- _32 for 32-bit data
- _64 for 64-bit data

Note that the _8 suffix is only applied to the load_sync_xx and store_sync_xx routines.

loadSwap_sync_xx loadSwap_sync_xx loads data from a given address and returns the corresponding byte swapped value. The membar #Sync instruction is issued once the data has been loaded.

The *addr* argument specifies the address from which to read.

storeSwap_sync_xx storeSwap_sync_xx stores into a given address the *value* byte swapped. The membar #Sync instruction is issued once the data is stored.

The *addr* argument specifies the address to which to write.

load_sync_xx load_sync_xx loads data from a given address. The membar #Sync instruction is issued once the data is loaded.

The *addr* argument specifies the address from which to read.

storeSwap_sync_xx storeSwap_sync_xx stores into a given address the *value* . The membar #Sync instruction is issued once the data is stored.

The *addr* argument specifies the address to which to write.

ALLOWED CALLING CONTEXTS

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
loadSwap_sync_xx	+	+	+	-
storeSwap_sync_xx	+	+	+	-
load_sync_xx	+	+	+	-
store_sync_xx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `svAsyncExcepAttach_usparc>`(9DKI) , `svAsyncExcepDetach_usparc>`(9DKI)

NAME

load_sync_8_usparc, store_sync_8_usparc, loadSwap_sync_16_usparc, storeSwap_sync_16_usparc, load_sync_16_usparc, store_sync_16_usparc, loadSwap_sync_32_usparc, storeSwap_sync_32_usparc, load_sync_32_usparc, store_sync_32_usparc, loadSwap_sync_64_usparc, storeSwap_sync_64_usparc, load_sync_64_usparc, store_sync_64_usparc – UltraSparc specific i/o services

SYNOPSIS

```
#include <dki/f_dki.h>
uint8_f load_sync_8(uint8_f * addr);

void store_sync_8(uint8_f * addr, uint8_f value);

uint16_f loadSwap_sync_16(uint16_f * addr);

void storeSwap_sync_16(uint16_f * addr, uint16_f value);

uint16_f load_sync_16(uint16_f * addr);

void store_sync_16(uint16_f * addr, uint16_f value);

uint32_f loadSwap_sync_32(uint32_f * addr);

void storeSwap_sync_32(uint32_f * addr, uint32_f value);

uint32_f load_sync_32(uint32_f * addr);

void store_sync_32(uint32_f * addr, uint32_f value);

uint64_f loadSwap_sync_64(uint64_f * addr);

void storeSwap_sync_64(uint64_f * addr, uint64_f value);

uint64_f load_sync_64(uint64_f * addr);

void store_sync_64(uint64_f * addr, uint64_f value);
```

FEATURES

DKI

DESCRIPTION

The UltraSPARC DKI provides specific I/O routines optimized to handle byte swapping. These services are analogous to the generic ones except the `membar #Sync` instruction is issued each time the data is read/written from/to the memory. Note that this allows recovery from an asynchronous exception caused by these types of service routines.

Typically, these services are intended to be used by a host bus driver to handle different byte ordering between the processor bus and the host bus.

In addtion, the UltraSPARC DKI provides specific synchronized I/O routines. These routines do not perform the byte swapping but they are safe and allow recovery from an asynchronous exception caused by this type of service routine.

Specific I/O services are defined below as sets of routines where the `_xx` suffix indicates the bit length of the data on which the service applies. This suffix may take one of the following values:

- _8 for 8-bit data
- _16 for 16-bit data
- _32 for 32-bit data
- _64 for 64-bit data

Note that the _8 suffix is only applied to the load_sync_xx and store_sync_xx routines.

loadSwap_sync_xx loadSwap_sync_xx loads data from a given address and returns the corresponding byte swapped value. The membar #Sync instruction is issued once the data has been loaded.

The *addr* argument specifies the address from which to read.

storeSwap_sync_xx storeSwap_sync_xx stores into a given address the *value* byte swapped. The membar #Sync instruction is issued once the data is stored.

The *addr* argument specifies the address to which to write.

load_sync_xx load_sync_xx loads data from a given address. The membar #Sync instruction is issued once the data is loaded.

The *addr* argument specifies the address from which to read.

storeSwap_sync_xx storeSwap_sync_xx stores into a given address the *value* . The membar #Sync instruction is issued once the data is stored.

The *addr* argument specifies the address to which to write.

ALLOWED CALLING CONTEXTS

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
loadSwap_sync_xx	+	+	+	-
storeSwap_sync_xx	+	+	+	-
load_sync_xx	+	+	+	-
store_sync_xx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `svAsyncExcepAttach_usparc>`(9DKI) ,
`svAsyncExcepDetach_usparc>`(9DKI)

NAME

load_sync_8_usparc, store_sync_8_usparc, loadSwap_sync_16_usparc, storeSwap_sync_16_usparc, load_sync_16_usparc, store_sync_16_usparc, loadSwap_sync_32_usparc, storeSwap_sync_32_usparc, load_sync_32_usparc, store_sync_32_usparc, loadSwap_sync_64_usparc, storeSwap_sync_64_usparc, load_sync_64_usparc, store_sync_64_usparc – UltraSparc specific i/o services

SYNOPSIS

```
#include <dki/f_dki.h>
uint8_f load_sync_8(uint8_f * addr);

void store_sync_8(uint8_f * addr, uint8_f value);

uint16_f loadSwap_sync_16(uint16_f * addr);

void storeSwap_sync_16(uint16_f * addr, uint16_f value);

uint16_f load_sync_16(uint16_f * addr);

void store_sync_16(uint16_f * addr, uint16_f value);

uint32_f loadSwap_sync_32(uint32_f * addr);

void storeSwap_sync_32(uint32_f * addr, uint32_f value);

uint32_f load_sync_32(uint32_f * addr);

void store_sync_32(uint32_f * addr, uint32_f value);

uint64_f loadSwap_sync_64(uint64_f * addr);

void storeSwap_sync_64(uint64_f * addr, uint64_f value);

uint64_f load_sync_64(uint64_f * addr);

void store_sync_64(uint64_f * addr, uint64_f value);
```

FEATURES

DKI

DESCRIPTION

The UltraSPARC DKI provides specific I/O routines optimized to handle byte swapping. These services are analogous to the generic ones except the `membar #Sync` instruction is issued each time the data is read/written from/to the memory. Note that this allows recovery from an asynchronous exception caused by these types of service routines.

Typically, these services are intended to be used by a host bus driver to handle different byte ordering between the processor bus and the host bus.

In addtion, the UltraSPARC DKI provides specific synchronized I/O routines. These routines do not perform the byte swapping but they are safe and allow recovery from an asynchronous exception caused by this type of service routine.

Specific I/O services are defined below as sets of routines where the `_xx` suffix indicates the bit length of the data on which the service applies. This suffix may take one of the following values:

- _8 for 8-bit data
- _16 for 16-bit data
- _32 for 32-bit data
- _64 for 64-bit data

Note that the _8 suffix is only applied to the load_sync_xx and store_sync_xx routines.

loadSwap_sync_xx loadSwap_sync_xx loads data from a given address and returns the corresponding byte swapped value. The membar #Sync instruction is issued once the data has been loaded.

The *addr* argument specifies the address from which to read.

storeSwap_sync_xx storeSwap_sync_xx stores into a given address the *value* byte swapped. The membar #Sync instruction is issued once the data is stored.

The *addr* argument specifies the address to which to write.

load_sync_xx load_sync_xx loads data from a given address. The membar #Sync instruction is issued once the data is loaded.

The *addr* argument specifies the address from which to read.

storeSwap_sync_xx storeSwap_sync_xx stores into a given address the *value* . The membar #Sync instruction is issued once the data is stored.

The *addr* argument specifies the address to which to write.

ALLOWED CALLING CONTEXTS

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
loadSwap_sync_xx	+	+	+	-
storeSwap_sync_xx	+	+	+	-
load_sync_xx	+	+	+	-
store_sync_xx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `svAsyncExcepAttach_usparc>`(9DKI) , `svAsyncExcepDetach_usparc>`(9DKI)

NAME

load_sync_8_usparc, store_sync_8_usparc, loadSwap_sync_16_usparc, storeSwap_sync_16_usparc, load_sync_16_usparc, store_sync_16_usparc, loadSwap_sync_32_usparc, storeSwap_sync_32_usparc, load_sync_32_usparc, store_sync_32_usparc, loadSwap_sync_64_usparc, storeSwap_sync_64_usparc, load_sync_64_usparc, store_sync_64_usparc – UltraSparc specific i/o services

SYNOPSIS

```
#include <dki/f_dki.h>
uint8_f load_sync_8(uint8_f * addr);

void store_sync_8(uint8_f * addr, uint8_f value);

uint16_f loadSwap_sync_16(uint16_f * addr);

void storeSwap_sync_16(uint16_f * addr, uint16_f value);

uint16_f load_sync_16(uint16_f * addr);

void store_sync_16(uint16_f * addr, uint16_f value);

uint32_f loadSwap_sync_32(uint32_f * addr);

void storeSwap_sync_32(uint32_f * addr, uint32_f value);

uint32_f load_sync_32(uint32_f * addr);

void store_sync_32(uint32_f * addr, uint32_f value);

uint64_f loadSwap_sync_64(uint64_f * addr);

void storeSwap_sync_64(uint64_f * addr, uint64_f value);

uint64_f load_sync_64(uint64_f * addr);

void store_sync_64(uint64_f * addr, uint64_f value);
```

FEATURES

DKI

DESCRIPTION

The UltraSPARC DKI provides specific I/O routines optimized to handle byte swapping. These services are analogous to the generic ones except the `membar #Sync` instruction is issued each time the data is read/written from/to the memory. Note that this allows recovery from an asynchronous exception caused by these types of service routines.

Typically, these services are intended to be used by a host bus driver to handle different byte ordering between the processor bus and the host bus.

In addtion, the UltraSPARC DKI provides specific synchronized I/O routines. These routines do not perform the byte swapping but they are safe and allow recovery from an asynchronous exception caused by this type of service routine.

Specific I/O services are defined below as sets of routines where the `_xx` suffix indicates the bit length of the data on which the service applies. This suffix may take one of the following values:

- _8 for 8-bit data
- _16 for 16-bit data
- _32 for 32-bit data
- _64 for 64-bit data

Note that the _8 suffix is only applied to the load_sync_xx and store_sync_xx routines.

loadSwap_sync_xx loadSwap_sync_xx loads data from a given address and returns the corresponding byte swapped value. The membar #Sync instruction is issued once the data has been loaded.

The *addr* argument specifies the address from which to read.

storeSwap_sync_xx storeSwap_sync_xx stores into a given address the *value* byte swapped. The membar #Sync instruction is issued once the data is stored.

The *addr* argument specifies the address to which to write.

load_sync_xx load_sync_xx loads data from a given address. The membar #Sync instruction is issued once the data is loaded.

The *addr* argument specifies the address from which to read.

storeSwap_sync_xx storeSwap_sync_xx stores into a given address the *value* . The membar #Sync instruction is issued once the data is stored.

The *addr* argument specifies the address to which to write.

ALLOWED CALLING CONTEXTS

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
loadSwap_sync_xx	+	+	+	-
storeSwap_sync_xx	+	+	+	-
load_sync_xx	+	+	+	-
store_sync_xx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `svAsyncExcepAttach_usparc>`(9DKI) ,
`svAsyncExcepDetach_usparc>`(9DKI)

NAME

load_sync_8_usparc, store_sync_8_usparc, loadSwap_sync_16_usparc, storeSwap_sync_16_usparc, load_sync_16_usparc, store_sync_16_usparc, loadSwap_sync_32_usparc, storeSwap_sync_32_usparc, load_sync_32_usparc, store_sync_32_usparc, loadSwap_sync_64_usparc, storeSwap_sync_64_usparc, load_sync_64_usparc, store_sync_64_usparc – UltraSparc specific i/o services

SYNOPSIS

```
#include <dki/f_dki.h>
uint8_f load_sync_8(uint8_f * addr);

void store_sync_8(uint8_f * addr, uint8_f value);

uint16_f loadSwap_sync_16(uint16_f * addr);

void storeSwap_sync_16(uint16_f * addr, uint16_f value);

uint16_f load_sync_16(uint16_f * addr);

void store_sync_16(uint16_f * addr, uint16_f value);

uint32_f loadSwap_sync_32(uint32_f * addr);

void storeSwap_sync_32(uint32_f * addr, uint32_f value);

uint32_f load_sync_32(uint32_f * addr);

void store_sync_32(uint32_f * addr, uint32_f value);

uint64_f loadSwap_sync_64(uint64_f * addr);

void storeSwap_sync_64(uint64_f * addr, uint64_f value);

uint64_f load_sync_64(uint64_f * addr);

void store_sync_64(uint64_f * addr, uint64_f value);
```

FEATURES

DKI

DESCRIPTION

The UltraSPARC DKI provides specific I/O routines optimized to handle byte swapping. These services are analogous to the generic ones except the `membar #Sync` instruction is issued each time the data is read/written from/to the memory. Note that this allows recovery from an asynchronous exception caused by these types of service routines.

Typically, these services are intended to be used by a host bus driver to handle different byte ordering between the processor bus and the host bus.

In addtion, the UltraSPARC DKI provides specific synchronized I/O routines. These routines do not perform the byte swapping but they are safe and allow recovery from an asynchronous exception caused by this type of service routine.

Specific I/O services are defined below as sets of routines where the `_xx` suffix indicates the bit length of the data on which the service applies. This suffix may take one of the following values:

- _8 for 8-bit data
- _16 for 16-bit data
- _32 for 32-bit data
- _64 for 64-bit data

Note that the _8 suffix is only applied to the load_sync_xx and store_sync_xx routines.

loadSwap_sync_xx loadSwap_sync_xx loads data from a given address and returns the corresponding byte swapped value. The membar #Sync instruction is issued once the data has been loaded.

The *addr* argument specifies the address from which to read.

storeSwap_sync_xx storeSwap_sync_xx stores into a given address the *value* byte swapped. The membar #Sync instruction is issued once the data is stored.

The *addr* argument specifies the address to which to write.

load_sync_xx load_sync_xx loads data from a given address. The membar #Sync instruction is issued once the data is loaded.

The *addr* argument specifies the address from which to read.

storeSwap_sync_xx storeSwap_sync_xx stores into a given address the *value* . The membar #Sync instruction is issued once the data is stored.

The *addr* argument specifies the address to which to write.

ALLOWED CALLING CONTEXTS

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
loadSwap_sync_xx	+	+	+	-
storeSwap_sync_xx	+	+	+	-
load_sync_xx	+	+	+	-
store_sync_xx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `svAsyncExcepAttach_usparc>`(9DKI) , `svAsyncExcepDetach_usparc>`(9DKI)

NAME

load_sync_8_usparc, store_sync_8_usparc, loadSwap_sync_16_usparc, storeSwap_sync_16_usparc, load_sync_16_usparc, store_sync_16_usparc, loadSwap_sync_32_usparc, storeSwap_sync_32_usparc, load_sync_32_usparc, store_sync_32_usparc, loadSwap_sync_64_usparc, storeSwap_sync_64_usparc, load_sync_64_usparc, store_sync_64_usparc – UltraSparc specific i/o services

SYNOPSIS

```
#include <dki/f_dki.h>
uint8_f load_sync_8(uint8_f * addr);

void store_sync_8(uint8_f * addr, uint8_f value);

uint16_f loadSwap_sync_16(uint16_f * addr);

void storeSwap_sync_16(uint16_f * addr, uint16_f value);

uint16_f load_sync_16(uint16_f * addr);

void store_sync_16(uint16_f * addr, uint16_f value);

uint32_f loadSwap_sync_32(uint32_f * addr);

void storeSwap_sync_32(uint32_f * addr, uint32_f value);

uint32_f load_sync_32(uint32_f * addr);

void store_sync_32(uint32_f * addr, uint32_f value);

uint64_f loadSwap_sync_64(uint64_f * addr);

void storeSwap_sync_64(uint64_f * addr, uint64_f value);

uint64_f load_sync_64(uint64_f * addr);

void store_sync_64(uint64_f * addr, uint64_f value);
```

FEATURES

DKI

DESCRIPTION

The UltraSPARC DKI provides specific I/O routines optimized to handle byte swapping. These services are analogous to the generic ones except the `membar #Sync` instruction is issued each time the data is read/written from/to the memory. Note that this allows recovery from an asynchronous exception caused by these types of service routines.

Typically, these services are intended to be used by a host bus driver to handle different byte ordering between the processor bus and the host bus.

In addtion, the UltraSPARC DKI provides specific synchronized I/O routines. These routines do not perform the byte swapping but they are safe and allow recovery from an asynchronous exception caused by this type of service routine.

Specific I/O services are defined below as sets of routines where the `_xx` suffix indicates the bit length of the data on which the service applies. This suffix may take one of the following values:

- _8 for 8-bit data
- _16 for 16-bit data
- _32 for 32-bit data
- _64 for 64-bit data

Note that the _8 suffix is only applied to the load_sync_xx and store_sync_xx routines.

loadSwap_sync_xx loadSwap_sync_xx loads data from a given address and returns the corresponding byte swapped value. The membar #Sync instruction is issued once the data has been loaded.

The *addr* argument specifies the address from which to read.

storeSwap_sync_xx storeSwap_sync_xx stores into a given address the *value* byte swapped. The membar #Sync instruction is issued once the data is stored.

The *addr* argument specifies the address to which to write.

load_sync_xx load_sync_xx loads data from a given address. The membar #Sync instruction is issued once the data is loaded.

The *addr* argument specifies the address from which to read.

storeSwap_sync_xx storeSwap_sync_xx stores into a given address the *value* . The membar #Sync instruction is issued once the data is stored.

The *addr* argument specifies the address to which to write.

ALLOWED CALLING CONTEXTS

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
loadSwap_sync_xx	+	+	+	-
storeSwap_sync_xx	+	+	+	-
load_sync_xx	+	+	+	-
store_sync_xx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `svAsyncExcepAttach_usparc>`(9DKI) ,
`svAsyncExcepDetach_usparc>`(9DKI)

NAME

loadSwap_16, storeSwap_16, swap_16, loadSwap_32, storeSwap_32, swap_32, loadSwap_64, storeSwap_64, swap_64 – specific i/o services

SYNOPSIS

```
#include <dki/dki.h>
uint16_f loadSwap_16(uint16_f * addr);

void storeSwap_16(uint16_f * addr, uint16_f value);

void swap_16(uint16_f * addr);

uint32_f loadSwap_32(uint32_f * addr);

void storeSwap_32(uint32_f * addr, uint32_f value);

void swap_32(uint32_f * addr);

uint64_f loadSwap_64(uint64_f * addr);

void storeSwap_64(uint64_f * addr, uint64_f value);

void swap_64(uint64_f * addr);
```

FEATURES

DKI

DESCRIPTION

This function provides DKI routines to handle different byte ordering between the processor bus and the host bus.

EXTENDED DESCRIPTION

Typically, these services are intended to be used by a host bus driver to handle different byte ordering between the processor bus and the host bus. Specific I/O services are defined below as sets of routines where the _xx suffix indicates the bit length of the data on which the service applies. This suffix may take one of the following values:

_16 for 16-bit data

_32 for 32-bit data

_64 for 64-bit data

`loadSwap_xx` loads data from a given address and returns the corresponding byte swapped value. The *addr* argument specifies the address to read from.

`storeSwap_xx` stores into a given address the *value* byte swapped. The *addr* argument specifies the address to write to.

`swap_xx` swap in place the bytes of the data stored at a given address. The *addr* argument specifies the address of the data to byte-swap.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
loadSwap_xx	+	+	+	-

storeSwap_xx	+	+	+	-
swap_xx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

loadSwap_16, storeSwap_16, swap_16, loadSwap_32, storeSwap_32, swap_32, loadSwap_64, storeSwap_64, swap_64 – specific i/o services

SYNOPSIS

```
#include <dki/dki.h>
uint16_f loadSwap_16(uint16_f * addr);

void storeSwap_16(uint16_f * addr, uint16_f value);

void swap_16(uint16_f * addr);

uint32_f loadSwap_32(uint32_f * addr);

void storeSwap_32(uint32_f * addr, uint32_f value);

void swap_32(uint32_f * addr);

uint64_f loadSwap_64(uint64_f * addr);

void storeSwap_64(uint64_f * addr, uint64_f value);

void swap_64(uint64_f * addr);
```

FEATURES

DKI

DESCRIPTION

This function provides DKI routines to handle different byte ordering between the processor bus and the host bus.

EXTENDED DESCRIPTION

Typically, these services are intended to be used by a host bus driver to handle different byte ordering between the processor bus and the host bus. Specific I/O services are defined below as sets of routines where the _xx suffix indicates the bit length of the data on which the service applies. This suffix may take one of the following values:

_16 for 16-bit data

_32 for 32-bit data

_64 for 64-bit data

`loadSwap_xx` loads data from a given address and returns the corresponding byte swapped value. The *addr* argument specifies the address to read from.

`storeSwap_xx` stores into a given address the *value* byte swapped. The *addr* argument specifies the address to write to.

`swap_xx` swap in place the bytes of the data stored at a given address. The *addr* argument specifies the address of the data to byte-swap.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
loadSwap_xx	+	+	+	-

storeSwap_xx	+	+	+	-
swap_xx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

loadSwap_16, storeSwap_16, swap_16, loadSwap_32, storeSwap_32, swap_32, loadSwap_64, storeSwap_64, swap_64 – specific i/o services

SYNOPSIS

```
#include <dki/dki.h>
uint16_f loadSwap_16(uint16_f * addr);

void storeSwap_16(uint16_f * addr, uint16_f value);

void swap_16(uint16_f * addr);

uint32_f loadSwap_32(uint32_f * addr);

void storeSwap_32(uint32_f * addr, uint32_f value);

void swap_32(uint32_f * addr);

uint64_f loadSwap_64(uint64_f * addr);

void storeSwap_64(uint64_f * addr, uint64_f value);

void swap_64(uint64_f * addr);
```

FEATURES

DKI

DESCRIPTION

This function provides DKI routines to handle different byte ordering between the processor bus and the host bus.

EXTENDED DESCRIPTION

Typically, these services are intended to be used by a host bus driver to handle different byte ordering between the processor bus and the host bus. Specific I/O services are defined below as sets of routines where the _xx suffix indicates the bit length of the data on which the service applies. This suffix may take one of the following values:

_16 for 16-bit data

_32 for 32-bit data

_64 for 64-bit data

loadSwap_xx loads data from a given address and returns the corresponding byte swapped value. The *addr* argument specifies the address to read from.

storeSwap_xx stores into a given address the *value* byte swapped. The *addr* argument specifies the address to write to.

swap_xx swap in place the bytes of the data stored at a given address. The *addr* argument specifies the address of the data to byte-swap.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
loadSwap_xx	+	+	+	-

storeSwap_xx	+	+	+	-
swap_xx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

loadSwapEieio_16, storeSwapEieio_16, swapEieio_16, loadSwapEieio_32, storeSwapEieio_32, swapEieio_32, eieio – i/o services

FEATURES

DKI

DESCRIPTION

Provides access to DKI i/o routines, optimized to facilitate byte swapping and synchronization of input/output.

See the architecture specific man pages:

- `loadSwapEieio_16_powerpc`(9DKI)
- `storeSwapEieio_16_powerpc`(9DKI)
- `swapEieio_16_powerpc`(9DKI)
- `loadSwapEieio_32_powerpc`(9DKI)
- `storeSwapEieio_32_powerpc`(9DKI)
- `swapEieio_32_powerpc`(9DKI)
- `eieio`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

loadSwapEieio_16_powerpc, loadSwapSync_16_powerpc, storeSwapEieio_16_powerpc, storeSwapSync_16_powerpc, swapEieio_16_powerpc, loadSwapEieio_32_powerpc, loadSwapSync_32_powerpc, storeSwapEieio_32_powerpc, storeSwapSync_32_powerpc, swapEieio_32_powerpc, eieio_powerpc, ioSync_powerpc – PowerPC specific i/o services

SYNOPSIS

```
#include <dki/dki.h>
uint16_f loadSwapEieio_16(uint16_f * addr);

unit16_f loadSwapSync_16(uint16_f * addr);

void storeSwapEieio_16(uint16_f * addr, uint16_f value);

void storeSwapSync_16(uint16_f * addr, uint16_f value);

void swapEieio_16(uint16_f * addr);

uint32_f loadSwapEieio_32(uint32_f * addr);

unit32_f loadSwapSync_32(uint32_f * addr);

void storeSwapEieio_32(uint32_f * addr, uint32_f value);

void storeSwapSync_32(uint32_f * addr, uint32_f value);

void swapEieio_32 (uint32_f * addr);

void ioSync(void);

void eieio(void);
```

FEATURES

DKI

DESCRIPTION

Provides access to PowerPC DKI I/O routines, optimized to facilitate byte swapping and synchronization of input/output.

EXTENDED DESCRIPTION

The PowerPC DKI provides specific I/O routines to handle byte swapping and synchronization of I/O. These services are based on PowerPC specific instructions to store/load with byte swapping, and to enforce in-order execution of I/O.

Typically, these services are intended to be used by a host bus driver to handle different byte ordering between the PowerPC bus and the host bus, or to enforce sequential execution of I/O. PowerPC specific I/O services are defined below as sets of routines where the _xx suffix indicates the bit length of the data on which the service applies. This suffix may take one of the following values:

_16	for 16-bit data
_32	for 32-bit data

`loadSwapEieio_xx` loads data from a given address and returns the corresponding byte swapped value. A PowerPC `eieio` instruction is issued after the data is read, to enforce execution of subsequent I/O sequentially with respect to the current load. The *addr* argument specifies the address to read from.

`loadSwapSync_xx` behaves like `loadSwapEieio_xx` but issues a PowerPC `isync` instruction after that data is read. This ensures that all possible exceptions are taken into account before returning.

`storeSwapEieio_xx` stores, at a given address, the *value* byte swapped. A PowerPC `eieio` instruction is issued after the data is written, to enforce execution of subsequent I/O sequentially with respect to the current store. The *addr* argument specifies the address to write to.

`storeSwapSync_xx` behaves like `storeSwapEieio_xx` but issues a PowerPC `isync` instruction after that data is read. This ensures that all possible exceptions are taken into account before returning.

`swapEieio_xx` swaps in place the bytes of the data stored at a given address. A PowerPC `eieio` instruction is issued between reading the data and re-writing the byte-swapped data, to enforce execution in order. The *addr* argument specifies the address of the data to byte-swap.

`ioSync` enforces context synchronization, completion of all instructions, and completion of all exceptions before returning.

`eieio` enforces execution in order of I/O and is mapped to the corresponding PowerPC assembler instruction. This routine is intended for use in managing shared data structures, in doing memory-mapped I/O, and in preventing load/store combining operations in main memory. Refer to PowerPC programming manuals for more details.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
loadSwapEieio_xx	+	+	+	-
loadSwapSync_xx	+	+	+	-
storeSwapEieio_xx	+	+	+	-
storeSwapSync_xx	+	+	+	-
swapEieio_xx	+	+	+	-
ioSync	+	+	+	-
eieio	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

loadSwap_16(9DKI) , svDkiThreadCall(9DKI)

NAME

loadSwapEieio_16, storeSwapEieio_16, swapEieio_16, loadSwapEieio_32, storeSwapEieio_32, swapEieio_32, eieio – i/o services

FEATURES

DKI

DESCRIPTION

Provides access to DKI i/o routines, optimized to facilitate byte swapping and synchronization of input/output.

See the architecture specific man pages:

- `loadSwapEieio_16_powerpc`(9DKI)
- `storeSwapEieio_16_powerpc`(9DKI)
- `swapEieio_16_powerpc`(9DKI)
- `loadSwapEieio_32_powerpc`(9DKI)
- `storeSwapEieio_32_powerpc`(9DKI)
- `swapEieio_32_powerpc`(9DKI)
- `eieio`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

loadSwapEieio_16_powerpc, loadSwapSync_16_powerpc, storeSwapEieio_16_powerpc, storeSwapSync_16_powerpc, swapEieio_16_powerpc, loadSwapEieio_32_powerpc, loadSwapSync_32_powerpc, storeSwapEieio_32_powerpc, storeSwapSync_32_powerpc, swapEieio_32_powerpc, eieio_powerpc, ioSync_powerpc – PowerPC specific i/o services

SYNOPSIS

```
#include <dki/dki.h>
uint16_f loadSwapEieio_16(uint16_f * addr);

unit16_f loadSwapSync_16(uint16_f * addr);

void storeSwapEieio_16(uint16_f * addr, uint16_f value);

void storeSwapSync_16(uint16_f * addr, uint16_f value);

void swapEieio_16(uint16_f * addr);

uint32_f loadSwapEieio_32(uint32_f * addr);

unit32_f loadSwapSync_32(uint32_f * addr);

void storeSwapEieio_32(uint32_f * addr, uint32_f value);

void storeSwapSync_32(uint32_f * addr, uint32_f value);

void swapEieio_32 (uint32_f * addr);

void ioSync(void);

void eieio(void);
```

FEATURES

DKI

DESCRIPTION

Provides access to PowerPC DKI I/O routines, optimized to facilitate byte swapping and synchronization of input/output.

EXTENDED DESCRIPTION

The PowerPC DKI provides specific I/O routines to handle byte swapping and synchronization of I/O. These services are based on PowerPC specific instructions to store/load with byte swapping, and to enforce in-order execution of I/O.

Typically, these services are intended to be used by a host bus driver to handle different byte ordering between the PowerPC bus and the host bus, or to enforce sequential execution of I/O. PowerPC specific I/O services are defined below as sets of routines where the _xx suffix indicates the bit length of the data on which the service applies. This suffix may take one of the following values:

_16	for 16-bit data
_32	for 32-bit data

`loadSwapEieio_xx` loads data from a given address and returns the corresponding byte swapped value. A PowerPC `eieio` instruction is issued after the data is read, to enforce execution of subsequent I/O sequentially with respect to the current load. The *addr* argument specifies the address to read from.

`loadSwapSync_xx` behaves like `loadSwapEieio_xx` but issues a PowerPC `isync` instruction after that data is read. This ensures that all possible exceptions are taken into account before returning.

`storeSwapEieio_xx` stores, at a given address, the *value* byte swapped. A PowerPC `eieio` instruction is issued after the data is written, to enforce execution of subsequent I/O sequentially with respect to the current store. The *addr* argument specifies the address to write to.

`storeSwapSync_xx` behaves like `storeSwapEieio_xx` but issues a PowerPC `isync` instruction after that data is read. This ensures that all possible exceptions are taken into account before returning.

`swapEieio_xx` swaps in place the bytes of the data stored at a given address. A PowerPC `eieio` instruction is issued between reading the data and re-writing the byte-swapped data, to enforce execution in order. The *addr* argument specifies the address of the data to byte-swap.

`ioSync` enforces context synchronization, completion of all instructions, and completion of all exceptions before returning.

`eieio` enforces execution in order of I/O and is mapped to the corresponding PowerPC assembler instruction. This routine is intended for use in managing shared data structures, in doing memory-mapped I/O, and in preventing load/store combining operations in main memory. Refer to PowerPC programming manuals for more details.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
loadSwapEieio_xx	+	+	+	-
loadSwapSync_xx	+	+	+	-
storeSwapEieio_xx	+	+	+	-
storeSwapSync_xx	+	+	+	-
swapEieio_xx	+	+	+	-
ioSync	+	+	+	-
eieio	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`loadSwap_16`(9DKI) , `svDkiThreadCall`(9DKI)

NAME

load_sync_8_usparc, store_sync_8_usparc, loadSwap_sync_16_usparc, storeSwap_sync_16_usparc, load_sync_16_usparc, store_sync_16_usparc, loadSwap_sync_32_usparc, storeSwap_sync_32_usparc, load_sync_32_usparc, store_sync_32_usparc, loadSwap_sync_64_usparc, storeSwap_sync_64_usparc, load_sync_64_usparc, store_sync_64_usparc – UltraSparc specific i/o services

SYNOPSIS

```
#include <dki/f_dki.h>
uint8_f load_sync_8(uint8_f * addr);

void store_sync_8(uint8_f * addr, uint8_f value);

uint16_f loadSwap_sync_16(uint16_f * addr);

void storeSwap_sync_16(uint16_f * addr, uint16_f value);

uint16_f load_sync_16(uint16_f * addr);

void store_sync_16(uint16_f * addr, uint16_f value);

uint32_f loadSwap_sync_32(uint32_f * addr);

void storeSwap_sync_32(uint32_f * addr, uint32_f value);

uint32_f load_sync_32(uint32_f * addr);

void store_sync_32(uint32_f * addr, uint32_f value);

uint64_f loadSwap_sync_64(uint64_f * addr);

void storeSwap_sync_64(uint64_f * addr, uint64_f value);

uint64_f load_sync_64(uint64_f * addr);

void store_sync_64(uint64_f * addr, uint64_f value);
```

FEATURES

DKI

DESCRIPTION

The UltraSPARC DKI provides specific I/O routines optimized to handle byte swapping. These services are analogous to the generic ones except the `membar #Sync` instruction is issued each time the data is read/written from/to the memory. Note that this allows recovery from an asynchronous exception caused by these types of service routines.

Typically, these services are intended to be used by a host bus driver to handle different byte ordering between the processor bus and the host bus.

In addtion, the UltraSPARC DKI provides specific synchronized I/O routines. These routines do not perform the byte swapping but they are safe and allow recovery from an asynchronous exception caused by this type of service routine.

Specific I/O services are defined below as sets of routines where the `_xx` suffix indicates the bit length of the data on which the service applies. This suffix may take one of the following values:

- _8 for 8-bit data
- _16 for 16-bit data
- _32 for 32-bit data
- _64 for 64-bit data

Note that the _8 suffix is only applied to the load_sync_xx and store_sync_xx routines.

loadSwap_sync_xx loadSwap_sync_xx loads data from a given address and returns the corresponding byte swapped value. The membar #Sync instruction is issued once the data has been loaded.

The *addr* argument specifies the address from which to read.

storeSwap_sync_xx storeSwap_sync_xx stores into a given address the *value* byte swapped. The membar #Sync instruction is issued once the data is stored.

The *addr* argument specifies the address to which to write.

load_sync_xx load_sync_xx loads data from a given address. The membar #Sync instruction is issued once the data is loaded.

The *addr* argument specifies the address from which to read.

storeSwap_sync_xx storeSwap_sync_xx stores into a given address the *value* . The membar #Sync instruction is issued once the data is stored.

The *addr* argument specifies the address to which to write.

ALLOWED CALLING CONTEXTS

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
loadSwap_sync_xx	+	+	+	-
storeSwap_sync_xx	+	+	+	-
load_sync_xx	+	+	+	-
store_sync_xx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `svAsyncExcepAttach_usparc>`(9DKI) , `svAsyncExcepDetach_usparc>`(9DKI)

NAME

load_sync_8_usparc, store_sync_8_usparc, loadSwap_sync_16_usparc, storeSwap_sync_16_usparc, load_sync_16_usparc, store_sync_16_usparc, loadSwap_sync_32_usparc, storeSwap_sync_32_usparc, load_sync_32_usparc, store_sync_32_usparc, loadSwap_sync_64_usparc, storeSwap_sync_64_usparc, load_sync_64_usparc, store_sync_64_usparc – UltraSparc specific i/o services

SYNOPSIS

```
#include <dki/f_dki.h>
uint8_f load_sync_8(uint8_f * addr);

void store_sync_8(uint8_f * addr, uint8_f value);

uint16_f loadSwap_sync_16(uint16_f * addr);

void storeSwap_sync_16(uint16_f * addr, uint16_f value);

uint16_f load_sync_16(uint16_f * addr);

void store_sync_16(uint16_f * addr, uint16_f value);

uint32_f loadSwap_sync_32(uint32_f * addr);

void storeSwap_sync_32(uint32_f * addr, uint32_f value);

uint32_f load_sync_32(uint32_f * addr);

void store_sync_32(uint32_f * addr, uint32_f value);

uint64_f loadSwap_sync_64(uint64_f * addr);

void storeSwap_sync_64(uint64_f * addr, uint64_f value);

uint64_f load_sync_64(uint64_f * addr);

void store_sync_64(uint64_f * addr, uint64_f value);
```

FEATURES

DKI

DESCRIPTION

The UltraSPARC DKI provides specific I/O routines optimized to handle byte swapping. These services are analogous to the generic ones except the `membar #Sync` instruction is issued each time the data is read/written from/to the memory. Note that this allows recovery from an asynchronous exception caused by these types of service routines.

Typically, these services are intended to be used by a host bus driver to handle different byte ordering between the processor bus and the host bus.

In addtion, the UltraSPARC DKI provides specific synchronized I/O routines. These routines do not perform the byte swapping but they are safe and allow recovery from an asynchronous exception caused by this type of service routine.

Specific I/O services are defined below as sets of routines where the `_xx` suffix indicates the bit length of the data on which the service applies. This suffix may take one of the following values:

- _8 for 8-bit data
- _16 for 16-bit data
- _32 for 32-bit data
- _64 for 64-bit data

Note that the _8 suffix is only applied to the load_sync_xx and store_sync_xx routines.

loadSwap_sync_xx loadSwap_sync_xx loads data from a given address and returns the corresponding byte swapped value. The membar #Sync instruction is issued once the data has been loaded.

The *addr* argument specifies the address from which to read.

storeSwap_sync_xx storeSwap_sync_xx stores into a given address the *value* byte swapped. The membar #Sync instruction is issued once the data is stored.

The *addr* argument specifies the address to which to write.

load_sync_xx load_sync_xx loads data from a given address. The membar #Sync instruction is issued once the data is loaded.

The *addr* argument specifies the address from which to read.

storeSwap_sync_xx storeSwap_sync_xx stores into a given address the *value* . The membar #Sync instruction is issued once the data is stored.

The *addr* argument specifies the address to which to write.

ALLOWED CALLING CONTEXTS

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
loadSwap_sync_xx	+	+	+	-
storeSwap_sync_xx	+	+	+	-
load_sync_xx	+	+	+	-
store_sync_xx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `svAsyncExcepAttach_usparc>`(9DKI) , `svAsyncExcepDetach_usparc>`(9DKI)

NAME

load_sync_8_usparc, store_sync_8_usparc, loadSwap_sync_16_usparc, storeSwap_sync_16_usparc, load_sync_16_usparc, store_sync_16_usparc, loadSwap_sync_32_usparc, storeSwap_sync_32_usparc, load_sync_32_usparc, store_sync_32_usparc, loadSwap_sync_64_usparc, storeSwap_sync_64_usparc, load_sync_64_usparc, store_sync_64_usparc – UltraSparc specific i/o services

SYNOPSIS

```
#include <dki/f_dki.h>
uint8_f load_sync_8(uint8_f * addr);

void store_sync_8(uint8_f * addr, uint8_f value);

uint16_f loadSwap_sync_16(uint16_f * addr);

void storeSwap_sync_16(uint16_f * addr, uint16_f value);

uint16_f load_sync_16(uint16_f * addr);

void store_sync_16(uint16_f * addr, uint16_f value);

uint32_f loadSwap_sync_32(uint32_f * addr);

void storeSwap_sync_32(uint32_f * addr, uint32_f value);

uint32_f load_sync_32(uint32_f * addr);

void store_sync_32(uint32_f * addr, uint32_f value);

uint64_f loadSwap_sync_64(uint64_f * addr);

void storeSwap_sync_64(uint64_f * addr, uint64_f value);

uint64_f load_sync_64(uint64_f * addr);

void store_sync_64(uint64_f * addr, uint64_f value);
```

FEATURES

DKI

DESCRIPTION

The UltraSPARC DKI provides specific I/O routines optimized to handle byte swapping. These services are analogous to the generic ones except the `membar #Sync` instruction is issued each time the data is read/written from/to the memory. Note that this allows recovery from an asynchronous exception caused by these types of service routines.

Typically, these services are intended to be used by a host bus driver to handle different byte ordering between the processor bus and the host bus.

In addtion, the UltraSPARC DKI provides specific synchronized I/O routines. These routines do not perform the byte swapping but they are safe and allow recovery from an asynchronous exception caused by this type of service routine.

Specific I/O services are defined below as sets of routines where the `_xx` suffix indicates the bit length of the data on which the service applies. This suffix may take one of the following values:

- _8 for 8-bit data
- _16 for 16-bit data
- _32 for 32-bit data
- _64 for 64-bit data

Note that the _8 suffix is only applied to the load_sync_xx and store_sync_xx routines.

loadSwap_sync_xx loadSwap_sync_xx loads data from a given address and returns the corresponding byte swapped value. The membar #Sync instruction is issued once the data has been loaded.

The *addr* argument specifies the address from which to read.

storeSwap_sync_xx storeSwap_sync_xx stores into a given address the *value* byte swapped. The membar #Sync instruction is issued once the data is stored.

The *addr* argument specifies the address to which to write.

load_sync_xx load_sync_xx loads data from a given address. The membar #Sync instruction is issued once the data is loaded.

The *addr* argument specifies the address from which to read.

storeSwap_sync_xx storeSwap_sync_xx stores into a given address the *value* . The membar #Sync instruction is issued once the data is stored.

The *addr* argument specifies the address to which to write.

ALLOWED CALLING CONTEXTS

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
loadSwap_sync_xx	+	+	+	-
storeSwap_sync_xx	+	+	+	-
load_sync_xx	+	+	+	-
store_sync_xx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `svAsyncExcepAttach_usparc>`(9DKI) , `svAsyncExcepDetach_usparc>`(9DKI)

NAME

load_sync_8_usparc, store_sync_8_usparc, loadSwap_sync_16_usparc, storeSwap_sync_16_usparc, load_sync_16_usparc, store_sync_16_usparc, loadSwap_sync_32_usparc, storeSwap_sync_32_usparc, load_sync_32_usparc, store_sync_32_usparc, loadSwap_sync_64_usparc, storeSwap_sync_64_usparc, load_sync_64_usparc, store_sync_64_usparc – UltraSparc specific i/o services

SYNOPSIS

```
#include <dki/f_dki.h>
uint8_f load_sync_8(uint8_f * addr);

void store_sync_8(uint8_f * addr, uint8_f value);

uint16_f loadSwap_sync_16(uint16_f * addr);

void storeSwap_sync_16(uint16_f * addr, uint16_f value);

uint16_f load_sync_16(uint16_f * addr);

void store_sync_16(uint16_f * addr, uint16_f value);

uint32_f loadSwap_sync_32(uint32_f * addr);

void storeSwap_sync_32(uint32_f * addr, uint32_f value);

uint32_f load_sync_32(uint32_f * addr);

void store_sync_32(uint32_f * addr, uint32_f value);

uint64_f loadSwap_sync_64(uint64_f * addr);

void storeSwap_sync_64(uint64_f * addr, uint64_f value);

uint64_f load_sync_64(uint64_f * addr);

void store_sync_64(uint64_f * addr, uint64_f value);
```

FEATURES

DKI

DESCRIPTION

The UltraSPARC DKI provides specific I/O routines optimized to handle byte swapping. These services are analogous to the generic ones except the `membar #Sync` instruction is issued each time the data is read/written from/to the memory. Note that this allows recovery from an asynchronous exception caused by these types of service routines.

Typically, these services are intended to be used by a host bus driver to handle different byte ordering between the processor bus and the host bus.

In addtion, the UltraSPARC DKI provides specific synchronized I/O routines. These routines do not perform the byte swapping but they are safe and allow recovery from an asynchronous exception caused by this type of service routine.

Specific I/O services are defined below as sets of routines where the `_xx` suffix indicates the bit length of the data on which the service applies. This suffix may take one of the following values:

- _8 for 8-bit data
- _16 for 16-bit data
- _32 for 32-bit data
- _64 for 64-bit data

Note that the _8 suffix is only applied to the load_sync_xx and store_sync_xx routines.

loadSwap_sync_xx loadSwap_sync_xx loads data from a given address and returns the corresponding byte swapped value. The membar #Sync instruction is issued once the data has been loaded.

The *addr* argument specifies the address from which to read.

storeSwap_sync_xx storeSwap_sync_xx stores into a given address the *value* byte swapped. The membar #Sync instruction is issued once the data is stored.

The *addr* argument specifies the address to which to write.

load_sync_xx load_sync_xx loads data from a given address. The membar #Sync instruction is issued once the data is loaded.

The *addr* argument specifies the address from which to read.

storeSwap_sync_xx storeSwap_sync_xx stores into a given address the *value* . The membar #Sync instruction is issued once the data is stored.

The *addr* argument specifies the address to which to write.

ALLOWED CALLING CONTEXTS

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
loadSwap_sync_xx	+	+	+	-
storeSwap_sync_xx	+	+	+	-
load_sync_xx	+	+	+	-
store_sync_xx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `svAsyncExcepAttach_usparc>`(9DKI) , `svAsyncExcepDetach_usparc>`(9DKI)

NAME

load_sync_8_usparc, store_sync_8_usparc, loadSwap_sync_16_usparc, storeSwap_sync_16_usparc, load_sync_16_usparc, store_sync_16_usparc, loadSwap_sync_32_usparc, storeSwap_sync_32_usparc, load_sync_32_usparc, store_sync_32_usparc, loadSwap_sync_64_usparc, storeSwap_sync_64_usparc, load_sync_64_usparc, store_sync_64_usparc – UltraSparc specific i/o services

SYNOPSIS

```
#include <dki/f_dki.h>
uint8_f load_sync_8(uint8_f * addr);

void store_sync_8(uint8_f * addr, uint8_f value);

uint16_f loadSwap_sync_16(uint16_f * addr);

void storeSwap_sync_16(uint16_f * addr, uint16_f value);

uint16_f load_sync_16(uint16_f * addr);

void store_sync_16(uint16_f * addr, uint16_f value);

uint32_f loadSwap_sync_32(uint32_f * addr);

void storeSwap_sync_32(uint32_f * addr, uint32_f value);

uint32_f load_sync_32(uint32_f * addr);

void store_sync_32(uint32_f * addr, uint32_f value);

uint64_f loadSwap_sync_64(uint64_f * addr);

void storeSwap_sync_64(uint64_f * addr, uint64_f value);

uint64_f load_sync_64(uint64_f * addr);

void store_sync_64(uint64_f * addr, uint64_f value);
```

FEATURES

DKI

DESCRIPTION

The UltraSPARC DKI provides specific I/O routines optimized to handle byte swapping. These services are analogous to the generic ones except the `membar #Sync` instruction is issued each time the data is read/written from/to the memory. Note that this allows recovery from an asynchronous exception caused by these types of service routines.

Typically, these services are intended to be used by a host bus driver to handle different byte ordering between the processor bus and the host bus.

In addtion, the UltraSPARC DKI provides specific synchronized I/O routines. These routines do not perform the byte swapping but they are safe and allow recovery from an asynchronous exception caused by this type of service routine.

Specific I/O services are defined below as sets of routines where the `_xx` suffix indicates the bit length of the data on which the service applies. This suffix may take one of the following values:

- _8 for 8-bit data
- _16 for 16-bit data
- _32 for 32-bit data
- _64 for 64-bit data

Note that the _8 suffix is only applied to the load_sync_xx and store_sync_xx routines.

loadSwap_sync_xx loadSwap_sync_xx loads data from a given address and returns the corresponding byte swapped value. The membar #Sync instruction is issued once the data has been loaded.

The *addr* argument specifies the address from which to read.

storeSwap_sync_xx storeSwap_sync_xx stores into a given address the *value* byte swapped. The membar #Sync instruction is issued once the data is stored.

The *addr* argument specifies the address to which to write.

load_sync_xx load_sync_xx loads data from a given address. The membar #Sync instruction is issued once the data is loaded.

The *addr* argument specifies the address from which to read.

storeSwap_sync_xx storeSwap_sync_xx stores into a given address the *value* . The membar #Sync instruction is issued once the data is stored.

The *addr* argument specifies the address to which to write.

ALLOWED CALLING CONTEXTS

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
loadSwap_sync_xx	+	+	+	-
storeSwap_sync_xx	+	+	+	-
load_sync_xx	+	+	+	-
store_sync_xx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `svAsyncExcepAttach_usparc>`(9DKI) , `svAsyncExcepDetach_usparc>`(9DKI)

NAME

load_sync_8_usparc, store_sync_8_usparc, loadSwap_sync_16_usparc, storeSwap_sync_16_usparc, load_sync_16_usparc, store_sync_16_usparc, loadSwap_sync_32_usparc, storeSwap_sync_32_usparc, load_sync_32_usparc, store_sync_32_usparc, loadSwap_sync_64_usparc, storeSwap_sync_64_usparc, load_sync_64_usparc, store_sync_64_usparc – UltraSparc specific i/o services

SYNOPSIS

```
#include <dki/f_dki.h>
uint8_f load_sync_8(uint8_f * addr);

void store_sync_8(uint8_f * addr, uint8_f value);

uint16_f loadSwap_sync_16(uint16_f * addr);

void storeSwap_sync_16(uint16_f * addr, uint16_f value);

uint16_f load_sync_16(uint16_f * addr);

void store_sync_16(uint16_f * addr, uint16_f value);

uint32_f loadSwap_sync_32(uint32_f * addr);

void storeSwap_sync_32(uint32_f * addr, uint32_f value);

uint32_f load_sync_32(uint32_f * addr);

void store_sync_32(uint32_f * addr, uint32_f value);

uint64_f loadSwap_sync_64(uint64_f * addr);

void storeSwap_sync_64(uint64_f * addr, uint64_f value);

uint64_f load_sync_64(uint64_f * addr);

void store_sync_64(uint64_f * addr, uint64_f value);
```

FEATURES

DKI

DESCRIPTION

The UltraSPARC DKI provides specific I/O routines optimized to handle byte swapping. These services are analogous to the generic ones except the `membar #Sync` instruction is issued each time the data is read/written from/to the memory. Note that this allows recovery from an asynchronous exception caused by these types of service routines.

Typically, these services are intended to be used by a host bus driver to handle different byte ordering between the processor bus and the host bus.

In addtion, the UltraSPARC DKI provides specific synchronized I/O routines. These routines do not perform the byte swapping but they are safe and allow recovery from an asynchronous exception caused by this type of service routine.

Specific I/O services are defined below as sets of routines where the `_xx` suffix indicates the bit length of the data on which the service applies. This suffix may take one of the following values:

- _8 for 8-bit data
- _16 for 16-bit data
- _32 for 32-bit data
- _64 for 64-bit data

Note that the _8 suffix is only applied to the load_sync_xx and store_sync_xx routines.

loadSwap_sync_xx loadSwap_sync_xx loads data from a given address and returns the corresponding byte swapped value. The membar #Sync instruction is issued once the data has been loaded.

The *addr* argument specifies the address from which to read.

storeSwap_sync_xx storeSwap_sync_xx stores into a given address the *value* byte swapped. The membar #Sync instruction is issued once the data is stored.

The *addr* argument specifies the address to which to write.

load_sync_xx load_sync_xx loads data from a given address. The membar #Sync instruction is issued once the data is loaded.

The *addr* argument specifies the address from which to read.

storeSwap_sync_xx storeSwap_sync_xx stores into a given address the *value* . The membar #Sync instruction is issued once the data is stored.

The *addr* argument specifies the address to which to write.

