This chapter contains reference pages describing routines and data structures used by drivers to communicate with the WSIO CDIO. Some earlier interfaces have been deprecated and are now located in Appendix A.
Functions, Macros and Structures
NAME

`bp_dma_cleanup` (WSIO3) – Cleanup after a DMA transfer for a list of buffers.

SYNOPSIS

```c
void bp_dma_cleanup (struct isc_table_type *isc, struct buf *bp, 
                     struct bp_dma_parms *bp_dma_parms);
```

PARAMETERS

- `isc` Pointer to an ISC structure.
- `bp` Pointer to the first `buf` structure
- `bp_dma_parms` Pointer to a `bp_dma_parms` structure.

DESCRIPTION

The `dma_cleanup()` WSIO function performs the required cleanup after a DMA transfer has completed for a list of buffers.

RETURN VALUES

None

CONSTRAINTS

EXAMPLES

```c
struct bp_dma_parms *bp_dma_parms = &lsp->bp_dma_parms;
struct dma_parms *dma_parms = &bp_dma_parms->dma_parms;

if (bp->b_merge_cnt == 0) {
    (void)dma_cleanup(isc, dma_parms);
} else {
    bp_dma_cleanup(isc, bp, bp_dma_parms);
}
```
SEE ALSO

bp_dma_setup(WSIO3), dma_cleanup(WSIO3), dma_setup(WSIO3)
NAME
bp_dma_setup(WSIO3) – Set up a DMA transfer for a list of buffers.

SYNOPSIS

```
int bp_dma_setup (struct isc_table_type *isc, struct buf *bp,
                 struct bp_dma_parms *bp_dma_parms);
```

PARAMETERS

- **isc**
  Pointer to an ISC structure.
- **bp**
  Pointer to the first buf structure.
- **bp_dma_parms**
  Pointer to a dma_parms structure.

DESCRIPTION

The `bp_dma_setup()` WSIO sets up a DMA transfer for a list of buffers. A driver calls this function when `bp->b_merge_cnt` is not zero (i.e., when the B2_LIST flag is set in `bp->b2_flags`). Buffer lists are sent to the driver only when the driver registers that it can handle B2_LIST buffers. The `bp_dma_parms` structure must be initialized by the driver before calling `bp_dma_setup()`.

RETURN VALUES

- **0**
  Successful completion.
- **<>0**
  Error.

CONSTRAINTS
EXAMPLES

struct bp_dma_parms *bp_dma_parms = &lsp->bp_dmaParms;
struct dma_parms *dma_parms = &bp_dma_parms->dma_parms;

dma_parms->flags = NO_WAIT;
dma_parms->channel = BUS_MASTER_DMA;
dma_parms->dma_options = DMA_8BYTE;

if (bp->b_merge_cnt == 0) {
    dma_parms->dma_options = (bp->b_flags & B_READ) ?
        DMA_READ : DMA_WRITE;
    dma_parms->spaddr = bp->b_spaddr;
    dma_parms->addr   = bp->b_un.b_addr;
    dma_parms->count  = bp->b_bcount;
    retval = dma_setup(isc, dma_parms);
} else {
    retval = bp_dma_setup(isc, bp, bp_dma parms);
}

SEE ALSO

hp_dma_cleanup(WSIO3), dma_cleanup(WSIO3), dma_setup(WSIO3)
NAME
dma_cleanup(WSIO3) – Clean up from a DMA transfer.

SYNOPSIS

int dma_cleanup (struct isc_table_type *isc,
                  struct dma_parms * dma_parms);

PARAMETERS

isc Pointer to an ISC structure.
dma_parms Pointer to a dma_parms structure.

DESCRIPTION

The dma_cleanup() WSIO function performs the required cleanup for a
DMA transfer.

RETURN VALUES

None.

