ChorusOS man pages section 9DDI: Device Driver Interfaces

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

Part No: 806-3341
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

Device Driver Interfaces

10

NAME

Intro – Device Driver Interface introduction

FEATURES

DDI

DESCRIPTION

Provides device driver interface services.

EXTENDED DESCRIPTION

Device Driver Interface (DDI) defines several layers of APIs between different driver components. Typically an API is defined for each class of bus or device, as a part of the DDI.

The DDI set of APIs is logically structured in 2 layers:

- The Bus Driver's APIs
- The Device Driver's APIs

The Bus driver's APIs are provided by the lower layers of driver components, and are built upon the DKI services. This set of drivers can itself be composed of multiple sub-layers to reflect the busses' hierarchy of a given platform.

As all different classes of I/O busses share a subset of features, and then have their particular specificities, a common subset of services has been defined in an API called "Common Bus Driver API". This API is independent of the bus class, and may be provided by a given bus driver in addition to its class specific set of services. This results in a "Common Bus Driver API" which can be used to implement simple multi-bus device drivers, and in a collection of bus class specific APIs write bus-aware device drivers.

The following table lists bus class APIs currently available in the ChorusOS DDI:

Name	Alias	API Header	API Name
"bus"	BUS_CLASS	<ddi/bus/bus.h>	Common bus API
"pci"	PCI_CLASS	<ddi/pci/pci.h>	PCI bus API
"isa"	ISA_CLASS	<ddi/isa/isa.h>	ISA bus API
"vme"	VME_CLASS	<ddi/vme/vme.h>	VME bus API

The Device driver's APIs are provided by the higher layers of driver components, and are typically built upon the Bus Driver APIs. A different API is provided for each different "class" of device. Each of these APIs may be used by a driver's client application to manage the associated devices.

The following table lists device class APIs currently available in the ChorusOS DDI.

Name	Alias	API Header	API Name
"timer"	TIMER_CLASS	<ddi/timer/timer.h>	Timer devices API
"uart"	UART_CLASS	<ddi/uart/uart.h>	UART devices API
"rtc"	RTC_CLASS	<ddi/rtc/rtc.h>	Real Time Clock API
"ether"	ETHER_CLASS	<ddi/net/ether/ether.h>	Ethernet devices API
“flahs”	FLASH_CLASS	<ddi/flash/flash.h>	Flash devices API
“nvram”	NVRAM_CLASS	<ddi/nvram/nvram.h>	NVRAM devices API
“bench”	BENCH_CLASS	<ddi/bench/bench.h>	Bench devices API

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`intro`(9DKI), `bus`(9DDI), `pci`(9DDI), `isa`(9DDI), `nvram`(9DDI), `timer`(9DDI), `uart`(9DDI), `rtc`(9DDI), `ether`(9DDI), `flash`(9DDI), `vme`(9DDI), `bench`(9DDI)

Name	Description
`bench`(9DDI)	Bench Device Driver Interface
`bus`(9DDI)	Common Bus Driver Interface
`ether`(9DDI)	Ethernet Device Driver Interface
`flash`(9DDI)	Flash Device Driver Interface
`isa`(9DDI)	ISA Bus Driver Interface
`netFrame`(9DDI)	Generic Representation of Network Frames
`nvram`(9DDI)	NVRAM Device Driver Interface
`pci`(9DDI)	PCI Bus Driver Interface
`quicc`(9DDI)	QUICC bus driver interface
`ric`(9DDI)	RIC Device Driver Interface
`rtc`(9DDI)	RTC Device Driver Interface
`timer`(9DDI)	TIMER Device Driver Interface
`uart`(9DDI)	UART Device Driver Interface

vme(9DDI) VME Bus Driver Interface

NAME

bench – Bench Device Driver Interface

SYNOPSIS

```
#include <ddi/bench/bench.h>
```

FEATURES

DDI

DESCRIPTION

The bench DDI describes a common interface through which bench programs can measure interrupt latencies in a manner independent of the device.

This interface can be implemented at any level of the device hierarchy, providing the hardware permits it. The (pseudo) bench DDI can thus be viewed as a DDI offered by potentially any device.

The bench driver service routines are declared by the `BenchOps` structure:

```
typedef void* BenchId;

    /*
     * Bench interrupt handler
     */
typedef void (*BenchIntrHandler) __((void* cookie));
    /*
     * Bench driver operations.
     */
typedef struct BenchOps {

    BenchVersion       version;

        KnError
    (*open)            __((BenchId            id,
                           BenchIntrHandler    hdl,
                           void*               cookie));

        void
    (*trigger_start) __((BenchId id));

        void
    (*trigger_stop) __((BenchId id));

        void
    (*trigger)          __((BenchId id));

        void
    (*trigger_overhead) __((BenchId id));

        void
    (*close)           __((BenchId id));

} BenchOps;
```

A pointer to the `BenchOps` structure is exported by adriver via the `svDeviceRegister` microkernel call. A driver client invokes the `svDeviceLookup` and `svDeviceEntry` microkernel calls in order to obtain a

pointer to the device service routines vector. Once the pointer is obtained, the driver client can invoke the driver service routines (via the indirect function call) in order to open/close the device, trigger device interrupts and also measure the overhead associated with the trigger.

The operations on the bench driver are the following:

1. The `open()` command establishes a connection between the driver client and a given device driver instance. The bench driver must be a mono client one.

   The parameter *devId* specifies a given bench device. The parameter *hdl* specifies the client interrupt handler that will be invoked by the driver. The parameter, *cookie*, points to the client handle passed as an argument to the call-back interrupt handler.

   The `open` function returns the following result:

   `K_OK` The bench driver has been started.

   `K_EFAIL` The open failed for reasons specific to each implementation.

2. The `trigger()` function triggers an interrupt on the device. The client handler (parameter to device open) will eventually be invoked with the device interrupts masked.
3. The `trigger_overhead()` function can be used to measure the overhead associated with triggering an interrupt. The client handler (parameter to device open) will be invoked before this call returns.
4. The `trigger_start()` and `trigger_stop()` functions allow the bench driver to do setup operations as may be necessary. The `trigger()` and `trigger_overhead()` functions must be invoked within the `trigger_start()` and `trigger_stop()` functions. The last two functions are to allow the bench device driver to do the necessary setup.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDeviceRegister`(9DKI), `svDeviceLookup`(9DKI), `sysTimerStartFreerun`(2K)

NAME

bus – Common Bus Driver Interface

SYNOPSIS

```
#include <ddi/bus/bus.h>
```

FEATURES

DDI

DESCRIPTION

Provides common bus driver interface services.

EXTENDED DESCRIPTION

The bus (that is, the bus bridge) driver is a keystone in the driver framework. The bus driver API enables development of platform independent device drivers. Note that the driver framework does not provide a fully generic bus driver API. Bus driver APIs are specific for each bus class (like ISA, PCI, USB, ...). However, in order to facilitate writing multi-bus device drivers, a part of a bus driver API is abstracted for some bus classes such as the Common Bus Driver Interface.

The common bus driver interface covers the following bus bridge services (provided to the device drivers):

- Bus / device driver connection establishment
- Interrupt management
- Interrupt handler connection / disconnection
- Interrupt source enabling / disabling
- Interrupt handler invocation
- I / O registers access
  - I / O registers mapping / unmapping
  - I / O registers load / store
- Memory-mapped regions

Usually, these services are provided by any bus bridge driver independent of its class. On the other hand, on some busses, these features may include some bus class specific features. Note that the driver framework does not guarantee that the Common Bus Driver Interface will be provided for all bus classes.

Instead, the driver framework explicitly specifies whether the Common Bus Driver Interface is provided for a particular bus class. Initially, the driver framework will provide the API specifications for the following bus classes:

- PCI
- ISA
- VME

All bus classes mentioned above support the Common Bus Driver Interface. It allows you to develop a multi-bus device driver. Obviously, APIs used by this

type of driver must be limited to the Common Bus Driver Interface described in this chapter. In other words, this type of driver must not use bus class specific routines. A multi-bus device driver specifies the "bus" parent class in the driver entry. In such a case, a bus driver supporting the Common Bus Driver Interface would invoke the driver initialization routine if a device serviced by the driver resides on the bus. The bus driver gives a *BusOps* / *BusId* pair to the device driver initialization routine. The *BusOps* / *BusId* pair specifies a bus driver instance which provides the Common Bus Driver Interface.

The *BusOps* structure contains pointers to the bus driver service routines.

The `BusOps` structure is the following:

```
typedef BusIntrStatus (*BusIntrHandler)  (void*);

typedef KnError       (*BusEventHandler) (void*, BusEvent, void*);

typedef void          (*BusLoadHandler)  (void*);

typedef void          (*BusErrHandler)   (void*, BusError*);

typedef struct {
    BusVersion  version;

       KnError
    (*open) (BusId           bus_id,
             DevNode         dev_node,
             BusEventHandler dev_event_handler,
             BusLoadHandler  dev_load_handler,
             void*           dev_cookie,
             BusDevId*       dev_id);

        void
    (*close) (BusDevId dev_id);

        KnError
    (*intr_attach) (BusDevId       dev_id,
                    void*          dev_intr,
                    BusIntrHandler dev_intr_handler,
                    void*          dev_intr_cookie,
                    BusIntrOps**   intr_ops,
                    BusIntrId*     intr_id));

        void
    (*intr_detach) (BusIntrId intr_id);

        KnError
    (*io_map) (BusDevId       dev_id,
               void*          io_regs,
               BusErrHandler  err_handler,
               void*          err_cookie,
               BusIoOps**     io_ops,
               BusIoId*       io_id);
```

```
        void
    (*io_unmap) (BusIoId io_id);

        KnError
    (*mem_map) (BusDevId       dev_id,
                void*          mem_rgn,
                BusMemAttr     mem_attr,
                BusErrHandler  err_handler,
                void*          err_cookie,
                void**         mem_addr,
                BusMemId*      mem_id);

        void
    (*mem_unmap) (BusMemId mem_id);

        void*
    (*intr_find) (void* propv, int index);

        void*
    (*io_regs_find) (void* propv, int index);

        void*
    (*mem_rgn_find) (void* propv, int index);

} BusOps;
```

The *BusId* is an opaque parameter for the device driver. It must be passed back to the bus driver as an argument of the open routine.

The *version* field specifies the bus driver API version number. The version number is incremented each time one of the bus driver structures is extended in order to include new service routines. In other words, a new symbol is added to the BusVersion (for example, BUS_VERSION_1) each time this type of an API extension is performed. Obviously, all extensions made in the bus driver API must be explicitly documented.

A device driver specifies a minimal API version number required by the driver within its registry entry. The device driver initialization routine is not invoked by the bus driver if the API version number provided by the bus driver is less than the API version number specified within the device driver registry entry.

open/close operations

open must be the first call issued by the device driver to the bus driver. It establishes a connection between the device and bus driver.

The *bus_id* argument specifies the bus driver instance.

The *dev_node* argument specifies the device node (in the device tree) which is serviced by the device driver instance. In the case of initialization, the device node is given as an argument of dev_init by the parent bus driver. In the 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 *event_handler* argument specifies the device driver handler which is invoked by the bus driver when a bus event occurs.

event_handler has three arguments. The first argument is *event_cookie*. The second one specifies the bus event type. The third argument points to a structure which is event type specific.

Among all bus events which are mostly bus class specific, there are three shut-down related events specified by the common bus API:

`BUS_DEV_SHUTDOWN` Notifies a device driver that the device should be shut down. The device driver should notify driver clients (via `svDeviceShutDown`) that the device is going to be shut down and then should return from the event handler. Once the device entry is released by the last driver client, the device registry module invokes a driver call-back handler. Within this handler, the device driver should reset the device hardware, release all allocated resources and close the bus connection invoking the `close` routine. Note that the `BUS_DEV_SHUTDOWN` event may be used by a bus driver in order to confiscate (or to re-allocate) bus resources.

`BUS_DEV_REMOVAL` Notifies a device driver that the device has been removed from the bus and therefore the device driver instance has to be shut down. The actions taken by the driver are similar to the `BUS_DEV_SHUTDOWN` case except that the device hardware must not be accessed by the driver

`BUS_SYS_SHUTDOWN` Notifies a device driver that the system is going to be shut down. The device driver should reset the device hardware and return from the event handler. Note that the driver should neither notify clients nor free allocated resources.

The *event_cookie* argument specifies a device driver cookie. It is an opaque argument for the bus driver. *event_cookie* is passed back to the device driver when *event_handler* or *load_handler* is invoked.

Typically, *event_handler* is called as an interrupt handler and therefore the handler implementation must be restricted to the API allowed at interrupt level.

The *load_handler* argument specifies the driver handler which is invoked by the bus driver when a new driver has been dynamically loaded. It is invoked passing *dev_cookie* as the only argument. Note that this *load_handler* argument 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. This should lead to the driver being associated with a device node and being initialized, in order to create a running instance of the newly loaded driver.

Upon successful completion, the bus driver returns a `BusDevId` identifier designating the bus/device connection. `BusDevId` is opaque for the device driver. It must be passed back to the bus driver as an argument in subsequent invocations of the `BusOps` service routines.

In case of failure, an error code is returned as described below:

`K_EINVAL`	The *dev_node* argument given is not a valid device tree node.
`K_EBUSY`	The *dev_node* device tree node given is already in use (associated with another driver).
`K_ENOMEM`	The system is out of memory.

The `close` routine releases the bus/driver connection. It must be the last call issued to the bus driver.

Bus Resource Properties

Bus resources used by the device are specified as properties attached to the device node. There are three types of bus resource properties which have standard names on buses providing the common bus interface:

"intr"	Device interrupts
"io-regs"	Device I/O registers sets
"mem-rgn"	Device memory regions

Note that the property value format (that is, the bus resource description) is bus class specific. However, the common bus driver API enables bus independent device drivers to use their resources without knowing what those resources are. In order to use a given resource type (for example, *"intr"*), a bus independent driver obtains a pointer to the corresponding property value using the device tree API.

It then invokes an appropriate bus service routine (for example, *intr_attach*) passing as the bus resource descriptor a pointer to the property value. Note that the property value may be an array of bus resource descriptors. In other words, the property value may specify a number of bus resources of the same type (for example, a number of interrupt lines).

In order to enable bus independent device drivers to parse this type of property value without knowing the bus resource descriptor format, the common bus driver interface provides the following service routines:

- `intr_find`
- `io_regs_find`
- `mem_rgn_find`

These routines allow a driver to obtain a pointer to an element within the array of corresponding types, specified by a pointer to the property value.

intr_attach/intr_detach

The `intr_attach` service routine connects the device specific handler to a given bus interrupt source. The *dev_id* argument specifies the given device on the given bus. It is an opaque returned by `open`. The *dev_intr* argument specifies the bus interrupt source. It points to an interrupt resource property value which is bus class specific.

Typically, the bus class API defines a structure (or a basic type) which specifies the value format for an interrupt resource property attached to the device node. A device driver should find such a property (attached to its device node) using the `"intr"` name, obtain a pointer to the property value and specify it in `intr_attach`. In a case where the property value specifies multiple interrupt sources (the property value is an array of the interrupt source descriptors), the driver should use the `intr_find` routine in order to obtain a pointer to a given interrupt source descriptor within the array.

The *intr_handler* argument specifies the interrupt handler invoked by the bus driver when an interrupt occurs.

intr_cookie specifies an argument which is passed back to the interrupt handler. If successful, the bus driver returns an *intr_ops* / *intr_id* pair. *intr_ops* points to the `BusIntrOps` structure which provides the service routines specific to the interrupt source connected (see section `BusIntrOps`).

intr_id is opaque for the device driver. It is passed back to the bus driver as an argument in subsequent invocations of the `BusIntrOps` and `intr_detach` service routines.

When the bus driver returns successfully from `intr_attach`, the corresponding interrupt source is enabled at bus level. Thus it may be necessary for the device

driver to disable device interrupts (at device level) prior to the `intr_attach` invocation. Depending on the bus, an interrupt source can be shared between multiple devices (for example, on a PCI bus). In such a case, multiple `intr_attach` requests can be issued on the same interrupt source.

The order in which these interrupt handlers are invoked is bus implementation specific. Usually, the handlers are invoked in the same order as they were attached (that is, first attached - first invoked). When the interrupt handler is invoked, the bus driver prevents re-entrance to the interrupt handler. Usually, interrupt acknowledgment at bus level is done by the bus driver once all interrupt handlers (attached to the serviced interrupt source) have been invoked. However, `BusIntrOps` provides the services allowing interrupts within the interrupt handler to be explicitly enabled and acknowledged (see section `BusIntrOps`). Note that the explicit interrupt acknowledgement enables re-entrance to the interrupt handler.

An interrupt handler must return a value specified by the `BusIntrStatus` type. An interrupt handler must return `BUS_INTR_UNCLAIMED` if the interrupt is unclaimed, indicating that there is no pending interrupt for the device. An interrupt handler must return `BUS_INTR_CLAIMED` if the interrupt has been claimed, meaning that there was a pending device interrupt which has been serviced by the handler.

In the case where an interrupt handler calls the `enable` routine, it must return the value returned by `enable`. Note that `enable` returns either `BUS_INTR_CLAIMED` or `BUS_INTR_ACKNOWLEDGED`.

The interrupt handler return code may potentially be used by the bus driver to detect spurious interrupts which have occurred on the bus. In addition, the return code notifies the bus driver whether the currently serviced interrupt has been acknowledged or not. This allows the bus driver to simplify (or even avoid) the current interrupt state management. In fact, the current interrupt state is kept by the corresponding interrupt handler rather than by the bus driver itself and the bus driver is notified about the interrupt state through the interrupt handler return code.

The `intr_detach` service routine disconnects the interrupt handler from the interrupt source. In other words, the corresponding interrupt routine is no longer invoked when interrupt occurs. *intr_id* specifies the bus driver handle returned by `intr_attach`.

BusIntrOps

```
typedef struct {

        void
    (*mask) (BusIntrId intr_id);
        void
    (*unmask) (BusIntrId intr_id);
```

```
        BusIntrStatus
    (*enable)  (BusIntrId intr_id);
        void
    (*disable) (BusIntrId intr_id);

} BusIntrOps;
```

The `mask` service routine masks the interrupt source specified by *intr_id*. Note that `mask` does not guarantee that all other interrupt sources are still unmasked.

The `unmask` service routine unmasks the interrupt source previously masked by `mask`. Note that `unmask` does not guarantee that the interrupt source is unmasked immediately. The real interrupt source unmasking may be deferred.

The `mask` / `unmask` pair may be used at either base or interrupt level. Note also that the `mask` / `unmask` pairs must not be nested. The `enable` and `disable` service routines are dedicated to interrupt handler usage only. In other words, these routines may be called only by an interrupt handler. The `enable` service routine enables (and acknowledges) the bus interrupt source specified by *intr_id*. `enable` returns either `BUS_INTR_ACKNOWLEDGED` or `BUS_INTR_CLAIMED`. The `BUS_INTR_ACKNOWLEDGED` return value means that the bus driver has enabled (and acknowledged) interrupts at bus level. The `BUS_INTR_CLAIMED` return value means that the bus driver has ignored the enable request and therefore the interrupt source is still disabled (and not acknowledged) at bus level.

Note that the bus driver typically refuses an explicit interrupt acknowledgement (issued by an interrupt handler) for the shared interrupts. In the latter case, the bus driver will acknowledge interrupts only when all interrupt handlers have been called. Note that in cases where the `enable` routine has been called by an interrupt handler, the handler must return the value which was returned by `enable`. Note also that once `enable` has been called, the driver should be able to handle an immediate re-entrance in the interrupt handler code.

The `disable` service routine disables the interrupt source previously enabled by `enable`. If `enable` returns `BUS_INTR_ACKNOWLEDGED`, the driver must call `disable` prior to returning from the interrupt handler. When an interrupt occurs, the attached *BusIntrHandler* is invoked with the interrupt source disabled at bus level. This has exactly the same effect as calling `disable` just prior to handler invocation. Note that the interrupt handler must return to the bus driver in the same context as it was called, that is, with the interrupt source disabled at bus level. On the other hand, the interrupt handler called may use the `enable`/`disable` pair to allow the interrupt to be nested. This feature is typically used by a bus-to-bus bridge driver when the secondary bus interrupts are multiplexed, that is, multiple secondary bus interrupts are reported through the same primary bus interrupt. Typically, an interrupt handler of this type of bus-to-bus bridge driver would take the following actions:

- Identify and disable the secondary bus interrupt source
- Enable the primary bus interrupt source (`enable`)
- Call handlers attached to the secondary bus interrupt source
- Disable the primary bus interrupt source (`disable`)
- Acknowledge (if needed) and enable the secondary bus interrupt source
- Return from the interrupt handler

io_map/io_unmap

The `io_map` service routine maps the contiguous I/O region to the supervisor address space. The *dev_id* argument specifies the given device on the given bus. It is an opaque returned by `bus_open`. The *io_regs* argument specifies a contiguous set of I/O registers. It points to an I/O registers set resource property value which is bus class specific.

The bus class API defines a structure which specifies the value format of this type of property attached to the device node. A device driver should find this type of property (attached to its device node) using the `"io-regs"` name, obtain a pointer to the property value and pass it to `io_map`. The property typically specifies the bus I/O space (for example, programmed or memory-mapped I/O on PCI bus), the start address and size of the I/O registers set.

When the property value specifies multiple I/O register sets, that is, the property value is an array of the I/O registers' set descriptors, the driver should use the `io_regs_find` routine in order to obtain a pointer to a given I/O register set descriptor within the array.

The *err_handler* argument specifies the access error handler which should be called when a bus error occurs during an access to the I/O region. The *err_cookie* argument is an opaque for the bus driver and is passed back to the device driver when the bus error handler is invoked.

Typically, a bus error is reported as an interrupt. Thus, the error handler is, in fact, an interrupt handler associated with an I/O region rather than with an interrupt source. When the error handler is invoked, all bus interrupts are disabled.

Note that some hardware does not support the bus error exception. On this type of hardware, the error handler is not invoked, and the system hangs.

If successful, the bus driver returns an *io_ops* / *io_id* pair. *io_ops* points to the `BusIoOps` structure which provides the access methods to the mapped I/O region.

io_id is opaque for the device driver. It is passed back to the bus driver as an argument in subsequent invocations of the `BusIoOps` and `io_unmap` service routines.

The `io_unmap` service routine destroys the mapping previously created by `io_map`. The *io_id* argument specifies the mapped I/O region.

BusIoOps

```
typedef struct {

        uint8_f
    (*load_8)  (BusIoId io_id, BusSize offset);
        uint16_f
    (*load_16) (BusIoId io_id, BusSize offset);
        uint32_f
    (*load_32) (BusIoId io_id, BusSize offset);
        uint64_f
    (*load_64) (BusIoId io_id, BusSize offset);

        void
    (*store_8) (BusIoId io_id, BusSize offset, uint8_f  val);
        void
    (*store_16) (BusIoId io_id, BusSize offset, uint16_f val);
        void
    (*store_32) (BusIoId io_id, BusSize offset, uint32_f val);
        void
    (*store_64) (BusIoId io_id, BusSize offset, uint64_f val);

        void
    (*read_8) (BusIoId io_id, BusSize off, uint8_f*  addr, BusSize cnt);
        void
    (*read_16) (BusIoId io_id, BusSize off, uint16_f* addr, BusSize cnt);
        void
    (*read_32) (BusIoId io_id, BusSize off, uint32_f* addr, BusSize cnt);
        void
    (*read_64) (BusIoId io_id, BusSize off, uint64_f* addr, BusSize cnt);

        void
    (*write_8) (BusIoId io_id, BusSize off, uint8_f*  addr, BusSize cnt);
        void
    (*write_16) (BusIoId io_id, BusSize off, uint16_f* addr, BusSize cnt);
        void
    (*write_32) (BusIoId io_id, BusSize off, uint32_f* addr, BusSize cnt);
        void
    (*write_64) (BusIoId io_id, BusSize off, uint64_f* addr, BusSize cnt);

} BusIoOps;
```

The bus driver provides four service routine sets to access a mapped I/O region:

- `load_xx`
- `store_xx`
- `read_xx`
- `write_xx`

There are four service routines in each set which deal with I/O registers of different width. `xx` specifies the I/O register size in bits: `8, 16, 32, 64`. In all service routines provided by `BusIoOps`, the *io_id* argument specifies the mapped I/O region and the *io_offset* argument specifies the register offset within the region.

The `load_xx` service routine loads a value from the I/O register.

The `store_xx` service routine stores a value into the I/O register. The *value* argument specifies the value being stored.

The `read_xx` service routine reads values sequentially from the I/O register and stores them into the memory buffer. The *addr* argument specifies the buffer start address. The *count* argument specifies the number of read-write transactions to perform.

The `write_xx` service routine reads values sequentially from the memory buffer and writes them into the I/O register. The *addr* argument specifies the buffer start address. The *count* argument specifies the number of read-write transactions to perform.

In a case where the CPU endian format differs from the bus endian format, the bus driver performs the endian format conversion.

mem_map/mem_unmap

The `mem_map` service routine is used to map a memory region from a device to the supervisor address space, enabling direct access (via a pointer) to this region.

The *dev_id* argument specifies the given device on the given bus. It is an opaque returned by `open`.

The *mem_rgn* argument specifies a memory region. It points to a memory region resource property value which is bus class specific. The bus class API defines a structure which specifies the value format of this type of property attached to the device node. The device driver identifies this type of property (attached to its device node) by using the `"mem-rgn"` name, obtains a pointer to the property value and passes it to `mem_map`.

The property typically specifies the start address and size of the memory region as well as its attributes. In a case where the property value specifies multiple memory regions, that is, the property value is an array of the memory region descriptors, the driver should use the `mem_rgn_find` routine in order to obtain a pointer to a given memory region descriptor within the array.

mem_attr specifies MMU attributes used for memory region mapping. A combination of the following flags is allowed:

`BUS_MEM_READABLE` The memory region is mapped as readable.

`BUS_MEM_WRITABLE` The memory region is mapped as writable.

BUS_MEM_EXECUTABLE	The memory region is mapped as executable.
BUS_MEM_CACHEABLE	The memory region is mapped as cacheable.
BUS_MEM_INVERTED	The memory region is mapped with an inverted endianness.

The BUS_MEM_INVERTED flag may be used by a device driver in order to avoid byte swapping within a memory mapped region. Note that mem_map returns K_EINVAL if this type of feature is not supported by MMU.

The *err_handler* argument specifies the access error handler which should be called when a bus error occurs during access to the memory region.

The *err_cookie* argument is an opaque for the bus driver. It is passed back to the device driver when the bus error handler is invoked.

If successful, the bus driver returns a *mem_addr* pointer which points to the first byte of the memory region (in the supervisor address space), and *mem_id* which is an opaque for the device driver. *mem_id* is passed back to the bus driver as an argument in a subsequent invocation of the mem_unmap service routine.

The device driver may now access the device memory by dereferencing the returned pointer. Note that in this mode, the driver must handle any endianning problems by itself. Note also that the bus endianness is available as the "byte-order" property attached to the bus node.

The "byte-order" property value is a 32-bit integer constant. Each byte within the constant contains its offset in the memory. In other words, the 0x00010203 and 0x03020100 hexadecimal constants define the big and little endian byte orders respectively.

The mem_unmap service routine destroys the mapping previously created by mem_map. The *mem_id* argument specifies the mapped memory region.

Error Handler

```
typedef struct {
    BusErrorCode  code;
    BusSize       offset;
} BusError;
```

When either an I/O or memory access is aborted because of an error, the bus driver invokes the error handler which was specified by the device driver in io_map or mem_map. *err_cookie* is passed back to the error handler as the first argument.

The second argument points to the BusError structure which provides additional information about the fault. The *code* field specifies the fault type

which is mainly bus class specific. The common bus driver API specifies only two error codes:

`BUS_ERR_UNKNOWN`	Unknown error type, in other words, the bus driver is unable to determine the reason for the exception.
`BUS_ERR_INVALID_SIZE`	Invalid (in other words, not supported) because access granularity has been used.

The *offset* field specifies the fault address offset within the I/O or memory region.

The `BusError` structure can be extended by a specific bus driver in order to include additional (bus dependent) fields. Typically, a bus error is reported as an interrupt. The error handler is thus an interrupt handler associated with an I/O or memory region rather than with an interrupt source. When the error handler is invoked, all bus interrupts are disabled.

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
BusOps.open	-	+	-	+
BusOps.close	-	+		+
BusOps.intr_attach	-	+	-	+
BusOps.intr_detach	-	+	-	+
BusOps.io_map	-	+	-	+
BusOps.io_unmap	-	+	-	+
BusOps.mem_map	-	+	-	+
BusOps.mem_unmap	-	+	-	+
BusOps.intr_find	+	+	-	-
BusOps.io_regs_find	+	+	-	-
BusOps.mem_rgn_find	+	+	-	-
BusIntrOps.mask	+	+	+	-
BusIntrOps.unmask	+	+	+	-
BusIntrOps.enable	-	-	+	-
BusIntrOps.disable	-	-	+	-
BusIoOps.load_xx	+	+	+	-

BusIoOps.store_xx	+	+	+	-
BusIoOps.read_xx	+	+	+	-
BusIoOps.write_xx	+	+	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`dtreeNodeRoot`(9DKI), `svDriverRegister`(9DKI), `svMemAlloc`(9DKI), `svPhysAlloc`(9DKI), `svPhysMap`(9DKI), `svDkiThreadCall`(9DKI), `svTimeoutSet`(9DKI), `usecBusyWait`(9DKI), `DISABLE_PREEMPT`(9DKI)

NAME

ether – Ethernet Device Driver Interface

SYNOPSIS

```
#include <ddi/net/ether/etherProp.h>
#include <ddi/net/ether/ether.h>
```

FEATURES

DDI

DESCRIPTION

Provides ethernet driver interface services.

EXTENDED DESCRIPTION

Ethernet Device Generic Properties

The Ethernet device's generic properties can only be set by the device driver at device initialization time and are read-only for the Ethernet client.

```
#define ETHER_PROP_ADDR           "ether-addr"
#define ETHER_PROP_THROUGHPUT     "link-throughput"


typedef uint8_f EtherPropAddr[6];

typedef enum {
    ETHER_10_MBS  =  10000000, /*  10 Megabits link */
    ETHER_100_MBS = 100000000  /* 100 Megabits link */
} EtherPropThroughput;
```

The value associated with the ETHER_PROP_ADDR property is an array of 6 bytes which contain the Ethernet Address of the device. The ETHER_PROP_ADDR property is mandatory for the client of the device. By default, its value is set by the device driver with the Ethernet address probed from the device's ROM.

The value associated with the ETHER_PROP_THROUGHPUT property is a 32 bit unsigned integer which specifies the throughput of the underlying Ethernet Link in bits per second. It must be set to 10000000 on a 10 Megabits link and to 100000000 on a 100 Megabits link. The ETHER_PROP_THROUGHPUT property is mandatory for the client of the device. By default, its value is set by the device driver which probes it from the device.

Ethernet Device Driver Service Routines

The Ethernet device driver service routines are declared by the EtherDevOps structure shown below. A pointer to the EtherDevOps structure is exported by the driver via the svDeviceRegister microkernel call. A driver client invokes the svDeviceLookup and svDeviceEntry microkernel calls in order to obtain a pointer to the device service routines vector.

```
#define ETHER_CLASS "ethernet"

typedef enum {
    ETHER_VERSION_INITIAL = 0,
} EtherVersion;
```

```
typedef struct {
    EtherVersion version;

        KnError
    (*open) (void* devId, void* client_cookie, EtherCallBack* clientOps);
        void
    (*close) (void* device_id);
        KnError
    (*frameTransmit) (void* device_id, NetFrame* outFrame);
        KnError
    (*frameReceive) (void* device_id, NetFrame* rcvFrame);
        void
    (*promiscuousEnable) (void* device_id);
        void
    (*promiscuousDisable) (void* device_id);
        KnError
    } EtherDevOps;

typedef struct EtherMcast {
     struct EtherMcast *next;
     char   mcastAddr[6];
} EtherMcast;

void
setMcastAddr(void* device_id, unsigned int nb, EtherMcast* mcastAddrs)
```

Arguments

device_id	The identifier assigned to the device by the device driver
client_cookie	The identifier assigned to the device by the Ethernet client
clientOps	A pointer to an `EtherCallBack` structure which contains the set of functions exported by the Ethernet client of the device

Synopsis:

The `open` function is only invoked by the Ethernet client of the device to start communication over the underlying Ethernet device designated by the first argument *device_id* . The device driver must record in the structure it associates to the device the *client_cookie* and the *clientOps* arguments to make further upcalls to the Ethernet client. The *client_cookie* is the first argument of every function exported by the Ethernet client to the device driver.

The device driver must start the Ethernet device in the following default receive mode:

- Receipt of unicast and broadcast frames (no multicast)

- Promiscuous disabled

Invocation Context:

The `open` function is always invoked at base level. The Ethernet client ensures the same device is not opened again before having closed it.

Return Value:

`K_OK`	If successful
`K_EFAIL`	Otherwise

`close`

device_id is the identifier assigned to the device by the device driver.

The `close` function tells the Ethernet driver that the device is no longer used by the Ethernet client. The `close` function must stop the Ethernet device cleanly. In particular, it must arrange to terminate all DMA-based I/O operations, if any, and to disable all device interrupts. The `close` function is responsible for ensuring that all the functions exported by the Ethernet client will no longer be invoked after it returns to the client. In addition, `close` must only return after all output frames which are currently transmitted have been freed.

Invocation Context:

The `close` function is only invoked once after a successful invocation of `open` for the same device. The `close` function is always invoked at base level. The `close` function is invoked in mutual exclusion with all other functions exported by the device driver on the same device. The Ethernet client guarantees that all the other functions exported by the device driver will never be invoked again (on the same device) after it invoked `close` .

`frameTransmit`

device_id is the identifier assigned to the device by the device driver. *outFrame* is the network frame to transmit.

Synopsis:

The `frameTransmit` function of the device driver is invoked by the Ethernet client to transmit an output frame. If the Ethernet device has enough transmission resources to start the transmit operation, `frameTransmit` returns immediately without waiting for the frame transmission to be completed. In this case, the device driver is responsible for freeing the output frame by invoking the `freeFrame` function embedded within the `NetFrame` structure associated to the output frame.

Invocation Context:

The `frameTransmit` function is guaranteed not to be re-entered. The `frameTransmit` function can be invoked at interrupt level.

Return Value:

`K_OK`	If the transmit operation has been started successfully
`K_EFAIL`	Otherwise

`frameReceive`

Argument(s):

device_id	Is the identifier assigned to the device by the device driver.
rcvFrame	Is a pointer to a structure of type `NetFrame` into which `frameReceive` must return: ■ The total size of the received frame in the `frameSize` field ■ List of network memory buffers which contain the received frame data in the `bufList` field

Synopsis:

The `frameReceive` function of the device driver is invoked by the Ethernet client of the device to retrieve the next input frame received by the device, if any.

If an input frame is available, `frameReceive` must invoke the `netBufAlloc` function exported by the Ethernet client to allocate a list of network memory buffers to which to copy the frame's content. Then, `frameReceive` returns this list of buffers and the total size of the frame into the `NetFrame` structure pointed to by its *rcvFrame* output argument.