ALLOWED CALLING CONTEXTS

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
loadSwap_sync_xx	+	+	+	-
storeSwap_sync_xx	+	+	+	-
load_sync_xx	+	+	+	-
store_sync_xx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `svAsyncExcepAttach_usparc>`(9DKI) ,
`svAsyncExcepDetach_usparc>`(9DKI)

NAME

load_sync_8_usparc, store_sync_8_usparc, loadSwap_sync_16_usparc, storeSwap_sync_16_usparc, load_sync_16_usparc, store_sync_16_usparc, loadSwap_sync_32_usparc, storeSwap_sync_32_usparc, load_sync_32_usparc, store_sync_32_usparc, loadSwap_sync_64_usparc, storeSwap_sync_64_usparc, load_sync_64_usparc, store_sync_64_usparc – UltraSparc specific i/o services

SYNOPSIS

```
#include <dki/f_dki.h>
uint8_f load_sync_8(uint8_f * addr);

void store_sync_8(uint8_f * addr, uint8_f value);

uint16_f loadSwap_sync_16(uint16_f * addr);

void storeSwap_sync_16(uint16_f * addr, uint16_f value);

uint16_f load_sync_16(uint16_f * addr);

void store_sync_16(uint16_f * addr, uint16_f value);

uint32_f loadSwap_sync_32(uint32_f * addr);

void storeSwap_sync_32(uint32_f * addr, uint32_f value);

uint32_f load_sync_32(uint32_f * addr);

void store_sync_32(uint32_f * addr, uint32_f value);

uint64_f loadSwap_sync_64(uint64_f * addr);

void storeSwap_sync_64(uint64_f * addr, uint64_f value);

uint64_f load_sync_64(uint64_f * addr);

void store_sync_64(uint64_f * addr, uint64_f value);
```

FEATURES

DKI

DESCRIPTION

The UltraSPARC DKI provides specific I/O routines optimized to handle byte swapping. These services are analogous to the generic ones except the `membar #Sync` instruction is issued each time the data is read/written from/to the memory. Note that this allows recovery from an asynchronous exception caused by these types of service routines.

Typically, these services are intended to be used by a host bus driver to handle different byte ordering between the processor bus and the host bus.

In addtion, the UltraSPARC DKI provides specific synchronized I/O routines. These routines do not perform the byte swapping but they are safe and allow recovery from an asynchronous exception caused by this type of service routine.

Specific I/O services are defined below as sets of routines where the `_xx` suffix indicates the bit length of the data on which the service applies. This suffix may take one of the following values:

- _8 for 8-bit data
- _16 for 16-bit data
- _32 for 32-bit data
- _64 for 64-bit data

Note that the _8 suffix is only applied to the load_sync_xx and store_sync_xx routines.

loadSwap_sync_xx loadSwap_sync_xx loads data from a given address and returns the corresponding byte swapped value. The membar #Sync instruction is issued once the data has been loaded.

The *addr* argument specifies the address from which to read.

storeSwap_sync_xx storeSwap_sync_xx stores into a given address the *value* byte swapped. The membar #Sync instruction is issued once the data is stored.

The *addr* argument specifies the address to which to write.

load_sync_xx load_sync_xx loads data from a given address. The membar #Sync instruction is issued once the data is loaded.

The *addr* argument specifies the address from which to read.

storeSwap_sync_xx storeSwap_sync_xx stores into a given address the *value* . The membar #Sync instruction is issued once the data is stored.

The *addr* argument specifies the address to which to write.

ALLOWED CALLING CONTEXTS

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
loadSwap_sync_xx	+	+	+	-
storeSwap_sync_xx	+	+	+	-
load_sync_xx	+	+	+	-
store_sync_xx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `svAsyncExcepAttach_usparc>`(9DKI) , `svAsyncExcepDetach_usparc>`(9DKI)

NAME

svAsyncExcepAttach, svAsyncExcepDetach – asynchronous exceptions management

FEATURES

DKI

DESCRIPTION

Provides asynchronous exception management services.

See the architecture specific man pages:

- `svAsyncExcepAttach_usparc`(9DKI)
- `svAsyncExcepDetach_usparc`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

svAsyncExcepAttach_usparc, svAsyncExcepDetach_usparc – UltraSPARC aynchronous exceptions management

SYNOPSIS

```
#include <dki/f_dki.h>
KnError svAsyncExcepAttach(CpuExcepHandler excepHandler, void * excepCookie,
CpuExcepId * excepId);

void svAsyncExcepDetach(CpuExcepId * excepId);
```

FEATURES

DKI

DESCRIPTION

svAsyncExcepAttach

Provides UltraSPARC asynchronous exceptions management services. `svAsyncExcepAttach` attaches a given exception handler to the UltraSPARC asynchronous exceptions.

The *excepHandler* argument specifies the handler to call back when an asynchronous exception occurs.

The *excepCookie* argument specifies a parameter to pass back to the handler when called. It is opaque to the microkernel.

On succes, `K_OK` is returned and an identifier for the attached exception handler is also returned in *excepId* . This identifier must be used in subsequent invocations of `svAsyncExcepDetach` .

On failure, an error code is returned as follows:

`K_ENOMEM` The system is out of memory.

```
typedef CpuExcepStatus (*CpuExcepHandler) (void*    cookie,
                                           uint64_f afsr,
                                           uint64_f afar);
```

The asynchronous exception handler is invoked by the microkernel when one of the follwing bits is set in the asynchronous fault status register (AFSR):

`AFSR_TO` Time-out from system bus.

`AFSR_BERR` Bus error from system bus.

`AFSR_CP` DMA external cache parity error.

The asynchronous exception handler is called with masked CPU interrupts and disabled asynchronous exceptions. In addition, the CPU instruction and data caches are disabled. A cookie specified in `svAsyncExcepAttach` is passed back to the exception handler as the first argument. The asynchronous fault status and address registers are given to the handler.

The exception handler should return either CPU_EXCEP_CLAIMED or CPU_EXCEP_UNCLAIMED . The CPU_EXCEP_CLAIMED value is returned by the handler in cases where the handler detects that the exception is due to an I/O or DMA access on the underlying bus, and the exception is recoverable. For instance, the host bus driver exception handler simply notifies an appropriate child driver (via the error handler invocation) if the exception is due to an I/O access initiated by this driver. In this case, the exception is recoverable and the host bus driver asks the microkernel to continue execution returning CPU_EXCEP_CLAIMED .

Note that multiple handlers may be attached to the asynchronous exceptions. When an exception occurs, the microkernel invokes all handlers sequentially iterating through the handlers list. Once CPU_EXCEP_CLAIMED is returned by a handler, the iteration is aborted and the microkernel tries to continue the execution. It re-enables the CPU instruction and data caches and asynchronous exceptions, sets NPC to PC+4 and returns from the exception. Note that NPC is incorrect when an asynchronous exception occurs, and therefore, it should be corrected prior to continuing the execution. As a consequence, accesses to the bus I/O space should be synchronized by issuing a membar #Sync instruction after a load/store one. Also, the membar instruction should not be put in a delayed slot. Otherwise, asynchronous exceptions related to the I/O accesses are not granted to be recoverable. Note that the specific I/O service routines provided by the UltrsSPARC DKI allow drivers to recover from an asynchronous exception caused by this type of service routine.

The microkernel considers the exception as unrecoverable and panics, if all exception handlers return CPU_EXCEP_UNCLAIMED .

svAsyncExcepDetach

svAsyncExcepDetach detaches the exception handler previously connected by svAsyncExcepAttach .

The *excepId* argument identifies the attached exception handler, previously returned by svAsyncExcepAttach .

ALLOWED CALLING CONTEXTS

The following table specifies the contexts in which a caller is allowed to invoke each exceptions management service.

Services	Base level	DKI thread	Interrupt	Blocking
svAsyncExcepAttach	+	+	-	+
svAsyncExcepDetach	+	+	-	+

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

load_sync_8_usparc(9DKI) , store_sync_8_usparc(9DKI) , loadSwap_sync_16_usparc(9DKI) , storeSwap_sync_16_usparc(9DKI) , load_sync_16_usparc(9DKI) , store_sync_16_usparc(9DKI) , loadSwap_sync_32_usparc(9DKI) , storeSwap_sync_32_usparc(9DKI) , load_sync_32_usparc(9DKI) , store_sync_32_usparc(9DKI) , loadSwap_sync_64_usparc(9DKI) , storeSwap_sync_64_usparc(9DKI) , load_sync_64_usparc(9DKI) , store_sync_64_usparc(9DKI) ,

NAME

svAsyncExcepAttach, svAsyncExcepDetach – asynchronous exceptions management

FEATURES

DKI

DESCRIPTION

Provides asynchronous exception management services.

See the architecture specific man pages:

- `svAsyncExcepAttach_usparc`(9DKI)
- `svAsyncExcepDetach_usparc`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

svAsyncExcepAttach_usparc, svAsyncExcepDetach_usparc – UltraSPARC aynchronous exceptions management

SYNOPSIS

```
#include <dki/f_dki.h>
KnError svAsyncExcepAttach(CpuExcepHandler excepHandler, void * excepCookie,
CpuExcepId * excepId);

void svAsyncExcepDetach(CpuExcepId * excepId);
```

FEATURES

DKI

DESCRIPTION

svAsyncExcepAttach

Provides UltraSPARC asynchronous exceptions management services. `svAsyncExcepAttach` attaches a given exception handler to the UltraSPARC asynchronous exceptions.

The *excepHandler* argument specifies the handler to call back when an asynchronous exception occurs.

The *excepCookie* argument specifies a parameter to pass back to the handler when called. It is opaque to the microkernel.

On succes, `K_OK` is returned and an identifier for the attached exception handler is also returned in *excepId* . This identifier must be used in subsequent invocations of `svAsyncExcepDetach` .

On failure, an error code is returned as follows:

`K_ENOMEM` The system is out of memory.

```
typedef CpuExcepStatus (*CpuExcepHandler) (void*    cookie,
                                           uint64_f afsr,
                                           uint64_f afar);
```

The asynchronous exception handler is invoked by the microkernel when one of the follwing bits is set in the asynchronous fault status register (AFSR):

`AFSR_TO` Time-out from system bus.

`AFSR_BERR` Bus error from system bus.

`AFSR_CP` DMA external cache parity error.

The asynchronous exception handler is called with masked CPU interrupts and disabled asynchronous exceptions. In addition, the CPU instruction and data caches are disabled. A cookie specified in `svAsyncExcepAttach` is passed back to the exception handler as the first argument. The asynchronous fault status and address registers are given to the handler.

The exception handler should return either CPU_EXCEP_CLAIMED or CPU_EXCEP_UNCLAIMED . The CPU_EXCEP_CLAIMED value is returned by the handler in cases where the handler detects that the exception is due to an I/O or DMA access on the underlying bus, and the exception is recoverable. For instance, the host bus driver exception handler simply notifies an appropriate child driver (via the error handler invocation) if the exception is due to an I/O access initiated by this driver. In this case, the exception is recoverable and the host bus driver asks the microkernel to continue execution returning CPU_EXCEP_CLAIMED .

Note that multiple handlers may be attached to the asynchronous exceptions. When an exception occurs, the microkernel invokes all handlers sequentially iterating through the handlers list. Once CPU_EXCEP_CLAIMED is returned by a handler, the iteration is aborted and the microkernel tries to continue the execution. It re-enables the CPU instruction and data caches and asynchronous exceptions, sets NPC to PC+4 and returns from the exception. Note that NPC is incorrect when an asynchronous exception occurs, and therefore, it should be corrected prior to continuing the execution. As a consequence, accesses to the bus I/O space should be synchronized by issuing a membar #Sync instruction after a load/store one. Also, the membar instruction should not be put in a delayed slot. Otherwise, asynchronous exceptions related to the I/O accesses are not granted to be recoverable. Note that the specific I/O service routines provided by the UltrsSPARC DKI allow drivers to recover from an asynchronous exception caused by this type of service routine.

The microkernel considers the exception as unrecoverable and panics, if all exception handlers return CPU_EXCEP_UNCLAIMED .

svAsyncExcepDetach

svAsyncExcepDetach detaches the exception handler previously connected by svAsyncExcepAttach .

The *excepId* argument identifies the attached exception handler, previously returned by svAsyncExcepAttach .

ALLOWED CALLING CONTEXTS

The following table specifies the contexts in which a caller is allowed to invoke each exceptions management service.

Services	Base level	DKI thread	Interrupt	Blocking
svAsyncExcepAttach	+	+	-	+
svAsyncExcepDetach	+	+	-	+

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

load_sync_8_usparc(9DKI), store_sync_8_usparc(9DKI), loadSwap_sync_16_usparc(9DKI), storeSwap_sync_16_usparc(9DKI), load_sync_16_usparc(9DKI), store_sync_16_usparc(9DKI), loadSwap_sync_32_usparc(9DKI), storeSwap_sync_32_usparc(9DKI), load_sync_32_usparc(9DKI), store_sync_32_usparc(9DKI), loadSwap_sync_64_usparc(9DKI), storeSwap_sync_64_usparc(9DKI), load_sync_64_usparc(9DKI), store_sync_64_usparc(9DKI),

NAME

svDeviceRegister, svDeviceAlloc, svDeviceFree, svDeviceUnregister, svDeviceEvent, svDeviceLookup, svDeviceEntry, svDeviceRelease – device registry operations

SYNOPSIS

```
#include <dki/dki.h>
void svDeviceRegister(DevRegId dev_id);

DevRegId svDeviceAlloc(DevRegEntry * entry, unsigned int version, Bool shared,
DevRelHandler handler);

KnError svDeviceUnregister(DevRegId dev_id);

void svDeviceEvent(DevRegId dev_id, DevEvent event, void* arg);

void svDeviceFree(DevRegId dev_id);

KnError svDeviceLookup(char * dev_class, unsigned int dev_version, unsigned int
dev_unit, DevEventHandler cli_handler, void * cli_cookie, DevClientId * cli_id);

DevRegEntry * svDeviceEntry(DevClientId cli_id);

void svDeviceRelease(DevClientId cli_id);
typedef struct {
        char* dev_class;
        void* dev_ops;
        void* dev_id;
        DevNode dev_node;
    } DevRegEntry;
```

FEATURES

DKI

DESCRIPTION

Provides access to microkernel services implementing the `device registry`.

EXTENDED DESCRIPTION

The `device registry` microkernel module implements a data base of driver instances servicing devices currently supported by the system. The `device registry` data base is populated by drivers which perform self-registration (using `svDeviceRegister`) at device initialization time. The `device registry` data base is accessed by driver clients in order to obtain a pointer to the driver instance servicing a given (logical) device. The `device registry` API is described below in detail.

Note that only the `svDeviceLookup`, `svDeviceRelease` and `svDeviceEntry` microkernel calls should be used by driver clients. The rest of the API is dedicated to device drivers.

`svDeviceAlloc` allocates a device registry entry for a given device driver instance. Note that the entry is allocated in an invalid state. This means that the entry is not included in the registry (it is not visible to clients via `svDeviceLookup`).

The entry becomes valid (visible to clients) when a subsequent `svDeviceRegister` is invoked. On the other hand, the driver is allowed to call `svDeviceEvent` on this type of entry. Shut down events signaled on an invalid entry are memorized by the device registry and they are processed when the entry becomes valid (that is, when `svDeviceRegister` is called).

The *entry* argument points to the `DevRegEntry` structure which designates the device driver instance. The DevRegEntry structure is described below:

dev_class	Points to a string specifying the device class name.
dev_ops	Points to a structure of driver service routines.
dev_id	Points to a handle which is usually passed back to the driver (as first argument) each time a driver service routine is invoked.
dev_node	Specifies the device node in the tree which is serviced by the driver. Note that a driver client must switch to the DKI thread context when accessing the device tree structure (see section “Device Node Properties”).

The *version* argument specifies the driver interface version implemented by the device driver instance.

The *dev_class* field and *version* argument specify APIs implemented by the driver. This type of API is provided to driver clients as a structure of indirect functions implementing the API service routines. For instance, the *"uart"* device API is specified by the `UartDevOps` structure, the *"timer"* device API is specified by the `TimerDevOps` structure. A pointer to this type of structure is obtained by a device driver client from the *dev_ops* field of the device registry entry.

The *shared* argument specifies whether the device driver instance may be shared between multiple clients. In other words, it indicates whether the registered instance may be looked up multiple times.

The *handler* argument specifies the driver handler which is invoked by the device registry module as an acknowledgement to a shut-down event (see `svDeviceEvent`). *handler* is called when the last reference to the device registry

entry goes away and a shut-down event has been previously signalled on the entry. *entry* is passed back to the driver as the argument of *handler* .

In case of success, a non zero *DevRegId* is returned, otherwise `NULL` is returned.

Device registry assigns a logical unit number to the physical device. The logical unit number is unique within *dev_class* . The device registry handles a logical unit counter per class. All counters are initialized to zero. Each time a new device entry's is allocated, the current counter value is assigned to the entry logical unit number and the counter is incremented. Thus, the logical device order within a class corresponds to the allocation order.

The logical unit number is mainly used by device driver clients to iterate through a given device class looking for a certain device. Using a valid logical unit number, a client is able to access the corresponding device registry entry, in particular, the *dev_node* field which points to the device node. The physical device corresponding to a given logical unit number may then be detected by the device node properties and the node position within the device tree.

Note - The device driver may allocate (and then register) multiple entries in order to declare different classes (APIs) for the same physical device. For instance, a driver may implement orthogonal sets of operations like power management which are additional to the main device functionality. Another example is a device driver servicing a multi-function device. This type of driver would register multiple entries: one entry per device function. From the client's point of view, this type of device would be visible as multiple (mono-function) independent devices.

`svDeviceRegister` adds a given device registry entry to the registry. The entry must be previously allocated by `svDeviceAlloc` . The entry becomes valid (and therefore visible for clients) only if there is no shut-down event signalled on the entry. Otherwise, the entry remains invalid and the device registry module invokes a handler previously specified via `svDeviceAlloc` . A pointer to the `DevRegEntry` structure (previously specified via `svDeviceAlloc`) is passed back to the driver as the argument of *handler* . The *handler* routine is called in the DKI thread context.

Note - When *handler* is called the entry is no longer registered. Thus, it is useless to invoke `svDeviceUnregister` once *handler* has been called. On the other hand, in order to free memory resources, the driver should release the device registry entry invoking `svDeviceFree` once the entry is no longer used by the driver. Typically, the entry is no longer used by the driver when the connection to the parent bus / nexus driver is closed, and the parent driver is unable to invoke a call-back handler signalling a bus / nexus event.

`svDeviceUnregister` removes the device entry from the device registry (if the entry is valid). The device entry being removed is specified by the *dev_id* argument. `svDeviceUnregister` returns the following results when a valid entry is specified:

K_OK	The device entry has been successfully removed from the registry.
K_EBUSY	The device entry has not been removed from the registry because it is locked by a driver client. In other words, the `svDeviceLookup/Release` pairs are not balanced.

`svDeviceUnregister` returns `K_EBUSY` when an invalid entry is specified. The driver must remove all related device entries from the registry prior to being unloaded from the system. Typically, `svDeviceUnregister` is issued by the device driver (for each device driver instance) when the driver code is requested to be unloaded via `drv_unload` . The driver unload fails when the driver code is still being used by the system.

`svDeviceEvent` notifies the device registry module that a shutdown event has occurred (for example, that a hot-pluggable device has been removed). The shutdown event basically means that the driver instance should no longer be used by driver clients and the device entry must be removed from the registry (that is, the driver instance will disappear). The shutdown event is immediately propagated to all driver clients in order to stop the device usage and to release the device entry as soon as possible. The device is specified by the *dev_id* argument.

`svDeviceEvent` propagates the shutdown event invoking the *event_handler* routines specified by the driver clients in `svDeviceLookup` . The *event* and *arg* arguments are opaque for the device registry. They are passed (as arguments) to the client event handlers. *event* specifies the reason for the shut-down as follows:

DEV_EVENT_SHUTDOWN	Normal device shut down
DEV_EVENT_REMOVAL	Hot-plug (surprise) device removal

arg is *event* specific.

All driver clients are requested to release the device entry (as soon as possible) invoking the `svDeviceRelease` routine. Note that, prior to the shutdown event propagation, `svDeviceEvent` removes the device entry from the registry in order to prevent the entry from being found (and locked) by new driver clients. Once all driver client handlers are invoked, `svDeviceEvent` returns to the driver. Note that the real device shutdown has to be deferred until the *handler* routine invocation.

Once the device entry is released by the last driver client, the device registry module invokes *handler* previously specified via `svDeviceAlloc` . A pointer to the `DevRegEntry` structure (previously specified via `svDeviceAlloc`) is passed back to the driver as the argument of *handler* . The handler routine is called in the DKI thread context.

Typically, `svDeviceEvent` is used by a device driver servicing a hot-pluggable device. `svDeviceEvent` is invoked by the device driver when the driver is notified (by its parent driver) that the device has been removed from the bus.

Note - In the case of hot-plug removal, the device driver must still be operational until the device entry is released (ignoring all requests to the driver, for example). The driver is allowed to call `svDeviceEvent` on an invalid (unregistered) entry. In this type of case, *handler* will be invoked only when a registration attempt is made, that is, `svDeviceRegister` will be invoked. In addition, the registration will fail and the entry will remain invalid.

`svDeviceFree` releases a given device registry entry previously allocated by `svDeviceAlloc` . The device entry being released is specified by the *dev_id* argument. The entry being released must be invalid (unregistered). Thus, if the entry was registered by `svDeviceRegister` , the driver is allowed to release it if either the entry is successfully unregistered by `svDeviceUnregister` or *handler* (previously specified by `svDeviceAlloc`) is invoked.

`svDeviceLookup` searches the device entry in the registry matching the specified device class and logical unit.

dev_class	Specifies the device class.
dev_version	Specifies the minimum device driver interface version required.
dev_unit	Specifies the logical device unit in the class.
cli_handler	Specifies the event handler which is called when a device event is signalled.
cli_cookie	Specifies the first argument of *cli_handler* .
cli_id	Is an output argument identifying the client token on the matching device entry. This *cli_id* is passed back as an argument to `svDeviceEntry` to get the associated device entry, or to `svDeviceRelease` to release the device driver instance.

`svDeviceLookup` returns the following results:

K_OK	The device entry matching the *dev_class* and *dev_unit* pair was found in the registry.
K_EBUSY	The device entry matching the *dev_class* and *dev_unit* pair was not found in the registry. Or, the device is being used by another client and device sharing is not allowed.
K_EUNKNOWN	There is no device entry in the registry matching *dev_class* .
K_UNDEF	There is no device entry in the registry matching *dev_unit* in the *dev_class* . Or, there is a device entry in the *dev_class* with a logical unit greater than *dev_unit* . In other words, *dev_unit* falls into a hole in the *dev_class* unit ranges. The device entry matching the *dev_class* and *dev_unit* pair found in the registry is implementing an older version of the interface than the one required.
K_ETOOMUCH	There is no device entry in the registry matching *dev_unit* in the *dev_class* . Moreover *dev_unit* is greater than all existing logical units in the *dev_class* .

In case of success, the corresponding device entry is locked in the registry until a subsequent `svDeviceRelease` . Note that the device registry lock may or may not be exclusive, depending on the value of the *shared* argument specified in `svDeviceAlloc` . In other words, the device may be shared between multiple driver clients, if *shared* was `TRUE` at registration time. The device driver instance must not disappear while its device entry is locked in the registry.

In the case of a catastrophic device shutdown (for example, hot-plug removal) the device entry lock owners are notified (through the *cli_handler* routine invocation) that the device entry must be released as soon as possible.

The `svDeviceEntry` routine returns the device entry associated with a given client identifier. The *cli_id* argument specifies the client identifier previously returned by `svDeviceLookup` .

The `svDeviceRelease` routine releases the lock on the given device entry. The device entry is specified by the *cli_id* argument. Obviously, a device driver client should no longer access the device driver instance and the device node once the device entry has been released.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
svDeviceAlloc	+	+	-	+
svDeviceFree	+	+	-	+
svDeviceRegister	+	+	-	+
svDeviceUnregister	+	+	-	+
svDeviceEvent	+	+	+	-
svDeviceLookup	+	+	-	+
svDeviceRelease	+	+	-	+
svDeviceEntry	+	+	-	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`dtreeNodeRoot`(9DKI) , `svDkiThreadCall`(9DKI)

NAME

svDeviceRegister, svDeviceAlloc, svDeviceFree, svDeviceUnregister, svDeviceEvent, svDeviceLookup, svDeviceEntry, svDeviceRelease – device registry operations

SYNOPSIS

```
#include <dki/dki.h>
void svDeviceRegister(DevRegId dev_id);

DevRegId svDeviceAlloc(DevRegEntry * entry, unsigned int version, Bool shared,
DevRelHandler handler);

KnError svDeviceUnregister(DevRegId dev_id);

void svDeviceEvent(DevRegId dev_id, DevEvent event, void* arg);

void svDeviceFree(DevRegId dev_id);

KnError svDeviceLookup(char * dev_class, unsigned int dev_version, unsigned int
dev_unit, DevEventHandler cli_handler, void * cli_cookie, DevClientId * cli_id);

DevRegEntry * svDeviceEntry(DevClientId cli_id);

void svDeviceRelease(DevClientId cli_id);
typedef struct {
        char* dev_class;
        void* dev_ops;
        void* dev_id;
        DevNode dev_node;
    } DevRegEntry;
```

FEATURES

DKI

DESCRIPTION

Provides access to microkernel services implementing the `device registry`.

EXTENDED DESCRIPTION

The `device registry` microkernel module implements a data base of driver instances servicing devices currently supported by the system. The `device registry` data base is populated by drivers which perform self-registration (using `svDeviceRegister`) at device initialization time. The `device registry` data base is accessed by driver clients in order to obtain a pointer to the driver instance servicing a given (logical) device. The `device registry` API is described below in detail.

Note that only the `svDeviceLookup`, `svDeviceRelease` and `svDeviceEntry` microkernel calls should be used by driver clients. The rest of the API is dedicated to device drivers.

`svDeviceAlloc` allocates a device registry entry for a given device driver instance. Note that the entry is allocated in an invalid state. This means that the entry is not included in the registry (it is not visible to clients via `svDeviceLookup`).

The entry becomes valid (visible to clients) when a subsequent `svDeviceRegister` is invoked. On the other hand, the driver is allowed to call `svDeviceEvent` on this type of entry. Shut down events signaled on an invalid entry are memorized by the device registry and they are processed when the entry becomes valid (that is, when `svDeviceRegister` is called).

The *entry* argument points to the `DevRegEntry` structure which designates the device driver instance. The DevRegEntry structure is described below:

dev_class	Points to a string specifying the device class name.
dev_ops	Points to a structure of driver service routines.
dev_id	Points to a handle which is usually passed back to the driver (as first argument) each time a driver service routine is invoked.
dev_node	Specifies the device node in the tree which is serviced by the driver. Note that a driver client must switch to the DKI thread context when accessing the device tree structure (see section “Device Node Properties”).

The *version* argument specifies the driver interface version implemented by the device driver instance.

The *dev_class* field and *version* argument specify APIs implemented by the driver. This type of API is provided to driver clients as a structure of indirect functions implementing the API service routines. For instance, the *"uart"* device API is specified by the `UartDevOps` structure, the *"timer"* device API is specified by the `TimerDevOps` structure. A pointer to this type of structure is obtained by a device driver client from the *dev_ops* field of the device registry entry.

The *shared* argument specifies whether the device driver instance may be shared between multiple clients. In other words, it indicates whether the registered instance may be looked up multiple times.

The *handler* argument specifies the driver handler which is invoked by the device registry module as an acknowledgement to a shut-down event (see `svDeviceEvent`). *handler* is called when the last reference to the device registry

entry goes away and a shut-down event has been previously signalled on the entry. *entry* is passed back to the driver as the argument of *handler* .

In case of success, a non zero *DevRegId* is returned, otherwise `NULL` is returned.

Device registry assigns a logical unit number to the physical device. The logical unit number is unique within *dev_class* . The device registry handles a logical unit counter per class. All counters are initialized to zero. Each time a new device entry's is allocated, the current counter value is assigned to the entry logical unit number and the counter is incremented. Thus, the logical device order within a class corresponds to the allocation order.

The logical unit number is mainly used by device driver clients to iterate through a given device class looking for a certain device. Using a valid logical unit number, a client is able to access the corresponding device registry entry, in particular, the *dev_node* field which points to the device node. The physical device corresponding to a given logical unit number may then be detected by the device node properties and the node position within the device tree.

Note - The device driver may allocate (and then register) multiple entries in order to declare different classes (APIs) for the same physical device. For instance, a driver may implement orthogonal sets of operations like power management which are additional to the main device functionality. Another example is a device driver servicing a multi-function device. This type of driver would register multiple entries: one entry per device function. From the client's point of view, this type of device would be visible as multiple (mono-function) independent devices.

`svDeviceRegister` adds a given device registry entry to the registry. The entry must be previously allocated by `svDeviceAlloc` . The entry becomes valid (and therefore visible for clients) only if there is no shut-down event signalled on the entry. Otherwise, the entry remains invalid and the device registry module invokes a handler previously specified via `svDeviceAlloc` . A pointer to the `DevRegEntry` structure (previously specified via `svDeviceAlloc`) is passed back to the driver as the argument of *handler* . The *handler* routine is called in the DKI thread context.

Note - When *handler* is called the entry is no longer registered. Thus, it is useless to invoke `svDeviceUnregister` once *handler* has been called. On the other hand, in order to free memory resources, the driver should release the device registry entry invoking `svDeviceFree` once the entry is no longer used by the driver. Typically, the entry is no longer used by the driver when the connection to the parent bus / nexus driver is closed, and the parent driver is unable to invoke a call-back handler signalling a bus / nexus event.

`svDeviceUnregister` removes the device entry from the device registry (if the entry is valid). The device entry being removed is specified by the *dev_id* argument. `svDeviceUnregister` returns the following results when a valid entry is specified:

K_OK	The device entry has been successfully removed from the registry.
K_EBUSY	The device entry has not been removed from the registry because it is locked by a driver client. In other words, the `svDeviceLookup/Release` pairs are not balanced.

`svDeviceUnregister` returns `K_EBUSY` when an invalid entry is specified. The driver must remove all related device entries from the registry prior to being unloaded from the system. Typically, `svDeviceUnregister` is issued by the device driver (for each device driver instance) when the driver code is requested to be unloaded via `drv_unload` . The driver unload fails when the driver code is still being used by the system.

`svDeviceEvent` notifies the device registry module that a shutdown event has occurred (for example, that a hot-pluggable device has been removed). The shutdown event basically means that the driver instance should no longer be used by driver clients and the device entry must be removed from the registry (that is, the driver instance will disappear). The shutdown event is immediately propagated to all driver clients in order to stop the device usage and to release the device entry as soon as possible. The device is specified by the *dev_id* argument.

`svDeviceEvent` propagates the shutdown event invoking the *event_handler* routines specified by the driver clients in `svDeviceLookup` . The *event* and *arg* arguments are opaque for the device registry. They are passed (as arguments) to the client event handlers. *event* specifies the reason for the shut-down as follows:

DEV_EVENT_SHUTDOWN	Normal device shut down
DEV_EVENT_REMOVAL	Hot-plug (surprise) device removal

arg is *event* specific.

All driver clients are requested to release the device entry (as soon as possible) invoking the `svDeviceRelease` routine. Note that, prior to the shutdown event propagation, `svDeviceEvent` removes the device entry from the registry in order to prevent the entry from being found (and locked) by new driver clients. Once all driver client handlers are invoked, `svDeviceEvent` returns to the driver. Note that the real device shutdown has to be deferred until the *handler* routine invocation.

Once the device entry is released by the last driver client, the device registry module invokes *handler* previously specified via `svDeviceAlloc` . A pointer to the `DevRegEntry` structure (previously specified via `svDeviceAlloc`) is passed back to the driver as the argument of *handler* . The handler routine is called in the DKI thread context.

Typically, `svDeviceEvent` is used by a device driver servicing a hot-pluggable device. `svDeviceEvent` is invoked by the device driver when the driver is notified (by its parent driver) that the device has been removed from the bus.

Note - In the case of hot-plug removal, the device driver must still be operational until the device entry is released (ignoring all requests to the driver, for example). The driver is allowed to call `svDeviceEvent` on an invalid (unregistered) entry. In this type of case, *handler* will be invoked only when a registration attempt is made, that is, `svDeviceRegister` will be invoked. In addition, the registration will fail and the entry will remain invalid.

`svDeviceFree` releases a given device registry entry previously allocated by `svDeviceAlloc` . The device entry being released is specified by the *dev_id* argument. The entry being released must be invalid (unregistered). Thus, if the entry was registered by `svDeviceRegister` , the driver is allowed to release it if either the entry is successfully unregistered by `svDeviceUnregister` or *handler* (previously specified by `svDeviceAlloc`) is invoked.

`svDeviceLookup` searches the device entry in the registry matching the specified device class and logical unit.

dev_class	Specifies the device class.
dev_version	Specifies the minimum device driver interface version required.
dev_unit	Specifies the logical device unit in the class.
cli_handler	Specifies the event handler which is called when a device event is signalled.
cli_cookie	Specifies the first argument of *cli_handler* .
cli_id	Is an output argument identifying the client token on the matching device entry. This *cli_id* is passed back as an argument to `svDeviceEntry` to get the associated device entry, or to `svDeviceRelease` to release the device driver instance.

`svDeviceLookup` returns the following results:

K_OK	The device entry matching the *dev_class* and *dev_unit* pair was found in the registry.
K_EBUSY	The device entry matching the *dev_class* and *dev_unit* pair was not found in the registry. Or, the device is being used by another client and device sharing is not allowed.
K_EUNKNOWN	There is no device entry in the registry matching *dev_class* .
K_UNDEF	There is no device entry in the registry matching *dev_unit* in the *dev_class* . Or, there is a device entry in the *dev_class* with a logical unit greater than *dev_unit* . In other words, *dev_unit* falls into a hole in the *dev_class* unit ranges. The device entry matching the *dev_class* and *dev_unit* pair found in the registry is implementing an older version of the interface than the one required.
K_ETOOMUCH	There is no device entry in the registry matching *dev_unit* in the *dev_class* . Moreover *dev_unit* is greater than all existing logical units in the *dev_class* .

In case of success, the corresponding device entry is locked in the registry until a subsequent `svDeviceRelease` . Note that the device registry lock may or may not be exclusive, depending on the value of the *shared* argument specified in `svDeviceAlloc` . In other words, the device may be shared between multiple driver clients, if *shared* was `TRUE` at registration time. The device driver instance must not disappear while its device entry is locked in the registry.

In the case of a catastrophic device shutdown (for example, hot-plug removal) the device entry lock owners are notified (through the *cli_handler* routine invocation) that the device entry must be released as soon as possible.

The `svDeviceEntry` routine returns the device entry associated with a given client identifier. The *cli_id* argument specifies the client identifier previously returned by `svDeviceLookup` .

The `svDeviceRelease` routine releases the lock on the given device entry. The device entry is specified by the *cli_id* argument. Obviously, a device driver client should no longer access the device driver instance and the device node once the device entry has been released.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
svDeviceAlloc	+	+	-	+
svDeviceFree	+	+	-	+
svDeviceRegister	+	+	-	+
svDeviceUnregister	+	+	-	+
svDeviceEvent	+	+	+	-
svDeviceLookup	+	+	-	+
svDeviceRelease	+	+	-	+
svDeviceEntry	+	+	-	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`dtreeNodeRoot`(9DKI) , `svDkiThreadCall`(9DKI)

NAME

svDeviceRegister, svDeviceAlloc, svDeviceFree, svDeviceUnregister, svDeviceEvent, svDeviceLookup, svDeviceEntry, svDeviceRelease – device registry operations

SYNOPSIS

```
#include <dki/dki.h>
void svDeviceRegister(DevRegId dev_id);

DevRegId svDeviceAlloc(DevRegEntry * entry, unsigned int version, Bool shared,
DevRelHandler handler);

KnError svDeviceUnregister(DevRegId dev_id);

void svDeviceEvent(DevRegId dev_id, DevEvent event, void* arg);

void svDeviceFree(DevRegId dev_id);

KnError svDeviceLookup(char * dev_class, unsigned int dev_version, unsigned int
dev_unit, DevEventHandler cli_handler, void * cli_cookie, DevClientId * cli_id);

DevRegEntry * svDeviceEntry(DevClientId cli_id);

void svDeviceRelease(DevClientId cli_id);
typedef struct {
        char* dev_class;
        void* dev_ops;
        void* dev_id;
        DevNode dev_node;
    } DevRegEntry;
```

FEATURES

DKI

DESCRIPTION

Provides access to microkernel services implementing the `device registry`.

EXTENDED DESCRIPTION

The `device registry` microkernel module implements a data base of driver instances servicing devices currently supported by the system. The `device registry` data base is populated by drivers which perform self-registration (using `svDeviceRegister`) at device initialization time. The `device registry` data base is accessed by driver clients in order to obtain a pointer to the driver instance servicing a given (logical) device. The `device registry` API is described below in detail.

Note that only the `svDeviceLookup`, `svDeviceRelease` and `svDeviceEntry` microkernel calls should be used by driver clients. The rest of the API is dedicated to device drivers.

`svDeviceAlloc` allocates a device registry entry for a given device driver instance. Note that the entry is allocated in an invalid state. This means that the entry is not included in the registry (it is not visible to clients via `svDeviceLookup`).

The entry becomes valid (visible to clients) when a subsequent `svDeviceRegister` is invoked. On the other hand, the driver is allowed to call `svDeviceEvent` on this type of entry. Shut down events signaled on an invalid entry are memorized by the device registry and they are processed when the entry becomes valid (that is, when `svDeviceRegister` is called).

The *entry* argument points to the `DevRegEntry` structure which designates the device driver instance. The DevRegEntry structure is described below:

dev_class	Points to a string specifying the device class name.
dev_ops	Points to a structure of driver service routines.
dev_id	Points to a handle which is usually passed back to the driver (as first argument) each time a driver service routine is invoked.
dev_node	Specifies the device node in the tree which is serviced by the driver. Note that a driver client must switch to the DKI thread context when accessing the device tree structure (see section “Device Node Properties”).

The *version* argument specifies the driver interface version implemented by the device driver instance.

The *dev_class* field and *version* argument specify APIs implemented by the driver. This type of API is provided to driver clients as a structure of indirect functions implementing the API service routines. For instance, the *"uart"* device API is specified by the `UartDevOps` structure, the *"timer"* device API is specified by the `TimerDevOps` structure. A pointer to this type of structure is obtained by a device driver client from the *dev_ops* field of the device registry entry.

The *shared* argument specifies whether the device driver instance may be shared between multiple clients. In other words, it indicates whether the registered instance may be looked up multiple times.

The *handler* argument specifies the driver handler which is invoked by the device registry module as an acknowledgement to a shut-down event (see `svDeviceEvent`). *handler* is called when the last reference to the device registry

entry goes away and a shut-down event has been previously signalled on the entry. *entry* is passed back to the driver as the argument of *handler* .

In case of success, a non zero *DevRegId* is returned, otherwise `NULL` is returned.

Device registry assigns a logical unit number to the physical device. The logical unit number is unique within *dev_class* . The device registry handles a logical unit counter per class. All counters are initialized to zero. Each time a new device entry's is allocated, the current counter value is assigned to the entry logical unit number and the counter is incremented. Thus, the logical device order within a class corresponds to the allocation order.

The logical unit number is mainly used by device driver clients to iterate through a given device class looking for a certain device. Using a valid logical unit number, a client is able to access the corresponding device registry entry, in particular, the *dev_node* field which points to the device node. The physical device corresponding to a given logical unit number may then be detected by the device node properties and the node position within the device tree.

Note - The device driver may allocate (and then register) multiple entries in order to declare different classes (APIs) for the same physical device. For instance, a driver may implement orthogonal sets of operations like power management which are additional to the main device functionality. Another example is a device driver servicing a multi-function device. This type of driver would register multiple entries: one entry per device function. From the client's point of view, this type of device would be visible as multiple (mono-function) independent devices.

`svDeviceRegister` adds a given device registry entry to the registry. The entry must be previously allocated by `svDeviceAlloc` . The entry becomes valid (and therefore visible for clients) only if there is no shut-down event signalled on the entry. Otherwise, the entry remains invalid and the device registry module invokes a handler previously specified via `svDeviceAlloc` . A pointer to the `DevRegEntry` structure (previously specified via `svDeviceAlloc`) is passed back to the driver as the argument of *handler* . The *handler* routine is called in the DKI thread context.

Note - When *handler* is called the entry is no longer registered. Thus, it is useless to invoke `svDeviceUnregister` once *handler* has been called. On the other hand, in order to free memory resources, the driver should release the device registry entry invoking `svDeviceFree` once the entry is no longer used by the driver. Typically, the entry is no longer used by the driver when the connection to the parent bus / nexus driver is closed, and the parent driver is unable to invoke a call-back handler signalling a bus / nexus event.

`svDeviceUnregister` removes the device entry from the device registry (if the entry is valid). The device entry being removed is specified by the *dev_id* argument. `svDeviceUnregister` returns the following results when a valid entry is specified:

K_OK	The device entry has been successfully removed from the registry.
K_EBUSY	The device entry has not been removed from the registry because it is locked by a driver client. In other words, the `svDeviceLookup/Release` pairs are not balanced.

`svDeviceUnregister` returns `K_EBUSY` when an invalid entry is specified. The driver must remove all related device entries from the registry prior to being unloaded from the system. Typically, `svDeviceUnregister` is issued by the device driver (for each device driver instance) when the driver code is requested to be unloaded via `drv_unload` . The driver unload fails when the driver code is still being used by the system.

`svDeviceEvent` notifies the device registry module that a shutdown event has occurred (for example, that a hot-pluggable device has been removed). The shutdown event basically means that the driver instance should no longer be used by driver clients and the device entry must be removed from the registry (that is, the driver instance will disappear). The shutdown event is immediately propagated to all driver clients in order to stop the device usage and to release the device entry as soon as possible. The device is specified by the *dev_id* argument.

`svDeviceEvent` propagates the shutdown event invoking the *event_handler* routines specified by the driver clients in `svDeviceLookup` . The *event* and *arg* arguments are opaque for the device registry. They are passed (as arguments) to the client event handlers. *event* specifies the reason for the shut-down as follows:

DEV_EVENT_SHUTDOWN	Normal device shut down
DEV_EVENT_REMOVAL	Hot-plug (surprise) device removal

arg is *event* specific.

All driver clients are requested to release the device entry (as soon as possible) invoking the `svDeviceRelease` routine. Note that, prior to the shutdown event propagation, `svDeviceEvent` removes the device entry from the registry in order to prevent the entry from being found (and locked) by new driver clients. Once all driver client handlers are invoked, `svDeviceEvent` returns to the driver. Note that the real device shutdown has to be deferred until the *handler* routine invocation.

Once the device entry is released by the last driver client, the device registry module invokes *handler* previously specified via `svDeviceAlloc` . A pointer to the `DevRegEntry` structure (previously specified via `svDeviceAlloc`) is passed back to the driver as the argument of *handler* . The handler routine is called in the DKI thread context.

Typically, `svDeviceEvent` is used by a device driver servicing a hot-pluggable device. `svDeviceEvent` is invoked by the device driver when the driver is notified (by its parent driver) that the device has been removed from the bus.

Note - In the case of hot-plug removal, the device driver must still be operational until the device entry is released (ignoring all requests to the driver, for example). The driver is allowed to call `svDeviceEvent` on an invalid (unregistered) entry. In this type of case, *handler* will be invoked only when a registration attempt is made, that is, `svDeviceRegister` will be invoked. In addition, the registration will fail and the entry will remain invalid.

`svDeviceFree` releases a given device registry entry previously allocated by `svDeviceAlloc` . The device entry being released is specified by the *dev_id* argument. The entry being released must be invalid (unregistered). Thus, if the entry was registered by `svDeviceRegister` , the driver is allowed to release it if either the entry is successfully unregistered by `svDeviceUnregister` or *handler* (previously specified by `svDeviceAlloc`) is invoked.

`svDeviceLookup` searches the device entry in the registry matching the specified device class and logical unit.

dev_class	Specifies the device class.
dev_version	Specifies the minimum device driver interface version required.
dev_unit	Specifies the logical device unit in the class.
cli_handler	Specifies the event handler which is called when a device event is signalled.
cli_cookie	Specifies the first argument of *cli_handler* .
cli_id	Is an output argument identifying the client token on the matching device entry. This *cli_id* is passed back as an argument to `svDeviceEntry` to get the associated device entry, or to `svDeviceRelease` to release the device driver instance.

`svDeviceLookup` returns the following results:

K_OK	The device entry matching the *dev_class* and *dev_unit* pair was found in the registry.
K_EBUSY	The device entry matching the *dev_class* and *dev_unit* pair was not found in the registry. Or, the device is being used by another client and device sharing is not allowed.
K_EUNKNOWN	There is no device entry in the registry matching *dev_class* .
K_UNDEF	There is no device entry in the registry matching *dev_unit* in the *dev_class* . Or, there is a device entry in the *dev_class* with a logical unit greater than *dev_unit* . In other words, *dev_unit* falls into a hole in the *dev_class* unit ranges. The device entry matching the *dev_class* and *dev_unit* pair found in the registry is implementing an older version of the interface than the one required.
K_ETOOMUCH	There is no device entry in the registry matching *dev_unit* in the *dev_class* . Moreover *dev_unit* is greater than all existing logical units in the *dev_class* .

In case of success, the corresponding device entry is locked in the registry until a subsequent `svDeviceRelease` . Note that the device registry lock may or may not be exclusive, depending on the value of the *shared* argument specified in `svDeviceAlloc` . In other words, the device may be shared between multiple driver clients, if *shared* was `TRUE` at registration time. The device driver instance must not disappear while its device entry is locked in the registry.

In the case of a catastrophic device shutdown (for example, hot-plug removal) the device entry lock owners are notified (through the *cli_handler* routine invocation) that the device entry must be released as soon as possible.

The `svDeviceEntry` routine returns the device entry associated with a given client identifier. The *cli_id* argument specifies the client identifier previously returned by `svDeviceLookup` .

The `svDeviceRelease` routine releases the lock on the given device entry. The device entry is specified by the *cli_id* argument. Obviously, a device driver client should no longer access the device driver instance and the device node once the device entry has been released.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
svDeviceAlloc	+	+	-	+
svDeviceFree	+	+	-	+
svDeviceRegister	+	+	-	+
svDeviceUnregister	+	+	-	+
svDeviceEvent	+	+	+	-
svDeviceLookup	+	+	-	+
svDeviceRelease	+	+	-	+
svDeviceEntry	+	+	-	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`dtreeNodeRoot`(9DKI) , `svDkiThreadCall`(9DKI)

NAME

svDeviceRegister, svDeviceAlloc, svDeviceFree, svDeviceUnregister, svDeviceEvent, svDeviceLookup, svDeviceEntry, svDeviceRelease – device registry operations

SYNOPSIS

#include <dki/dki.h>
void **svDeviceRegister**(DevRegId *dev_id*);

DevRegId **svDeviceAlloc**(DevRegEntry * *entry*, unsigned int *version*, Bool *shared*, DevRelHandler *handler*);

KnError **svDeviceUnregister**(DevRegId *dev_id*);

void **svDeviceEvent**(DevRegId *dev_id*, DevEvent *event*, void* *arg*);

void **svDeviceFree**(DevRegId *dev_id*);

KnError **svDeviceLookup**(char * *dev_class*, unsigned int *dev_version*, unsigned int *dev_unit*, DevEventHandler *cli_handler*, void * *cli_cookie*, DevClientId * *cli_id*);

DevRegEntry * **svDeviceEntry**(DevClientId *cli_id*);

void **svDeviceRelease**(DevClientId *cli_id*);

```
typedef struct {
        char* dev_class;
        void* dev_ops;
        void* dev_id;
        DevNode dev_node;
    } DevRegEntry;
```

FEATURES

DKI

DESCRIPTION

Provides access to microkernel services implementing the `device registry`.

EXTENDED DESCRIPTION

The `device registry` microkernel module implements a data base of driver instances servicing devices currently supported by the system. The `device registry` data base is populated by drivers which perform self-registration (using `svDeviceRegister`) at device initialization time. The `device registry` data base is accessed by driver clients in order to obtain a pointer to the driver instance servicing a given (logical) device. The `device registry` API is described below in detail.

Note that only the `svDeviceLookup`, `svDeviceRelease` and `svDeviceEntry` microkernel calls should be used by driver clients. The rest of the API is dedicated to device drivers.

`svDeviceAlloc` allocates a device registry entry for a given device driver instance. Note that the entry is allocated in an invalid state. This means that the entry is not included in the registry (it is not visible to clients via `svDeviceLookup`).

The entry becomes valid (visible to clients) when a subsequent `svDeviceRegister` is invoked. On the other hand, the driver is allowed to call `svDeviceEvent` on this type of entry. Shut down events signaled on an invalid entry are memorized by the device registry and they are processed when the entry becomes valid (that is, when `svDeviceRegister` is called).

The *entry* argument points to the `DevRegEntry` structure which designates the device driver instance. The DevRegEntry structure is described below:

dev_class	Points to a string specifying the device class name.
dev_ops	Points to a structure of driver service routines.
dev_id	Points to a handle which is usually passed back to the driver (as first argument) each time a driver service routine is invoked.
dev_node	Specifies the device node in the tree which is serviced by the driver. Note that a driver client must switch to the DKI thread context when accessing the device tree structure (see section “Device Node Properties”).

The *version* argument specifies the driver interface version implemented by the device driver instance.

The *dev_class* field and *version* argument specify APIs implemented by the driver. This type of API is provided to driver clients as a structure of indirect functions implementing the API service routines. For instance, the *"uart"* device API is specified by the `UartDevOps` structure, the *"timer"* device API is specified by the `TimerDevOps` structure. A pointer to this type of structure is obtained by a device driver client from the *dev_ops* field of the device registry entry.

The *shared* argument specifies whether the device driver instance may be shared between multiple clients. In other words, it indicates whether the registered instance may be looked up multiple times.

The *handler* argument specifies the driver handler which is invoked by the device registry module as an acknowledgement to a shut-down event (see `svDeviceEvent`). *handler* is called when the last reference to the device registry

entry goes away and a shut-down event has been previously signalled on the entry. *entry* is passed back to the driver as the argument of *handler* .

In case of success, a non zero *DevRegId* is returned, otherwise `NULL` is returned.

Device registry assigns a logical unit number to the physical device. The logical unit number is unique within *dev_class* . The device registry handles a logical unit counter per class. All counters are initialized to zero. Each time a new device entry's is allocated, the current counter value is assigned to the entry logical unit number and the counter is incremented. Thus, the logical device order within a class corresponds to the allocation order.

The logical unit number is mainly used by device driver clients to iterate through a given device class looking for a certain device. Using a valid logical unit number, a client is able to access the corresponding device registry entry, in particular, the *dev_node* field which points to the device node. The physical device corresponding to a given logical unit number may then be detected by the device node properties and the node position within the device tree.