CONSTRAINTS

SEE ALSO

hp_dma_setup(WSIO3), dma_setup(WSIO3)
NAME

dma_parms(CDIO4) – DMA information structure

SYNOPSIS

#include <sys/io.h>

PARAMETERS

struct dma_parms
{
    int channel;
    int dma_options;
    int flags;
    int key;
    int num entries;
    buflet_info_type * buflet_key;
    struct iovec * chain_ptr;
    int chain_count;
    int chain_index;
    int (*drv_routine)(caddr_t drv_arg);
    caddr_t drv_arg;
    int transfer_size;
    caddr_t addr;
    space_t spaddr;
    int count;
};

#include <sys/eisa.h>

/* dma_options bits */

#define DMA_ISA 0x1
#define DMA_TYPEA 0x2
#define DMA_TYPEB 0x4
#define DMA_BURST 0x8
#define DMA_TYPEC DMA_BURST
#define DMA_DEMAND 0x10
#define DMA_SINGLE 0x20
#define DMA_BLOCK 0x40
#define DMACASCADE 0x80
#define DMA_8BYTE 0x100
#define DMA_16WORD 0x200
#define DMA_16BYTE 0x400
#define DMA_32BYTE 0x800
#define DMA_READ 0x1000
#define DMA_WRITE 0x2000

DESCRIPTION

Legacy interface drivers can use the dma_parms structure for setting up a DMA transfer. The kernel routines dma_setup() and dma_cleanup() use a pointer to this structure as one of the parameters.

New interface drivers use the various WSIO mapping services in conjunction with iovec (KER4) structures to set up bus master mapping.

SEE ALSO

bp_dma_cleanup(WSIO3), bp_dma_setup(WSIO3),
dma_cleanup(WSIO3), dma_setup(WSIO3), iovec(KER4),
wsio_map(WSIO3)
NAME

dma_setup (WSIO3) – Set up a DMA transfer

SYNOPSIS

#include <wsio/wsio.h>

int * dma_setup (struct isc_table_type * isc,
                struct dma_parms * dma_parms);

PARAMETERS

isc  Pointer to an ISC structure.

dma_parms Pointer to a dma_parms structure.

DESCRIPTION

The dma_setup() WSIO function sets up a DMA transfer. The
dma_parms structure must be initialized before calling dma_setup().

RETURN VALUES

0      Successful completion.

<>0    Error.

CONSTRAINTS
EXAMPLES

struct dma_parms *thisdma;

/*
 * Common DMA setup code. Most of this code does not
 * vary between READ and WRITE. The dma channel has been
 * allocated during attach/init so that does not have to
 * be done here.
 */
thisdma = &pdp->pd_dma_parms;
bzero(thisdma, sizeof(struct dma_parms));
thisdma->flags = NO_WAIT;
thisdma->channel = BUS_MASTER_DMA;
thisdma->addr = bp->b_un.b_addr;
thisdma->spaddr = bp->b_spaddr;
thisdma->count = bp->b_bcount;
thisdma->dma_options = DMA_8BYTE;

if (bp->b_flags & B_READ) {
    thisdma->dma_options |= DMA_READ;
} else {
    thisdma->dma_options |= DMA_WRITE;
}

if (dma_setup(pdp->isc,thisdma) == 0) {
    /* DMA setup successfully completed. */
}

SEE ALSO

bp_dma_setup(WSIO3), dma_cleanup(WSIO3)
NAME

\texttt{driver_addr_probe} (WSIO_DRV) – Interface driver address probing function.

SYNOPSIS

\begin{verbatim}
#include <wsio/wsio.h>
#include <sys/ioparams.h>

int driver_addr_probe (void *this_node, int (*dev_probe)(),
                      drv_info_t *drv_info, void *probe_id,
                      hw_path_t *hw_path, struct ics_table_type *isc,
                      int probe_type, char *name, char *desc);
\end{verbatim}

PARAMETERS

\begin{description}
  \item[\texttt{this_node}]{A pointer to an \texttt{io_tree_node} struct.}
  \item[\texttt{dev_probe}]{Probe function registered by device driver to be called by \texttt{driver_addr_probe}().}
  \item[\texttt{drv_info}]{The \texttt{drv_into_t} struct registered with \texttt{wsio_install}().}
  \item[\texttt{probe_id}]{A unique identifier (for example, first 4 bytes of SCSI Inquiry data).}
  \item[\texttt{hw_path}]{A pointer to a structure containing the hardware path information of the module being probed.}
  \item[\texttt{isc}]{A pointer to the ISC structure assigned to the interface node that is being probed.}
  \item[\texttt{probe_type}]{The type of hardware probe to perform. Defined types are:}
    \begin{description}
      \item[\texttt{PROBE_FIRST}]{Start at first available address.}
      \item[\texttt{PROBE_NEXT}]{Increment the last address and start looking from there.}
      \item[\texttt{PROBE_ADDRESS}]{Look only for this address.}
    \end{description}
  \item[\texttt{name}]{A string describing the device.}
\end{description}
A string describing the device found by the probe (usually 8 bytes of Vendor Id followed by 16 bytes of Product Id).