In the presence of an erroneous input frame, `frameReceive` must notify the I/O error through an upcall to the `ioErrorNotify` function exported by the Ethernet client and skip the erroneous frame.

Invocation Context:

The `frameReceive` function is only invoked once the device driver has notified the Ethernet client through an upcall to `receiptNotify` . The `frameReceive` function can be invoked at interrupt level. The `frameReceive` function is guaranteed not to be re-entered.

Return Value:

K_OK	If successful.
K_EFAIL	If no input frame is available.
K_ENOMEM	If the allocation of network memory buffers failed.

`promiscuousEnable`

Argument(s):

device_id is the identifier assigned to the device by the device driver.

Synopsis:

The `promiscuousEnable` function starts the receipt of all Ethernet frames which are carried over the Ethernet link.

Invocation Context:

The Ethernet client ensures that `promiscuousEnable` is invoked in mutual exclusion with the `promiscuousDisable` function described below. The `promiscuousEnable` function is not re-entered. The `promiscuousEnable` function is always invoked at base level.

`promiscuousDisable`

Argument(s):

device_id is the identifier assigned to the device by the device driver.

Synopsis:

The `promiscuousDisable` function puts the Ethernet device in the default receipt mode.

Invocation Context:

The Ethernet client ensures that `promiscuousDisable` is invoked in mutual exclusion with the `promiscuousEnable` function described above. The `promiscuousDisable` function is not re-entered. The `promiscuousDisable` function is always invoked at base level.

`setMcastAddr`

Argument(s):

device_id	Is the identifier assigned to the device by the device driver.

nb	Is the number of multicast addresses given in the list.
mcastAddrs	Points to the item's 6–byte multicast ethernet address.

Synopsis:

The `setMcastAddr` function is invoked to reset the Ethernet Multicast Address (which is invoked by the device's controller).

The controller does not maintain the current or previous state. Each time `setMcastAddr` is called, the state is reset. It is the responsability of the upper layers to keep track of the multicast addresses, as well as additions / deletions.

All addresses are guaranteed to be unique multicast addresses.

This function can invoke three reset states for the multicast address, depening on the value of the `nb` argument.

nb = 0	The controller state is reset to no-multicast mode.
nb > 0	The controller filters the given multicast addresses. If there are too many for it to handle, the controller switches automatically to the "all-multicast" mode. In this mode, no addresses are filtered, and therefore upper layers must handle filtering functions.

nb = ETHDEV_MCAST_ALL controller switches to "all-multicast" mode.

Invocation Context:

The `setMcastAddr` function is not re-entered, and is always invoked at base level.

Ethernet Device Client Up-Call Handlers

The set of functions exported by the Ethernet client is represented by the `EtherCallBack` structure. A pointer to a structure of type `EtherCallBack` is provided to the Ethernet driver by the Ethernet client as an argument of the `open` function.

```
typedef enum {
    ETHER_TX_CARRIER_ERROR,
    ETHER_TX_HEARTBEAT_ERROR,
    ETHER_TX_COLLISION_ERROR,
    ETHER_RX_TOOLONG_ERROR,
    ETHER_RX_CRC_ERROR,
    ETHER_RX_FIFO_ERROR,
} EtherIoError;
```

```
typedef enum {
    ETHER_LINK_UP,
    ETHER_LINK_DOWN,
} EtherLinkState;


typedef struct {
        void
    (*transmitNotify) (void* client_cookie);
        void
    (*receiptNotify) (void* client_cookie);
        NetBuf*
    (*netBufAlloc) (void* client_cookie, uint32_f frameSize);
        void
    (*netBufFree) (void* client_cookie, NetBuf* bufList);
        void
    (*ioErrorNotify) (void* client_cookie, EtherIoError ioError);
        void
    (*linkStateNotify) (void* client_cookie, EtherLinkState linkState);
} EtherCallBack;
```

transmitNotify

Argument(s):

device_id	Is the identifier assigned to the device by the Ethernet client.

Synopsis:

The `transmitNotify` function must be invoked by the device driver to inform the Ethernet client that transmission resources of the Ethernet device are available. If pending output frames need to be transmitted, the Ethernet client will invoke the `frameTransmit` function of the device driver, either in sequence or asynchronously.

Invocation Context:

The `transmitNotify` function can be invoked at interrupt level, which is usually the case. The device driver is allowed to invoke `transmitNotify` at any moment and multiple times, even if no output frame needs to be transmitted.

receiptNotify

Argument(s):

device_id	Is the identifier assigned to the device by the Ethernet client.

Synopsis:

The `receiptNotify` function must be invoked by the device driver to inform the Ethernet client that new input frames have been received by the Ethernet device. As a result, the Ethernet client will invoke the `frameReceive` function of the device driver, either in sequence or asynchronously.

Invocation Context:

The `receiptNotify` function can be invoked at interrupt level, which is usually the case. The device driver is allowed to invoke `receiptNotify` at any moment and multiple times, even if no (valid) input frame has been received.

`netBufAlloc`

Argument(s):

device_id	Is the identifier assigned to the device by the Ethernet client
frameSize	Is the requested memory size

Synopsis:

The `netBufAlloc` function must be invoked by the `frameReceive` function of the device driver to allocate a list of network memory buffers into which it copies the contents of the next received frame. This list of memory buffers is returned to the Ethernet client into the second argument of the `frameReceive` function.

Invocation Context:

The `netBufAlloc` function can only by invoked by the `frameReceive` function of the device driver.

Return Value:

If successful	A list of NetBuf structures which fit the requested size
Otherwise	NULL

`netBufFree`

Argument(s):

device_id	Is the identifier assigned to the device by the Ethernet client
netBuff	Is the list of network memory buffers to free

Synopsis:

The `netBufFree` function must be invoked by the frameReceive function of the device driver to free network memory buffers which it previously allocated through an upcall to the `netBufAlloc` function. This usually occurs when `frameReceive` detects an error while copying the contents of the next input frame from the device into the memory buffers.

Invocation Context:

The `netBufFree` function can only by invoked by the `frameReceive` function of the device driver.

`ioErrorNotify`

Argument(s):

device_id	Is the identifier assigned to the device by the device driver
ioError	Specifies the I/O error detected by the device controller

Synopsis:

The `ioErrorNotify` function is invoked by the device driver to notify the Ethernet client of an I/O related error detected by the Ethernet device controller. The `ioErrorNotify` function is mainly used for recording statistics.

Invocation Context:

This function can be invoked at interrupt level.

`linkStateNotify`

Argument(s):

device_id	Is the identifier assigned to the device by the device driver
linkState	Specifies the new state of the Ethernet link detected by the Ethernet device controller

Synopsis:

The `linkStateNotify` function provides to the Ethernet client the new state of the Ethernet link detected by the Ethernet device controller.

Invocation Context:

This function can be invoked at interrupt level.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`dtreeNodeRoot`(9DKI), `svDeviceRegister`(9DKI), `svDriverRegister`(9DKI), `netFrame`(9DDI)

NAME

flash – Flash Device Driver Interface

SYNOPSIS

```
#include <ddi/flash/flash.h>
```

FEATURES

DDI

DESCRIPTION

The flash device operates asynchronously, as any other device may operate asynchronously. See section Asynchronous Operation for details (at the end of this page). Note that the synchronous operation model is mappable to the asynchronous model.

Device properties are attached to a device's node in a device tree (that stores all device nodes for all devices). The device properties supply information to drivers and upper layer files. The generic flash device is defined by its properties. An important property in the flash device node is "endurance". This property defines the device's media, which specifies that one more translation layer must be used to organize media usage.

The flash device is defined as a variable block size access device. The block access functionality is defines `read`/`write` and `erase` access properties. These properties define the block size and alignment for each kind of functionality. For example, the `read` block access only allows the flash device to provide minimal functionality. All other functionality is optional. Some flash devices use an erase-on-write method. For these devices, the erase block access property must not be specified. The absence of this property informs the upper layers of the erase-on-write settings. Note that devices such as SRAM and ROM may also export the flash device interface in order to integrate with the upper layers. The absence of the block-write property informs the upper layers that the medium is read-only.

Some flash devices are mapped to memory. Such devices may be accessed (at least read) directly. For devices that are memory mapped, the flash interface defines a variable region size with `lock`/`unlock` functionality. Devices which allow execution in place (XIP) should specify region-exec properties.

All other device specifics, as well as board mounting issues, should be handled inside the flash device driver implementation.

Service Routines

The character string "flash" labels the flash device class. The device clients use the device class name to obtain the driver service routines from the device registry as described in `svDeviceLookup`(9DKI). Note that the flash device interface specifies flash as a single thread and as a single client device. This means that only one client thread, at a any given time, may have access to the device. It is a client's responsibility to avoid re-entrance of down-calls to the driver. Usually, this is a file or block translation layer agent.

Most functions within the flash device interface are base-level methods and can not therefore be invoked from the interrupt level. Exceptions to this are

asynchronous operations (`read`/`write`/`erase`), which may be started in the context of a device asynchronous operation handler.

The `FlashOps` structure specifies the interface of the flash device driver:

```
typedef struct FlashOps {

     FlashVersion version; /* version number of the Flash driver API */

        void
    (*open)     (FlashId  dev_id,  /*registry device id*/
                 FlashAsync*  async_tbl); /*async-io handlers*/
        void
    (*close)    (FlashId dev_id);

        void
    (*mask)     (FlashId dev_id);

         void
    (*unmask)   (FlashId dev_id);

        KnError
    (*set_verify)(FlashIoId dev_id, Bool f_on);

        KnError
    (*lock)     (FlashId     dev_id,      /*device id */
                 FlashOffset  offset,      /*start offset (bytes)*/
                 FlashSize    size,        /*region size in bytes*/
                 FlashAsyncInfo* info,     /*driver will fill it*/
                 FlashRgnId*  flash_rgn_id,/*returned by driver*/
                 PhAddr*         rgn_addr);/* returned phys. addr*/

        void
    (*unlock)   (FlashRgnId  flash_rgn_id);/*locked region id*/

        KnError
    (*read)     (FlashId     dev_id,      /*device id */
                 FlashOffset  offset,      /*start offset (bytes)*/
                 FlashSize    size,        /*buffer size in bytes*/
                 FlashBuffer  buff,        /*buffer address */
                 void*        io_cookie,   /*user given cookie */
                 FlashIoId*   flash_io_id);/*returned by driver*/

        KnError
    (*write)    (FlashId     dev_id,      /*device id */
                 FlashOffset  offset,      /*start offset (bytes)*/
                 FlashSize    size,        /*buffer size in bytes*/
                 FlashBuffer  buff,        /*buffer address */
                 void*        io_cookie,   /*user given cookie */
                 FlashIoId*   flash_io_id);/*returned by driver*/

        KnError
    (*erase)    (FlashId     dev_id,      /*device id */
                 FlashOffset  offset,      /*start offset (bytes)*/
                 FlashSize    size,        /*buffer size in bytes*/
```

```
                void*        io_cookie,   /*user given cookie */
                FlashIoId*   flash_io_id);/*returned by driver*/

        KnError
    (*erase_media) (FlashId   dev_id,      /*device id */
                    void*     io_cookie,   /*user given cookie */
                    FlashIoId* flash_io_id);/*returned by driver*/

        void
    (*abort_io) (FlashIoId io_id);

} FlashOps;
```

A pointer to the `FlashOps` structure is exported by the driver via the `svDeviceRegister()` microkernel call. A driver client invokes the `svDeviceLookup()` and `svDeviceEntry()` microkernel calls in order to obtain a pointer to the device service routines table. Once a pointer is obtained, the driver client is able to invoke the driver service routines (via an indirect function call).

open

```
void
(*open)   (FlashId    dev_id,     /* registery device id */
          FlashAsync* async_tbl); /* async-io handlers */
```

This call takes the following arguments:

dev_id

Specifies the Flash device identifier

async_tbl

Points to the `FlashAsync` structure, filled with asynchronous user callback handlers for each kind of operation

The `FlashAsync` structure is shown below:

```
typedef struct FlashAsync
    /* the async-io handlers for flash device*/
    AsyncIoHandler  read_io;    /* read-async io handler*/
    AsyncIoHandler  write_io;   /* write-async io handler*/
    AsyncIoHandler  erase_io;   /* erase-async io handler*/
 FlashAsync;
```

The `AsyncIoHandler` type is defined in `<ddi/async.h>`.

The `open()` function establishes a connection between device and device client. The `open()` function does not return a value. `open()` puts the device in a masked state. The `unmask()` down-call should be invoked in order to obtain device functionality.

close

```
void
(*close)   (FlashId  dev_id);  /* registery device id */
```

This call takes the following argument:

dev_id
Specifies the Flash device identifier

The `close()` function terminates work with the device. Note that all started operations must be finished or cancelled before `close()` is called, and the device should be put in a masked state.

mask

```
void
(*mask)     (FlashId  dev_id);  /* registery device id */
```

This call takes the following argument:

dev_id
Specifies the Flash device identifier

The `mask()` function disables the asynchronous handler's invocation.

unmask

```
void
(*ummask)   (FlashId dev_id);  /* registery device id */
```

This call takes the following argument:

dev_id
Specifies the Flash device identifier

The `unmask()` function enables the asynchronous handler's invocation.

Typically, the `mask()`/`unmask()` pair is used by the device driver client to implement a critical section of code, which must be synchronized with asynchronous handlers.

set_verify

```
void
(*set_verify) (FlashId dev_id, Bool f_on);
```

This call takes the following arguments:

dev_id
Specifies the Flash device identifier

f_on
Zero value disables `write`/`erase` verification functionality

The `set_verify()` function disables or enables the `write`/`erase` verification functionality. Note that devices that do not have built-in hardware verification should implement software verification. The default state of verification functionality for all flash devices is "on".

The `set_verify()` function returns the following results:

K_OK
The verification functionality was successfully switched.

K_ENOTIMP
The verification functionality can not be switched off.

lock

```
KnError
(*lock)  (FlashId        dev_id,   /* device id */
         FlashOffset     offset,   /* start offset (bytes)*/
         FlashSize       size,     /* region size in bytes*/
         FlashAsyncInfo* info,     /* driver will fill it */
         FlashRgnId*     flash_rgn_id, /* returned by driver */
         PhAddr*         rgn_addr);    /* returned phys. addr*/
```

This call takes the following arguments:

dev_id
Specifies the Flash device identifier

offset
Byte offset of the region to lock

size
Size of the region to lock in bytes

info
Pointer to the structure, returned by the device driver in case of error

flash_rgn_id
Region ID returned by the device driver

rgn_addr
Physical address of the locked region returned by the device driver

The `lock( )` function can only be implemented for flash media that are mapped to memory. The flash region locked in memory should be mapped into virtual space by the device driver client, according to the device node properties. The `lock( )` function hides the media implementation and board-mounting specifics, like banking (layers) and interleaving.

The `lock( )` function returns the following results:

K_OK
The region was successfully locked.

K_EBUSY
Another region that is already locked is blocking the lock.

K_EFULL
An asynchronous device operation is blocking the lock.

K_ENOTIMP

The device is not memory mapped media.

unlock

```
void
(*unlock) (FlashRgnId flash_rgn_id);
```

This call takes the following argument:

flash_rgn_id
Identifies a previously locked region

The `unlock()` function frees a previously locked region. The device client should not access the region directly after it has been unlocked. All regions should be unlocked before closing the device.

read

```
KnError
(*read)   (FlashId        dev_id,       /* device id */
           FlashOffset    offset,       /* start offset (bytes)*/
           FlashSize      size,         /* buffer size in bytes*/
           FlashBuffer    buff,         /* buffer address */
           void*          io_cookie,    /* user given cookie */
           FlashIoId*     flash_io_id); /* returned by driver*/
```

This call takes the following arguments:

dev_id
Specifies the flash device identifier

offset
Byte offset of the block of media

size
Size of the block to read in bytes

buff
Pointer to the client buffer to fill, aligned as specified by the device properties

io_cookie
Device client cookie; this value is passed to the asynchronous handler

flash_io_id
Device I/O operation identifier; transparent to the client

The `read()` function is to be implemented by all kinds of flash devices. This is a minimal common functionality for all media types. The `read()` function either starts an asynchronous read operation on the device or fails if the device can not accept more operations. Parameters (size and alignment) must respect the block-read property specified for the device node.

The `read()` function returns the following results:

K_OK
The read operation has been started on the device.

K_EFULL
The device cannot accept more asynchronous device operations.

write

```
KnError
(*write)  (FlashId        dev_id,       /* device id */
          FlashOffset     offset,       /* start offset (bytes)*/
          FlashSize       size,         /* buffer size in bytes*/
          FlashBuffer     buff,         /* buffer address */
          void*           io_cookie,    /* user given cookie */
          FlashIoId*      flash_io_id); /* returned by driver*/
```

This call takes the following arguments:

dev_id
Specifies the flash device identifier

offset
Byte offset of the block of media

size
Size of the block to read in bytes

buff
Pointer to the client buffer to fill, aligned using the device property settings

io_cookie
Device client cookie; this value is passed to the asynchronous handler

flash_io_id
Device I/O operation identifier; transparent to the client

The `write()` function either starts an asynchronous write operation on the device or fails if the device can not accept more operations. Parameters (size and alignment) must respect the block-write property specified for the device node. The `write()` function should not be implemented for read-only media. If it is, however, the block-write property must not be specified in the device node.

The `write()` function returns the following results:

K_OK
The write operation has been started on the device.

K_EFULL
The device cannot accept more asynchronous device operations.

K_ENOTIMP
The media is non-writable.

erase

```
KnError
(*erase)  (FlashId        dev_id,       /* device id */
          FlashOffset     offset,       /* start offset (bytes)*/
          FlashSize       size,         /* size in bytes*/
          void*           io_cookie,    /* user given cookie */
          FlashIoId*      flash_io_id); /* returned by driver*/
```

This call takes the following arguments:

dev_id
Specifies the Flash device identifier

offset
Byte offset of the block of media

size
Size of the block to read in bytes

io_cookie
Device client cookie; this value is passed to the asynchronous handler

flash_io_id
Device I/O operation identifier; transparent to the client

The `erase()` function either starts an asynchronous partial erase operation on the device or fails if the device cannot accept more operations. Parameters (size and alignment) must respect the block-erase property specified for the device node. The `erase()` function should be not be implemented for read-only, write-once or erase-on-write media. If the media is non-erasable then the block-erase property should not be specified in the device node.

The `erase()` function returns the following results:

`K_OK`
The erase operation has been started on the device.

`K_EFULL`
The device cannot accept more asynchronous device operations.

`K_ENOTIMP`
The media is non-erasable..

erase_media

```
KnError
(*erase_media) (FlashId     dev_id,       /* device id */
                void*       io_cookie,    /* user given cookie */
                FlashIoId*  flash_io_id); /* returned by driver*/
```

This call takes the following arguments:

dev_id
Specifies the flash device identifier

io_cookie

Device client cookie; this value is passed to the asynchronous handler

flash_io_id
Device I/O operation identifier; transparent to the client

The `erase_media()` function either starts a full erase operation on the device or fails if the device cannot accept more operations. This operation is rarely used and should be used only when the upper logical layer cannot recover the logical structure. Normally, all erasable devices implement this function, even if it is emulated sector by sector. This function may not be implemented for read-only, write-once or erase-on-write media. If the media is non-erasable then the block-erase property should not be specified in the device node.

The `erase_media()` function returns the following results:

`K_OK`
The erase operation has been started on the device.

`K_EFULL`
The device cannot accept more asynchronous device operations.

`K_ENOTIMP`
The media is non-erasable.

abort_io

```
void
(*abort_io) (FlashIoId   io_id);
```

This call takes the following argument:

io_id
Identifies the asynchronous operation

The `abort_io()` initiates cancellation of the specified asynchronous operation. The *codetextio_id* must be valid. The best way to ensure this is to call this function on a previously masked device. The function returns immediately and the device driver client is notified via an asynchronous I/O handler invocation. Note that the operation may return a successful notification even after `abort_io()` has been called.

Up-Call Handlers

The driver client up-call handlers are specified in the `codetextFlashAsync` structure and passed to the device driver as an argument of `open()`. All handlers are prototyped as:

```
typedef enum {
   FLASH_ERR_OK    = 0,
   FLASH_ERR_READ,       /*read errors (internal check) */
   FLASH_ERR_WRITE,      /*write errors (internal check) */
   FLASH_ERR_NOT_ERASED, /*attempt to write '1' over '0' */
   FLASH_ERR_PROTECTED,  /*attempt to write/erase protected zone */
   FLASH_ERR_ERASE,      /*errors on erase */
   FLASH_ERR_VERIFY,     /*write/erase verify errors (hard- or software)*/
```

```
   FLASH_ERR_LOCKED,     /*another region locks the view */
   FLASH_ERR_IO,         /*concurrent I/O is in progress in same area */
   FLASH_ERR_POWER       /*no power to perform write/erase */
} FlashIoError;

typedef struct FlashAsyncInfo { /*pointed by async_info parameter in async_io*/
    FlashIoError  error; /*enumerated error code (see above) */
    uint32_f      count; /*bytes transferred (proceed) so far */
    void*         id;    /*either FlashIoId in progress or FlashRgnId */
    void*         cookie;/*user given cookie for blocking io_id or rgn_id*/
} FlashAsyncInfo;

typedef void (*AsyncIoHandler)   (void*            io_cookie,
                                  KnError          async_err,
                                  FlashAsyncIoInfo  async_info);
```

This call takes the following arguments:

io_cookie

Device client cookie given at the start of the operation

async_err

An error code having one of the following values:

K_OK	Operation terminated successfully
K_EABORT	Operation has been cancelled
K_EFAIL	Operation failed

async_info

additional error information, given in cases of error when not `NULL`, points to the `FlashAsyncInfo` structure

All up-call handlers are invoked in a masked state. Only interrupt level routines are available in the context of the up-call handlers, since an asynchronous I/O handler should be treated as an interrupt. However, the device driver client may try to start more asynchronous operations from the asynchronous I/O handler.

Asynchronous Operation

Most devices are asynchronous; their I/O operation is initiated by the direct invocation of one of the device driver routines. This type of execution, however, can take a considerable time. It is the device's responsibility to notify the end of an operation, normally via an interrupt. The device driver interrupt routine can determine whether the operation has successfully terminated or failed. Another end-condition of an operation (usually fail) is a time-out, which is also an interrupt routine invocation.

The general asynchronous IO handler is represented as:

```
typedef void* AsyncIoInfo;
/* device dependent information structure */

typedef void (*AsyncIoHandler)   (void*       io_cookie,
```

```
                KnError      async_err,
                AsyncIoInfo  async_info);
```

`io_cookie` is a device client pointer, given on operation initiation.

`async_err` is an error code, which may have the following values:

`K_OK`	The operation terminated successfully.
`K_EFAIL`	The operation has failed for some reason. Additional error information may be returned via `async_info structure`, however this is device specific.
`K_EABORT`	The operation was aborted (if possible) by the cancel() driver call.

`async_info` is a pointer to the device specific structure, filled by the device driver.

Synchronization Issues

The asynchronous I/O execution model described above raises some synchronization principles.

Each asynchronous operation initiation returns an *io_id* reference which is valid until the operation returns from the asynchronous operation handler. The scope of `AsyncIoHandler` should be treated as an interrupt handler, which is normally the case. This means that only the interrupt-level routines are available in `AsyncIoHandler`. The device driver ensures non-reentrancy to each `AsyncIoHandler`, specified in the device driver interface. The content of the `async_info` structure remains valid until an operation returns from the handler. Some devices, however, may allow the initiation of further operations from the handler. In this case, the other operation being initiated invalidates the content of the `async_info` structure, as well as the *io_id* which may be recycled by the device driver.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDeviceRegister`(9DKI), `svDriverRegister`(9DKI), `svDeviceLookup`(9DKI), `svDeviceEntry`(9DKI), `svDeviceRelease`(9DKI)

NAME

isa – ISA Bus Driver Interface

SYNOPSIS

```
#include <ddi/isa/isa.h>
```

FEATURES

DDI

DESCRIPTION

Provides an API for development of ISA device drivers.

EXTENDED DESCRIPTION

The ISA bus driver offers an API for the ISA device drivers' development. This ISA bus API is an abstraction of the low-level ISA bus services and covers the following ISA functional modules:

- ISA interrupts management
- PIO and memory-mapped device registers access
- Device memory access
- DMA management

Connecting a Device Driver to the ISA Bus

First of all, an ISA device driver must register itself in the kernel driver registry. This should be done from its main routine using `svDriverRegister` The driver must set the bus class to `"isa"` in its registry entry, and specify the lowest version number of ISA bus interface required to run correctly. This registration allows the ISA bus driver to call back the ISA device driver's `drv_probe` and `drv_init` routines at bus probing and bus initialization time, respectively.

Within the `drv_probe` or `drv_init` routine, the ISA device driver may establish a connection with the parent ISA bus driver as described below. Once such a connection is established, the ISA device driver may use services provided by the ISA bus driver. Note that a connection to the bus driver can be established only within the driver's `drv_probe` or `drv_init` routine.

Initialization Connection

If the `drv_init` routine is defined the ISA bus driver invokes it at bus initialization time.

Note that the `drv_init` routine is called in the context of a DKI thread which makes it possible to invoke the ISA bus services allowed in the DKI thread context only. The `drv_init` routine is defined if the *drv_init* field of the driver registry entry is set to a non NULL value.

The `drv_init` routine is called with three arguments:

DevNode	The ISA device's own node, which identifies the node associated with the device in the device tree.
IsaBusOps	The ISA bus operations structure, which defines the ISA bus API and its version.

IsaBusId — The ISA bus identifier, which is an opaque to pass back when calling the operation `IsaBusOps.open` to connect the device driver to the ISA bus.

From this point, `IsaBusOps.open` must be the first call issued by the ISA device driver to the ISA bus driver:

```
    KnError
(*open) (IsaId           busId,
         DevNode         devNode,
         IsaEventHandler eventHandler,
         IsaLoadHandler  loadHandler,
         void*           cookie,
         IsaDevId*       devId);
```

This establishes a connection to the ISA bus identified by the *busId*. The returned parameter *devId* identifies the new connection.

This *devId* identifier is an argument for many other services defined in the `IsaBusOps` structure.

busId and *devNode* are given by the ISA bus driver as parameters to the driver's `drv_init` routine.

eventHandler is a handler in the device driver which is invoked by the ISA bus driver, when an ISA bus event occurs. Note that *eventHandler* is optional and must be set to NULL when it is not implemented by the device driver. The *cookie* argument is passed back to this handler as first argument.

This ISA bus event handler takes two additionnal parameters.

The first additional parameter is the bus event type, which is one of the following:

`ISA_SYS_SHUTDOWN` — Notifies a device driver that the system is going to be shut down. The device driver should reset the device hardware and return from the event handler. Note that the driver must neither notify clients nor free allocated resources.

`ISA_DEV_SHUTDOWN` — Notifies a device driver that the device should be shut down. The device driver should notify driver clients (via `svDeviceShutDown`) that the device is going to be shut down and should then return from the event handler. Once the device entry is released by the last driver client, the device registry module invokes a driver call-back handler. Within this handler, the device driver

should reset the device hardware, release all used resources and close the bus connection invoking `IsaBusOps.close`. Note that the `ISA_DEV_SHUTDOWN` event may be used by a bus driver in order to confiscate (or to re-allocate) bus resources.

`ISA_DEV_REMOVAL` — Notifies a device driver that the device has been removed from the bus and therefore the device driver instance has to be shut down. The actions taken by the driver are similar to the `ISA_DEV_SHUTDOWN` case except that the device hardware must not be accessed by the driver.

The second additional parameter is an event type specific structure pointer: this argument is always NULL for ISA bus events.

As the event handler may be executed in the context of an interrupt, its implementation must be restricted to the API allowed at interrupt level.

loadHandler is a handler in the device driver which is invoked by the ISA bus driver, when a new driver appears in the system (for example, a new driver is downloaded at run time). Note that *loadHandler* is optional and must be set to NULL when it is not implemented by the device driver. The *cookie* argument is also passed back to this handler as a unique argument. Typically, a leaf ISA device driver is not affected by this type of event. *loadHandler* is usually used by an ISA nexus driver in order to apply a newly downloaded driver to its child devices which are not yet serviced. Note that *loadHandler* is called in the DKI thread context.

On success, `IsaBusOps.open` returns `K_OK` and a valid identifier is returned in the *devId* argument. This identifier must be used to call other services defined in the `IsaBusOps` structure. Some of these services also return an "Ops" structure defining new services on a new object instance, like an I/O or a memory region. Others just perform a simple service for the device driver, without giving access to a subpart of the API.

On failure, `IsaBusOps.open` returns an error code as follows:

`K_EINVAL` — The device node is invalid, that is, the device node is not a child of the bus node.

`K_EBUSY` — A connection is already open for the given device node.

`K_ENOMEM` — The system is out of memory.

To release the connection with the ISA bus, the driver must call the `IsaBusOps.close` service:

```
    void
(*close) (IsaDevId devId);
```

After being disconnected from the ISA bus, the device driver can no longer use any of the ISA bus API services.

Probing Connection

If the `drv_probe` routine is defined, the ISA bus driver invokes it at bus probing time. Note that the `drv_probe` routine is called in the context of the DKI thread which makes it possible to invoke the ISA bus services allowed in the DKI thread context.

The probe routine is not defined if the *drv_probe* field of `DrvRegEntry` is set to NULL.

The `drv_probe` routine is called with three arguments:

DevNode	The ISA bus node, which identifies the node associated to the parent ISA bus in the device tree.
IsaBusOps	The ISA bus operations structure, which defines the ISA bus API and its version.
IsaBusId	The ISA bus identifier, which is an opaque to pass back when calling the `IsaBusOps.open` routine to connect the device driver to the ISA bus.

The `drv_probe` invocation precedes the `drv_init` invocation. That gives an opportunity to the ISA device driver to discover a device (which can be serviced by the device driver) on the bus and create a device node for this device in the device tree. If `drv_probe` creates a node corresponding to a device residing on the bus, it should put properties specifying needed bus resources to the device node (for example, `"intr"`, `"io-regs"`). Looking at the device node properties, the ISA bus driver will allocate bus resources for the device (if needed) and/or will check resource conflicts with other devices residing on the bus.

Note that once such a device is discovered and the device node is created, it is reasonable to bind the driver component to the device. This type of driver initiated binding is done by putting the `"driver"` property to the device node. The `"driver"` property value is a NULL terminated ASCII string specifying the driver name. Note that the driver name specified in the property must match the driver name specified in the driver registry entry. Under such conditions,

the ISA bus driver will invoke the driver's `drv_init` routine for this device node as described above. Note that the `drv_init` routine is invoked by the bus driver if, and only if, the bus resources required for the device are successfully allocated / checked by the bus driver.

Despite the presence of other busses, the ISA bus does not provide a mechanism to identify devices connected on the bus. Only heuristic methods can be used, reading I / O locations, but one cannot be sure that the device identified is the one expected. Therefore it is not recommended to use the `drv_probe` routine to discover and identify ISA devices. ISA devices are, in general, built statically by the ChorusOS loader.

ISA Interrupt Handling

To connect a handler to an ISA interrupt, an ISA driver should:

- Disable the interrupt at the device level, typically by accessing the device registers
- Attach a handler to the interrupt
- Enable the interrupt at the device level
- Use services defined by the attached interrupt object
- Disable the interrupt at the device level
- Detach the interrupt handler when no longer needed

Attaching a Handler to an ISA Interrupt

An ISA device driver can connect a handler to an ISA interrupt by calling the `IsaBusOps.intr_attach` service:

```
    KnError
(*intr_attach) (IsaDevId       devId,
                IsaPropIntr*   intr,
                IsaIntrHandler intrHandler,
                void*          intrCookie,
                IsaIntrOps**   intrOps,
                IsaIntrId*     intrId);
```

The *devId* argument identifies the connection with the ISA bus driver.

The *intr* argument indicates the ISA interrupt to which the handler will be connected. This property should be retrieved by the driver using the device tree API, prior to calling `IsaBusOps.intr_attach`. The name of the property used to describe an ISA interrupt is `"intr"`, its value contains an array of `IsaPropIntr` structures shown below:

```
typedef struct IsaPropIntr {
    IsaIntr      irq;     /* intr number */
    IsaIntrType  type;    /* interrupt type (edge, level) */
    uint32_f     mask;    /* bit mask of unwanted intrs   */
```

```
} IsaPropIntr;
```

The *irq* field specifies the ISA interrupt to attach.

The *type* field specifies the ISA interrupt type and can take one of the following values:

`ISA_INTR_T_EDGE`	The interrupt is triggered by driving the line low. The pull-up resistors will then generate the rising edge. This low to high transition registers an interrupt request managed by the interrupt controller.
`ISA_INTR_T_LEVEL`	The interrupt is triggered by a high signal.

ISA devices generally use edge interrupt types, which normally prevents interrupt sharing.

The *mask* field is used for dynamic interrupt acquisition in the following way: a bit set means that the device is capable of triggering (working with) the corresponding interrupt. For example, in an ISA PnP subsystem, an ISA device driver can acquire an ISA interrupt dynamically; the *mask* field specifies unwanted interrupts, if any.

The *intrHandler* argument is a handler in the device driver which is invoked by the ISA bus driver when the corresponding interrupt occurs on the bus:

```
typedef IsaIntrStatus (*IsaIntrHandler) (void* intrCookie);
```

The *intrCookie* is passed back to this interrupt handler as an argument. This type of ISA interrupt handler must return an *IsaIntrStatus* value which indicates to the bus driver whether the interrupt was claimed by the handler, and how it was handled:

`ISA_INTR_UNCLAIMED`	Must be returned by the interrupt handler if there is no pending interrupt for the device.
`ISA_INTR_CLAIMED`	Must be returned by the interrupt handler if a pending device interrupt has been serviced by the interrupt handler and the interrupt has not been enabled (acknowledged) at ISA bus level (see section Enabling/Disabling a Serviced Interrupt).
`ISA_INTR_ACKNOWLEDGED`	Must be returned by the interrupt handler if a pending device interrupt has been serviced by the interrupt handler and the interrupt has been enabled (acknowledged) at ISA bus level

(see section Enabling / Disabling an Attached Interrupt).

On success, K_OK is returned and services defined on an attached interrupt object are returned in the *intrOps* parameter. An identifier for the attached interrupt is also returned in *intrId*. This identifier must be used as first parameter to further calls to the IsaIntrOps services.

As explained above, the ISA bus architecture does not allow an interrupt to be shared among multiple devices residing on the bus.

Masking/Unmasking an Attached interrupt

The IsaIntrOps.mask service routine masks the interrupt source specified by *intrId*:

```
    void
(*mask) (IsaIntrId intrId);
```

Note that IsaIntrOps.mask does not guarantee that all other interrupt sources are still unmasked.

The IsaIntrOps.unmask service routine unmasks the interrupt source previously masked by IsaIntrOps.mask:

```
    void
(*unmask) (IsaIntrId intrId);
```

Note that IsaIntrOps.unmask does not guarantee that the interrupt source is unmasked immediately. The real interrupt source unmasking may be deferred.