Note - The device driver may allocate (and then register) multiple entries in order to declare different classes (APIs) for the same physical device. For instance, a driver may implement orthogonal sets of operations like power management which are additional to the main device functionality. Another example is a device driver servicing a multi-function device. This type of driver would register multiple entries: one entry per device function. From the client's point of view, this type of device would be visible as multiple (mono-function) independent devices.

`svDeviceRegister` adds a given device registry entry to the registry. The entry must be previously allocated by `svDeviceAlloc` . The entry becomes valid (and therefore visible for clients) only if there is no shut-down event signalled on the entry. Otherwise, the entry remains invalid and the device registry module invokes a handler previously specified via `svDeviceAlloc` . A pointer to the `DevRegEntry` structure (previously specified via `svDeviceAlloc`) is passed back to the driver as the argument of *handler* . The *handler* routine is called in the DKI thread context.

Note - When *handler* is called the entry is no longer registered. Thus, it is useless to invoke `svDeviceUnregister` once *handler* has been called. On the other hand, in order to free memory resources, the driver should release the device registry entry invoking `svDeviceFree` once the entry is no longer used by the driver. Typically, the entry is no longer used by the driver when the connection to the parent bus / nexus driver is closed, and the parent driver is unable to invoke a call-back handler signalling a bus / nexus event.

`svDeviceUnregister` removes the device entry from the device registry (if the entry is valid). The device entry being removed is specified by the *dev_id* argument. `svDeviceUnregister` returns the following results when a valid entry is specified:

K_OK	The device entry has been successfully removed from the registry.
K_EBUSY	The device entry has not been removed from the registry because it is locked by a driver client. In other words, the `svDeviceLookup/Release` pairs are not balanced.

`svDeviceUnregister` returns `K_EBUSY` when an invalid entry is specified. The driver must remove all related device entries from the registry prior to being unloaded from the system. Typically, `svDeviceUnregister` is issued by the device driver (for each device driver instance) when the driver code is requested to be unloaded via `drv_unload` . The driver unload fails when the driver code is still being used by the system.

`svDeviceEvent` notifies the device registry module that a shutdown event has occurred (for example, that a hot-pluggable device has been removed). The shutdown event basically means that the driver instance should no longer be used by driver clients and the device entry must be removed from the registry (that is, the driver instance will disappear). The shutdown event is immediately propagated to all driver clients in order to stop the device usage and to release the device entry as soon as possible. The device is specified by the *dev_id* argument.

`svDeviceEvent` propagates the shutdown event invoking the *event_handler* routines specified by the driver clients in `svDeviceLookup` . The *event* and *arg* arguments are opaque for the device registry. They are passed (as arguments) to the client event handlers. *event* specifies the reason for the shut-down as follows:

DEV_EVENT_SHUTDOWN	Normal device shut down
DEV_EVENT_REMOVAL	Hot-plug (surprise) device removal

arg is *event* specific.

All driver clients are requested to release the device entry (as soon as possible) invoking the `svDeviceRelease` routine. Note that, prior to the shutdown event propagation, `svDeviceEvent` removes the device entry from the registry in order to prevent the entry from being found (and locked) by new driver clients. Once all driver client handlers are invoked, `svDeviceEvent` returns to the driver. Note that the real device shutdown has to be deferred until the *handler* routine invocation.

Once the device entry is released by the last driver client, the device registry module invokes *handler* previously specified via `svDeviceAlloc` . A pointer to the `DevRegEntry` structure (previously specified via `svDeviceAlloc`) is passed back to the driver as the argument of *handler* . The handler routine is called in the DKI thread context.

Typically, `svDeviceEvent` is used by a device driver servicing a hot-pluggable device. `svDeviceEvent` is invoked by the device driver when the driver is notified (by its parent driver) that the device has been removed from the bus.

Note - In the case of hot-plug removal, the device driver must still be operational until the device entry is released (ignoring all requests to the driver, for example). The driver is allowed to call `svDeviceEvent` on an invalid (unregistered) entry. In this type of case, *handler* will be invoked only when a registration attempt is made, that is, `svDeviceRegister` will be invoked. In addition, the registration will fail and the entry will remain invalid.

`svDeviceFree` releases a given device registry entry previously allocated by `svDeviceAlloc` . The device entry being released is specified by the *dev_id* argument. The entry being released must be invalid (unregistered). Thus, if the entry was registered by `svDeviceRegister` , the driver is allowed to release it if either the entry is successfully unregistered by `svDeviceUnregister` or *handler* (previously specified by `svDeviceAlloc`) is invoked.

`svDeviceLookup` searches the device entry in the registry matching the specified device class and logical unit.

dev_class	Specifies the device class.
dev_version	Specifies the minimum device driver interface version required.
dev_unit	Specifies the logical device unit in the class.
cli_handler	Specifies the event handler which is called when a device event is signalled.
cli_cookie	Specifies the first argument of *cli_handler* .
cli_id	Is an output argument identifying the client token on the matching device entry. This *cli_id* is passed back as an argument to `svDeviceEntry` to get the associated device entry, or to `svDeviceRelease` to release the device driver instance.

`svDeviceLookup` returns the following results:

K_OK	The device entry matching the *dev_class* and *dev_unit* pair was found in the registry.
K_EBUSY	The device entry matching the *dev_class* and *dev_unit* pair was not found in the registry. Or, the device is being used by another client and device sharing is not allowed.
K_EUNKNOWN	There is no device entry in the registry matching *dev_class* .
K_UNDEF	There is no device entry in the registry matching *dev_unit* in the *dev_class* . Or, there is a device entry in the *dev_class* with a logical unit greater than *dev_unit* . In other words, *dev_unit* falls into a hole in the *dev_class* unit ranges. The device entry matching the *dev_class* and *dev_unit* pair found in the registry is implementing an older version of the interface than the one required.
K_ETOOMUCH	There is no device entry in the registry matching *dev_unit* in the *dev_class* . Moreover *dev_unit* is greater than all existing logical units in the *dev_class* .

In case of success, the corresponding device entry is locked in the registry until a subsequent `svDeviceRelease` . Note that the device registry lock may or may not be exclusive, depending on the value of the *shared* argument specified in `svDeviceAlloc` . In other words, the device may be shared between multiple driver clients, if *shared* was `TRUE` at registration time. The device driver instance must not disappear while its device entry is locked in the registry.

In the case of a catastrophic device shutdown (for example, hot-plug removal) the device entry lock owners are notified (through the *cli_handler* routine invocation) that the device entry must be released as soon as possible.

The `svDeviceEntry` routine returns the device entry associated with a given client identifier. The *cli_id* argument specifies the client identifier previously returned by `svDeviceLookup` .

The `svDeviceRelease` routine releases the lock on the given device entry. The device entry is specified by the *cli_id* argument. Obviously, a device driver client should no longer access the device driver instance and the device node once the device entry has been released.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
svDeviceAlloc	+	+	-	+
svDeviceFree	+	+	-	+
svDeviceRegister	+	+	-	+
svDeviceUnregister	+	+	-	+
svDeviceEvent	+	+	+	-
svDeviceLookup	+	+	-	+
svDeviceRelease	+	+	-	+
svDeviceEntry	+	+	-	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`dtreeNodeRoot`(9DKI) , `svDkiThreadCall`(9DKI)

NAME

svDeviceRegister, svDeviceAlloc, svDeviceFree, svDeviceUnregister, svDeviceEvent, svDeviceLookup, svDeviceEntry, svDeviceRelease – device registry operations

SYNOPSIS

```
#include <dki/dki.h>
void svDeviceRegister(DevRegId dev_id);

DevRegId svDeviceAlloc(DevRegEntry * entry, unsigned int version, Bool shared,
DevRelHandler handler);

KnError svDeviceUnregister(DevRegId dev_id);

void svDeviceEvent(DevRegId dev_id, DevEvent event, void* arg);

void svDeviceFree(DevRegId dev_id);

KnError svDeviceLookup(char * dev_class, unsigned int dev_version, unsigned int
dev_unit, DevEventHandler cli_handler, void * cli_cookie, DevClientId * cli_id);

DevRegEntry * svDeviceEntry(DevClientId cli_id);

void svDeviceRelease(DevClientId cli_id);
typedef struct {
        char* dev_class;
        void* dev_ops;
        void* dev_id;
        DevNode dev_node;
    } DevRegEntry;
```

FEATURES

DKI

DESCRIPTION

Provides access to microkernel services implementing the `device registry`.

EXTENDED DESCRIPTION

The `device registry` microkernel module implements a data base of driver instances servicing devices currently supported by the system. The `device registry` data base is populated by drivers which perform self-registration (using `svDeviceRegister`) at device initialization time. The `device registry` data base is accessed by driver clients in order to obtain a pointer to the driver instance servicing a given (logical) device. The `device registry` API is described below in detail.

Note that only the `svDeviceLookup`, `svDeviceRelease` and `svDeviceEntry` microkernel calls should be used by driver clients. The rest of the API is dedicated to device drivers.

`svDeviceAlloc` allocates a device registry entry for a given device driver instance. Note that the entry is allocated in an invalid state. This means that the entry is not included in the registry (it is not visible to clients via `svDeviceLookup`).

The entry becomes valid (visible to clients) when a subsequent `svDeviceRegister` is invoked. On the other hand, the driver is allowed to call `svDeviceEvent` on this type of entry. Shut down events signaled on an invalid entry are memorized by the device registry and they are processed when the entry becomes valid (that is, when `svDeviceRegister` is called).

The *entry* argument points to the `DevRegEntry` structure which designates the device driver instance. The DevRegEntry structure is described below:

dev_class	Points to a string specifying the device class name.
dev_ops	Points to a structure of driver service routines.
dev_id	Points to a handle which is usually passed back to the driver (as first argument) each time a driver service routine is invoked.
dev_node	Specifies the device node in the tree which is serviced by the driver. Note that a driver client must switch to the DKI thread context when accessing the device tree structure (see section “Device Node Properties”).

The *version* argument specifies the driver interface version implemented by the device driver instance.

The *dev_class* field and *version* argument specify APIs implemented by the driver. This type of API is provided to driver clients as a structure of indirect functions implementing the API service routines. For instance, the *"uart"* device API is specified by the `UartDevOps` structure, the *"timer"* device API is specified by the `TimerDevOps` structure. A pointer to this type of structure is obtained by a device driver client from the *dev_ops* field of the device registry entry.

The *shared* argument specifies whether the device driver instance may be shared between multiple clients. In other words, it indicates whether the registered instance may be looked up multiple times.

The *handler* argument specifies the driver handler which is invoked by the device registry module as an acknowledgement to a shut-down event (see `svDeviceEvent`). *handler* is called when the last reference to the device registry

entry goes away and a shut-down event has been previously signalled on the entry. *entry* is passed back to the driver as the argument of *handler* .

In case of success, a non zero *DevRegId* is returned, otherwise `NULL` is returned.

Device registry assigns a logical unit number to the physical device. The logical unit number is unique within *dev_class* . The device registry handles a logical unit counter per class. All counters are initialized to zero. Each time a new device entry's is allocated, the current counter value is assigned to the entry logical unit number and the counter is incremented. Thus, the logical device order within a class corresponds to the allocation order.

The logical unit number is mainly used by device driver clients to iterate through a given device class looking for a certain device. Using a valid logical unit number, a client is able to access the corresponding device registry entry, in particular, the *dev_node* field which points to the device node. The physical device corresponding to a given logical unit number may then be detected by the device node properties and the node position within the device tree.

Note - The device driver may allocate (and then register) multiple entries in order to declare different classes (APIs) for the same physical device. For instance, a driver may implement orthogonal sets of operations like power management which are additional to the main device functionality. Another example is a device driver servicing a multi-function device. This type of driver would register multiple entries: one entry per device function. From the client's point of view, this type of device would be visible as multiple (mono-function) independent devices.

`svDeviceRegister` adds a given device registry entry to the registry. The entry must be previously allocated by `svDeviceAlloc` . The entry becomes valid (and therefore visible for clients) only if there is no shut-down event signalled on the entry. Otherwise, the entry remains invalid and the device registry module invokes a handler previously specified via `svDeviceAlloc` . A pointer to the `DevRegEntry` structure (previously specified via `svDeviceAlloc`) is passed back to the driver as the argument of *handler* . The *handler* routine is called in the DKI thread context.

Note - When *handler* is called the entry is no longer registered. Thus, it is useless to invoke `svDeviceUnregister` once *handler* has been called. On the other hand, in order to free memory resources, the driver should release the device registry entry invoking `svDeviceFree` once the entry is no longer used by the driver. Typically, the entry is no longer used by the driver when the connection to the parent bus / nexus driver is closed, and the parent driver is unable to invoke a call-back handler signalling a bus / nexus event.

`svDeviceUnregister` removes the device entry from the device registry (if the entry is valid). The device entry being removed is specified by the *dev_id* argument. `svDeviceUnregister` returns the following results when a valid entry is specified:

K_OK	The device entry has been successfully removed from the registry.
K_EBUSY	The device entry has not been removed from the registry because it is locked by a driver client. In other words, the `svDeviceLookup/Release` pairs are not balanced.

`svDeviceUnregister` returns `K_EBUSY` when an invalid entry is specified. The driver must remove all related device entries from the registry prior to being unloaded from the system. Typically, `svDeviceUnregister` is issued by the device driver (for each device driver instance) when the driver code is requested to be unloaded via `drv_unload` . The driver unload fails when the driver code is still being used by the system.

`svDeviceEvent` notifies the device registry module that a shutdown event has occurred (for example, that a hot-pluggable device has been removed). The shutdown event basically means that the driver instance should no longer be used by driver clients and the device entry must be removed from the registry (that is, the driver instance will disappear). The shutdown event is immediately propagated to all driver clients in order to stop the device usage and to release the device entry as soon as possible. The device is specified by the *dev_id* argument.

`svDeviceEvent` propagates the shutdown event invoking the *event_handler* routines specified by the driver clients in `svDeviceLookup` . The *event* and *arg* arguments are opaque for the device registry. They are passed (as arguments) to the client event handlers. *event* specifies the reason for the shut-down as follows:

DEV_EVENT_SHUTDOWN	Normal device shut down
DEV_EVENT_REMOVAL	Hot-plug (surprise) device removal

arg is *event* specific.

All driver clients are requested to release the device entry (as soon as possible) invoking the `svDeviceRelease` routine. Note that, prior to the shutdown event propagation, `svDeviceEvent` removes the device entry from the registry in order to prevent the entry from being found (and locked) by new driver clients. Once all driver client handlers are invoked, `svDeviceEvent` returns to the driver. Note that the real device shutdown has to be deferred until the *handler* routine invocation.

Once the device entry is released by the last driver client, the device registry module invokes *handler* previously specified via `svDeviceAlloc` . A pointer to the `DevRegEntry` structure (previously specified via `svDeviceAlloc`) is passed back to the driver as the argument of *handler* . The handler routine is called in the DKI thread context.

Typically, `svDeviceEvent` is used by a device driver servicing a hot-pluggable device. `svDeviceEvent` is invoked by the device driver when the driver is notified (by its parent driver) that the device has been removed from the bus.

Note - In the case of hot-plug removal, the device driver must still be operational until the device entry is released (ignoring all requests to the driver, for example). The driver is allowed to call `svDeviceEvent` on an invalid (unregistered) entry. In this type of case, *handler* will be invoked only when a registration attempt is made, that is, `svDeviceRegister` will be invoked. In addition, the registration will fail and the entry will remain invalid.

`svDeviceFree` releases a given device registry entry previously allocated by `svDeviceAlloc` . The device entry being released is specified by the *dev_id* argument. The entry being released must be invalid (unregistered). Thus, if the entry was registered by `svDeviceRegister` , the driver is allowed to release it if either the entry is successfully unregistered by `svDeviceUnregister` or *handler* (previously specified by `svDeviceAlloc`) is invoked.

`svDeviceLookup` searches the device entry in the registry matching the specified device class and logical unit.

dev_class	Specifies the device class.
dev_version	Specifies the minimum device driver interface version required.
dev_unit	Specifies the logical device unit in the class.
cli_handler	Specifies the event handler which is called when a device event is signalled.
cli_cookie	Specifies the first argument of *cli_handler* .
cli_id	Is an output argument identifying the client token on the matching device entry. This *cli_id* is passed back as an argument to `svDeviceEntry` to get the associated device entry, or to `svDeviceRelease` to release the device driver instance.

`svDeviceLookup` returns the following results:

K_OK	The device entry matching the *dev_class* and *dev_unit* pair was found in the registry.
K_EBUSY	The device entry matching the *dev_class* and *dev_unit* pair was not found in the registry. Or, the device is being used by another client and device sharing is not allowed.
K_EUNKNOWN	There is no device entry in the registry matching *dev_class* .
K_UNDEF	There is no device entry in the registry matching *dev_unit* in the *dev_class* . Or, there is a device entry in the *dev_class* with a logical unit greater than *dev_unit* . In other words, *dev_unit* falls into a hole in the *dev_class* unit ranges. The device entry matching the *dev_class* and *dev_unit* pair found in the registry is implementing an older version of the interface than the one required.
K_ETOOMUCH	There is no device entry in the registry matching *dev_unit* in the *dev_class* . Moreover *dev_unit* is greater than all existing logical units in the *dev_class* .

In case of success, the corresponding device entry is locked in the registry until a subsequent `svDeviceRelease` . Note that the device registry lock may or may not be exclusive, depending on the value of the *shared* argument specified in `svDeviceAlloc` . In other words, the device may be shared between multiple driver clients, if *shared* was `TRUE` at registration time. The device driver instance must not disappear while its device entry is locked in the registry.

In the case of a catastrophic device shutdown (for example, hot-plug removal) the device entry lock owners are notified (through the *cli_handler* routine invocation) that the device entry must be released as soon as possible.

The `svDeviceEntry` routine returns the device entry associated with a given client identifier. The *cli_id* argument specifies the client identifier previously returned by `svDeviceLookup` .

The `svDeviceRelease` routine releases the lock on the given device entry. The device entry is specified by the *cli_id* argument. Obviously, a device driver client should no longer access the device driver instance and the device node once the device entry has been released.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
svDeviceAlloc	+	+	-	+
svDeviceFree	+	+	-	+
svDeviceRegister	+	+	-	+
svDeviceUnregister	+	+	-	+
svDeviceEvent	+	+	+	-
svDeviceLookup	+	+	-	+
svDeviceRelease	+	+	-	+
svDeviceEntry	+	+	-	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`dtreeNodeRoot`(9DKI) , `svDkiThreadCall`(9DKI)

NAME

svDeviceRegister, svDeviceAlloc, svDeviceFree, svDeviceUnregister, svDeviceEvent, svDeviceLookup, svDeviceEntry, svDeviceRelease – device registry operations

SYNOPSIS

```
#include <dki/dki.h>
void svDeviceRegister(DevRegId dev_id);

DevRegId svDeviceAlloc(DevRegEntry * entry, unsigned int version, Bool shared,
DevRelHandler handler);

KnError svDeviceUnregister(DevRegId dev_id);

void svDeviceEvent(DevRegId dev_id, DevEvent event, void* arg);

void svDeviceFree(DevRegId dev_id);

KnError svDeviceLookup(char * dev_class, unsigned int dev_version, unsigned int
dev_unit, DevEventHandler cli_handler, void * cli_cookie, DevClientId * cli_id);

DevRegEntry * svDeviceEntry(DevClientId cli_id);

void svDeviceRelease(DevClientId cli_id);
typedef struct {
        char* dev_class;
        void* dev_ops;
        void* dev_id;
        DevNode dev_node;
    } DevRegEntry;
```

FEATURES

DKI

DESCRIPTION

Provides access to microkernel services implementing the `device registry`.

EXTENDED DESCRIPTION

The `device registry` microkernel module implements a data base of driver instances servicing devices currently supported by the system. The `device registry` data base is populated by drivers which perform self-registration (using `svDeviceRegister`) at device initialization time. The `device registry` data base is accessed by driver clients in order to obtain a pointer to the driver instance servicing a given (logical) device. The `device registry` API is described below in detail.

Note that only the `svDeviceLookup`, `svDeviceRelease` and `svDeviceEntry` microkernel calls should be used by driver clients. The rest of the API is dedicated to device drivers.

`svDeviceAlloc` allocates a device registry entry for a given device driver instance. Note that the entry is allocated in an invalid state. This means that the entry is not included in the registry (it is not visible to clients via `svDeviceLookup`).

The entry becomes valid (visible to clients) when a subsequent `svDeviceRegister` is invoked. On the other hand, the driver is allowed to call `svDeviceEvent` on this type of entry. Shut down events signaled on an invalid entry are memorized by the device registry and they are processed when the entry becomes valid (that is, when `svDeviceRegister` is called).

The *entry* argument points to the `DevRegEntry` structure which designates the device driver instance. The DevRegEntry structure is described below:

dev_class	Points to a string specifying the device class name.
dev_ops	Points to a structure of driver service routines.
dev_id	Points to a handle which is usually passed back to the driver (as first argument) each time a driver service routine is invoked.
dev_node	Specifies the device node in the tree which is serviced by the driver. Note that a driver client must switch to the DKI thread context when accessing the device tree structure (see section “Device Node Properties”).

The *version* argument specifies the driver interface version implemented by the device driver instance.

The *dev_class* field and *version* argument specify APIs implemented by the driver. This type of API is provided to driver clients as a structure of indirect functions implementing the API service routines. For instance, the *"uart"* device API is specified by the `UartDevOps` structure, the *"timer"* device API is specified by the `TimerDevOps` structure. A pointer to this type of structure is obtained by a device driver client from the *dev_ops* field of the device registry entry.

The *shared* argument specifies whether the device driver instance may be shared between multiple clients. In other words, it indicates whether the registered instance may be looked up multiple times.

The *handler* argument specifies the driver handler which is invoked by the device registry module as an acknowledgement to a shut-down event (see `svDeviceEvent`). *handler* is called when the last reference to the device registry

entry goes away and a shut-down event has been previously signalled on the entry. *entry* is passed back to the driver as the argument of *handler* .

In case of success, a non zero *DevRegId* is returned, otherwise `NULL` is returned.

Device registry assigns a logical unit number to the physical device. The logical unit number is unique within *dev_class* . The device registry handles a logical unit counter per class. All counters are initialized to zero. Each time a new device entry's is allocated, the current counter value is assigned to the entry logical unit number and the counter is incremented. Thus, the logical device order within a class corresponds to the allocation order.

The logical unit number is mainly used by device driver clients to iterate through a given device class looking for a certain device. Using a valid logical unit number, a client is able to access the corresponding device registry entry, in particular, the *dev_node* field which points to the device node. The physical device corresponding to a given logical unit number may then be detected by the device node properties and the node position within the device tree.

Note - The device driver may allocate (and then register) multiple entries in order to declare different classes (APIs) for the same physical device. For instance, a driver may implement orthogonal sets of operations like power management which are additional to the main device functionality. Another example is a device driver servicing a multi-function device. This type of driver would register multiple entries: one entry per device function. From the client's point of view, this type of device would be visible as multiple (mono-function) independent devices.

`svDeviceRegister` adds a given device registry entry to the registry. The entry must be previously allocated by `svDeviceAlloc` . The entry becomes valid (and therefore visible for clients) only if there is no shut-down event signalled on the entry. Otherwise, the entry remains invalid and the device registry module invokes a handler previously specified via `svDeviceAlloc` . A pointer to the `DevRegEntry` structure (previously specified via `svDeviceAlloc`) is passed back to the driver as the argument of *handler* . The *handler* routine is called in the DKI thread context.

Note - When *handler* is called the entry is no longer registered. Thus, it is useless to invoke `svDeviceUnregister` once *handler* has been called. On the other hand, in order to free memory resources, the driver should release the device registry entry invoking `svDeviceFree` once the entry is no longer used by the driver. Typically, the entry is no longer used by the driver when the connection to the parent bus / nexus driver is closed, and the parent driver is unable to invoke a call-back handler signalling a bus / nexus event.

`svDeviceUnregister` removes the device entry from the device registry (if the entry is valid). The device entry being removed is specified by the *dev_id* argument. `svDeviceUnregister` returns the following results when a valid entry is specified:

K_OK	The device entry has been successfully removed from the registry.
K_EBUSY	The device entry has not been removed from the registry because it is locked by a driver client. In other words, the `svDeviceLookup/Release` pairs are not balanced.

`svDeviceUnregister` returns `K_EBUSY` when an invalid entry is specified. The driver must remove all related device entries from the registry prior to being unloaded from the system. Typically, `svDeviceUnregister` is issued by the device driver (for each device driver instance) when the driver code is requested to be unloaded via `drv_unload` . The driver unload fails when the driver code is still being used by the system.

`svDeviceEvent` notifies the device registry module that a shutdown event has occurred (for example, that a hot-pluggable device has been removed). The shutdown event basically means that the driver instance should no longer be used by driver clients and the device entry must be removed from the registry (that is, the driver instance will disappear). The shutdown event is immediately propagated to all driver clients in order to stop the device usage and to release the device entry as soon as possible. The device is specified by the *dev_id* argument.

`svDeviceEvent` propagates the shutdown event invoking the *event_handler* routines specified by the driver clients in `svDeviceLookup` . The *event* and *arg* arguments are opaque for the device registry. They are passed (as arguments) to the client event handlers. *event* specifies the reason for the shut-down as follows:

DEV_EVENT_SHUTDOWN	Normal device shut down
DEV_EVENT_REMOVAL	Hot-plug (surprise) device removal

arg is *event* specific.

All driver clients are requested to release the device entry (as soon as possible) invoking the `svDeviceRelease` routine. Note that, prior to the shutdown event propagation, `svDeviceEvent` removes the device entry from the registry in order to prevent the entry from being found (and locked) by new driver clients. Once all driver client handlers are invoked, `svDeviceEvent` returns to the driver. Note that the real device shutdown has to be deferred until the *handler* routine invocation.

Once the device entry is released by the last driver client, the device registry module invokes *handler* previously specified via `svDeviceAlloc` . A pointer to the `DevRegEntry` structure (previously specified via `svDeviceAlloc`) is passed back to the driver as the argument of *handler* . The handler routine is called in the DKI thread context.

Typically, `svDeviceEvent` is used by a device driver servicing a hot-pluggable device. `svDeviceEvent` is invoked by the device driver when the driver is notified (by its parent driver) that the device has been removed from the bus.

Note - In the case of hot-plug removal, the device driver must still be operational until the device entry is released (ignoring all requests to the driver, for example). The driver is allowed to call `svDeviceEvent` on an invalid (unregistered) entry. In this type of case, *handler* will be invoked only when a registration attempt is made, that is, `svDeviceRegister` will be invoked. In addition, the registration will fail and the entry will remain invalid.

`svDeviceFree` releases a given device registry entry previously allocated by `svDeviceAlloc` . The device entry being released is specified by the *dev_id* argument. The entry being released must be invalid (unregistered). Thus, if the entry was registered by `svDeviceRegister` , the driver is allowed to release it if either the entry is successfully unregistered by `svDeviceUnregister` or *handler* (previously specified by `svDeviceAlloc`) is invoked.

`svDeviceLookup` searches the device entry in the registry matching the specified device class and logical unit.

dev_class	Specifies the device class.
dev_version	Specifies the minimum device driver interface version required.
dev_unit	Specifies the logical device unit in the class.
cli_handler	Specifies the event handler which is called when a device event is signalled.
cli_cookie	Specifies the first argument of *cli_handler* .
cli_id	Is an output argument identifying the client token on the matching device entry. This *cli_id* is passed back as an argument to `svDeviceEntry` to get the associated device entry, or to `svDeviceRelease` to release the device driver instance.

`svDeviceLookup` returns the following results:

K_OK	The device entry matching the *dev_class* and *dev_unit* pair was found in the registry.
K_EBUSY	The device entry matching the *dev_class* and *dev_unit* pair was not found in the registry. Or, the device is being used by another client and device sharing is not allowed.
K_EUNKNOWN	There is no device entry in the registry matching *dev_class* .
K_UNDEF	There is no device entry in the registry matching *dev_unit* in the *dev_class* . Or, there is a device entry in the *dev_class* with a logical unit greater than *dev_unit* . In other words, *dev_unit* falls into a hole in the *dev_class* unit ranges. The device entry matching the *dev_class* and *dev_unit* pair found in the registry is implementing an older version of the interface than the one required.
K_ETOOMUCH	There is no device entry in the registry matching *dev_unit* in the *dev_class* . Moreover *dev_unit* is greater than all existing logical units in the *dev_class* .

In case of success, the corresponding device entry is locked in the registry until a subsequent `svDeviceRelease` . Note that the device registry lock may or may not be exclusive, depending on the value of the *shared* argument specified in `svDeviceAlloc` . In other words, the device may be shared between multiple driver clients, if *shared* was `TRUE` at registration time. The device driver instance must not disappear while its device entry is locked in the registry.

In the case of a catastrophic device shutdown (for example, hot-plug removal) the device entry lock owners are notified (through the *cli_handler* routine invocation) that the device entry must be released as soon as possible.

The `svDeviceEntry` routine returns the device entry associated with a given client identifier. The *cli_id* argument specifies the client identifier previously returned by `svDeviceLookup` .

The `svDeviceRelease` routine releases the lock on the given device entry. The device entry is specified by the *cli_id* argument. Obviously, a device driver client should no longer access the device driver instance and the device node once the device entry has been released.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
svDeviceAlloc	+	+	-	+
svDeviceFree	+	+	-	+
svDeviceRegister	+	+	-	+
svDeviceUnregister	+	+	-	+
svDeviceEvent	+	+	+	-
svDeviceLookup	+	+	-	+
svDeviceRelease	+	+	-	+
svDeviceEntry	+	+	-	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`dtreeNodeRoot`(9DKI) , `svDkiThreadCall`(9DKI)

NAME

svDeviceRegister, svDeviceAlloc, svDeviceFree, svDeviceUnregister, svDeviceEvent, svDeviceLookup, svDeviceEntry, svDeviceRelease – device registry operations

SYNOPSIS

```
#include <dki/dki.h>
void svDeviceRegister(DevRegId dev_id);

DevRegId svDeviceAlloc(DevRegEntry * entry, unsigned int version, Bool shared,
DevRelHandler handler);

KnError svDeviceUnregister(DevRegId dev_id);

void svDeviceEvent(DevRegId dev_id, DevEvent event, void* arg);

void svDeviceFree(DevRegId dev_id);

KnError svDeviceLookup(char * dev_class, unsigned int dev_version, unsigned int
dev_unit, DevEventHandler cli_handler, void * cli_cookie, DevClientId * cli_id);

DevRegEntry * svDeviceEntry(DevClientId cli_id);

void svDeviceRelease(DevClientId cli_id);
typedef struct {
        char* dev_class;
        void* dev_ops;
        void* dev_id;
        DevNode dev_node;
    } DevRegEntry;
```

FEATURES

DKI

DESCRIPTION

Provides access to microkernel services implementing the `device registry`.

EXTENDED DESCRIPTION

The `device registry` microkernel module implements a data base of driver instances servicing devices currently supported by the system. The `device registry` data base is populated by drivers which perform self-registration (using `svDeviceRegister`) at device initialization time. The `device registry` data base is accessed by driver clients in order to obtain a pointer to the driver instance servicing a given (logical) device. The `device registry` API is described below in detail.

Note that only the `svDeviceLookup`, `svDeviceRelease` and `svDeviceEntry` microkernel calls should be used by driver clients. The rest of the API is dedicated to device drivers.

`svDeviceAlloc` allocates a device registry entry for a given device driver instance. Note that the entry is allocated in an invalid state. This means that the entry is not included in the registry (it is not visible to clients via `svDeviceLookup`).

The entry becomes valid (visible to clients) when a subsequent `svDeviceRegister` is invoked. On the other hand, the driver is allowed to call `svDeviceEvent` on this type of entry. Shut down events signaled on an invalid entry are memorized by the device registry and they are processed when the entry becomes valid (that is, when `svDeviceRegister` is called).

The *entry* argument points to the `DevRegEntry` structure which designates the device driver instance. The DevRegEntry structure is described below:

dev_class	Points to a string specifying the device class name.
dev_ops	Points to a structure of driver service routines.
dev_id	Points to a handle which is usually passed back to the driver (as first argument) each time a driver service routine is invoked.
dev_node	Specifies the device node in the tree which is serviced by the driver. Note that a driver client must switch to the DKI thread context when accessing the device tree structure (see section “Device Node Properties”).

The *version* argument specifies the driver interface version implemented by the device driver instance.

The *dev_class* field and *version* argument specify APIs implemented by the driver. This type of API is provided to driver clients as a structure of indirect functions implementing the API service routines. For instance, the *"uart"* device API is specified by the `UartDevOps` structure, the *"timer"* device API is specified by the `TimerDevOps` structure. A pointer to this type of structure is obtained by a device driver client from the *dev_ops* field of the device registry entry.

The *shared* argument specifies whether the device driver instance may be shared between multiple clients. In other words, it indicates whether the registered instance may be looked up multiple times.

The *handler* argument specifies the driver handler which is invoked by the device registry module as an acknowledgement to a shut-down event (see `svDeviceEvent`). *handler* is called when the last reference to the device registry

entry goes away and a shut-down event has been previously signalled on the entry. *entry* is passed back to the driver as the argument of *handler* .

In case of success, a non zero *DevRegId* is returned, otherwise `NULL` is returned.

Device registry assigns a logical unit number to the physical device. The logical unit number is unique within *dev_class* . The device registry handles a logical unit counter per class. All counters are initialized to zero. Each time a new device entry's is allocated, the current counter value is assigned to the entry logical unit number and the counter is incremented. Thus, the logical device order within a class corresponds to the allocation order.

The logical unit number is mainly used by device driver clients to iterate through a given device class looking for a certain device. Using a valid logical unit number, a client is able to access the corresponding device registry entry, in particular, the *dev_node* field which points to the device node. The physical device corresponding to a given logical unit number may then be detected by the device node properties and the node position within the device tree.

Note - The device driver may allocate (and then register) multiple entries in order to declare different classes (APIs) for the same physical device. For instance, a driver may implement orthogonal sets of operations like power management which are additional to the main device functionality. Another example is a device driver servicing a multi-function device. This type of driver would register multiple entries: one entry per device function. From the client's point of view, this type of device would be visible as multiple (mono-function) independent devices.

`svDeviceRegister` adds a given device registry entry to the registry. The entry must be previously allocated by `svDeviceAlloc` . The entry becomes valid (and therefore visible for clients) only if there is no shut-down event signalled on the entry. Otherwise, the entry remains invalid and the device registry module invokes a handler previously specified via `svDeviceAlloc` . A pointer to the `DevRegEntry` structure (previously specified via `svDeviceAlloc`) is passed back to the driver as the argument of *handler* . The *handler* routine is called in the DKI thread context.

Note - When *handler* is called the entry is no longer registered. Thus, it is useless to invoke `svDeviceUnregister` once *handler* has been called. On the other hand, in order to free memory resources, the driver should release the device registry entry invoking `svDeviceFree` once the entry is no longer used by the driver. Typically, the entry is no longer used by the driver when the connection to the parent bus / nexus driver is closed, and the parent driver is unable to invoke a call-back handler signalling a bus / nexus event.

`svDeviceUnregister` removes the device entry from the device registry (if the entry is valid). The device entry being removed is specified by the *dev_id* argument. `svDeviceUnregister` returns the following results when a valid entry is specified:

K_OK	The device entry has been successfully removed from the registry.
K_EBUSY	The device entry has not been removed from the registry because it is locked by a driver client. In other words, the `svDeviceLookup/Release` pairs are not balanced.

`svDeviceUnregister` returns `K_EBUSY` when an invalid entry is specified. The driver must remove all related device entries from the registry prior to being unloaded from the system. Typically, `svDeviceUnregister` is issued by the device driver (for each device driver instance) when the driver code is requested to be unloaded via `drv_unload` . The driver unload fails when the driver code is still being used by the system.

`svDeviceEvent` notifies the device registry module that a shutdown event has occurred (for example, that a hot-pluggable device has been removed). The shutdown event basically means that the driver instance should no longer be used by driver clients and the device entry must be removed from the registry (that is, the driver instance will disappear). The shutdown event is immediately propagated to all driver clients in order to stop the device usage and to release the device entry as soon as possible. The device is specified by the *dev_id* argument.

`svDeviceEvent` propagates the shutdown event invoking the *event_handler* routines specified by the driver clients in `svDeviceLookup` . The *event* and *arg* arguments are opaque for the device registry. They are passed (as arguments) to the client event handlers. *event* specifies the reason for the shut-down as follows:

DEV_EVENT_SHUTDOWN	Normal device shut down
DEV_EVENT_REMOVAL	Hot-plug (surprise) device removal

arg is *event* specific.

All driver clients are requested to release the device entry (as soon as possible) invoking the `svDeviceRelease` routine. Note that, prior to the shutdown event propagation, `svDeviceEvent` removes the device entry from the registry in order to prevent the entry from being found (and locked) by new driver clients. Once all driver client handlers are invoked, `svDeviceEvent` returns to the driver. Note that the real device shutdown has to be deferred until the *handler* routine invocation.

Once the device entry is released by the last driver client, the device registry module invokes *handler* previously specified via `svDeviceAlloc` . A pointer to the `DevRegEntry` structure (previously specified via `svDeviceAlloc`) is passed back to the driver as the argument of *handler* . The handler routine is called in the DKI thread context.

Typically, `svDeviceEvent` is used by a device driver servicing a hot-pluggable device. `svDeviceEvent` is invoked by the device driver when the driver is notified (by its parent driver) that the device has been removed from the bus.

Note - In the case of hot-plug removal, the device driver must still be operational until the device entry is released (ignoring all requests to the driver, for example). The driver is allowed to call `svDeviceEvent` on an invalid (unregistered) entry. In this type of case, *handler* will be invoked only when a registration attempt is made, that is, `svDeviceRegister` will be invoked. In addition, the registration will fail and the entry will remain invalid.

`svDeviceFree` releases a given device registry entry previously allocated by `svDeviceAlloc` . The device entry being released is specified by the *dev_id* argument. The entry being released must be invalid (unregistered). Thus, if the entry was registered by `svDeviceRegister` , the driver is allowed to release it if either the entry is successfully unregistered by `svDeviceUnregister` or *handler* (previously specified by `svDeviceAlloc`) is invoked.

`svDeviceLookup` searches the device entry in the registry matching the specified device class and logical unit.

dev_class	Specifies the device class.
dev_version	Specifies the minimum device driver interface version required.
dev_unit	Specifies the logical device unit in the class.
cli_handler	Specifies the event handler which is called when a device event is signalled.
cli_cookie	Specifies the first argument of *cli_handler* .
cli_id	Is an output argument identifying the client token on the matching device entry. This *cli_id* is passed back as an argument to `svDeviceEntry` to get the associated device entry, or to `svDeviceRelease` to release the device driver instance.

`svDeviceLookup` returns the following results:

K_OK	The device entry matching the *dev_class* and *dev_unit* pair was found in the registry.
K_EBUSY	The device entry matching the *dev_class* and *dev_unit* pair was not found in the registry. Or, the device is being used by another client and device sharing is not allowed.
K_EUNKNOWN	There is no device entry in the registry matching *dev_class* .
K_UNDEF	There is no device entry in the registry matching *dev_unit* in the *dev_class* . Or, there is a device entry in the *dev_class* with a logical unit greater than *dev_unit* . In other words, *dev_unit* falls into a hole in the *dev_class* unit ranges. The device entry matching the *dev_class* and *dev_unit* pair found in the registry is implementing an older version of the interface than the one required.
K_ETOOMUCH	There is no device entry in the registry matching *dev_unit* in the *dev_class* . Moreover *dev_unit* is greater than all existing logical units in the *dev_class* .

In case of success, the corresponding device entry is locked in the registry until a subsequent `svDeviceRelease` . Note that the device registry lock may or may not be exclusive, depending on the value of the *shared* argument specified in `svDeviceAlloc` . In other words, the device may be shared between multiple driver clients, if *shared* was `TRUE` at registration time. The device driver instance must not disappear while its device entry is locked in the registry.

In the case of a catastrophic device shutdown (for example, hot-plug removal) the device entry lock owners are notified (through the *cli_handler* routine invocation) that the device entry must be released as soon as possible.

The `svDeviceEntry` routine returns the device entry associated with a given client identifier. The *cli_id* argument specifies the client identifier previously returned by `svDeviceLookup` .

The `svDeviceRelease` routine releases the lock on the given device entry. The device entry is specified by the *cli_id* argument. Obviously, a device driver client should no longer access the device driver instance and the device node once the device entry has been released.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
svDeviceAlloc	+	+	-	+
svDeviceFree	+	+	-	+
svDeviceRegister	+	+	-	+
svDeviceUnregister	+	+	-	+
svDeviceEvent	+	+	+	-
svDeviceLookup	+	+	-	+
svDeviceRelease	+	+	-	+
svDeviceEntry	+	+	-	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`dtreeNodeRoot`(9DKI) , `svDkiThreadCall`(9DKI)

NAME

svDeviceRegister, svDeviceAlloc, svDeviceFree, svDeviceUnregister, svDeviceEvent, svDeviceLookup, svDeviceEntry, svDeviceRelease – device registry operations

SYNOPSIS

```
#include <dki/dki.h>
void svDeviceRegister(DevRegId dev_id);

DevRegId svDeviceAlloc(DevRegEntry * entry, unsigned int version, Bool shared,
DevRelHandler handler);

KnError svDeviceUnregister(DevRegId dev_id);

void svDeviceEvent(DevRegId dev_id, DevEvent event, void* arg);

void svDeviceFree(DevRegId dev_id);

KnError svDeviceLookup(char * dev_class, unsigned int dev_version, unsigned int
dev_unit, DevEventHandler cli_handler, void * cli_cookie, DevClientId * cli_id);

DevRegEntry * svDeviceEntry(DevClientId cli_id);

void svDeviceRelease(DevClientId cli_id);
typedef struct {
        char* dev_class;
        void* dev_ops;
        void* dev_id;
        DevNode dev_node;
    } DevRegEntry;
```

FEATURES

DKI

DESCRIPTION

Provides access to microkernel services implementing the `device registry`.

EXTENDED DESCRIPTION

The `device registry` microkernel module implements a data base of driver instances servicing devices currently supported by the system. The `device registry` data base is populated by drivers which perform self-registration (using `svDeviceRegister`) at device initialization time. The `device registry` data base is accessed by driver clients in order to obtain a pointer to the driver instance servicing a given (logical) device. The `device registry` API is described below in detail.

Note that only the `svDeviceLookup`, `svDeviceRelease` and `svDeviceEntry` microkernel calls should be used by driver clients. The rest of the API is dedicated to device drivers.

`svDeviceAlloc` allocates a device registry entry for a given device driver instance. Note that the entry is allocated in an invalid state. This means that the entry is not included in the registry (it is not visible to clients via `svDeviceLookup`).

The entry becomes valid (visible to clients) when a subsequent `svDeviceRegister` is invoked. On the other hand, the driver is allowed to call `svDeviceEvent` on this type of entry. Shut down events signaled on an invalid entry are memorized by the device registry and they are processed when the entry becomes valid (that is, when `svDeviceRegister` is called).

The *entry* argument points to the `DevRegEntry` structure which designates the device driver instance. The DevRegEntry structure is described below:

dev_class	Points to a string specifying the device class name.
dev_ops	Points to a structure of driver service routines.
dev_id	Points to a handle which is usually passed back to the driver (as first argument) each time a driver service routine is invoked.
dev_node	Specifies the device node in the tree which is serviced by the driver. Note that a driver client must switch to the DKI thread context when accessing the device tree structure (see section “Device Node Properties”).

The *version* argument specifies the driver interface version implemented by the device driver instance.

The *dev_class* field and *version* argument specify APIs implemented by the driver. This type of API is provided to driver clients as a structure of indirect functions implementing the API service routines. For instance, the *"uart"* device API is specified by the `UartDevOps` structure, the *"timer"* device API is specified by the `TimerDevOps` structure. A pointer to this type of structure is obtained by a device driver client from the *dev_ops* field of the device registry entry.

The *shared* argument specifies whether the device driver instance may be shared between multiple clients. In other words, it indicates whether the registered instance may be looked up multiple times.

The *handler* argument specifies the driver handler which is invoked by the device registry module as an acknowledgement to a shut-down event (see `svDeviceEvent`). *handler* is called when the last reference to the device registry

entry goes away and a shut-down event has been previously signalled on the entry. *entry* is passed back to the driver as the argument of *handler* .

In case of success, a non zero *DevRegId* is returned, otherwise `NULL` is returned.

Device registry assigns a logical unit number to the physical device. The logical unit number is unique within *dev_class* . The device registry handles a logical unit counter per class. All counters are initialized to zero. Each time a new device entry's is allocated, the current counter value is assigned to the entry logical unit number and the counter is incremented. Thus, the logical device order within a class corresponds to the allocation order.

The logical unit number is mainly used by device driver clients to iterate through a given device class looking for a certain device. Using a valid logical unit number, a client is able to access the corresponding device registry entry, in particular, the *dev_node* field which points to the device node. The physical device corresponding to a given logical unit number may then be detected by the device node properties and the node position within the device tree.

Note - The device driver may allocate (and then register) multiple entries in order to declare different classes (APIs) for the same physical device. For instance, a driver may implement orthogonal sets of operations like power management which are additional to the main device functionality. Another example is a device driver servicing a multi-function device. This type of driver would register multiple entries: one entry per device function. From the client's point of view, this type of device would be visible as multiple (mono-function) independent devices.

`svDeviceRegister` adds a given device registry entry to the registry. The entry must be previously allocated by `svDeviceAlloc` . The entry becomes valid (and therefore visible for clients) only if there is no shut-down event signalled on the entry. Otherwise, the entry remains invalid and the device registry module invokes a handler previously specified via `svDeviceAlloc` . A pointer to the `DevRegEntry` structure (previously specified via `svDeviceAlloc`) is passed back to the driver as the argument of *handler* . The *handler* routine is called in the DKI thread context.

Note - When *handler* is called the entry is no longer registered. Thus, it is useless to invoke `svDeviceUnregister` once *handler* has been called. On the other hand, in order to free memory resources, the driver should release the device registry entry invoking `svDeviceFree` once the entry is no longer used by the driver. Typically, the entry is no longer used by the driver when the connection to the parent bus / nexus driver is closed, and the parent driver is unable to invoke a call-back handler signalling a bus / nexus event.

`svDeviceUnregister` removes the device entry from the device registry (if the entry is valid). The device entry being removed is specified by the *dev_id* argument. `svDeviceUnregister` returns the following results when a valid entry is specified:

K_OK	The device entry has been successfully removed from the registry.
K_EBUSY	The device entry has not been removed from the registry because it is locked by a driver client. In other words, the `svDeviceLookup/Release` pairs are not balanced.

`svDeviceUnregister` returns `K_EBUSY` when an invalid entry is specified. The driver must remove all related device entries from the registry prior to being unloaded from the system. Typically, `svDeviceUnregister` is issued by the device driver (for each device driver instance) when the driver code is requested to be unloaded via `drv_unload` . The driver unload fails when the driver code is still being used by the system.

`svDeviceEvent` notifies the device registry module that a shutdown event has occurred (for example, that a hot-pluggable device has been removed). The shutdown event basically means that the driver instance should no longer be used by driver clients and the device entry must be removed from the registry (that is, the driver instance will disappear). The shutdown event is immediately propagated to all driver clients in order to stop the device usage and to release the device entry as soon as possible. The device is specified by the *dev_id* argument.

`svDeviceEvent` propagates the shutdown event invoking the *event_handler* routines specified by the driver clients in `svDeviceLookup` . The *event* and *arg* arguments are opaque for the device registry. They are passed (as arguments) to the client event handlers. *event* specifies the reason for the shut-down as follows:

DEV_EVENT_SHUTDOWN	Normal device shut down
DEV_EVENT_REMOVAL	Hot-plug (surprise) device removal

arg is *event* specific.

All driver clients are requested to release the device entry (as soon as possible) invoking the `svDeviceRelease` routine. Note that, prior to the shutdown event propagation, `svDeviceEvent` removes the device entry from the registry in order to prevent the entry from being found (and locked) by new driver clients. Once all driver client handlers are invoked, `svDeviceEvent` returns to the driver. Note that the real device shutdown has to be deferred until the *handler* routine invocation.

Once the device entry is released by the last driver client, the device registry module invokes *handler* previously specified via `svDeviceAlloc` . A pointer to the `DevRegEntry` structure (previously specified via `svDeviceAlloc`) is passed back to the driver as the argument of *handler* . The handler routine is called in the DKI thread context.

Typically, `svDeviceEvent` is used by a device driver servicing a hot-pluggable device. `svDeviceEvent` is invoked by the device driver when the driver is notified (by its parent driver) that the device has been removed from the bus.

Note - In the case of hot-plug removal, the device driver must still be operational until the device entry is released (ignoring all requests to the driver, for example). The driver is allowed to call `svDeviceEvent` on an invalid (unregistered) entry. In this type of case, *handler* will be invoked only when a registration attempt is made, that is, `svDeviceRegister` will be invoked. In addition, the registration will fail and the entry will remain invalid.

`svDeviceFree` releases a given device registry entry previously allocated by `svDeviceAlloc` . The device entry being released is specified by the *dev_id* argument. The entry being released must be invalid (unregistered). Thus, if the entry was registered by `svDeviceRegister` , the driver is allowed to release it if either the entry is successfully unregistered by `svDeviceUnregister` or *handler* (previously specified by `svDeviceAlloc`) is invoked.

`svDeviceLookup` searches the device entry in the registry matching the specified device class and logical unit.

dev_class	Specifies the device class.
dev_version	Specifies the minimum device driver interface version required.
dev_unit	Specifies the logical device unit in the class.
cli_handler	Specifies the event handler which is called when a device event is signalled.
cli_cookie	Specifies the first argument of *cli_handler* .
cli_id	Is an output argument identifying the client token on the matching device entry. This *cli_id* is passed back as an argument to `svDeviceEntry` to get the associated device entry, or to `svDeviceRelease` to release the device driver instance.

`svDeviceLookup` returns the following results:

K_OK	The device entry matching the *dev_class* and *dev_unit* pair was found in the registry.
K_EBUSY	The device entry matching the *dev_class* and *dev_unit* pair was not found in the registry. Or, the device is being used by another client and device sharing is not allowed.
K_EUNKNOWN	There is no device entry in the registry matching *dev_class* .
K_UNDEF	There is no device entry in the registry matching *dev_unit* in the *dev_class* . Or, there is a device entry in the *dev_class* with a logical unit greater than *dev_unit* . In other words, *dev_unit* falls into a hole in the *dev_class* unit ranges. The device entry matching the *dev_class* and *dev_unit* pair found in the registry is implementing an older version of the interface than the one required.
K_ETOOMUCH	There is no device entry in the registry matching *dev_unit* in the *dev_class* . Moreover *dev_unit* is greater than all existing logical units in the *dev_class* .

In case of success, the corresponding device entry is locked in the registry until a subsequent `svDeviceRelease` . Note that the device registry lock may or may not be exclusive, depending on the value of the *shared* argument specified in `svDeviceAlloc` . In other words, the device may be shared between multiple driver clients, if *shared* was `TRUE` at registration time. The device driver instance must not disappear while its device entry is locked in the registry.

In the case of a catastrophic device shutdown (for example, hot-plug removal) the device entry lock owners are notified (through the *cli_handler* routine invocation) that the device entry must be released as soon as possible.

The `svDeviceEntry` routine returns the device entry associated with a given client identifier. The *cli_id* argument specifies the client identifier previously returned by `svDeviceLookup` .

The `svDeviceRelease` routine releases the lock on the given device entry. The device entry is specified by the *cli_id* argument. Obviously, a device driver client should no longer access the device driver instance and the device node once the device entry has been released.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
svDeviceAlloc	+	+	-	+
svDeviceFree	+	+	-	+
svDeviceRegister	+	+	-	+
svDeviceUnregister	+	+	-	+
svDeviceEvent	+	+	+	-
svDeviceLookup	+	+	-	+
svDeviceRelease	+	+	-	+
svDeviceEntry	+	+	-	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`dtreeNodeRoot`(9DKI) , `svDkiThreadCall`(9DKI)

NAME

svDkiOpen, svDkiClose, svDkiEvent – system event management

SYNOPSIS

```
#include <dki/dki.h>
KnError svDkiOpen(DevNode dev_node, DkiEventHandler dev_evt_handler,
DkiLoadHandler dev_load_handler, void * dev_cookie, DkiDevId * dev_id);

void svDkiClose(DkiDevId dev_id);

void svDkiEvent(DkiEvent event, void * arg);
```

FEATURES

DKI

DESCRIPTION

The system event management services are provided by the microkernel to the lowest layer drivers. They are used mainly to register event handlers for all running drivers, and to start propagating an event from the microkernel.

EXTENDED DESCRIPTION

Typically, a system reboot starts propagating a specific event from the microkernel to the lowest-layer level drivers. Those drivers then recursively propagate the event to the upper layer drivers by calling their `BusEventHandler` handler, registered at `open` time.

`svDkiOpen` must be issued by the lowest layer level drivers. It establishes a connection between the device driver and DKI.

The *dev_node* argument specifies the device node (in the device tree) which is serviced by the device driver instance. In case of initialization, the device node is given as an argument of `drv_init` by the parent bus driver. In case of probing, the device node is either found (among existing child nodes attached to the parent node) or created (and attached to the parent node) by the device driver.

The *dev_evt_handler* argument specifies the device driver handler which is invoked by the DKI when an event occurs. It takes three arguments. The first argument is the *dev_cookie* . The second one specifies the DKI event type. The third argument points to a structure which is event type specific.

DKI events are defined below:

DKI_SYS_SHUTDOWN	Notifies a device driver that the system is going to be shut down. The device driver should propagate the event to all client drivers if it is a bus driver. Otherwise, it should reset the device hardware and return from the event handler. Note that the driver must neither notify clients nor free allocated resources.

Typically, *dev_evt_handler* is called as an interrupt handler and therefore the handler implementation must be restricted to the API allowed at interrupt level.

The *dev_load_handler* argument specifies the device driver handler which is invoked by the DKI when a new driver has been dynamically loaded. It is invoked passing *dev_cookie* as the only argument. Note that this *dev_load_handler* handler is optional.

Typically, it should be used only by bus drivers supporting dynamically loadable device drivers, and should be set to `NULL` by all other drivers. This type of bus driver handler should manage the newly loaded driver in a similar way to the driver's initialization at boot time. That is, associate the driver with a device node, and initialize it, in order to create a running instance of the newly loaded driver. Note that the *dev_load_handler* routine is invoked in the DKI thread context.

The *dev_cookie* argument specifies a device driver cookie. It is opaque for the DKI. *dev_cookie* is passed back to the driver when *dev_evt_handler* or *dev_load_handler* is invoked.

Upon successful completion, `svDkiOpen` returns `K_OK` and an identifier designating the DKI / device connection is returned in the *dev_id* argument. The *dev_id* is opaque for the driver, it must be passed back to the DKI as an argument of the `svDkiClose` service routine. In case of failure, an error code is returned as described below:

K_EINVAL	The *dev_node* argument provided is not a valid device tree node.
K_EBUSY	The *dev_node* device tree node provided is already in use (associated to another driver).
K_ENOMEM	The system is out of memory.

The `svDkiClose` routine releases the DKI / driver connection. It must be the last call issued by the driver.

`svDkiEvent` is used to start propagating a given event to the device driver hierarchy. It calls all `DkiEventHandler` handlers registered through `svDkiOpen` to signal all running drivers that the given event occurred. The *event* argument specifies the event to propagate. The *arg* argument specifies the specific parameter associated with the event type.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service.

Services	Base level	DKI thread	Interrupt	Blocking
svDkiOpen	-	+	-	+

svDkiClose	-	+	-	+
svDkiEvent	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

svDkiOpen, svDkiClose, svDkiEvent – system event management

SYNOPSIS

#include <dki/dki.h>
KnError **svDkiOpen**(DevNode *dev_node*, DkiEventHandler *dev_evt_handler*, DkiLoadHandler *dev_load_handler*, void * *dev_cookie*, DkiDevId * *dev_id*);

void **svDkiClose**(DkiDevId *dev_id*);

void **svDkiEvent**(DkiEvent *event*, void * *arg*);

FEATURES

DKI

DESCRIPTION

The system event management services are provided by the microkernel to the lowest layer drivers. They are used mainly to register event handlers for all running drivers, and to start propagating an event from the microkernel.

EXTENDED DESCRIPTION

Typically, a system reboot starts propagating a specific event from the microkernel to the lowest-layer level drivers. Those drivers then recursively propagate the event to the upper layer drivers by calling their `BusEventHandler` handler, registered at `open` time.

`svDkiOpen` must be issued by the lowest layer level drivers. It establishes a connection between the device driver and DKI.

The *dev_node* argument specifies the device node (in the device tree) which is serviced by the device driver instance. In case of initialization, the device node is given as an argument of `drv_init` by the parent bus driver. In case of probing, the device node is either found (among existing child nodes attached to the parent node) or created (and attached to the parent node) by the device driver.

The *dev_evt_handler* argument specifies the device driver handler which is invoked by the DKI when an event occurs. It takes three arguments. The first argument is the *dev_cookie* . The second one specifies the DKI event type. The third argument points to a structure which is event type specific.

DKI events are defined below:

DKI_SYS_SHUTDOWN — Notifies a device driver that the system is going to be shut down. The device driver should propagate the event to all client drivers if it is a bus driver. Otherwise, it should reset the device hardware and return from the event handler. Note that the driver must neither notify clients nor free allocated resources.

Typically, *dev_evt_handler* is called as an interrupt handler and therefore the handler implementation must be restricted to the API allowed at interrupt level.

The *dev_load_handler* argument specifies the device driver handler which is invoked by the DKI when a new driver has been dynamically loaded. It is invoked passing *dev_cookie* as the only argument. Note that this *dev_load_handler* handler is optional.

Typically, it should be used only by bus drivers supporting dynamically loadable device drivers, and should be set to `NULL` by all other drivers. This type of bus driver handler should manage the newly loaded driver in a similar way to the driver's initialization at boot time. That is, associate the driver with a device node, and initialize it, in order to create a running instance of the newly loaded driver. Note that the *dev_load_handler* routine is invoked in the DKI thread context.

The *dev_cookie* argument specifies a device driver cookie. It is opaque for the DKI. *dev_cookie* is passed back to the driver when *dev_evt_handler* or *dev_load_handler* is invoked.

Upon successful completion, `svDkiOpen` returns `K_OK` and an identifier designating the DKI / device connection is returned in the *dev_id* argument. The *dev_id* is opaque for the driver, it must be passed back to the DKI as an argument of the `svDkiClose` service routine. In case of failure, an error code is returned as described below:

K_EINVAL	The *dev_node* argument provided is not a valid device tree node.
K_EBUSY	The *dev_node* device tree node provided is already in use (associated to another driver).
K_ENOMEM	The system is out of memory.

The `svDkiClose` routine releases the DKI / driver connection. It must be the last call issued by the driver.

`svDkiEvent` is used to start propagating a given event to the device driver hierarchy. It calls all `DkiEventHandler` handlers registered through `svDkiOpen` to signal all running drivers that the given event occurred. The *event* argument specifies the event to propagate. The *arg* argument specifies the specific parameter associated with the event type.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service.

Services	Base level	DKI thread	Interrupt	Blocking
svDkiOpen	-	+	-	+

svDkiClose	-	+	-	+
svDkiEvent	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

svDkiOpen, svDkiClose, svDkiEvent – system event management

SYNOPSIS

```
#include <dki/dki.h>
KnError svDkiOpen(DevNode dev_node, DkiEventHandler dev_evt_handler,
DkiLoadHandler dev_load_handler, void * dev_cookie, DkiDevId * dev_id);

void svDkiClose(DkiDevId dev_id);

void svDkiEvent(DkiEvent event, void * arg);
```

FEATURES

DKI

DESCRIPTION

The system event management services are provided by the microkernel to the lowest layer drivers. They are used mainly to register event handlers for all running drivers, and to start propagating an event from the microkernel.

EXTENDED DESCRIPTION

Typically, a system reboot starts propagating a specific event from the microkernel to the lowest-layer level drivers. Those drivers then recursively propagate the event to the upper layer drivers by calling their `BusEventHandler` handler, registered at `open` time.

`svDkiOpen` must be issued by the lowest layer level drivers. It establishes a connection between the device driver and DKI.

The *dev_node* argument specifies the device node (in the device tree) which is serviced by the device driver instance. In case of initialization, the device node is given as an argument of `drv_init` by the parent bus driver. In case of probing, the device node is either found (among existing child nodes attached to the parent node) or created (and attached to the parent node) by the device driver.

The *dev_evt_handler* argument specifies the device driver handler which is invoked by the DKI when an event occurs. It takes three arguments. The first argument is the *dev_cookie* . The second one specifies the DKI event type. The third argument points to a structure which is event type specific.

DKI events are defined below:

DKI_SYS_SHUTDOWN	Notifies a device driver that the system is going to be shut down. The device driver should propagate the event to all client drivers if it is a bus driver. Otherwise, it should reset the device hardware and return from the event handler. Note that the driver must neither notify clients nor free allocated resources.

Typically, *dev_evt_handler* is called as an interrupt handler and therefore the handler implementation must be restricted to the API allowed at interrupt level.

The *dev_load_handler* argument specifies the device driver handler which is invoked by the DKI when a new driver has been dynamically loaded. It is invoked passing *dev_cookie* as the only argument. Note that this *dev_load_handler* handler is optional.

Typically, it should be used only by bus drivers supporting dynamically loadable device drivers, and should be set to `NULL` by all other drivers. This type of bus driver handler should manage the newly loaded driver in a similar way to the driver's initialization at boot time. That is, associate the driver with a device node, and initialize it, in order to create a running instance of the newly loaded driver. Note that the *dev_load_handler* routine is invoked in the DKI thread context.

The *dev_cookie* argument specifies a device driver cookie. It is opaque for the DKI. *dev_cookie* is passed back to the driver when *dev_evt_handler* or *dev_load_handler* is invoked.

Upon successful completion, `svDkiOpen` returns `K_OK` and an identifier designating the DKI / device connection is returned in the *dev_id* argument. The *dev_id* is opaque for the driver, it must be passed back to the DKI as an argument of the `svDkiClose` service routine. In case of failure, an error code is returned as described below:

K_EINVAL	The *dev_node* argument provided is not a valid device tree node.
K_EBUSY	The *dev_node* device tree node provided is already in use (associated to another driver).
K_ENOMEM	The system is out of memory.

The `svDkiClose` routine releases the DKI / driver connection. It must be the last call issued by the driver.

`svDkiEvent` is used to start propagating a given event to the device driver hierarchy. It calls all `DkiEventHandler` handlers registered through `svDkiOpen` to signal all running drivers that the given event occurred. The *event* argument specifies the event to propagate. The *arg* argument specifies the specific parameter associated with the event type.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service.

Services	Base level	DKI thread	Interrupt	Blocking
svDkiOpen	-	+	-	+

svDkiClose	-	+	-	+
svDkiEvent	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

svDkiThreadCall, svDkiThreadTrigger – call a routine in the DKI thread context

SYNOPSIS

#include <dki/dki.h>
void **svDkiThreadCall**(DkiCallToken * *token*, DkiCallHandler *handler*, void * *cookie*);

void **svDkiThreadTrigger**(DkiCallToken * *token*, DkiCallHandler *handler*, void * *cookie*);

FEATURES

DKI

DESCRIPTION

Provides synchronization services through the DKI thread.

EXTENDED DESCRIPTION

The DKI thread is launched by the microkernel at initialization time.

The purpose of this thread is to synchronize identical calls to DKI and DDI services issued from different threads at drivers' initialization time as well as at runtime. This is typically used for initialization and shutdown of drivers. By ensuring such synchronization, the DKI thread avoids using any other synchronization mechanism (locks) in the driver implementations.

Refer to the tables in each section, that indicate the allowed calling level, to know which services should to be called in the context of the DKI thread to ensure coherency. The DKI thread is involved as a synchronization mechanism in the following two cases:

- *Normal case*

  In the normal case, all calls related to initialization / shutdown of the drivers are done implicitly in the context of the DKI thread. That means that drivers should not worry about synchronization, and do nothing, because their routines are called directly from the DKI thread.

- *Specific cases*

  There are two special cases in which a driver should use DKI thread services explicitly to ensure synchronization:

  - *Hot-pluggable device drivers*

    In the case of a hot-pluggable device driver, the initialization/shutdown process has to be executed at runtime and not as part of the kernel/drivers initialization process. In this type of case, drivers should use the DKI thread services below to synchronize explicitly with drivers already running.

  - *Deferred driver initialization*

    In some cases, a driver may defer its device initialization until it is opened. This is a way to resolve conflicts about usage of the same resource by multiple drivers. In that way, drivers sharing a resource can be loaded at same time, if they are not opened at the same time. In this type of deferred initialization scheme, the initialization/shutdown process must be executed at runtime (at time of open/close) and not as

part of the kernel / drivers initialization process. Thus, these kinds of drivers should also use the DKI thread services below to synchronize explicitly with drivers already running.

`svDkiThreadCall` synchronously invokes a routine in the context of the DKI thread. Synchronously means that the caller is blocked until the invoked routine returns.

The *token* argument is the address of a `DkiCallToken` structure which must be allocated by the caller. This structure is opaque for the driver, and is used only by the DKI thread. Note that the same structure may be reused for subsequent calls. The *handler* argument specifies the routine to call. The *cookie* argument specifies the argument to pass to the *handler* routine when called.

`svDkiThreadTrigger` asynchronously invokes a routine in the context of the DKI thread. Asynchronously means that the function immediately returns to the caller without waiting for the invoked routine to return. If the driver needs to know when the handler returns, it should use any synchronization mechanism inside the handler itself. The *token* argument is the address of a `DkiCallToken` structure which must be allocated by the caller. This structure is opaque to the driver, and is used only by the DKI thread. Note that the same structure may be reused for subsequent calls, once the handler is invoked. The *handler* argument specifies the routine to call. The *cookie* argument specifies the argument to pass to the *handler* routine when called.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
svDkiThreadCall	+	+	-	+
svDkiThreadTrigger	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

svDkiThreadCall, svDkiThreadTrigger – call a routine in the DKI thread context

SYNOPSIS

#include <dki/dki.h>
void **svDkiThreadCall**(DkiCallToken * *token*, DkiCallHandler *handler*, void * *cookie*);

void **svDkiThreadTrigger**(DkiCallToken * *token*, DkiCallHandler *handler*, void * *cookie*);

FEATURES

DKI

DESCRIPTION

Provides synchronization services through the DKI thread.

EXTENDED DESCRIPTION

The DKI thread is launched by the microkernel at initialization time.

The purpose of this thread is to synchronize identical calls to DKI and DDI services issued from different threads at drivers' initialization time as well as at runtime. This is typically used for initialization and shutdown of drivers. By ensuring such synchronization, the DKI thread avoids using any other synchronization mechanism (locks) in the driver implementations.

Refer to the tables in each section, that indicate the allowed calling level, to know which services should to be called in the context of the DKI thread to ensure coherency. The DKI thread is involved as a synchronization mechanism in the following two cases:

- *Normal case*

  In the normal case, all calls related to initialization / shutdown of the drivers are done implicitly in the context of the DKI thread. That means that drivers should not worry about synchronization, and do nothing, because their routines are called directly from the DKI thread.

- *Specific cases*

  There are two special cases in which a driver should use DKI thread services explicitly to ensure synchronization:

  - *Hot-pluggable device drivers*

    In the case of a hot-pluggable device driver, the initialization/shutdown process has to be executed at runtime and not as part of the kernel/drivers initialization process. In this type of case, drivers should use the DKI thread services below to synchronize explicitly with drivers already running.

  - *Deferred driver initialization*

    In some cases, a driver may defer its device initialization until it is opened. This is a way to resolve conflicts about usage of the same resource by multiple drivers. In that way, drivers sharing a resource can be loaded at same time, if they are not opened at the same time. In this type of deferred initialization scheme, the initialization/shutdown process must be executed at runtime (at time of open/close) and not as

part of the kernel / drivers initialization process. Thus, these kinds of drivers should also use the DKI thread services below to synchronize explicitly with drivers already running.

`svDkiThreadCall` synchronously invokes a routine in the context of the DKI thread. Synchronously means that the caller is blocked until the invoked routine returns.

The *token* argument is the address of a `DkiCallToken` structure which must be allocated by the caller. This structure is opaque for the driver, and is used only by the DKI thread. Note that the same structure may be reused for subsequent calls. The *handler* argument specifies the routine to call. The *cookie* argument specifies the argument to pass to the *handler* routine when called.

`svDkiThreadTrigger` asynchronously invokes a routine in the context of the DKI thread. Asynchronously means that the function immediately returns to the caller without waiting for the invoked routine to return. If the driver needs to know when the handler returns, it should use any synchronization mechanism inside the handler itself. The *token* argument is the address of a `DkiCallToken` structure which must be allocated by the caller. This structure is opaque to the driver, and is used only by the DKI thread. Note that the same structure may be reused for subsequent calls, once the handler is invoked. The *handler* argument specifies the routine to call. The *cookie* argument specifies the argument to pass to the *handler* routine when called.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
svDkiThreadCall	+	+	-	+
svDkiThreadTrigger	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

svDriverRegister, svDriverLookupFirst, svDriverLookupNext, svDriverRelease, svDriverEntry, svDriverCap, svDriverUnregister – driver registry operations

SYNOPSIS

```
#include <dki/dki.h>
KnError svDriverRegister(DrvRegEntry * drv_entry);

DrvRegId svDriverLookupFirst(void);

DrvRegId svDriverLookupNext(DrvRegId drv_id);

void svDriverRelease(DrvRegId drv_id);

DrvRegEntry * svDriverEntry(DrvRegId drv_id);

KnCap * svDriverCap(DrvRegId drv_id);

KnError svDriverUnregister(DrvRegId drv_id);
typedef struct {
  char* drv_name;
  char* drv_info;
  char* bus_class;
  int   bus_version;

    void
  (*drv_probe) (DevNode  bus_node,
          void*    bus_ops,
          void*    bus_id);

    void
  (*drv_bind) (DevNode  bus_node);

    void
  (*drv_init) (DevNode  dev_node,
          void*    bus_ops,
          void*    bus_id);

    KnError
  (*drv_unload) ();
} DrvRegEntry;
```

FEATURES

DKI

DESCRIPTION

The `driver registry` module implements a data base of drivers which have registered within the system. The `driver registry` data base is populated by drivers which perform self-registration (using `svDriverRegister`) at driver initialization time.

EXTENDED DESCRIPTION

`svDriverRegister` adds the driver entry to the driver registry. It returns `K_OK` in case of success, otherwise `K_ENOMEM` is returned to indicate that the system is out of memory. The *drv_entry* argument points to the `DrvRegEntry` structure specifying driver properties and static driver routines. The `DrvRegEntry` structure is described below.

drv_name	Points to a string specifying the driver name. For example, the driver file name.
drv_info	Points to a string specifying extra information about the driver component, such as version or author.
bus_class	Points to a string specifying the class of the parent driver API required for the driver, such as “pci”.
bus_version	Specifies the minimum version of the parent driver API required for the driver. Note that if a bus/nexus driver provides an API version which is less than *bus_version* , the corresponding driver component will never be called by this type of bus/nexus driver. In other words, the *drv_probe* and *drv_init* routines will never be invoked by this type of bus/nexus driver.
drv_probe	Points to a static driver routine which performs the device enumeration/probing on the bus.
drv_bind	Points to a static driver routine which performs the driver-to-device binding.
drv_init	Points to a static driver routine which clones an instance of the driver for the given device.
drv_unload	Points to a static driver routine which is invoked by the driver registry module when somebody wishes to unload the driver code from the system.

The *bus_class* and *bus_version* fields specify a parent bus/nexus API required for the driver. This type of API is provided to the driver as a structure of indirect

functions implementing the API service routines. For instance, the *"pci"* bus API is specified by the `PciBusOps` structure, the *"isa"* bus API is specified by `IsaBusOps` . When a bus driver invokes the `drv_probe` or `drv_init` driver's routine, it provides a pointer to the structure of the bus service routines. The structure type corresponds to the `bus_class` field value.

Note that a bus driver may provide multiple API's. A typical example is a PCI bus driver providing the common and PCI bus APIs. The common bus driver API is named by *"bus"* and specified by the `BusOps` structure. The PCI bus driver API is named by *"pci"* and specified by the `PciBusOps` structure. `BusOps` actually, provides a subset of services provided by `PciBusOps` . This type of bus driver is able to support drivers which use either common (*"bus"*) or PCI (*"pci"*) parent bus interfaces. When the bus driver invokes a child driver, it gives a pointer to either the `BusOps` or `PciBusOps` structure depending on the *bus_class* specified in the child driver registry entry.

Note also that a PROP_DRIVER may be adaptive to the parent bus API. In other words, such a driver is able to run on top of a number of different buses (for example, *"pci"* and *"isa"*). Typically, this type of driver is composed of two parts: bus class specific and bus class independent.

The bus class specific part of the driver code mainly deals with the device probing and initialization. In addition, it provides an abstraction layer in order to hide the bus class dependencies from the bus class independent part of the driver code. This kind of multi-bus driver should be registered multiple times in the driver registry. Each entry specifies a given bus class API (via the *bus_class* field) on top of which the driver may run. In order to determine to which bus class the driver is applied, the `drv_probe` , `drv_bind` and `drv_init` routines have to be entry specific (that is, bus class API specific). Under these conditions, when a given driver's routine is invoked by a bus driver, the driver detects the bus class to which it is applied and casts the *bus_ops* argument to the appropriate structure (for example, `PciBusOps`).

The `drv_probe` routine is invoked by a bus / nexus driver when the *bus_class* specified in the registry entry matches the bus / nexus driver class. `drv_probe` is called with three arguments.

The *dev_node* argument specifies the parent device node.

The *bus_ops* / *bus_id* pair specifies the parent device driver.

bus_ops points to a structure of service routines implementing a bus driver API. This structure is bus class specific and corresponds to the *bus_class* specified by the driver registry entry.

bus_id is opaque to the driver. It must be passed back to the bus driver when the `open` bus service routine is invoked. The `drv_probe` routine is optional. In cases where the `drv_probe` routine is not provided by the driver, the *drv_probe* field must be set to `NULL` .

The `drv_bind` routine is invoked by a bus / nexus driver when the *bus_class* specified in the registry entry matches the bus / nexus driver class.

`drv_bind` is called with one argument, *dev_node* , which specifies a given device node. The `drv_bind` routine enables the driver to perform a driver-to-device binding. Typically, the driver examines properties attached to the device node in order to determine the type of device at the node and to check whether the device may be serviced by the driver. If the check is positive, the driver binds the driver to the device node by attaching a PROP_DRIVER property to the device node. The property value specifies the driver name. The parent bus driver uses such a property to determine the name of the driver servicing the device. In other words, via the PROP_DRIVER property, the child driver gives its name to the parent bus driver, asking it to invoke the `drv_init` routine on that device. Note that, if a `"driver"` property is already present in the device node, then the `drv_bind` routine can not continue; `drv_bind` should not override existing driver-to-device binding.

The `drv_bind` routine is optional. If the `drv_bind` routine is not provided by the driver, the `drv_bind` field must be set to NULL.

The `drv_init` routine is invoked by a bus / nexus driver when the *bus_class* specified in the registry entry matches the bus / nexus driver class and a given driver is bound to a given device node.

`drv_init` is called with three arguments. The *dev_node* argument specifies the device node for which a device driver instance should be created. The *bus_ops* / *bus_id* pair specifies the parent device driver. *bus_ops* points to a structure of service routines implementing a bus driver API.

This structure is bus class specific and corresponds to the *bus_class* specified by the driver registry entry. *bus_id* is opaque to the driver. It must be passed back to the bus driver when the `open` bus service routine is invoked. The purpose of `drv_init` is to create an instance of the driver servicing the given device, to perform a device hardware initialization and to register the driver instance in the device registry .

Typically, `drv_init` would read the device and bus node properties in order to obtain the assigned bus resources and tunable parameters related to the bus / device. The `drv_init` routine is optional. In a case when the drv_init routine is not provided by the driver, the drv_init field must be set to NULL. A typical example of a probe-only driver is a self-identifying bus enumerator (for example, a PCI enumerator) which is implemented as driver. This type of

driver has the drv_probe routine which enumerates devices residing on the bus and creates device nodes. This type of driver obviously does not have the drv_init routine.

drv_unload is called by the driver registry module (more exactly by the svDriverUnregister routine) when somebody wishes to unload the driver code from the system. The purpose of drv_unload is to check that the device driver code is not currently being used. The driver must check, for each driver instance, whether it is locked in the device registry.

In case of success, all device instances are removed from the device registry and K_OK is returned. Otherwise, the device entries are unchanged in the device registry and K_EBUSY is returned.

The drv_unload routine is optional. In cases when drv_unload is not provided by the driver, the drv_unload field must be set to NULL . Note that, in this case, the driver code cannot be unloaded.

svDriverLookupFirst returns the first driver entry in the registry. When the registry is not empty a non zero DrvRegId (designating the entry) is returned, otherwise NULL is returned. In case of success, the driver entry is locked in the registry. It should be unlocked by a subsequent invocation of svDriverRelease or svDriverUnregister .

The driver entry lock is not exclusive. It is simply a counter which prevents the driver component from being unloaded when it is still being used by a bus / nexus driver for probing or initialization.

svDriverLookupNext returns the next driver entry in the registry. The current entry is specified by the *drv_id* argument. The current entry must be locked by a previously called svDriverLookupFirst or svDriverLookupNext . If the current entry is not the last one in the registry a non zero DrvRegId (designating the entry) is returned, otherwise NULL is returned. In case of success, the next driver entry is locked in the registry. It should be unlocked by a subsequent invocation of svDriverRelease or svDriverUnregister .

svDriverRelease releases the lock of the driver entry specified by the *drv_id* argument.

svDriverEntry returns a pointer to the driver entry structure specified by the *drv_id* argument. The device entry being accessed must have been previously locked using svDriverLookupFirst or svDriverLookupNext . Note that the device entry structure is read-only.

`svDriverCap` returns a pointer to the driver actor capability. The driver entry is specified by the *drv_id* argument. The driver entry being accessed must have been previously locked using `svDriverLookupFirst` or `svDriverLookupNext` . Note that if a given driver entry is registered by a built-in driver, a `NULL` pointer is returned. In other words, the driver actor capability makes sense only for dynamically loaded drivers. The actor capability may be used by an application in order to delete the driver actor once the driver entry is unregistered. Note that the driver capability structure is read-only.

`svDriverUnregister` tries to remove the driver entry specified by the *dev_id* argument from the driver registry. The device entry being removed must have been previously locked using `svDriverLookupFirst` or `svDriverLookupNext` .

In case of success, `K_OK` is returned, otherwise `K_EBUSY` is returned. The `K_EBUSY` result means that either the driver entry is locked in the driver registry (that is, a static driver routine is currently being used by a bus/nexus driver) or an instance of the device driver is locked in the device registry (that is, there is a driver instance which is currently being used by a driver client). Note that when `K_EBUSY` is returned, the driver entry remains locked in the registry and should be unlocked explicitly by `svDriverRelease` .

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service.

Services	Base level	DKI thread	Interrupt	Blocking
svDriverRegister	+	+	-	+
svDriverLookupFirst	+	+	-	+
svDriverLookupNext	+	+	-	+
svDriverRelease	+	+	-	+
svDriverEntry	+	+	-	-
svDriverCap	+	+	-	-
svDriverUnregister	+	+	-	+

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`dtreeNodeRoot`(9DKI) , `svDkiThreadCall`(9DKI)

NAME

svDriverRegister, svDriverLookupFirst, svDriverLookupNext, svDriverRelease, svDriverEntry, svDriverCap, svDriverUnregister – driver registry operations

SYNOPSIS

```
#include <dki/dki.h>
KnError svDriverRegister(DrvRegEntry * drv_entry);

DrvRegId svDriverLookupFirst(void);

DrvRegId svDriverLookupNext(DrvRegId drv_id);

void svDriverRelease(DrvRegId drv_id);

DrvRegEntry * svDriverEntry(DrvRegId drv_id);

KnCap * svDriverCap(DrvRegId drv_id);

KnError svDriverUnregister(DrvRegId drv_id);
typedef struct {
  char* drv_name;
  char* drv_info;
  char* bus_class;
  int   bus_version;

    void
  (*drv_probe) (DevNode  bus_node,
          void*    bus_ops,
          void*    bus_id);

    void
  (*drv_bind) (DevNode  bus_node);

    void
  (*drv_init) (DevNode  dev_node,
          void*    bus_ops,
          void*    bus_id);

    KnError
  (*drv_unload) ();
} DrvRegEntry;
```

FEATURES

DKI

DESCRIPTION

The `driver registry` module implements a data base of drivers which have registered within the system. The `driver registry` data base is populated by drivers which perform self-registration (using `svDriverRegister`) at driver initialization time.

EXTENDED DESCRIPTION

`svDriverRegister` adds the driver entry to the driver registry. It returns `K_OK` in case of success, otherwise `K_ENOMEM` is returned to indicate that the system is out of memory. The *drv_entry* argument points to the `DrvRegEntry` structure specifying driver properties and static driver routines. The `DrvRegEntry` structure is described below.

drv_name	Points to a string specifying the driver name. For example, the driver file name.
drv_info	Points to a string specifying extra information about the driver component, such as version or author.
bus_class	Points to a string specifying the class of the parent driver API required for the driver, such as “pci”.
bus_version	Specifies the minimum version of the parent driver API required for the driver. Note that if a bus/nexus driver provides an API version which is less than *bus_version* , the corresponding driver component will never be called by this type of bus/nexus driver. In other words, the *drv_probe* and *drv_init* routines will never be invoked by this type of bus/nexus driver.
drv_probe	Points to a static driver routine which performs the device enumeration/probing on the bus.
drv_bind	Points to a static driver routine which performs the driver-to-device binding.
drv_init	Points to a static driver routine which clones an instance of the driver for the given device.
drv_unload	Points to a static driver routine which is invoked by the driver registry module when somebody wishes to unload the driver code from the system.

The *bus_class* and *bus_version* fields specify a parent bus/nexus API required for the driver. This type of API is provided to the driver as a structure of indirect

functions implementing the API service routines. For instance, the *"pci"* bus API is specified by the `PciBusOps` structure, the *"isa"* bus API is specified by `IsaBusOps` . When a bus driver invokes the `drv_probe` or `drv_init` driver's routine, it provides a pointer to the structure of the bus service routines. The structure type corresponds to the `bus_class` field value.

Note that a bus driver may provide multiple API's. A typical example is a PCI bus driver providing the common and PCI bus APIs. The common bus driver API is named by *"bus"* and specified by the `BusOps` structure. The PCI bus driver API is named by *"pci"* and specified by the `PciBusOps` structure. `BusOps` actually, provides a subset of services provided by `PciBusOps` . This type of bus driver is able to support drivers which use either common (*"bus"*) or PCI (*"pci"*) parent bus interfaces. When the bus driver invokes a child driver, it gives a pointer to either the `BusOps` or `PciBusOps` structure depending on the *bus_class* specified in the child driver registry entry.

Note also that a PROP_DRIVER may be adaptive to the parent bus API. In other words, such a driver is able to run on top of a number of different buses (for example, *"pci"* and *"isa"*). Typically, this type of driver is composed of two parts: bus class specific and bus class independent.

The bus class specific part of the driver code mainly deals with the device probing and initialization. In addition, it provides an abstraction layer in order to hide the bus class dependencies from the bus class independent part of the driver code. This kind of multi-bus driver should be registered multiple times in the driver registry. Each entry specifies a given bus class API (via the *bus_class* field) on top of which the driver may run. In order to determine to which bus class the driver is applied, the `drv_probe` , `drv_bind` and `drv_init` routines have to be entry specific (that is, bus class API specific). Under these conditions, when a given driver's routine is invoked by a bus driver, the driver detects the bus class to which it is applied and casts the *bus_ops* argument to the appropriate structure (for example, `PciBusOps`).

The `drv_probe` routine is invoked by a bus / nexus driver when the *bus_class* specified in the registry entry matches the bus / nexus driver class. `drv_probe` is called with three arguments.

The *dev_node* argument specifies the parent device node.

The *bus_ops* / *bus_id* pair specifies the parent device driver.

bus_ops points to a structure of service routines implementing a bus driver API. This structure is bus class specific and corresponds to the *bus_class* specified by the driver registry entry.

bus_id is opaque to the driver. It must be passed back to the bus driver when the `open` bus service routine is invoked. The `drv_probe` routine is optional. In cases where the `drv_probe` routine is not provided by the driver, the *drv_probe* field must be set to `NULL` .

The `drv_bind` routine is invoked by a bus / nexus driver when the *bus_class* specified in the registry entry matches the bus / nexus driver class.

`drv_bind` is called with one argument, *dev_node* , which specifies a given device node. The `drv_bind` routine enables the driver to perform a driver-to-device binding. Typically, the driver examines properties attached to the device node in order to determine the type of device at the node and to check whether the device may be serviced by the driver. If the check is positive, the driver binds the driver to the device node by attaching a PROP_DRIVER property to the device node. The property value specifies the driver name. The parent bus driver uses such a property to determine the name of the driver servicing the device. In other words, via the PROP_DRIVER property, the child driver gives its name to the parent bus driver, asking it to invoke the `drv_init` routine on that device. Note that, if a `"driver"` property is already present in the device node, then the `drv_bind` routine can not continue; `drv_bind` should not override existing driver-to-device binding.

The `drv_bind` routine is optional. If the `drv_bind` routine is not provided by the driver, the `drv_bind` field must be set to NULL.

The `drv_init` routine is invoked by a bus / nexus driver when the *bus_class* specified in the registry entry matches the bus / nexus driver class and a given driver is bound to a given device node.

`drv_init` is called with three arguments. The *dev_node* argument specifies the device node for which a device driver instance should be created. The *bus_ops* / *bus_id* pair specifies the parent device driver. *bus_ops* points to a structure of service routines implementing a bus driver API.

This structure is bus class specific and corresponds to the *bus_class* specified by the driver registry entry. *bus_id* is opaque to the driver. It must be passed back to the bus driver when the `open` bus service routine is invoked. The purpose of `drv_init` is to create an instance of the driver servicing the given device, to perform a device hardware initialization and to register the driver instance in the device registry .

Typically, `drv_init` would read the device and bus node properties in order to obtain the assigned bus resources and tunable parameters related to the bus / device. The `drv_init` routine is optional. In a case when the drv_init routine is not provided by the driver, the drv_init field must be set to NULL. A typical example of a probe-only driver is a self-identifying bus enumerator (for example, a PCI enumerator) which is implemented as driver. This type of

driver has the drv_probe routine which enumerates devices residing on the bus and creates device nodes. This type of driver obviously does not have the drv_init routine.

drv_unload is called by the driver registry module (more exactly by the svDriverUnregister routine) when somebody wishes to unload the driver code from the system. The purpose of drv_unload is to check that the device driver code is not currently being used. The driver must check, for each driver instance, whether it is locked in the device registry.

In case of success, all device instances are removed from the device registry and K_OK is returned. Otherwise, the device entries are unchanged in the device registry and K_EBUSY is returned.

The drv_unload routine is optional. In cases when drv_unload is not provided by the driver, the drv_unload field must be set to NULL . Note that, in this case, the driver code cannot be unloaded.

svDriverLookupFirst returns the first driver entry in the registry. When the registry is not empty a non zero DrvRegId (designating the entry) is returned, otherwise NULL is returned. In case of success, the driver entry is locked in the registry. It should be unlocked by a subsequent invocation of svDriverRelease or svDriverUnregister .

The driver entry lock is not exclusive. It is simply a counter which prevents the driver component from being unloaded when it is still being used by a bus / nexus driver for probing or initialization.

svDriverLookupNext returns the next driver entry in the registry. The current entry is specified by the *drv_id* argument. The current entry must be locked by a previously called svDriverLookupFirst or svDriverLookupNext . If the current entry is not the last one in the registry a non zero DrvRegId (designating the entry) is returned, otherwise NULL is returned. In case of success, the next driver entry is locked in the registry. It should be unlocked by a subsequent invocation of svDriverRelease or svDriverUnregister .

svDriverRelease releases the lock of the driver entry specified by the *drv_id* argument.

svDriverEntry returns a pointer to the driver entry structure specified by the *drv_id* argument. The device entry being accessed must have been previously locked using svDriverLookupFirst or svDriverLookupNext . Note that the device entry structure is read-only.

`svDriverCap` returns a pointer to the driver actor capability. The driver entry is specified by the *drv_id* argument. The driver entry being accessed must have been previously locked using `svDriverLookupFirst` or `svDriverLookupNext` . Note that if a given driver entry is registered by a built-in driver, a `NULL` pointer is returned. In other words, the driver actor capability makes sense only for dynamically loaded drivers. The actor capability may be used by an application in order to delete the driver actor once the driver entry is unregistered. Note that the driver capability structure is read-only.

`svDriverUnregister` tries to remove the driver entry specified by the *dev_id* argument from the driver registry. The device entry being removed must have been previously locked using `svDriverLookupFirst` or `svDriverLookupNext` .

In case of success, `K_OK` is returned, otherwise `K_EBUSY` is returned. The `K_EBUSY` result means that either the driver entry is locked in the driver registry (that is, a static driver routine is currently being used by a bus/nexus driver) or an instance of the device driver is locked in the device registry (that is, there is a driver instance which is currently being used by a driver client). Note that when `K_EBUSY` is returned, the driver entry remains locked in the registry and should be unlocked explicitly by `svDriverRelease` .

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service.

Services	Base level	DKI thread	Interrupt	Blocking
svDriverRegister	+	+	-	+
svDriverLookupFirst	+	+	-	+
svDriverLookupNext	+	+	-	+
svDriverRelease	+	+	-	+
svDriverEntry	+	+	-	-
svDriverCap	+	+	-	-
svDriverUnregister	+	+	-	+

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`dtreeNodeRoot`(9DKI) , `svDkiThreadCall`(9DKI)

NAME

svDriverRegister, svDriverLookupFirst, svDriverLookupNext, svDriverRelease, svDriverEntry, svDriverCap, svDriverUnregister – driver registry operations

SYNOPSIS

```
#include <dki/dki.h>
KnError svDriverRegister(DrvRegEntry * drv_entry);

DrvRegId svDriverLookupFirst(void);

DrvRegId svDriverLookupNext(DrvRegId drv_id);

void svDriverRelease(DrvRegId drv_id);

DrvRegEntry * svDriverEntry(DrvRegId drv_id);

KnCap * svDriverCap(DrvRegId drv_id);

KnError svDriverUnregister(DrvRegId drv_id);
typedef struct {
  char* drv_name;
  char* drv_info;
  char* bus_class;
  int   bus_version;

    void
  (*drv_probe) (DevNode  bus_node,
          void*    bus_ops,
          void*    bus_id);

    void
  (*drv_bind) (DevNode  bus_node);

    void
  (*drv_init) (DevNode  dev_node,
          void*    bus_ops,
          void*    bus_id);

    KnError
  (*drv_unload) ();
} DrvRegEntry;
```

FEATURES

DKI

DESCRIPTION

The `driver registry` module implements a data base of drivers which have registered within the system. The `driver registry` data base is populated by drivers which perform self-registration (using `svDriverRegister`) at driver initialization time.

EXTENDED DESCRIPTION

`svDriverRegister` adds the driver entry to the driver registry. It returns `K_OK` in case of success, otherwise `K_ENOMEM` is returned to indicate that the system is out of memory. The *drv_entry* argument points to the `DrvRegEntry` structure specifying driver properties and static driver routines. The `DrvRegEntry` structure is described below.

drv_name	Points to a string specifying the driver name. For example, the driver file name.
drv_info	Points to a string specifying extra information about the driver component, such as version or author.
bus_class	Points to a string specifying the class of the parent driver API required for the driver, such as “pci”.
bus_version	Specifies the minimum version of the parent driver API required for the driver. Note that if a bus / nexus driver provides an API version which is less than *bus_version* , the corresponding driver component will never be called by this type of bus / nexus driver. In other words, the *drv_probe* and *drv_init* routines will never be invoked by this type of bus / nexus driver.
drv_probe	Points to a static driver routine which performs the device enumeration / probing on the bus.
drv_bind	Points to a static driver routine which performs the driver-to-device binding.
drv_init	Points to a static driver routine which clones an instance of the driver for the given device.
drv_unload	Points to a static driver routine which is invoked by the driver registry module when somebody wishes to unload the driver code from the system.

The *bus_class* and *bus_version* fields specify a parent bus / nexus API required for the driver. This type of API is provided to the driver as a structure of indirect

functions implementing the API service routines. For instance, the *"pci"* bus API is specified by the `PciBusOps` structure, the *"isa"* bus API is specified by `IsaBusOps` . When a bus driver invokes the `drv_probe` or `drv_init` driver's routine, it provides a pointer to the structure of the bus service routines. The structure type corresponds to the `bus_class` field value.

Note that a bus driver may provide multiple API's. A typical example is a PCI bus driver providing the common and PCI bus APIs. The common bus driver API is named by *"bus"* and specified by the `BusOps` structure. The PCI bus driver API is named by *"pci"* and specified by the `PciBusOps` structure. `BusOps` actually, provides a subset of services provided by `PciBusOps` . This type of bus driver is able to support drivers which use either common (*"bus"*) or PCI (*"pci"*) parent bus interfaces. When the bus driver invokes a child driver, it gives a pointer to either the `BusOps` or `PciBusOps` structure depending on the *bus_class* specified in the child driver registry entry.

Note also that a PROP_DRIVER may be adaptive to the parent bus API. In other words, such a driver is able to run on top of a number of different buses (for example, *"pci"* and *"isa"*). Typically, this type of driver is composed of two parts: bus class specific and bus class independent.

The bus class specific part of the driver code mainly deals with the device probing and initialization. In addition, it provides an abstraction layer in order to hide the bus class dependencies from the bus class independent part of the driver code. This kind of multi-bus driver should be registered multiple times in the driver registry. Each entry specifies a given bus class API (via the *bus_class* field) on top of which the driver may run. In order to determine to which bus class the driver is applied, the `drv_probe` , `drv_bind` and `drv_init` routines have to be entry specific (that is, bus class API specific). Under these conditions, when a given driver's routine is invoked by a bus driver, the driver detects the bus class to which it is applied and casts the *bus_ops* argument to the appropriate structure (for example, `PciBusOps`).

The `drv_probe` routine is invoked by a bus / nexus driver when the *bus_class* specified in the registry entry matches the bus / nexus driver class. `drv_probe` is called with three arguments.

The *dev_node* argument specifies the parent device node.

The *bus_ops* / *bus_id* pair specifies the parent device driver.

bus_ops points to a structure of service routines implementing a bus driver API. This structure is bus class specific and corresponds to the *bus_class* specified by the driver registry entry.

bus_id is opaque to the driver. It must be passed back to the bus driver when the `open` bus service routine is invoked. The `drv_probe` routine is optional. In cases where the `drv_probe` routine is not provided by the driver, the *drv_probe* field must be set to `NULL` .

The `drv_bind` routine is invoked by a bus / nexus driver when the *bus_class* specified in the registry entry matches the bus / nexus driver class.

`drv_bind` is called with one argument, *dev_node* , which specifies a given device node. The `drv_bind` routine enables the driver to perform a driver-to-device binding. Typically, the driver examines properties attached to the device node in order to determine the type of device at the node and to check whether the device may be serviced by the driver. If the check is positive, the driver binds the driver to the device node by attaching a PROP_DRIVER property to the device node. The property value specifies the driver name. The parent bus driver uses such a property to determine the name of the driver servicing the device. In other words, via the PROP_DRIVER property, the child driver gives its name to the parent bus driver, asking it to invoke the `drv_init` routine on that device. Note that, if a `"driver"` property is already present in the device node, then the `drv_bind` routine can not continue; `drv_bind` should not override existing driver-to-device binding.

The `drv_bind` routine is optional. If the `drv_bind` routine is not provided by the driver, the `drv_bind` field must be set to NULL.

The `drv_init` routine is invoked by a bus / nexus driver when the *bus_class* specified in the registry entry matches the bus / nexus driver class and a given driver is bound to a given device node.

`drv_init` is called with three arguments. The *dev_node* argument specifies the device node for which a device driver instance should be created. The *bus_ops* / *bus_id* pair specifies the parent device driver. *bus_ops* points to a structure of service routines implementing a bus driver API.

This structure is bus class specific and corresponds to the *bus_class* specified by the driver registry entry. *bus_id* is opaque to the driver. It must be passed back to the bus driver when the `open` bus service routine is invoked. The purpose of `drv_init` is to create an instance of the driver servicing the given device, to perform a device hardware initialization and to register the driver instance in the device registry .

Typically, `drv_init` would read the device and bus node properties in order to obtain the assigned bus resources and tunable parameters related to the bus / device. The `drv_init` routine is optional. In a case when the drv_init routine is not provided by the driver, the drv_init field must be set to NULL. A typical example of a probe-only driver is a self-identifying bus enumerator (for example, a PCI enumerator) which is implemented as driver. This type of

driver has the drv_probe routine which enumerates devices residing on the bus and creates device nodes. This type of driver obviously does not have the drv_init routine.

drv_unload is called by the driver registry module (more exactly by the svDriverUnregister routine) when somebody wishes to unload the driver code from the system. The purpose of drv_unload is to check that the device driver code is not currently being used. The driver must check, for each driver instance, whether it is locked in the device registry.

In case of success, all device instances are removed from the device registry and K_OK is returned. Otherwise, the device entries are unchanged in the device registry and K_EBUSY is returned.

The drv_unload routine is optional. In cases when drv_unload is not provided by the driver, the drv_unload field must be set to NULL . Note that, in this case, the driver code cannot be unloaded.

svDriverLookupFirst returns the first driver entry in the registry. When the registry is not empty a non zero DrvRegId (designating the entry) is returned, otherwise NULL is returned. In case of success, the driver entry is locked in the registry. It should be unlocked by a subsequent invocation of svDriverRelease or svDriverUnregister .

The driver entry lock is not exclusive. It is simply a counter which prevents the driver component from being unloaded when it is still being used by a bus / nexus driver for probing or initialization.

svDriverLookupNext returns the next driver entry in the registry. The current entry is specified by the *drv_id* argument. The current entry must be locked by a previously called svDriverLookupFirst or svDriverLookupNext . If the current entry is not the last one in the registry a non zero DrvRegId (designating the entry) is returned, otherwise NULL is returned. In case of success, the next driver entry is locked in the registry. It should be unlocked by a subsequent invocation of svDriverRelease or svDriverUnregister .

svDriverRelease releases the lock of the driver entry specified by the *drv_id* argument.

svDriverEntry returns a pointer to the driver entry structure specified by the *drv_id* argument. The device entry being accessed must have been previously locked using svDriverLookupFirst or svDriverLookupNext . Note that the device entry structure is read-only.

`svDriverCap` returns a pointer to the driver actor capability. The driver entry is specified by the *drv_id* argument. The driver entry being accessed must have been previously locked using `svDriverLookupFirst` or `svDriverLookupNext` . Note that if a given driver entry is registered by a built-in driver, a `NULL` pointer is returned. In other words, the driver actor capability makes sense only for dynamically loaded drivers. The actor capability may be used by an application in order to delete the driver actor once the driver entry is unregistered. Note that the driver capability structure is read-only.

`svDriverUnregister` tries to remove the driver entry specified by the *dev_id* argument from the driver registry. The device entry being removed must have been previously locked using `svDriverLookupFirst` or `svDriverLookupNext` .

In case of success, `K_OK` is returned, otherwise `K_EBUSY` is returned. The `K_EBUSY` result means that either the driver entry is locked in the driver registry (that is, a static driver routine is currently being used by a bus/nexus driver) or an instance of the device driver is locked in the device registry (that is, there is a driver instance which is currently being used by a driver client). Note that when `K_EBUSY` is returned, the driver entry remains locked in the registry and should be unlocked explicitly by `svDriverRelease` .

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service.

Services	Base level	DKI thread	Interrupt	Blocking
svDriverRegister	+	+	-	+
svDriverLookupFirst	+	+	-	+
svDriverLookupNext	+	+	-	+
svDriverRelease	+	+	-	+
svDriverEntry	+	+	-	-
svDriverCap	+	+	-	-
svDriverUnregister	+	+	-	+

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`dtreeNodeRoot`(9DKI) , `svDkiThreadCall`(9DKI)

NAME

svDriverRegister, svDriverLookupFirst, svDriverLookupNext, svDriverRelease, svDriverEntry, svDriverCap, svDriverUnregister – driver registry operations

SYNOPSIS

```
#include <dki/dki.h>
KnError svDriverRegister(DrvRegEntry * drv_entry);

DrvRegId svDriverLookupFirst(void);

DrvRegId svDriverLookupNext(DrvRegId drv_id);

void svDriverRelease(DrvRegId drv_id);

DrvRegEntry * svDriverEntry(DrvRegId drv_id);

KnCap * svDriverCap(DrvRegId drv_id);

KnError svDriverUnregister(DrvRegId drv_id);
typedef struct {
  char* drv_name;
  char* drv_info;
  char* bus_class;
  int   bus_version;

    void
  (*drv_probe) (DevNode  bus_node,
          void*    bus_ops,
          void*    bus_id);

    void
  (*drv_bind) (DevNode  bus_node);

    void
  (*drv_init) (DevNode  dev_node,
          void*    bus_ops,
          void*    bus_id);

    KnError
  (*drv_unload) ();
} DrvRegEntry;
```

FEATURES

DKI

DESCRIPTION

The `driver registry` module implements a data base of drivers which have registered within the system. The `driver registry` data base is populated by drivers which perform self-registration (using `svDriverRegister`) at driver initialization time.