The IsaIntrOps.mask / IsaIntrOps.unmask pair may be used at either base or interrupt level. Note that the IsaIntrOps.mask / IsaIntrOps.unmask pairs must not be nested.

Enabling/Disabling a Serviced Interrupt

The IsaIntrOps.enable and IsaIntrOps.disable service routines are dedicated to interrupt handler usage only. In other words, these routines may be called only by an interrupt handler.

The IsaIntrOps.enable service routine enables (and acknowledges) the bus interrupt source specified by *intrId*:

```
    IsaIntrStatus
(*enable) (IsaIntrId intrId);
```

`IsaIntrOps.enable` returns either `ISA_INTR_ACKNOWLEDGED` or `ISA_INTR_CLAIMED`. The `ISA_INTR_ACKNOWLEDGED` return value means that the ISA bus driver has enabled (and acknowledged) interrupt at bus level. The `ISA_INTR_CLAIMED` return value means that the ISA bus driver has ignored the enable request and therefore the interrupt source is still disabled (and not acknowledged) at bus level.

Note that in cases where the `IsaIntrOps.enable` routine has been called by an interrupt handler, the handler must return the value which was returned by `IsaIntrOps.enable`. Note also, that once `IsaIntrOps.enable` is called, the driver should be able to handle an immediate re-entrance in the interrupt handler code.

The `IsaIntrOps.disable` service routine disables the interrupt source previously enabled by `IsaIntrOps.enable`:

```
    void
(*disable) (IsaIntrId intrId);
```

If `IsaIntrOps.enable` returns `ISA_INTR_ACKNOWLEDGED`, the driver must call `IsaIntrOps.disable` prior to returning from the interrupt handler.

When an interrupt occurs, the attached *IsaIntrHandler* is invoked with the interrupt source disabled at bus level. This produces the same result as calling `IsaIntrOps.disable` just prior to the handler invocation. Note that the interrupt handler must return to the bus driver in the same context as it was called, that is, with the interrupt source disabled at bus level.

On the other hand, the called interrupt handler may use the `IsaIntrOps.enable/disable` pair to allow the interrupt to be nested.

Detaching an Attached Interrupt

When a driver no longer needs to handle an interrupt, or before closing the connection to the ISA bus, it should detach the handler attached to an interrupt line.

The `IsaBusOps.intr_detach` is used to detach a handler, previously attached with `IsaBusOps.intr_attach`, and to release any resources allocated for this attachement:

```
    void
(*intr_detach) (IsaIntrId intrId);
```

The *intrId* argument identifies the attachement to release.

When the `IsaBusOps.intr_detach` function is called, the driver can no longer use the identifier *intrId*.

Accessing ISA I/O registers

To perform I/O access to registers of an ISA device, an ISA driver must:

- Map the device's registers into an I/O region
- Use services defined by the mapped I/O region to access registers
- Unmap the region when access is no longer needed

Mapping Device I/O Registers

The `IsaBusOps.io_map` service is used to map an I/O region from an ISA device to enable access to this region:

```
     KnError
(*io_map) (IsaDevId       devId,
           IsaPropIoRegs* ioRegs,
           IsaErrHandler  errHandler,
           void*          errCookie,
           IsaIoOps**     ioOps,
           IsaIoId*       ioId);
```

devId is returned by `IsaBusOps.open`.

The *ioRegs* structure defines the I/O region to map. This structure is an element of the array stored in a property of the device node. This property should be retrieved by the driver using the device tree API prior to calling `IsaBusOps.io_map`.

The name of the property used to describe device I/O regions is `"io-regs"`, its value contains an array of `IsaPropIoRegs` structures shown below:

```
typedef struct IsaPropIoRegs {
    IsaAddr    addr;     /* requested/allocated start address */
    IsaSize    size;     /* size */
} IsaPropIoRegs;
```

The *addr* field specifies the ISA start address of the register's range.

The *size* field specifies the register's range size in bytes.

The *errHandler* argument is a handler in the device driver, which is invoked by the ISA bus driver, when an ISA bus error occurs while accessing the mapped region. Please note that it is not always possible for an ISA bus driver to detect this type of error, and in this case, the *errHandler* will never be called. *errCookie* is passed back to the *errHandler* as first parameter (see section ISA Error Handling).

On success, `K_OK` is returned and appropriate I/O services are returned in the *ioOps* parameter. An identifier for the mapped I/O region is also returned in *ioId*. This identifier must be used as first parameter to further calls to the ISA I/O operations services, as defined in `IsaIoOps`.

On failure, `IsaBusOps.io_map` returns an error code as follows:

`K_ESIZE`	A size of zero was specified.
`K_EOVERLAP`	Not enough virtual address space to map I/O registers.
`K_ENOMEM`	The system is out of memory.

Performing I/O Access to a Mapped I/O Region

Once the region is successfully mapped, the driver can use the services defined by the `IsaIoOps` structure.

ISA bus provides four service routine sets to access a mapped I/O region:

- `IsaIoOps.load_8/16/32/64`
- `IsaIoOps.store_8/16/32/64`
- `IsaIoOps.read_8/16/32/64`
- `IsaIoOps.write_8/16/32/64`

There are three service routines in each set which deal with I/O registers of different widths. The `_8`, `_16`, `_32` or `_64` suffix indicates the data size of the transfer on the ISA bus.

In all service routines provided by `IsaIoOps`, the *ioId* argument identifies the mapped I/O region as returned by `IsaBusOps.io_map`.

The *offset* argument specifies the register offset, in bytes, within the mapped region.

All these routines handle byte swapping in the case of the endian being different for the ISA bus and CPU.

Load from a Register

The `IsaIoOps.load_xx` routine set returns a value loaded from a device register:

```
     uintxx_f
(*load_xx) (IsaIoId ioId, IsaSize offset);
```

The size of the returned value and of the data transfer on the ISA bus is specified by the `_xx` suffix.

Store to a Register

The `IsaIoOps.store_xx` routine set stores a given value into a device register:

```
    void
(*store_xx) (IsaIoId ioId, IsaSize offset, uintxx_f value);
```

The size of the given value and of the data transfer on the ISA bus is specified by the _xx suffix.

Multiple Read from a Register

The `IsaIoOps.read_xx` routine set loads values from a specified device register and sequentially writes them into a memory buffer:

```
    void
(*read_xx) (IsaIoId ioId, IsaSize offset, uintxx_f* buf, IsaSize count);
```

The size of each value loaded and of each data transfer on the ISA bus is specified by the _xx suffix.

The *count* argument specifies the number of read transactions to perform.

The *buf* argument specifies the address of the memory buffer. The size of this buffer must be at least (*count* * size of each data transfer).

Multiple Write to a Register

The `IsaIoOps.write_xx` routine set sequentially reads values from a memory buffer and stores them into a device register:

```
    void
(*write_xx) (IsaIoId ioId, IsaSize offset, uintxx_f* buf, IsaSize count);
```

The size of each value stored and of each data transfer on the ISA bus is specified by the _xx suffix.

The *count* argument specifies the number of write transactions to perform.

The *buf* argument specifies the address of the memory buffer. The size of this buffer must be at least (*count* * size of each data transfer).

Unmapping Device I/O Registers

When a driver no longer needs access to the device I/O register, or before closing the connection to the ISA bus, it should unmap the I/O region used for these devices.

The `IsaBusOps.io_unmap` is used to unmap an I/O region previously mapped with `IsaBusOps.io_map`, and to release any resources allocated for this mapping:

```
    void
(*io_unmap) (IsaIoId ioId);
```

The *ioId* argument identifies the region to unmap.

When the `IsaBusOps.io_unmap` function is called, the driver is no longer allowed to use any services defined for the unmapped I/O region.

Accessing ISA Memory

To perform access to the memory of an ISA device, an ISA driver must:

- Map device memory to the virtual memory space
- Access device memory through the mapped region
- Unmap the region when access is no longer needed

The mapped memory region may be accessed directly using the virtual address of the mapped memory region.

Mapping Device Memory

The `IsaBusOps.mem_map` service is used to map a memory region from an ISA device, enabling access to this region:

```
    KnError
(*mem_map) (IsaDevId       devId,
            IsaPropMemRgn* memRgn,
            IsaMemAttr     memAttr,
            IsaErrHandler  errHandler,
            void*          errCookie,
            void**         memAddr,
            IsaMemId*      memId);
```

devId is the identifier returned by `IsaBusOps.open`.

The *memRgn* structure defines the memory region to map. This structure is an element of the array stored in a property of the device node. This property should be retrieved by the driver using the device tree API prior to calling `IsaBusOps.mem_map`.

The name of the property used to describe device memory regions is `"mem-rgn"`, its value contains an array of `IsaPropMemRgn` structures shown below:

```
typedef struct IsaPropMemRgn {
    IsaAddr addr;    /* requested/allocated start address */
    IsaSize size;    /* size */
} IsaPropMemRgn;
```

The *addr* field specifies the ISA start address of the memory region.

The *size* field specifies the memory region size in bytes.

The *memAttr* argument specifies the mapping attributes. A combination of the following flags is allowed:

ISA_MEM_READABLE	The memory region is readable.
ISA_MEM_WRITABLE	The memory region is writable.
ISA_MEM_EXCUTABLE	The memory region is executable.
ISA_MEM_CACHEABLE	The memory region is cacheable.

errHandler is a handler in the device driver, which is invoked by the ISA bus driver, when an ISA bus error occurs while accessing the mapped region. *errCookie* is passed back to the errHandler as first parameter (see section ISA Error Handling).

On success, K_OK is returned and the starting virtual address of the mapped memory region is returned in the *memAddr* argument. An identifier for the mapped memory region is also returned in *memId*. This identifier must be used to unmap the memory region.

On failure, IsaBusOps.mem_map returns an error code as follows:

K_ESIZE	A size of zero was specified.
K_EINVAL	Invalid or unsupported memory mapping attributes.
K_EOVERLAP	Not enough virtual address space to map.
K_ENOMEM	The system is out of memory.

Once the region is successfully mapped, the device driver may now access the device memory by dereferencing the returned pointer. Note that in this mode, the driver must handle endianning problems itself.

Unmapping a Memory Region

When a driver no longer needs access to the device memory, or before closing the connection to the ISA bus, it should unmap the memory region.

The IsaBusOps.mem_unmap is used to unmap a memory region, previously mapped with IsaBusOps.mem_map, and to release any resources allocated for this mapping:

```
    void
(*mem_unmap) (IsaMemId memId);
```

The *memId* argument identifies the memory region to unmap.

Once the memory region is unmapped, the driver can no longer access the memory region.

Direct Memory Access

The ISA bus driver provides an API for a standard DMA chip controller (Intel8237 or compatible).

To perform Direct Memory Access to the system memory, an ISA device driver must:

- Attach a DMA channel
- Allocate a DMA region on the bus
- Use services defined in `IsaDmaOps` to retrieve the region's virtual and ISA addresses
- Program the DMA engine to perform DMA transfers
- Release the DMA region when no longer needed
- Release the DMA channel

Attaching a DMA Channel

The `IsaBusOps.dma_attach` service attaches an ISA DMA channel. The specified DMA channel must be in a free state, as DMA channels are not shared.

```
    KnError
(*dma_attach) (IsaDevId     devId,
               IsaPropDma*  propDma,
               IsaDmaOps**  dmaOps,
               IsaDmaId*    dmaId);
```

The *devId* argument is the identifier returned by `IsaBusOps.open`.

The *propDma* structure defines the channel to attach.

This structure is an element of the array stored in a property of the device node. This property should be retrieved by the driver using the device tree API prior to calling `IsaBusOps.dma_attach`.

The name of the property used to describe DMA channels is `"dma"`, and its value contains an array of `IsaPropDma` structures shown below:

```
typedef struct IsaPropDma {
    IsaDmaChan  chan;       /* DMA channel requested/allocated */
    IsaSize     max_size;   /* Max size of transfer requested/allowed */
} IsaPropDma;
```

The *chan* is the channel number.

The *max_size* contains the maximum size of the transfer in bytes, and may vary depending on the DMA channel.

On success, `K_OK` is returned and appropriate services are returned in the *dmaOps* parameter. An identifier for the DMA channel is also returned in

dmaId. This identifier must be used as first parameter to further calls to the `IsaDmaOps` services.

On failure, `IsaBusOps.dma_attach` returns an error code as follows:

`K_EBUSY`	The requested channel is already in use.
`K_ENOMEM`	The system is out of memory.

Allocating a DMA Region

The `IsaDmaOps.mem_alloc` service allocates a system memory region of the specified size and contiguously maps it to the supervisor address space:

```
    KnError
(*mem_alloc) (IsaDmaId      dmaId,
              IsaSize       size,
              IsaDmaAttr    dmaAttr,
              IsaMemAttr    memAttr,
              IsaErrHandler errHandler,
              void*         errCookie,
              IsaDmaMemId*  dmaMemId);
```

The *dmaId* argument is the identifier returned by `IsaBusOps.dma_attach`.

The *size* argument is the requested size in bytes to allocate for the DMA memory region.

The *dmaAttr* argument specifies the DMA transfer type. A combination of the following flags is allowed:

`ISA_DMA_READABLE`	Region is used for DMA read transfer.
`ISA_DMA_WRITABLE`	Region is used for DMA write transfer.
`ISA_DMA_SYNC`	Allocate a synchronous DMA region for which no synchronization is needed between access from DMA engine and CPU. This attribute may not be supported on certain platforms. In this case, `IsaDmaOps.mem_alloc` returns `K_EINVAL`.

The *memAttr* argument specifies the mapping attributes (see section Mapping Device Memory).

errHandler is a handler in the device driver, which is invoked by the ISA bus driver, when an ISA bus error occurs while accessing the DMA region. *errCookie* is passed back to the errHandler as first parameter (see section ISA Error Handling).

On success, K_OK is returned and an identifier for the DMA region is returned in *dmaMemId*. This identifier must be used as first parameter to certain services in IsaDmaOps.

On failure, IsaDmaOps.mem_alloc returns an error code as follows:

K_ESIZE	A size of zero was specified.
K_EINVAL	Invalid or unsupported memory mapping attributes.
K_EOVERLAP	Not enough virtual address space to map.
K_ENOMEM	The system is out of memory.

Getting the Virtual Address of a DMA Region

The IsaDmaOps.virt_addr routine returns the virtual starting address of a DMA region, given the identifier of the region (*dmaMemId*):

```
    void*
(*virt_addr) (IsaDmaMemId dmaMemId);
```

The driver uses this address to access the DMA region using CPU instructions. Note that the driver should synchronize the region as appropriate, depending on the attributes used when allocating the region (see section Synchronizing the DMA Region).

Getting the ISA Address of a DMA Region

The IsaDmaOps.phys_addr routine returns the ISA starting address of a DMA region, given the identifier of the region (*dmaMemId*):

```
    IsaAddr
(*phys_addr) (IsaDmaMemId dmaMemId);
```

The driver uses this address to program the DMA engine. Note that the driver should synchronize the region as appropriate, depending on the attributes used when allocating the region (see section Synchronizing the DMA Region).

Starting a DMA Transfer

Once a DMA channel has been attached, and a DMA region allocated, the device driver can start DMA transfer, using the IsaDmaOps.read or IsaDmaOps.write services.

The IsaDmaOps.read is used to transfer from an ISA memory region to the peripheral device, while IsaDmaOps.write transfers data from the device to the ISA memory region.

```
    KnError
(*read) (IsaDmaId      dmaId,
         IsaDmaMemId   dmaMemId,
         IsaSize       offset,
         IsaSize       size,
         IsaDmaMode    mode);

    KnError
(*write) (IsaDmaId      dmaId,
          IsaDmaMemId   dmaMemId,
          IsaSize       offset,
          IsaSize       size,
          IsaDmaMode    mode);
```

For both routines:

The *dmaId* argument identifies the DMA channel, as returned by `IsaBusOps.dma_attach`.

The *dmaMemId* argument identifies the DMA region, as returned by `IsaDmaOps.mem_alloc`.

offset is the offset in bytes from the beginning of the DMA memory region.

size is the size in bytes of the transfer.

mode is a combination of DMA engine specific operation modes, as follows:

`ISA_DMAMODE_SINGLE`	A single transfer is requested. The DMA engine relinquishes the bus between each byte or word transferred.
`ISA_DMAMODE_BLOCK`	This mode causes the DMA engine to transfer a buffer of data, keeping bus control until the end of the transfer.
`ISA_DMAMODE_DEMAND`	This mode is identical to `ISA_DMAMODE_BLOCK`, but the DMA engine is programmed to relinquish the bus between bursts of transfer.
`ISA_DMAMODE_AUTOINIT`	This mode causes the DMA engine to initialize itself at the end of the current transfer, with the same parameter (*offset* and *size*) values. Afterwards, the channel concerned is ready for a new DMA transfer.

On success `K_OK` is returned. On failure, `IsaDmaOps.read` and `IsaDmaOps.write` return an error code as follows:

`K_ESIZE`	A size of zero was specified.

K_EINVAL	Invalid parameter detected. The alignment and size for parameters *offset* and *size* should be specified according to the hardware and DMA channel used.
K_ENOTIMP	Invalid mode (or not implemented) specified in *mode* parameter.
K_ENOMEM	The system is out of memory.

Aborting a DMA Transfer

Once a DMA transfer has been started, it is possible to abort it, using the IsaDmaOps.abort service.

```
    KnError
(*abort) (IsaDmaId    dmaId);
```

On success, K_OK is returned. Otherwise IsaDmaOps.abort returns the K_ENOTIMP error code.

Synchronizing the DMA Region

If the DMA region was not allocated using the ISA_DMA_SYNC attribute, the device driver should synchronize accesses to the same system memory correctly (the DMA region) from the CPU and the DMA engine. Depending on the platform, these routines handle possible problems of cache coherency, DMA engine buffers, and others.

IsaDmaOps.read_sync is a barrier between CPU writes and DMA reads from a given DMA sub-region:

```
    void
(*read_sync) (IsaDmaId     dmaId,
              IsaDmaMemId  dmaMemId,
              IsaSize      offset,
              IsaSize      size);
```

IsaDmaOps.write_sync is a barrier between DMA writes and CPU reads from a given DMA sub-region:

```
    void
(*write_sync) (IsaDmaId     dmaId,
               IsaDmaMemId  dmaMemId,
               IsaSize      offset,
               IsaSize      size);
```

For both routines:

The *dmaId* argument identifies the DMA id, as returned by `IsaBusOps.dma_attach`.

The *dmaMemId* argument identifies the DMA region, as returned by `IsaDmaOps.mem_alloc`.

offset is the offset in bytes from the beginning of the DMA memory region.

offset and *size* both define the sub-region to synchronize.

Releasing a DMA Region

When a driver no longer needs an allocated DMA region, or before closing the connection to the ISA bus, it should release the DMA region.

The `IsaBusOps.dma_free` is used to release a DMA region, previously allocated with `IsaBusOps.dma_alloc`:

```
    void
(*dma_free) (IsaDmaId dmaId);
```

The *dmaId* argument identifies the DMA region to free.

Once the DMA region is released, the driver can no longer use the DMA region.

Releasing a DMA Channel

The `IsaBusOps.dma_detach` service detaches a previously attached DMA channel.

```
    void
(*dma_detach) (IsaDmaId dmaId);
```

Once the DMA channel is released, the driver can no longer use any DMA services from `IsaDmaOps`, and the identifier *dmaId*.

ISA Error Handling

When mapping ISA bus space, an ISA driver provides an error handler, which is invoked by the ISA bus driver when a bus error occurs.

Error handlers are used for the following services:

- `IsaBusOps.io_map`
- `IsaBusOps.mem_map`
- `IsaDmaOps.mem_map`

When a programmed or direct memory access is aborted because of a bus error, the ISA bus driver invokes the error handler that is connected for the ISA address and space concerned.

Note that depending on hardware, it is not always possible for the ISA bus driver to detect these types of errors. In this case, the ISA bus driver will not invoke device driver error handlers.

An `IsaErrHandler` is defined as follow:

```
typedef void (*IsaErrHandler) (void* errCookie, IsaBusError* err);
```

The *errCookie* is an opaque given by the caller and passed back when the handler is called.

The *err* argument points to the `IsaBusError` structure describing the error as follows:

```
typedef struct IsaBusError {
    IsaErrorCode  code;       /* error type */
    IsaSize       offset;     /* faulted address offset within region */
} IsaBusError;
```

The *code* field indicates the type of error that occurred.

The *offset* field indicates the offset within the associated region at which the error occurred.

The error code can be one of the following::

`ISA_ERR_UNKNOWN`	Unknown error, that is, the ISA bus driver is unable to determine the reason for the error.
`ISA_ERR_INVALID_SIZE`	Invalid (that is, unsupported) access granularity has been used.
`ISA_ERR_PARITY`	A parity error has been detected on the ISA bus.

As a bus error may be reported in the context of an exception or an interrupt, the implementation of the error handler must be restricted to the interrupt level API.

PROPERTIES

ISA Specific Node Properties

The ISA Specific Properties table lists the ISA specific node properties. The *alias* column specifies the alias name which should be used by an ISA bus or device driver to reference the property name. The *name* column specifies the property name ASCII string. The *value* column specifies the type of property value. The *bus* column specifies properties specific to the ISA bus node. The *dev* column specifies properties specific to the ISA device node. The *bus* and *dev* fields can take the following values:

`m`	Flags' mandatory properties
`o`	Flags' optional properties

-	Flags' properties which are not applied to a given node type

Alias	Name	Value	Bus	Dev
ISA_PROP_INTR	"intr"	IsaPropIntr[]	-	o
ISA_PROP_IO_REGS	"io-regs"	IsaPropIoRegs[]	-	o
ISA_PROP_MEM_RGN	"mem-rgn"	IsaPropMemRgn[]	-	o
ISA_PROP_DMA_BURST	"dma-burst	IsaPropDmaBurst	o	-
ISA_PROP_DMA_MIN_SIZE	"dma-min-size"	IsaPropDmaMinSize	o	-
ISA_PROP_BYTE_ORDER	"byte-order"	IsaPropByteOrder	o	-
ISA_PROP_CLOCK_FREQ	"clock-freq"	IsaPropClockFreq	o	-

When the value of the property is an array of (for example, `IsaPropIoRegs`), the size of this array divided by the size of its element type (for example, `sizeof(IsaPropIoRegs)`), defines the number of elements in the array. An ISA device driver may then iterate through the array in order to perform an action for each element (for example, to map several I/O register ranges). The size of the array is the size of the property value returned by `dtreePropValue`.

Dynamic Resource Allocation

The ISA bus driver may support dynamic allocation on the ISA bus resources. Typically, this feature must be supported by an ISA PnP bus driver. In cases where dynamic resource allocation is supported, an ISA device driver may use the `IsaBusOps.resource_alloc` service routine in order to allocate a bus resource at run time:

```
    KnError
(*resource_alloc) (IsaDevId devId, DevProperty prop);
```

devId is returned by `IsaBusOps.open`.

prop specifies the bus resource being allocated.

The `IsaBusOps.resource_alloc` service routine allocates a given bus resource and, if the allocation request is satisfied, updates the device node properties in order to add the newly allocated bus resource.

On success, `IsaBusOps.resource_alloc` returns `K_OK`.

On failure, one of the following error codes is returned:

K_ENOTIMP	Dynamic resource allocation is not supported by the bus driver.
K_EUNKNOWN	The property name is unknown.
K_EINVAL	The property value is invalid.
K_EFAIL	The bus resource is not available.
K_ENOMEM	The system is out of memory.

When a dynamically allocated bus resource is no longer used, an ISA device driver may release it by calling the IsaBusOps.resource_free service routine:

```
    void
(*resource_free) (IsaDevId devId, DevProperty prop);
```

devId is returned by IsaBusOps.open.

prop specifies the bus resource being released.

The IsaBusOps.resource_free service routine releases a given bus resource and updates the device node properties in order to remove the bus resource being released.

The following bus resource properties may be dynamically allocated and released:

- ISA_PROP_INTR
- ISA_PROP_IO_REGS
- ISA_PROP_MEM_RGN

Note that the IsaBusOps.resource_alloc and IsaBusOps.resource_free routines should not be used by a simple device driver. This type of driver should assume that all resources needed are already allocated and specified as properties in the device node by the bus driver. The driver should only find this type of property and call an appropriate service routine passing a pointer to the property value.

However, it is not always possible to determine all needed resources prior to device initialization. A typical case is a bus-to-bus bridge driver which can discover devices residing on the secondary bus only when the bus bridge hardware has already been initialized. In this case, when drv_init is called, the bus-to-bus bridge node would contain only resources needed for the bus-to-bus bridge device itself (for example, internal bus-to-bus bridge registers). Once

devices residing on the secondary bus are discovered, the bus-to-bus bridge driver would request additional primary bus resources in order to satisfy the resource requirements for these devices. Another example is a bus which supports hot-pluggable devices. On this type of bus, the primary bus resources allocated by the hot-pluggable bus driver depend on devices currently plugged into the secondary bus. The resource requirements usually change when a hot-plug insertion / removal occurs.

Note that dynamic resource allocation might be also used by a device driver which implements a lazy resource allocation. In this type of driver, the bus resource allocation might be performed at open time using `IsaBusOps.resource_alloc`. The dynamically allocated resources might then be released at close time using `IsaBusOps.resource_free`.

ALLOWED CALLING CONTEXTS

ISA Bus Interface 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
IsaBusOps.open	-	+	-	+
IsaBusOps.close	-	+	-	+
IsaBusOps.intr_attach	-	+	-	+
IsaBusOps.intr_detach	-	+	-	+
IsaBusOps.io_map	-	+	-	+
IsaBusOps.io_unmap	-	+	-	+
IsaBusOps.mem_map	-	+	-	+
IsaBusOps.mem_unmap	-	+	-	+
IsaBusOps.dma_attach	-	+	-	+
IsaBusOps.dma_attach	-	+	-	+
IsaBusOps.dma_free	-	+	-	+
IsaBusOps.resource_alloc	-	+	-	+
IsaBusOps.resource_free	-	+	-	+
IsaIoOps.load_xx	+	+	+	-
IsaIoOps.store_xx	+	+	+	-
IsaIoOps.read_xx	+	+	+	-

Services	Base level	DKI thread	Interrupt	Blocking
IsaIoOps.write_xx	+	+	+	-
IsaDmaOps.mem_alloc	-	+	-	+
IsaDmaOps.mem_free	-	+	-	+
IsaDmaOps.phys_addr	+	+	+	-
IsaDmaOps.virt_addr	+	+	+	-
IsaDmaOps.read	+	+	+	-
IsaDmaOps.write	+	+	+	-
IsaDmaOps.read_sync	+	+	+	-
IsaDmaOps.write_sync	+	+	+	-
IsaDmaOps.abort	+	+	+	-
IsaDmaOps.mask	+	+	+	-
IsaDmaOps.unmask	+	+	+	-
IsaDmaOps.enable	-	-	+	-
IsaDmaOps.disable	-	-	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`dtreeNodeRoot`(9DKI), `svDriverRegister`(9DKI), `svMemAlloc`(9DKI), `svPhysAlloc`(9DKI), `svPhysMap`(9DKI), `svDkiThreadCall`(9DKI), `svTimeoutSet`(9DKI), `usecBusyWait`(9DKI), `DISABLE_PREEMPT`(9DKI)

NAME

netFrame – Generic Representation of Network Frames

SYNOPSIS

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

FEATURES

DDI

DESCRIPTION

Provides a generic representation of network frames.

EXTENDED DESCRIPTION

As the implementation of network protocols must be independent of the network devices through which they communicate, a common representation of output and input network frames must be used between physical network device drivers and all upper layers.

The contents of a network frame are scattered into a set of contiguous memory buffers which is represented by a list of `NetBuf` structures.

Network Memory Buffer

```
typedef struct NetBuf {
    struct NetBuf* next;    /* next buffer in the list or NULL */
    uint32_f       bufSize; /* size of memory buffer */
    char*          bufAddr; /* starting address of memory buffer */
} NetBuf;
```

The *next* field points to the next element in the list of `NetBuf` structures, if any. It is set to `zero` in the last element of the list.

The *bufAddr* field points to the beginning of the data area of the memory buffer.

The *bufSize* is the size of the memory buffer. All the fields of the `NetBuf` structures associated to a network frame are read-only for the consumer to which the network frame is provided. This constraint is respected by the following library functions which provide basic copy services of network memory buffers:

```
extern_C void copyToNetBuf   (char* src, NetBuf* dstBufs, uint32_f size);
extern_C void copyFromNetBuf (NetBuf* srcBufs, char* dst, uint32_f size);
extern_C void netBufCopy     (NetBuf* srcBufs, NetBuf* dstBufs, uint32_f size);
```

The `copyToNetBuf` function copies *size* bytes from the contiguous memory area starting at address *src* to the *dstBufs* list of network memory buffers. The `copyFromNetBuf` function copies *size* bytes from the *srcBufs* list of network memory buffers to a contiguous memory area starting at address *dst*. The `netBufCopy` function copies *size* bytes from the *srcBufs* list of network memory buffers to the *dstBufs* list of network memory buffers.

Network Frame

```
typedef struct NetFrame* NetFramePtr;

typedef void (*NetFrameFree)(NetFramePtr);

typedef struct NetFrame {
    struct NetFrame* next;      /* to build list of NetFrames */
    uint32_f         frameSize; /* total length of frame */
```

```
    NetBuf*          bufList;   /* list of buffers holding the data */
    NetFrameFree     freeFrame; /* the free function of the frame */
} NetFrame;

#define NET_FRAME_FREE(netFrame) (*netFrame->freeFrame)(netFrame)
```

The *next* field is for recording network frames as lists. The *frameSize* field specifies the total size in bytes of the frame. The *bufList* field points to a list of `NetBuf` structures which represent the memory buffers holding the frame data.

The *freeFrame* field points to the function which must be invoked to free the network frame. This function takes one argument, the address of the `NetFrame` structure itself. The *freeFrame* function of a network frame must be invoked by the consumer to return the network frame to its producer.

The *NET_FRAME_FREE* macro is provided as a short-cut for invoking the *freeFrame* function of a network frame. The *next* field of the `NetFrame` structure can be freely used by the consumer of a network frame until it frees it. All the other fields of the `NetFrame` structure are read-only for the consumer to which the network frame is provided.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

nvram – NVRAM Device Driver Interface

SYNOPSIS

```
#include <ddi/nvram/nvram.h>
```

FEATURES

DDI

DESCRIPTION

The NVRAM device driver provides a basic interface by abstracting the access methods to the underlying NVRAM hardware. Basically, the driver implements routines that allow the device driver client to read / write from / to the NVRAM space.

It is assumend that there is an intermediate layer (logical NVRAM driver) between an application and the device driver. Such a logical NVRAM driver would typically synchronize concurrent accesses to the NVRAM space. It would also protect some NVRAM areas against application accesses. Such protected NVRAM areas may be reserved for firmware usage or may map another device (e.g. a real-time-clock device).

Service Routines

The character string “nvram” names the timer device class. The driver client use the device class name to get the driver service routines from the device registry as described in `svDeviceLookup`(9DKI).

The NVRAM device driver is a mono-client driver. The driver registy prevents multiple look-ups to be done on the same NVRAM device driver instance. Typically, only a logical NVRAM driver interfaces directly to the NVRAM device driver. The logical NVRAM driver, in turn, provides an application interface (which might be multi-client oriented).

The `NvramDevOps` structure specifies the interface of the NVRAM device driver service routines.

```
typedef struct NvramDevOps {

     NvramVersion version;

         KnError
     (*open) (NvramId id);

         KnError
     (*read) (NvramId    id,
              NvramSize  offset,
              void*      buffer,
              NvramSize  size);

         KnError
     (*write) (NvramId   id,
               NvramSize offset,
               void*     buffer,
               NvramSize size);

         void
```

```
    (*close) (NvramId id);
} NvramDevOps;
```

version — The *version* field specifies the highest version number of the NVRAM device driver interface supported by the driver. Currently, only one version is available and therefore the *version* field must be set to zero (NVRAM_VERSION_INITIAL). In the future, the NVRAM device driver interface may be extended, but backward compatibility is guaranteed; extra methods may be added to the NvramDevOps structure but the existing methods will remain unchanged.

open — The open routine must be the first call to the driver. The open routine makes the NVRAM device operational and enables subsequent invocations of the read, write and close routines. The *id* argument specifies a given NVRAM device. It is obtained from the device registry entry *dev_id* field. open returns K_OK on success. The K_EFAIL error code is returned if the driver is not able to put the device in an operation state.

read — The read routine (synchronously) reads data from the NVRAM device to a memory buffer. The *id* argument specifies a given NVRAM device and is obtained from the device registry entry *dev_id* field. The *offset* argument specifies the start offset in bytes from the beginning of the NVRAM space. The *buffer* argument specifies the start address of a memory buffer. The *size* argument specifies the number of bytes to read. Note that read does not check correctness of input arguments. For example, if *offset* is beyond the NVRAM space, the driver behavior is unpredictable. The read routine returns K_OK on success. The K_EFAIL error code is returned if the driver encounters an (hardware) error condition during the read.

write — The write routine (synchronously) writes data from a memory buffer to the NVRAM device. The *id* argument specifies a given NVRAM device. It

is obtained from the device registry entry *dev_id* field. The *offset* argument specifies the start offset in bytes from the beginning of the NVRAM space. The *buffer* argument specifies the start address of a memory buffer. The *size* argument specifies the number of bytes to write. Note that `write` does not check correctness of input arguments. For example, if *offset* is beyond the NVRAM space, the driver behavior is unpredictable. The `write` routine returns `K_OK` on success. The `K_EFAIL` error code is returned if the driver encounters an (hardware) error condition during the write.

`close` — The `close` routine must be the last call to the driver because it makes the NVRAM device non-operational. Once the device is closed, the only routine allowed to call is `open`. The *id* argument specifies a given NVRAM device. It is obtained from the device registry entry *dev_id* field.

Device Node Properties

Properties attached to the NVRAM device node give the device driver client some (physical) characteristics of the NVRAM device. The device driver client must take into account such characteristics in order to properly perform operations on the NVRAM device.

The 'nvram-layout' (alias NVRAM_PROP_LAYOUT) specifies a physical layout of the NVRAM device space. The propery value is an array of `NvramPropChunk's`.

```
typedef struct NvramPropChunk {
        NvramSize size;
        uint32_f  attr;
    } NvramPropChunk;
```

The *size* field specifies the chunk size in bytes. Note that the `nvram-layout` property array covers the entire NVRAM space. So, the start offset of a given chunk is calculated as the sum of the *size* fields of all previous chunks.

The *attr* field specifies the chunk attributes. A combintion of the following flags is allowed:

NVRAM_READABLE — The chunk data can be read via the `read` routine.

NVRAM_WRITABLE — The chunk data can be updated via the `write` routine.

Basically, the *attr* field specifies access permissions to a given chunk of the NVRAM space and may take the following typical values:

0	The chunk should not be accessed by a device driver client. Such attributes are typically used if the chunk maps another device (a real-time-clock, for example).
NVRAM_READABLE	The chunk contents should not be altered by a device driver client. Such attributes are typically used if the chunk is used by firmware.
NVRAM_WRITABLE	A driver client can write to the nvram chunk.