EXTENDED DESCRIPTION

`svDriverRegister` adds the driver entry to the driver registry. It returns `K_OK` in case of success, otherwise `K_ENOMEM` is returned to indicate that the system is out of memory. The *drv_entry* argument points to the `DrvRegEntry` structure specifying driver properties and static driver routines. The `DrvRegEntry` structure is described below.

drv_name	Points to a string specifying the driver name. For example, the driver file name.
drv_info	Points to a string specifying extra information about the driver component, such as version or author.
bus_class	Points to a string specifying the class of the parent driver API required for the driver, such as “pci”.
bus_version	Specifies the minimum version of the parent driver API required for the driver. Note that if a bus/nexus driver provides an API version which is less than *bus_version* , the corresponding driver component will never be called by this type of bus/nexus driver. In other words, the *drv_probe* and *drv_init* routines will never be invoked by this type of bus/nexus driver.
drv_probe	Points to a static driver routine which performs the device enumeration/probing on the bus.
drv_bind	Points to a static driver routine which performs the driver-to-device binding.
drv_init	Points to a static driver routine which clones an instance of the driver for the given device.
drv_unload	Points to a static driver routine which is invoked by the driver registry module when somebody wishes to unload the driver code from the system.

The *bus_class* and *bus_version* fields specify a parent bus/nexus API required for the driver. This type of API is provided to the driver as a structure of indirect

functions implementing the API service routines. For instance, the *"pci"* bus API is specified by the `PciBusOps` structure, the *"isa"* bus API is specified by `IsaBusOps` . When a bus driver invokes the `drv_probe` or `drv_init` driver's routine, it provides a pointer to the structure of the bus service routines. The structure type corresponds to the `bus_class` field value.

Note that a bus driver may provide multiple API's. A typical example is a PCI bus driver providing the common and PCI bus APIs. The common bus driver API is named by *"bus"* and specified by the `BusOps` structure. The PCI bus driver API is named by *"pci"* and specified by the `PciBusOps` structure. `BusOps` actually, provides a subset of services provided by `PciBusOps` . This type of bus driver is able to support drivers which use either common (*"bus"*) or PCI (*"pci"*) parent bus interfaces. When the bus driver invokes a child driver, it gives a pointer to either the `BusOps` or `PciBusOps` structure depending on the *bus_class* specified in the child driver registry entry.

Note also that a PROP_DRIVER may be adaptive to the parent bus API. In other words, such a driver is able to run on top of a number of different buses (for example, *"pci"* and *"isa"*). Typically, this type of driver is composed of two parts: bus class specific and bus class independent.

The bus class specific part of the driver code mainly deals with the device probing and initialization. In addition, it provides an abstraction layer in order to hide the bus class dependencies from the bus class independent part of the driver code. This kind of multi-bus driver should be registered multiple times in the driver registry. Each entry specifies a given bus class API (via the *bus_class* field) on top of which the driver may run. In order to determine to which bus class the driver is applied, the `drv_probe` , `drv_bind` and `drv_init` routines have to be entry specific (that is, bus class API specific). Under these conditions, when a given driver's routine is invoked by a bus driver, the driver detects the bus class to which it is applied and casts the *bus_ops* argument to the appropriate structure (for example, `PciBusOps`).

The `drv_probe` routine is invoked by a bus / nexus driver when the *bus_class* specified in the registry entry matches the bus / nexus driver class. `drv_probe` is called with three arguments.

The *dev_node* argument specifies the parent device node.

The *bus_ops* / *bus_id* pair specifies the parent device driver.

bus_ops points to a structure of service routines implementing a bus driver API. This structure is bus class specific and corresponds to the *bus_class* specified by the driver registry entry.

bus_id is opaque to the driver. It must be passed back to the bus driver when the `open` bus service routine is invoked. The `drv_probe` routine is optional. In cases where the `drv_probe` routine is not provided by the driver, the *drv_probe* field must be set to `NULL` .

The `drv_bind` routine is invoked by a bus / nexus driver when the *bus_class* specified in the registry entry matches the bus / nexus driver class.

`drv_bind` is called with one argument, *dev_node* , which specifies a given device node. The `drv_bind` routine enables the driver to perform a driver-to-device binding. Typically, the driver examines properties attached to the device node in order to determine the type of device at the node and to check whether the device may be serviced by the driver. If the check is positive, the driver binds the driver to the device node by attaching a PROP_DRIVER property to the device node. The property value specifies the driver name. The parent bus driver uses such a property to determine the name of the driver servicing the device. In other words, via the PROP_DRIVER property, the child driver gives its name to the parent bus driver, asking it to invoke the `drv_init` routine on that device. Note that, if a `"driver"` property is already present in the device node, then the `drv_bind` routine can not continue; `drv_bind` should not override existing driver-to-device binding.

The `drv_bind` routine is optional. If the `drv_bind` routine is not provided by the driver, the `drv_bind` field must be set to NULL.

The `drv_init` routine is invoked by a bus / nexus driver when the *bus_class* specified in the registry entry matches the bus / nexus driver class and a given driver is bound to a given device node.

`drv_init` is called with three arguments. The *dev_node* argument specifies the device node for which a device driver instance should be created. The *bus_ops* / *bus_id* pair specifies the parent device driver. *bus_ops* points to a structure of service routines implementing a bus driver API.

This structure is bus class specific and corresponds to the *bus_class* specified by the driver registry entry. *bus_id* is opaque to the driver. It must be passed back to the bus driver when the `open` bus service routine is invoked. The purpose of `drv_init` is to create an instance of the driver servicing the given device, to perform a device hardware initialization and to register the driver instance in the device registry .

Typically, `drv_init` would read the device and bus node properties in order to obtain the assigned bus resources and tunable parameters related to the bus / device. The `drv_init` routine is optional. In a case when the drv_init routine is not provided by the driver, the drv_init field must be set to NULL. A typical example of a probe-only driver is a self-identifying bus enumerator (for example, a PCI enumerator) which is implemented as driver. This type of

driver has the drv_probe routine which enumerates devices residing on the bus and creates device nodes. This type of driver obviously does not have the drv_init routine.

drv_unload is called by the driver registry module (more exactly by the svDriverUnregister routine) when somebody wishes to unload the driver code from the system. The purpose of drv_unload is to check that the device driver code is not currently being used. The driver must check, for each driver instance, whether it is locked in the device registry.

In case of success, all device instances are removed from the device registry and K_OK is returned. Otherwise, the device entries are unchanged in the device registry and K_EBUSY is returned.

The drv_unload routine is optional. In cases when drv_unload is not provided by the driver, the drv_unload field must be set to NULL . Note that, in this case, the driver code cannot be unloaded.

svDriverLookupFirst returns the first driver entry in the registry. When the registry is not empty a non zero DrvRegId (designating the entry) is returned, otherwise NULL is returned. In case of success, the driver entry is locked in the registry. It should be unlocked by a subsequent invocation of svDriverRelease or svDriverUnregister .

The driver entry lock is not exclusive. It is simply a counter which prevents the driver component from being unloaded when it is still being used by a bus / nexus driver for probing or initialization.

svDriverLookupNext returns the next driver entry in the registry. The current entry is specified by the *drv_id* argument. The current entry must be locked by a previously called svDriverLookupFirst or svDriverLookupNext . If the current entry is not the last one in the registry a non zero DrvRegId (designating the entry) is returned, otherwise NULL is returned. In case of success, the next driver entry is locked in the registry. It should be unlocked by a subsequent invocation of svDriverRelease or svDriverUnregister .

svDriverRelease releases the lock of the driver entry specified by the *drv_id* argument.

svDriverEntry returns a pointer to the driver entry structure specified by the *drv_id* argument. The device entry being accessed must have been previously locked using svDriverLookupFirst or svDriverLookupNext . Note that the device entry structure is read-only.

`svDriverCap` returns a pointer to the driver actor capability. The driver entry is specified by the *drv_id* argument. The driver entry being accessed must have been previously locked using `svDriverLookupFirst` or `svDriverLookupNext` . Note that if a given driver entry is registered by a built-in driver, a `NULL` pointer is returned. In other words, the driver actor capability makes sense only for dynamically loaded drivers. The actor capability may be used by an application in order to delete the driver actor once the driver entry is unregistered. Note that the driver capability structure is read-only.

`svDriverUnregister` tries to remove the driver entry specified by the *dev_id* argument from the driver registry. The device entry being removed must have been previously locked using `svDriverLookupFirst` or `svDriverLookupNext` .

In case of success, `K_OK` is returned, otherwise `K_EBUSY` is returned. The `K_EBUSY` result means that either the driver entry is locked in the driver registry (that is, a static driver routine is currently being used by a bus/nexus driver) or an instance of the device driver is locked in the device registry (that is, there is a driver instance which is currently being used by a driver client). Note that when `K_EBUSY` is returned, the driver entry remains locked in the registry and should be unlocked explicitly by `svDriverRelease` .

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service.

Services	Base level	DKI thread	Interrupt	Blocking
svDriverRegister	+	+	-	+
svDriverLookupFirst	+	+	-	+
svDriverLookupNext	+	+	-	+
svDriverRelease	+	+	-	+
svDriverEntry	+	+	-	-
svDriverCap	+	+	-	-
svDriverUnregister	+	+	-	+

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`dtreeNodeRoot`(9DKI) , `svDkiThreadCall`(9DKI)

NAME

svDriverRegister, svDriverLookupFirst, svDriverLookupNext, svDriverRelease, svDriverEntry, svDriverCap, svDriverUnregister – driver registry operations

SYNOPSIS

```
#include <dki/dki.h>
KnError svDriverRegister(DrvRegEntry * drv_entry);

DrvRegId svDriverLookupFirst(void);

DrvRegId svDriverLookupNext(DrvRegId drv_id);

void svDriverRelease(DrvRegId drv_id);

DrvRegEntry * svDriverEntry(DrvRegId drv_id);

KnCap * svDriverCap(DrvRegId drv_id);

KnError svDriverUnregister(DrvRegId drv_id);
typedef struct {
  char* drv_name;
  char* drv_info;
  char* bus_class;
  int   bus_version;

    void
  (*drv_probe) (DevNode  bus_node,
          void*    bus_ops,
          void*    bus_id);

    void
  (*drv_bind) (DevNode  bus_node);

    void
  (*drv_init) (DevNode  dev_node,
          void*    bus_ops,
          void*    bus_id);

    KnError
  (*drv_unload) ();
} DrvRegEntry;
```

FEATURES

DKI

DESCRIPTION

The `driver registry` module implements a data base of drivers which have registered within the system. The `driver registry` data base is populated by drivers which perform self-registration (using `svDriverRegister`) at driver initialization time.

EXTENDED DESCRIPTION

`svDriverRegister` adds the driver entry to the driver registry. It returns `K_OK` in case of success, otherwise `K_ENOMEM` is returned to indicate that the system is out of memory. The *drv_entry* argument points to the `DrvRegEntry` structure specifying driver properties and static driver routines. The `DrvRegEntry` structure is described below.

drv_name	Points to a string specifying the driver name. For example, the driver file name.
drv_info	Points to a string specifying extra information about the driver component, such as version or author.
bus_class	Points to a string specifying the class of the parent driver API required for the driver, such as “pci”.
bus_version	Specifies the minimum version of the parent driver API required for the driver. Note that if a bus/nexus driver provides an API version which is less than *bus_version* , the corresponding driver component will never be called by this type of bus/nexus driver. In other words, the *drv_probe* and *drv_init* routines will never be invoked by this type of bus/nexus driver.
drv_probe	Points to a static driver routine which performs the device enumeration/probing on the bus.
drv_bind	Points to a static driver routine which performs the driver-to-device binding.
drv_init	Points to a static driver routine which clones an instance of the driver for the given device.
drv_unload	Points to a static driver routine which is invoked by the driver registry module when somebody wishes to unload the driver code from the system.

The *bus_class* and *bus_version* fields specify a parent bus/nexus API required for the driver. This type of API is provided to the driver as a structure of indirect

functions implementing the API service routines. For instance, the *"pci"* bus API is specified by the `PciBusOps` structure, the *"isa"* bus API is specified by `IsaBusOps` . When a bus driver invokes the `drv_probe` or `drv_init` driver's routine, it provides a pointer to the structure of the bus service routines. The structure type corresponds to the `bus_class` field value.

Note that a bus driver may provide multiple API's. A typical example is a PCI bus driver providing the common and PCI bus APIs. The common bus driver API is named by *"bus"* and specified by the `BusOps` structure. The PCI bus driver API is named by *"pci"* and specified by the `PciBusOps` structure. `BusOps` actually, provides a subset of services provided by `PciBusOps` . This type of bus driver is able to support drivers which use either common (*"bus"*) or PCI (*"pci"*) parent bus interfaces. When the bus driver invokes a child driver, it gives a pointer to either the `BusOps` or `PciBusOps` structure depending on the *bus_class* specified in the child driver registry entry.

Note also that a PROP_DRIVER may be adaptive to the parent bus API. In other words, such a driver is able to run on top of a number of different buses (for example, *"pci"* and *"isa"*). Typically, this type of driver is composed of two parts: bus class specific and bus class independent.

The bus class specific part of the driver code mainly deals with the device probing and initialization. In addition, it provides an abstraction layer in order to hide the bus class dependencies from the bus class independent part of the driver code. This kind of multi-bus driver should be registered multiple times in the driver registry. Each entry specifies a given bus class API (via the *bus_class* field) on top of which the driver may run. In order to determine to which bus class the driver is applied, the `drv_probe` , `drv_bind` and `drv_init` routines have to be entry specific (that is, bus class API specific). Under these conditions, when a given driver's routine is invoked by a bus driver, the driver detects the bus class to which it is applied and casts the *bus_ops* argument to the appropriate structure (for example, `PciBusOps`).

The `drv_probe` routine is invoked by a bus / nexus driver when the *bus_class* specified in the registry entry matches the bus / nexus driver class. `drv_probe` is called with three arguments.

The *dev_node* argument specifies the parent device node.

The *bus_ops* / *bus_id* pair specifies the parent device driver.

bus_ops points to a structure of service routines implementing a bus driver API. This structure is bus class specific and corresponds to the *bus_class* specified by the driver registry entry.

bus_id is opaque to the driver. It must be passed back to the bus driver when the `open` bus service routine is invoked. The `drv_probe` routine is optional. In cases where the `drv_probe` routine is not provided by the driver, the *drv_probe* field must be set to `NULL` .

The `drv_bind` routine is invoked by a bus / nexus driver when the *bus_class* specified in the registry entry matches the bus / nexus driver class.

`drv_bind` is called with one argument, *dev_node* , which specifies a given device node. The `drv_bind` routine enables the driver to perform a driver-to-device binding. Typically, the driver examines properties attached to the device node in order to determine the type of device at the node and to check whether the device may be serviced by the driver. If the check is positive, the driver binds the driver to the device node by attaching a PROP_DRIVER property to the device node. The property value specifies the driver name. The parent bus driver uses such a property to determine the name of the driver servicing the device. In other words, via the PROP_DRIVER property, the child driver gives its name to the parent bus driver, asking it to invoke the `drv_init` routine on that device. Note that, if a `"driver"` property is already present in the device node, then the `drv_bind` routine can not continue; `drv_bind` should not override existing driver-to-device binding.

The `drv_bind` routine is optional. If the `drv_bind` routine is not provided by the driver, the `drv_bind` field must be set to NULL.

The `drv_init` routine is invoked by a bus / nexus driver when the *bus_class* specified in the registry entry matches the bus / nexus driver class and a given driver is bound to a given device node.

`drv_init` is called with three arguments. The *dev_node* argument specifies the device node for which a device driver instance should be created. The *bus_ops* / *bus_id* pair specifies the parent device driver. *bus_ops* points to a structure of service routines implementing a bus driver API.

This structure is bus class specific and corresponds to the *bus_class* specified by the driver registry entry. *bus_id* is opaque to the driver. It must be passed back to the bus driver when the `open` bus service routine is invoked. The purpose of `drv_init` is to create an instance of the driver servicing the given device, to perform a device hardware initialization and to register the driver instance in the device registry .

Typically, `drv_init` would read the device and bus node properties in order to obtain the assigned bus resources and tunable parameters related to the bus / device. The `drv_init` routine is optional. In a case when the drv_init routine is not provided by the driver, the drv_init field must be set to NULL. A typical example of a probe-only driver is a self-identifying bus enumerator (for example, a PCI enumerator) which is implemented as driver. This type of

driver has the drv_probe routine which enumerates devices residing on the bus and creates device nodes. This type of driver obviously does not have the drv_init routine.

drv_unload is called by the driver registry module (more exactly by the svDriverUnregister routine) when somebody wishes to unload the driver code from the system. The purpose of drv_unload is to check that the device driver code is not currently being used. The driver must check, for each driver instance, whether it is locked in the device registry.

In case of success, all device instances are removed from the device registry and K_OK is returned. Otherwise, the device entries are unchanged in the device registry and K_EBUSY is returned.

The drv_unload routine is optional. In cases when drv_unload is not provided by the driver, the drv_unload field must be set to NULL . Note that, in this case, the driver code cannot be unloaded.

svDriverLookupFirst returns the first driver entry in the registry. When the registry is not empty a non zero DrvRegId (designating the entry) is returned, otherwise NULL is returned. In case of success, the driver entry is locked in the registry. It should be unlocked by a subsequent invocation of svDriverRelease or svDriverUnregister .

The driver entry lock is not exclusive. It is simply a counter which prevents the driver component from being unloaded when it is still being used by a bus / nexus driver for probing or initialization.

svDriverLookupNext returns the next driver entry in the registry. The current entry is specified by the *drv_id* argument. The current entry must be locked by a previously called svDriverLookupFirst or svDriverLookupNext . If the current entry is not the last one in the registry a non zero DrvRegId (designating the entry) is returned, otherwise NULL is returned. In case of success, the next driver entry is locked in the registry. It should be unlocked by a subsequent invocation of svDriverRelease or svDriverUnregister .

svDriverRelease releases the lock of the driver entry specified by the *drv_id* argument.

svDriverEntry returns a pointer to the driver entry structure specified by the *drv_id* argument. The device entry being accessed must have been previously locked using svDriverLookupFirst or svDriverLookupNext . Note that the device entry structure is read-only.

`svDriverCap` returns a pointer to the driver actor capability. The driver entry is specified by the *drv_id* argument. The driver entry being accessed must have been previously locked using `svDriverLookupFirst` or `svDriverLookupNext` . Note that if a given driver entry is registered by a built-in driver, a `NULL` pointer is returned. In other words, the driver actor capability makes sense only for dynamically loaded drivers. The actor capability may be used by an application in order to delete the driver actor once the driver entry is unregistered. Note that the driver capability structure is read-only.

`svDriverUnregister` tries to remove the driver entry specified by the *dev_id* argument from the driver registry. The device entry being removed must have been previously locked using `svDriverLookupFirst` or `svDriverLookupNext` .

In case of success, `K_OK` is returned, otherwise `K_EBUSY` is returned. The `K_EBUSY` result means that either the driver entry is locked in the driver registry (that is, a static driver routine is currently being used by a bus/nexus driver) or an instance of the device driver is locked in the device registry (that is, there is a driver instance which is currently being used by a driver client). Note that when `K_EBUSY` is returned, the driver entry remains locked in the registry and should be unlocked explicitly by `svDriverRelease` .

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service.

Services	Base level	DKI thread	Interrupt	Blocking
svDriverRegister	+	+	-	+
svDriverLookupFirst	+	+	-	+
svDriverLookupNext	+	+	-	+
svDriverRelease	+	+	-	+
svDriverEntry	+	+	-	-
svDriverCap	+	+	-	-
svDriverUnregister	+	+	-	+

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`dtreeNodeRoot`(9DKI) , `svDkiThreadCall`(9DKI)

NAME

svDriverRegister, svDriverLookupFirst, svDriverLookupNext, svDriverRelease, svDriverEntry, svDriverCap, svDriverUnregister – driver registry operations

SYNOPSIS

```
#include <dki/dki.h>
KnError svDriverRegister(DrvRegEntry * drv_entry);

DrvRegId svDriverLookupFirst(void);

DrvRegId svDriverLookupNext(DrvRegId drv_id);

void svDriverRelease(DrvRegId drv_id);

DrvRegEntry * svDriverEntry(DrvRegId drv_id);

KnCap * svDriverCap(DrvRegId drv_id);

KnError svDriverUnregister(DrvRegId drv_id);
typedef struct {
  char* drv_name;
  char* drv_info;
  char* bus_class;
  int   bus_version;

    void
  (*drv_probe) (DevNode  bus_node,
          void*    bus_ops,
          void*    bus_id);

    void
  (*drv_bind) (DevNode  bus_node);

    void
  (*drv_init) (DevNode  dev_node,
          void*    bus_ops,
          void*    bus_id);

    KnError
  (*drv_unload) ();
} DrvRegEntry;
```

FEATURES

DKI

DESCRIPTION

The `driver registry` module implements a data base of drivers which have registered within the system. The `driver registry` data base is populated by drivers which perform self-registration (using `svDriverRegister`) at driver initialization time.

EXTENDED DESCRIPTION

`svDriverRegister` adds the driver entry to the driver registry. It returns `K_OK` in case of success, otherwise `K_ENOMEM` is returned to indicate that the system is out of memory. The *drv_entry* argument points to the `DrvRegEntry` structure specifying driver properties and static driver routines. The `DrvRegEntry` structure is described below.

drv_name	Points to a string specifying the driver name. For example, the driver file name.
drv_info	Points to a string specifying extra information about the driver component, such as version or author.
bus_class	Points to a string specifying the class of the parent driver API required for the driver, such as “pci”.
bus_version	Specifies the minimum version of the parent driver API required for the driver. Note that if a bus/nexus driver provides an API version which is less than *bus_version* , the corresponding driver component will never be called by this type of bus/nexus driver. In other words, the *drv_probe* and *drv_init* routines will never be invoked by this type of bus/nexus driver.
drv_probe	Points to a static driver routine which performs the device enumeration/probing on the bus.
drv_bind	Points to a static driver routine which performs the driver-to-device binding.
drv_init	Points to a static driver routine which clones an instance of the driver for the given device.
drv_unload	Points to a static driver routine which is invoked by the driver registry module when somebody wishes to unload the driver code from the system.

The *bus_class* and *bus_version* fields specify a parent bus/nexus API required for the driver. This type of API is provided to the driver as a structure of indirect

functions implementing the API service routines. For instance, the *"pci"* bus API is specified by the `PciBusOps` structure, the *"isa"* bus API is specified by `IsaBusOps` . When a bus driver invokes the `drv_probe` or `drv_init` driver's routine, it provides a pointer to the structure of the bus service routines. The structure type corresponds to the `bus_class` field value.

Note that a bus driver may provide multiple API's. A typical example is a PCI bus driver providing the common and PCI bus APIs. The common bus driver API is named by *"bus"* and specified by the `BusOps` structure. The PCI bus driver API is named by *"pci"* and specified by the `PciBusOps` structure. `BusOps` actually, provides a subset of services provided by `PciBusOps` . This type of bus driver is able to support drivers which use either common (*"bus"*) or PCI (*"pci"*) parent bus interfaces. When the bus driver invokes a child driver, it gives a pointer to either the `BusOps` or `PciBusOps` structure depending on the *bus_class* specified in the child driver registry entry.

Note also that a PROP_DRIVER may be adaptive to the parent bus API. In other words, such a driver is able to run on top of a number of different buses (for example, *"pci"* and *"isa"*). Typically, this type of driver is composed of two parts: bus class specific and bus class independent.

The bus class specific part of the driver code mainly deals with the device probing and initialization. In addition, it provides an abstraction layer in order to hide the bus class dependencies from the bus class independent part of the driver code. This kind of multi-bus driver should be registered multiple times in the driver registry. Each entry specifies a given bus class API (via the *bus_class* field) on top of which the driver may run. In order to determine to which bus class the driver is applied, the `drv_probe` , `drv_bind` and `drv_init` routines have to be entry specific (that is, bus class API specific). Under these conditions, when a given driver's routine is invoked by a bus driver, the driver detects the bus class to which it is applied and casts the *bus_ops* argument to the appropriate structure (for example, `PciBusOps`).

The `drv_probe` routine is invoked by a bus / nexus driver when the *bus_class* specified in the registry entry matches the bus / nexus driver class. `drv_probe` is called with three arguments.

The *dev_node* argument specifies the parent device node.

The *bus_ops* / *bus_id* pair specifies the parent device driver.

bus_ops points to a structure of service routines implementing a bus driver API. This structure is bus class specific and corresponds to the *bus_class* specified by the driver registry entry.

bus_id is opaque to the driver. It must be passed back to the bus driver when the `open` bus service routine is invoked. The `drv_probe` routine is optional. In cases where the `drv_probe` routine is not provided by the driver, the *drv_probe* field must be set to `NULL` .

The `drv_bind` routine is invoked by a bus / nexus driver when the *bus_class* specified in the registry entry matches the bus / nexus driver class.

`drv_bind` is called with one argument, *dev_node* , which specifies a given device node. The `drv_bind` routine enables the driver to perform a driver-to-device binding. Typically, the driver examines properties attached to the device node in order to determine the type of device at the node and to check whether the device may be serviced by the driver. If the check is positive, the driver binds the driver to the device node by attaching a PROP_DRIVER property to the device node. The property value specifies the driver name. The parent bus driver uses such a property to determine the name of the driver servicing the device. In other words, via the PROP_DRIVER property, the child driver gives its name to the parent bus driver, asking it to invoke the `drv_init` routine on that device. Note that, if a `"driver"` property is already present in the device node, then the `drv_bind` routine can not continue; `drv_bind` should not override existing driver-to-device binding.

The `drv_bind` routine is optional. If the `drv_bind` routine is not provided by the driver, the `drv_bind` field must be set to NULL.

The `drv_init` routine is invoked by a bus / nexus driver when the *bus_class* specified in the registry entry matches the bus / nexus driver class and a given driver is bound to a given device node.

`drv_init` is called with three arguments. The *dev_node* argument specifies the device node for which a device driver instance should be created. The *bus_ops* / *bus_id* pair specifies the parent device driver. *bus_ops* points to a structure of service routines implementing a bus driver API.

This structure is bus class specific and corresponds to the *bus_class* specified by the driver registry entry. *bus_id* is opaque to the driver. It must be passed back to the bus driver when the `open` bus service routine is invoked. The purpose of `drv_init` is to create an instance of the driver servicing the given device, to perform a device hardware initialization and to register the driver instance in the device registry .

Typically, `drv_init` would read the device and bus node properties in order to obtain the assigned bus resources and tunable parameters related to the bus / device. The `drv_init` routine is optional. In a case when the drv_init routine is not provided by the driver, the drv_init field must be set to NULL. A typical example of a probe-only driver is a self-identifying bus enumerator (for example, a PCI enumerator) which is implemented as driver. This type of

driver has the drv_probe routine which enumerates devices residing on the bus and creates device nodes. This type of driver obviously does not have the drv_init routine.

drv_unload is called by the driver registry module (more exactly by the svDriverUnregister routine) when somebody wishes to unload the driver code from the system. The purpose of drv_unload is to check that the device driver code is not currently being used. The driver must check, for each driver instance, whether it is locked in the device registry.

In case of success, all device instances are removed from the device registry and K_OK is returned. Otherwise, the device entries are unchanged in the device registry and K_EBUSY is returned.

The drv_unload routine is optional. In cases when drv_unload is not provided by the driver, the drv_unload field must be set to NULL . Note that, in this case, the driver code cannot be unloaded.

svDriverLookupFirst returns the first driver entry in the registry. When the registry is not empty a non zero DrvRegId (designating the entry) is returned, otherwise NULL is returned. In case of success, the driver entry is locked in the registry. It should be unlocked by a subsequent invocation of svDriverRelease or svDriverUnregister .

The driver entry lock is not exclusive. It is simply a counter which prevents the driver component from being unloaded when it is still being used by a bus / nexus driver for probing or initialization.

svDriverLookupNext returns the next driver entry in the registry. The current entry is specified by the *drv_id* argument. The current entry must be locked by a previously called svDriverLookupFirst or svDriverLookupNext . If the current entry is not the last one in the registry a non zero DrvRegId (designating the entry) is returned, otherwise NULL is returned. In case of success, the next driver entry is locked in the registry. It should be unlocked by a subsequent invocation of svDriverRelease or svDriverUnregister .

svDriverRelease releases the lock of the driver entry specified by the *drv_id* argument.

svDriverEntry returns a pointer to the driver entry structure specified by the *drv_id* argument. The device entry being accessed must have been previously locked using svDriverLookupFirst or svDriverLookupNext . Note that the device entry structure is read-only.

`svDriverCap` returns a pointer to the driver actor capability. The driver entry is specified by the *drv_id* argument. The driver entry being accessed must have been previously locked using `svDriverLookupFirst` or `svDriverLookupNext` . Note that if a given driver entry is registered by a built-in driver, a `NULL` pointer is returned. In other words, the driver actor capability makes sense only for dynamically loaded drivers. The actor capability may be used by an application in order to delete the driver actor once the driver entry is unregistered. Note that the driver capability structure is read-only.

`svDriverUnregister` tries to remove the driver entry specified by the *dev_id* argument from the driver registry. The device entry being removed must have been previously locked using `svDriverLookupFirst` or `svDriverLookupNext` .

In case of success, `K_OK` is returned, otherwise `K_EBUSY` is returned. The `K_EBUSY` result means that either the driver entry is locked in the driver registry (that is, a static driver routine is currently being used by a bus/nexus driver) or an instance of the device driver is locked in the device registry (that is, there is a driver instance which is currently being used by a driver client). Note that when `K_EBUSY` is returned, the driver entry remains locked in the registry and should be unlocked explicitly by `svDriverRelease` .

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service.

Services	Base level	DKI thread	Interrupt	Blocking
svDriverRegister	+	+	-	+
svDriverLookupFirst	+	+	-	+
svDriverLookupNext	+	+	-	+
svDriverRelease	+	+	-	+
svDriverEntry	+	+	-	-
svDriverCap	+	+	-	-
svDriverUnregister	+	+	-	+

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`dtreeNodeRoot`(9DKI) , `svDkiThreadCall`(9DKI)

NAME

svDriverRegister, svDriverLookupFirst, svDriverLookupNext, svDriverRelease, svDriverEntry, svDriverCap, svDriverUnregister – driver registry operations

SYNOPSIS

```
#include <dki/dki.h>
KnError svDriverRegister(DrvRegEntry * drv_entry);

DrvRegId svDriverLookupFirst(void);

DrvRegId svDriverLookupNext(DrvRegId drv_id);

void svDriverRelease(DrvRegId drv_id);

DrvRegEntry * svDriverEntry(DrvRegId drv_id);

KnCap * svDriverCap(DrvRegId drv_id);

KnError svDriverUnregister(DrvRegId drv_id);
typedef struct {
  char* drv_name;
  char* drv_info;
  char* bus_class;
  int   bus_version;

    void
  (*drv_probe) (DevNode  bus_node,
          void*    bus_ops,
          void*    bus_id);

    void
  (*drv_bind) (DevNode  bus_node);

    void
  (*drv_init) (DevNode  dev_node,
          void*    bus_ops,
          void*    bus_id);

    KnError
  (*drv_unload) ();
} DrvRegEntry;
```

FEATURES

DKI

DESCRIPTION

The `driver registry` module implements a data base of drivers which have registered within the system. The `driver registry` data base is populated by drivers which perform self-registration (using `svDriverRegister`) at driver initialization time.

EXTENDED DESCRIPTION

`svDriverRegister` adds the driver entry to the driver registry. It returns `K_OK` in case of success, otherwise `K_ENOMEM` is returned to indicate that the system is out of memory. The *drv_entry* argument points to the `DrvRegEntry` structure specifying driver properties and static driver routines. The `DrvRegEntry` structure is described below.

drv_name	Points to a string specifying the driver name. For example, the driver file name.
drv_info	Points to a string specifying extra information about the driver component, such as version or author.
bus_class	Points to a string specifying the class of the parent driver API required for the driver, such as “pci”.
bus_version	Specifies the minimum version of the parent driver API required for the driver. Note that if a bus/nexus driver provides an API version which is less than *bus_version* , the corresponding driver component will never be called by this type of bus/nexus driver. In other words, the *drv_probe* and *drv_init* routines will never be invoked by this type of bus/nexus driver.
drv_probe	Points to a static driver routine which performs the device enumeration/probing on the bus.
drv_bind	Points to a static driver routine which performs the driver-to-device binding.
drv_init	Points to a static driver routine which clones an instance of the driver for the given device.
drv_unload	Points to a static driver routine which is invoked by the driver registry module when somebody wishes to unload the driver code from the system.

The *bus_class* and *bus_version* fields specify a parent bus/nexus API required for the driver. This type of API is provided to the driver as a structure of indirect

functions implementing the API service routines. For instance, the *"pci"* bus API is specified by the `PciBusOps` structure, the *"isa"* bus API is specified by `IsaBusOps` . When a bus driver invokes the `drv_probe` or `drv_init` driver's routine, it provides a pointer to the structure of the bus service routines. The structure type corresponds to the `bus_class` field value.

Note that a bus driver may provide multiple API's. A typical example is a PCI bus driver providing the common and PCI bus APIs. The common bus driver API is named by *"bus"* and specified by the `BusOps` structure. The PCI bus driver API is named by *"pci"* and specified by the `PciBusOps` structure. `BusOps` actually, provides a subset of services provided by `PciBusOps` . This type of bus driver is able to support drivers which use either common (*"bus"*) or PCI (*"pci"*) parent bus interfaces. When the bus driver invokes a child driver, it gives a pointer to either the `BusOps` or `PciBusOps` structure depending on the *bus_class* specified in the child driver registry entry.

Note also that a PROP_DRIVER may be adaptive to the parent bus API. In other words, such a driver is able to run on top of a number of different buses (for example, *"pci"* and *"isa"*). Typically, this type of driver is composed of two parts: bus class specific and bus class independent.

The bus class specific part of the driver code mainly deals with the device probing and initialization. In addition, it provides an abstraction layer in order to hide the bus class dependencies from the bus class independent part of the driver code. This kind of multi-bus driver should be registered multiple times in the driver registry. Each entry specifies a given bus class API (via the *bus_class* field) on top of which the driver may run. In order to determine to which bus class the driver is applied, the `drv_probe` , `drv_bind` and `drv_init` routines have to be entry specific (that is, bus class API specific). Under these conditions, when a given driver's routine is invoked by a bus driver, the driver detects the bus class to which it is applied and casts the *bus_ops* argument to the appropriate structure (for example, `PciBusOps`).

The `drv_probe` routine is invoked by a bus / nexus driver when the *bus_class* specified in the registry entry matches the bus / nexus driver class. `drv_probe` is called with three arguments.

The *dev_node* argument specifies the parent device node.

The *bus_ops* / *bus_id* pair specifies the parent device driver.

bus_ops points to a structure of service routines implementing a bus driver API. This structure is bus class specific and corresponds to the *bus_class* specified by the driver registry entry.

bus_id is opaque to the driver. It must be passed back to the bus driver when the `open` bus service routine is invoked. The `drv_probe` routine is optional. In cases where the `drv_probe` routine is not provided by the driver, the *drv_probe* field must be set to `NULL` .

The `drv_bind` routine is invoked by a bus / nexus driver when the *bus_class* specified in the registry entry matches the bus / nexus driver class.

`drv_bind` is called with one argument, *dev_node* , which specifies a given device node. The `drv_bind` routine enables the driver to perform a driver-to-device binding. Typically, the driver examines properties attached to the device node in order to determine the type of device at the node and to check whether the device may be serviced by the driver. If the check is positive, the driver binds the driver to the device node by attaching a PROP_DRIVER property to the device node. The property value specifies the driver name. The parent bus driver uses such a property to determine the name of the driver servicing the device. In other words, via the PROP_DRIVER property, the child driver gives its name to the parent bus driver, asking it to invoke the `drv_init` routine on that device. Note that, if a `"driver"` property is already present in the device node, then the `drv_bind` routine can not continue; `drv_bind` should not override existing driver-to-device binding.

The `drv_bind` routine is optional. If the `drv_bind` routine is not provided by the driver, the `drv_bind` field must be set to NULL.

The `drv_init` routine is invoked by a bus / nexus driver when the *bus_class* specified in the registry entry matches the bus / nexus driver class and a given driver is bound to a given device node.

`drv_init` is called with three arguments. The *dev_node* argument specifies the device node for which a device driver instance should be created. The *bus_ops* / *bus_id* pair specifies the parent device driver. *bus_ops* points to a structure of service routines implementing a bus driver API.

This structure is bus class specific and corresponds to the *bus_class* specified by the driver registry entry. *bus_id* is opaque to the driver. It must be passed back to the bus driver when the `open` bus service routine is invoked. The purpose of `drv_init` is to create an instance of the driver servicing the given device, to perform a device hardware initialization and to register the driver instance in the device registry .

Typically, `drv_init` would read the device and bus node properties in order to obtain the assigned bus resources and tunable parameters related to the bus / device. The `drv_init` routine is optional. In a case when the drv_init routine is not provided by the driver, the drv_init field must be set to NULL. A typical example of a probe-only driver is a self-identifying bus enumerator (for example, a PCI enumerator) which is implemented as driver. This type of

driver has the drv_probe routine which enumerates devices residing on the bus and creates device nodes. This type of driver obviously does not have the drv_init routine.

drv_unload is called by the driver registry module (more exactly by the svDriverUnregister routine) when somebody wishes to unload the driver code from the system. The purpose of drv_unload is to check that the device driver code is not currently being used. The driver must check, for each driver instance, whether it is locked in the device registry.

In case of success, all device instances are removed from the device registry and K_OK is returned. Otherwise, the device entries are unchanged in the device registry and K_EBUSY is returned.

The drv_unload routine is optional. In cases when drv_unload is not provided by the driver, the drv_unload field must be set to NULL . Note that, in this case, the driver code cannot be unloaded.

svDriverLookupFirst returns the first driver entry in the registry. When the registry is not empty a non zero DrvRegId (designating the entry) is returned, otherwise NULL is returned. In case of success, the driver entry is locked in the registry. It should be unlocked by a subsequent invocation of svDriverRelease or svDriverUnregister .

The driver entry lock is not exclusive. It is simply a counter which prevents the driver component from being unloaded when it is still being used by a bus / nexus driver for probing or initialization.

svDriverLookupNext returns the next driver entry in the registry. The current entry is specified by the *drv_id* argument. The current entry must be locked by a previously called svDriverLookupFirst or svDriverLookupNext . If the current entry is not the last one in the registry a non zero DrvRegId (designating the entry) is returned, otherwise NULL is returned. In case of success, the next driver entry is locked in the registry. It should be unlocked by a subsequent invocation of svDriverRelease or svDriverUnregister .

svDriverRelease releases the lock of the driver entry specified by the *drv_id* argument.

svDriverEntry returns a pointer to the driver entry structure specified by the *drv_id* argument. The device entry being accessed must have been previously locked using svDriverLookupFirst or svDriverLookupNext . Note that the device entry structure is read-only.

`svDriverCap` returns a pointer to the driver actor capability. The driver entry is specified by the *drv_id* argument. The driver entry being accessed must have been previously locked using `svDriverLookupFirst` or `svDriverLookupNext` . Note that if a given driver entry is registered by a built-in driver, a `NULL` pointer is returned. In other words, the driver actor capability makes sense only for dynamically loaded drivers. The actor capability may be used by an application in order to delete the driver actor once the driver entry is unregistered. Note that the driver capability structure is read-only.

`svDriverUnregister` tries to remove the driver entry specified by the *dev_id* argument from the driver registry. The device entry being removed must have been previously locked using `svDriverLookupFirst` or `svDriverLookupNext` .

In case of success, `K_OK` is returned, otherwise `K_EBUSY` is returned. The `K_EBUSY` result means that either the driver entry is locked in the driver registry (that is, a static driver routine is currently being used by a bus/nexus driver) or an instance of the device driver is locked in the device registry (that is, there is a driver instance which is currently being used by a driver client). Note that when `K_EBUSY` is returned, the driver entry remains locked in the registry and should be unlocked explicitly by `svDriverRelease` .

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service.

Services	Base level	DKI thread	Interrupt	Blocking
svDriverRegister	+	+	-	+
svDriverLookupFirst	+	+	-	+
svDriverLookupNext	+	+	-	+
svDriverRelease	+	+	-	+
svDriverEntry	+	+	-	-
svDriverCap	+	+	-	-
svDriverUnregister	+	+	-	+

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`dtreeNodeRoot`(9DKI) , `svDkiThreadCall`(9DKI)

NAME

svIntrAttach, svIntrDetach, svSoftIntrAttach, svSoftIntrDetach, svTimerIntrAttach, svTimerIntrDetach, svIntrCtxGet – interrupts management

FEATURES

DKI

DESCRIPTION

Provides interrupts management services.

See the architecture specific man pages:

- `svIntrAttach_usparc`(9DKI)
- `svIntrDetach_usparc`(9DKI)
- `svSoftIntrAttach_usparc`(9DKI)
- `svSoftIntrDetach_usparc`(9DKI)
- `svTimerIntrAttach_usparc`(9DKI)
- `svTimerIntrDetach_usparc`(9DKI)
- `svIntrCtxGet_usparc`(9DKI)
- `svIntrAttach_powerpc`(9DKI)
- `svIntrDetach_powerpc`(9DKI)
- `svIntrCtxGet_powerpc`(9DKI)
- `svIntrAttach_x86`(9DKI)
- `svIntrDetach_x86`(9DKI)
- `svIntrCtxGet_x86`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

svIntrAttach_powerpc, svIntrDetach_powerpc, svIntrCtxGet_powerpc – PowerPC interrupts management

SYNOPSIS

```
#include <dki/f_dki.h>
KnError svIntrAttach(CpuIntr intr, CpuIntrHandler intrHandler, void * intrCookie,
CpuIntrOps ** intrOps, CpuIntrId * intrId);

void svIntrDetach(CpuIntrId intrId);

KnError svIntrCtxGet(KnIntrCtx ** intrCtx);
typedef struct {
    void
  (*unmask) (CpuIntrId intrId);
    void
  (*mask) (CpuIntrId intrId);
} CpuIntrOps;
```

FEATURES

DKI

DESCRIPTION

Provides access to PowerPC DKI interrupts services.

EXTENDED DESCRIPTION

For PowerPC family processors, interrupts are defined as all asynchronous exceptions. These interrupts are identified using unsigned integer numbers which correspond to the exception number. The microkernel provides services to allow device drivers to manage PowerPC interrupts, mainly to attach / detach handlers to these interrupts.

`svIntrAttach` attaches a handler to a given CPU interrupt. The *intr* argument specifies the interrupt to attach to. The *intrHandler* argument specifies the handler to call back when the specified interrupt occurs. The *intrCookie* argument specifies a parameter to pass back to the handler when called. It is opaque to the microkernel.

On success, `K_OK` is returned and the services defined on the attached interrupt are returned in the *intrOps* parameter. An identifier for the attached interrupt is also returned in *intrId* . This identifier must be used as the first parameter to further calls to *intrOps* services.

Services available on an attached interrupt are defined by the `CpuIntrOps` structure as follows:

The `CpuIntrOps.unmask` routine enables the interrupt at CPU level identified by *intrId* .

The `CpuIntrOps.mask` routine disables the interrupt at CPU level identified by *intrId* .

Note that SYSTEM_RESET_INTR and MACHINE_CHECK_INTR are non-maskable interrupts. Thus unmask/mask have no effect on these interrupts.

Note also, that all other interrupts are masked using the same bit in the PowerPC processor MSR register. Thus there is no way to unmask/mask only one of these interrupts separately from the others. When an interrupt occurs, the attached *CpuIntrHandler* is invoked with interrupts masked at processor level. This behaves in the same way as if imsIntrMask_f() was just called prior to invoking the handler (imsIntrMaskCount_f is positive). It is up to the interrupt handler called to use unmask to allow the interrupt to be nested or not. Typically, a host bus driver handler should handle PowerPC external interrupts as follows:

- Identify the interrupt source (through a PIC or special cycle).
- Unmask interrupts at processor level (unmask).
- Call handlers attached to the identified source.
- Optionally mask interrupts to do critical tasks, such as notifying the end of the interrupt to a PIC.
- Return to the DKI.

On failure, an error code is returned as follows:

K_EINVAL	The interrupt specified in intr is invalid (not in the list above).
K_EFAIL	The interrupt specified in intr is valid, but a handler is already attached to it.
K_ENOMEM	The system is out of physical memory.

svIntrDetach detaches an interrupt handler previously connected by svIntrAttach . The *intrId* argument identifies the attached interrupt handler, and was previously returned by svIntrAttach .

svIntrCtxGet retrieves the current level interrupt context. It is mainly to be used for profiling purposes. On success, K_OK is returned and a pointer to the current level interrupt saved context is returned in the *intrCtx* argument. The CPU context saved on interrupt contains the volatile general purpose and floating point registers, condition register, Machine Status Register, instruction pointer, link register and the interrupt number itself. On failure, an error code is returned as follows:

K_EINVAL	The interrupt level is zero (not called from an interrupt handler.

K_ENOTAVAILABLE	There is no context available for the currently handled interrupt.

ALLOWED CALLING CONTEXTS

The following table specifies the contexts in which a caller is allowed to invoke each service.

Services	Base level	DKI thread	Interrupt	Blocking
svIntrAttach	+	+	-	+
CpuIntrOps.unmask	+	+	+	-
CpuIntrOps.mask	+	+	+	-
svIntrDetach	+	+	-	+
svIntrCtxGet	-	-	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`loadSwap`(9DKI) , `svDkiThreadCall`(9DKI) , `imsIntrMask_f`(9DKI)

NAME

svIntrAttach_usparc, svIntrDetach_usparc, svSoftIntrAttach_usparc, svSoftIntrDetach_usparc, svTimerIntrAttach_uparc, svTimerIntrDetach_uparc, svIntrCtxGet_usparc – UltraSPARC interrupts management

SYNOPSIS

#include <dki/f_dki.h>

KnError **svIntrAttach**(unsigned int *intrNumb*, unsigned int *intrLevel*, CpuIntrHandler *intrHandler*, void * *intrCookie*, CpuIntrOps ** *intrOps*, CpuIntrId * *intrIdp*);

void **svIntrDetach**(CpuIntrId *intrId*);

KnError **svSoftIntrAttach**(unsigned int *intrLevel*, CpuIntrHandler *intrHandler*, void * *intrCookie*, CpuIntrOps ** *intrOps*, CpuIntrId * *intrId*);

void **svSoftIntrDetach**(CpuIntrId *intrId*);

KnError **svTimerIntrAttach**(CpuIntrHandler *intrHandler*, void * *intrCookie*, CpuIntrOps ** *intrOps*, CpuIntrId * *intrId*);

void **svTimerIntrDetach**(CpuIntrId *intrId*);

KnError **svIntrCtxGet**(KnIntrCtx ** *intrCtx*);

FEATURES

DKI

DESCRIPTION

Provides UltraSPARC interrupts management services.

An UltraSPARC processor is an implementation of the SPARC-V9 CPU architecture. As specified by the SPARC-V9 architecture, there are fifteen CPU interrupt sources assigned to the fifteen CPU interrupt levels from 1 up to 15. However, the UltraSPARC processor uses these fifteen interrupt levels only for software generated interrupts. The hardware interrupts are delivered to the processor using the "Mondo" interrupt transfer mechanism. The hardware interrupt source is designated by the interrupt number. Typically, the interrupt number is 11-bit width and composed of the interrupt group number (5 MSB) and the interrupt offset number (6 LSB).

The "Mondo" interrupt dispatch handler is built into the microkernel. The microkernel handles a mapping between the hardware interrupt numbers and software interrupt levels. When a mondo interrupt packet is received by the microkernel, the interrupt request descriptor is queued and an associated software interrupt is triggered. The microkernel software interrupt handler then dequeues the interrupt request descriptor and invokes a handler associated to the given interrupt number. In this way, a driver interrupt handler is always invoked in the UltraSPARC software interrupt context. This provides the interrupt handler with an environment analogous to the SPARC-V8 one (interrupt levels).

The microkernel provides services which allow device drivers to manage UltraSPARC interrupts, mainly to attach/detach handlers to the CPU interrupts.

```
typedef void (*CpuIntrHandler)(void*);
```

svIntrAttach

`svIntrAttach` attaches a given handler to a given interrupt number at a given processor interrupt level.

The *intrNumb* argument specifies the interrupt number to which to attach.

The *intrLevel* argument specifies the interrupt level to which to attach.

The *intrHandler* argument specifies the handler to call back when the given interrupt occurs.

The *intrCookie* argument specifies a parameter to pass back to the handler when called. It is opaque to the microkernel.

On success, `K_OK` is returned and the services defined on the attached interrupt are returned in the *intrOps* parameter. An identifier for the attached interrupt is also returned in *intrId* . This identifier must be used as first argument to subsequent calls to *intrOps* services.

On failure, an error code is returned as follows:

`K_EINVAL`	The specified interrupt number or interrupt level are invalid.
`K_BUSY`	Another handler is already attached to the given *intrNumb* .
`K_ENOMEM`	The system is out of memory.

Services available on an attached interrupt are defined by the `CpuIntrOps` structure as follows:

```
typedef struct CpuIntrOps {

        void
    (*mask) (CpuIntrId intrId);

        void
    (*unmask) (CpuIntrId intrId);

        void
    (*enable) (CpuIntrId intrId);

        void
    (*disable) (CpuIntrId intrId);

        void
    (*trigger) (CpuIntrId intrId);

} CpuIntrOps;
```

The `CpuIntrOps.mask` routine disables the interrupt at CPU level identified by *intrId* . In other words, the PIL register is set to the level corresponding to *intrId* . Note that the original value of the PIL register is saved by DKI in order to be restored later by the `CpuIntrOps.unmask` routine.

The `CpuIntrOps.unmask` routine enables the interrupt at CPU level identified by *intrId* .

In other words, the PIL register is restored to the original value saved by the previously called `CpuIntrOps.mask` routine.

The `mask` / `unmask` pair may be called from base level only and must not be nested. The `mask/unmask` pair is typically used to implement a critical section of code which needs to be protected against the interrupt. Note that with respect to the SPARC-V9 architecture, when an interrupt level `N` is masked, all interrupts with a level less than `N` are also masked.Thus, there is no way to `mask` only one CPU interrupt level except the lowest one.

The `enable` / `disable` pair may only be called from the attached interrupt handler.When an interrupt occurs, the attached *CpuIntrHandler* is invoked with the interrupt masked at processor level. This behaves in exactly the same way as if `CpuIntrOps.disable` was called just prior to the handler invocation. In other words, the PIL register is set to the interrupt level and the original interrupt processor level (which was when the interrupt occurred) is saved by DKI. Note that the interrupt handler must return to DKI in the same context as it was called, that is with the interrupt disabled at processor level.

On the other hand, the called interrupt handler may use the `enable/disable` pair to allow the interrupt to be nested. This feature is typically used by a host bus driver when the bus interrupts are multiplexed, that is, multiple bus interrupts are reported at the same CPU interrupt level. Typically, an interrupt handler of this type of host bus driver would take the following actions:

- Identify the bus interrupt source (through a PIC or special cycle).
- Disable the bus interrupt source at bus level (through PIC).
- Enable interrupt at processor level (`enable`).
- Call handlers attached to the identified bus interrupt source.
- Disable interrupt at processor level (\ \command{disable}).
- Acknowledge (if needed) and enable the bus interrupt source at bus level (through PIC).
- Return to the DKI.