Note that the NVRAM device driver does not check the access permissions when a read or write is issued. It is up to the driver client to respect the physical NVRAM layout specified by the 'nvram-layout' property.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDeviceRegister`(9DKI), `svDriverRegister`(9DKI),

NAME

pci – PCI Bus Driver Interface

SYNOPSIS

```
#include <ddi/pci/pci.h>
```

FEATURES

DDI

DESCRIPTION

Provides an API for the development of PCI bus device drivers.

EXTENDED DESCRIPTION

The PCI bus driver offers an API for PCI device drivers' development. This PCI bus API is an abstraction of the low-level PCI bus services and covers the following PCI functional modules:

- PCI interrupts management
- Access to the PCI configuration space
- Access to the PIO and memory-mapped device registers
- Access to the PCI devices' memory
- DMA management

Connect a Device Driver to the PCI Bus

First of all, a PCI device driver must register itself in the kernel driver registry.

This should be done from its main routine using `svDriverRegister`.

The driver must set the bus class to `"pci"` in its registry entry, and specify the lowest version number of the PCI bus interface required to run correctly.

This registration allows the PCI bus driver to call back the PCI device driver's `drv_probe`, `drv_bind` and `drv_init` routines at bus probing, binding and initialization phases, respectively.

Within the `drv_probe` or `drv_init` routine, the PCI device driver may establish a connection with the parent PCI bus driver as described below.

Once this connection is established, the PCI device driver may use services provided by the PCI bus driver. Note that a connection to the bus driver can be established only within the driver's `drv_probe` or `drv_init` routine.

Initialization Connection

If the `drv_init` routine is defined, the PCI bus driver invokes it at bus initialization time. Note that the `drv_init` routine is called in the DKI thread context (refer to the table "Allowed Calling Context" to check the list of QUICC bus services allowed in that context). The `drv_init` routine is defined if the `drv_init` field of the driver registry entry is set to a non-zero value.

The `drv_init` routine is called with three arguments:

DevNode	The PCI device driver's own node, which identifies the node associated to the device in the device tree.

PciBusOps	The PCI bus operations structure, which defines the PCI bus API and its version.
PciId	The PCI bus identifier, which is an opaque to pass back when calling the `PciBusOps.open` operation to connect the device driver to the PCI bus.

From this point, `PciBusOps.open` must be the first call issued by the PCI device driver to the PCI bus driver:

```
    KnError
(*open) (PciId           busId,
         DevNode         devNode,
         PciEventHandler eventHandler,
         PciLoadHandler  loadHandler,
         void*           cookie,
         PciDevId*       devId);
```

This establishes a connection to the bus, identified by the *devId* value returned in the last argument. This *devId* identifier is an argument for many other services defined in the `PciBusOps` structure.

busId and *devNode* are given by the PCI bus driver as parameters to the driver's `drv_init` routine. *eventHandler* is a handler in the device driver which is invoked by the PCI bus driver, when a PCI bus event occurs. Note that *eventHandler* is optional and must be set to NULL when it is not implemented by the device driver. The *cookie* argument is passed back to this handler as first argument. This PCI bus event handler takes two additional parameters, an event type specific argument (which is always `NULL` for the PCI bus events), and the bus event type which is one of the following:

PCI_SYS_SHUTDOWN
: Notifies a device driver that the system is going to be shut down. The device driver should reset the device hardware and return from the event handler. Note that the driver must neither notify clients nor release used resources.

PCI_DEV_SHUTDOWN
: Notifies a device driver that the device is going to be shut down. The device driver should notify driver clients (via `svDeviceShutDown`) that the device is going to be shut down and should then return from the event handler. Once the device entry is released by the last driver client, the device registry module invokes a driver call-back handler. Within this handler, the device driver should reset the device hardware, release all used resources and close the bus connection invoking `PciBusOps.close`. Note that the `PCI_DEV_SHUTDOWN` event may be used by a bus driver in order to

confiscate (or to re-allocate) bus resources. In this case, the PCI bus driver will invoke the driver's `drv_init` routine again once the bus resources are re-allocated for the device.

`PCI_DEV_REMOVAL`
Notifies a device driver that the device has been removed from the bus and therefore the device driver instance has to be shut down. The actions taken by the driver are similar to the `PCI_DEV_SHUTDOWN` case except that the device hardware must not be accessed by the driver.

PCI_SYS_ERROR
Notifies the device driver that a system error has been detected on the bus. Actions taken by the driver are driver implementation specific.

PCI_VGA_PALETTE_SNOOP_ENABLE
Notifies a device driver that `PciBusOps.vga_palette_snoop_enable` has been issued to the PCI bus driver and the `PCI_CMD_PALETTE_SNOOP` bit within the bus command register is already set. Basically, this type of event means that there is another (downstream) device on the PCI system that is ready to claim transactions to the VGA palette registers. If a given device does not implement VGA palette snooping or the `PCI_CMD_PALETTE_SNOOP` bit is already set in the device command register, the device driver must ignore this type of event and simply return `K_ENOTIMP`. Otherwise, the device driver must set this bit (within the device command register) and return `K_OK`.

PCI_VGA_PALETTE_SNOOP_DISABLE
Notifies the device driver that `PciBusOps.vga_palette_snoop_disable` has been issued to the PCI bus driver and the `PCI_CMD_PALETTE_SNOOP` bit within the bus command register is already set. Basically, this type of event means that a (downstream) device on the PCI system which previously claimed transactions to VGA palette registers is going to be shut down and therefore the PCI system is looking for another device to claim these transactions in the future. If a given device does not implement the VGA palette snooping or the `PCI_CMD_PALETTE_SNOOP` bit is already cleared in the device command register, the device driver must ignore this type of event and simply return `K_ENOTIMP`. Otherwise, the device driver must clear this bit (within the device command register) and return `K_OK`.

As the event handler may be executed in the context of an interrupt, its implementation must be restricted to the API allowed at interrupt level. *loadHandler* is a handler in the device driver which is invoked by the PCI bus driver, when a new driver appears in the system (as in when a new driver is

downloaded at run time). Note that *loadHandler* is optional and must be set to NULL when it is not implemented by the device driver. The *cookie* argument is passed back to this handler as unique argument. Typically, a leaf PCI device driver is not affected by this type of event.

`loadHandler` is usually used by a PCI nexus driver (such as a PCI/ISA bus bridge) to apply a newly downloaded driver to its child devices which are not yet serviced. Note that `loadHandler` is called in the DKI thread context.

On success, `PciBusOps.open` returns `K_OK` and a valid identifier is returned in the *devId* argument. This identifier must be used to call other services defined in the `PciBusOps` structure. Some of these services also return an `"Ops"` structure defining new services on a new object instance, for example an I/O or a memory region. Others just perform a simple service for the device driver, without giving access to a subpart of the API.

On failure, `PciBusOps.open` returns an error code as follows:

K_EINVAL	The device node is invalid, for example, the device node is not a child of the bus node.
K_EBUSY	A connection is already open for the given device node.
K_ENOMEM	The system is out of memory.

To release the connection with the PCI bus, the driver must call the `PciBusOps.close` service.

```
    void
(*close) (PciDevId devId);
```

After being disconnected from the PCI bus, the device driver can no longer use any of the PCI bus API services.

Probe Connection

If the `drv_probe` routine is defined (the probe routine is defined if the *drv_probe* field of `DrvRegEntry` is set to a non-zero value) the PCI bus driver invokes it at bus probing time. Note that the `drv_probe` routine is called in the DKI thread context (refer to the table "Allowed Calling Context" to check the list of QUICC bus services allowed in that context).

The `drv_probe` routine is called with three arguments:

DevNode	The PCI bus node, which identifies the node associated to the parent PCI bus in the device tree.

PciBusOps	The PCI bus operations structure, which defines the PCI bus API and its version.
PciId	The PCI bus identifier, which is an opaque to pass back when calling the `PciBusOps`open routine to connect the device driver to the PCI bus.

The `drv_probe` routine is called first by the bus driver. This allows the PCI device driver to discover a device (which can be serviced by the device driver) on the bus and create a device node for this device in the device tree. If `drv_probe` creates a node corresponding to a device residing on the bus, it should put properties specifying needed bus resources on the device node (for example, `"intr"`, `"io-regs"`).

After checking the device node properties, the PCI bus driver will allocate bus resources for the device (if needed) and / or will check resource conflicts with other devices residing on the bus.

The `drv_bind` routine is called by the bus driver when the probing phase is complete. It allows the device driver to perform a driver to device binding. Typically, a PCI device driver examines the vendor and device identifier properties in order to check whether a given device can be serviced by the driver. If the check is positive, the driver binds itself to the device. A driver initiated binding is performed by putting the `"driver"` property into the device node. The `"driver"` property value is a `NULL`terminated ASCII string specifying the driver name.

The driver name specified in the property must match the driver name specified in the driver registry entry. Under these conditions, the PCI bus driver will invoke the driver's `drv_init` routine for this device node as described above.

The `drv_init` routine is invoked by the bus driver if, and only if, the bus resources required for the device are successfully allocated / checked by the bus driver.

The `drv_bind` routine is called in the context of a DKI thread. It is therefore only possible to invoke PCI bus services that are allowed in the DKI thread context.

Devices residing on the PCI bus are discovered by accessing the PCI configuration space and analyzing the PCI header structure. Note that the PCI bus driver provides services to access the PCI configuration space, but they may be used only when a connection is established between the PCI bus driver and the PCI device driver (see section Accessing PCI configuration space).

In order to establish this type of connection, the device driver has to create a device node and attach it to the bus node. This device node may be temporary, but it is mandatory as an argument to the `PciBusOps.open` routine. When

the probing process is finished and the connection is closed, depending on the probing results, the device driver should either delete the temporary device node or change it into a real device node. The latter case means that a device which may be serviced by the driver will be found on the bus.

The device driver should avoid creating redundant device nodes. In particular, when the device driver discovers a device through the PCI configuration space and wants to create a device node corresponding to this device, it must ensure that there is no existing device node (among the bus child nodes) which represents the same device.

If a connection to the PCI bus is established by the `drv_probe` routine, it should be closed (via `PciBusOps.close`) before returning from the routine.

To summarize, a basic scenario of the PCI device driver's `drv_probe` routine may be considered as follows:

- Create a temporary node and attach it to the bus node.
- Open a connection to the PCI bus driver (PciBusOps.open).
- Read the PCI configuration headers looking for the given device and vendor ID.
- Once the header matches the given device and vendor ID:
- Go through the bus child nodes looking for a node corresponding to the configuration header. The `"bus-num"` property (attached to the bus node) and the `"dev-num"`, `"func-num"` properties (attached to the device node) specify the configuration header address within the PCI configuration space.
- If the node corresponding to the configuration header does not exist, create a new device node and attach it to the bus node (or use the temporary node), put the mandatory PCI properties on the node and (optionally) add extra properties.
- Detach and delete the temporary node (if it was not changed into a real device node).
- Close the connection to the PCI bus driver.
- Return to the PCI bus driver.

Access the PCI I/O Registers

To perform I/O access to registers of a PCI device, a PCI driver must:

- Map the device's registers into an I/O region
- Use services defined by the mapped I/O region to access registers
- Unmap the region when access is no longer needed

Map Device I/O Registers

The PciBusOps.io_map service is used to map an I/O region from a PCI device to enable access to this region:

```
    KnError
(*io_map) (PciDevId       devId,
           PciPropIoRegs* ioRegs,
           PciErrHandler  errHandler,
           void*          errCookie,
           PciIoOps**     ioOps,
           PciIoId*       ioId);
```

devId is returned by PciBusOps.open. The *ioRegs* structure defines the I/O region to map.

This structure is an element of the array stored in a property of the device node. This property should be retrieved by the driver using the device tree API dtreePropFind, dtreePropLength, dtreePropValue, prior to calling PciBusOps.io_map.

The name of the property used to describe device I/O regions is "io-regs", its value contains an array of PciPropIoRegs structures:

```
typedef struct {
    PciIoSpace space;
    PciAddr    addr;
    PciSize    size;
    PciAddr    mask;
} PciPropIoRegs;
```

The *space* field specifies the PCI address space where the registers reside:

PCI_PIO_SPACE PCI PIO space

PCI_MEM_SPACE PCI memory space

The *addr* field specifies the PCI start address of the register's range. The *size* field specifies the register's range size in bytes.

The *mask* field specifies which address bits are fixed/floating. This field is used by the PCI bus driver at the resource allocation stage in order to determine the device address decoder constraints. A bit set to zero within *mask* specifies a fixed address bit within *addr*. In other words, the corresponding bit within an allocated address must be the same as within *addr*. A bit set to one within *mask* specifies a floating address bit, that is, any value of the corresponding bit within an allocated address is acceptable.

When the PciPropIoRegs structure is passed to PciBusOps.io_map the I/O register range is already allocated and therefore the *mask* field is meaningless.

The *errHandler* argument is a handler in the device driver, which is invoked by the PCI bus driver, when a PCI bus error occurs while accessing the mapped region. *errCookie* is passed back to the *errHandler* as first argument. (see section PCI Error Handling.)

On success, `K_OK` is returned and appropriate I/O services are returned in the *ioOps* argument. An identifier for the mapped I/O region is also returned in *ioId*. This identifier must be used as first argument to further calls to the `PciIoOps` services.

On failure, `PciBusOps.io_map` returns an error code as follows:

K_ESIZE	A size of zero was specified.
K_ESPACE	Invalid or unsupported PCI address space.
K_EOVERLAP	Not enough virtual address space to map I/O registers.
K_ENOMEM	The system is out of memory.

Note that, before accessing mapped I/O registers, the device driver should initialize an appropriate base address register within the device configuration header correctly.

Perform I/O Access to a Mapped I/O Region

Once the region is successfully mapped, the driver can use the services defined by the `PciIoOps` structure. The PCI bus provides four service routine sets to access a mapped I/O region:

- `PciIoOps.load_8/16/32/64`
- `PciIoOps.store_8/16/32/64`
- `PciIoOps.read_8/16/32/64`
- `PciIoOps.write_8/16/32/64`

There are four service routines in each set which deal with I/O registers of different widths. The `_8`, `_16`, `_32` or `_64` suffix indicates the data size of the transfer on the PCI bus. In all service routines provided by `PciIoOps` the *ioId* argument identifies the mapped I/O region. The *offset* argument specifies the register offset, in bytes, within the mapped region. All these routines handle byte swapping in cases where the endian should be different for the PCI bus and CPU.

Load From a Register

The `PciIoOps.load_xx` routine set returns a value loaded from a device register.

```
    uintxx_f
(*load_xx)  (PciIoId ioId, PciSize offset);
```

The size of the returned value and the data transfer on the PCI bus is specified by the _xx suffix.

Store to a Register

The PciIoOps.store_xx routine set stores a given value into a device register.

```
    void
(*store_xx) (PciIoId ioId, PciSize offset, uintxx_f  value);
```

The size of the given value and the data transfer on the PCI bus is specified by the _xx suffix.

Multiple Read from a Register

The PciIoOps.read_xx routine set loads values from a device register and sequentially writes them into a memory buffer.

```
    void
(*read_xx)  (PciIoId ioId, PciSize offset, uintxx_f* buf, PciSize count);
```

The size of each value loaded and of each data transfer on the PCI bus is specified by the _xx suffix. The *count* argument specifies the number of read transactions to perform. The *buf* argument specifies the address of the memory buffer. The size of this buffer must be at least (*count** size of each data transfer).

Multiple Write to a Register

The PciIoOps.write_xx routine set sequentially reads values from a memory buffer and stores them into a device register.

```
    void
(*write_xx) (PciIoId ioId, PciSize offset, uintxx_f* buf, PciSize count);
```

The size of each value stored and of each data transfer on the PCI bus is specified by the _xx suffix. The count argument specifies the number of write transactions to perform. The buf argument specifies the address of the memory buffer. The size of this buffer must be at least (*count** size of each data transfer).

UnmapDevice I/O Registers

When a driver no longer needs access to the device I/O register, or before closing the connection to the PCI bus, it should unmap the I/O region used for these devices. The PciBusOps.io_unmap is used to unmap an I/O region previously mapped with PciBusOps.io_map, and to release any system resources used for this mapping.

```
    void
(*io_unmap) (PciIoId ioId);
```

The *ioId* argument identifies the region to unmap. When the `PciBusOps.io_unmap` function is called, the driver can no longer use any services defined for the unmapped I/O region.

Access PCI Memory

To perform access to the memory of a PCI device, a PCI driver must:

- Map device memory to the virtual memory space
- Access device memory through the mapped region
- Unmap the region when access is no longer needed

The mapped memory region may be accessed directly using its virtual address.

Map Device Memory

The `PciBusOps.mem_map` service is used to map a memory region from a PCI device, enabling access to this region.

```
    KnError
(*mem_map) (PciDevId       devId,
            PciPropMemRgn* memRgn,
            PciMemAttr     memAttr,
            PciErrHandler  errHandler,
            void*          errCookie,
            void**         memAddr,
            PciMemId*      memId);
```

devId is the identifier returned by `PciBusOps.open`. The *memRgn* structure defines the memory region to map. This structure is an element of the array stored in a property of the device node.

This property should be retrieved by the driver using the device tree API (`dtreePropFind`, `dtreePropLength`, `dtreePropValue`) prior to calling `PciBusOps.mem_map`.

The name of the property used to describe device memory regions is `"mem-rgn"`, its value contains an array of `PciPropMemRgn` structures.

```
typedef struct {
    PciAddr addr;
    PciSize size;
    PciAddr mask;
} PciPropMemRgn;
```

The *addr* field specifies the PCI start address of the memory region. The *size* field specifies the memory region size in bytes. The *mask* field specifies the device address decoder constraints (see section Mapping device I/O registers).

The *memAttr* argument specifies the mapping attributes. A combination of the following flags is allowed:

PCI_MEM_READABLE	The memory region is readable.
PCI_MEM_WRITABLE	The memory region is writable.
PCI_MEM_EXCUTABLE	The memory region is executable.
PCI_MEM_CACHEABLE	The memory region is cacheable.
PCI_MEM_INVERTED	The memory region mapping inverts the byte order. Some MMUs support this type of mapping attribute. This feature may be used by a PCI device driver in order to avoid byte swapping within a memory mapped region. Note that `PciBusOps.mem_map` returns `K_EINVAL` if this type of feature is not supported.

errHandler is a handler in the device driver which is invoked by the PCI bus driver when a PCI bus error occurs while accessing the mapped region. *errCookie* is passed back to the errHandler as first argument. (see section PCI error Handling) .

On success, `K_OK` is returned and the start virtual address of the mapped memory region is returned in the *memAddr* argument. An identifier for the mapped memory region is also returned in *memId.* This identifier must be used to unmap the memory region. On failure, `PciBusOps.mem_map` returns an error code as follows:

K_ESIZE	A size of zero was specified.
K_EINVAL	Invalid or unsupported memory mapping attributes.
K_EOVERLAP	Not enough virtual address space to map.
K_ENOMEM	The system is out of memory.

Once the region is successfully mapped, the device driver may now access the device memory by dereferencing the returned pointer. Note that in this mode, the driver must handle endianning problems independently..

The PCI bus driver may support the 64-bit PCI memory space. In this case, a PCI device driver may use the `PciBusOps.mem64_map` service in order to map a 64-bit memory space region from a PCI device, enabling access to this region.

```
    KnError
(*mem64_map) (PciDevId        devId,
```

```
            Pci64PropMemRgn* memRgn,
            PciMemAttr       memAttr,
            Pci64ErrHandler  errHandler,
            void*            errCookie,
            void**           memAddr,
            Pci64MemId*      memId);
```

`PciBusOps.mem64_map` is analogous to `PciBusOps.mem_map` except the PCI address and size are 64-bit unsigned integer values.

The 64-bit PCI memory region is specified by the `Pci64PropMemRgn` structure.

```
typedef struct {
    Pci64Addr addr;
    Pci64Size size;
    Pci64Addr mask;
} Pci64PropMemRgn;
```

Note that `PciBusOps.mem64_map` returns `K_ENOTIMP` if the 64-bit PCI memory space is not supported by the PCI bus driver. Note that, before accessing a mapped memory region, the device driver should initialize an appropriate base address register within the device configuration header correctly.

Unmap a Memory Region

When a driver no longer needs access to the device memory, or before closing the connection to the PCI bus, it should unmap the memory region. The `PciBusOps.mem_unmap` command is used to unmap a memory region previously mapped with `PciBusOps.mem_map`, and to release any system resources used for this mapping.

```
    void
(*mem_unmap) (PciMemId memId);
```

`PciBusOps.mem64_unmap` is used to unmap a region within the 64-bit PCI memory space, previously mapped with `PciBusOps.mem64_map`, and to release any system resources used for this mapping.

```
    void
(*mem64_unmap) (Pci64MemId memId);
```

The *memId* argument identifies the memory region to unmap. Once the memory region is unmapped, the driver is no longer allowed access to the region.

Direct Memory Access

To perform Direct Memory Access to the system memory, a PCI device driver must:

- Allocate a DMA region on the bus.
- Use services defined by the DMA region object to retrieve the DMA region properties (the virtual and PCI addresses)..
- Program the device DMA engine to perform a DMA transfer.
- Release the DMA region when it is no longer needed.

Allocate a DMA Region

The `PciBusOps.dma_alloc` service allocates a system memory region a of specified size and contiguously maps it to the supervisor address space.

```
    KnError
(*dma_alloc) (PciDevId      devId,
              PciSize       size,
              PciDmaAlign*  dmaConstr,
              PciDmaAttr    dmaAttr,
              PciMemAttr    memAttr,
              PciErrHandler errHandler,
              void*         errCookie,
              PciDmaOps**   dmaOps,
              PciDmaId*     dmaId);
```

The *devId* argument is the identifier returned by `PciBusOps.open`. The *size* argument is the requested size in bytes to allocate for the DMA memory region. The *dmaConstr* argument defines the address decoder constraints for the DMA memory region to allocate. Note that the *dmaConstr* argument may be set to `NULL` specifying that there are no constraints on the DMA region to allocate. Otherwise, it should point to a *PciDmaAlign*structure shown below:

```
typedef struct PciDmaAlign {
        PciAddr   addr;
        PciAddr   alignment;
        PciAddr   floating;
    } PciDmaAlign;
```

The *addr* field specifies the PCI start address of the DMA region. The *alignment* field is a mask which specifies constraints on the start address of the DMA region being allocated. A bit set within *alignment* specifies that the corresponding bit within the start address may take any value, that is, 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 *addr* field. This mask allows the caller to specify an alignment for the start address of the allocated DMA region, by zeroing the required number of least significant bits. By resetting the required number of most significant bits, the caller may also indicate where, in the PCI space, the memory must be allocated.

The *floating* field is a mask which indicates which bits of the returned address can vary while looking through the allocated DMA region for the required size. Bits cleared in the mask must be constant for all addresses in the range of the allocated DMA region. This mask may be used to specify that the allocated amount of memory must not span across a given address boundary.

The *dmaAttr* argument specifies the DMA transfer type. A combination of the following flags is allowed:

PCI_DMA_READABLE	The region is used for DMA read transfer.
PCI_DMA_WRITABLE	The region is used for DMA write transfer.
PCI_DMA_SYNC	Allocate a synchronous DMA region for which no synchronization is needed between access from the DMA engine and the CPU. This attribute may be not supported on all platforms. In this case, `PciBusOps.dma_alloc` returns `K_EINVAL`.

The *memAttr* argument specifies the mapping attributes (see section Mapping Device Memory).

errHandler is a handler in the device driver, which is invoked by the PCI bus driver when a PCI bus error occurs while accessing the DMA region. *errCookie* is passed back to the *errHandler* as first argument. (see section PCI Error Handling) .

On success, `K_OK` is returned and appropriate services are returned in the *dmaOps* argument. An identifier for the DMA region is also returned in *dmaId*. This identifier must be used as first argument to further calls to the `PciDmaOps` services.

On failure, `PciBusOps.dma_alloc` returns an error code as follows:

K_ESIZE	A size of zero was specified.
K_EINVAL	Invalid or unsupported memory mapping attributes.
K_EOVERLAP	Not enough virtual address space to map.
K_ENOMEM	The system is out of memory.

The PCI bus driver may support 64-bit PCI space. In this case, a PCI device driver may use the PciBusOps.dma64_alloc service in order to allocate a DMA region in the 64–bit PCI space:

```
    KnError
(*dma64_alloc) (PciDevId        devId,
                Pci64Size       size,
                Pci64DmaAlign*  dmaConstr,
                PciDmaAttr      dmaAttr,
                PciMemAttr      memAttr,
                Pci64ErrHandler errHandler,
                void*           errCookie,
                Pci64DmaOps**   dmaOps,
                Pci64DmaId*     dmaId);
```

`PciBusOps.dma64_alloc` is analogous to `PciBusOps.dma_alloc` except the PCI address and size are 64-bit unsigned integer values. The `Pci64DmaOps` services are analogous to the `PciDmaOps` ones. Note that `PciBusOps.dma64_alloc` returns `K_ENOTIMP` if the 64-bit PCI space is not supported by the PCI bus driver.

Get the Virtual Address of a DMA Region

The `PciDmaOps.virt_addr` routine returns the virtual start address of a given DMA region (specified by *dmaId*).

```
    void*
(*virt_addr) (PciDmaId dmaId);
```

The driver uses this address to access the DMA region using CPU instructions. Note that the driver should synchronize the region as appropriate, depending on the attributes used when allocating the region (see section Synchronizing the DMA Region).

Get the PCI Address of a DMA Region

The `PciDmaOps.phys_addr` routine returns the PCI start address of a given DMA region (specified by *dmaId*).

```
    PciAddr
(*phys_addr) (PciDmaId dmaId);

    Pci64Addr
(*phys_addr) (PciDmaId dmaId);
```

The driver uses this address to program its DMA engine. Note that the driver should synchronize the region as appropriate, depending on the attributes used when allocating the region (see section Synchronizing the DMA region).

Synchronize the DMA Region

If the DMA region was not allocated using PCI_DMA_SYNC attribute, the device driver should synchronize accesses to the same system memory (the DMA region) from the CPU and the DMA engine correctly.

Depending on the platform, these routines handle possible problems of cache coherency, DMA engine buffers, and others. PciDmaOps.read_sync is a barrier between CPU writes and DMA reads from a given DMA sub-region.

```
    void
(*read_sync) (PciDmaId dmaId, PciSize offset, PciSize size);

    void
(*read_sync) (PciDmaId dmaId, Pci64Size offset, Pci64Size size);
```

PciDmaOps.write_sync is a barrier between DMA writes and CPU reads from a given DMA sub-region.

```
    void
(*write_sync) (PciDmaId dmaId, PciSize offset, PciSize size);

    void
(*write_sync) (PciDmaId dmaId, Pci64Size offset, Pci64Size size);
```

For both routines: the *dmaId* argument identifies the DMA region. *offset* is the offset in bytes from the beginning of the DMA memory region. *offset* and *size* both define the sub-region to synchronize.

Release a DMA Region

When a driver no longer needs an allocated DMA region, or before closing the connection to the PCI bus, it should release the DMA region. The PciBusOps.dma_free is used to release a DMA region, previously allocated with PciBusOps.dma_alloc.

```
    void
(*dma_free) (PciDmaId dmaId);
```

The PciBusOps.dma64_free is used to release a DMA region, previously allocated with PciBusOps.dma64_alloc.

```
    void
(*dma64_free) (Pci64DmaId dmaId);
```

The *dmaId* argument identifies the DMA region to free. Once the DMA region is released, the driver may no longer use the DMA region.

Access PCI Configuration Space

To perform access to the PCI configuration space, a PCI driver must:

- Map a PCI configuration header

- Use services defined by the mapped PCI configuration header object
- Unmap the PCI configuration header when access is no longer needed

Map a PCI Configuration Header

The `PciBusOps.conf_map` service is used to map a PCI configuration header enabling access to its contents.

```
    KnError
(*conf_map) (PciDevId     devId,
             uint8_f      busNum,
             uint8_f      devNum,
             uint8_f      funcNum,
             PciConfOps** confOps,
             PciConfId*   confId);
```

devId is returned by `PciBusOps.open`.

The *busNum*, *devNum* and *funcNum* arguments specify the header address in the PCI configuration space. *busNum* specifies the bus number within the PCI system. It is available for a PCI driver as the `"bus-num"` property is attached to the parent bus node. The bus number is an integer value in a range from `0` to `15`.

devNum specifies the device number on the bus. The device number is an integer value in a range from `0` to `31`. *funcNum* specifies the function number within a multifunctional device. The function number is an integer value in a range from `0` to `3`.

On success, `K_OK` is returned and appropriate PCI configuration services are returned in the *confOps* argument. An identifier for the mapped PCI configuration header is also returned in *confId*. This identifier must be used as first argument to further calls to the `PciConfOps` services.

On failure, `PciBusOps.conf_map` returns an error code as follows:

K_EINVAL	The configuration header with the given address does not exist.
K_ENOMEM	The system is out of memory.

Access a PCI Configuration Header

Once the PCI configuration header is successfully mapped, the driver can use the services defined by the `PciConfOps` structure. The PCI bus provides two service routine sets to access a mapped PCI configuration header:

- PciConfOps.load_8 / 16 / 32
- PciConfOps.store_8 / 16 / 32

There are three service routines in each set which deal with different data sizes. The _8, _16 or _32 suffix indicates the location size.

In all service routines provided by PciConfOps the *confId* argument identifies the mapped PCI configuration header. The *offset* argument specifies the location offset (in bytes) within the PCI configuration header. *offset* must be in a range from 0 to 255. All these routines handle byte swapping should the endian be different for the PCI bus and CPU.

Load From a PCI Configuration Header Location

The PciConfOps.load_xx routine set returns a value loaded from a PCI configuration header location.

```
    uintxx_f
(*load_xx)  (PciConfId confId, uintxx_f offset);
```

The size of the returned value is specified by the _xx suffix.

Store to a PCI Configuration Header Location

The PciConfOps.store_xxroutine set stores a given value into a PCI configuration header location.

```
    void
(*store_xx)  (PciConfId confId, uint8_f offset, uintxx_f value);
```

The size of the given value is specified by the _xx suffix.

Unmap a PCI Configuration Header

When a driver no longer needs access to the PCI configuration header, or before closing the connection to the PCI bus, it should unmap the PCI configuration header.

The PciBusOps.conf_unmap is used to unmap a PCI configuration header previously mapped with PciBusOps.conf_map and to release any resources allocated for this mapping.

```
    void
(*conf_unmap) (PciConfId confId);
```

The *confId* argument identifies the PCI configuration header to unmap. When the PciBusOps.conf_unmap function is called, the driver may no longer use any services defined for the unmapped PCI configuration header.

PCI Interrupt Handling

To connect a handler to a PCI interrupt line, a PCI driver should:

- Disable the interrupt at the device level, typically by accessing the device registers.
- Attach a handler to the interrupt.
- Enable the interrupt at the device level.
- Use services defined by the attached interrupt object.
- Disable the interrupt at the device level.
- Detach the interrupt handler when no longer needed.

Attach a Handler to a PCI Interrupt Line

A PCI device driver can connect a handler to a PCI interrupt line by calling the `PciBusOps.intr_attach` service.

```
    KnError
(*intr_attach) (PciDevId       devId,
                PciPropIntr*   intr,
                PciIntrHandler intrHandler,
                void*          intrCookie,
                PciIntrOps**   intrOps,
                PciIntrId*     intrId);
```

The *devId* argument identifies the connection with the PCI bus driver.

The *intr* argument indicates the PCI interrupt line to which the handler will be connected. There are four interrupt lines available on the PCI bus:

- PCI_INTR_A
- PCI_INTR_B
- PCI_INTR_C
- PCI_INTR_D

Typically, a PCI device driver should find the interrupt to attach to by looking up the `"intr"` property in the device node. The *intrHandler* argument is a handler in the device driver which is invoked by the PCI bus driver when the corresponding interrupt occurs on the bus.

```
typedef PciIntrStatus (*PciIntrHandler) (void* intrCookie);
```

The *intrCookie* is passed back to this interrupt handler as an argument.

This type of PCI interrupt handler must return a *PciIntrStatus* value which indicates to the bus driver whether the interrupt was claimed by the handler, and how it was handled:

PCI_INTR_UNCLAIMED	Must be returned by the interrupt handler if there is no pending interrupt for the device.
PCI_INTR_CLAIMED	Must be returned by the interrupt handler if a pending device interrupt has been serviced by the interrupt handler, and the interrupt has not been enabled (acknowledged) at PCI bus level (see section Enabling / Disabling a serviced interrupt).
PCI_INTR_ACKNOWLEDGED	Must be returned by the interrupt handler if a pending device interrupt has been serviced by the interrupt handler and the interrupt has been enabled (acknowledged) at PCI bus level (see section: Enabling / Disabling an Attached Interrupt).

On success, `K_OK` is returned and services defined on an attached interrupt object are returned in the *intrOps* argument. An identifier for the attached interrupt is also returned in *intrId*. This identifier must be used as first argument to further calls to the `PciIntrOps` services.

The PCI bus architecture allows an interrupt line to be shared among multiple devices residing on the bus. This means that multiple drivers may attach a handler to the same PCI interrupt line. In that case, when this type of interrupt occurs on the bus, all the attached handlers are called in an order which is not defined and is implementation specific. When called, each handler can test the device status to check whether the interrupt is triggered by the device it manages.

As an interrupt line is shareable, a driver must take into account that the interrupt it attaches to is enabled at the bus level, and that it may occur as soon as the attachment is complete. The fact that a driver could already have attached a handler to the interrupt, and enabled it, dictates this behavior. If this is not acceptable for a given driver, it must disable the interrupt at the device level, prior to attaching the handler. This ensures that the handler will not have to service an interrupt (the handler will return PCI_INTR_UNCLAIMED) until it is enabled again at device level.

Mask/Unmask an Attached Interrupt

The PciIntrOps.mask service routine masks the interrupt source specified by *intrId*. Note that PciIntrOps.mask does not guarantee that all other interrupt sources are still unmasked.

```
    void
(*mask) (PciIntrId intrId);
```

The PciIntrOps.unmask service routine unmasks the interrupt source previously masked by PciIntrOps.mask.

```
    void
(*unmask) (PciIntrId intrId);
```

Note that PciIntrOps.unmask does not guarantee that the interrupt source is unmasked immediately. The real interrupt source unmasking may be deferred.

The PciIntrOps.mask / PciIntrOps.unmask pair may be used at either base or interrupt level.

Note that the PciIntrOps.mask / PciIntrOps.unmask pairs must not be nested.

Enable/Disable a Serviced Interrupt

The PciIntrOps.enable and PciIntrOps.disable service routines are dedicated to interrupt handler usage only. In other words, these routines may be called only by an interrupt handler.

The PciIntrOps.enable service routine enables (and acknowledges) the bus interrupt source specified by *intrId*.

```
    PciIntrStatus
(*enable) (PciIntrId intrId);
```

PciIntrOps.enable returns either PCI_INTR_ACKNOWLEDGED or PCI_INTR_CLAIMED. The PCI_INTR_ACKNOWLEDGED return value means that the PCI bus driver has enabled (and acknowledged) interrupt at bus level. The PCI_INTR_CLAIMED return value means that the PCI bus driver has ignored the request and therefore the interrupt source is still disabled (and not acknowledged) at bus level.