The `CpuIntrOps.trigger` routine allows the interrupt to be triggered by software. Basically, this routine acts like the mondo interrupt dispatcher except the interrupt number is obtained from *intrId* rather than from the mondo interrupt packet. The `CpuIntrOps.trigger` routine is mainly dedicated to the

software interrupts attached by `svSoftIntrAttach`. However, it may be also used for hardware interrupts, for instance, for debugging or diagnostic purposes.

svIntrDetach

`svIntrDetach` detaches an interrupt handler previously connected by `svIntrAttach`.

The *intrId* argument identifies the attached interrupt handler, previously returned by `svIntrAttach`.

svSoftIntrDetach

`svSoftIntrAttach` attaches a given software interrupt handler to a given processor interrupt level.

The *intrLevel* argument specifies the interrupt level to which to attach.

The *intrHandler* argument specifies the handler to call back when the given interrupt is triggered.

The *intrCookie* argument specifies a parameter to pass back to the handler when called. It is opaque to the microkernel.

On success, `K_OK` is returned and the services defined on the attached interrupt are returned in the *intrOps* parameter.

An identifier for the attached interrupt is also returned in *intrId*. This identifier must be used as the first argument to subsequent calls to *intrOps* services.

On failure, an error code is returned as follows:

`K_EINVAL`	The specified interrupt level is invalid.
`K_ENOMEM`	The system is out of memory.

Services available on an attached interrupt are defined by the `CpuIntrOps` structure described above.

svSoftIntrDetach

`svSoftIntrDetach` detaches a software interrupt handler previously connected by `svSoftIntrAttach`.

The *intrId* argument identifies the attached interrupt handler, previously returned by `svSoftIntrAttach`.

svTimerIntrAttach

`svTimerIntrAttach` attaches a given interrupt handler to the UltraSPARC tick-counter interrupt.

The *intrHandler* argument specifies the handler to call back when the given interrupt is triggered.

The *intrCookie* argument specifies a parameter to pass back to the handler when called. It is opaque the microkernel.

On success, K_OK is returned and the services defined on the attached interrupt are returned in the *intrOps* parameter. An identifier for the attached interrupt is also returned in *intrId* . This identifier must be used as the first argument to subsequent calls to *intrOps* services.

On failure, an error code is returned as follows:

K_BUSY	Another handler is already attached to the tick-counter interrupt.
K_ENOMEM	The system is out of memory.

Services available on an attached interrupt are defined by the CpuIntrOps structure described above.

svTimerIntrDetach

svTimerIntrDetach detaches the interrupt handler previously connected by svTimerIntrAttach .

The *intrId* argument identifies the attached interrupt handler, previously returned by svTimerIntrAttach .

svIntrCtxGet

svIntrCtxGet retrieves the current level interrupt context. It is typically used for profiling purposes.

On successs, K_OK is returned and a pointer to the recently saved interrupt context is returned in the *intrCtx* argument. The CPU context saved on interrupt has the same structure as the thread context saved on exception or trap. It contains the global registers %g1-%g7 , the output registers of the interrupted window %o0-%o7 and the following processor registers: %tstate , %pc , %npc , %tt , %y . In addition, the thread context contains the number of outstanding windows and the pointer to the outstanding windows buffer if the number of windows is greater than zero.

On failure, an error code is returned as follows:

K_EINVAL	Interrupt level is zero (not called from an interrupt handler).
K_ENOTAVAILABLE	There is no context available for currently handled interrupt.

ALLOWED CALLING CONTEXTS

The following table specifies the contexts in which a caller is allowed to invoke each interrupt management service.

Services	Base level	DKI thread	Interrupt	Blocking
svIntrAttach	+	+	-	+

svIntrDetach	+	+	-	+
svIntrSoftAttach	+	+	-	+
svIntrSoftDetach	+	+	-	+
svTimerIntrAttach	+	+	-	+
svTimerIntrDetach	+	+	-	+
svIntrCtxGet	-	-	+	-
CpuIntrOps.mask	+	+	-	-
CpuIntrOps.unmask	+	+	-	-
CpuIntrOps.enable	-	-	+	-
CpuIntrOps.disable	-	-	+	-
CpuIntrOps.trigger	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

svIntrAttach_x86, svIntrDetach_x86, svIntrCtxGet_x86 – Intel x86 interrupts management

SYNOPSIS

```
#include <dki/f_dki.h>
KnError svIntrAttach(CpuIntr intr, CpuIntrHandler intrHandler, void * intrCookie,
CpuIntrOps ** intrOps, CpuIntrId * intrId);

void svIntrDetach(CpuIntrId intrId);

KnError svIntrCtxGet(KnIntrCtx ** intrCtx);
typedef struct CpuIntrOps {
    void
  (*enable) (CpuIntrId intrId);
    void
  (*disable) (CpuIntrId intrId);
} CpuIntrOps;
```

FEATURES

DKI

DESCRIPTION

Provides Intel x86 interrupts management services.

EXTENDED DESCRIPTION

For Intel family processors, there are two kinds of interrupts:

Exceptions	Exceptions are generated and detected by the processor when excecuting instructions, such as divsion by 0. These exceptions are classified as faults, traps and aborts. Another type of exception exists: Programmed exceptions generated by INTx instructions.
Interrupts	Interrupts are generated by events external to the processor, such as requests to service peripheral devices.

There are two further types of interrupts:

Maskable	Maskable interrupts are received on the INTR input pin of the Intel x86 processor. They do not occur if the IF flag of the EFLAGS register is not set.
Non Maskable	Non maskable interrupts are received on the NMI input pin of the Intel x86 processor. There is no mechanism to prevent non maskable interrupts.

The DKI does not provide support for exceptions, only for interrupts. The processor associates an identifying number with each different interrupt (non maskable and maskable). This number is called a vector in the range from 0-255 with the range of 0-31 reserved for exceptions. The DKI associates an identifying numeric constant with each interrupt.

The following numeric constants are available:

- `NMI_INT` (non maskable).
- `INTR_BASE_NUMBER` (maskable)

The DKI provides support for 64 interrupts, ranging from INTR_BASE_NUMBER to INTR_BASE_NUMBER + 64.

The DKI provides services to allow host bus drivers to manage Intel ix86 interrupts, mainly to attach / detach handlers to / from these interrupts.

`svIntrAttach`

`svIntrAttach` attaches a handler to a given CPU interrupt.

The *intr* argument specifies the interrupt to which to attach.

The *intrHandler* argument specifies the handler to call back when the specified interrupt occurs.

The *intrCookie* argument specifies a parameter to pass back to the handler when called. It is opaque to the microkernel.

On succes, `K_OK` is returned and services defined on the attached interrupt are returned in the *intrOps* parameter. An identifier for the attached interrupt is also returned in *intrId* . This identifier must be used as the first parameter to further calls to *intrOps* services.

Services available on an attached interrupt are defined by the `CpuIntrOps` structure as follows:

The `CpuIntrOps.enable` routine enables the interrupt at CPU level identified by *intrId* .

The `CpuIntrOps.disable` routine disables the interrupt at CPU level identified by *intrId* .

Note that `NMI_INT` is a non maskable interrupt. Thus, `enable/disable` have no effect on these interrupts..

Note also that all other interrupts are masked using the same bit on an Intel x86 processor `EFLAGS` register. Thus there is no way to `enable/disable` a single interrupt separately from the others.

When an interrupt occurs, the attached *CpuIntrHandler* is invoked with interrupts masked at processor level. This produces the same effect as if `imsIntrMask_f()` was called just prior to invoking the handler (`imsIntrMaskCount_f` is positive). Two parameters are passed to this handler:

- Provided *intrCookie* parameter
- Detected *intr*

It is up to the called interrupt handler to use `enable` in order to allow or disallow interrupt nesting.

Typically, a host bus driver handler should handle x86 external interrupts as follows:

- Enable interrupts at processor level (`enable`).
- Call handler(s) attached to the identified cpu interrupt.
- Optionally `disable` interrupts in order to perform critical tasks, such as notifying the end an of interrupt to a PIC.
- Return to the DKI.

On failure, an error code is returned as follows:

`K_EINVAL`	The interrupt specified in *intr* is invalid (not in the list above). Or, the interrupt specified in *intr* is valid, but a handler is already attached to it.

`svIntrDetach`

`svIntrDetach` detaches an interrupt handler previously connected by `svIntrAttach` .

The *intrId* argument identifies the attached interrupt handler, and was previously returned by `svIntrAttach` .

`svIntrCtxGet`

`svIntrCtxGet` retrieves the current level interrupt context. It is mainly to be used for profiling purposes.

On success, `K_OK` is returned and a pointer to the current level of interrupt saved context is returned in the *intrCtx* argument. The CPU context saved on interrupt contains the interrupted PC composed of the `eip` and `cs` registers, the `EFLAGS` register, and the stack pointer.

On failure, an error code is returned as follows:

K_EINVAL	Interrupt level is zero (not called from an interrupt handler).
K_ENOTAVAILABLE	There is no context available for the currently handled interrupt.

Intel x86 Interrupts Management Allowed Calling Contexts

The table below specifies the contexts in which a caller is allowed to invoke each services.

Services	Base level	DKI thread	Interrupt	Blocking
svIntrAttach	+	+	-	+
CpuIntrOps.enable	+	+	+	-
CpuIntrOps.disable	+	+	+	-
svIntrDetach	+	+	-	+
svIntrCtxGet	-	-	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

svDkiThreadCall(9DKI)

NAME

svIntrAttach, svIntrDetach, svSoftIntrAttach, svSoftIntrDetach, svTimerIntrAttach, svTimerIntrDetach, svIntrCtxGet – interrupts management

FEATURES

DKI

DESCRIPTION

Provides interrupts management services.

See the architecture specific man pages:

- `svIntrAttach_usparc`(9DKI)
- `svIntrDetach_usparc`(9DKI)
- `svSoftIntrAttach_usparc`(9DKI)
- `svSoftIntrDetach_usparc`(9DKI)
- `svTimerIntrAttach_usparc`(9DKI)
- `svTimerIntrDetach_usparc`(9DKI)
- `svIntrCtxGet_usparc`(9DKI)
- `svIntrAttach_powerpc`(9DKI)
- `svIntrDetach_powerpc`(9DKI)
- `svIntrCtxGet_powerpc`(9DKI)
- `svIntrAttach_x86`(9DKI)
- `svIntrDetach_x86`(9DKI)
- `svIntrCtxGet_x86`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

svIntrAttach_powerpc, svIntrDetach_powerpc, svIntrCtxGet_powerpc – PowerPC interrupts management

SYNOPSIS

```
#include <dki/f_dki.h>
KnError svIntrAttach(CpuIntr intr, CpuIntrHandler intrHandler, void * intrCookie,
CpuIntrOps ** intrOps, CpuIntrId * intrId);

void svIntrDetach(CpuIntrId intrId);

KnError svIntrCtxGet(KnIntrCtx ** intrCtx);
typedef struct {
    void
  (*unmask) (CpuIntrId intrId);
    void
  (*mask) (CpuIntrId intrId);
} CpuIntrOps;
```

FEATURES

DKI

DESCRIPTION

Provides access to PowerPC DKI interrupts services.

EXTENDED DESCRIPTION

For PowerPC family processors, interrupts are defined as all asynchronous exceptions. These interrupts are identified using unsigned integer numbers which correspond to the exception number. The microkernel provides services to allow device drivers to manage PowerPC interrupts, mainly to attach / detach handlers to these interrupts.

`svIntrAttach` attaches a handler to a given CPU interrupt. The *intr* argument specifies the interrupt to attach to. The *intrHandler* argument specifies the handler to call back when the specified interrupt occurs. The *intrCookie* argument specifies a parameter to pass back to the handler when called. It is opaque to the microkernel.

On success, `K_OK` is returned and the services defined on the attached interrupt are returned in the *intrOps* parameter. An identifier for the attached interrupt is also returned in *intrId* . This identifier must be used as the first parameter to further calls to *intrOps* services.

Services available on an attached interrupt are defined by the `CpuIntrOps` structure as follows:

The `CpuIntrOps.unmask` routine enables the interrupt at CPU level identified by *intrId* .

The `CpuIntrOps.mask` routine disables the interrupt at CPU level identified by *intrId* .

Note that SYSTEM_RESET_INTR and MACHINE_CHECK_INTR are non-maskable interrupts. Thus unmask/mask have no effect on these interrupts.

Note also, that all other interrupts are masked using the same bit in the PowerPC processor MSR register. Thus there is no way to unmask/mask only one of these interrupts separately from the others. When an interrupt occurs, the attached *CpuIntrHandler* is invoked with interrupts masked at processor level. This behaves in the same way as if imsIntrMask_f() was just called prior to invoking the handler (imsIntrMaskCount_f is positive). It is up to the interrupt handler called to use unmask to allow the interrupt to be nested or not. Typically, a host bus driver handler should handle PowerPC external interrupts as follows:

- Identify the interrupt source (through a PIC or special cycle).
- Unmask interrupts at processor level (unmask).
- Call handlers attached to the identified source.
- Optionally mask interrupts to do critical tasks, such as notifying the end of the interrupt to a PIC.
- Return to the DKI.

On failure, an error code is returned as follows:

K_EINVAL	The interrupt specified in intr is invalid (not in the list above).
K_EFAIL	The interrupt specified in intr is valid, but a handler is already attached to it.
K_ENOMEM	The system is out of physical memory.

svIntrDetach detaches an interrupt handler previously connected by svIntrAttach . The *intrId* argument identifies the attached interrupt handler, and was previously returned by svIntrAttach .

svIntrCtxGet retrieves the current level interrupt context. It is mainly to be used for profiling purposes. On success, K_OK is returned and a pointer to the current level interrupt saved context is returned in the *intrCtx* argument. The CPU context saved on interrupt contains the volatile general purpose and floating point registers, condition register, Machine Status Register, instruction pointer, link register and the interrupt number itself. On failure, an error code is returned as follows:

K_EINVAL	The interrupt level is zero (not called from an interrupt handler.

K_ENOTAVAILABLE	There is no context available for the currently handled interrupt.

ALLOWED CALLING CONTEXTS

The following table specifies the contexts in which a caller is allowed to invoke each service.

Services	Base level	DKI thread	Interrupt	Blocking
svIntrAttach	+	+	-	+
CpuIntrOps.unmask	+	+	+	-
CpuIntrOps.mask	+	+	+	-
svIntrDetach	+	+	-	+
svIntrCtxGet	-	-	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`loadSwap`(9DKI) , `svDkiThreadCall`(9DKI) , `imsIntrMask_f`(9DKI)

NAME

svIntrAttach_usparc, svIntrDetach_usparc, svSoftIntrAttach_usparc, svSoftIntrDetach_usparc, svTimerIntrAttach_uparc, svTimerIntrDetach_uparc, svIntrCtxGet_usparc – UltraSPARC interrupts management

SYNOPSIS

#include <dki/f_dki.h>

KnError **svIntrAttach**(unsigned int *intrNumb*, unsigned int *intrLevel*, CpuIntrHandler *intrHandler*, void * *intrCookie*, CpuIntrOps ** *intrOps*, CpuIntrId * *intrIdp*);

void **svIntrDetach**(CpuIntrId *intrId*);

KnError **svSoftIntrAttach**(unsigned int *intrLevel*, CpuIntrHandler *intrHandler*, void * *intrCookie*, CpuIntrOps ** *intrOps*, CpuIntrId * *intrId*);

void **svSoftIntrDetach**(CpuIntrId *intrId*);

KnError **svTimerIntrAttach**(CpuIntrHandler *intrHandler*, void * *intrCookie*, CpuIntrOps ** *intrOps*, CpuIntrId * *intrId*);

void **svTimerIntrDetach**(CpuIntrId *intrId*);

KnError **svIntrCtxGet**(KnIntrCtx ** *intrCtx*);

FEATURES

DKI

DESCRIPTION

Provides UltraSPARC interrupts management services.

An UltraSPARC processor is an implementation of the SPARC-V9 CPU architecture. As specified by the SPARC-V9 architecture, there are fifteen CPU interrupt sources assigned to the fifteen CPU interrupt levels from 1 up to 15. However, the UltraSPARC processor uses these fifteen interrupt levels only for software generated interrupts. The hardware interrupts are delivered to the processor using the "Mondo" interrupt transfer mechanism. The hardware interrupt source is designated by the interrupt number. Typically, the interrupt number is 11-bit width and composed of the interrupt group number (5 MSB) and the interrupt offset number (6 LSB).

The "Mondo" interrupt dispatch handler is built into the microkernel. The microkernel handles a mapping between the hardware interrupt numbers and software interrupt levels. When a mondo interrupt packet is received by the microkernel, the interrupt request descriptor is queued and an associated software interrupt is triggered. The microkernel software interrupt handler then dequeues the interrupt request descriptor and invokes a handler associated to the given interrupt number. In this way, a driver interrupt handler is always invoked in the UltraSPARC software interrupt context. This provides the interrupt handler with an environment analogous to the SPARC-V8 one (interrupt levels).

The microkernel provides services which allow device drivers to manage UltraSPARC interrupts, mainly to attach/detach handlers to the CPU interrupts.

```
typedef void (*CpuIntrHandler)(void*);
```

svIntrAttach

`svIntrAttach` attaches a given handler to a given interrupt number at a given processor interrupt level.

The *intrNumb* argument specifies the interrupt number to which to attach.

The *intrLevel* argument specifies the interrupt level to which to attach.

The *intrHandler* argument specifies the handler to call back when the given interrupt occurs.

The *intrCookie* argument specifies a parameter to pass back to the handler when called. It is opaque to the microkernel.

On success, `K_OK` is returned and the services defined on the attached interrupt are returned in the *intrOps* parameter. An identifier for the attached interrupt is also returned in *intrId* . This identifier must be used as first argument to subsequent calls to *intrOps* services.

On failure, an error code is returned as follows:

`K_EINVAL`	The specified interrupt number or interrupt level are invalid.
`K_BUSY`	Another handler is already attached to the given *intrNumb* .
`K_ENOMEM`	The system is out of memory.

Services available on an attached interrupt are defined by the `CpuIntrOps` structure as follows:

```
typedef struct CpuIntrOps {

        void
    (*mask) (CpuIntrId intrId);

        void
    (*unmask) (CpuIntrId intrId);

        void
    (*enable) (CpuIntrId intrId);

        void
    (*disable) (CpuIntrId intrId);

        void
    (*trigger) (CpuIntrId intrId);

} CpuIntrOps;
```

The `CpuIntrOps.mask` routine disables the interrupt at CPU level identified by *intrId* . In other words, the PIL register is set to the level corresponding to *intrId* . Note that the original value of the PIL register is saved by DKI in order to be restored later by the `CpuIntrOps.unmask` routine.

The `CpuIntrOps.unmask` routine enables the interrupt at CPU level identified by *intrId* .

In other words, the PIL register is restored to the original value saved by the previously called `CpuIntrOps.mask` routine.

The `mask` / `unmask` pair may be called from base level only and must not be nested. The `mask/unmask` pair is typically used to implement a critical section of code which needs to be protected against the interrupt. Note that with respect to the SPARC-V9 architecture, when an interrupt level `N` is masked, all interrupts with a level less than `N` are also masked.Thus, there is no way to `mask` only one CPU interrupt level except the lowest one.

The `enable` / `disable` pair may only be called from the attached interrupt handler.When an interrupt occurs, the attached *CpuIntrHandler* is invoked with the interrupt masked at processor level. This behaves in exactly the same way as if `CpuIntrOps.disable` was called just prior to the handler invocation. In other words, the PIL register is set to the interrupt level and the original interrupt processor level (which was when the interrupt occurred) is saved by DKI. Note that the interrupt handler must return to DKI in the same context as it was called, that is with the interrupt disabled at processor level.

On the other hand, the called interrupt handler may use the `enable/disable` pair to allow the interrupt to be nested. This feature is typically used by a host bus driver when the bus interrupts are multiplexed, that is, multiple bus interrupts are reported at the same CPU interrupt level. Typically, an interrupt handler of this type of host bus driver would take the following actions:

- Identify the bus interrupt source (through a PIC or special cycle).
- Disable the bus interrupt source at bus level (through PIC).
- Enable interrupt at processor level (`enable`).
- Call handlers attached to the identified bus interrupt source.
- Disable interrupt at processor level (\ \command{disable}).
- Acknowledge (if needed) and enable the bus interrupt source at bus level (through PIC).
- Return to the DKI.

The `CpuIntrOps.trigger` routine allows the interrupt to be triggered by software. Basically, this routine acts like the mondo interrupt dispatcher except the interrupt number is obtained from *intrId* rather than from the mondo interrupt packet. The `CpuIntrOps.trigger` routine is mainly dedicated to the

software interrupts attached by `svSoftIntrAttach`. However, it may be also used for hardware interrupts, for instance, for debugging or diagnostic purposes.

svIntrDetach

`svIntrDetach` detaches an interrupt handler previously connected by `svIntrAttach`.

The *intrId* argument identifies the attached interrupt handler, previously returned by `svIntrAttach`.

svSoftIntrDetach

`svSoftIntrAttach` attaches a given software interrupt handler to a given processor interrupt level.

The *intrLevel* argument specifies the interrupt level to which to attach.

The *intrHandler* argument specifies the handler to call back when the given interrupt is triggered.

The *intrCookie* argument specifies a parameter to pass back to the handler when called. It is opaque to the microkernel.

On success, `K_OK` is returned and the services defined on the attached interrupt are returned in the *intrOps* parameter.

An identifier for the attached interrupt is also returned in *intrId*. This identifier must be used as the first argument to subsequent calls to *intrOps* services.

On failure, an error code is returned as follows:

`K_EINVAL`	The specified interrupt level is invalid.
`K_ENOMEM`	The system is out of memory.

Services available on an attached interrupt are defined by the `CpuIntrOps` structure described above.

svSoftIntrDetach

`svSoftIntrDetach` detaches a software interrupt handler previously connected by `svSoftIntrAttach`.

The *intrId* argument identifies the attached interrupt handler, previously returned by `svSoftIntrAttach`.

svTimerIntrAttach

`svTimerIntrAttach` attaches a given interrupt handler to the UltraSPARC tick-counter interrupt.

The *intrHandler* argument specifies the handler to call back when the given interrupt is triggered.

The *intrCookie* argument specifies a parameter to pass back to the handler when called. It is opaque the microkernel.

On success, K_OK is returned and the services defined on the attached interrupt are returned in the *intrOps* parameter. An identifier for the attached interrupt is also returned in *intrId* . This identifier must be used as the first argument to subsequent calls to *intrOps* services.

On failure, an error code is returned as follows:

K_BUSY	Another handler is already attached to the tick-counter interrupt.
K_ENOMEM	The system is out of memory.

Services available on an attached interrupt are defined by the CpuIntrOps structure described above.

svTimerIntrDetach

svTimerIntrDetach detaches the interrupt handler previously connected by svTimerIntrAttach .

The *intrId* argument identifies the attached interrupt handler, previously returned by svTimerIntrAttach .

svIntrCtxGet

svIntrCtxGet retrieves the current level interrupt context. It is typically used for profiling purposes.

On successs, K_OK is returned and a pointer to the recently saved interrupt context is returned in the *intrCtx* argument. The CPU context saved on interrupt has the same structure as the thread context saved on exception or trap. It contains the global registers %g1-%g7 , the output registers of the interrupted window %o0-%o7 and the following processor registers: %tstate , %pc , %npc , %tt , %y . In addition, the thread context contains the number of outstanding windows and the pointer to the outstanding windows buffer if the number of windows is greater than zero.

On failure, an error code is returned as follows:

K_EINVAL	Interrupt level is zero (not called from an interrupt handler).
K_ENOTAVAILABLE	There is no context available for currently handled interrupt.

ALLOWED CALLING CONTEXTS

The following table specifies the contexts in which a caller is allowed to invoke each interrupt management service.

Services	Base level	DKI thread	Interrupt	Blocking
svIntrAttach	+	+	-	+

svIntrDetach	+	+	-	+
svIntrSoftAttach	+	+	-	+
svIntrSoftDetach	+	+	-	+
svTimerIntrAttach	+	+	-	+
svTimerIntrDetach	+	+	-	+
svIntrCtxGet	-	-	+	-
CpuIntrOps.mask	+	+	-	-
CpuIntrOps.unmask	+	+	-	-
CpuIntrOps.enable	-	-	+	-
CpuIntrOps.disable	-	-	+	-
CpuIntrOps.trigger	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

svIntrAttach_x86, svIntrDetach_x86, svIntrCtxGet_x86 – Intel x86 interrupts management

SYNOPSIS

```
#include <dki/f_dki.h>
KnError svIntrAttach(CpuIntr intr, CpuIntrHandler intrHandler, void * intrCookie,
CpuIntrOps ** intrOps, CpuIntrId * intrId);

void svIntrDetach(CpuIntrId intrId);

KnError svIntrCtxGet(KnIntrCtx ** intrCtx);
typedef struct CpuIntrOps {
    void
  (*enable) (CpuIntrId intrId);
    void
  (*disable) (CpuIntrId intrId);
} CpuIntrOps;
```

FEATURES

DKI

DESCRIPTION

Provides Intel x86 interrupts management services.

EXTENDED DESCRIPTION

For Intel family processors, there are two kinds of interrupts:

Exceptions	Exceptions are generated and detected by the processor when excecuting instructions, such as divsion by 0. These exceptions are classified as faults, traps and aborts. Another type of exception exists: Programmed exceptions generated by INTx instructions.
Interrupts	Interrupts are generated by events external to the processor, such as requests to service peripheral devices.

There are two further types of interrupts:

Maskable	Maskable interrupts are received on the INTR input pin of the Intel x86 processor. They do not occur if the IF flag of the EFLAGS register is not set.
Non Maskable	Non maskable interrupts are received on the NMI input pin of the Intel x86 processor. There is no mechanism to prevent non maskable interrupts.

The DKI does not provide support for exceptions, only for interrupts. The processor associates an identifying number with each different interrupt (non maskable and maskable). This number is called a vector in the range from 0-255 with the range of 0-31 reserved for exceptions. The DKI associates an identifying numeric constant with each interrupt.

The following numeric constants are available:

- `NMI_INT` (non maskable).
- `INTR_BASE_NUMBER` (maskable)

The DKI provides support for 64 interrupts, ranging from INTR_BASE_NUMBER to INTR_BASE_NUMBER + 64.

The DKI provides services to allow host bus drivers to manage Intel ix86 interrupts, mainly to attach / detach handlers to / from these interrupts.

`svIntrAttach`

`svIntrAttach` attaches a handler to a given CPU interrupt.

The *intr* argument specifies the interrupt to which to attach.

The *intrHandler* argument specifies the handler to call back when the specified interrupt occurs.

The *intrCookie* argument specifies a parameter to pass back to the handler when called. It is opaque to the microkernel.

On succes, `K_OK` is returned and services defined on the attached interrupt are returned in the *intrOps* parameter. An identifier for the attached interrupt is also returned in *intrId* . This identifier must be used as the first parameter to further calls to *intrOps* services.

Services available on an attached interrupt are defined by the `CpuIntrOps` structure as follows:

The `CpuIntrOps.enable` routine enables the interrupt at CPU level identified by *intrId* .

The `CpuIntrOps.disable` routine disables the interrupt at CPU level identified by *intrId* .

Note that `NMI_INT` is a non maskable interrupt. Thus, `enable/disable` have no effect on these interrupts..

Note also that all other interrupts are masked using the same bit on an Intel x86 processor `EFLAGS` register. Thus there is no way to `enable/disable` a single interrupt separately from the others.

When an interrupt occurs, the attached *CpuIntrHandler* is invoked with interrupts masked at processor level. This produces the same effect as if `imsIntrMask_f()` was called just prior to invoking the handler (`imsIntrMaskCount_f` is positive). Two parameters are passed to this handler:

- Provided *intrCookie* parameter
- Detected *intr*

It is up to the called interrupt handler to use `enable` in order to allow or disallow interrupt nesting.

Typically, a host bus driver handler should handle x86 external interrupts as follows:

- Enable interrupts at processor level (`enable`).
- Call handler(s) attached to the identified cpu interrupt.
- Optionally `disable` interrupts in order to perform critical tasks, such as notifying the end an of interrupt to a PIC.
- Return to the DKI.

On failure, an error code is returned as follows:

`K_EINVAL`	The interrupt specified in *intr* is invalid (not in the list above). Or, the interrupt specified in *intr* is valid, but a handler is already attached to it.

`svIntrDetach`

`svIntrDetach` detaches an interrupt handler previously connected by `svIntrAttach` .

The *intrId* argument identifies the attached interrupt handler, and was previously returned by `svIntrAttach` .

`svIntrCtxGet`

`svIntrCtxGet` retrieves the current level interrupt context. It is mainly to be used for profiling purposes.

On success, `K_OK` is returned and a pointer to the current level of interrupt saved context is returned in the *intrCtx* argument. The CPU context saved on interrupt contains the interrupted PC composed of the `eip` and `cs` registers, the `EFLAGS` register, and the stack pointer.

On failure, an error code is returned as follows:

K_EINVAL	Interrupt level is zero (not called from an interrupt handler).
K_ENOTAVAILABLE	There is no context available for the currently handled interrupt.

Intel x86 Interrupts Management Allowed Calling Contexts

The table below specifies the contexts in which a caller is allowed to invoke each services.

Services	Base level	DKI thread	Interrupt	Blocking
svIntrAttach	+	+	-	+
CpuIntrOps.enable	+	+	+	-
CpuIntrOps.disable	+	+	+	-
svIntrDetach	+	+	-	+
svIntrCtxGet	-	-	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

svDkiThreadCall(9DKI)

NAME

svIntrAttach, svIntrDetach, svSoftIntrAttach, svSoftIntrDetach, svTimerIntrAttach, svTimerIntrDetach, svIntrCtxGet – interrupts management

FEATURES

DKI

DESCRIPTION

Provides interrupts management services.

See the architecture specific man pages:

- `svIntrAttach_usparc`(9DKI)
- `svIntrDetach_usparc`(9DKI)
- `svSoftIntrAttach_usparc`(9DKI)
- `svSoftIntrDetach_usparc`(9DKI)
- `svTimerIntrAttach_usparc`(9DKI)
- `svTimerIntrDetach_usparc`(9DKI)
- `svIntrCtxGet_usparc`(9DKI)
- `svIntrAttach_powerpc`(9DKI)
- `svIntrDetach_powerpc`(9DKI)
- `svIntrCtxGet_powerpc`(9DKI)
- `svIntrAttach_x86`(9DKI)
- `svIntrDetach_x86`(9DKI)
- `svIntrCtxGet_x86`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

svIntrAttach_powerpc, svIntrDetach_powerpc, svIntrCtxGet_powerpc – PowerPC interrupts management

SYNOPSIS

```
#include <dki/f_dki.h>
KnError svIntrAttach(CpuIntr intr, CpuIntrHandler intrHandler, void * intrCookie,
CpuIntrOps ** intrOps, CpuIntrId * intrId);

void svIntrDetach(CpuIntrId intrId);

KnError svIntrCtxGet(KnIntrCtx ** intrCtx);
typedef struct {
    void
  (*unmask) (CpuIntrId intrId);
    void
  (*mask) (CpuIntrId intrId);
} CpuIntrOps;
```

FEATURES

DKI

DESCRIPTION

Provides access to PowerPC DKI interrupts services.

EXTENDED DESCRIPTION

For PowerPC family processors, interrupts are defined as all asynchronous exceptions. These interrupts are identified using unsigned integer numbers which correspond to the exception number. The microkernel provides services to allow device drivers to manage PowerPC interrupts, mainly to attach / detach handlers to these interrupts.

`svIntrAttach` attaches a handler to a given CPU interrupt. The *intr* argument specifies the interrupt to attach to. The *intrHandler* argument specifies the handler to call back when the specified interrupt occurs. The *intrCookie* argument specifies a parameter to pass back to the handler when called. It is opaque to the microkernel.

On success, `K_OK` is returned and the services defined on the attached interrupt are returned in the *intrOps* parameter. An identifier for the attached interrupt is also returned in *intrId* . This identifier must be used as the first parameter to further calls to *intrOps* services.

Services available on an attached interrupt are defined by the `CpuIntrOps` structure as follows:

The `CpuIntrOps.unmask` routine enables the interrupt at CPU level identified by *intrId* .

The `CpuIntrOps.mask` routine disables the interrupt at CPU level identified by *intrId* .

Note that SYSTEM_RESET_INTR and MACHINE_CHECK_INTR are non-maskable interrupts. Thus unmask/mask have no effect on these interrupts.

Note also, that all other interrupts are masked using the same bit in the PowerPC processor MSR register. Thus there is no way to unmask/mask only one of these interrupts separately from the others. When an interrupt occurs, the attached *CpuIntrHandler* is invoked with interrupts masked at processor level. This behaves in the same way as if imsIntrMask_f() was just called prior to invoking the handler (imsIntrMaskCount_f is positive). It is up to the interrupt handler called to use unmask to allow the interrupt to be nested or not. Typically, a host bus driver handler should handle PowerPC external interrupts as follows:

- Identify the interrupt source (through a PIC or special cycle).
- Unmask interrupts at processor level (unmask).
- Call handlers attached to the identified source.
- Optionally mask interrupts to do critical tasks, such as notifying the end of the interrupt to a PIC.
- Return to the DKI.

On failure, an error code is returned as follows:

K_EINVAL	The interrupt specified in intr is invalid (not in the list above).
K_EFAIL	The interrupt specified in intr is valid, but a handler is already attached to it.
K_ENOMEM	The system is out of physical memory.

svIntrDetach detaches an interrupt handler previously connected by svIntrAttach . The *intrId* argument identifies the attached interrupt handler, and was previously returned by svIntrAttach .

svIntrCtxGet retrieves the current level interrupt context. It is mainly to be used for profiling purposes. On success, K_OK is returned and a pointer to the current level interrupt saved context is returned in the *intrCtx* argument. The CPU context saved on interrupt contains the volatile general purpose and floating point registers, condition register, Machine Status Register, instruction pointer, link register and the interrupt number itself. On failure, an error code is returned as follows:

K_EINVAL	The interrupt level is zero (not called from an interrupt handler.

K_ENOTAVAILABLE	There is no context available for the currently handled interrupt.

ALLOWED CALLING CONTEXTS

The following table specifies the contexts in which a caller is allowed to invoke each service.

Services	Base level	DKI thread	Interrupt	Blocking
svIntrAttach	+	+	-	+
CpuIntrOps.unmask	+	+	+	-
CpuIntrOps.mask	+	+	+	-
svIntrDetach	+	+	-	+
svIntrCtxGet	-	-	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`loadSwap`(9DKI) , `svDkiThreadCall`(9DKI) , `imsIntrMask_f`(9DKI)

NAME

svIntrAttach_usparc, svIntrDetach_usparc, svSoftIntrAttach_usparc, svSoftIntrDetach_usparc, svTimerIntrAttach_uparc, svTimerIntrDetach_uparc, svIntrCtxGet_usparc – UltraSPARC interrupts management

SYNOPSIS

#include <dki/f_dki.h>

KnError **svIntrAttach**(unsigned int *intrNumb*, unsigned int *intrLevel*, CpuIntrHandler *intrHandler*, void * *intrCookie*, CpuIntrOps ** *intrOps*, CpuIntrId * *intrIdp*);

void **svIntrDetach**(CpuIntrId *intrId*);

KnError **svSoftIntrAttach**(unsigned int *intrLevel*, CpuIntrHandler *intrHandler*, void * *intrCookie*, CpuIntrOps ** *intrOps*, CpuIntrId * *intrId*);

void **svSoftIntrDetach**(CpuIntrId *intrId*);

KnError **svTimerIntrAttach**(CpuIntrHandler *intrHandler*, void * *intrCookie*, CpuIntrOps ** *intrOps*, CpuIntrId * *intrId*);

void **svTimerIntrDetach**(CpuIntrId *intrId*);

KnError **svIntrCtxGet**(KnIntrCtx ** *intrCtx*);

FEATURES

DKI

DESCRIPTION

Provides UltraSPARC interrupts management services.

An UltraSPARC processor is an implementation of the SPARC-V9 CPU architecture. As specified by the SPARC-V9 architecture, there are fifteen CPU interrupt sources assigned to the fifteen CPU interrupt levels from 1 up to 15. However, the UltraSPARC processor uses these fifteen interrupt levels only for software generated interrupts. The hardware interrupts are delivered to the processor using the "Mondo" interrupt transfer mechanism. The hardware interrupt source is designated by the interrupt number. Typically, the interrupt number is 11-bit width and composed of the interrupt group number (5 MSB) and the interrupt offset number (6 LSB).

The "Mondo" interrupt dispatch handler is built into the microkernel. The microkernel handles a mapping between the hardware interrupt numbers and software interrupt levels. When a mondo interrupt packet is received by the microkernel, the interrupt request descriptor is queued and an associated software interrupt is triggered. The microkernel software interrupt handler then dequeues the interrupt request descriptor and invokes a handler associated to the given interrupt number. In this way, a driver interrupt handler is always invoked in the UltraSPARC software interrupt context. This provides the interrupt handler with an environment analogous to the SPARC-V8 one (interrupt levels).

The microkernel provides services which allow device drivers to manage UltraSPARC interrupts, mainly to attach/detach handlers to the CPU interrupts.

```
typedef void (*CpuIntrHandler)(void*);
```

svIntrAttach

`svIntrAttach` attaches a given handler to a given interrupt number at a given processor interrupt level.

The *intrNumb* argument specifies the interrupt number to which to attach.

The *intrLevel* argument specifies the interrupt level to which to attach.

The *intrHandler* argument specifies the handler to call back when the given interrupt occurs.

The *intrCookie* argument specifies a parameter to pass back to the handler when called. It is opaque to the microkernel.

On success, `K_OK` is returned and the services defined on the attached interrupt are returned in the *intrOps* parameter. An identifier for the attached interrupt is also returned in *intrId* . This identifier must be used as first argument to subsequent calls to *intrOps* services.

On failure, an error code is returned as follows:

`K_EINVAL`	The specified interrupt number or interrupt level are invalid.
`K_BUSY`	Another handler is already attached to the given *intrNumb* .
`K_ENOMEM`	The system is out of memory.

Services available on an attached interrupt are defined by the `CpuIntrOps` structure as follows:

```
typedef struct CpuIntrOps {

        void
    (*mask) (CpuIntrId intrId);

        void
    (*unmask) (CpuIntrId intrId);

        void
    (*enable) (CpuIntrId intrId);

        void
    (*disable) (CpuIntrId intrId);

        void
    (*trigger) (CpuIntrId intrId);

} CpuIntrOps;
```

The `CpuIntrOps.mask` routine disables the interrupt at CPU level identified by *intrId* . In other words, the PIL register is set to the level corresponding to *intrId* . Note that the original value of the PIL register is saved by DKI in order to be restored later by the `CpuIntrOps.unmask` routine.

The `CpuIntrOps.unmask` routine enables the interrupt at CPU level identified by *intrId* .

In other words, the PIL register is restored to the original value saved by the previously called `CpuIntrOps.mask` routine.

The `mask` / `unmask` pair may be called from base level only and must not be nested. The `mask/unmask` pair is typically used to implement a critical section of code which needs to be protected against the interrupt. Note that with respect to the SPARC-V9 architecture, when an interrupt level `N` is masked, all interrupts with a level less than `N` are also masked.Thus, there is no way to `mask` only one CPU interrupt level except the lowest one.

The `enable` / `disable` pair may only be called from the attached interrupt handler.When an interrupt occurs, the attached *CpuIntrHandler* is invoked with the interrupt masked at processor level. This behaves in exactly the same way as if `CpuIntrOps.disable` was called just prior to the handler invocation. In other words, the PIL register is set to the interrupt level and the original interrupt processor level (which was when the interrupt occurred) is saved by DKI. Note that the interrupt handler must return to DKI in the same context as it was called, that is with the interrupt disabled at processor level.

On the other hand, the called interrupt handler may use the `enable/disable` pair to allow the interrupt to be nested. This feature is typically used by a host bus driver when the bus interrupts are multiplexed, that is, multiple bus interrupts are reported at the same CPU interrupt level. Typically, an interrupt handler of this type of host bus driver would take the following actions:

- Identify the bus interrupt source (through a PIC or special cycle).
- Disable the bus interrupt source at bus level (through PIC).
- Enable interrupt at processor level (`enable`).
- Call handlers attached to the identified bus interrupt source.
- Disable interrupt at processor level (\ \command{disable}).
- Acknowledge (if needed) and enable the bus interrupt source at bus level (through PIC).
- Return to the DKI.

The `CpuIntrOps.trigger` routine allows the interrupt to be triggered by software. Basically, this routine acts like the mondo interrupt dispatcher except the interrupt number is obtained from *intrId* rather than from the mondo interrupt packet. The `CpuIntrOps.trigger` routine is mainly dedicated to the

software interrupts attached by `svSoftIntrAttach`. However, it may be also used for hardware interrupts, for instance, for debugging or diagnostic purposes.

svIntrDetach

`svIntrDetach` detaches an interrupt handler previously connected by `svIntrAttach`.

The *intrId* argument identifies the attached interrupt handler, previously returned by `svIntrAttach`.

svSoftIntrDetach

`svSoftIntrAttach` attaches a given software interrupt handler to a given processor interrupt level.

The *intrLevel* argument specifies the interrupt level to which to attach.

The *intrHandler* argument specifies the handler to call back when the given interrupt is triggered.

The *intrCookie* argument specifies a parameter to pass back to the handler when called. It is opaque to the microkernel.

On success, `K_OK` is returned and the services defined on the attached interrupt are returned in the *intrOps* parameter.

An identifier for the attached interrupt is also returned in *intrId*. This identifier must be used as the first argument to subsequent calls to *intrOps* services.

On failure, an error code is returned as follows:

`K_EINVAL`	The specified interrupt level is invalid.
`K_ENOMEM`	The system is out of memory.

Services available on an attached interrupt are defined by the `CpuIntrOps` structure described above.

svSoftIntrDetach

`svSoftIntrDetach` detaches a software interrupt handler previously connected by `svSoftIntrAttach`.

The *intrId* argument identifies the attached interrupt handler, previously returned by `svSoftIntrAttach`.

svTimerIntrAttach

`svTimerIntrAttach` attaches a given interrupt handler to the UltraSPARC tick-counter interrupt.

The *intrHandler* argument specifies the handler to call back when the given interrupt is triggered.

The *intrCookie* argument specifies a parameter to pass back to the handler when called. It is opaque the microkernel.

On success, K_OK is returned and the services defined on the attached interrupt are returned in the *intrOps* parameter. An identifier for the attached interrupt is also returned in *intrId* . This identifier must be used as the first argument to subsequent calls to *intrOps* services.

On failure, an error code is returned as follows:

K_BUSY	Another handler is already attached to the tick-counter interrupt.
K_ENOMEM	The system is out of memory.

Services available on an attached interrupt are defined by the CpuIntrOps structure described above.

svTimerIntrDetach

svTimerIntrDetach detaches the interrupt handler previously connected by svTimerIntrAttach .

The *intrId* argument identifies the attached interrupt handler, previously returned by svTimerIntrAttach .

svIntrCtxGet

svIntrCtxGet retrieves the current level interrupt context. It is typically used for profiling purposes.

On successs, K_OK is returned and a pointer to the recently saved interrupt context is returned in the *intrCtx* argument. The CPU context saved on interrupt has the same structure as the thread context saved on exception or trap. It contains the global registers %g1-%g7 , the output registers of the interrupted window %o0-%o7 and the following processor registers: %tstate , %pc , %npc , %tt , %y . In addition, the thread context contains the number of outstanding windows and the pointer to the outstanding windows buffer if the number of windows is greater than zero.

On failure, an error code is returned as follows:

K_EINVAL	Interrupt level is zero (not called from an interrupt handler).
K_ENOTAVAILABLE	There is no context available for currently handled interrupt.

ALLOWED CALLING CONTEXTS

The following table specifies the contexts in which a caller is allowed to invoke each interrupt management service.

Services	Base level	DKI thread	Interrupt	Blocking
svIntrAttach	+	+	-	+

svIntrDetach	+	+	-	+
svIntrSoftAttach	+	+	-	+
svIntrSoftDetach	+	+	-	+
svTimerIntrAttach	+	+	-	+
svTimerIntrDetach	+	+	-	+
svIntrCtxGet	-	-	+	-
CpuIntrOps.mask	+	+	-	-
CpuIntrOps.unmask	+	+	-	-
CpuIntrOps.enable	-	-	+	-
CpuIntrOps.disable	-	-	+	-
CpuIntrOps.trigger	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

svIntrAttach_x86, svIntrDetach_x86, svIntrCtxGet_x86 – Intel x86 interrupts management

SYNOPSIS

```
#include <dki/f_dki.h>
KnError svIntrAttach(CpuIntr intr, CpuIntrHandler intrHandler, void * intrCookie,
CpuIntrOps ** intrOps, CpuIntrId * intrId);

void svIntrDetach(CpuIntrId intrId);

KnError svIntrCtxGet(KnIntrCtx ** intrCtx);
typedef struct CpuIntrOps {
    void
  (*enable) (CpuIntrId intrId);
    void
  (*disable) (CpuIntrId intrId);
} CpuIntrOps;
```

FEATURES

DKI

DESCRIPTION

Provides Intel x86 interrupts management services.

EXTENDED DESCRIPTION

For Intel family processors, there are two kinds of interrupts:

Exceptions	Exceptions are generated and detected by the processor when excecuting instructions, such as divsion by 0. These exceptions are classified as faults, traps and aborts. Another type of exception exists: Programmed exceptions generated by INTx instructions.
Interrupts	Interrupts are generated by events external to the processor, such as requests to service peripheral devices.

There are two further types of interrupts:

Maskable	Maskable interrupts are received on the INTR input pin of the Intel x86 processor. They do not occur if the IF flag of the EFLAGS register is not set.
Non Maskable	Non maskable interrupts are received on the NMI input pin of the Intel x86 processor. There is no mechanism to prevent non maskable interrupts.

The DKI does not provide support for exceptions, only for interrupts. The processor associates an identifying number with each different interrupt (non maskable and maskable). This number is called a vector in the range from 0-255 with the range of 0-31 reserved for exceptions. The DKI associates an identifying numeric constant with each interrupt.

The following numeric constants are available:

- `NMI_INT` (non maskable).
- `INTR_BASE_NUMBER` (maskable)

The DKI provides support for 64 interrupts, ranging from INTR_BASE_NUMBER to INTR_BASE_NUMBER + 64.

The DKI provides services to allow host bus drivers to manage Intel ix86 interrupts, mainly to attach / detach handlers to / from these interrupts.

`svIntrAttach`

`svIntrAttach` attaches a handler to a given CPU interrupt.

The *intr* argument specifies the interrupt to which to attach.

The *intrHandler* argument specifies the handler to call back when the specified interrupt occurs.

The *intrCookie* argument specifies a parameter to pass back to the handler when called. It is opaque to the microkernel.

On succes, `K_OK` is returned and services defined on the attached interrupt are returned in the *intrOps* parameter. An identifier for the attached interrupt is also returned in *intrId* . This identifier must be used as the first parameter to further calls to *intrOps* services.

Services available on an attached interrupt are defined by the `CpuIntrOps` structure as follows:

The `CpuIntrOps.enable` routine enables the interrupt at CPU level identified by *intrId* .

The `CpuIntrOps.disable` routine disables the interrupt at CPU level identified by *intrId* .

Note that `NMI_INT` is a non maskable interrupt. Thus, `enable/disable` have no effect on these interrupts..

Note also that all other interrupts are masked using the same bit on an Intel x86 processor `EFLAGS` register. Thus there is no way to `enable/disable` a single interrupt separately from the others.

When an interrupt occurs, the attached *CpuIntrHandler* is invoked with interrupts masked at processor level. This produces the same effect as if `imsIntrMask_f()` was called just prior to invoking the handler (`imsIntrMaskCount_f` is positive). Two parameters are passed to this handler:

- Provided *intrCookie* parameter
- Detected *intr*

It is up to the called interrupt handler to use `enable` in order to allow or disallow interrupt nesting.

Typically, a host bus driver handler should handle x86 external interrupts as follows:

- Enable interrupts at processor level (`enable`).
- Call handler(s) attached to the identified cpu interrupt.
- Optionally `disable` interrupts in order to perform critical tasks, such as notifying the end an of interrupt to a PIC.
- Return to the DKI.

On failure, an error code is returned as follows:

`K_EINVAL`	The interrupt specified in *intr* is invalid (not in the list above). Or, the interrupt specified in *intr* is valid, but a handler is already attached to it.

`svIntrDetach`

`svIntrDetach` detaches an interrupt handler previously connected by `svIntrAttach` .

The *intrId* argument identifies the attached interrupt handler, and was previously returned by `svIntrAttach` .

`svIntrCtxGet`

`svIntrCtxGet` retrieves the current level interrupt context. It is mainly to be used for profiling purposes.

On success, `K_OK` is returned and a pointer to the current level of interrupt saved context is returned in the *intrCtx* argument. The CPU context saved on interrupt contains the interrupted PC composed of the `eip` and `cs` registers, the `EFLAGS` register, and the stack pointer.

On failure, an error code is returned as follows:

K_EINVAL	Interrupt level is zero (not called from an interrupt handler).
K_ENOTAVAILABLE	There is no context available for the currently handled interrupt.

Intel x86 Interrupts Management Allowed Calling Contexts

The table below specifies the contexts in which a caller is allowed to invoke each services.

Services	Base level	DKI thread	Interrupt	Blocking
svIntrAttach	+	+	-	+
CpuIntrOps.enable	+	+	+	-
CpuIntrOps.disable	+	+	+	-
svIntrDetach	+	+	-	+
svIntrCtxGet	-	-	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

svDkiThreadCall(9DKI)

NAME

svMemAlloc, svMemFree – A general purpose memory allocator

SYNOPSIS

#include <dki/dki.h>
void * **svMemAlloc**(unsigned int *size*);

void * **svMemFree**(void * *mem*, unsigned int *size*);

FEATURES

DKI

DESCRIPTION

Provides general purpose memory allocation.

EXTENDED DESCRIPTION

The microkernel provides general purpose memory management services for device drivers that need to allocate and free sized pieces of memory dynamically in a supervisor address space. As the initialization scheme is mainly dynamic, device drivers need to dynamically allocate/free small pieces of supervisor data. Typically, a device driver needs to dynamically allocate data associated with each instance to be registered in the Device Registry at initialization time.

Moreover, most of the DDI services called from base level by the driver clients leads to dynamic allocation/freeing of a number of linked list elements for internal management purposes. Note that the memory allocated using these services is anonymous; it is not associated with any actor context. For this reason, all the memory allocated must be explicitly freed by the drivers before they terminate, as the kernel cannot do it at actor deletion time.

`svMemAlloc` allocates memory of a given size which is accessible within the supervisor address space. The size argument specifies the required memory size. In case of success, `svMemAlloc` returns a pointer to the supervisor memory allocated, otherwise a `NULL` pointer is returned. The allocated memory returned by `svMemAlloc` is suitably aligned for any use.

The `svMemFree` routine frees memory previously allocated using `svMemAlloc` . The *mem* argument specifies the pointer to the memory previously returned by `svMemAlloc` . The *size* argument must be exactly the same as the size required when previously allocating the memory using `svMemAlloc` .

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service.

Services	Base level	DKI thread	Interrupt	Blocking
svMemAlloc	+	+	-	+
svMemFree	+	+	-	+

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

svMemAlloc, svMemFree – A general purpose memory allocator

SYNOPSIS

```
#include <dki/dki.h>
void * svMemAlloc(unsigned int size);

void * svMemFree(void * mem, unsigned int size);
```

FEATURES

DKI

DESCRIPTION

Provides general purpose memory allocation.

EXTENDED DESCRIPTION

The microkernel provides general purpose memory management services for device drivers that need to allocate and free sized pieces of memory dynamically in a supervisor address space. As the initialization scheme is mainly dynamic, device drivers need to dynamically allocate/free small pieces of supervisor data. Typically, a device driver needs to dynamically allocate data associated with each instance to be registered in the Device Registry at initialization time.

Moreover, most of the DDI services called from base level by the driver clients leads to dynamic allocation/freeing of a number of linked list elements for internal management purposes. Note that the memory allocated using these services is anonymous; it is not associated with any actor context. For this reason, all the memory allocated must be explicitly freed by the drivers before they terminate, as the kernel cannot do it at actor deletion time.

`svMemAlloc` allocates memory of a given size which is accessible within the supervisor address space. The size argument specifies the required memory size. In case of success, `svMemAlloc` returns a pointer to the supervisor memory allocated, otherwise a `NULL` pointer is returned. The allocated memory returned by `svMemAlloc` is suitably aligned for any use.

The `svMemFree` routine frees memory previously allocated using `svMemAlloc` . The *mem* argument specifies the pointer to the memory previously returned by `svMemAlloc` . The *size* argument must be exactly the same as the size required when previously allocating the memory using `svMemAlloc` .

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service.

Services	Base level	DKI thread	Interrupt	Blocking
svMemAlloc	+	+	-	+
svMemFree	+	+	-	+

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

svPhysAlloc, svPhysFree – A special purpose physical memory allocator

SYNOPSIS

```
#include <dki/dki.h>
KnError svPhysAlloc(KnPhMemChunk * chunk, KnPhMemAlign * align);

void * svPhysFree(KnPhMemChunk * chunk);
typedef struct {
  PhAddr  paddr;
  PhSize  psize;
  VmAddr  vaddr;
} KnPhMemChunk;

typedef struct {
  PhAddr  alignment;
  PhAddr  floating;
} KnPhMemAlign;
```

FEATURES

DKI

DESCRIPTION

Provides special purpose physical memory allocation.

EXTENDED DESCRIPTION

In order to satisfy all physical memory constraints imposed by the different I/O buses, mainly for DMA purposes, the DKI provides an interface to allocate and free special purpose physical memory that satisfies the given constraints.

Typically, different I/O buses may impose different constraints on the memory used by their devices for Direct Memory Access, such as alignement, specific boundary crossing, maximum size, or specific locations within the physical memory space.

`svPhysAlloc` allocates an amount of contiguous physical memory that satisfies the required constraints. The *chunk* argument points to the `KnPhMemChunk` structure. The *psize* field is an input argument specifying the size in bytes of the memory to be allocated. The *paddr* field is an input/output argument specifying the physical start address of the allocated memory chunk.

The *vaddr* field is not used by `svPhysAlloc` .

The *align* argument points to the `KnPhMemAlign` structure.

The `KnPhMemAlign` structure specifies the constraints on the memory chunk being allocated.

The *alignment* field is a mask which specifies constraints on the start address of the memory chunk being allocated. A bit set within *alignment* specifies that the corresponding bit within the start address may take any value, i.e. there are

no specific constraints on that bit. A bit cleared within *alignment* specifies that the corresponding bit within the start address must take the same value as the corresponding bit of the *paddr* field. This mask allows the caller to specify an alignment for the start of the allocated memory, by zeroing the required number of least significant bits. By resetting the required number of most significant bits, the caller may also indicate which part of the physical space the memory should be allocated to.

The *floating* field is a mask which indicates which bits of the returned address can vary while walking through the allocated memory for the required size. In other words, bits cleared in the mask must be constant for all addresses in the range of the allocated memory. This mask may be used to specify that the amount of memory allocated must not span across a given address boundary.

Note that the *align* argument may be set to `NULL` specifying that there are no constraints on the memory chunks allocated. In case of success, the starting address of the allocated memory is returned in the *paddr* field and the function returns `K_OK` . The returned address is aligned on a page boundary and is therefore suitably aligned to be used subsequently in mapping services. The size of effectively allocated memory is rounded up to the next page boundary.

On failure, an error code is returned as follows:

K_ENOMEM	There is no contiguous physical memory which satisfies the requirements specified by both significant and floating arguments.
K_ESIZE	The size argument is equal to zero.

`svPhysFree` releases physical memory previously allocated with `svPhysAlloc` .

The *chunk* argument points to the `KnPhMemChunk` structure which specifies the physical chunk start addresses and the chunk size.

Note - The *psize* field must have the value previously specified in `svPhysAlloc` . The *paddr* field must have the value previously returned by `svPhysAlloc` .

Typically, a driver uses the same `KnPhMemChunk` structure for the `svPhysAlloc` and `svPhysFree` calls and the structure fields are not modified by the driver once `svPhysAlloc` has been done.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service.

Services	Base level	DKI thread	Interrupt	Blocking
svPhysAlloc	+	+	-	+
svPhysFree	+	+	-	+

EXAMPLES

As an example, to allocate a 16KB contiguous memory region in the first Megabyte of physical space, that is aligned on a 4KB address, which does not span a 64KB boundary, the call should look like:

```
KnPhMemChunk chunk;
KnPhMemAlign align;

chunk.psize     = 0x00004000;          /* psize     = 16KB   */
align.alignment = 0x000ff000;          /* alignment = (1024KB-1) & ~(4KB-1) */
align.floating  = 0x0000ffff;          /* floating  = 64KB-1 */

res = svPhysAlloc(&chunk, &align);
```

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

svPhysAlloc, svPhysFree – A special purpose physical memory allocator

SYNOPSIS

```
#include <dki/dki.h>
KnError svPhysAlloc(KnPhMemChunk * chunk, KnPhMemAlign * align);

void * svPhysFree(KnPhMemChunk * chunk);

typedef struct {
  PhAddr  paddr;
  PhSize  psize;
  VmAddr  vaddr;
} KnPhMemChunk;

typedef struct {
  PhAddr  alignment;
  PhAddr  floating;
} KnPhMemAlign;
```

FEATURES

DKI

DESCRIPTION

Provides special purpose physical memory allocation.

EXTENDED DESCRIPTION

In order to satisfy all physical memory constraints imposed by the different I/O buses, mainly for DMA purposes, the DKI provides an interface to allocate and free special purpose physical memory that satisfies the given constraints.

Typically, different I/O buses may impose different constraints on the memory used by their devices for Direct Memory Access, such as alignement, specific boundary crossing, maximum size, or specific locations within the physical memory space.

`svPhysAlloc` allocates an amount of contiguous physical memory that satisfies the required constraints. The *chunk* argument points to the `KnPhMemChunk` structure. The *psize* field is an input argument specifying the size in bytes of the memory to be allocated. The *paddr* field is an input/output argument specifying the physical start address of the allocated memory chunk.

The *vaddr* field is not used by `svPhysAlloc` .

The *align* argument points to the `KnPhMemAlign` structure.

The `KnPhMemAlign` structure specifies the constraints on the memory chunk being allocated.

The *alignment* field is a mask which specifies constraints on the start address of the memory chunk being allocated. A bit set within *alignment* specifies that the corresponding bit within the start address may take any value, i.e. there are

no specific constraints on that bit. A bit cleared within *alignment* specifies that the corresponding bit within the start address must take the same value as the corresponding bit of the *paddr* field. This mask allows the caller to specify an alignment for the start of the allocated memory, by zeroing the required number of least significant bits. By resetting the required number of most significant bits, the caller may also indicate which part of the physical space the memory should be allocated to.

The *floating* field is a mask which indicates which bits of the returned address can vary while walking through the allocated memory for the required size. In other words, bits cleared in the mask must be constant for all addresses in the range of the allocated memory. This mask may be used to specify that the amount of memory allocated must not span across a given address boundary.

Note that the *align* argument may be set to `NULL` specifying that there are no constraints on the memory chunks allocated. In case of success, the starting address of the allocated memory is returned in the *paddr* field and the function returns `K_OK` . The returned address is aligned on a page boundary and is therefore suitably aligned to be used subsequently in mapping services. The size of effectively allocated memory is rounded up to the next page boundary.

On failure, an error code is returned as follows:

K_ENOMEM	There is no contiguous physical memory which satisfies the requirements specified by both significant and floating arguments.
K_ESIZE	The size argument is equal to zero.

`svPhysFree` releases physical memory previously allocated with `svPhysAlloc` .

The *chunk* argument points to the `KnPhMemChunk` structure which specifies the physical chunk start addresses and the chunk size.

Note - The *psize* field must have the value previously specified in `svPhysAlloc` . The *paddr* field must have the value previously returned by `svPhysAlloc` .

Typically, a driver uses the same `KnPhMemChunk` structure for the `svPhysAlloc` and `svPhysFree` calls and the structure fields are not modified by the driver once `svPhysAlloc` has been done.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service.

Services	Base level	DKI thread	Interrupt	Blocking
svPhysAlloc	+	+	-	+
svPhysFree	+	+	-	+

EXAMPLES

As an example, to allocate a 16KB contiguous memory region in the first Megabyte of physical space, that is aligned on a 4KB address, which does not span a 64KB boundary, the call should look like:

```
KnPhMemChunk chunk;
KnPhMemAlign align;

chunk.psize     = 0x00004000;          /* psize     = 16KB   */
align.alignment = 0x000ff000;          /* alignment = (1024KB-1) & ~(4KB-1) */
align.floating  = 0x0000ffff;          /* floating  = 64KB-1 */

res = svPhysAlloc(&chunk, &align);
```

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

svPhysMap, svPhysUnmap, vmMapToPhys – physical to virtual memory mapping

FEATURES

DKI

DESCRIPTION

The microkernel provides services to allow device drivers to map physical space to virtual memory space. This services should be used mainly by primary bus drivers to map bus I/O space, or DMA memory.

See the architecture specific man pages:

- `svPhysMap_usparc`(9DKI)
- `svPhysUnmap_usparc`(9DKI)
- `vmMapToPhys_usparc`(9DKI)
- `svPhysMap_powerpc`(9DKI)
- `svPhysUnmap_powerpc`(9DKI)
- `vmMapToPhys_powerpc`(9DKI)
- `svPhysMap_x86`(9DKI)
- `svPhysUnmap_x86`(9DKI)
- `vmMapToPhys_x86`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

svPhysMap_powerpc, svPhysUnmap_powerpc, vmMapToPhys_powerpc – PowerPC physical to virtual memory mapping

Synopsis

```
#include <dki/dki.h>
KnError svPhysMap(KnPhMemChunk * chunk, PteCntlBits cntlBits);

void * svPhysUnmap(KnPhMemChunk * chunk);

KnError vmMapToPhys(KnCap * actor, KnPhMemChunk * chunk, PteCntlBits cntlBits);
typedef struct {
  PhAddr paddr;   /* physical start addree */
  PhSize psize;   /* size */
  VmAddr vaddr;   /* virtual start address */
} KnPhMemChunk;
```

FEATURES

DKI

DESCRIPTION

Provides physical to virtual memory mapping services.

EXTENDED DESCRIPTION

The microkernel provides services to allow device drivers to map physical space to virtual memory space. These services should be used mainly by primary bus drivers to map bus I/O space or DMA memory.

`svPhysMap` allocates a range of virtual addresses within the supervisor address space and maps it to a given range of physical addresses.

The *chunk* argument points to the `KnPhMemChunk` structure.

The *paddr* and *psize* fields specify the chunk of physical memory being mapped to.

The *vaddr* field is used by `svPhysMap` to return the virtual address to which the physical one is mapped.

For PowerPC family processors, the *cntlBits* argument should be constructed by 'oring' the following values:

PTE_READ_WRITE	The mapped memory allows you to perform write accesses.
PTE_CACHE_WRITE_THROUGH	The memory is mapped with Write-Through attribute (bit `W`). Thus a store operation updates the cache if necessary, and in addition the update is written to memory.
PTE_CACHE_DISABLE	The memory is mapped with Cache-Inhibited attribute (bit I). Thus

	accesses are made to main memory, bypassing the caches.
PTE_MEMORY_COHERENCY	The memory is mapped with Memory-Coherency attribute (bit M). This enforce coherency of memory shared between processors in a system. When performing an access to memory, there is a hardware indication to the rest of the system that the access is global. Other processors affected by the access must then respond to this global access. Typically, this is used for a snooping bus design.
PTE_MEMORY_GUARDED	The memory is mapped with Memory-Guarded attribute (bit G). This prevents the processor from making out-of-order access to that memory (that is, access not directly dictated by the program). This may be useful if there are holes in physical memory, or to prevent these accesses to certain peripheral devices.

If *cntlBits* is equal to zero, the memory is mapped as read-only, not guarded, with cache enabled in write-back mode, and no coherency is enforced.

Note - Any combination where PTE_CACHE_WRITE_THROUGH and PTE_CACHE_DISABLE are both set is not supported.

On success K_OK is returned, otherwise a negative error code is returned:

K_ENOMEM	The system is out of memory.
K_ESIZE	The *psize* argument is equal to zero.

svPhysUnmap unmaps the physical memory chunk previously mapped by svPhysMap . The *chunk* argument points to the KnPhMemChunk structure which specifies the virtual and physical chunk start addresses and the chunk size. Note that the *paddr* and *psize* fields must have the values previously specified in svPhysMap . The *vaddr* field must have the value previously returned by svPhysMap .

Note - Typically, a driver uses the same KnPhMemChunk structure for the svPhysMap and svPhysUnmap calls and the structure fields are not modified by the driver once svPhysMap has been done.

vmMapToPhys maps a given physical memory chunk to the target actor address space. The *actor* argument specifies the target actor capability.

If *actor* is K_MYACTOR , the address space of the current actor is used.

If *actor* is K_SVACTOR , the supervisor address space is used.

The *chunk* argument points to the KnPhMemChunk structure which specifies the physical and virtual chunk start addresses and the chunk size.

The specified virtual address range must be allocated using the K_RESERVED option (see rgnAllocate(2)) prior to the invocation of vmMapToPhys .

The specified actor can be a supervisor actor as well as a user actor.

The mapping produced by vmMapToPhys can only be invalidated by rgnFree . *vaddr* , *paddr* and *psize* must be page-aligned.

For PowerPC family processors, the *cntlBits* is defined in the same way as for the svPhysMap routine above.

On success K_OK is returned, otherwise a negative error code is returned:

K_EFAULT	The *actor* argument points outside the caller's address space.
K_EINVAL	An inconsistent actor capability was provided.
K_EUNKNOWN	*actorcap* does not specify a reachable actor.
K_ENOMEM	The system is out of memory.
K_ESIZE	The *psize* argument is equal to zero.
K_EROUND	*vaddr* or *paddr* or *psize* is not page-aligned.
K_EADDR	Some or all addresses from the target virtual address range are out of a region allocated with the K_RESERVED option.

Note - For performance reasons, the current implementation does not guarantee that any attempt to map a physical address to a virtual address out of a region allocated with K_RESERVED option would effectively produce a K_EADDR error.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service.

Services	Base level	DKI thread	Interrupt	Blocking
svPhysMap	+	+	-	+
svPhysUnmap	+	+	-	+
vmMapToPhys	+	+	-	+

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svPhysAlloc`(9DKI) , `svDkiThreadCall`(9DKI) , `rgnAllocate`(2K)

NAME

svPhysMap_usparc, svPhysUnmap_usparc, vmMapToPhys_usparc – UltraSPARC physical to virtual memory mapping

SYNOPSIS

```
#include <dki/f_dki.h>
KnError svPhysMap(KnPhMemChunk * chunk, KnPteAttr attr);

void svPhysUnmap(KnPhMemChunk * chunk);

KnError vmMapToPhys(KnCap * actor, KnPhMemChunk * chunk, KnPteAttr attr);
```

FEATURES

DKI

DESCRIPTION

The microkernel provides services to allow device drivers to map physical space to virtual memory space. These services should be used mainly by a host bus driver to map bus I/O space or DMA memory.

svPhysMap

svPhysMap allocates a range of virtual addresses within the supervisor address space and maps it to a given range of physical addresses.

The *chunk* argument points to the KnPhMemChunk structure shown below:

```
typedef struct {
    PhAddr  paddr;    /* physical start addree */
    PhSize  psize;    /* size */
    VmAddr  vaddr;    /* virtual start address */
} KnPhMemChunk;
```

The *paddr* and *psize* fields specify the chunk of physical memory being mapped to.

The *vaddr* field is used by svPhysMap to return the virtual address to which the physical one is mapped.

The *attr* argument specifies the mapping attributes. *attr* is a bit-mask composed of two independent parts:

Data access attributes

Instruction access attributes

Basically, each part of the attributes is a sub-set of bits defined by the translation table entry (TTE) of the UltraSPARC MMU.

A combination of the following attributes may be specified for data access:

PTE_DATTR_G — global
The PTE_DATTR_G bit set allows the mapping to be shared among all (user and supervisor) contexts.

PTE_DATTR_W — writable
The PTE_DATTR_W bit set grants write permission for the mapping.

PTE_DATTR_P — privileged

The PTE_DATTR_P bit set restricts access to the mapping for the supervisor only.

PTE_DATTR_E — side-effect
The PTE_DATTR_E bit set makes noncacheable memory accesses to be strongly ordered against other E-bit accesses, and noncacheable stores are not merged. This bit should be set for I/O devices having side-effects. Note that the E-bit does not force an uncacheable access. It is expected that the PTE_DATTR_CV and PTE_DATTR_CP bits will be set to zero when the E-bit is set.

PTE_DATTR_CV — L1-cacheable
The PTE_DATTR_CV bit set allows data to be cached in the (L1) CPU data cache. Note that if the PTE_DATTR_CV bit is set, the PTE_DATTR_CP bit must also be set.

PTE_DATTR_CP — L2-cacheable
The PTE_DATTR_CP bit set allows data to be cached in the (L2) external cache.

PTE_DATTR_IE — invert endianness
The PTE_DATTR_IE bit set causes data accesses to the mapping to be processed with inverse endianness from that specified by the instruction.

PTE_DATTR_V — valid
The PTE_DATTR_V bit set enables data accesses to the mapping. If this bit is not set, all other bits (described above) are ignored and a data acess to the mapping will result in a data access exception.

Combinations of the following attributes may be specified for instruction access:

PTE_IATTR_G — global
The PTE_IATTR_G bit set allows the mapping to be shared among all (user and supervisor) contexts.

PTE_IATTR_P — privileged
The PTE_IATTR_P bit set restricts access to the mapping to the supervisor only.

PTE_IATTR_CV — L1-cacheable
The PTE_IATTR_CV bit set allows instructions to be cached in the (L1) CPU instruction cache. Note that if the PTE_IATTR_CV bit is set, the PTE_IATTR_CP bit must also be set.

PTE_IATTR_CP — L2-cacheable
The PTE_IATTR_CP bit set allows instructions to be cached in the (L2) external cache.

PTE_IATTR_V — valid
The PTE_IATTR_V bit set enables instructions to be obtained from the mapping. If this bit is not set, all other bits (described above) are ignored and an instruction to fetch from the mapping will result in an instruction access exception.

On success svPhysMap returns K_OK , otherwise a negative error code is returned as follows:

K_ENOMEM	The system is out of memory.
K_ESIZE	The *psize* argument is equal to zero.

svPhysUnmap

svPhysUnmap unmaps the physical memory chunk previously mapped by svPhysMap . The *chunk* argument points to the KnPhMemChunk structure which specifies the virtual and physical chunk start addresses and the chunk size. Note that the *paddr* and *psize* fields must have the values previously specified in svPhysMap . The *vaddr* fields must have the value previously returned by svPhysMap . Typically, a driver uses the same KnPhMemChunk structure for the svPhysMap and svPhysUnmap calls and the structure fields are not modified by the driver once svPhysMap has bee performed.

vmMapToPhys

vmMapToPhys maps a given physical memory chunk to the target actor address space.

The *actor* argument specifies the target actor capability. If *actor* is K_MYACTOR , the address space of the current actor is used. If *actor* is K_SVACTOR , the supervisor address space is used.

The *chunk* argument points to the KnPhMemChunk structure which specifies the physical and virtual chunk start addresses and the chunk size. The specified virtual address range must be allocated using the K_RESERVED option (see *rgnAllocate(2K)* prior to the invocation of vmMapToPhys . The specified actor can be a supervisor actor as well as an user actor. The mapping produced by vmMapToPhys can only be invalidated by rgnFree .

vaddr , *paddr* and *psize* must be page-aligned.

For UltraSPARC family processors, the *attr* argument is defined in the same way as for the svPhysMap routine above.

On success K_OK is returned, otherwise a negative error code is returned as follows:

K_EFAULT	The *actor* argument points outside of the caller's address space.

K_EINVAL	An inconsistent actor capability was specified.
K_EUNKNOWN	*actorcap* does not specify a reachable actor.
K_ENOMEM	The system is out of memory.
K_ESIZE	The *psize* argument is equal to zero.
K_EROUND	*vaddr* or *paddr* or *psize* is not page-aligned.
K_EADDR	Some or all addresses from the target virual address range are out of a region allocated with the K_RESERVED option. For performance reasons, the current implementation does not guarantee that any attempt to map a physical address to a virtual address outside a region allocated with K_RESERVED option would effectively produce a K_EADDR error.

ALLOWED CALLING CONTEXTS

The following table specifies the contexts in which a caller is allowed to invoke each service.

Services	Base level	DKI thread	Interrupt	Blocking
svPhysMap	+	+	-	+
svPhysUnmap	+	+	-	+
vmMapToPhys	+	+	-	+

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

svPhysAlloc(9DKI) , svDkiThreadCall(9DKI) , rgnAllocate(2K)

NAME

svPhysMap_x86, svPhysUnmap_x86, vmMapToPhys_x86 – Intel x86 physical to virtual memory mapping

SYNOPSIS

```
#include <dki/f_dki.h>
KnError svPhysMap(KnPhMemChunk * chunk, PteCntlBits cntlBits);

void svPhysUnmap(KnPhMemChunk * chunk);

KnError vmMapToPhys(KnCap * actor, KnPhMemChunk * chunk, PteCntlBits cntlBits);

typedef struct {
  PhAddr  paddr;   /* physical start address */
  PhSize  psize;   /* size */
  VmAddr  vaddr;   /* virtual start address */
} KnPhMemChunk;
```

FEATURES

DKI

EXTENDED DESCRIPTION

The microkernel provides services to allow device drivers to map physical space to virtual memory space. These services should be used mainly by primary bus drivers to map bus I/O space, or DMA memory.

`svPhysMap`

`svPhysMap` allocates a range of virtual addresses within the supervisor address space and maps it to a given range of physical addresses.

The *chunk* argument points to the `KnPhMemChunk` structure.

The *paddr* and *psize* fields specify the chunk of physical memory being mapped to.

The *vaddr* field is used by `svPhysMap` to return the virtual address to which the physical one is mapped.

For Intel ix86 family processor, the cntlBits argument is a direct mapping of the Page Control bits of Page Table Entries, as described in the Intel 386/486 Programming Reference Manual. Therefore, the cntlBits argument should be constructed by “oring” the following values:

`PTE_PRESENT`	The mapped memory must be present in physical memory.
`PTE_READ_WRITE`	The mapped memory allows write accesses to be performed.
`PTE_USER_SUPERVISOR`	The mapped memory can be used in user space.
`PTE_WRITE_TRANSPARENT`	The memory is mapped with the Write-Through attribute (bit PWT). Thus, a store operation updates the cache if necessary, and in addition the update is written to memory.

PTE_CACHE_DISABLE	The memory is mapped with caching disabled (bit PCD). Thus, memory accesses go to main memory, bypassing the caches.

On success K_OK is returned, otherwise a negative error code is returned:

K_ENOMEM	The system is out of memory.
K_ESIZE	The *psize* argument is equal to zero.

svPhysUnmap

svPhysUnmap unmaps the physical memory chunk previously mapped by svPhysMap . The *chunk* argument points to the KnPhMemChunk structure which specifies the virtual and physical chunk start addresses and the chunk size. Note that the *paddr* and *psize* fields must have the values previously specified in svPhysMap . The *vaddr* field must have the value previously returned by svPhysMap (typically, a driver uses the same KnPhMemChunk structure for the svPhysMap and svPhysUnmap calls and the structure fields are not modified by the driver once svPhysMap is done).

vmMapToPhys

vmMapToPhys maps a given physical memory chunk to the target actor address space.

The *actor* argument specifies the target actor capability. If *actor* is K_MYACTOR , the address space of the current actor is used. If *actor* is K_SVACTOR , the supervisor address space is used.

The *chunk* argument points to the KnPhMemChunk structure which specifies the physical and virtual chunk start addresses and the chunk size. The specified virtual address range must be allocated using the K_RESERVED option (see *rgnAllocate* (2K)) prior to the invocation of vmMapToPhys . The specified actor can be a supervisor actor as well as a user actor. The mapping produced by vmMapToPhys can only be invalidated by rgnFree .

vaddr , *paddr* and *psize* must be page-aligned.

For Intel x86 family processors, the *cntlBits* is defined in the same way as for the svPhysMap routine above.

On success K_OK is returned, otherwise a negative error code is returned:

K_EFAULT	The *actor* argument points to the outside of the caller's address space.
K_EINVAL	An inconsistent actor capability was provided.

K_EUNKNOWN	*actorcap* does not specify a reachable actor.
K_ENOMEM	The system is out of memory.
K_ESIZE	The *psize* argument is equal to zero.
K_EROUND	*vaddr* or *paddr* or *psize* is not page-aligned.
K_EADDR	Some or all addresses from the target virual address range are out of a region allocated with the K_RESERVED option (for performance reasons the current implementation does not guarantee that any attempt to map a physical address to a virtual address out of a region allocated with K_RESERVED option would effectively produce a K_EADDR error).

Intel x86 Memory Mapping Allowed Calling Contexts

The table below specifies the contexts in which a caller is allowed to invoke each service.

Services	Base level	DKI thread	Interrupt	Blocking
svPhysMap	+	+	-	+
svPhysUnmap	+	+	-	+
vmMapToPhys	+	+	-	+

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

svPhysAlloc(9DKI) , svDkiThreadCall(9DKI) , rgnAllocate(9DKI)

NAME

svPhysMap, svPhysUnmap, vmMapToPhys – physical to virtual memory mapping

FEATURES

DKI

DESCRIPTION

The microkernel provides services to allow device drivers to map physical space to virtual memory space. This services should be used mainly by primary bus drivers to map bus I/O space, or DMA memory.

See the architecture specific man pages:

- `svPhysMap_usparc`(9DKI)
- `svPhysUnmap_usparc`(9DKI)
- `vmMapToPhys_usparc`(9DKI)
- `svPhysMap_powerpc`(9DKI)
- `svPhysUnmap_powerpc`(9DKI)
- `vmMapToPhys_powerpc`(9DKI)
- `svPhysMap_x86`(9DKI)
- `svPhysUnmap_x86`(9DKI)
- `vmMapToPhys_x86`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

svPhysMap_powerpc, svPhysUnmap_powerpc, vmMapToPhys_powerpc – PowerPC physical to virtual memory mapping

Synopsis

```
#include <dki/dki.h>
KnError svPhysMap(KnPhMemChunk * chunk, PteCntlBits cntlBits);

void * svPhysUnmap(KnPhMemChunk * chunk);

KnError vmMapToPhys(KnCap * actor, KnPhMemChunk * chunk, PteCntlBits cntlBits);

typedef struct {
  PhAddr  paddr;   /* physical start addree */
  PhSize  psize;   /* size */
  VmAddr  vaddr;   /* virtual start address */
} KnPhMemChunk;
```

FEATURES

DKI

DESCRIPTION

Provides physical to virtual memory mapping services.

EXTENDED DESCRIPTION

The microkernel provides services to allow device drivers to map physical space to virtual memory space. These services should be used mainly by primary bus drivers to map bus I/O space or DMA memory.

`svPhysMap` allocates a range of virtual addresses within the supervisor address space and maps it to a given range of physical addresses.

The *chunk* argument points to the `KnPhMemChunk` structure.

The *paddr* and *psize* fields specify the chunk of physical memory being mapped to.

The *vaddr* field is used by `svPhysMap` to return the virtual address to which the physical one is mapped.

For PowerPC family processors, the *cntlBits* argument should be constructed by 'oring' the following values:

PTE_READ_WRITE	The mapped memory allows you to perform write accesses.
PTE_CACHE_WRITE_THROUGH	The memory is mapped with Write-Through attribute (bit `W`). Thus a store operation updates the cache if necessary, and in addition the update is written to memory.
PTE_CACHE_DISABLE	The memory is mapped with Cache-Inhibited attribute (bit I). Thus

	accesses are made to main memory, bypassing the caches.
PTE_MEMORY_COHERENCY	The memory is mapped with Memory-Coherency attribute (bit M). This enforce coherency of memory shared between processors in a system. When performing an access to memory, there is a hardware indication to the rest of the system that the access is global. Other processors affected by the access must then respond to this global access. Typically, this is used for a snooping bus design.
PTE_MEMORY_GUARDED	The memory is mapped with Memory-Guarded attribute (bit G). This prevents the processor from making out-of-order access to that memory (that is, access not directly dictated by the program). This may be useful if there are holes in physical memory, or to prevent these accesses to certain peripheral devices.

If *cntlBits* is equal to zero, the memory is mapped as read-only, not guarded, with cache enabled in write-back mode, and no coherency is enforced.

Note - Any combination where PTE_CACHE_WRITE_THROUGH and PTE_CACHE_DISABLE are both set is not supported.

On success K_OK is returned, otherwise a negative error code is returned:

K_ENOMEM	The system is out of memory.
K_ESIZE	The *psize* argument is equal to zero.

svPhysUnmap unmaps the physical memory chunk previously mapped by svPhysMap . The *chunk* argument points to the KnPhMemChunk structure which specifies the virtual and physical chunk start addresses and the chunk size. Note that the *paddr* and *psize* fields must have the values previously specified in svPhysMap . The *vaddr* field must have the value previously returned by svPhysMap .

Note - Typically, a driver uses the same KnPhMemChunk structure for the svPhysMap and svPhysUnmap calls and the structure fields are not modified by the driver once svPhysMap has been done.

vmMapToPhys maps a given physical memory chunk to the target actor address space. The *actor* argument specifies the target actor capability.

If *actor* is K_MYACTOR , the address space of the current actor is used.

If *actor* is K_SVACTOR , the supervisor address space is used.

The *chunk* argument points to the KnPhMemChunk structure which specifies the physical and virtual chunk start addresses and the chunk size.

The specified virtual address range must be allocated using the K_RESERVED option (see rgnAllocate(2)) prior to the invocation of vmMapToPhys .

The specified actor can be a supervisor actor as well as a user actor.

The mapping produced by vmMapToPhys can only be invalidated by rgnFree . *vaddr* , *paddr* and *psize* must be page-aligned.

For PowerPC family processors, the *cntlBits* is defined in the same way as for the svPhysMap routine above.

On success K_OK is returned, otherwise a negative error code is returned:

K_EFAULT	The *actor* argument points outside the caller's address space.
K_EINVAL	An inconsistent actor capability was provided.
K_EUNKNOWN	*actorcap* does not specify a reachable actor.
K_ENOMEM	The system is out of memory.
K_ESIZE	The *psize* argument is equal to zero.
K_EROUND	*vaddr* or *paddr* or *psize* is not page-aligned.
K_EADDR	Some or all addresses from the target virtual address range are out of a region allocated with the K_RESERVED option.

Note - For performance reasons, the current implementation does not guarantee that any attempt to map a physical address to a virtual address out of a region allocated with K_RESERVED option would effectively produce a K_EADDR error.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service.

Services	Base level	DKI thread	Interrupt	Blocking
svPhysMap	+	+	-	+
svPhysUnmap	+	+	-	+
vmMapToPhys	+	+	-	+

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svPhysAlloc`(9DKI) , `svDkiThreadCall`(9DKI) , `rgnAllocate`(2K)

NAME

svPhysMap_usparc, svPhysUnmap_usparc, vmMapToPhys_usparc – UltraSPARC physical to virtual memory mapping

SYNOPSIS

```
#include <dki/f_dki.h>
KnError svPhysMap(KnPhMemChunk * chunk, KnPteAttr attr);

void svPhysUnmap(KnPhMemChunk * chunk);

KnError vmMapToPhys(KnCap * actor, KnPhMemChunk * chunk, KnPteAttr attr);
```

FEATURES

DKI

DESCRIPTION

The microkernel provides services to allow device drivers to map physical space to virtual memory space. These services should be used mainly by a host bus driver to map bus I/O space or DMA memory.

svPhysMap

svPhysMap allocates a range of virtual addresses within the supervisor address space and maps it to a given range of physical addresses.

The *chunk* argument points to the KnPhMemChunk structure shown below:

```
typedef struct {
    PhAddr  paddr;    /* physical start addree */
    PhSize  psize;    /* size */
    VmAddr  vaddr;    /* virtual start address */
} KnPhMemChunk;
```

The *paddr* and *psize* fields specify the chunk of physical memory being mapped to.

The *vaddr* field is used by svPhysMap to return the virtual address to which the physical one is mapped.

The *attr* argument specifies the mapping attributes. *attr* is a bit-mask composed of two independent parts:

Data access attributes

Instruction access attributes

Basically, each part of the attributes is a sub-set of bits defined by the translation table entry (TTE) of the UltraSPARC MMU.

A combination of the following attributes may be specified for data access:

PTE_DATTR_G — global
The PTE_DATTR_G bit set allows the mapping to be shared among all (user and supervisor) contexts.

PTE_DATTR_W — writable
The PTE_DATTR_W bit set grants write permission for the mapping.

PTE_DATTR_P — privileged

The PTE_DATTR_P bit set restricts access to the mapping for the supervisor only.

PTE_DATTR_E — side-effect
The PTE_DATTR_E bit set makes noncacheable memory accesses to be strongly ordered against other E-bit accesses, and noncacheable stores are not merged. This bit should be set for I/O devices having side-effects. Note that the E-bit does not force an uncacheable access. It is expected that the PTE_DATTR_CV and PTE_DATTR_CP bits will be set to zero when the E-bit is set.

PTE_DATTR_CV — L1-cacheable
The PTE_DATTR_CV bit set allows data to be cached in the (L1) CPU data cache. Note that if the PTE_DATTR_CV bit is set, the PTE_DATTR_CP bit must also be set.

PTE_DATTR_CP — L2-cacheable
The PTE_DATTR_CP bit set allows data to be cached in the (L2) external cache.

PTE_DATTR_IE — invert endianness
The PTE_DATTR_IE bit set causes data accesses to the mapping to be processed with inverse endianness from that specified by the instruction.

PTE_DATTR_V — valid
The PTE_DATTR_V bit set enables data accesses to the mapping. If this bit is not set, all other bits (described above) are ignored and a data acess to the mapping will result in a data access exception.

Combinations of the following attributes may be specified for instruction access:

PTE_IATTR_G — global
The PTE_IATTR_G bit set allows the mapping to be shared among all (user and supervisor) contexts.

PTE_IATTR_P — privileged
The PTE_IATTR_P bit set restricts access to the mapping to the supervisor only.

PTE_IATTR_CV — L1-cacheable
The PTE_IATTR_CV bit set allows instructions to be cached in the (L1) CPU instruction cache. Note that if the PTE_IATTR_CV bit is set, the PTE_IATTR_CP bit must also be set.

PTE_IATTR_CP — L2-cacheable
The PTE_IATTR_CP bit set allows instructions to be cached in the (L2) external cache.

PTE_IATTR_V — valid
The PTE_IATTR_V bit set enables instructions to be obtained from the mapping. If this bit is not set, all other bits (described above) are ignored and an instruction to fetch from the mapping will result in an instruction access exception.

On success svPhysMap returns K_OK , otherwise a negative error code is returned as follows:

K_ENOMEM	The system is out of memory.
K_ESIZE	The *psize* argument is equal to zero.

svPhysUnmap

svPhysUnmap unmaps the physical memory chunk previously mapped by svPhysMap . The *chunk* argument points to the KnPhMemChunk structure which specifies the virtual and physical chunk start addresses and the chunk size. Note that the *paddr* and *psize* fields must have the values previously specified in svPhysMap . The *vaddr* fields must have the value previously returned by svPhysMap . Typically, a driver uses the same KnPhMemChunk structure for the svPhysMap and svPhysUnmap calls and the structure fields are not modified by the driver once svPhysMap has bee performed.

vmMapToPhys

vmMapToPhys maps a given physical memory chunk to the target actor address space.

The *actor* argument specifies the target actor capability. If *actor* is K_MYACTOR , the address space of the current actor is used. If *actor* is K_SVACTOR , the supervisor address space is used.

The *chunk* argument points to the KnPhMemChunk structure which specifies the physical and virtual chunk start addresses and the chunk size. The specified virtual address range must be allocated using the K_RESERVED option (see *rgnAllocate(2K)* prior to the invocation of vmMapToPhys . The specified actor can be a supervisor actor as well as an user actor. The mapping produced by vmMapToPhys can only be invalidated by rgnFree .

vaddr , *paddr* and *psize* must be page-aligned.

For UltraSPARC family processors, the *attr* argument is defined in the same way as for the svPhysMap routine above.

On success K_OK is returned, otherwise a negative error code is returned as follows:

K_EFAULT	The *actor* argument points outside of the caller's address space.

K_EINVAL	An inconsistent actor capability was specified.
K_EUNKNOWN	*actorcap* does not specify a reachable actor.
K_ENOMEM	The system is out of memory.
K_ESIZE	The *psize* argument is equal to zero.
K_EROUND	*vaddr* or *paddr* or *psize* is not page-aligned.
K_EADDR	Some or all addresses from the target virual address range are out of a region allocated with the K_RESERVED option. For performance reasons, the current implementation does not guarantee that any attempt to map a physical address to a virtual address outside a region allocated with K_RESERVED option would effectively produce a K_EADDR error.

ALLOWED CALLING CONTEXTS

The following table specifies the contexts in which a caller is allowed to invoke each service.

Services	Base level	DKI thread	Interrupt	Blocking
svPhysMap	+	+	-	+
svPhysUnmap	+	+	-	+
vmMapToPhys	+	+	-	+

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

svPhysAlloc(9DKI) , svDkiThreadCall(9DKI) , rgnAllocate(2K)

NAME

svPhysMap_x86, svPhysUnmap_x86, vmMapToPhys_x86 – Intel x86 physical to virtual memory mapping

SYNOPSIS

```
#include <dki/f_dki.h>
KnError svPhysMap(KnPhMemChunk * chunk, PteCntlBits cntlBits);

void svPhysUnmap(KnPhMemChunk * chunk);

KnError vmMapToPhys(KnCap * actor, KnPhMemChunk * chunk, PteCntlBits cntlBits);

typedef struct {
  PhAddr  paddr;   /* physical start address */
  PhSize  psize;   /* size */
  VmAddr  vaddr;   /* virtual start address */
} KnPhMemChunk;
```

FEATURES

DKI

EXTENDED DESCRIPTION

The microkernel provides services to allow device drivers to map physical space to virtual memory space. These services should be used mainly by primary bus drivers to map bus I/O space, or DMA memory.

`svPhysMap`

`svPhysMap` allocates a range of virtual addresses within the supervisor address space and maps it to a given range of physical addresses.

The *chunk* argument points to the `KnPhMemChunk` structure.

The *paddr* and *psize* fields specify the chunk of physical memory being mapped to.

The *vaddr* field is used by `svPhysMap` to return the virtual address to which the physical one is mapped.

For Intel ix86 family processor, the cntlBits argument is a direct mapping of the Page Control bits of Page Table Entries, as described in the Intel 386/486 Programming Reference Manual. Therefore, the cntlBits argument should be constructed by “oring” the following values:

`PTE_PRESENT`	The mapped memory must be present in physical memory.
`PTE_READ_WRITE`	The mapped memory allows write accesses to be performed.
`PTE_USER_SUPERVISOR`	The mapped memory can be used in user space.
`PTE_WRITE_TRANSPARENT`	The memory is mapped with the Write-Through attribute (bit PWT). Thus, a store operation updates the cache if necessary, and in addition the update is written to memory.

PTE_CACHE_DISABLE	The memory is mapped with caching disabled (bit PCD). Thus, memory accesses go to main memory, bypassing the caches.

On success K_OK is returned, otherwise a negative error code is returned:

K_ENOMEM	The system is out of memory.
K_ESIZE	The *psize* argument is equal to zero.

svPhysUnmap

svPhysUnmap unmaps the physical memory chunk previously mapped by svPhysMap . The *chunk* argument points to the KnPhMemChunk structure which specifies the virtual and physical chunk start addresses and the chunk size. Note that the *paddr* and *psize* fields must have the values previously specified in svPhysMap . The *vaddr* field must have the value previously returned by svPhysMap (typically, a driver uses the same KnPhMemChunk structure for the svPhysMap and svPhysUnmap calls and the structure fields are not modified by the driver once svPhysMap is done).

vmMapToPhys

vmMapToPhys maps a given physical memory chunk to the target actor address space.

The *actor* argument specifies the target actor capability. If *actor* is K_MYACTOR , the address space of the current actor is used. If *actor* is K_SVACTOR , the supervisor address space is used.

The *chunk* argument points to the KnPhMemChunk structure which specifies the physical and virtual chunk start addresses and the chunk size. The specified virtual address range must be allocated using the K_RESERVED option (see *rgnAllocate* (2K)) prior to the invocation of vmMapToPhys . The specified actor can be a supervisor actor as well as a user actor. The mapping produced by vmMapToPhys can only be invalidated by rgnFree .

vaddr , *paddr* and *psize* must be page-aligned.

For Intel x86 family processors, the *cntlBits* is defined in the same way as for the svPhysMap routine above.

On success K_OK is returned, otherwise a negative error code is returned:

K_EFAULT	The *actor* argument points to the outside of the caller's address space.
K_EINVAL	An inconsistent actor capability was provided.

K_EUNKNOWN	*actorcap* does not specify a reachable actor.
K_ENOMEM	The system is out of memory.
K_ESIZE	The *psize* argument is equal to zero.
K_EROUND	*vaddr* or *paddr* or *psize* is not page-aligned.
K_EADDR	Some or all addresses from the target virual address range are out of a region allocated with the K_RESERVED option (for performance reasons the current implementation does not guarantee that any attempt to map a physical address to a virtual address out of a region allocated with K_RESERVED option would effectively produce a K_EADDR error).

Intel x86 Memory Mapping Allowed Calling Contexts

The table below specifies the contexts in which a caller is allowed to invoke each service.

Services	Base level	DKI thread	Interrupt	Blocking
svPhysMap	+	+	-	+
svPhysUnmap	+	+	-	+
vmMapToPhys	+	+	-	+

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

svPhysAlloc(9DKI) , svDkiThreadCall(9DKI) , rgnAllocate(9DKI)

NAME

svIntrAttach_usparc, svIntrDetach_usparc, svSoftIntrAttach_usparc, svSoftIntrDetach_usparc, svTimerIntrAttach_uparc, svTimerIntrDetach_uparc, svIntrCtxGet_usparc – UltraSPARC interrupts management

SYNOPSIS

#include <dki/f_dki.h>

KnError **svIntrAttach**(unsigned int *intrNumb*, unsigned int *intrLevel*, CpuIntrHandler *intrHandler*, void * *intrCookie*, CpuIntrOps ** *intrOps*, CpuIntrId * *intrIdp*);

void **svIntrDetach**(CpuIntrId *intrId*);

KnError **svSoftIntrAttach**(unsigned int *intrLevel*, CpuIntrHandler *intrHandler*, void * *intrCookie*, CpuIntrOps ** *intrOps*, CpuIntrId * *intrId*);

void **svSoftIntrDetach**(CpuIntrId *intrId*);

KnError **svTimerIntrAttach**(CpuIntrHandler *intrHandler*, void * *intrCookie*, CpuIntrOps ** *intrOps*, CpuIntrId * *intrId*);

void **svTimerIntrDetach**(CpuIntrId *intrId*);

KnError **svIntrCtxGet**(KnIntrCtx ** *intrCtx*);

FEATURES

DKI

DESCRIPTION

Provides UltraSPARC interrupts management services.

An UltraSPARC processor is an implementation of the SPARC-V9 CPU architecture. As specified by the SPARC-V9 architecture, there are fifteen CPU interrupt sources assigned to the fifteen CPU interrupt levels from 1 up to 15. However, the UltraSPARC processor uses these fifteen interrupt levels only for software generated interrupts. The hardware interrupts are delivered to the processor using the "Mondo" interrupt transfer mechanism. The hardware interrupt source is designated by the interrupt number. Typically, the interrupt number is 11-bit width and composed of the interrupt group number (5 MSB) and the interrupt offset number (6 LSB).

The "Mondo" interrupt dispatch handler is built into the microkernel. The microkernel handles a mapping between the hardware interrupt numbers and software interrupt levels. When a mondo interrupt packet is received by the microkernel, the interrupt request descriptor is queued and an associated software interrupt is triggered. The microkernel software interrupt handler then dequeues the interrupt request descriptor and invokes a handler associated to the given interrupt number. In this way, a driver interrupt handler is always invoked in the UltraSPARC software interrupt context. This provides the interrupt handler with an environment analogous to the SPARC-V8 one (interrupt levels).

The microkernel provides services which allow device drivers to manage UltraSPARC interrupts, mainly to attach/detach handlers to the CPU interrupts.

```
typedef void (*CpuIntrHandler)(void*);
```

svIntrAttach

`svIntrAttach` attaches a given handler to a given interrupt number at a given processor interrupt level.

The *intrNumb* argument specifies the interrupt number to which to attach.

The *intrLevel* argument specifies the interrupt level to which to attach.

The *intrHandler* argument specifies the handler to call back when the given interrupt occurs.

The *intrCookie* argument specifies a parameter to pass back to the handler when called. It is opaque to the microkernel.

On success, `K_OK` is returned and the services defined on the attached interrupt are returned in the *intrOps* parameter. An identifier for the attached interrupt is also returned in *intrId* . This identifier must be used as first argument to subsequent calls to *intrOps* services.

On failure, an error code is returned as follows:

`K_EINVAL`	The specified interrupt number or interrupt level are invalid.
`K_BUSY`	Another handler is already attached to the given *intrNumb* .
`K_ENOMEM`	The system is out of memory.

Services available on an attached interrupt are defined by the `CpuIntrOps` structure as follows:

```
typedef struct CpuIntrOps {

        void
    (*mask) (CpuIntrId intrId);

        void
    (*unmask) (CpuIntrId intrId);

        void
    (*enable) (CpuIntrId intrId);

        void
    (*disable) (CpuIntrId intrId);

        void
    (*trigger) (CpuIntrId intrId);

} CpuIntrOps;
```

The `CpuIntrOps.mask` routine disables the interrupt at CPU level identified by *intrId* . In other words, the PIL register is set to the level corresponding to *intrId* . Note that the original value of the PIL register is saved by DKI in order to be restored later by the `CpuIntrOps.unmask` routine.

The `CpuIntrOps.unmask` routine enables the interrupt at CPU level identified by *intrId* .

In other words, the PIL register is restored to the original value saved by the previously called `CpuIntrOps.mask` routine.

The `mask` / `unmask` pair may be called from base level only and must not be nested. The `mask/unmask` pair is typically used to implement a critical section of code which needs to be protected against the interrupt. Note that with respect to the SPARC-V9 architecture, when an interrupt level `N` is masked, all interrupts with a level less than `N` are also masked.Thus, there is no way to `mask` only one CPU interrupt level except the lowest one.

The `enable` / `disable` pair may only be called from the attached interrupt handler.When an interrupt occurs, the attached *CpuIntrHandler* is invoked with the interrupt masked at processor level. This behaves in exactly the same way as if `CpuIntrOps.disable` was called just prior to the handler invocation. In other words, the PIL register is set to the interrupt level and the original interrupt processor level (which was when the interrupt occurred) is saved by DKI. Note that the interrupt handler must return to DKI in the same context as it was called, that is with the interrupt disabled at processor level.

On the other hand, the called interrupt handler may use the `enable/disable` pair to allow the interrupt to be nested. This feature is typically used by a host bus driver when the bus interrupts are multiplexed, that is, multiple bus interrupts are reported at the same CPU interrupt level. Typically, an interrupt handler of this type of host bus driver would take the following actions:

- Identify the bus interrupt source (through a PIC or special cycle).
- Disable the bus interrupt source at bus level (through PIC).
- Enable interrupt at processor level (`enable`).
- Call handlers attached to the identified bus interrupt source.
- Disable interrupt at processor level (\ \command{disable}).
- Acknowledge (if needed) and enable the bus interrupt source at bus level (through PIC).
- Return to the DKI.

The `CpuIntrOps.trigger` routine allows the interrupt to be triggered by software. Basically, this routine acts like the mondo interrupt dispatcher except the interrupt number is obtained from *intrId* rather than from the mondo interrupt packet. The `CpuIntrOps.trigger` routine is mainly dedicated to the

software interrupts attached by `svSoftIntrAttach`. However, it may be also used for hardware interrupts, for instance, for debugging or diagnostic purposes.

svIntrDetach

`svIntrDetach` detaches an interrupt handler previously connected by `svIntrAttach`.

The *intrId* argument identifies the attached interrupt handler, previously returned by `svIntrAttach`.

svSoftIntrDetach

`svSoftIntrAttach` attaches a given software interrupt handler to a given processor interrupt level.

The *intrLevel* argument specifies the interrupt level to which to attach.

The *intrHandler* argument specifies the handler to call back when the given interrupt is triggered.

The *intrCookie* argument specifies a parameter to pass back to the handler when called. It is opaque to the microkernel.

On success, `K_OK` is returned and the services defined on the attached interrupt are returned in the *intrOps* parameter.

An identifier for the attached interrupt is also returned in *intrId*. This identifier must be used as the first argument to subsequent calls to *intrOps* services.

On failure, an error code is returned as follows:

`K_EINVAL`	The specified interrupt level is invalid.
`K_ENOMEM`	The system is out of memory.

Services available on an attached interrupt are defined by the `CpuIntrOps` structure described above.

svSoftIntrDetach

`svSoftIntrDetach` detaches a software interrupt handler previously connected by `svSoftIntrAttach`.

The *intrId* argument identifies the attached interrupt handler, previously returned by `svSoftIntrAttach`.

svTimerIntrAttach

`svTimerIntrAttach` attaches a given interrupt handler to the UltraSPARC tick-counter interrupt.

The *intrHandler* argument specifies the handler to call back when the given interrupt is triggered.

The *intrCookie* argument specifies a parameter to pass back to the handler when called. It is opaque the microkernel.

On success, K_OK is returned and the services defined on the attached interrupt are returned in the *intrOps* parameter. An identifier for the attached interrupt is also returned in *intrId* . This identifier must be used as the first argument to subsequent calls to *intrOps* services.