The bus driver typically refuses an explicit interrupt acknowledgement (issued by an interrupt handler) for the shared interrupts. In this case, the bus driver will acknowledge interrupts only when all interrupt handlers have been called. In cases where the `PciIntrOps.enable` routine has been called by an interrupt handler, the handler must return the value which has been returned by `PciIntrOps.enable`. Once `PciIntrOps.enable` is called, drivers should be able to handle an immediate re-entrance in the interrupt handler code.

The `PciIntrOps.disable` service routine disables the interrupt source previously enabled by `PciIntrOps.enable`.

```
    void
(*disable) (PciIntrId intrId);
```

If `PciIntrOps.enable` returns `PCI_INTR_ACKNOWLEDGED`, the driver must call `PciIntrOps.disable` prior to returning from the interrupt handler.

When an interrupt occurs, the attached *PciIntrHandler* is invoked with the interrupt source disabled at bus level. This produces the same effect as calling`PciIntrOps.disable` just prior to the handler invocation.

Note that the interrupt handler must return to the bus driver in the same context as it was called, that is, with the interrupt source disabled at bus level. However, the called interrupt handler may use the `PciIntrOps.enable/disable` pair to allow the interrupt to be nested. This feature is typically used by a PCI-to-bus bridge driver when the secondary bus interrupts are multiplexed; for example, multiple secondary bus interrupts are reported through the same primary PCI bus interrupt.

Typically, an interrupt handler of this type of PCI-to-bus bridge driver would take the following actions:

- Identify and disable the secondary bus interrupt source
- Enable the primary PCI bus interrupt source (`enable`)
- Call handlers attached to the secondary bus interrupt source
- Disable the primary PCI bus interrupt source (`disable`)
- Return from the interrupt handler

Detach an Attached Interrupt

When a driver no longer needs to handle an interrupt, or before closing the connection to the PCI bus, it should detach the handler attached to an interrupt line. The `PciBusOps.intr_detach` is used to detach a handler, previously

attached with PciBusOps.intr_attach, and to release any resources allocated for this attachment.

```
    void
(*intr_detach) (PciIntrId intrId);
```

The *intrId* argument identifies the attachment to release. When the PciBusOps.intr_detach function is called, the driver may no longer use the identifier (*intrId*).

PCI Interrupt Acknowledge Transactions

The PciBusOps.intr_vector routine performs a PCI interrupt acknowledge transaction on the PCI bus and returns the interrupt vector delivered on the bus.

```
    uint32_f
(*intr_vector) (PciDevId devId);
```

The *devId* argument identifies the connection with the PCI bus driver.

In cases where the PCI bus driver does not implement this type of feature, the value 0xffffffff is returned, otherwise an interrupt vector / number is returned whose value (semantic) depends on the PCI device which responds to the cycle. Note that only one PCI device on the bus should respond to this type of interrupt acknowledgement cycle.

PciBusOps.intr_vector must be called only from the interrupt handler of the PCI device responding to the PCI interrupt acknowledgement transaction. Typically, this type of device implements PIC features for a sub-bus bridge.

PCI Error Handling

When mapping PCI bus space, a PCI device driver gives an error handler which is invoked by the PCI bus driver when a bus error occurs. Error handlers are used for the following services:

- PciBusOps.io_map
- PciBusOps.mem_map
- PciBusOps.mem64_map
- PciBusOps.dma_map
- PciBusOps.dma64_map

When a programmed or direct memory access is aborted because of a bus error, the PCI bus driver invokes the error handler which is connected for the PCI address and space concerned. A PciErrHandler is called with two arguments: *errCookie* and *err*. The *errCookie* is an opaque given by the caller and passed back when the handler is called.

The *err* argument points to the `PciBusError` structure.

```
typedef struct {
    PciErrorCode  code;
    PciSize       offset;
} PciBusError;
```

The *code* field indicates the type of error that occurred. The *offset* field indicates the offset within the associated region at which the error occurred.

The error code can be as follows:

PCI_ERR_UNKNOWN	Unknown error, for example, the PCI bus driver is unable to determine the reason for the exception.
PCI_ERR_INVALID_SIZE	Invalid (that is, unsupported) access granularity has been used.
PCI_ERR_PARITY	A parity error was detected on the PCI bus.
PCI_ERR_MASTER_ABORT	A master abort is signaled on the PCI bus, that is, a transaction has been aborted by a master because it has not been claimed by any target on the bus.
PCI_ERR_TARGET_ABORT	A target abort is detected on the PCI bus, for example, a claimed transaction has been aborted by a target.

For 64-bit PCI address space, a `PciErrHandler` is called with two arguments: *errCookie* and *err64*. The *err64* argument points to the `Pci64BusError` structure.

```
typedef struct {
    PciErrorCode  code;
    Pci64Size     offset;
} Pci64BusError;
```

As a bus error may be reported in the context of an exception or an interrupt, the implementation of the error handler must be restricted to the interrupt level API.

PCI to PCI Bridge Control Register

The 16-bit bridge control register is specific to the PCI-to-PCI bridge. Bits within the bridge control register may be set or cleared by a device driver using

the PciBusOps.control_set and PciBusOps.control_clr routines respectively.

```
    void
(*control_set) (PciDevId devId, uint16_f ctrl_bits);

    void
(*control_clr) (PciDevId devId, uint16_f ctrl_bits);
```

The *devId* argument identifies the connection with the PCI bus driver. The *mask* argument specifies the bits to set/clear in the bridge control register. *mask* is a combination of the following flags:

PCI_PCI_CTL_VGA	Flag when set/cleared, enables/disables the VGA address decoding (0x000a0000..0x000bffff
PCI_PCI_CTL_ISA	Flag when set/cleared, enables/disables the positive decoding of non-aliased ISA addresses ([0x0000..0xffff] when A[9:8] are zero)

Note that the PciBusOps.control_set/control_clr routines are mainly dedicated to a PCI-to-ISA bridge driver. When the bridge control register is implemented, the PCI-to-PCI bus driver sets/clears the *mask* bits within its bridge control register and, in addition, propagates the bits setting/clearing upstream. In other words, the PCI-to-PCI bus driver invokes an appropriate routine of its parent PCI bus driver. In cases where the PCI bridge does not implement the bridge control register, the PCI bus driver provides an empty implementation of the PciBusOps.control_set and PciBusOps.control_clr routines.

VGA Palette Snooping arbitration on the PCI System

A PCI system may have two video controllers present in the system: a VGA-compatible interface and another graphics controller. In this case, both devices implement a set of color palette registers at the same I/O addresses. When the CPU performs an I/O write to update the palette registers, the palette in both devices must be updated. On the other hand, only one device should claim this I/O transaction and another one should quietly snoop the write data on the bus.

Only a VGA-compatible device can claim read transactions to the palette registers, other graphics controllers do not participate at all in such transactions.

When these types of devices reside on different busses, the upstream device (which is closed to the host bus) should quietly snoop these transactions and the downstream device (which is far from the host bus) should claim these transactions (that is, participate as a target). Otherwise, the transactions will not be visible to the downstream device. Snooping only works when both devices reside along the same bus path on the system.

The `PciBusOps.vga_palette_snoop_enable` and `PciBusOps.vga_palette_snoop_disable` routines provide a means for graphics controllers to dynamically arbitrate palette snooping on the PCI system.

```
    void
(*vga_palette_snoop_enable) (PciDevId devId);

    void
(*vga_palette_snoop_disable) (PciDevId devId);
```

In both routines, the *devId* argument identifies the connection with the PCI bus driver.

At bus initialization time, the PCI bus driver disables VGA palette snooping in its command register. At device initialization time, a graphics controller driver should disable VGA palette snooping in its command register and should invoke `vga_palette_snoop_enable`. This means that the video driver requests the PCI bus to enable delivery on the bus write transactions to the VGA palette registers.

The PCI bus driver checks whether VGA palette snooping is disabled on the bus and, if so, enables VGA palette snooping in its command register and calls `vga_palette_snoop_enable` on its parent PCI bus (if any) in order to propagate the VGA snooping enable request upstream on the PCI system. If VGA palette snooping is already enabled in the bus command register, the bus driver invokes the event handlers with the `PCI_VGA_PALETTE_SNOOP_ENABLE` event code.

The event handler behavior must be as follows. If a given device does not implement VGA palette snooping or the `PCI_CMD_PALETTE_SNOOP` bit is already set in the device command register, the device driver ignores the event and simply returns `K_ENOTIMP`. Otherwise, the device driver sets this bit (within the device command register) and returns `K_OK`. The iteration is interrupted and the bus driver returns from the `vga_palette_snoop_enable` routine, once an event handler returns `K_OK`.

This means that a device claiming write transactions to the VGA palette registers was found and was switched to snooping mode. Note that it may be the device which issued `vga_palette_snoop_enable`. If the iteration is performed and nobody returns `K_OK`, an error condition is met. This means that graphics controllers do not reside along the same bus path on the system. Under this type of error condition, VGA palette snooping does not work and the bus driver behavior is bus driver implementation specific.

When a graphics controller driver is going to be shut down, it should check whether it claims write transactions to the VGA palette registers; for example, whether the `PCI_CMD_PALETTE_SNOOP` bit is cleared in the device command register. If so, the driver should call `vga_palette_snoop_disable` prior to closing the connection to the PCI bus driver.

The PCI bus driver invokes the event handlers with the `PCI_VGA_PALETTE_SNOOP_DISABLE` event code.

The event handler behavior must be as follows. If a given device does not implement VGA palette snooping or the `PCI_CMD_PALETTE_SNOOP` bit is already cleared in the device command register, the device driver must ignore the event and simply return `K_ENOTIMP`. Otherwise, the device driver must clear this bit (within the device command register) and return `K_OK`. The iteration is interrupted and the bus driver returns from the `vga_palette_snoop_disable` routine, once an event handler returns `K_OK`. This means that a device snooping write transactions to the VGA palette registers was found and has been switched to claiming mode. Note that it will not be the device which issued `vga_palette_snoop_disable` because the `PCI_CMD_PALETTE_SNOOP` bit is cleared in its command register. If the iteration is performed and nobody returns `K_OK`, the PCI bus driver invokes `vga_palette_snoop_disable` on its parent PCI bus (if any) and then returns from the `vga_palette_snoop_disable` routine.

PROPERTIES

PCI Specific Node Properties

The *alias* column specifies the alias name which should be used by a PCI bus or device driver referencing the property name. The *name* column specifies the property name ASCII string. The *value* column specifies the type of property value. The *bus* column specifies properties specific to the PCI bus node. The *dev* column specifies properties specific to the PCI device node.

The *m* symbol in the *bus/dev* columns flags mandatory properties. The *o* symbol in the *bus/dev* columns flags optional properties. The - symbol in the *bus/dev* columns flags properties which are not applied to a given node type.

Note that if a node represents a PCI-to-PCI bridge device, both *bus* and *dev* properties are applied to the node. In other words, this type of node is simultaneously a PCI device node on the primary PCI bus and a PCI bus node on the secondary PCI bus.

Alias	Name	Value	Bus	Dev
PCI_PROP_INTR	"intr"	PciPropIntr[]	-	o
PCI_PROP_IO_REGS	"io-regs"	PciPropIoRegs[]	-	o
PCI_PROP_MEM_RGN	"mem-rgn"	PciPropMemRgn[]	-	o
PCI64_PROP_MEM_RGN	"mem64-rgn"	Pci64PropMemRgn[]	-	o
PCI_PROP_DMA_BURST	"dma-burst"	PciPropDmaBurst	m	-
PCI_PROP_DMA_MIN_SIZE	"dma-min-size"	PciPropDmaMinSize	m	-
PCI_PROP_DEV_ID	"dev-id"	PciPropDevId	o	m
PCI_PROP_VEND_ID	"vend-id"	PciPropVendId	o	m
PCI_PROP_BUS_NUM	"bus-num"	PciPropBusNum	m	-
PCI_PROP_SUB_BUS_NUM	"sub-bus-num"	PciPropSubBusNum	o	-
PCI_PROP_DEV_NUM	"dev-num"	PciPropDevNum	-	m
PCI_PROP_FUNC_NUM	"func-num"	PciPropFuncNum	-	m
PCI_PROP_BYTE_ORDER	"byte-order"	PciPropByteOrder	m	-
PCI_PROP_CLOCK_FREQ	"clock-freq"	PciPropClockFreq	o	-

When the value of the property is an array (for example, `PciPropIntr[]`), the size of the array defines the number of elements in the array. A PCI device driver may then iterate through the array in order to perform an action for each element (that is, to attach an interrupt handler to each device interrupt line). The size of the array is the size of the property value returned by `dtreePropValue` divided by the size of its element type (that is, `sizeof(PciPropIntr)`.

Dynamic Resource Allocation

The PCI bus driver may support dynamic allocation on the PCI bus resources.

In cases where dynamic resource allocation is supported, a PCI device driver may use the `PciBusOps.resource_alloc` service routine in order to allocate a bus resource at run time.

```
    KnError
(*resource_alloc) (PciDevId dev_id, DevProperty prop);
```

devId is returned by `PciBusOps.open`.

prop specifies the bus resource being allocated The `PciBusOps.resource_alloc` service routine allocates a given bus resource and, if the allocation request is satisfied, updates the device node properties in order to add the newly allocated bus resource.

On success, `PciBusOps.resource_alloc` returns `K_OK`. On failure, one of the following error codes is returned:

K_ENOTIMP	Dynamic resource allocation is not supported by the bus driver.
K_EUNKNOWN	The property name is unknown.
K_EINVAL	The property value is invalid.
K_EFAIL	The bus resource is not available.
K_ENOMEM	The system is out of memory.

When a dynamically allocated bus resource is no longer being used, a PCI device driver may release it by calling the `PciBusOps.resource_free` service routine.

```
    void
(*resource_free)  (PciDevId dev_id, DevProperty prop);
```

devId is returned by `PciBusOps.open`. *prop* specifies the bus resource being released. The `PciBusOps.resource_free` service routine releases a given bus resource and updates the device node properties in order to remove the bus resource being released. The following bus resource properties may be dynamically allocated and released:

- PCI_PROP_INTR
- PCI_PROP_IO_REGS
- PCI_PROP_MEM_RGN
- PCI_PROP_BUS_NUM
- PCI64_PROP_MEM_RGN
- PCI_PROP_SUB_BUS_NUM

Note that the `PciBusOps.resource_alloc` and `PciBusOps.resource_free` routines should not be used by a simple device driver. This type of driver should assume that all needed resources have already been allocated and specified as properties in the device node by the bus driver. The driver should only find the property and call an appropriate service routine to pass a pointer to the property value.

It is not always possible to determine all needed resources prior to device initialization. A typical case is a bus-to-bus bridge driver which will only discover devices residing on the secondary bus after the bus bridge hardware has been initialized. In this case, when `drv_init` is called, the bus-to-bus bridge node will only contain resources needed for the bus-to-bus bridge device itself (for example, internal bus-to-bus bridge registers). Once devices residing on the secondary bus are discovered, the bus-to-bus bridge driver would request additional primary bus resources in order to satisfy the resource requirements for these devices.

Another example is a bus which supports hot-pluggable devices. On this type of bus, the primary bus resources allocated by the hot-pluggable bus driver depend on devices currently plugged into the secondary bus. Usually, the resource requirements are changed when a hot-plug insertion / removal occurs. Note that dynamic resource allocation might also be used by a device driver which implements lazy resource allocation. In this type of driver, the bus resource allocation could be performed at open time using `PciBusOps.resource_alloc`. The dynamically allocated resources could then be released at close time using `PciBusOps.resource_free`.

ALLOWED CALLING CONTEXTS

Services	Base level	DKI thread	Interrupt	Blocking
PciBusOps.open	-	+	-	+
PciBusOps.close	-	+	-	+
PciBusOps.intr_attach	-	+	-	+
PciBusOps.intr_detach	-	+	-	+
PciBusOps.intr_vector	-	+	-	+
PciBusOps.io_map	-	+	-	+
PciBusOps.io_unmap	-	+	-	+
PciBusOps.mem_map	-	+	-	+
PciBusOps.mem_unmap	-	+	-	+
PciBusOps.mem64_map	-	+	-	+
PciBusOps.mem64_unmap	-	+	-	+
PciBusOps.dma_alloc	-	+	-	+
PciBusOps.dma_free	-	+	-	+
PciBusOps.dma64_alloc	-	+	-	+
PciBusOps.dma64_free	-	+	-	+
PciBusOps.conf_map	-	+	-	+
PciBusOps.conf_unmap	-	+	-	+
PciBusOps.vga_palette_snoop_enable	+	+	-	-
PciBusOps.vga_palette_snoop_disable	+	+	-	-
PciBusOps.control_set	+	+	-	-
PciBusOps.control_clr	+	+	-	-
PciBusOps.resource_alloc	-	+	-	+
PciBusOps.resource_free	-	+	-	+
PciIoOps.load_xx	+	+	+	-
PciIoOps.store_xx	+	+	+	-
PciIoOps.read_xx	+	+	+	-
PciIoOps.write_xx	+	+	+	-
PciDmaOps.phys_addr	+	+	+	-
PciDmaOps.virt_addr	+	+	+	-
PciDmaOps.read_sync	+	+	+	-
PciDmaOps.write_sync	+	+	+	-
PciIntrOps.mask	+	+	+	-
PciIntrOps.unmask	+	+	+	-
PciIntrOps.enable	-	-	+	-
PciIntrOps.disable	-	-	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`dtreeNodeRoot`(9DKI), `svDriverRegister`(9DKI), `svMemAlloc`(9DKI), `svPhysAlloc`(9DKI), `svPhysMap`(9DKI), `svDkiThreadCall`(9DKI), `svTimeoutSet`(9DKI), `usecBusyWait`(9DKI), `DISABLE_PREEMPT`(9DKI)

NAME

quicc – QUICC bus driver interface

SYNOPSIS

```
#include <ddi/quicc/quicc.h>
```

FEATURES

DDI

DESCRIPTION

The QUICC bus driver offers an API for QUICC device driver development.

EXTENDED DESCRIPTION

The QUICC bus API is an abstraction of Motorola's QUICC micro-contollers' internal peripheral bus and external bus mechanisms.

The API caters for the following QUICC functionalities:

- QUICC internal and external interrupt management
- access to QUICC device I/O registers
- access to QUICC parallel I/O port's pins
- DMA management
- management of QUICC shared resources (Baud Rate Generators, Communication processor's channel commands)

Connecting a device driver to the QUICC bus

Initially, a QUICC device driver must register itself in the kernel driver registry. This should be done from its main function using `svDriverRegister`. The driver must set the bus class to "quicc" in its registry entry, and specify the lowest version number of the QUICC bus interface required to run correctly. This registration allows the kernel to call back the QUICC device driver's `drv_probe`, `drv_bind` and `drv_init` routines, at bus probing, binding and initialization phases respectively.

Within the `drv_probe` or `drv_init` routine, the QUICC device driver may establish a connection with the parent QUICC bus driver, as described below. Once such a connection is established, the QUICC device driver may use services provided by the QUICC bus driver. Note that a connection to the bus driver can only be established from within the driver's `drv_probe` or `drv_init` routine.

Initialization Connection

The `drv_init` routine is defined if the *drv_init* field of the driver registry entry is set to a non-zero value. If `drv_init` is defined, then the QUICC bus driver will invoke it at bus initialization time. Note that the `drv_init` routine is called in the DKI thread context which means that is it only possible to invoke the QUICC bus services allowed in the DKI thread context.

The `drv_init` routine is called with three arguments:

DevNode	The QUICC device driver's own node, which identifies the node associated with the device in the device tree.
QuiccBusOps	The QUICC bus operations structure, which defines the QUICC bus API and its version.

QuiccId The QUICC bus identifier, which is an opaque value used to connect the device driver to the QUICC bus when calling the `QuiccBusOps.open` operation.

From this point, `QuiccBusOps.open` must be the first call issued by the QUICC device driver to the QUICC bus driver:

```
          KnError
   (*open) (QuiccId           quiccId,
            DevNode           devNode,
            QuiccEventHandler eventHandler,
            QuiccLoadHandler  loadHandler,
            void*             cookie,
            QuiccDevId*       devId);
```

The above function establishes a connection between the device and the bus. The device is identified by the *devId* value, returned as the last argument. The *devId* identifier is used for many other services defined in the `QuiccBusOps` structure.

The *quiccId* and *devNode* fields are given by the QUICC bus driver as parameters to the driver's `drv_init` routine.

eventHandler is a handler in the device driver which is invoked by the QUICC bus driver when a QUICC bus event occurs. Note that *eventHandler* is optional and must be set to NULL when it is not implemented by the device driver. The *cookie* argument is passed back to this handler as a first argument.

This QUICC bus event handler takes two additionnal parameters:

- the bus event type, which is one of the following:

`QUICC_SYS_SHUTDOWN` Notifies a device driver that the system is going to be shut down. The device driver should reset the device hardware and return from the event handler. Note that the driver must not notify clients nor release used resources.

`QUICC_DEV_SHUTDOWN` Notifies a device driver that the device should be shut down. The device driver should notify driver clients (via `svDeviceEvent`) that the device is going to be shut down and then it should return from the event handler. Once the device entry is released from the last driver client, the device registry module invokes a driver call-back handler. Within this handler, the device driver should reset the device hardware, release all used resources and close the bus connection, invoking QuiccBusOps.close. Note that the `QUICC_DEV_SHUTDOWN` event may be used by

a bus driver to confiscate (or to re-allocate) bus resources. The QUICC bus driver will invoke the driver's `drv_init` routine again once the bus resources are re-allocated for the device.

`QUICC_DEV_REMOVAL` — Notifies a device driver that the device has been removed from the bus and therefore the device driver instance has to be shut down. The actions taken by the driver are similar to those for `QUICC_DEV_SHUTDOWN`, detailed above, except that the device hardware must not be accessed by the driver.

- an event type specific structure pointer:

`NULL` — This argument is always NULL for QUICC bus events.

As the event handler may be executed in the context of an interrupt, its implementation must be restricted to the API allowed at interrupt level.

loadHandler is a handler in the device driver which is invoked by the QUICC bus driver when a new driver appears in the system (for example, a new driver is downloaded at run time). Note that *loadHandler* is optional and must be set to NULL when it is not implemented by the device driver. The *cookie* argument is passed back to this handler as a unique argument. Typically, a leaf QUICC device driver is not affected by such an event. *loadHandler* is usually used by a QUICC nexus driver (for example, a PCI/ISA bus bridge) when trying to apply a newly downloaded driver to its child devices which are not yet being serviced. Not that *loadHandler* is called in the DKI thread context.

On success, `QuiccBusOps.open` returns *K_OK* and a valid identifier is returned as the *devId* argument. This identifier must be used to call other services defined in the `QuiccBusOps` structure. Some of these services also return an "Ops" structure which defines the new services on a new object instance (for exampe, a memory region). Others perform a simple service for the device driver, without giving access to a subpart of the API.

On failure, `QuiccBusOps.open` returns an error code as follows

`K_EINVAL` — The device node is invalid, that is, the device node is not a child of the bus node.

`K_EBUSY` — A connection is already open for the given device node.

`K_ENOMEM` — The system is out of memory.

To release the connection with the QUICC bus, the driver must call the `QuiccBusOps.close` service:

```
void
        (*close) (QuiccDevId devId);
```

After being disconnected from the QUICC bus, the device driver may no longer use any of the QUICC bus API services.

Probing Connection

The probe routine is defined if the *drv_probe* field of `DrvRegEntry` is set to a non-zero value. If the `drv_probe` routine is defined, then the QUICC bus driver invokes it at bus probing time. Note that the drv_probe routine is called in the context of a DKI thread which means that it is only possible to invoke the QUICC bus services allowed in a DKI thread context.

The `drv_probe` routine is called with three arguments:

DevNode	The node associated with the parent QUICC bus in the device tree.
QuiccBusOps	The QUICC bus operations structure, which defines the QUICC bus API and its version.
QuiccId	Tthe QUICC bus identifier, which is an opaque value used to connect the device driver to the QUICC bus when calling the `QuiccBusOps.open` operation.

The `drv_probe` routine is called first by the bus driver. That allows the QUICC device driver to discover a device (which can be serviced by the device driver) on the bus and to create a device node for the device in the device tree. If `drv_probe` creates a node corresponding to a device residing on the bus, it should attach appropriate properties to the device node, specifying the driver's required bus resources (for example, "intr" and / or "mem-rgn"). By examining the node properties, the QUICC bus driver will allocate bus resources for the device (if needed) and / or will check resource conflicts with other devices residing on the bus.

The `drv_bind` routine is called by the bus driver when the probing phase is complete. It allows the device driver to perform a driver-to-device binding. Typically, a QUICC device driver examines the "channel" device node property in order to check whether it can service the device. If the check is positive, the driver binds itself to the device. This binding is done by attaching the "driver" property to the device node. The "driver" property value is a NULL terminated ASCII string specifying the driver name. The driver name specified in the property must match the driver name specified in the driver registry entry. Under these conditions, the QUICC bus driver will invoke the driver's `drv_init` routine for the device node as described above. Note that the `drv_init` routine is invoked by the bus driver if, and only if, the bus resources required for the device are successfully allocated, or checked, by the bus driver. Note also, that the `drv_bind` routine is called in the context of a DKI thread

which means that it is only possible to invoke the QUICC bus services allowed in a DKI thread context.

Devices residing on the QUICC bus can be discovered by using the QUICC bus services. To enable this, a connection must be established between the QUICC bus driver and the QUICC device driver. In order to establish such a connection, the device driver has to create a device node and attach it to the bus node. This device node may be temporary but it is mandatory as an argument to the QuiccBusOps.open routine. When the probing process is complete and the connection is closed, depending on the probing results, the device driver should either delete the temporary device node or change it to a real device node. In the latter case, the device that may be serviced by the driver is located on the bus.

It is important that the device driver does not create redundant device nodes. In particular, when the device driver discovers a device and wishes to create a device node corresponding to the device, it should check that there is no existing device node (among the bus child nodes) representing the same device.

If a connection to the QUICC bus is established by the `drv_probe` routine, it should be closed (via `QuiccBusOps.close)` before leaving the routine.

Accessing QUICC devices I/O registers

To perform I/O access to QUICC device registers, a QUICC driver must:

- map the device's registers into an I/O region
- use services defined by the mapped I/O region to access the registers
- unmap the region when access is no longer needed

Mapping device I/O registers

The `QuiccBusOps.io_map` service is used to map an I/O region of a QUICC device so that it is accessible:

```
KnError
    (*io_map)(QuiccDevId       devId,
              QuiccPropIoRegs* ioRgn,
              QuiccErrHandler  errHandler,
              void*            errCookie,
              QuiccIoOps**     ioOps,
              QuiccIoId*       ioId)
```

The *devId* field is returned by `QuiccBusOps.open`.

The *ioRgn* structure defines the I/O region to map. This structure is an element of the array stored in a property of the device node. This property should be retrieved by the driver using the device tree API (see the `dtreePropFind`, `dtreePropLength`, `dtreePropValue` manpages) prior to the call to `QuiccBusOps.io_map`.

The name of the property used to describe device I/O regions is "io-regs". Its value contains an array which stores the `QuiccPropIoRegs` structure, shown below:

```
typedef struct QuiccPropIoRegs {
        QuiccSpace   space;
        QuiccAddr    addr;
        QuiccSize    size;
        QuiccAddr    mask;
    } QuiccPropIoRegs;
```

The *space* field specifies the QUICC address space where the registers reside:

`QUICC_INTERNAL`	QUICC internal space (internal memory space).
`QUICC_EXTERNAL`	QUICC external space (bus space).

The *addr* field specifies the QUICC start address of the registers range.

The *size* field specifies the registers' range size in bytes.

The *mask* field specifies which address bits are fixed and which are floating. This field is used by the QUICC bus driver at the resource allocation stage in order to determine the device address decoder constraints. A bit set to zero within *mask* specifies a fixed address bit within *addr*. In other words, a bit within an allocated address must correspond to the same bit within *addr*. A bit set to one within *mask* specifies a floating address bit, that is, any value of the corresponding bit within an allocated address is acceptable. When the `QuiccPropIoRegs` structure is passed to `QuiccBusOps.io_map` the I/O register range is already allocated and therefore the mask field is meaningless.

errHandler is a handler in the device driver which is invoked by the QUICC bus driver when a QUICC bus error occurs while the driver is accessing the mapped region. *errCookie* is passed back to the `errHandler` as a first parameter.

On failure, `QuiccBusOps.io_map` will return one of the following error codes:

`K_ESIZE`	A size of zero was specified.
`K_ESPACE`	Invalid, or not supported, QUICC address space.
`K_EOVERLAP`	Not enough virtual address space to map I/O registers.
`K_ENOMEM`	The system is out of memory.

Performing I/O access to a mapped I/O region

Once the region is successfully mapped, the driver can use the services defined by the `QuiccIoOps` structure.

The QUICC bus provides four service routine sets to access a mapped I/O region:

- `QuiccIoOps.load_8/16/32`
- `QuiccIoOps.store_8/16/32`
- `QuiccIoOps.read_8/16/32`
- `QuiccIoOps.write_8/16/32`

There are three service routines in each set which deal with I/O registers of different widths. The _8, _16 or _32 suffix indicates the data size of the transfer on the QUICC bus.

In all service routines provided by QuiccIoOps the *ioId* argument identifies the mapped I/O region.

The offset argument specifies the register offset, in bytes, within the mapped region.

All these routines handle byte swapping in case the endian is different for the QUICC bus and the CPU.

Load from a register

The QuiccIoOps.load_xx routine set returns a value loaded from a device register:

```
uintxx_f
    (load_xx)(QuiccIoId ioId, QuiccSize offset);
```

The size of the returned value and of the data transfer on the QUICC bus is specified by the _xx suffix.

Store to a register

The QuiccIoOps.store_xx routine set stores a given value into a device register:

```
void
    (*store_xx)(QuiccIoId ioId, QuiccSize offset, uintxx_f value);
```

The size of the given value and of the data transfer on the QUICC bus is specified by the _xx suffix.

Multiple read from a register

The QuiccIoOps.read_xx routine set loads values from a device register and sequentially writes them into a memory buffer:

```
void
    (*read_xx)(QuiccIoId ioId, QuiccSize offset, uintxx_f* buf, QuiccSize count);
```

The size of each value loaded, and of each data transfer on the QUICC bus is specified by the _xx suffix.

The *count* argument specifies the number of write transactions to perform.

The *buf* argument specifies the address of the memory buffer. The size of this buffer must be at least the value of *count* multiplied by the size of each data transfer.

Multiple write to a register

The QuiccIoOps.write_xx routine set sequentially reads values from a memory buffer and stores them in a device register:

```
void
    (*write_xx)(QuiccIoId ioId, QuiccSize offset, uintxx_f* buf, QuiccSize count);
```

The size of each value stored, and of each data transfer on the QUICC bus, is specified by the _xx suffix.

The *count* argument specifies the number of write transactions to perform.

The *buf* argument specifies the address of the memory buffer. The size of this buffer must be at least the value of *count* multiplied by the size of each data transfer.

Unmapping device I/O registers

When a driver no longer needs access to the device I/O registers, or before closing the connection to the QUICC bus, it should unmap the I/O region used for these devices.

`QuiccBusOps.io_unmap` is used to unmap an I/O region previously mapped with `QuiccBusOps.io_map` and to release any resources allocated for this mapping:

```
void
    (*io_unmap)(QuiccIoId ioId);
```

The *ioId* argument identifies the region to unmap.

When the `QuiccBusOps.io_unmap` function is called, the driver is no longer able to use any services defined for the unmapped I/O region.

Accessing QUICC memory

To access the memory of a QUICC device, a QUICC driver must:

- map device memory to the local virtual memory space
- access device memory through the mapped region
- unmap the region when access is no longer needed

The mapped memory region may be accessed directly using its virtual address.

Mapping device memory

The `QuiccBusOps.mem_map` service is used to map a memory region from a QUICC device, enabling access to this region:

```
KnError
    (*mem_map) (QuiccDevId       devId,
                QuiccPropMemRgn* memRgn,
                QuiccMemAttr     memAttr,
                QuiccErrHandler  errHandler,
                void*            errCookie,
                void**           memAddr,
                QuiccMemId*      memId);
```

devId is the identifier returned by `QuiccBusOps.open`.

The *memRgn* structure defines the memory region to map. This structure is an element of the array stored in a property of the device node. The property should be retrieved by the driver using the device tree (see the `dtreePropFind, dtreePropLength`, `dtreePropValue` manpages) prior to a call to `QuiccBusOps.mem_map` .

The name of the property used to describe device memory regions is "mem-rgn". Its value contains an array of `QuiccPropMemRgn` structures shown below:

```
typedef struct QuiccPropMemRgn {
        QuiccSpace  space;
        QuiccAddr   addr;
        QuiccSize   size;
        QuiccAddr   mask;
    } QuiccPropMemRgn;
```

The *space* field specifies where the memory region resides within the QUICC memory address space. Either in:

`QUICC_INTERNAL` QUICC internal space (internal memory space) , or

`QUICC_EXTERNAL` QUICC external space (bus spaces).

The *addr* field specifies the QUICC start address of the memory region.

The *size* field specifies the memory region size in bytes.

The *mask* field specifies the device address decoder constraints. (See section *Mapping device I/O registers*).

The *memAttr* argument specifies the mapping attributes. A combination of the following flags is allowed:

`QUICC_MEM_READABLE` the memory region is readable

`QUICC_MEM_WRITABLE` the memory region is writable

`QUICC_MEM_EXCUTABLE` the memory region is executable

`QUICC_MEM_CACHEABLE` the memory region is cacheable

`errHandler` is a handler in the device driver which is invoked by the QUICC bus driver when a QUICC bus error occurs while the driver is accessing the mapped region. *errCookie* is passed back to the `errHandler` as a first argument. (see the section *QUICC error Handling*.)

On success, `K_OK` is returned and the start virtual address of the mapped memory region is returned in the *memAddr* argument. An identifier for the mapped memory region is also returned in *memId*. This identifier must be used to unmap the memory region.

On failure, `QuiccBusOps.mem_map` will return one of the following error codes:

K_ESIZE A size of zero has been specified.

K_EINVAL Invalid, or not supported, memory mapping attributes.

K_EOVERLAP Not enough virtual address space to map.

K_ENOMEM The system is out of memory.

Once the region is successfully mapped, the device driver may access the device memory by de-referencing the returned pointer. Note that in this mode, the driver must handle all problems of endianning.

Unmapping a memory region

When a driver does not require access to the device memory, or before closing the connection to the QUICC bus, it should unmap the memory region.

The `QuiccBusOps.mem_unmap` is used to unmap a memory region (previously mapped with `QuiccBusOps.mem_map`) and to release any system resources used for this mapping:

```
void
    (*mem_unmap) (QuiccMemId memId);
```

The *memId* argument identifies the memory region to unmap.

Once the memory region is unmapped, the driver may no longer access that region.

Direct Memory Access (DMA)

To provide DMA to the system memory, a QUICC device driver must:

- allocate a DMA region
- use services defined by the DMA region object to retrieve the DMA region properties (the virtual and QUICC addresses)
- program the device DMA engine to perform a DMA transfer
- release the DMA region when no longer needed

Allocating a DMA region

The `QuiccBusOps.dma_alloc` service allocates a memory region of a specified size and contiguously maps it to the supervisor address space:

```
KnError
    (*dma_alloc) (QuiccDevId      devId,
                  QuiccSize       size,
                  QuiccDmaAlign*  dmaConstr, /* NULL if no constraints */
                  QuiccDmaAttr    dmaAttr,
                  QuiccMemAttr    memAttr,
                  QuiccErrHandler errHandler,
                  void*           errCookie,
                  QuiccDmaOps**   dmaOps,
                  QuiccDmaId*     dmaId);
```

The *devId* argument is the identifier returned by `QuiccBusOps.open`.

The *size* argument is the requested size, in bytes, to allocate for the DMA memory region.

The *dmaConstr* argument defines the address decoder constraints for the DMA memory region being allocated. Note that the `dmaConstr` argument may be set to `NULL`, specifying that there are no constraints on the DMA region being allocated. Otherwise, it should point to a `QuiccDmaAlign` structure, shown below:

```
typedef struct QuiccDmaAlign {
        QuiccAddr   addr;
        QuiccAddr   alignment;
        QuiccAddr   floating;
    } QuiccDmaAlign;
```

The *addr* field specifies the QUICC start address of the DMA region.

The *alignment* field is a mask which specifies constraints on the start address of the DMA region being allocated. A bit set within *alignment* specifies that the corresponding bit within the start address may take any value, that is, 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 *addr* field. This mask allows the caller to specify an alignment for the start address of the allocated DMA region, by zeroing the required number of least significant bits. By resetting the required number of most significant bits, the caller may also indicate in which part of the QUICC space the memory must be allocated.

The *floating* field is a mask which indicates which bits of the returned address can vary while searching the allocated DMA region for the required size. Bits cleared in the mask must be constant for all addresses in the allocated DMA region range. This mask may be used to specify that the allocated amount of memory must not span across a given address boundary.

The *dmaAttr* argument specifies the DMA transfer type. A combination of the following flags is allowed:

`QUICC_DMA_READABLE`	Region used for DMA read transfer.
`QUICC_DMA_WRITABLE`	Region used for DMA write transfer.
`QUICC_DMA_SYNC`	Allocate a synchronous DMA region for which no synchronization is needed for access to the DMA engine and the CPU. This attribute may not be supported on a given platform. In this case, `QuiccBusOps.dma_alloc` returns `K_EINVAL`.
`QUICC_DMA_DPRAM`	Allocate a DMA region in Dual Ported RAM, rather than in system memory.

The *memAttr* argument specifies the mapping attributes (see section Mapping Device Memory).

errHandler is a handler in the device driver, which is invoked by the QUICC bus driver when a QUICC bus error occurs while accessing the DMA region. *errCookie* is passed back to the `errHandler` as a first argument. (See section QUICC Error Handling.)

On success, K_OK is returned and appropriate services are returned in the *dmaOps* argument. An identifier for the DMA region is also returned in *dmaId*. This identifier must be used as a first argument to call the QuiccDmaOps services.

On failure, QuiccBusOps.dma_alloc will return one of the following errors:

K_ESIZE	A size of zero was specified.
K_EINVAL	Invalid, or not supported, memory mapping attributes.
K_EOVERLAP	Not enough virtual address space to map.
K_ENOMEM	The system is out of memory.

Getting the virtual address of DMA region

The QuiccDmaOps.virt_addr routine returns the virtual start address of a given DMA region (specified by *dmaId*):

```
        void*
      (*virt_addr) (QuiccDmaId dmaId);
```

The driver uses this address to access the DMA region using the CPU instructions.

Getting the QUICC address of DMA region

The QuiccDmaOps.phys_addr routine returns the QUICC start address of a given DMA region (specified by *dmaId*):

```
QuiccAddr
    (*phys_addr) (QuiccDmaId dmaId);
```

The driver uses this address to program its DMA engine. (Some QUICC internal controllers need to be progammed with the offset of the DMA region within QUICC DPRAM. In such cases, the start address of the DPRAM should be substracted from the address returned by *phys_addr*. The start address of DPRAM is stored in the QUICC bus node "dpram-rgn" property.)

Note that the driver should synchronize the region depending on the attributes used when allocating the region (see section *Synchronizing the DMA Region*).

Synchronizing the DMA region

If the DMA region was not allocated using the QUICC_DMA_SYNC attribute, the device driver should correctly synchronize access for the CPU and the DMA engine to the same memory (the DMA region). Depending on the platform, these routines handle possible problems with cache coherency, DMA engine buffers, and others.

QuiccDmaOps.read_sync is a barrier between CPU writes and DMA reads from a given DMA sub-region:

```
void
    (*read_sync) (QuiccDmaId dmaId, QuiccSize offset, QuiccSize size);
```

`QuiccDmaOps.write_sync` is a barrier between DMA writes and CPU reads from a given DMA sub-region:

```
void
    (*write_sync) (QuiccDmaId dmaId, QuiccSize offset, QuiccSize size);
```

For both routines:

- *dmaId* identifies the DMA region
- *offset* is the offset, in bytes, from the beginning of the DMA memory region
- *offset* and *size* define the sub-region to synchronize

Releasing a DMA region

When a driver no longer needs an allocated DMA region, or before closing the connection to the QUICC bus, it should release the DMA region.

The `QuiccBusOps.dma_free` is used to release a DMA region, previously allocated with `QuiccBusOps.dma_alloc`:

```
void
    (*dma_free) (QuiccDmaId dmaId);
```

The *dmaId* argument identifies the DMA region to be freed.

Once the DMA region is released, the driver is no longer able to access it.

QUICC interrupt handling

To connect a handler to a QUICC interrupt source, a QUICC driver should:

- disable the interrupt at the device level, usually by accessing the device registers
- attach a handler to the interrupt
- enable the interrupt at the device level
- use services defined for the attached interrupt object
- disable the interrupt at the device level
- detach the interrupt handler when no longer needed

Attaching a Handler to a QUICC Interrupt Source

A QUICC device driver can connect a handler to a QUICC interrupt source by calling the `QuiccBusOps.intr_attach` service:

```
KnError
    (*intr\_attach) (QuiccDevId       devId,
                     QuiccPropIntr*   devIntr,
                     QuiccIntrHandler devIntrHandler,
                     void*            devIntrCookie,
                     QuiccIntrOps**   intrOps,
                     QuiccIntrId*     intrId);
```

The *devId* argument identifies the connection with the QUICC bus driver.

The *intr* argument indicates which QUICC interrupt source the handler will be connected to. Typically, a QUICC device driver should find the interrupt to

attach to by looking at the "intr" property in the device node. The value of this type of property is a `QuiccPropIntr` structure, shown below:

```
typedef struct QuiccPropIntr {
        QuiccIntrSrc  src;
        QuiccIntrMode mode;
    } QuiccPropIntr;
```

The *src* field specifies the QUICC interrupt source. The interrupt sources available are specific to each QUICC bus driver implementation, and depend on the underlying QUICC micro-controller.

The *mode* field specifies the QUICC interrupt mode:

`QUICC_LEVEL_SENSITIVE` QUICC interrupt source is level sensitive (triggered on level)

`QUICC_EDGE_SENSITIVE` QUICC interrupt source is edge sensitive (triggered on edge)

The *intrHandler* argument is a handler in the device driver which is invoked by the QUICC bus driver when the corresponding interrupt occurs:

```
typedef QuiccIntrStatus (*QuiccIntrHandler) (void* intrCookie);
```

The *intrCookie* is passed back to the interrupt handler as an argument. The QUICC interrupt handler must return a `QuiccIntrStatus` value, which indicates to the bus driver whether the interrupt was claimed by the handler, and, if it was, how it was handled:

`QUICC_INTR_UNCLAIMED` Must be returned by the interrupt handler if there is no pending interrupt for device.

`QUICC_INTR_CLAIMED` Must be returned by the interrupt handler if a pending device interrupt has been serviced by the interrupt handler and the interrupt has not been enabled (acknowledged) at QUICC bus level. (See section Enabling/Disabling a Serviced Interrupt.)

`QUICC_INTR_ACKNOWLEDGED` Must be returned by the interrupt handler if a pending device interrupt has been serviced by the interrupt handler and the interrupt has been enabled (acknowledged) at QUICC bus level. (See section Enabling/Disabling an Attached Interrupt.)

On success, `K_OK` is returned and services defined on an attached interrupt object are returned in the *intrOps* argument. An identifier for the attached interrupt is also returned in *intrId*. This identifier must be used as a first argument to call the `QuiccIntrOps` services.

The QUICC bus allows an interrupt source to be shared among multiple devices if the interrupt is level sensitive. Multiple drivers may attach a handler to the same QUICC interrupt line. In that case, when an interrupt occurs on the bus, all the attached handlers are called in the order they were attached. When called, each handler can test the device status to check whether the interrupt was triggered by the device it manages.

As an interrupt line is shareable, a driver must consider that the interrupt it attaches to is enabled at the bus level and that the interrupt may occur as soon as the attachment is complete. The fact that a driver could already have attached a handler to the interrupt, and enabled it, dictates this behaviour.

If this is not acceptable for a given driver, it must disable the interrupt at the device level, prior to attaching to the handler. This ensures that the interrupt will not occur, and that the handler will not be called until the interrupt is enabled again at the device level.

On failure, `QuiccBusOps.intr_attach` will return one of the following error messages:

K_EINVAL	specified *intr* is invalid
K_EFAIL	specified interrupt source is already configured differently than required in *devIntr*
K_EBUSY	specified edge sensitive interrupt source is already attached to another driver (edge sensitive interrupts are not shareable)
K_ENOMEM	not enough system memory

Masking/Unmasking an attached interrupt

The `QuiccIntrOps.mask` service routine masks the interrupt source specified by *intrId*:

```
void
    (*mask) (QuiccIntrId intrId);
```

Note that `QuiccIntrOps.mask` does not guarantee that all other interrupt sources are still unmasked.

The `QuiccIntrOps.unmask` service routine unmasks the interrupt source, previously masked by `QuiccIntrOps.mask`:

```
void
    (*unmask) (QuiccIntrId intrId);
```

Note that `QuiccIntrOps.unmask` does not guarantee that the interrupt source is unmasked immediately. The real interrupt source unmasking may be deferred.

The `QuiccIntrOps.mask/QuiccIntrOps.unmask` pair may be used at either base or interrupt level. Note that the pair must not be nested.

Enabling/Disabling a serviced interrupt

The `QuiccIntrOps.enable` and `QuiccIntrOps.disable` service routines are dedicated to interrupt handler usage only. In other words, these routines may be called only by an interrupt handler.

The `QuiccIntrOps.enable` service routine enables (and acknowledges) the interrupt source specified by *intrId*:

```
QuiccIntrStatus
    (*enable) (QuiccIntrId intrId);
```

QuiccIntrOps.enable returns either `QUICC_INTR_ACKNOWLEDGED` or `QUICC_INTR_CLAIMED`. The `QUICC_INTR_ACKNOWLEDGED` return value means that the QUICC bus driver has enabled (and acknowledged) the interrupt at bus level. The `QUICC_INTR_CLAIMED` return value means that the QUICC bus driver has ignored the request and therefore the interrupt source is still disabled (and not acknowledged) at bus level.

The bus driver typically refuses an explicit interrupt acknowledge (issued by an interrupt handler) for shared interrupts. In this case, the bus driver will acknowledge interrupts only when all interrupt handlers have been called.

When the `QuiccIntrOps.enable` routine has been called by an interrupt handler, the handler must return the value which has been returned by `QuiccIntrOps.enable`. Once `QuiccIntrOps.enable` is called, the driver should be able to handle an immediate re-entrance in the interrupt handler code.

The `QuiccIntrOps.disable` service routine disables the interrupt source previously enabled by `QuiccIntrOps.enable`:

```
void
    (*disable) (QuiccIntrId intrId);
```

If `QuiccIntrOps.enable` returns `QUICC_INTR_ACKNOWLEDGED`, the driver must call `QuiccIntrOps.disable` prior to returning from the interrupt handler.

When an interrupt occurs, the attached `QuiccIntrHandler` is invoked with the interrupt source disabled at bus level. This behaves in exactly the way as if `QuiccIntrOps.disable` has been called prior to the handler invocation. Note that the interrupt handler must return to the bus driver in the same context as it was called, that is, with the interrupt source disabled at bus level.

On the other hand, the called interrupt handler may use the `QuiccIntrOps.enable/disable` pair to allow the interrupt to be nested. This feature is typically used by a QUICC-to-bus bridge driver when the secondary bus interrupts are multiplexed, that is, multiple secondary bus interrupts are reported through the same primary QUICC bus interrupt. Typically, an interrupt handler of this type of QUICC-to-bus bridge driver would take the following actions:

- identify and disable the secondary bus interrupt source
- enable the primary QUICC bus interrupt source, using `QuiccIntrOps.enable`
- call handlers attached to the secondary bus interrupt source
- disable the primary QUICC bus interrupt source, using `QuiccIntrOps.disable`
- acknowledge (if needed) and enable the secondary bus interrupt source
- return from the interrupt handler

Detaching an attached interrupt

When a driver no longer needs to handle an interrupt, or before closing the connection to the QUICC bus, it should detach the handler attached to an interrupt source.

The `QuiccBusOps.intr_detach` is used to detach a handler, previously attached with `QuiccBusOps.intr_attach`, and to release any resources allocated for this attachement:

```
void
    (*intr\_detach) (QuiccIntrId intrId);
```

The *intrId* argument identifies the attachement to release.

When the `QuiccBusOps.intr_detach` function is called, the driver may no longer use the identifier *intrId*.

QUICC error Handling

When mapping QUICC bus space, a QUICC device driver uses an error handler, invoked by the QUICC bus driver, when a bus error occurs.

Error handlers are used for the following services:

- QuiccBusOps.io_map
- QuiccBusOps.mem_map
- QuiccBusOps.dma_map

When a programmed or direct memory access is aborted because of a bus error, the QUICC bus driver invokes the error handler for the QUICC address and space concerned.

A `QuiccErrHandler` is defined as follows:

```
typedef void (*QuiccErrHandler) (void* errCookie, QuiccBusError* err);
```

The *errCookie* is an opaque value given by the device driver and is passed back when the handler is called.

The *err* argument points to a `QuiccBusError` structure, describing the error as follows:

```
typedef struct QuiccBusError{
        QuiccErrorCode  code;
```

```
        QuiccSize       offset;
    } QuiccBusError;
```

The *code* field indicates the type of error that occurred.

The *offset* field indicates the offset within the associated region at which the error occurred.

The error code can be as follows:

QUICC_ERR_UNKNOWN	unknown error, that is, the QUICC bus driver is unable to determine the reason for the exception
QUICC_ERR_INVALID_SIZE	invalid (not supported) access granularity has been used
QUICC_ERR_PARITY	a parity error was detected on the QUICC busses
QUICC_ERR_EXTERNAL	a bus error was asserted by an external bus slave
QUICC_ERR_INTERNAL	an error occurred in a transaction to internal memory (Dual Port RAM or internal registers)
QUICC_ERR_TIMEOUT	a bus transaction was timed out

As a bus error may be reported in the context of an exception or an interrupt, the implementation of the error handler must be restricted to the interrupt level API.

Sending commands to Communication Processor

A QUICC device driver can send a command to the Communication Processor, for a given internal controller channel, by calling the `QuiccBusOps.cmd_send` service:

```
KnError
   (*cmd_send) (QuiccDevId        devId,
                QuiccPropChannel* channel,
                QuiccSubChannel   subChannel,
                QuiccCommand      cmd);
```

The *devId* argument identifies the connection with the QUICC bus driver.

The *channel* argument indicates the QUICC internal controller channel to which the command will be applied. Typically, a QUICC device driver should find the channel to use by looking at the "channel" property in the device node. The value of this type of property is a *QuiccPropChannel* enumeration value in:

- QUICC_SMC1
- QUICC_DSP_R
- QUICC_SMC2
- QUICC_DSP_T
- QUICC_SCC1
- QUICC_SCC2

- QUICC_SCC3
- QUICC_SCC4
- QUICC_RISC_TIMER
- QUICC_MCC1
- QUICC_MCC2
- QUICC_FCC1
- QUICC_FCC1_HDLC
- QUICC_FCC1_ATM
- QUICC_FCC1_ETHER
- QUICC_FCC2
- QUICC_FCC2_HDLC
- QUICC_FCC2_ATM
- QUICC_FCC2_ETHER
- QUICC_FCC3
- QUICC_FCC3_HDLC
- QUICC_FCC3_ATM
- QUICC_FCC3_ETHER

The *subChannel* argument indicates the sub-channel for a Multi-channel controller. Otherwise, the argument should be zero.

The *cmd* argument indicates the QUICC command to execute and should be in:

`QUICC_INIT`	initialize receive and transmit parameters
`QUICC_INIT_RX`	initialize receive parameters
`QUICC_INIT_TX`	initialize transmit parameters
`QUICC_HUNT_MODE`	enter hunt mode
`QUICC_ABORT_TX`	stop transmit when transmit fifo is empty
`QUICC_STOP_TX`	stop transmit after current frame is sent
`QUICC_RESTART_TX`	restart transmit after stopped
`QUICC_CLOSE_RX_BD`	make receive buffer descriptor available to CPU
`QUICC_SET_TIMER`	risc (CP) timers management
`QUICC_SET_GROUP_ADDR`	ethernet group address management
`QUICC_GCI_TIMEOUT`	GCI performs timeout functions
`QUICC_STOP_RX`	stop receive after current frame

QUICC_RESET_BCS	reset block check sequence
QUICC_GCI_ABORT_REQ	GCI receiver sends abort
QUICC_ATM_TRANSMIT	activate an ATM channel
QUICC_START_DSP	start current DSP chain
QUICC_INIT_DSP	initialize DSP chain

On success, K_OK is returned.

On failure, QuiccBusOps.cmd_send returns an error code as follows:

K_ETIMEOUT	CPU was not able to execute the command in the maximum time required to execute any of the available commands
K_EINVAL	invalid *channel*, or *command* is invalid for the given channel

Configuring Baud-Rate-Generator frequency

A QUICC device driver can configure and enable a Baud-Rate-Generator to work at a given frequency by calling the QuiccBusOps.brg_conf service:

```
KnError
   (*brg_conf) (QuiccDevId    devId,
                QuiccPropBrg* brg,
                QuiccFreq     freq);
```

The *devId* argument identifies the connection with the QUICC bus driver.

The *brg* argument indicates which QUICC bus Baud-Rate-Generator to use. Typically, a QUICC device driver should select BRG by looking at the "brg" property in the device node. The value of such a property is a QuiccPropBrg BRG number. The available BRG numbers are specific to each QUICC bus driver implementation and depend on the underlying QUICC micro-controller.

The *freq* argument indicates the frequency, in Hertz, required for the clock, delivered by the BRG being configured.

On success, K_OK is returned.

On failure, QuiccBusOps.brg_confreturns one of the following error codes:

K_EINVAL	the *brg* argument is invalid
K_EFAIL	the specified *brg* is not able to work at the required frequency

Accessing general purpose I/O port pins

To configure a parallel I/O port pin as a general purpose I/O pin, a QUICC driver should:

- attach to the I/O pin
- use services defined for the attached I/O pin

- detach from the I/O pin when no longer needed

Attaching to a general purpose I/O port pin

A QUICC device driver can attach to a general purpose I/O port pin by calling the `QuiccBusOps.io_pin_attach` service:

```
KnError
   (*io_pin_attach) (QuiccDevId      devId,
                     QuiccPropIoPin* ioPin,
                     QuiccIoPinOps** ioPinOps,
                     QuiccIoPinId*   ioPinId);
```

The *devId* argument identifies the connection with the QUICC bus driver.

The *ioPin* argument indicates the QUICC general purpose I/O pin to attach to. Typically, a QUICC device driver should find the I/O pin to attach to by looking up the "io-pin" property in the device node. The value of such a property is a `QuiccPropIoPin` structure, shown below:

```
typedef struct QuiccPropIoPin {
       QuiccIoPin    pin;
       QuiccIoPort   port;
       Bool          intrUsed;
       QuiccPropIntr intr;
   } QuiccPropIoPin;
```

The *port* field indicates the parallel I/O port and should be in:

- `QUICC_IO_PORT_A`
- `QUICC_IO_PORT_B`
- `QUICC_IO_PORT_C`
- `QUICC_IO_PORT_D`

The *pin* field is a bit field specifying the *port* pin to attach to. Only one bit among the bit fields should be set, indicating the I/O pin to use. This field may be zero (no bit set) to indicate that the external signal logically associated to that pin is not available (physically not connected).

The *intrUsed* argument indicates whether a QUICC interrupt may be triggered on the I/O pin value changes. If *intrUsed* is `TRUE` then the *intr* field defines the interrupt source used to attach a handler to the interrupt. (See section "*Attaching a handler to a QUICC interrupt source*".)

On success, *K_OK* is returned and services defined on an attached I/O pin object are returned in the *ioPinOps* argument. An identifier for the attached interrupt is also returned in *ioPinId*. This identifier must be used as a first argument to further calls to the `QuiccIoPinOps` services.

Getting general purpose I/O port pin state

To read an attached I/O pin state, a driver can call `QuiccIoPinOps.get`:

```
uint32_f
    (*get)  (QuiccIoPinId pinId);
```

The *ioPinId* argument identifies the I/O port pin to read.

`QuiccIoPinOps.get` returns the current state of the I/O port pin (0/1).

Setting the general purpose I/O port pin

A driver can set an attached I/O port pin to high level (1) by calling `QuiccIoPinOps.set`:

```
void
    (*set)  (QuiccIoPinId pinId);
```

The *ioPinId* argument identifies the I/O port pin to set.

Clearing the general purpose I/O port pin

A driver can set an attached I/O port pin to low level (1) by calling `QuiccIoPinOps.clear`:

```
 void
    (*clear)  (QuiccIoPinId pinId);
```

The *ioPinId* argument identifies the I/O port pin to clear.

Detaching from a general purpose I/O port pin

When a driver no longer needs to use an I/O port pin, or before closing the connection to the QUICC bus, it should detach from the attached pin.

The `QuiccBusOps.io_pin_detach` is used to release a pin, previously attached with `QuiccBusOps.io_pin_attach`, and to release any resources allocated for this attachement:

```
void
    (*io\_pin\_detach) (QuiccIoPinId ioPinId);
```

The `ioPinId` argument identifies the attachment to release.

When the `QuiccBusOps.io_pin_detach` function is called, the driver is no longer able to use the identifier *ioPinId*.

QUICC specific node properties

The following table lists the QUICC specific node properties.

The *alias* column specifies the alias name which should be used by a QUICC bus or device driver referencing the property name. The *name* column specifies the property name ASCII string. The *value* column specifies the type of the property value. The *bus* column specifies properties used by a QUICC bus node. The *dev* column specifies properties used by a QUICC device node. The symbol *m* in the *bus* and *dev* columns flags mandatory properties. The symbol *o* in the *bus* and *dev* columns flags optional properties. The symbol - in the *bus* and *dev* columns flags properties which are not applied to the given node type.

Table 1 QUICC specific properties

alias	name	value	bus	dev
QUICC_PROP_INTR	"intr"	QuiccPropIntr[]	—	o
QUICC_PROP_IO_REGS	"io-regs"	QuiccPropIoRegs[]	—	o

QUICC_PROP_MEM_RGN	"mem-rgn"	QuiccPropMemRgn[]	—	o
QUICC_PROP_CHANNEL	"channel"	QuiccPropChannel	—	o
QUICC_PROP_BRG	"brg"	QuiccPropBrg	—	o
QUICC_PROP_IO_PIN	"io-pin"	QuiccPropIoPin[]	—	o
QUICC_PROP_DMA_BURST	"dma-burst"	QuiccPropDmaBurst	m	—
QUICC_PROP_DMA_MIN_SIZE	"dma-min-size"	QuiccPropDmaMinSize	m	—
QUICC_PROP_BYTE_ORDER	"byte-order"	QuiccPropByteOrder	m	—
QUICC_PROP_IMAP_RGN	"imap-rgn"	QuiccPropMemRgn	m	—
QUICC_PROP_DPRAM_RGN	"dpram-rgn"	QuiccPropMemRgn	m	—
QUICC_PROP_CLOCK_FREQ	"clock-freq"	QuiccPropClockFreq	m	—
QUICC_PROP_BRG_CLOCK_FREQ	"brg-clock-freq"	QuiccPropClockFreq	m	—
QUICC_PROP_CPM_CLOCK_FREQ	"cpm-clock-freq"	QuiccPropClockFreq	m	—

When the value of a property is an array (for example, `QuiccPropIntr[]`), the size of the array is the size of the property value returned by `dtreePropLength`. The size of the array divided by the size of its element type (for exmample, `sizeof(QuiccPropIntr)`) defines the number of elements in the array. Determining this allows a QUICC device driver to iterate through the array in order to perform an action for each element (for instance, to attach an interrupt handler to each device interrupt line).

Dynamic resource allocation

The QUICC bus driver may support the dynamic allocation of QUICC bus resources. When dynamic resource allocation is supported, a QUICC device driver may use the `QuiccBusOps.resource_alloc` service routine to allocate a bus resource at run time:

```
KnError
    (*resource_alloc) (QuiccDevId devId, DevProperty prop);
```

devId is returned by `QuiccBusOps.open`

prop specifies the bus resource being allocated

The `QuiccBusOps.resource_alloc` service routine allocates a given bus resource and, if the allocation request is satisfied, updates the device node properties in order to add the newly allocated bus resource.

On success, `QuiccBusOps.resource_alloc` returns *K_OK*.

On failure, one of the following error codes is returned:

K_ENOTIMP	dynamic resource allocation is not supported by the bus driver
K_EUNKNOWN	the property name is unknown
K_EINVAL	the property value is invalid
K_EFAIL	the bus resource is not available
K_ENOMEM	the system is out of memory

When a dynamically allocated bus resource is no longer used, a QUICC device driver may release it by calling the `QuiccBusOps.resource_free` service routine:

```
void
    (*resource_free) (QuiccDevId devId, DevProperty prop);
```

devId is returned by `QuiccBusOps.open`

prop specifies the bus resource being released

The `QuiccBusOps.resource_free` service routine releases a given bus resource and updates the device node properties accordingly.

The following bus resource properties may be dynamically allocated and released:

- QUICC_PROP_INTR
- QUICC_PROP_IO_REGS
- QUICC_PROP_MEM_RGN
- QUICC_PROP_BRG

Note that the `QuiccBusOps.resource_alloc` and `QuiccBusOps.resource_free` routines should not be used by a simple device driver. This type of a driver should assume that all needed resources are already allocated and specified as properties in the device node by the bus driver. The driver should only find the relevant property and call an appropriate service routine, passing a pointer to the property value.

Occassionally, it is not possible to determine all of the required resources prior to device initialization. An example is a bus-to-bus bridge driver which can discover devices residing on the secondary bus only when the bus bridge hardware is already initialized. In this case, when `drv_init` is called, the bus-to-bus bridge node would contain only the resources needed for the bus-to-bus bridge device itself (the internal bus-to-bus bridge registers). Once the devices residing on the secondary bus are discovered, the bus-to-bus bridge driver would request additional primary bus resources in order to satisfy the resource requirements for the devices. Another example is a bus which supports hot-pluggable devices. On this type of bus, the primary bus resources allocated

by the hot-pluggable bus driver depend on the devices currently plugged into the secondary bus. Usually, the resource requirements are changed when a hot-plug insertion or removal occurs.

Note that dynamic resource allocation might also be used by a device driver which implements a lazy resource allocation. In this type of driver, the bus resource allocation might be performed at open time using `QuiccBusOps.resource_alloc`. Then, the dynamically allocated resources might be released at close time using `QuiccBusOps.resource_free`.

QUICC bus interface allowed calling contexts

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

Table 2 QUICC bus invocation context

Services	Base Level	DKI thread	Intr	Blocking
QuiccBusOps.open	—	+	—	+
QuiccBusOps.close	—	+	—	+
QuiccBusOps.intr_attach	—	+	—	+
QuiccBusOps.intr_detach	—	+	—	+
QuiccBusOps.io_pin_attach	—	+	—	+
QuiccBusOps.io_pin_detach	—	+	—	+
QuiccBusOps.io_map	—	+	—	+
QuiccBusOps.io_unmap	—	+	—	+
QuiccBusOps.mem_map	—	+	—	+
QuiccBusOps.mem_unmap	—	+	—	+
QuiccBusOps.dma_alloc	—	+	—	+
QuiccBusOps.dma_free	—	+	—	+
QuiccBusOps.resource_alloc	—	+	—	+
QuiccBusOps.resource_free	—	+	—	+
QuiccBusOps.cmd_send	+	+	+	—
QuiccBusOps.brg_conf	+	+	+	—
QuiccIoPinOps.get	+	+	+	—
QuiccIoPinOps.set	+	+	+	—
QuiccIoPinOps.clear	+	+	+	—
QuiccIoOps.load_xx	+	+	+	—

QuiccIoOps.store_xx	+	+	+	—
QuiccIoOps.read_xx	+	+	+	—
QuiccIoOps.write_xx	+	+	+	—
QuiccDmaOps.phys_addr	+	+	+	—
QuiccDmaOps.virt_addr	+	+	+	—
QuiccDmaOps.read_sync	+	+	+	—
QuiccDmaOps.write_sync	+	+	+	—
QuiccIntrOps.mask	+	+	+	—
QuiccIntrOps.unmask	+	+	+	—
QuiccIntrOps.enable	+	+	+	—
QuiccIntrOps.disable	+	+	+	—

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`dtreeNodeRoot`(9DKI), `svDriverRegister`(9DKI), `svMemAlloc`(9DKI), `svPhysAlloc`(9DKI), `svPhysMap`(9DKI), `svDkiThreadCall`(9DKI), `svTimeoutSet`(9DKI), `usecBusyWait`(9DKI), `DISABLE_PREEMPT`(9DKI)

NAME

ric – RIC Device Driver Interface

SYNOPSIS

```
#include <ddi/ric/ric.h>
```

FEATURES

DDI

DESCRIPTION

Provides RIC device driver services.

EXTENDED DESCRIPTION

The RIC device driver service routines are declared by the RicDevOps structure.

```
typedef struct RicDevOps {

            RicVersion version;

                KnError
            (*intr_attach) (RicDevId            dev_id,
                            RicInt              dev_intr,
                            RicIntrHandler      dev_intr_handler,
                            void*               dev_intr_cookie,
                            RicIntrOps**        ric_intr_ops,
                            RicIntrId*          ric_intr_id);

                void
            (*intr_detach) (RicIntrId           ric_intr_id);

        } RicDevOps;
```

A pointer to the `RicDevOps` structure is exported by the driver via the `svDeviceRegister` microkernel call. A driver client invokes the `svDeviceLookup` and `svDeviceEntry` microkernel calls in order to obtain a pointer to the device service routines vector. Once the pointer is obtained, the driver client is able to invoke the driver service routines (via the indirect function call) in order to attach / detach interrupts.

`intr_attach` establishes a connection between a driver client and a given device driver instance.

The `intr_attach` routine is called with the following arguments:

dev_id	RIC device identifier.
dev_intr	RIC interrupt number offset.
dev_intr_handler	Handler in the device driver which is invoked by the RIC driver when the corresponding interrupt occurs.
dev_intr_cookie	Cookie passed to the interrupt handler.
ric_intr_ops	Points to the `RicIntrOps` structure which specifies the client call-back handlers (see section: Up-Call Handlers).
ric_intr_id	RIC interrupt identifier returned.

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 *ric_intr_id*. This identifier must be used as first argument to subsequent calls to *ric_intr_ops* services. The corresponding interrupt is enabled.

On failure, an error code is returned as follows:

K_EINVAL	The specified RIC interrupt offset is invalid.
K_BUSY	Another handler is already attached to the given interrupt .
K_ENOMEM	The system is out of memory.

The `intr_detach` routine disconnects a driver client using the following arguement:

ric_intr_id	RIC interrupt identifier.

The corresponding interrupt is disabled.

Up-Call Handlers

The driver client up-call handlers are specified by the `RicIntrOps` structure and given to the device driver as an argument of *intr_attach*.

```
typedef struct RicIntrOps {

                void
            (*mask) (RicIntrId ric_intr_id);

                void
            (*unmask) (RicIntrId ric_intr_id);

                void
            (*enable) (RicIntrId ric_intr_id);

                void
            (*disable) (RicIntrId ric_intr_id);

        } RicIntrOps;
```

Mask/Unmask an Attached Interrupt

The `RicIntrOps.mask` service routine masks the interrupt source specified by `ric_intr_id`.

The `RicIntrOps.unmask` service routine unmasks the interrupt source previously masked by `RicIntrOps.mask`.

The `RicIntrOps.mask` / `RicIntrOps.unmask` pair may be used at either base or interrupt level.

Enable/Disable a Serviced Interrupt

The `RicIntrOps.enable` and `RicIntrOps.disable` service routines are dedicated to interrupt handler usage only. These routines may only be called by an interrupt handler.

The `RicIntrOps.enable` service routine enables the interrupt source specified by ric_intr_id.

The `RicIntrOps.enable` service routine diabless the interrupt source specified by ric_intr_id.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

rtc – RTC Device Driver Interface

SYNOPSIS

```
#include <ddi/rtc/rtc.h>
```

FEATURES

DDI

DESCRIPTION

The Real Time Clock (RTC) interface is used to get / set the current date stored in non-volatile memory, generally powered by a battery.

EXTENDED DESCRIPTION

Service Routines

The RTC device driver service routines are declared by the `RtcDevOps` structure.

```
typedef struct {
    RtcVersion version;

        KnError
    (*set_date) (RtcId    device_id,
                 RtcDate* rtc_date);

        KnError
    (*get_date) (RtcId    device_id,
                 RtcDate* rtc_date);

} RtcDevOps;
```

A pointer to the `RtcDevOps` structure is exported by a driver via the `svDeviceRegister` microkernel call. A driver client invokes the `svDeviceLookup` and `svDeviceEntry` microkernel calls in order to obtain a pointer to the device service routines vector. Once the pointer is obtained, the driver client is able to invoke the driver service routines (via the indirect function call) in order to read the current date (`get_date` function), or write it (`set_date` function).

`set_date`

The `set_date` function updates the current date stored in the RTC device specified by *device_id*.

Arguments:

device_id	Specifies the RTC device identifier (returned by `svDeviceEntry`).
rtc_date	Specifies the memory location containing the new date to be written.

The `set_date` function returns the following results:

`K_OK`	The operation succeeded.

K_EFAIL	The operation has failed, the RTC device is busy.
K_EINVAL	The *rtc_date* is invalid.

get_date

The get_date function reads the current date stored in the RTC device specified by *device_id*, and stores it at the memory location pointed to by *rtc_date*.

Arguments:

device_id	Specifies the RTC device identifier.
rtc_date	Specifies the memory location to be filled with the current date value.

The get_date function returns the following results:

K_OK	The operation succeeded.
K_EFAIL	The RTC device has failed to read the current date or *rtc_date* is null.