On failure, an error code is returned as follows:

K_BUSY	Another handler is already attached to the tick-counter interrupt.
K_ENOMEM	The system is out of memory.

Services available on an attached interrupt are defined by the CpuIntrOps structure described above.

svTimerIntrDetach

svTimerIntrDetach detaches the interrupt handler previously connected by svTimerIntrAttach .

The *intrId* argument identifies the attached interrupt handler, previously returned by svTimerIntrAttach .

svIntrCtxGet

svIntrCtxGet retrieves the current level interrupt context. It is typically used for profiling purposes.

On successs, K_OK is returned and a pointer to the recently saved interrupt context is returned in the *intrCtx* argument. The CPU context saved on interrupt has the same structure as the thread context saved on exception or trap. It contains the global registers %g1-%g7 , the output registers of the interrupted window %o0-%o7 and the following processor registers: %tstate , %pc , %npc , %tt , %y . In addition, the thread context contains the number of outstanding windows and the pointer to the outstanding windows buffer if the number of windows is greater than zero.

On failure, an error code is returned as follows:

K_EINVAL	Interrupt level is zero (not called from an interrupt handler).
K_ENOTAVAILABLE	There is no context available for currently handled interrupt.

ALLOWED CALLING CONTEXTS

The following table specifies the contexts in which a caller is allowed to invoke each interrupt management service.

Services	Base level	DKI thread	Interrupt	Blocking
svIntrAttach	+	+	-	+

svIntrDetach	+	+	-	+
svIntrSoftAttach	+	+	-	+
svIntrSoftDetach	+	+	-	+
svTimerIntrAttach	+	+	-	+
svTimerIntrDetach	+	+	-	+
svIntrCtxGet	-	-	+	-
CpuIntrOps.mask	+	+	-	-
CpuIntrOps.unmask	+	+	-	-
CpuIntrOps.enable	-	-	+	-
CpuIntrOps.disable	-	-	+	-
CpuIntrOps.trigger	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

svIntrAttach_usparc, svIntrDetach_usparc, svSoftIntrAttach_usparc, svSoftIntrDetach_usparc, svTimerIntrAttach_uparc, svTimerIntrDetach_uparc, svIntrCtxGet_usparc – UltraSPARC interrupts management

SYNOPSIS

#include <dki/f_dki.h>

KnError **svIntrAttach**(unsigned int *intrNumb*, unsigned int *intrLevel*, CpuIntrHandler *intrHandler*, void * *intrCookie*, CpuIntrOps ** *intrOps*, CpuIntrId * *intrIdp*);

void **svIntrDetach**(CpuIntrId *intrId*);

KnError **svSoftIntrAttach**(unsigned int *intrLevel*, CpuIntrHandler *intrHandler*, void * *intrCookie*, CpuIntrOps ** *intrOps*, CpuIntrId * *intrId*);

void **svSoftIntrDetach**(CpuIntrId *intrId*);

KnError **svTimerIntrAttach**(CpuIntrHandler *intrHandler*, void * *intrCookie*, CpuIntrOps ** *intrOps*, CpuIntrId * *intrId*);

void **svTimerIntrDetach**(CpuIntrId *intrId*);

KnError **svIntrCtxGet**(KnIntrCtx ** *intrCtx*);

FEATURES

DKI

DESCRIPTION

Provides UltraSPARC interrupts management services.

An UltraSPARC processor is an implementation of the SPARC-V9 CPU architecture. As specified by the SPARC-V9 architecture, there are fifteen CPU interrupt sources assigned to the fifteen CPU interrupt levels from 1 up to 15. However, the UltraSPARC processor uses these fifteen interrupt levels only for software generated interrupts. The hardware interrupts are delivered to the processor using the "Mondo" interrupt transfer mechanism. The hardware interrupt source is designated by the interrupt number. Typically, the interrupt number is 11-bit width and composed of the interrupt group number (5 MSB) and the interrupt offset number (6 LSB).

The "Mondo" interrupt dispatch handler is built into the microkernel. The microkernel handles a mapping between the hardware interrupt numbers and software interrupt levels. When a mondo interrupt packet is received by the microkernel, the interrupt request descriptor is queued and an associated software interrupt is triggered. The microkernel software interrupt handler then dequeues the interrupt request descriptor and invokes a handler associated to the given interrupt number. In this way, a driver interrupt handler is always invoked in the UltraSPARC software interrupt context. This provides the interrupt handler with an environment analogous to the SPARC-V8 one (interrupt levels).

The microkernel provides services which allow device drivers to manage UltraSPARC interrupts, mainly to attach/detach handlers to the CPU interrupts.

```
typedef void (*CpuIntrHandler)(void*);
```

svIntrAttach

`svIntrAttach` attaches a given handler to a given interrupt number at a given processor interrupt level.

The *intrNumb* argument specifies the interrupt number to which to attach.

The *intrLevel* argument specifies the interrupt level to which to attach.

The *intrHandler* argument specifies the handler to call back when the given interrupt occurs.

The *intrCookie* argument specifies a parameter to pass back to the handler when called. It is opaque to the microkernel.

On success, `K_OK` is returned and the services defined on the attached interrupt are returned in the *intrOps* parameter. An identifier for the attached interrupt is also returned in *intrId* . This identifier must be used as first argument to subsequent calls to *intrOps* services.

On failure, an error code is returned as follows:

`K_EINVAL`	The specified interrupt number or interrupt level are invalid.
`K_BUSY`	Another handler is already attached to the given *intrNumb* .
`K_ENOMEM`	The system is out of memory.

Services available on an attached interrupt are defined by the `CpuIntrOps` structure as follows:

```
typedef struct CpuIntrOps {

        void
    (*mask) (CpuIntrId intrId);

        void
    (*unmask) (CpuIntrId intrId);

        void
    (*enable) (CpuIntrId intrId);

        void
    (*disable) (CpuIntrId intrId);

        void
    (*trigger) (CpuIntrId intrId);

} CpuIntrOps;
```

The `CpuIntrOps.mask` routine disables the interrupt at CPU level identified by *intrId* . In other words, the PIL register is set to the level corresponding to *intrId* . Note that the original value of the PIL register is saved by DKI in order to be restored later by the `CpuIntrOps.unmask` routine.

The `CpuIntrOps.unmask` routine enables the interrupt at CPU level identified by *intrId* .

In other words, the PIL register is restored to the original value saved by the previously called `CpuIntrOps.mask` routine.

The `mask` / `unmask` pair may be called from base level only and must not be nested. The `mask/unmask` pair is typically used to implement a critical section of code which needs to be protected against the interrupt. Note that with respect to the SPARC-V9 architecture, when an interrupt level `N` is masked, all interrupts with a level less than `N` are also masked.Thus, there is no way to `mask` only one CPU interrupt level except the lowest one.

The `enable` / `disable` pair may only be called from the attached interrupt handler.When an interrupt occurs, the attached *CpuIntrHandler* is invoked with the interrupt masked at processor level. This behaves in exactly the same way as if `CpuIntrOps.disable` was called just prior to the handler invocation. In other words, the PIL register is set to the interrupt level and the original interrupt processor level (which was when the interrupt occurred) is saved by DKI. Note that the interrupt handler must return to DKI in the same context as it was called, that is with the interrupt disabled at processor level.

On the other hand, the called interrupt handler may use the `enable/disable` pair to allow the interrupt to be nested. This feature is typically used by a host bus driver when the bus interrupts are multiplexed, that is, multiple bus interrupts are reported at the same CPU interrupt level. Typically, an interrupt handler of this type of host bus driver would take the following actions:

- Identify the bus interrupt source (through a PIC or special cycle).
- Disable the bus interrupt source at bus level (through PIC).
- Enable interrupt at processor level (`enable`).
- Call handlers attached to the identified bus interrupt source.
- Disable interrupt at processor level (\ \command{disable}).
- Acknowledge (if needed) and enable the bus interrupt source at bus level (through PIC).
- Return to the DKI.

The `CpuIntrOps.trigger` routine allows the interrupt to be triggered by software. Basically, this routine acts like the mondo interrupt dispatcher except the interrupt number is obtained from *intrId* rather than from the mondo interrupt packet. The `CpuIntrOps.trigger` routine is mainly dedicated to the

software interrupts attached by `svSoftIntrAttach`. However, it may be also used for hardware interrupts, for instance, for debugging or diagnostic purposes.

svIntrDetach

`svIntrDetach` detaches an interrupt handler previously connected by `svIntrAttach`.

The *intrId* argument identifies the attached interrupt handler, previously returned by `svIntrAttach`.

svSoftIntrDetach

`svSoftIntrAttach` attaches a given software interrupt handler to a given processor interrupt level.

The *intrLevel* argument specifies the interrupt level to which to attach.

The *intrHandler* argument specifies the handler to call back when the given interrupt is triggered.

The *intrCookie* argument specifies a parameter to pass back to the handler when called. It is opaque to the microkernel.

On success, `K_OK` is returned and the services defined on the attached interrupt are returned in the *intrOps* parameter.

An identifier for the attached interrupt is also returned in *intrId*. This identifier must be used as the first argument to subsequent calls to *intrOps* services.

On failure, an error code is returned as follows:

`K_EINVAL`	The specified interrupt level is invalid.
`K_ENOMEM`	The system is out of memory.

Services available on an attached interrupt are defined by the `CpuIntrOps` structure described above.

svSoftIntrDetach

`svSoftIntrDetach` detaches a software interrupt handler previously connected by `svSoftIntrAttach`.

The *intrId* argument identifies the attached interrupt handler, previously returned by `svSoftIntrAttach`.

svTimerIntrAttach

`svTimerIntrAttach` attaches a given interrupt handler to the UltraSPARC tick-counter interrupt.

The *intrHandler* argument specifies the handler to call back when the given interrupt is triggered.

The *intrCookie* argument specifies a parameter to pass back to the handler when called. It is opaque the microkernel.

On success, K_OK is returned and the services defined on the attached interrupt are returned in the *intrOps* parameter. An identifier for the attached interrupt is also returned in *intrId* . This identifier must be used as the first argument to subsequent calls to *intrOps* services.

On failure, an error code is returned as follows:

K_BUSY	Another handler is already attached to the tick-counter interrupt.
K_ENOMEM	The system is out of memory.

Services available on an attached interrupt are defined by the CpuIntrOps structure described above.

svTimerIntrDetach

svTimerIntrDetach detaches the interrupt handler previously connected by svTimerIntrAttach .

The *intrId* argument identifies the attached interrupt handler, previously returned by svTimerIntrAttach .

svIntrCtxGet

svIntrCtxGet retrieves the current level interrupt context. It is typically used for profiling purposes.

On successs, K_OK is returned and a pointer to the recently saved interrupt context is returned in the *intrCtx* argument. The CPU context saved on interrupt has the same structure as the thread context saved on exception or trap. It contains the global registers %g1-%g7 , the output registers of the interrupted window %o0-%o7 and the following processor registers: %tstate , %pc , %npc , %tt , %y . In addition, the thread context contains the number of outstanding windows and the pointer to the outstanding windows buffer if the number of windows is greater than zero.

On failure, an error code is returned as follows:

K_EINVAL	Interrupt level is zero (not called from an interrupt handler).
K_ENOTAVAILABLE	There is no context available for currently handled interrupt.

ALLOWED CALLING CONTEXTS

The following table specifies the contexts in which a caller is allowed to invoke each interrupt management service.

Services	Base level	DKI thread	Interrupt	Blocking
svIntrAttach	+	+	-	+

svIntrDetach	+	+	-	+
svIntrSoftAttach	+	+	-	+
svIntrSoftDetach	+	+	-	+
svTimerIntrAttach	+	+	-	+
svTimerIntrDetach	+	+	-	+
svIntrCtxGet	-	-	+	-
CpuIntrOps.mask	+	+	-	-
CpuIntrOps.unmask	+	+	-	-
CpuIntrOps.enable	-	-	+	-
CpuIntrOps.disable	-	-	+	-
CpuIntrOps.trigger	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

svTimeoutSet, svTimeoutCancel, svTimeoutGetRes – timeout operations

SYNOPSIS

#include <dki/dki.h>
KnError **svTimeoutSet**(KnTimeout * *timeout*, KnToHdl *handler*, KnTimeVal * *waitLimit*, int *flag*);

Bool **svTimeoutCancel**(KnTimeout * *timeout*);

int **svTimeoutGetRes**(KnTimeVal * *resolution*);

FEATURES

DKI

DESCRIPTION

Provides timeout operations services.

EXTENDED DESCRIPTION

Device drivers may need `Timeout` services to check whether there is any activity on a device, or to verify that a started action will terminate before a given time limit is reached. The `svTimeout` API is used for DKI purposes. Note that as these services should be implemented using drivers, they are not available and must not be used by drivers at initialization time.

`svTimeoutSet` sets a timeout request. When the interval of time specified by *waitLimit* has expired, the *handler* is invoked with *timeout* passed as its only argument. Fields within the `KnTimeout` structure are initialized and modified solely within the nucleus and are inaccessible to an application.

The *handler* argument points to a timeout handler routine. A timeout handler is a special kind of interrupt handler which is executed solely in supervisor execution mode, its code and accessed data must be within the locked-in-memory regions of the supervisor space. The set of kernel calls that can be used in a timeout handler are limited, as for any interrupt handler.

The *waitLimit* pointer argument refers to a `KnTimeVal` structure whose members are defined in sysTime(2K).

`svTimeoutSet` expects the `KnTimeVal` structure parameter to be set to a relative interval of time after which the handler is invoked. The *flag* argument is unused and should therefore be set to `0`.

On success, `svTimeoutSet` returns `K_OK`.

Otherwise, `K_EINVAL` is returned specifying that the *waitLimit* value is invalid or that no handler was specified.

`svTimeoutSet` must not be called using a `KnTimeout` which has an existing timeout pending (use `svTimeoutCancel` to cancel the request first).

`svTimeoutCancel` attempts to cancel a timeout request. It takes as an argument the address of the `KnTimeout` object used in the call to `svTimeoutSet` . If the timeout request is still pending, it is immediately cancelled and `TRUE` is returned signifying that the timer had not yet expired. Otherwise, the timeout interval had already passed and the timeout handler was called, in which case nothing occurs and `FALSE` is returned.

`svTimeoutGetRes` returns in *resolution* the smallest possible difference between two distinct *waitLimit* values. Note that `Timeout` resolution usually corresponds to the microkernel tick period (usually 10 milli-seconds).

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service.

Services	Base level	DKI thread	Interrupt	Blocking
svTimeoutSet	+	+	+	-
svTimeoutCancel	+	+	+	-
svTimeoutGetRes	+	+	-	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`sysTime`(2K) , `svDkiThreadCall`(9DKI)

NAME

svTimeoutSet, svTimeoutCancel, svTimeoutGetRes – timeout operations

SYNOPSIS

#include <dki/dki.h>
KnError **svTimeoutSet**(KnTimeout * *timeout*, KnToHdl *handler*, KnTimeVal * *waitLimit*, int *flag*);

Bool **svTimeoutCancel**(KnTimeout * *timeout*);

int **svTimeoutGetRes**(KnTimeVal * *resolution*);

FEATURES

DKI

DESCRIPTION

Provides timeout operations services.

EXTENDED DESCRIPTION

Device drivers may need `Timeout` services to check whether there is any activity on a device, or to verify that a started action will terminate before a given time limit is reached. The `svTimeout` API is used for DKI purposes. Note that as these services should be implemented using drivers, they are not available and must not be used by drivers at initialization time.

`svTimeoutSet` sets a timeout request. When the interval of time specified by *waitLimit* has expired, the *handler* is invoked with *timeout* passed as its only argument. Fields within the `KnTimeout` structure are initialized and modified solely within the nucleus and are inaccessible to an application.

The *handler* argument points to a timeout handler routine. A timeout handler is a special kind of interrupt handler which is executed solely in supervisor execution mode, its code and accessed data must be within the locked-in-memory regions of the supervisor space. The set of kernel calls that can be used in a timeout handler are limited, as for any interrupt handler.

The *waitLimit* pointer argument refers to a `KnTimeVal` structure whose members are defined in sysTime(2K).

`svTimeoutSet` expects the `KnTimeVal` structure parameter to be set to a relative interval of time after which the handler is invoked. The *flag* argument is unused and should therefore be set to `0`.

On success, `svTimeoutSet` returns `K_OK`.

Otherwise, `K_EINVAL` is returned specifying that the *waitLimit* value is invalid or that no handler was specified.

`svTimeoutSet` must not be called using a `KnTimeout` which has an existing timeout pending (use `svTimeoutCancel` to cancel the request first).

`svTimeoutCancel` attempts to cancel a timeout request. It takes as an argument the address of the `KnTimeout` object used in the call to `svTimeoutSet` . If the timeout request is still pending, it is immediately cancelled and `TRUE` is returned signifying that the timer had not yet expired. Otherwise, the timeout interval had already passed and the timeout handler was called, in which case nothing occurs and `FALSE` is returned.

`svTimeoutGetRes` returns in *resolution* the smallest possible difference between two distinct *waitLimit* values. Note that `Timeout` resolution usually corresponds to the microkernel tick period (usually 10 milli-seconds).

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service.

Services	Base level	DKI thread	Interrupt	Blocking
svTimeoutSet	+	+	+	-
svTimeoutCancel	+	+	+	-
svTimeoutGetRes	+	+	-	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`sysTime`(2K) , `svDkiThreadCall`(9DKI)

NAME

svTimeoutSet, svTimeoutCancel, svTimeoutGetRes – timeout operations

SYNOPSIS

#include <dki/dki.h>
KnError **svTimeoutSet**(KnTimeout * *timeout*, KnToHdl *handler*, KnTimeVal * *waitLimit*, int *flag*);

Bool **svTimeoutCancel**(KnTimeout * *timeout*);

int **svTimeoutGetRes**(KnTimeVal * *resolution*);

FEATURES

DKI

DESCRIPTION

Provides timeout operations services.

EXTENDED DESCRIPTION

Device drivers may need `Timeout` services to check whether there is any activity on a device, or to verify that a started action will terminate before a given time limit is reached. The `svTimeout` API is used for DKI purposes. Note that as these services should be implemented using drivers, they are not available and must not be used by drivers at initialization time.

`svTimeoutSet` sets a timeout request. When the interval of time specified by *waitLimit* has expired, the *handler* is invoked with *timeout* passed as its only argument. Fields within the `KnTimeout` structure are initialized and modified solely within the nucleus and are inaccessible to an application.

The *handler* argument points to a timeout handler routine. A timeout handler is a special kind of interrupt handler which is executed solely in supervisor execution mode, its code and accessed data must be within the locked-in-memory regions of the supervisor space. The set of kernel calls that can be used in a timeout handler are limited, as for any interrupt handler.

The *waitLimit* pointer argument refers to a `KnTimeVal` structure whose members are defined in sysTime(2K).

`svTimeoutSet` expects the `KnTimeVal` structure parameter to be set to a relative interval of time after which the handler is invoked. The *flag* argument is unused and should therefore be set to `0`.

On success, `svTimeoutSet` returns `K_OK`.

Otherwise, `K_EINVAL` is returned specifying that the *waitLimit* value is invalid or that no handler was specified.

`svTimeoutSet` must not be called using a `KnTimeout` which has an existing timeout pending (use `svTimeoutCancel` to cancel the request first).

`svTimeoutCancel` attempts to cancel a timeout request. It takes as an argument the address of the `KnTimeout` object used in the call to `svTimeoutSet` . If the timeout request is still pending, it is immediately cancelled and `TRUE` is returned signifying that the timer had not yet expired. Otherwise, the timeout interval had already passed and the timeout handler was called, in which case nothing occurs and `FALSE` is returned.

`svTimeoutGetRes` returns in *resolution* the smallest possible difference between two distinct *waitLimit* values. Note that `Timeout` resolution usually corresponds to the microkernel tick period (usually 10 milli-seconds).

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service.

Services	Base level	DKI thread	Interrupt	Blocking
svTimeoutSet	+	+	+	-
svTimeoutCancel	+	+	+	-
svTimeoutGetRes	+	+	-	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`sysTime`(2K) , `svDkiThreadCall`(9DKI)

NAME

loadSwap_16, storeSwap_16, swap_16, loadSwap_32, storeSwap_32, swap_32, loadSwap_64, storeSwap_64, swap_64 – specific i/o services

SYNOPSIS

```
#include <dki/dki.h>
uint16_f loadSwap_16(uint16_f * addr);

void storeSwap_16(uint16_f * addr, uint16_f value);

void swap_16(uint16_f * addr);

uint32_f loadSwap_32(uint32_f * addr);

void storeSwap_32(uint32_f * addr, uint32_f value);

void swap_32(uint32_f * addr);

uint64_f loadSwap_64(uint64_f * addr);

void storeSwap_64(uint64_f * addr, uint64_f value);

void swap_64(uint64_f * addr);
```

FEATURES

DKI

DESCRIPTION

This function provides DKI routines to handle different byte ordering between the processor bus and the host bus.

EXTENDED DESCRIPTION

Typically, these services are intended to be used by a host bus driver to handle different byte ordering between the processor bus and the host bus. Specific I/O services are defined below as sets of routines where the _xx suffix indicates the bit length of the data on which the service applies. This suffix may take one of the following values:

_16 for 16-bit data

_32 for 32-bit data

_64 for 64-bit data

`loadSwap_xx` loads data from a given address and returns the corresponding byte swapped value. The *addr* argument specifies the address to read from.

`storeSwap_xx` stores into a given address the *value* byte swapped. The *addr* argument specifies the address to write to.

`swap_xx` swap in place the bytes of the data stored at a given address. The *addr* argument specifies the address of the data to byte-swap.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
loadSwap_xx	+	+	+	-

storeSwap_xx	+	+	+	-
swap_xx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

loadSwap_16, storeSwap_16, swap_16, loadSwap_32, storeSwap_32, swap_32, loadSwap_64, storeSwap_64, swap_64 – specific i/o services

SYNOPSIS

```
#include <dki/dki.h>
uint16_f loadSwap_16(uint16_f * addr);

void storeSwap_16(uint16_f * addr, uint16_f value);

void swap_16(uint16_f * addr);

uint32_f loadSwap_32(uint32_f * addr);

void storeSwap_32(uint32_f * addr, uint32_f value);

void swap_32(uint32_f * addr);

uint64_f loadSwap_64(uint64_f * addr);

void storeSwap_64(uint64_f * addr, uint64_f value);

void swap_64(uint64_f * addr);
```

FEATURES

DKI

DESCRIPTION

This function provides DKI routines to handle different byte ordering between the processor bus and the host bus.

EXTENDED DESCRIPTION

Typically, these services are intended to be used by a host bus driver to handle different byte ordering between the processor bus and the host bus. Specific I/O services are defined below as sets of routines where the _xx suffix indicates the bit length of the data on which the service applies. This suffix may take one of the following values:

_16 for 16-bit data

_32 for 32-bit data

_64 for 64-bit data

`loadSwap_xx` loads data from a given address and returns the corresponding byte swapped value. The *addr* argument specifies the address to read from.

`storeSwap_xx` stores into a given address the *value* byte swapped. The *addr* argument specifies the address to write to.

`swap_xx` swap in place the bytes of the data stored at a given address. The *addr* argument specifies the address of the data to byte-swap.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
loadSwap_xx	+	+	+	-

storeSwap_xx	+	+	+	-
swap_xx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

loadSwap_16, storeSwap_16, swap_16, loadSwap_32, storeSwap_32, swap_32, loadSwap_64, storeSwap_64, swap_64 – specific i/o services

SYNOPSIS

```
#include <dki/dki.h>
uint16_f loadSwap_16(uint16_f * addr);

void storeSwap_16(uint16_f * addr, uint16_f value);

void swap_16(uint16_f * addr);

uint32_f loadSwap_32(uint32_f * addr);

void storeSwap_32(uint32_f * addr, uint32_f value);

void swap_32(uint32_f * addr);

uint64_f loadSwap_64(uint64_f * addr);

void storeSwap_64(uint64_f * addr, uint64_f value);

void swap_64(uint64_f * addr);
```

FEATURES

DKI

DESCRIPTION

This function provides DKI routines to handle different byte ordering between the processor bus and the host bus.

EXTENDED DESCRIPTION

Typically, these services are intended to be used by a host bus driver to handle different byte ordering between the processor bus and the host bus. Specific I/O services are defined below as sets of routines where the _xx suffix indicates the bit length of the data on which the service applies. This suffix may take one of the following values:

_16 for 16-bit data

_32 for 32-bit data

_64 for 64-bit data

`loadSwap_xx` loads data from a given address and returns the corresponding byte swapped value. The *addr* argument specifies the address to read from.

`storeSwap_xx` stores into a given address the *value* byte swapped. The *addr* argument specifies the address to write to.

`swap_xx` swap in place the bytes of the data stored at a given address. The *addr* argument specifies the address of the data to byte-swap.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
loadSwap_xx	+	+	+	-

storeSwap_xx	+	+	+	-
swap_xx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

loadSwapEieio_16, storeSwapEieio_16, swapEieio_16, loadSwapEieio_32, storeSwapEieio_32, swapEieio_32, eieio – i/o services

FEATURES

DKI

DESCRIPTION

Provides access to DKI i/o routines, optimized to facilitate byte swapping and synchronization of input/output.

See the architecture specific man pages:

- `loadSwapEieio_16_powerpc`(9DKI)
- `storeSwapEieio_16_powerpc`(9DKI)
- `swapEieio_16_powerpc`(9DKI)
- `loadSwapEieio_32_powerpc`(9DKI)
- `storeSwapEieio_32_powerpc`(9DKI)
- `swapEieio_32_powerpc`(9DKI)
- `eieio`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

loadSwapEieio_16_powerpc, loadSwapSync_16_powerpc, storeSwapEieio_16_powerpc, storeSwapSync_16_powerpc, swapEieio_16_powerpc, loadSwapEieio_32_powerpc, loadSwapSync_32_powerpc, storeSwapEieio_32_powerpc, storeSwapSync_32_powerpc, swapEieio_32_powerpc, eieio_powerpc, ioSync_powerpc – PowerPC specific i/o services

SYNOPSIS

```
#include <dki/dki.h>
uint16_f loadSwapEieio_16(uint16_f * addr);

unit16_f loadSwapSync_16(uint16_f * addr);

void storeSwapEieio_16(uint16_f * addr, uint16_f value);

void storeSwapSync_16(uint16_f * addr, uint16_f value);

void swapEieio_16(uint16_f * addr);

uint32_f loadSwapEieio_32(uint32_f * addr);

unit32_f loadSwapSync_32(uint32_f * addr);

void storeSwapEieio_32(uint32_f * addr, uint32_f value);

void storeSwapSync_32(uint32_f * addr, uint32_f value);

void swapEieio_32 (uint32_f * addr);

void ioSync(void);

void eieio(void);
```

FEATURES

DKI

DESCRIPTION

Provides access to PowerPC DKI I/O routines, optimized to facilitate byte swapping and synchronization of input/output.

EXTENDED DESCRIPTION

The PowerPC DKI provides specific I/O routines to handle byte swapping and synchronization of I/O. These services are based on PowerPC specific instructions to store/load with byte swapping, and to enforce in-order execution of I/O.

Typically, these services are intended to be used by a host bus driver to handle different byte ordering between the PowerPC bus and the host bus, or to enforce sequential execution of I/O. PowerPC specific I/O services are defined below as sets of routines where the _xx suffix indicates the bit length of the data on which the service applies. This suffix may take one of the following values:

_16	for 16-bit data
_32	for 32-bit data

`loadSwapEieio_xx` loads data from a given address and returns the corresponding byte swapped value. A PowerPC `eieio` instruction is issued after the data is read, to enforce execution of subsequent I/O sequentially with respect to the current load. The *addr* argument specifies the address to read from.

`loadSwapSync_xx` behaves like `loadSwapEieio_xx` but issues a PowerPC `isync` instruction after that data is read. This ensures that all possible exceptions are taken into account before returning.

`storeSwapEieio_xx` stores, at a given address, the *value* byte swapped. A PowerPC `eieio` instruction is issued after the data is written, to enforce execution of subsequent I/O sequentially with respect to the current store. The *addr* argument specifies the address to write to.

`storeSwapSync_xx` behaves like `storeSwapEieio_xx` but issues a PowerPC `isync` instruction after that data is read. This ensures that all possible exceptions are taken into account before returning.

`swapEieio_xx` swaps in place the bytes of the data stored at a given address. A PowerPC `eieio` instruction is issued between reading the data and re-writing the byte-swapped data, to enforce execution in order. The *addr* argument specifies the address of the data to byte-swap.

`ioSync` enforces context synchronization, completion of all instructions, and completion of all exceptions before returning.

`eieio` enforces execution in order of I/O and is mapped to the corresponding PowerPC assembler instruction. This routine is intended for use in managing shared data structures, in doing memory-mapped I/O, and in preventing load/store combining operations in main memory. Refer to PowerPC programming manuals for more details.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
loadSwapEieio_xx	+	+	+	-
loadSwapSync_xx	+	+	+	-
storeSwapEieio_xx	+	+	+	-
storeSwapSync_xx	+	+	+	-
swapEieio_xx	+	+	+	-
ioSync	+	+	+	-
eieio	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

loadSwap_16(9DKI) , svDkiThreadCall(9DKI)

NAME

loadSwapEieio_16, storeSwapEieio_16, swapEieio_16, loadSwapEieio_32, storeSwapEieio_32, swapEieio_32, eieio – i/o services

FEATURES

DKI

DESCRIPTION

Provides access to DKI i/o routines, optimized to facilitate byte swapping and synchronization of input/output.

See the architecture specific man pages:

- `loadSwapEieio_16_powerpc`(9DKI)
- `storeSwapEieio_16_powerpc`(9DKI)
- `swapEieio_16_powerpc`(9DKI)
- `loadSwapEieio_32_powerpc`(9DKI)
- `storeSwapEieio_32_powerpc`(9DKI)
- `swapEieio_32_powerpc`(9DKI)
- `eieio`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

loadSwapEieio_16_powerpc, loadSwapSync_16_powerpc, storeSwapEieio_16_powerpc, storeSwapSync_16_powerpc, swapEieio_16_powerpc, loadSwapEieio_32_powerpc, loadSwapSync_32_powerpc, storeSwapEieio_32_powerpc, storeSwapSync_32_powerpc, swapEieio_32_powerpc, eieio_powerpc, ioSync_powerpc – PowerPC specific i/o services

SYNOPSIS

```
#include <dki/dki.h>
uint16_f loadSwapEieio_16(uint16_f * addr);

unit16_f loadSwapSync_16(uint16_f * addr);

void storeSwapEieio_16(uint16_f * addr, uint16_f value);

void storeSwapSync_16(uint16_f * addr, uint16_f value);

void swapEieio_16(uint16_f * addr);

uint32_f loadSwapEieio_32(uint32_f * addr);

unit32_f loadSwapSync_32(uint32_f * addr);

void storeSwapEieio_32(uint32_f * addr, uint32_f value);

void storeSwapSync_32(uint32_f * addr, uint32_f value);

void swapEieio_32 (uint32_f * addr);

void ioSync(void);

void eieio(void);
```

FEATURES

DKI

DESCRIPTION

Provides access to PowerPC DKI I/O routines, optimized to facilitate byte swapping and synchronization of input/output.

EXTENDED DESCRIPTION

The PowerPC DKI provides specific I/O routines to handle byte swapping and synchronization of I/O. These services are based on PowerPC specific instructions to store/load with byte swapping, and to enforce in-order execution of I/O.

Typically, these services are intended to be used by a host bus driver to handle different byte ordering between the PowerPC bus and the host bus, or to enforce sequential execution of I/O. PowerPC specific I/O services are defined below as sets of routines where the _xx suffix indicates the bit length of the data on which the service applies. This suffix may take one of the following values:

_16	for 16-bit data
_32	for 32-bit data

`loadSwapEieio_xx` loads data from a given address and returns the corresponding byte swapped value. A PowerPC `eieio` instruction is issued after the data is read, to enforce execution of subsequent I/O sequentially with respect to the current load. The *addr* argument specifies the address to read from.

`loadSwapSync_xx` behaves like `loadSwapEieio_xx` but issues a PowerPC `isync` instruction after that data is read. This ensures that all possible exceptions are taken into account before returning.

`storeSwapEieio_xx` stores, at a given address, the *value* byte swapped. A PowerPC `eieio` instruction is issued after the data is written, to enforce execution of subsequent I/O sequentially with respect to the current store. The *addr* argument specifies the address to write to.

`storeSwapSync_xx` behaves like `storeSwapEieio_xx` but issues a PowerPC `isync` instruction after that data is read. This ensures that all possible exceptions are taken into account before returning.

`swapEieio_xx` swaps in place the bytes of the data stored at a given address. A PowerPC `eieio` instruction is issued between reading the data and re-writing the byte-swapped data, to enforce execution in order. The *addr* argument specifies the address of the data to byte-swap.

`ioSync` enforces context synchronization, completion of all instructions, and completion of all exceptions before returning.

`eieio` enforces execution in order of I/O and is mapped to the corresponding PowerPC assembler instruction. This routine is intended for use in managing shared data structures, in doing memory-mapped I/O, and in preventing load/store combining operations in main memory. Refer to PowerPC programming manuals for more details.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
loadSwapEieio_xx	+	+	+	-
loadSwapSync_xx	+	+	+	-
storeSwapEieio_xx	+	+	+	-
storeSwapSync_xx	+	+	+	-
swapEieio_xx	+	+	+	-
ioSync	+	+	+	-
eieio	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

loadSwap_16(9DKI) , svDkiThreadCall(9DKI)

NAME

usecBusyWait – precise busy wait service

SYNOPSIS

#include <dki/dki.h>
void **usecBusyWait**(unsigned int *micro*);

FEATURES

DKI

DESCRIPTION

Provides precise busy wait services.

EXTENDED DESCRIPTION

Device drivers may use precise busy wait services to wait for a very short time. Note that busy wait means that the caller waits without releasing the CPU, as if executing a busy loop. Typically, this service may be used by a driver to comply with a specific timing of its device (accessing a serial ROM), or to wait for a command to complete on the device before starting another one (resetting a device).

`usecBusyWait` waits for at least *micro* micro-seconds, before returning. Note that *micro* must be in the range of `1 .. 1000`. Behavior is unpredictable for values out of that range.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service:

Services	Base level	DKI thread	Interrupt	Blocking
usecBusyWait	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDkiThreadCall`(9DKI)

NAME

svPhysMap, svPhysUnmap, vmMapToPhys – physical to virtual memory mapping

FEATURES

DKI

DESCRIPTION

The microkernel provides services to allow device drivers to map physical space to virtual memory space. This services should be used mainly by primary bus drivers to map bus I/O space, or DMA memory.

See the architecture specific man pages:

- `svPhysMap_usparc`(9DKI)
- `svPhysUnmap_usparc`(9DKI)
- `vmMapToPhys_usparc`(9DKI)
- `svPhysMap_powerpc`(9DKI)
- `svPhysUnmap_powerpc`(9DKI)
- `vmMapToPhys_powerpc`(9DKI)
- `svPhysMap_x86`(9DKI)
- `svPhysUnmap_x86`(9DKI)
- `vmMapToPhys_x86`(9DKI)

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

svPhysMap_powerpc, svPhysUnmap_powerpc, vmMapToPhys_powerpc – PowerPC physical to virtual memory mapping

Synopsis

```
#include <dki/dki.h>
KnError svPhysMap(KnPhMemChunk * chunk, PteCntlBits cntlBits);

void * svPhysUnmap(KnPhMemChunk * chunk);

KnError vmMapToPhys(KnCap * actor, KnPhMemChunk * chunk, PteCntlBits cntlBits);

typedef struct {
  PhAddr paddr;   /* physical start addree */
  PhSize psize;   /* size */
  VmAddr vaddr;   /* virtual start address */
} KnPhMemChunk;
```

FEATURES

DKI

DESCRIPTION

Provides physical to virtual memory mapping services.

EXTENDED DESCRIPTION

The microkernel provides services to allow device drivers to map physical space to virtual memory space. These services should be used mainly by primary bus drivers to map bus I/O space or DMA memory.

`svPhysMap` allocates a range of virtual addresses within the supervisor address space and maps it to a given range of physical addresses.

The *chunk* argument points to the `KnPhMemChunk` structure.

The *paddr* and *psize* fields specify the chunk of physical memory being mapped to.

The *vaddr* field is used by `svPhysMap` to return the virtual address to which the physical one is mapped.

For PowerPC family processors, the *cntlBits* argument should be constructed by 'oring' the following values:

PTE_READ_WRITE	The mapped memory allows you to perform write accesses.
PTE_CACHE_WRITE_THROUGH	The memory is mapped with Write-Through attribute (bit `W`). Thus a store operation updates the cache if necessary, and in addition the update is written to memory.
PTE_CACHE_DISABLE	The memory is mapped with Cache-Inhibited attribute (bit I). Thus

	accesses are made to main memory, bypassing the caches.
PTE_MEMORY_COHERENCY	The memory is mapped with Memory-Coherency attribute (bit M). This enforce coherency of memory shared between processors in a system. When performing an access to memory, there is a hardware indication to the rest of the system that the access is global. Other processors affected by the access must then respond to this global access. Typically, this is used for a snooping bus design.
PTE_MEMORY_GUARDED	The memory is mapped with Memory-Guarded attribute (bit G). This prevents the processor from making out-of-order access to that memory (that is, access not directly dictated by the program). This may be useful if there are holes in physical memory, or to prevent these accesses to certain peripheral devices.

If *cntlBits* is equal to zero, the memory is mapped as read-only, not guarded, with cache enabled in write-back mode, and no coherency is enforced.

Note - Any combination where PTE_CACHE_WRITE_THROUGH and PTE_CACHE_DISABLE are both set is not supported.

On success K_OK is returned, otherwise a negative error code is returned:

K_ENOMEM	The system is out of memory.
K_ESIZE	The *psize* argument is equal to zero.

svPhysUnmap unmaps the physical memory chunk previously mapped by svPhysMap . The *chunk* argument points to the KnPhMemChunk structure which specifies the virtual and physical chunk start addresses and the chunk size. Note that the *paddr* and *psize* fields must have the values previously specified in svPhysMap . The *vaddr* field must have the value previously returned by svPhysMap .

Note - Typically, a driver uses the same KnPhMemChunk structure for the svPhysMap and svPhysUnmap calls and the structure fields are not modified by the driver once svPhysMap has been done.

vmMapToPhys maps a given physical memory chunk to the target actor address space. The *actor* argument specifies the target actor capability.

If *actor* is K_MYACTOR , the address space of the current actor is used.

If *actor* is K_SVACTOR , the supervisor address space is used.

The *chunk* argument points to the KnPhMemChunk structure which specifies the physical and virtual chunk start addresses and the chunk size.

The specified virtual address range must be allocated using the K_RESERVED option (see rgnAllocate(2)) prior to the invocation of vmMapToPhys .

The specified actor can be a supervisor actor as well as a user actor.

The mapping produced by vmMapToPhys can only be invalidated by rgnFree . *vaddr* , *paddr* and *psize* must be page-aligned.

For PowerPC family processors, the *cntlBits* is defined in the same way as for the svPhysMap routine above.

On success K_OK is returned, otherwise a negative error code is returned:

K_EFAULT	The *actor* argument points outside the caller's address space.
K_EINVAL	An inconsistent actor capability was provided.
K_EUNKNOWN	*actorcap* does not specify a reachable actor.
K_ENOMEM	The system is out of memory.
K_ESIZE	The *psize* argument is equal to zero.
K_EROUND	*vaddr* or *paddr* or *psize* is not page-aligned.
K_EADDR	Some or all addresses from the target virtual address range are out of a region allocated with the K_RESERVED option.

Note - For performance reasons, the current implementation does not guarantee that any attempt to map a physical address to a virtual address out of a region allocated with K_RESERVED option would effectively produce a K_EADDR error.

Allowed Calling Contexts

The following table specifies the contexts in which a caller is allowed to invoke each service.

Services	Base level	DKI thread	Interrupt	Blocking
svPhysMap	+	+	-	+
svPhysUnmap	+	+	-	+
vmMapToPhys	+	+	-	+

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svPhysAlloc`(9DKI) , `svDkiThreadCall`(9DKI) , `rgnAllocate`(2K)

NAME

svPhysMap_usparc, svPhysUnmap_usparc, vmMapToPhys_usparc – UltraSPARC physical to virtual memory mapping

SYNOPSIS

```
#include <dki/f_dki.h>
KnError svPhysMap(KnPhMemChunk * chunk, KnPteAttr attr);

void svPhysUnmap(KnPhMemChunk * chunk);

KnError vmMapToPhys(KnCap * actor, KnPhMemChunk * chunk, KnPteAttr attr);
```

FEATURES

DKI

DESCRIPTION

The microkernel provides services to allow device drivers to map physical space to virtual memory space. These services should be used mainly by a host bus driver to map bus I/O space or DMA memory.

svPhysMap

svPhysMap allocates a range of virtual addresses within the supervisor address space and maps it to a given range of physical addresses.

The *chunk* argument points to the KnPhMemChunk structure shown below:

```
typedef struct {
    PhAddr  paddr;    /* physical start addree */
    PhSize  psize;    /* size */
    VmAddr  vaddr;    /* virtual start address */
} KnPhMemChunk;
```

The *paddr* and *psize* fields specify the chunk of physical memory being mapped to.

The *vaddr* field is used by svPhysMap to return the virtual address to which the physical one is mapped.

The *attr* argument specifies the mapping attributes. *attr* is a bit-mask composed of two independent parts:

Data access attributes

Instruction access attributes

Basically, each part of the attributes is a sub-set of bits defined by the translation table entry (TTE) of the UltraSPARC MMU.

A combination of the following attributes may be specified for data access:

PTE_DATTR_G — global
The PTE_DATTR_G bit set allows the mapping to be shared among all (user and supervisor) contexts.

PTE_DATTR_W — writable
The PTE_DATTR_W bit set grants write permission for the mapping.

PTE_DATTR_P — privileged

The PTE_DATTR_P bit set restricts access to the mapping for the supervisor only.

PTE_DATTR_E — side-effect
The PTE_DATTR_E bit set makes noncacheable memory accesses to be strongly ordered against other E-bit accesses, and noncacheable stores are not merged. This bit should be set for I/O devices having side-effects. Note that the E-bit does not force an uncacheable access. It is expected that the PTE_DATTR_CV and PTE_DATTR_CP bits will be set to zero when the E-bit is set.

PTE_DATTR_CV — L1-cacheable
The PTE_DATTR_CV bit set allows data to be cached in the (L1) CPU data cache. Note that if the PTE_DATTR_CV bit is set, the PTE_DATTR_CP bit must also be set.

PTE_DATTR_CP — L2-cacheable
The PTE_DATTR_CP bit set allows data to be cached in the (L2) external cache.

PTE_DATTR_IE — invert endianness
The PTE_DATTR_IE bit set causes data accesses to the mapping to be processed with inverse endianness from that specified by the instruction.

PTE_DATTR_V — valid
The PTE_DATTR_V bit set enables data accesses to the mapping. If this bit is not set, all other bits (described above) are ignored and a data acess to the mapping will result in a data access exception.

Combinations of the following attributes may be specified for instruction access:

PTE_IATTR_G — global
The PTE_IATTR_G bit set allows the mapping to be shared among all (user and supervisor) contexts.

PTE_IATTR_P — privileged
The PTE_IATTR_P bit set restricts access to the mapping to the supervisor only.

PTE_IATTR_CV — L1-cacheable
The PTE_IATTR_CV bit set allows instructions to be cached in the (L1) CPU instruction cache. Note that if the PTE_IATTR_CV bit is set, the PTE_IATTR_CP bit must also be set.

PTE_IATTR_CP — L2-cacheable
The PTE_IATTR_CP bit set allows instructions to be cached in the (L2) external cache.

PTE_IATTR_V — valid
The PTE_IATTR_V bit set enables instructions to be obtained from the mapping. If this bit is not set, all other bits (described above) are ignored and an instruction to fetch from the mapping will result in an instruction access exception.

On success svPhysMap returns K_OK , otherwise a negative error code is returned as follows:

K_ENOMEM	The system is out of memory.
K_ESIZE	The *psize* argument is equal to zero.

svPhysUnmap

svPhysUnmap unmaps the physical memory chunk previously mapped by svPhysMap . The *chunk* argument points to the KnPhMemChunk structure which specifies the virtual and physical chunk start addresses and the chunk size. Note that the *paddr* and *psize* fields must have the values previously specified in svPhysMap . The *vaddr* fields must have the value previously returned by svPhysMap . Typically, a driver uses the same KnPhMemChunk structure for the svPhysMap and svPhysUnmap calls and the structure fields are not modified by the driver once svPhysMap has bee performed.

vmMapToPhys

vmMapToPhys maps a given physical memory chunk to the target actor address space.

The *actor* argument specifies the target actor capability. If *actor* is K_MYACTOR , the address space of the current actor is used. If *actor* is K_SVACTOR , the supervisor address space is used.

The *chunk* argument points to the KnPhMemChunk structure which specifies the physical and virtual chunk start addresses and the chunk size. The specified virtual address range must be allocated using the K_RESERVED option (see *rgnAllocate(2K)* prior to the invocation of vmMapToPhys . The specified actor can be a supervisor actor as well as an user actor. The mapping produced by vmMapToPhys can only be invalidated by rgnFree .

vaddr , *paddr* and *psize* must be page-aligned.

For UltraSPARC family processors, the *attr* argument is defined in the same way as for the svPhysMap routine above.

On success K_OK is returned, otherwise a negative error code is returned as follows:

K_EFAULT	The *actor* argument points outside of the caller's address space.

K_EINVAL	An inconsistent actor capability was specified.
K_EUNKNOWN	*actorcap* does not specify a reachable actor.
K_ENOMEM	The system is out of memory.
K_ESIZE	The *psize* argument is equal to zero.
K_EROUND	*vaddr* or *paddr* or *psize* is not page-aligned.
K_EADDR	Some or all addresses from the target virual address range are out of a region allocated with the K_RESERVED option. For performance reasons, the current implementation does not guarantee that any attempt to map a physical address to a virtual address outside a region allocated with K_RESERVED option would effectively produce a K_EADDR error.

ALLOWED CALLING CONTEXTS

The following table specifies the contexts in which a caller is allowed to invoke each service.

Services	Base level	DKI thread	Interrupt	Blocking
svPhysMap	+	+	-	+
svPhysUnmap	+	+	-	+
vmMapToPhys	+	+	-	+

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

svPhysAlloc(9DKI) , svDkiThreadCall(9DKI) , rgnAllocate(2K)

NAME

svPhysMap_x86, svPhysUnmap_x86, vmMapToPhys_x86 – Intel x86 physical to virtual memory mapping

SYNOPSIS

```
#include <dki/f_dki.h>
KnError svPhysMap(KnPhMemChunk * chunk, PteCntlBits cntlBits);

void svPhysUnmap(KnPhMemChunk * chunk);

KnError vmMapToPhys(KnCap * actor, KnPhMemChunk * chunk, PteCntlBits cntlBits);

typedef struct {
  PhAddr  paddr;   /* physical start address */
  PhSize  psize;   /* size */
  VmAddr  vaddr;   /* virtual start address */
} KnPhMemChunk;
```

FEATURES

DKI

EXTENDED DESCRIPTION

The microkernel provides services to allow device drivers to map physical space to virtual memory space. These services should be used mainly by primary bus drivers to map bus I/O space, or DMA memory.

`svPhysMap`

`svPhysMap` allocates a range of virtual addresses within the supervisor address space and maps it to a given range of physical addresses.

The *chunk* argument points to the `KnPhMemChunk` structure.

The *paddr* and *psize* fields specify the chunk of physical memory being mapped to.

The *vaddr* field is used by `svPhysMap` to return the virtual address to which the physical one is mapped.

For Intel ix86 family processor, the cntlBits argument is a direct mapping of the Page Control bits of Page Table Entries, as described in the Intel 386/486 Programming Reference Manual. Therefore, the cntlBits argument should be constructed by “oring” the following values:

`PTE_PRESENT`	The mapped memory must be present in physical memory.
`PTE_READ_WRITE`	The mapped memory allows write accesses to be performed.
`PTE_USER_SUPERVISOR`	The mapped memory can be used in user space.
`PTE_WRITE_TRANSPARENT`	The memory is mapped with the Write-Through attribute (bit PWT). Thus, a store operation updates the cache if necessary, and in addition the update is written to memory.

PTE_CACHE_DISABLE	The memory is mapped with caching disabled (bit PCD). Thus, memory accesses go to main memory, bypassing the caches.

On success K_OK is returned, otherwise a negative error code is returned:

K_ENOMEM	The system is out of memory.
K_ESIZE	The *psize* argument is equal to zero.

svPhysUnmap

svPhysUnmap unmaps the physical memory chunk previously mapped by svPhysMap . The *chunk* argument points to the KnPhMemChunk structure which specifies the virtual and physical chunk start addresses and the chunk size. Note that the *paddr* and *psize* fields must have the values previously specified in svPhysMap . The *vaddr* field must have the value previously returned by svPhysMap (typically, a driver uses the same KnPhMemChunk structure for the svPhysMap and svPhysUnmap calls and the structure fields are not modified by the driver once svPhysMap is done).

vmMapToPhys

vmMapToPhys maps a given physical memory chunk to the target actor address space.

The *actor* argument specifies the target actor capability. If *actor* is K_MYACTOR , the address space of the current actor is used. If *actor* is K_SVACTOR , the supervisor address space is used.

The *chunk* argument points to the KnPhMemChunk structure which specifies the physical and virtual chunk start addresses and the chunk size. The specified virtual address range must be allocated using the K_RESERVED option (see *rgnAllocate* (2K)) prior to the invocation of vmMapToPhys . The specified actor can be a supervisor actor as well as a user actor. The mapping produced by vmMapToPhys can only be invalidated by rgnFree .

vaddr , *paddr* and *psize* must be page-aligned.

For Intel x86 family processors, the *cntlBits* is defined in the same way as for the svPhysMap routine above.

On success K_OK is returned, otherwise a negative error code is returned:

K_EFAULT	The *actor* argument points to the outside of the caller's address space.
K_EINVAL	An inconsistent actor capability was provided.

K_EUNKNOWN	*actorcap* does not specify a reachable actor.
K_ENOMEM	The system is out of memory.
K_ESIZE	The *psize* argument is equal to zero.
K_EROUND	*vaddr* or *paddr* or *psize* is not page-aligned.
K_EADDR	Some or all addresses from the target virual address range are out of a region allocated with the K_RESERVED option (for performance reasons the current implementation does not guarantee that any attempt to map a physical address to a virtual address out of a region allocated with K_RESERVED option would effectively produce a K_EADDR error).

Intel x86 Memory Mapping Allowed Calling Contexts

The table below specifies the contexts in which a caller is allowed to invoke each service.

Services	Base level	DKI thread	Interrupt	Blocking
svPhysMap	+	+	-	+
svPhysUnmap	+	+	-	+
vmMapToPhys	+	+	-	+

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

svPhysAlloc(9DKI) , svDkiThreadCall(9DKI) , rgnAllocate(9DKI)

Index

D

E

I

L

S

U

V