Both the set_date and get_date functions must be called from base level.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

svDeviceRegister(9DKI), svDriverRegister(9DKI), svDeviceLookup(9DKI), svDeviceEntry(9DKI), svDeviceRelease(9DKI)

NAME

timer – TIMER Device Driver Interface

SYNOPSIS

```
#include <ddi/timer/timer.h>
```

FEATURES

DDI

DESCRIPTION

The timer is an incremental counter. There are wo modes of operation available: free-running and periodic timer. In free-running mode the timer will overflow after reaching its maximum value and continue to count up from its minimum value. In periodic timer mode the timer will generate an interrupt at a constant time interval, and the client handler provided can be called at each interval expiration.

Service Routines

The TIMER device driver service routines are declared by the `TimerDevOps` structure.

```
typedef struct {
    TimerVersion version;

        KnError
    (*start_freerun) (TimerId    device_id,
                      KnTimeVal* min_period);

        KnError
    (*start_periodic) (TimerId          device_id,
                       KnTimeVal*       tick_interval,
                       TimerTickHandler tick_handler,
                       void*            tick_cookie);

        void
    (*mask) (TimerId device_id);

        void
    (*unmask) (TimerId device_id);

        unsigned long
    (*get_counter_frequency) (TimerId device_id);

        unsigned long
    (*get_counter_period) (TimerId device_id);

        unsigned long
    (*read_counter) (TimerId device_id);

        void
    (*stop) (TimerId device_id);

} TimerDevOps;
```

A pointer to the `TimerDevOps` structure is exported by the driver via the `svDeviceRegister` microkernel call. A driver client invokes the `svDeviceLookup` and `svDeviceEntry` microkernel calls in order to obtain a pointer to the device service routines vector. Once the pointer is obtained,

the driver client is able to invoke the driver service routines (via the indirect function call) in order to start / stop the device, get counter frequency and period, and read the current counter value.

`start_freerun`

`start_freerun` starts the specified timer device in a freerun mode. In this mode, the timer continuously counts without interrupt generation. The client can then use `read_counter` to read the counter's current value.

Arguments:

device_id	Specifies the TIMER device identifier (returned by `svDeviceEntry`).
min_period	Specifies the minimum required period of the timer. This specifies the minimum interval during which no counter reload should occur.

The `start_freerun` function returns the following results:

`K_OK`	TIMER is successfully started.
`K_EFAIL`	TIMER cannot be initialized within the given period.
`K_EBUSY`	TIMER is already open.
`K_ENOTIMP`	The `start_freerun` is not implemented for this device.

Once started, the timer device must be stopped with the `stop` function (see Section `stop`) before being started again.

`start_periodic`

The `start_periodic` function starts the specified timer device in a periodic mode, where the client provided handler is called at periodic intervals (specified in the *tick_interval* parameter).

Arguments:

device_id	Specifies the TIMER device identifier.
tick_interval	Specifies the interval between two calls of the *tick_handler* handler.
tick_handler	Specifies the routine to call when the specified interval expires.

tick_cookie Specifies a parameter passed back to the handler specified..

The `start_periodic` function returns the following results:

`K_OK`	TIMER has been started successfully.
`K_EFAIL`	TIMER cannot be initialized with the given interval.
`K_EBUSY`	TIMER is already opened.
`K_ENOTIMP`	The `start_periodic` is not implemented for this device.

Once the timer has been started, it should be stopped using the `stop` function prior to being started again. The timer is started with the interrupt disabled. The client should explicitly enable interrupts with the `unmask` function.

`mask`

The `mask` function disables the timer interrupts. Once interrupts have been disabled, the client handler given in the *tick_handler* parameter of the `start_periodic` function will not be called until timer interrupts are enabled again using the `unmask` function.

Arguments:

device_id Specifies the TIMER device identifier.

The `mask` routine may be called from interrupt level.

`unmask`

The `unmask` function enables timer interrupts allowing the provided *tick_handler* to be called at specified periodic intervals.

Arguments:

device_id Specifies the TIMER device identifier.

The `unmask` routine may be called from interrupt level. The `mask`/`unmask` pairs must not be nested.

`get_counter_frequency`

The `get_counter_frequency` function returns the frequency in hertz of the given timer device.

Arguments:

device_id	Specifies the TIMER device identifier.

`get_counter_period`

The `get_counter_period` function returns the difference between the maximum and minimum values of the timer counter.

Arguments:

device_id	Specifies the TIMER device identifier.

`read_counter`

The `read_counter` function returns the current value of the timer counter. The read value can be used to measure the time elapsed between two calls to the `read_counter` function.

Arguments:

device_id	Specifies the TIMER device identifier.

The `read_counter` must be used after the timer has been started in *free_run* mode. The read values are guaranteed to be incremental if the `read_counter` function is executed before the *min_period* expires.

`stop`

The `stop` function stops the timer specified.

Arguments:

device_id	Specifies the TIMER device identifier.

Once the timer device has been stopped, the `read_counter` function must no longer be called.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`svDeviceRegister`(9DKI), `svDriverRegister`(9DKI), `svDeviceLookup`(9DKI), `svDeviceEntry`(9DKI), `svDeviceRelease`(9DKI)

NAME

uart – UART Device Driver Interface

SYNOPSIS

```
#include <ddi/uart/uart.h>
```

FEATURES

DDI

DESCRIPTION

Provides UART device driver services.

EXTENDED DESCRIPTION

Service Routines

The UART device driver service routines are declared by the `UartDevOps` structure.

```
typedef struct {
    UartVersion version;

        KnError
    (*open) (UartId        device_id,
             UartConfig*   device_cfg,
             void*         client_cookie,
             UartCallBack* client_ops,
             unsigned int* signals);

        void
    (*mask) (UartId device_id);

        void
    (*unmask) (UartId device_id);

        void
    (*transmit) (UartId         device_id,
                 unsigned char* buffer,
                 unsigned int   size);

        void
    (*abort) (UartId device_id);

        void
    (*txbreak) (UartId device_id);

        void
    (*control) (UartId       device_id,
                unsigned int signals);

        void
    (*rxbuffer) (UartId         device_id,
                 unsigned char* buffer,
                 unsigned int   size);

        void
    (*close) (UartId device_id);
} UartDevOps;
```

A pointer to the `UartDevOps` structure is exported by a driver via the `svDeviceRegister` microkernel call. A driver client invokes the `svDeviceLookup` and `svDeviceEntry` microkernel calls in order to obtain a pointer to the device service routines vector. Once the pointer is obtained, the driver client is able to invoke the driver service routines (via the indirect function call) in order to open / close devices, enable / disable device interrupts, start the data / break transmission, and raise / drop the modem control signals.

`open` establishes a connection between a driver client and a given device driver instance.

Arguments:

device_id	Specifies the UART device identifier.
device_cfg	Points to the `UartConfig` structure which specifies the serial line configuration (see below) .
client_cookie	Points to the client handle. The client handle is passed back to the client (as the first argument) each time a call-back handler is invoked.
client_ops	Points to the `UartCallBack` structure which specifies the client call-back handlers (see section: Up-Call Handlers).
signals	Points to a location where the initial modem status is returned.

The `UartConfig` structure is the following:

```
typedef struct {
       unsigned int        baudRate;
       unsigned char       dataBits;
       unsigned char       stopBits;
       unsigned char       parity;
       unsigned char       fifoTrigger;
} UartConfig;
```

The *baudRate* field specifies the input and output baud rate as an integer value (for example, `9600`). The *dataBits* field specifies the number of data bits per character.

The following constants are allowed:

UART_CFG_DATABITS_5	5 data bits
UART_CFG_DATABITS_6	6 data bits
UART_CFG_DATABITS_7	7 data bits

UART_CFG_DATABITS_8	8 data bits

The *stopBits* field specifies the number of stop bits. The following constants are allowed:

UART_CFG_STOPBITS_1	1 stop bit
UART_CFG_STOPBITS_1_5	1.5 stop bits
UART_CFG_STOPBITS_2	2 stop bits

The *parity* field specifies the parity bit. The following constants are allowed:

UART_CFG_PARITY_NONE	No parity bit
UART_CFG_PARITY_ODD	Odd parity bit
UART_CFG_PARITY_EVEN	Even parity bit
UART_CFG_PARITY_MARK	Mark (1) parity bit
UART_CFG_PARITY_SPACE	Space (0) parity bit

The *fifoTrigger* field specifies the programmable fifo trigger level. It is an integer value which provides an indication to the driver as to how to configure the trigger level of the receive FIFO. When *fifoTrigger* is zero, the FIFO (if any) is disabled. When the programmable fifo trigger is not available on hardware, this field is ignored by the device driver.

Note that only the generic UART options can be dynamically configured via open. All other device-specific options can only be configured statically through the device node properties.

open keeps the modem control signals (data-set-ready and request-to-send) unchanged (that is, does not drop down the signals), and returns the initial state of modem status signals as the bit-mask of the following flags:

UART_SIG_RX_CTS_UP / UART_SIG_RX_CTS_DOWN
: The clear-to-send modem status has been raised / dropped.

UART_SIG_RX_DSR_UP / UART_SIG_RX_DSR_DOWN
: The data-set-ready modem status has been raised / dropped.

UART_SIG_RX_DCD_UP / UART_SIG_RX_DCD_DOWN
: The data-carrier-detect modem status has been raised / dropped.

UART_SIG_RX_RI_UP / UART_SIG_RX_RI_DOWN
: The ring-indicator modem status has been raised / dropped.

When a particular modem status is not supported by the UART device, neither the "UP" nor "DOWN" flag is returned. The open function returns the following results:

K_OK	The UART line has been iniatlized successfully.
K_EFAIL	The UART cannot be initialized with the configuration provided (for example the baud rate specified is not supported).

open must be the first call issued to the device driver. When K_OK is returned, open puts the device into the interrupt masked state. In order to start the device (that is, to enable interrupts), the unmask down-call must be invoked.

mask

The mask function disables all device interrupts (input, output and special).

Arguments:

device_id	Specifies the UART device identifier.

Note that it is up to the client to disable preemption prior to the mask invocation in order to prevent the current thread being preempted while device interrupts are masked.

unmask

The unmask function enables all device interrupts (input, output and special).

Arguments:

device_id	Specifies the UART device identifier.

Typically, the mask / unmask pair would be used by the driver client to implement a critical section of code which must be synchronized with the call-back handlers. The mask / unmask pair must not be called by the call-back handlers. This is unnecessary because the interrupts are already masked when the call-back handlers are invoked. The mask / unmask pairs must not be nested.

transmit

The transmit function starts the given buffer transmission.

Arguments:

device_id	Specifies the UART device identifier.
buffer	Points to the first byte of the output buffer.

size Specifies the output buffer size.

It returns immediately without waiting for the buffer transmission to be completed. The driver client is notified about the end of transmission via the `txdone` call-back handler. The driver client must not issue another `transmit` or `txbreak` request until the `txdone` call-back handler is invoked by the device driver. The `transmit` down-call must be issued in the masked state. In other words, it must be called from either a critical section protected by `mask` / `unmask` or a call-back handler.

`abort`

The `abort` function allows the driver client to interrupt the transmission in progress.

Arguments:

device_id Specifies the UART device identifier.

If there is an output in progress, the device driver interrupts (aborts) it and invokes the `txdone` call-back handler with the `UART_SIG_TX_ABORTED` flag set. Typically, the `abort` service routine would be used by the client to implement flow control based on the CTS/RTS modem signals as well as on the XON/XOFF protocol. The `abort` down-call must be issued in the masked state. In other words, it must be called from either a critical section protected by `mask` / `unmask` or a call-back handler.

`txbreak`

The `txbreak` function starts the BREAK signal transmission.

Arguments:

device_id Specifies the UART device identifier.

It returns immediately without waiting for the BREAK signal to be sent. The driver client is notified about the end of transmission by the `txdone` call-back handler with the `UART_SIG_TX_BREAK` flag set in the *signals* argument. The driver client must not issue another `transmit` or `txbreak` request until the `txdone` call-back handler is invoked by the device driver. The `txbreak` down-call must be issued in the masked state. In other words, it must be called from either a critical section protected by `mask` / `unmask` or a call-back handler.

`control`

The `control` function allows the driver client to raise / drop the modem control signals.

Arguments:

device_id	Specifies the UART device identifier.
signals	Specifies the modem control signals to raise / drop.

A combination of the following flags may be specified:

UART_SIG_TX_DTR_UP / UART_SIG_TX_DTR_DOWN
Raise / drop the data-terminal-ready modem control signal

UART_SIG_TX_RTS_UP / UART_SIG_TX_RTS_DOWN
Raise / drop the request-to-send modem control signal

When both (`"UP"` and `"DOWN"`) flags are set for the same modem control signal, the result is undefined. The `control` down-call must be issued in the masked state. In other words, it must be called from either a critical section protected by `mask` / `unmask` or a call-back handler.

`rxbuffer`

The `rxbuffer` function allows the driver client to specify the buffer for received characters.

Arguments:

device_id	Specifies the UART device identifier.
buffer	Points to the first byte of the receive buffer.
size	Specifies the receive buffer size.

The driver handles a free position pointer inside the buffer. The pointer is initialized by `rxbuffer` and updated each time a character is put into the buffer. Typically, the driver invokes the `receive` call-back handler when the receive FIFO becomes empty. The *count* argument is passed to the `receive` handler specifying the number of received characters since the previous receive invocation.

When the receive buffer becomes full (that is, the last available position in the buffer has just been used) the `receive` call-back handler is invoked immediately with the `UART_SIG_RX_BUF_FULL` flag set in the *signals* argument. When a received character cannot be put into the buffer (there is no room) the `receive` call-back handler is invoked with the `UART_SIG_RX_BUF_OVERRUN` flag set in the *signals* argument.

When the receive buffer is not set at all or set to zero, the device driver ignores all input characters and does not invoke the `receive` handler when a character is received. The `rxbuffer` down-call must be issued in the masked state. In other words, it must be called from either a critical section protected by `mask` / `unmask` or a call-back handler.

`close`

The `close` down-call releases the serial line device and puts it in a hung state.

Arguments:

device_id	Specifies the UART device identifier.

The `close` function must be called in the masked state, that is, the `mask` down-call must be issued before the `close` one.

`close` must be the last call issued to the driver. In particular, the driver client must not invoke `unmask` after returning from `close`.

Up-Call Handlers

The driver client up-call handlers are specified by the `UartCallBack` structure and given to the device driver as an argument of `open`.

```
typedef struct {
        void
    (*txdone) (void*        client_cookie,
               unsigned int count,
               unsigned int signals);

        void
    (*receive) (void*          client_cookie,
                unsigned int   count,
                unsigned int   signals);
} UartCallBack;
```

`txdone`

The `txdone` up-call notifies the driver client that the requested transmission (`transmit` or `txbreak`) is finished.

Arguments:

client_cookie	Is the client handle specified in `open`.
count	Specifies the number of transmitted characters.
signals	Specifies whether or not the transmission has been aborted.

In the case of data transmission (`transmit`), the `txdone` up-call notifies the driver client that output is completed or has been interrupted by the `abort` down-call.

In the case of `abort`, the `UART_SIG_TX_ABORTED` flag is set in the *signals* argument. In both cases, the *count* argument specifies the number of transmitted characters.

In the case of a `BREAK` signal transmission (`txbreak`), the `txdone` up-call notifies the driver client that the `BREAK` signal has been sent. The `UART_SIG_TX_BREAK` flag is set in the *signals* argument. The `txdone` up-call makes it possible to issue another request (`transmit` or `txbreak`) to the device driver.

The `txdone` up-call handler is invoked in the masked state. Typically, it is called from interrupt level.

`receive`

The `receive` up-call notifies the driver client that *count* number of characters have been written to the receive buffer or / and some *signals* have occurred.

Arguments:

client_cookie	Is the client handle specified in open.
count	Specifies the number of received characters.
signals	Specifies the received signals and / or error conditions.

The following signals may be delivered by receive :

UART_SIG_RX_PARITY_ERROR
The last character in the buffer was received with a parity error.

UART_SIG_RX_FRAMING_ERROR
The last character in the buffer was received with a framing error.

UART_SIG_RX_FIFO_OVERFLOW
Characters have been lost because of a FIFO overflow.

UART_SIG_RX_BUF_FULL
The receive buffer is full.

UART_SIG_RX_BUF_OVERRUN
Characters have been lost because of the receive buffer overflow.

UART_SIG_RX_BREAK
A BREAK signal has been received.

UART_SIG_RX_CTS_UP / UART_SIG_RX_CTS_DOWN
The clear-to-send modem status has been raised / dropped.

UART_SIG_RX_DSR_UP / UART_SIG_RX_DSR_DOWN
The data-set-ready modem status has been raised / dropped.

UART_SIG_RX_DCD_UP / UART_SIG_RX_DCD_
The data-carrier-detect modem status has been raised / dropped.

UART_SIG_RX_RI_UP / UART_SIG_RX_RI_DOWN
The ring-indicator modem status has been raised / dropped.

The `receive` up-call handler is invoked in the masked state. Typically, it is called from interrupt level.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`dtreeNodeRoot`(9DKI), `svDeviceRegister`(9DKI), `svDriverRegister`(9DKI), `svMemAlloc`(9DKI), `svTimeoutSet`(9DKI), `usecBusyWait`(9DKI), `DISABLE_PREEMPT`(9DKI)

NAME

vme – VME Bus Driver Interface

SYNOPSIS

```
#include <ddi/vme/vme.h>
```

FEATURES

DDI

DESCRIPTION

Provides VME bus driver interface services.

EXTENDED DESCRIPTION

The VME bus driver offers an API for VME device driver development. This VME bus API is an abstraction of the low-level VME bus services and covers the following VME functional modules:

Master
- Access to slaves' I/O registers.
- Access to slaves' memory.

Slave
- Exporting memory to other bus Masters.

Interrupt handler

Interrupter

The following bus options are covered by the API when available:

- A16/A24/A32 address spaces.
- D8/D16/D32 data sizes for I/O access.
- D8/D16/D32/D64 data sizes and block transfers for memory access.
- Read Modify Write cycles.
- Bus arbiter, requester, time-out configuration through properties..

Connecting a Device Driver to the VME bus

First of all, a VME device driver must register itself in the kernel driver registry. This should be done from its main function using `svDriverRegister`. The driver must set the bus class to `"vme"` in its registry entry, and specify the lowest version number of VME bus interface required to run correctly. This registration allows the kernel to call back the VME device driver's `drv_probe`, `drv_bind` and `drv_init` routines at bus probing, binding and initialization phases respectively.

Within the `drv_probe` or `drv_init` routine, the VME device driver may establish a connection with the parent VME bus driver, as described below. Once a connection is established, the VME device driver may use services provided by the VME bus driver. Note that a connection to the bus driver can be established only within the driver's `drv_probe` or `drv_init` routine.

If the `drv_init` routine is defined, the VME bus driver invokes it at bus initialization time. Note that the `drv_init` routine is called in the DKI thread

context, restricting the services available to it (drv_init can only invoke QUICC bus services).

The `drv_init` routine is called with three arguments:

DevNode — The VME device driver's own node, which identifies the node associated to the device in the device tree.

VmeBusOps — The VME bus operations structure, which defines the VME bus API and its version.

VmeId — The VME bus identifier, which is an opaque to be passed back when calling the `VmeBusOps.open` operation to connect the device driver to the VME bus.

From this point on, `VmeBusOps.open` must be the first call issued by the VME device driver to the VME bus driver:

```
    KnError
(*open)(VmeId           vmeId,
        DevNode         devNode,
        VmeEventHandler evtHandler,
        VmeLoadHandler  loadHandler,
        void*           cookie,
        VmeDevId*       devId);
```

This establishes a connection to the bus, identified by the *devId* value returned in the last parameter.

This *devId* identifier is an argument for most of the other services defined in the `VmeBusOps` structure.

vmeId and *devNode* are given by the kernel as parameters to the `drv_init` driver routine.

evtHandler is a handler in the device driver, which is invoked by the VME bus driver when a VME bus event occurs. The *cookie* argument is passed back to this handler as first parameter.

This VME bus event handler takes two additional parameters: the bus event type and an event type specific structure pointer.

The bus event type can be one of the following:

`VME_SYS_SHUTDOWN` — Notifies a device driver that the system is going to be shut down. The device driver should reset the device hardware and return from the event

	handler. Note that the driver must neither notify clients nor free allocated resources.
VME_DEV_SHUTDOWN	Notifies a device driver that the device should be shut down. The device driver should notify driver clients (via svDeviceEvent) that the device is going to be shut down and should then return from the event handler. Once the device entry is released by the last driver client, the device registry module invokes a driver call-back handler. Within this handler, the device driver should reset the device hardware, release all used resources and close the bus connection invoking VmeBusOps.close. Note that the VME_DEV_SHUTDOWN event may be used by a bus driver in order to confiscate (or to re-allocate) bus resources.
VME_DEV_REMOVAL	Notifies a device driver that the device has been removed from the bus and therefore the device driver instance has to be shut down. The actions taken by the driver are similar to the VME_DEV_SHUTDOWN case except that the device hardware must not be accessed by the driver.
VME_SYSFAIL	Indicates that a VME bus module has detected a system failure.
VME_ACFAIL	Indicates that the system power source is about to stop.
VME_ARBITRATION_TIMEOUT	Indicates that the VME bus arbiter has detected an arbitration timeout.

The event type specific structure pointer argument is always NULL for VME bus events.

As the event handler is executed in the context of an interrupt, its implementation must be restricted to the API allowed at interrupt level.

loadHandler is a handler in the device driver which is invoked by the VME bus driver when a new driver appears on the system (for example, a new driver is downloaded at run time). Note that *loadHandler* is optional and must be set to NULL when it is not implemented by the device driver. The *cookie* argument is passed back to this handler as an argument. Typically, this type of event is not important for a leaf VME device driver. `loadHandler` is usually used by a VME nexus driver (for example, SCSI host bus adapter) which has to try to apply a newly downloaded driver to its child devices which are not yet serviced. Note that the `loadHandler` routine is called in the DKI thread context (refer to the table "Allowed Calling Context" to check the list of QUICC bus services allowed in that context).

If `VmeBusOps.open` returns `K_OK`, a valid identifier is returned in the *devId* argument. This identifier must be used to call other services defined in the `VmeBusOps` structure. Some of these services also return an "Ops" structure defining new services on a new object instance, an I/O or a memory region, for example. Others simply perform a service for the device driver, without giving access to a sub-part of the API.

On failure, `VmeBusOps.open` returns an error code as follows:

`K_EINVAL`	The device node is invalid, that is, the device node is not a child of the bus node.
`K_EBUSY`	A connection is already open for the device node specified.
`K_ENOMEM`	The system is out of memory.

To release the connection to the VME bus, the driver must call the `VmeBusOps.close` service as follows:

```
    void
(*close)(VmeDevId devId);
```

After having been disconnected from the VME bus, the device driver can no longer use any of the VME bus API services except `VmeBusOps.open`.

Probing Connection

If the `drv_probe` routine is defined, the VME bus driver invokes it at bus probing time. Note that the `drv_probe` routine is called in the DKI thread context (refer to the table "Allowed Calling Context" to check the list of QUICC bus services allowed in that context).

The `drv_probe` routine is called with three arguments:

DevNode	The VME device driver's own node, which identifies the node associated to the device in the device tree.

VmeBusOps	The VME bus operations structure, which defines the VME bus API and its version.
VmeId	The VME bus identifier, which is an opaque to be passed back when calling the `VmeBusOps.open` operation to connect the device driver to the VME bus.

The `drv_probe` routine is called first by the bus driver. This allows the VME device driver to discover a device (which can be serviced by the driver) on the bus and to create a device node for the device in the device tree. If `drv_probe` creates a node, corresponding to a device residing on the bus, it should attach properties, specifying required bus resources, to the device node (for exmample, "intr", "mem-rgn"). The VME bus driver is able to refer to the device node properties and allocate bus resources for the device (if needed) . The bus driver can also check resource conflicts with other devices residing on the bus.

The `drv_bind` routine is called by the bus driver when the probing phase is complete. It enables the device driver to perform a driver-to-device binding. A VME device driver should examine the device's properties to check whether a given device can be serviced by the driver. If the check is positive, the driver binds itself to the device. This driver initiated binding is achievd by attaching the "driver" property to the device node. The "driver" property value is a `NULL` terminated ASCII string, specifying the driver name. Note that the driver name specified in the property must match the driver name specified in the driver registry entry. Under these conditions, the VME bus driver will invoke the driver's `drv_init` routine for the device node, as described above. Note that the `drv_init` routine is invoked by the bus driver if, and only if, the bus resources required for the device are successfully allocated / checked by the bus driver. Note that the `drv_bind` routine is called in the DKI thread context (refer to the table "Allowed Calling Context" to check the list of QUICC bus services allowed in that context).

Devices residing on the VME bus may be discovered by using the VME bus services. In such cases, a connection must be established between the VME bus driver and the VME device driver. In order to establish a connection, the device driver has to create a device node and to attach it to the bus node. This device node may be temporary but it is mandatory as an argument to the `VmeBusOps.open` routine. When the probing process is finished and the connection is closed, depending on the probing results, the device driver should either delete the temporary device node or transform it to a real device node. In the latter case, the device that may be serviced by the driver is located on the bus.

It is important that the device driver does not create redundant device nodes. In particular, when the device driver discovers a device and wishes to create a

device node corresponding to the device, it should check that there is no existing device node (among the bus child nodes) representing the same device.

If a connection to the VME bus is established by the `drv_probe` routine, it should be closed (via VmeBusOps.close) before leaving the routine.

Accessing VME Slaves' I/O Registers

To perform I/O access to registers of a VME device, a VME driver must:

- Map the device's registers into an I/O region.
- Use services defined by the mapped I/O region to access registers.
- Unmap the region when access is no longer needed.

Mapping Device I/O Registers

The `VmeBusOps.io_map` service is used to map an I/O region from a VME device to enable access to this region as follows:

```
KnError
    (*io_map)(VmeDevId       devId,
              VmePropIoRegs* ioRgn,
              VmeErrHandler  errHandler,
              void*          errCookie,
              VmeIoOps**     ioOps,
              VmeIoId*       ioId);
```

The *ioRgn* structure defines the I/O region to map. This structure is an element of the array stored in a property of the device node. This property should be retrieved by the driver using the device tree API: (`dtreePropFind`, `dtreePropLength`, `dtreePropValue`), prior to calling `VmeBusOps.io_map`.

The name of the property used to described device I/O regions is `"io-regs"`, its value contains an array of `VmePropIoRegs` structures.

```
typedef struct VmePropIoRegs {
        VmeAddrSpace space;
        VmeAddr      addr;
        VmeSize      size;
        VmeAddr      mask;
    } VmePropIoRegs;
```

The *space* field specifies the VME address space where the registers reside:

- VME_A16 VME short address space
- VME_A24 VME standard address space
- VME_A32 VME extended address space
- VME_CSR VME configuration / control and status registers' address space

The *addr* field specifies the VME start address of the registers' range.

The *size* field specifies the registers' range size in bytes.

The *mask* field specifies which address bits are fixed / floating. This field is used by the VME bus driver at the resource allocation stage to determine the device address decoder constraints. A bit set to zero within *mask* specifies a fixed address bit within *addr*. In other words, the corresponding bit within an allocated address must be the same as within *addr*. A bit set to one within *mask* specifies a floating address bit (any value of the corresponding bit within an allocated address is acceptable). When the `VmePropIoRegs` structure is passed to `VmeBusOps.io_map`, the I / O register range is already allocated and therefore the mask field is meaningless.

On success, `K_OK` is returned and appropriate I / O services are returned in the *ioOps* parameter. An identifier for the mapped I / O region is also returned in *ioId*. This identifier must be used as first parameter to further calls to the `VmeIoOps` services.

errHandler is a handler in the device driver, which is invoked by the VME bus driver when a VME bus error occurs while accessing the mapped region. *errCookie* is passed back to the *errHandler* as first parameter.

On failure, `VmeBusOps.io_map` returns an error code as follows:

`K_ESIZE`	A size of zero was specified.
`K_ESPACE`	Invalid or unsupported VME address space.
`K_EOVERLAP`	Not enough virtual address space to map I / O registers.
`K_ENOMEM`	The system is out of memory.

Note that VME I / O access is restricted to supervisory data access.

Performing I/O Access to a Mapped I/O Region

Once the region has been successfully mapped, the driver can use the services defined by the `VmeIoOps` structure.

The VME bus provides five service routine sets to access a mapped I / O region as follows:

- `VmeIoOps.load_8/16/32`
- `VmeIoOps.store_8/16/32`
- `VmeIoOps.read_8/16/32`
- `VmeIoOps.write_8/16/32`
- `VmeIoOps.tas_8/16/32`

There are three service routines in each set which deal with I/O registers of different widths. The _8, _16 or _32 suffix indicates the data size of the transfer on the VME bus.

Usable data sizes for I/O depend on the address space in which the region is mapped. If the driver tries to access a register using too large a data size transfer, the VME bus driver will generate an error, and notify the driver through the error handler attached to the mapped region. The error code will be VME_ERR_INVALID_SIZE.

In all service routines provided by VmeIoOps the *ioId* argument identifies the mapped I/O region.

The *offset* argument specifies the register offset, in bytes, within the mapped region.

These routines all handle byte swapping should the endian be different for the VME bus and the CPU.

Load from a Register

The VmeIoOps.load_xx routine set returns a value loaded from a device register, as follows:

```
    uintxx_f
(load_xx)(VmeIoId ioId, VmeSize offset);
```

The size of the returned value and of the data transfer on the VME bus is specified by the _xx suffix.

Store to a Register

The VmeIoOps.store_xx routine set stores a given value into a device register, as follows:

```
    void
(*store_xx)(VmeIoId ioId, VmeSize offset, uintxx_f value);
```

The size of the given value and the data transfer on the VME bus is specified by the _xx suffix.

Multiple Read from a Register

The VmeIoOps.read_xx routine set loads values from a device register and writes them sequentially into a memory buffer, as follows:

```
    void
(*read_xx)(VmeIoId ioId, VmeSize offset, uintxx_f* buf, VmeSize count);
```

The size of each value loaded and of each data transfer on the VME bus is specified by the _xx suffix.

The *count* argument specifies the number of read transactions to perform.

The *buf* argument specifies the address of the memory buffer. The size of this buffer must be at least (*count* * size of each data transfer).

Multiple Write to a Register

The `VmeIoOps.write_xx` routine set reads values sequentially from a memory buffer and stores them in a device register, as follows:

```
    void
(*write_xx)(VmeIoId ioId, VmeSize offset, uintxx_f* buf, VmeSize count);
```

The size of each value stored and of each data transfer on the VME bus is specified by the `_xx` suffix.

The *count* argument specifies the number of write transactions to perform.

The *buf* argument specifies the address of the memory buffer. The size of this buffer must be at least (*count* * size of each data transfer).

Test and Set a Register (indivisible)

The `VmeIoOps.tas_xx` routine set reads a device register, compares it to a given value, and then writes back to the register the bits that compared true, as follows:

```
    uintxx_f
(*tas_xx)(VmeIoId  ioId,
          VmeSize  offset,
          uintxx_f enable,
          uintxx_f compare,
          uintxx_f set);
```

The size of each value and of each data transfer on the VME bus is specified by the `_xx` suffix.

The *enable* argument specifies a bit mask in order to select (enable) the bits to be modified, if compared true. Each bit set enables comparison and setting for that bit.

The *compare* argument specifies a value which is bitwise compared to the device register read value.

The *set* argument specifies a value to be written back to the device register for valid compared and enabled bits.

The device register read value is bitwise compared with the *compare* value and the *enable* value. Each enabled bit that compares true is replaced by the corresponding bit in the *set* value. A false comparison results in the original bit being unchanged.

The original value read from the device register is returned by the functions.

Assuming that `readVal` is the original value read and `writeVal` is the new value to write back, the following code illustrates the behavior of the `tas_xx` services:

```
  /* read value into readVal */
  replaced = ~(readVal ^ compare) & enable;
  writeVal = (set & replaced) | (readVal & ~replaced);
  /* write back writeVal */
  return readVal;
```

Unmapping Device I/O Registers

When a driver no longer needs access to the device I/O register, or before closing the connection to the VME bus, it should unmap the I/O region used for these devices.

The `VmeBusOps.io_unmap` is used to unmap an I/O region previously mapped with `VmeBusOps.io_map`, and to release any resources allocated for this mapping, as follows:

```
    void
(*io_unmap)(VmeIoId ioId);
```

The *ioId* argument identifies the region to unmap.

When the `VmeBusOps.io_unmap` function is called, the driver can no longer use any services defined for the unmapped I/O region.

Accessing VME Slaves' Memory

In order to access the memory of a VME device, a VME driver must:

- Map device memory to the local virtual memory space.
- Access device memory through the mapped region.
- Unmap the region when access is no longer needed.

The mapped memory region may be accessed directly using its virtual address.

When accessing a memory mapped region, the driver must handle endianning problems.

Mapping Device Memory

The `VmeBusOps.mem_map` service is used to map a memory region from a VME device, enabling access to this region, as follows:

```
    KnError
(*mem_map)(VmeDevId       devId,
           VmePropMemRgn* memRgn,
           VmeErrHandler  errHandler,
           void*          errCookie,
           VmeMemOps**    memOps,
           VmeMemId*      memId);
```

devId is the identifier returned by the `VmeBusOps.open`.

The *memRgn* structure defines the memory region to map. This structure is an element of the array stored in a property of the device node. This

property should be retrieved by the driver using the device tree API: (`dtreePropFind`, `dtreePropLength`, `dtreePropValue`), prior to calling `VmeBusOps.mem_map`.

The name of the property used to describe device memory regions is `"mem-rgn"`, its value contains an array of `VmePropMemRgn` structures, shown below:

```
typedef struct VmePropMemRgn {
        VmeAddrSpace space;
        VmeAddr      addr;
        VmeSize      size;
        VmeAddr      mask;
        VmeMemAttr   attr;
    } VmePropMemRgn;
```

The *space* field specifies the VME memory address space where the memory region resides:

VME_A16	VME short address space
VME_A24	VME standard address space
VME_A32	VME extended address space

The *addr* field specifies the VME start address of the memory region.

The *size* field specifies the memory region size in bytes.

The *mask* field specifies the device address decoder constraints (see section Mapping device I/O registers).

The *attr* argument specifies the mapping attributes. A combination of the following flags is allowed:

`VME_MEM_READABLE`	The memory region is readable.
`VME_MEM_WRITABLE`	The memory region is writable.
`VME_MEM_EXCUTABLE`	The memory region is executable.
`VME_MEM_CACHEABLE`	The memory region is cacheable.
`VME_MEM_INVERTED`	The memory region mapping inverts the byte order. Some MMUs support this type of mapping attribute. This feature may be used by a VME device driver in order to avoid byte swapping within a memory mapped region. Note that `VmeBusOps.mem_map` returns `K_EINVAL` if this type of feature is not supported.
`VME_SUPERVISOR`	the memory region is accessed using Supervisor AM codes

VME_USER	the memory region is accessed using User AM codes
VME_PROGRAM	the memory region is accessed using Program AM codes.
VME_DATA	the memory region is accessed using Data AM codes.
VME_D8	the memory region is accessed using 8-bits data width cycles
VME_D16	the memory region is accessed using 16-bits data width cycles
VME_D32	the memory region is accessed using 32-bits data width cycles
VME_D64	the memory region is accessed using 64-bits data width cycles
VME_BLOCK	Enables block transfers for this memory region.
VME_POSTED_WRITE	Enables posted-write transfers for this memory region.

errHandler is a handler in the device driver which is invoked by the VME bus driver when a VME bus error occurs while accessing the mapped region. *errCookie* is passed back to *errHandler* as a first parameter (see section VME error Handling) .

On success, K_OK is returned and appropriate services are returned in the *memOps* parameter. An identifier for the mapped memory region is also returned in *memId*.

The identifier *memId* must be used as the first parameter to further calls to the VmeMemOps services, and to unmap the region when no longer needed.

On failure, VmeBusOps.mem_map returns an error code as follows:

K_ESIZE	A size of zero was specified.
K_EINVAL	Invalid or unsupported memory mapping attributes .
K_EOVERLAP	Not enough virtual address space to map.
K_ENOMEM	The system is out of memory.

errHandler is a handler in the device driver which is invoked by the VME bus driver when a VME bus error occurs while accessing the mapped region. *errCookie* is passed back to the errHandler as first parameter.

On success, K_OK is returned and appropriate services are returned in the *memOps* parameter. An identifier for the mapped memory region is also returned in *memId*.

This identifier must be used as a first parameter to unmap the region when it is no longer needed.

On failure, VmeBusOps.mem_map returns an error code as follows:

K_ESIZE	A size of zero has been specified.
K_EINVAL	Invalid or unsupported memory mapping attributes.
K_EOVERLAP	Not enough virtual address space to map.
K_ENOMEM	The system is out of memory.

Programmed Access to a Mapped Memory Region

Once the region has been successfully mapped, the driver can retrieve its virtual address to access memory directly, using the VmeMemOps.virt_addr service, as follows:

```
    void*
(*virt_addr)(VmeMemId memId);
```

The *memId* argument is the region identifier returned by VmeBusOps.mem_map.

This means that the device driver may now access the device memory by dereferencing the returned pointer.

Note that in this mode, the driver is responsible for handling any endianning problems.

Test And Set on a Mapped Memory Region (indivisible)

The VmeMemOps.tas_xx routine set reads a memory location, compares it to a given value, and then writes back to this location the bits that compared true, as follows:

```
    uintxx_f
(*tas_xx)(VmeMemId  memId,
          VmeSize  offset,
          uintxx_f enable,
          uintxx_f compare,
          uintxx_f set);
```

The size of each value and of each data transfer on the VME bus is specified by the _xx suffix.

The *memId* is the memory region identifier returned by VmeBusOps.mem_map.

The *offset* argument indicates the offset in bytes, from the beginning of the region, of the memory location to access.

The *enable* argument specifies a bit mask to select (enable) the bits to be modified, if compared true. Each bit set enables comparison and setting for this bit.

The *compare* argument specifies a value which is bitwise compared to the read value.

The *set* argument specifies a value to be written back to the memory location for valid compared and enabled bits.

The memory location read value is bitwise compared with the *compare* value and the *enable* value. Each enabled bit that compares true is replaced by the corresponding bit in the *set* value. A false comparison results in the original bit being unchanged.

The original value read from the memory location is returned by the functions.

Assuming that `readVal` is the original value read and `writeVal` is the new value to write back, the following code illustrates the behavior of the `tas_xx` services:

```
/* read value into readVal */
replaced = ~(readVal ^ compare) & enable;
writeVal = (set & replaced) | (readVal & ~replaced);
/* write back writeVal */
return readVal;
```

Unmapping a memory region

When a driver no longer requires access to the device memory, or before closing the connection to the VME bus, it should unmap the memory region.

The `VmeBusOps.mem_unmap` is used to unmap a memory region, previously mapped with `VmeBusOps.mem_map`, and to release any system resources allocated for this mapping:

```
void
    (*mem_unmap)(VmeMemId memId);
```

The *memId* argument identifies the region to unmap.

Once the memory region is unmapped, the driver can no longer access it.

Direct Memory Access

To perform Direct Memory Access to the memory of a VME device, a VME driver must:

- Allocate a DMA region object in local space.
- Use services defined by the DMA region to perform transfers.
- Release the DMA region when no longer needed.

Allocating a DMA Region

The VmeBusOps.dma_alloc service allocates a physical memory region of a specified size and maps it contiguously to the supervisor address space, as follows:

```
    KnError
(*dma_alloc)(VmeDevId        devId,
             VmeSize         size,
             VmeDmaAlign     dmaConstr,
             VmeDmaAttr      dmaAttr,
             uint32_f        dmaMaxReq,
             VmeDmaHandler   errHandler,
             void            errCookie,
             VmeDmaRgnOps**  dmaRgnOps,
             VmeDmaRgnId*    dmaRgnId);
```

The *devId* argument is the identifier returned by VmeBusOps.open.

The *size* argument is the requested size in bytes to allocate for the DMA memory region.

The *dmaConstr* argument defines the address decoder constraints for the DMA memory region being allocated. Note that the *dmaConstr* argument may be set to NULL, specifying that there are no constraints on the DMA region. Otherwise, it should point to a *VmeDmaAlign* structure shown below:

```
typedef struct VmeDmaAlign {
        VmeAddr addr;
        VmeAddr alignment;
        VmeAddr floating;
    } VmeDmaAlign;
```

The *addr* field specifies the VME start address of the DMA region.

The *alignment* field is a mask which specifies constraints on the start address of the DMA region being allocated. A bit set within *alignment* specifies that the corresponding bit within the start address may take any value, that is, 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 *addr* field. This mask allows the caller to specify an alignment for the start address of the allocated DMA region by zeroing the required number of least significant bits. By resetting the required number of most significant bits, the caller may also indicate in which part of the VME space the memory must be allocated.

The *floating* field is a mask which indicates which bits of the returned address can vary while searching the allocated DMA region for the required size. In other words, bits cleared in the mask must be constant for all addresses in the range of the allocated DMA region. This mask may be used to specify that the allocated amount of memory must not span across a given address boundary.

The *dmaAttr* argument specifies the DMA transfer type. A combination of the following flags is allowed:

VME_DMA_READ	The region is used for DMA read transfer.
VME_DMA_WRITE	The region is used for DMA write transfer.
VME_DMA_SYNC	Allocate a synchronous DMA region for which no synchronization is needed between accesses from a DMA engine and the CPU. This attribute may not be supported on certain platforms.In that case, dma_alloc should return K_EINVAL.

The *memAttr* argument specifies the mapping attributes (see section Mapping device memory).

The *dmaMaxReq* argument indicates the maximum number of simultaneous transfers that may be further requested by the device driver for that region. The VME bus driver ensures that enough resources are reserved to handle at least *dmaMaxReq* simultaneous transfer requests on that region. As these resources are reserved on a per region basis, that is, on a per driver basis, this allows the DMA engine to be shared between drivers.

errHandler is a handler in the device driver. It is invoked by the VME bus driver when a VME bus error occurs whilst accessing to the DMA region. *errCookie* is passed back to the errHandler as a first argument. (see section VME error Handling) .

On success, K_OK is returned and appropriate services are returned in the *dmaRgnOps* parameter. An identifier for the mapped I/O region is also returned in *dmaRgnId*. This identifier must be used as first parameter to further calls to the VmeDmaOps services.

On failure, VmeBusOps.dma_alloc returns an error code as follows:

K_ESIZE	A size of zero was specified.
K_EINVAL	Invalid or unsupported DMA attributes.
K_EOVERLAP	Not enough virtual address space to map.
K_ENOMEM	The system is out of memory.

Getting the Virtual Address of DMA Region

The VmeDmaRgn.virt_addr routine returns the virtual start address of a DMA region, given the identifier of the region (*dmaId*):

```
    void*
(*virt_addr)(VmeDmaRgnId dmaRgnId);
```

The driver uses this address to access the DMA region. Note that the driver should synchronize the region as appropriate, depending on the attributes used when allocating the region.

Synchronizing the DMA region

If the DMA region was not allocated using the `VME_DMA_SYNC` attribute, the device driver should synchronize accesses to the same physical memory (the DMA region) from the CPU and the DMA engine correctly. Depending on the platform, these routines handle possible problems of cache flushing, DMA engine buffers, and others.

`VmeDmaRgnOps.read_sync` is a barrier between CPU writes and DMA reads from a given DMA sub-region:

```
    void
(*read_sync)(VmeDmaRgnId dmaRgnId, VmeSize offset, VmeSize size);
```

`VmeDmaRgnOps.write_sync` is a barrier between DMA writes and CPU reads from a given DMA sub-region:

```
    void
(*write_sync)(VmeDmaRgnId dmaRgnId, VmeSize offset, VmeSize size);
```

For both routines:

The *dmaRgnId* argument identifies the DMA region.

offset is the offset in bytes from the beginning of the DMA memory region.

offset and *size* both define the sub-region to synchronize.

Requesting DMA Transfer

The `VmeDmaRgnOps.read` function requests that a DMA transfer reads from the local memory region identified by *srcDmaRgnId* and writes to the mapped VME memory region identified by *dstMemId*:

```
    KnError
(*read)(VmeDmaRgnId    srcDmaRgnId,
        VmeSize        srcDmaRgnOff,
        VmeMemId       dstMemId,
        VmePropMemRgn* dstMemRgn,
        VmeSize        dstMemOff,
        VmeSize        size,
        VmeDmaHandler  dmaHandler,
        void*          dmaCookie,
        VmeDmaReqId*   dmaReqId);
```

The portion of the local memory region to read is defined by:

srcDmaRgnId	Identifier of the DMA region
srcDmaRgnOff	Offset in bytes from the beginning of the region

size Number of bytes to read starting from *srcDmaRgnOff*

The portion of the VME memory region to which to write is defined by:

dstMemId Points to a `VmePropMemRgn` structure defining the VME region to write to.

dstMemOff Offset in bytes from the beginning of this region

size Number of bytes to write starting at *dstMemOff*

Before calling `VmeDmaRgnOps.read` the VME driver should call `VmeDmaRgnOps.read_sync` to synchronize the DMA region, if it was not allocated using the `VME_DMA_SYNC` attribute.

The `VmeDmaRgnOps.write` function requests a DMA transfer to write to a local memory region identified by *dstDmaRgnId* and to read from the mapped VME memory region identified by *srcMemId*:

```
    KnError
(*write)(VmeDmaRgnId    dstDmaRgnId,
         VmeSize        dstDmaRgnOff,
         VmeMemId       srcMemId,
         VmePropMemRgn* srcMemRgn
         VmeSize        srcMemOff,
         VmeSize        size,
         VmeDmaHandler  dmaHandler,
         void*          dmaCookie,
         VmeDmaReqId*   dmaReqId);
```

The portion of the local memory region to write to is defined by:

dstDmaRgnId Identifier of the DMA region

dstDmaRgnOff Offset in bytes from the beginning of the region

size Number of bytes to write starting at *dstDmaRgnOff*

The portion of the VME memory region to read is defined by:

srcMemId Points to a VmeBusOps.mem_map structure defininf the VME bue memory region to map to (see section Mapping Device Memory).

srcMemOff Offset in bytes from the beginning of this region

size Number of bytes to read starting from *srcMemOff*

After a transfer requested using `VmeDmaRgnOps.write` is completed, the VME driver should call `VmeDmaRgnOps.write_sync` to synchronize the DMA region, if it was not allocated with the `VME_DMA_SYNC` attribute.

For both `read` and `write` routines:

On success, an identifier of the request is returned in *dmaReqId* that may be used to abort the transfer before it completes, using `VmeDmaRgnOps.abort`:

```
    void
(*abort)(VmeDmaReqId req);
```

On failure, `K_ETOOMUCH` is returned, to indicate that the maximum number of simultaneous transfer requests for that region has been reached. In that case, the driver should wait for the completion of the first pending request. That means, the driver should not call `read` or `write` routines before the *dmaHandler* for that region has been called back. The maximum number of simultaneous transfer requests that is accepted for a given DMA region is defined by the *maxDmaReq* argument of the `VmeBusOps.dma_alloc` service.

The *dmaHandler* argument is a handler in the device driver which is invoked by the VME bus driver when a DMA transfer has been completed.

This handler is a standard `VmeErrHandler` handler (see section VME Error Handling). However, if the request completed successfully, the *VmeError* pointer is set to `NULL` to indicate that no error occurred.

The *dmaCookie* is passed back to the *dmaHandler* as a parameter.

Exporting Memory to Other VME Masters

To ensure that the memory exported to other masters can be accessed through all possible VME cycle types, without alignment or other problems, only DMA memory regions can be exported.

This means that only memory regions allocated using `VmeBusOps.dma_alloc` can be made available for access by other masters on the bus.

Other masters may be of 2 kinds:

- Devices which have their own DMA engine, and which need a VME address on the bus to transfer data to the memory of a CPU module
- Other CPU modules, which need to access local memory to implement a communication protocol over a VME bus, either through programmed accesses, or through DMA (which in fact leads back to the first case).

In either case, the VmeDmaRgnOps.map service defined for an allocated DMA memory region must be used to export memory:

```
    KnError
(*map)(VmeDmaRgnId    dmaRgnId,
       VmeSize        dmaRgnOff,
       VmePropMemRgn* memRgn,
       VmeDmaMapId*   dmaMapId);
```

The *dmaRgnId* argument identifies the allocated DMA region.

The *memRgn* argument points to a VmePropMemRgn structure, defining the VME bus memory region that should be responded to (see section Mapping device memory).

The *dmaRgnOff* is the offset in bytes from the beginning of the DMA region. Together with the *size* parameter, they define the local memory address range exported to the VME bus.

On success, K_OK is returned and an identifier for the DMA mapped region is returned in *dmaMapId.* This identifier must be used to unmap the region using VmeDmaRgnOps.unmap.

On failure, VmeDmaOps.map returns one of the following error codes::

K_EFAIL	the VME bus driver does not have enough resources to map the required DMA sub-region with the required attributes
K_ENOMEM	the system is out of memory.

Unmapping a DMA region from VME bus

To unmap a DMA region from the VME bus space, a VME driver must use VmeDmaRgnOps.unmap:

```
void
    (*unmap)(VmeDmaMapId dmaMapId);
```

dmaMapId is the identifier for the DMA sub-region previously mapped to the VME bus (using VmeDmaRgnOps.map).

Releasing a DMA Region

When a driver no longer needs an allocated DMA region, or before closing the connection to the VME bus, it should release the DMA region.

The VmeBusOps.dma_free is used to release a DMA region, previously allocated with VmeBusOps.dma_alloc:

```
    void
(*dma_free)(VmeDmaRgnId dmaRgnId);
```

The *dmaRgnId* argument identifies the region to free.

When the `VmeBusOps.dma_free` function is called, the driver may no longer use the identifier of the DMA region (*dmaRgnId*).

VME INTERRUPT HANDLING

To connect a handler to a VME interrupt request, a VME driver should:

- Disable the interrupt at the device level, typically by accessing the device registers.
- Attach a handler to the interrupt.
- Enable the interrupt at the device level.
- Use services defined by the attached interrupt object.
- Disable the interrupt at the device level.
- Detach the interrupt handler when no longer needed.

Attaching a Handler to a VME Interrupt Source

A VME device driver can connect a handler to a VME interrupt source by calling the `VmeBusOps.intr_attach` service:

```
    KnError
(*intr_attach)(VmeDevId       devId,
               VmePropIntr*   intr,
               VmeIntrHandler intrHandler,
               void*          intrCookie,
               VmeIntrOps**   intrOps,
               VmeIntrId*     intrId);
```

The *devId* argument identifies the connection with the VME bus driver.

The *intr* argument indicates the VME interrupt source to which the handler will be connected. Available VME interrupt sources are VME bus interrupt requests 1 through 7. Typically, a VME device driver should find the interrupt to attach to by looking up the `"intr"` property in the device node.

The *intrHandler* argument is a handler in the device driver which is invoked by the VME bus driver when the corresponding interrupt request is triggered by a module on the bus:

```
typedef VmeIntrStatus (*VmeIntrHandler)(void*         intrCookie,
                                        VmeIntrVector vector);
```

The *intrCookie* is passed back to this interrupt handler as first parameter. The second argument of this handler (*vector*) is the STATUS/ID value read on the bus during the interrupt acknowledge cycle. This type of VME interrupt handler must return a *VmeIntrStatus* value which indicates to the bus driver whether the interrupt was claimed by the handler, and how it was handled:

`VME_INTR_UNCLAIMED` Must be returned by the interrupt handler if there is no pending interrupt for the device.

VME_INTR_CLAIMED	Must be returned by the interrupt handler if a pending device interrupt has been serviced by the interrupt handler and the interrupt has not been enabled (acknowledged) at VME bus level.
VME_INTR_ACKNOWLEDGED	Must be returned by the interrupt handler if a pending device interrupt has been serviced by the interrupt handler and the interrupt has been enabled (acknowledged) at VME bus level.

On success, K_OK is returned and services defined on an attached interrupt object are returned in the *intrOps* parameter. An identifier for the attached interrupt is also returned in *intrId*. This identifier must be used as first parameter to further calls to the VmeIntrOps services.

A given VME bus interrupt request may be shared by multiple VME drivers on the same CPU module. Note that due to the VME bus design, the same interrupt request can not be handled on different CPU modules.

That means multiple drivers may attach a handler to the same VME interrupt request. In that case, when this type of interrupt request occurs on the bus, all the attached handlers are called in an order that is not defined and is implementation specific. When called, each handler can test the *vector* argument to check whether the request was triggered by the device it manages.

As an interrupt request is shareable, a driver must treat the interrupt it attaches to as being enabled at the bus level, and that it may occur as soon as the attachement is done. The fact that a driver could already have attached a handler to the interrupt, and enabled it, dictates this behavior.

If this is not acceptable for a given driver, it must disable the interrupt at the device level prior to attaching the handler. This ensures that the interrupt will not be called with the expected vector value until the interrupt is enabled again at the device level.

On failure, VmeBusOps.intr_attach returns one of the following error codes:

K_EINVAL	specified *intr* is invalid
K_ENOMEM	not enough system memory

Masking/Unmasking an Attached Interrupt

The VmeIntrOps.mask service routine masks the interrupt source specified by *intrId*:

```
    void
(*mask) (VmeIntrId intrId);
```

Note that VmeIntrOps.mask does not guarantee that all other interrupt sources are still unmasked.

The VmeIntrOps.unmask service routine unmasks the interrupt source previously masked by VmeIntrOps.mask:

```
    void
(*unmask) (VmeIntrId intrId);
```

Note that VmeIntrOps.unmask does not guarantee that the interrupt source is unmasked immediately. The real interrupt source unmasking may be deferred.

The VmeIntrOps.mask/VmeIntrOps.unmask pair may be used at either base or interrupt level. Note that the VmeIntrOps.mask/VmeIntrOps.unmask pairs must not be nested.

Enabling/Disabling a Serviced Interrupt

The VmeIntrOps.enable and VmeIntrOps.disable service routines are dedicated to interrupt handler usage only. In other words, these routines may be called only by an interrupt handler.

The VmeIntrOps.enable service routine enables (and acknowledges) the bus interrupt source specified by *intrId*:

```
    VmeIntrStatus
(*enable) (VmeIntrId intrId);
```

VmeIntrOps.enable returns either VME_INTR_ACKNOWLEDGED or VME_INTR_CLAIMED. The VME_INTR_ACKNOWLEDGED return value means that the VME bus driver has enabled (and acknowledged) the interrupt at bus level. The VME_INTR_CLAIMED return value means that the VME bus driver has ignored the enable request and therefore the interrupt source is still disabled (and not acknowledged) at bus level.

Note that the bus driver typically refuses an explicit interrupt acknowledgement (issued by an interrupt handler) for the shared interrupts. In the latter case, the bus driver will acknowledge interrupts only when all interrupt handlers have been called.

Note that in cases where the VmeIntrOps.enable routine was called by an interrupt handler, the handler must return the value which was returned by VmeIntrOps.enable. Note also that once VmeIntrOps.enable has been called, the driver should be able to handle an immediate re-entrance into the interrupt handler code.

The VmeIntrOps.disable service routine disables the interrupt source previously enabled by VmeIntrOps.enable:

```
    void
(*disable) (VmeIntrId intrId);
```

If `VmeIntrOps.enable` returns `VME_INTR_ACKNOWLEDGED`, the driver must call `VmeIntrOps.disable` prior to returning from the interrupt handler.

When an interrupt occurs, the attached *VmeIntrHandler* is invoked with the interrupt source disabled at bus level. This behaves in exactly the same way as if `VmeIntrOps.disable` was called just prior to the handler invocation. Note that the interrupt handler must return to the bus driver in the same context as it was called, that is, with the interrupt source disabled at bus level.

On the other hand, the called interrupt handler may use the `VmeIntrOps.enable/disable` pair to allow the interrupt to be nested. This feature is typically used by a VME-to-bus bridge driver when the secondary bus interrupts are multiplexed, that is, multiple secondary bus interrupts are reported through the same primary bus interrupt. Typically, an interrupt handler of a VME-to-bus bridge driver would take the following actions:

- Identify and disable the secondary bus interrupt source
- Enable the primary bus interrupt source (`enable`)
- Call handlers attached to the secondary bus interrupt source
- Disable the primary bus interrupt source (`disable`)
- Acknowledge (if needed) and enable the secondary bus interrupt source
- Return from the interrupt handler

Detaching an Attached Interrupt

When a driver no longer needs to handle an interrupt, or before closing the connection to the VME bus, it should detach the handler attached to an interrupt source.

The `VmeBusOps.intr_detach` is used to detach a handler, previously attached with `VmeBusOps.intr_attach`, and to release any resources allocated for this attachment:

```
    void
(*intr_detach)(VmeIntrId intrId);
```

The *intrId* argument identifies the attachment to release.

When the `VmeBusOps.intr_detach` function is called, the driver is no longer allowed to use the identifier (*intrId*).

VME INTERRUPT REQUESTING

A VME driver may need to trigger (via software) an interrupt request on the VME bus, to implement a protocol between CPU modules, for example.

This can be done using the `VmeBusOps.intr_trigger` service:

```
        KnError
    intr_trigger (VmeDevId          devId,
                  VmePropIntr       intr,
                  VmeIntrVector     vector,
                  VmeIntrAckHandler iackHandler,
                  void*             iackCookie);
```

The *intr* argument indicates which VME interrupt request to trigger on the bus.

The *vector* argument indicates which STATUS/ID to deliver on the bus during the interrupt acknowledge cycle.

The *iackHandler* argument is a handler in the device driver. It is invoked by the VME bus driver once the STATUS/ID has been provided to an interrupt handler on the VME bus:

```
typedef void (*VmeIntrAckHandler) (void* iackCookie);
```

The *iackCookie* is passed back to this handler as the first argument.

On success, `intr_trigger` returns `K_OK`, and the `iackHandler` handler is called once the triggered interrupt has been handled by an interrupt handler VME module.

On failure, `VmeBusOps.intr_trigger` returns one of the following error codes:

K_EINVAL	specified *intr* is invalid
K_EBUSY	a previously triggered interrupt has not yet been handled

VME ERROR HANDLING

When mapping VME bus space, a VME driver provides an error handler that is invoked by the VME bus driver when a bus error occurs.

Error handlers are used for the following services:

- VmeBusOps.io_map
- VmeBusOps.mem_map
- VmeBusOps.dma_alloc
- VmeDmaOps.read
- VmeDmaOps.write

When a programmed, or direct, memory access is aborted because of a bus error, the VME bus invokes the error handler for the VME address and space concerned.

A `VmeErrHandler` is defined as follow:

```
typedef void (*VmeErrHandler)(void* errCookie, VmeError* err);
```

The *errCookie* is an opaque value that is given by the caller and passed back when the handler is called.

The *err* argument points to a structure describing the error as follows:

```
typedef struct VmeError {
        VmeErrCode code;
        VmeSize    offset;
    } VmeError;
```

The *code* field indicates the type of error that occurred.

The *offset* field indicates the offset, within the associated region, at which the error occurred.

The error code can be as follows:

VME_ERR_UNKNOWN	Unknown error, the VME bus driver is unable to determine the reason for the exception.
VME_ERR_INVALID_SIZE	Invalid (not supported) access granularity has been used.
VME_ERR_PARITY	A parity error occurred during a transfer.
VME_ERR_BUS	A VME bus error (BERR) occurred. The offset indicates the first address in the region at which a BERR occurred.
VME_DMA_ABORTED	A DMA transfer has been aborted by the driver before it had completed. The offset indicates the address in the region where the transfer was stopped.

As a bus error may be reported in the context of an exception or an interrupt, the implementation of the error handler must be restricted to the interrupt level API.

Checking for a VME bus system controller

To check whether a given VME bus controller is the VME system controller, a VME device driver should use `VmeBusOps.is_sys_con`:

```
Bool
    is_sys_con (VmeDevId  devId);
```

The *devId* argument identifies the connection with the VME bus driver.

`VmeBusOps.is_sys_con` returns `TRUE` if the VME bus controller is the VME bus system controller. Otherwise, the routine returns `FALSE`.

VME Bus Locking/Unlocking

The `VmeBusOps.lock/unlock` services allow a driver to implement critical sections, by ensuring mastership of the VME bus between a call to `lock` and a call to `unlock`.

```
    void
    (*lock) (VmeDevId  devId);
```

```
      void
(*unlock) (VmeDevId  devId);
```

The `lock` command requires and holds mastership of the VME bus until `unlock` is called. This allows a CPU module on the VME bus to run multiple cycles on the bus without losing mastership of the bus. This means that no other master can access the bus until `unlock` is called.

VME Specific Node Properties

The table below lists the VME specific node properties. The *alias* column specifies the alias name which should be used by a VME bus or device driver referencing the property name. The **name** column specifies the property name ASCII string. The **value** column specifies the type of the property value. The **bus** column specifies properties specific to the VME bus node. The **dev** column specifies properties specific to the VME device node. The `m` in the **bus/dev** columns flags mandatory properties. The `o` in the **bus/dev** columns flags optional properties. The `-` symbol in the **bus/dev** columns flags properties which are not applied to a given node type.

alias	name	value	bus	dev
VME_PROP_INTR	“intr”	VmePropIntr[]	-	o
VME_PROP_IO_REGS	“io-regs“	VmePropIoRegs[]	-	o
VME_PROP_MEM_RGN	“mem-rgn“	VmePropMemRgn[]	-	o
VME_PROP_DMA_BURST	“dma-burst“	VmePropDmaBurst	m	-
VME_PROP_DMA_OFF_TIMER	“dma-off“	VmePropDmaOffTimer	o	-
VME_PROP_DMA_MIN_SIZE	“dma-min-size“	VmePropDmaMinSize	m	-
VME_PROP_BYTE_ORDER	“byte-order“	VmePropByteOrder	m	-
VME_PROP_CLOCK_FREQ	“clock-freq“	VmePropClockFreq	o	-
VME_PROP_REL_MODE	“rel-mode“	VmePropRelMode	o	-
VME_PROP_REQ_MODE	“req-mode“	VmePropReqMode	o	-
VME_PROP_REQ_LEVEL	“req-level“	VmePropReqLevel	o	-
VME_PROP_ARB_MODE	“arb-mode“	VmePropArbMode	o	-
VME_PROP_ARB_TIMEOUT	“arb-timeout“	VmePropArbTimeout	o	-
VME_PROP_BUS_TIMEOUT	“bus-timeout“	VmePropBusTimeout	o	-

When the value of a property is an array (for example, `VmePropIntr[]`). The size of this array (that is, the size of the array is the size of the property value returned by `dtreePropValue`) divided by the size of its element type (for example, `sizeof(VmePropIntr)`), defines the number of elements in the array. A VME device driver may then iterate through the array in order to perform an action for each element (for example, to attach an interrupt handler to each device interrupt line).

VME Device Node Properties

The VME_PROP_INTR property of a VME device node indicates the VME interrupt sources required / allocated for this device, and whether these interrupts may be shared or are exclusive to that device. This property value should be used by a driver for this device to attach an interrupt handler to each device interrupt (`VmeBusOps.intr_attach`).

The VME_PROP_IO_REGS property of a VME device node defines the VME I / O regions required / allocated for this device. This property value should be used by a driver for this device to map each I / O region for further access to the device (`VmeBusOps.io_map`).

The VME_PROP_MEM_RGN property of a VME device node defines the VME memory regions required / allocated for this device. This property value should be used by a driver for this device to map each memory region for further access to the device (`VmeBusOps.mem_map`).

VME Bus Node Properties

Specific properties are used to configure the behavior of the VME bus controller. These properties are defined only for a VME bus node / nexus. Depending on the underlying controller, one or another feature described below may or may not be implemented. In cases where a feature is not implemented, the properties will not exist. On the other hand, a controller may allow for specific tuning not listed here. Thus, users should refer to the VME bus driver specific documentation to get the exact list of available properties.

Requester Properties

Bus release mode (VME_PROP_REL_MODE):

- Release On Request (ROR)
- Release When Done (RWD)
- Release On BCLR (ROC)

The Default setting is RWD

Bus request level is one of (VME_PROP_REQ_LEVEL):

- BR0
- BR1
- BR2
- BR3 (highest priority)

The default setting is BR3

Bus request mode (VME_PROP_REQ_MODE) is one of:

- Demand: requester asserts its bus request regardless of the state of BRn.
- Fair: requester does not request the bus until there are no other requests pending at its level.

The default setting is Demand

System Controller Properties

Note that these options are used only when the module is a system controller.
Bus arbitration mode (VME_PROP_ARB_MODE) is one of:

- Fixed priority (PRI) (BR3...BR0)
- Single Level (SGL) bus is granted permission only to request at level BR3
- Round Robin Select (RRS)

The Default setting is RRS

Bus arbitration time-out (VME_PROP_ARB_TIMEOUT):

Value in micro-seconds

The Default setting is 16us

Bus time-out (VME_PROP_BUS_TIMEOUT):

Value in micro-seconds

The Default setting is 64us

DMA Transfer Properties

DMA burst maximum size (VME_PROP_DMA_BURST):

Value in bytes

Defines how much data the DMA will transfer before giving the opportunity to another master to assume ownership of the bus. 0 means "until transfer completion".

DMA bus off timer (VME_PROP_DMA_OFF_TIMER), interleave period:

Value in micro-seconds

Defines the minimum period the DMA is off the VME bus between tenures. 0 means the bus is immediatly re-requested.

Dynamic Resource Allocation

The VME bus driver may support dynamic allocation on the VME bus resources. In cases where dynamic resource allocation is supported, a VME device driver may use the `VmeBusOps.resource_alloc` service routine in order to allocate a bus resource at run time:

```
    KnError
(*resource_alloc) (VmeDevId devId, DevProperty prop);
```

devId is returned by `VmeBusOps.open`.

prop specifies the bus resource being allocated.

The `VmeBusOps.resource_alloc` service routine allocates a given bus resource and, if the allocation request is satisfied, updates the device node properties in order to add the newly allocated bus resource.

On success, `VmeBusOps.resource_alloc` returns `K_OK`.

On failure, one of the following error codes is returned:

`K_ENOTIMP`	Dynamic resource allocation is not supported by the bus driver.
`K_EUNKNOWN`	The property name is unknown.
`K_EINVAL`	The property value is invalid.
`K_EFAIL`	The bus resource is not available.
`K_ENOMEM`	The system is out of memory.

When a dynamically allocated bus resource is no longer used, a VME device driver may release it by calling the `VmeBusOps.resource_free` service routine:

```
    void
(*resource_free) (VmeDevId devId, DevProperty prop);
```

devId is returned by `VmeBusOps.open`.

prop specifies the bus resource being released.

The `VmeBusOps.resource_free` service routine releases a given bus resource and updates the device node properties in order to remove the bus resource being released.

Note that the `VmeBusOps.resource_alloc` and `VmeBusOps.resource_free` routines should not be used by a simple device driver. This type of driver should opearate as though all needed resources have already been allocated and specified as properties in the device

node by the bus driver. The driver should simply find the property and call an appropriate service routine, passing a pointer to the property value.

However, it is not always possible to determine all needed resources prior to device initialization. A typical case is a bus-to-bus bridge driver which can discover devices residing on the secondary bus only once the bus bridge hardware has already been initialized. In this case, when `drv_init` is called, the bus-to-bus bridge node would contain only resources needed for the bus-to-bus bridge device itself (for example, internal bus-to-bus bridge registers). Once devices residing on the secondary bus are discovered, the bus-to-bus bridge driver would request additional primary bus resources in order to satisfy the resource requirements for these devices. Another example is a bus which supports hot-pluggable devices. On this type of bus, the primary bus resources allocated by the hot-pluggable bus driver depend on devices currently plugged into the secondary bus. Usually, the resource requirements are changed when a hot-plug insertion / removal occurs.

Note that dynamic resource allocation might be also used by a device driver which implements lazy resource allocation. In this type of driver, the bus resource allocation might be performed at open time using `VmeBusOps.resource_alloc`. Then, the dynamically allocated resources might be released at close time using `VmeBusOps.resource_free`.

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
VmeBusOps.open	-	+	-	+
VmeBusOps.close	-	+	-	+
VmeBusOps.intr_attach	-	+	-	+
VmeBusOps.intr_detach	-	+	-	+
VmeBusOps.intr_trigger	+	+	+	-
VmeBusOps.io_map	-	+	-	+
VmeBusOps.io_unmap	-	+	-	+
VmeBusOps.dma_alloc	-	+	-	+
VmeBusOps.dma_free	-	+	-	+
VmeBusOps.mem_map	-	+	-	+

Services	Base level	DKI Thread	Interrupt	Blocking
VmeBusOps.mem_unmap	-	+	-	+
VmeBusOps.resource_alloc	-	+	-	+
VmeBusOps.resource_free	-	+	-	+
VmeBusOps.is_sys_con	+	+	+	-
VmeBusOps.lock	+	+	+	-
VmeBusOps.unlock	+	+	+	-
VmeBusOps.cpuId	+	+	+	-
VmeIoOps.load_xx	+	+	+	-
VmeIoOps.store_xx	+	+	+	-
VmeIoOps.read_xx	+	+	+	-
VmeIoOps.write_xx	+	+	+	-
VmeIoOps.tas_xx	+	+	+	-
VmeMemOps.virt_addr	+	+	+	-
VmeMemOps.tas_xx	+	+	+	-
VmeDmaRgnOps.virt_addr	+	+	+	-
VmeDmaRgnOps.read_sync	+	+	+	-
VmeDmaRgnOps.write_sync	+	+	+	-
VmeDmaRgnOps.read	+	+	+	-
VmeDmaRgnOps.write	+	+	+	-
VmeDmaRgnOps.abort	+	+	+	-
VmeDmaRgnOps.map	-	+	-	+
VmeDmaRgnOps.unmap	-	+	-	+
VmeIntrOps.mask	+	+	+	-
VmeIntrOps.unmask	+	+	+	-
VmeIntrOps.enable	-	-	+	-
VmeIntrOps.disable	-	-	+	-

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

Index