**DESCRIPTION**

The `driver_addr_probe()` WSIO function is provided by the driver writer. It can have any unique name. If you need this function, you register it with WSIO by executing the `wsio_register_addr_probe()` routine as part of your interface `driver_attach()` routine. Commonly, `driver` is replaced by your driver’s name.

See *HP-UX Driver Development Guide* for details;

**RETURN VALUES**

- **PROBE_SUCCESS** Successfully found something identified it.
- **PROBE_UNSUCCESSFUL** Got to end of the appropriate address range without finding anything, or something went wrong with the probe.

**CONSTRAINTS**

**SEE ALSO**

- `driver_class_probe(WSIO_DRV)`, `wsio_probe_dev_info(WSIO4)`, `wsio_register_probe_func(WSIO3)`, `wsio_register_addr_probe(WSIO3)`
NAME

driver_attach(WSIO_DRV) – Claim a device for a driver.

SYNOPSIS

int driver_attach(uint32_t card_id, struct isc_table_type *isc);

PARAMETERS

card_id A four-byte card or product identifier.
isc A pointer to the ISC structure assigned to the interface node that is being probed.

DESCRIPTION

The driver_attach() WSIO function is provided by the driver writer. It can have any unique name. You pass the name to WSIO Services by including it in an attach chain with the driver’s driver_install() routine. Commonly, driver is replaced by your driver’s name.

The kernel searches the I/O backplane for hardware. When it finds a device, it first does preliminary initialization. Then it calls the driver_attach() routine at the head of the corresponding attach chain, e.g., eisa_attach(), That was created by the driver_install() routine.

Each driver_attach() routine in the chain looks at the card_id. If it recognizes the device as its own, it claims the device with the isc_claim() function, optionally puts a pointer to its driver_if_init() routine in isc->gfsw->init, and performs any other appropriate initialization. Then, whether it claims the device or not, it passes the same parameters to the next routine in the chain, using the routine name it saved in the driver_install() routine.

Since isc_claim() sets the INITIALIZED flag in isc->if_info->flags, you can also test this flag to see if there was a prior claim.

See HP-UX Driver Development Guide for details;
RETURN VALUES

Each driver_attach() routine is expected to return the return value returned by the next driver_attach() routine in the chain. The end-of-chain function returns a unique completion code.

CONSTRAINTS

EXAMPLES

Be very careful with the card_id parameter. It is tempting to just define it as PCI_ID in a PCI driver_attach() routine. Due to 64-bit kernel parameter passing conventions you need to use code similar to the following example in determining that the device is your driver's:

```c
int zzz_attach(uint32_t idparm, struct isc_table_type *isc) {
    PCI_ID *id = (PCI_ID *)&idparm;
    if ((id->vendor_id != MY_VENDOR_ID&&
        (id->device_ID != MY_DEV_ID))
        return (my_saved_attach(idparm, isc);
    else {
        /* code to claim card - set up isr, etc. */
        return (my_saved_attach(idparm, isc);
    }
}
```

SEE ALSO

get_new_isc(WSIO3), isc_table_type(KER4)
NAME

driver_close(WSIO_DRV) – Close a device

SYNOPSIS

#include<sys/conf.h>

int driver_close (dev_t dev, int flag, int mode);

PARAMETERS

dev The device number of the file to be closed. The
driver_close() routine can extract the major and
minor numbers from the device number (see major
(WSIO3) and minor (WSIO3)).

(A user process specifies a file descriptor in the
close() system call.)

flag A value corresponding to the flag field in the
driver_open() call. See driver_open (WSIO_DRV).

mode Determines whether this is a call to a block or char
driver. This parameter is not accessible from a close (2)
call.

DESCRIPTION

The driver_close() WSIO function is provided by the driver writer. It
can have any unique name. You pass the name to WSIO Services by
specifying it in the d_close field of the drv_ops structure. Commonly,
driver is replaced by your driver's name.

See HP-UX Driver Development Guide for details;

RETURN VALUES

The file system function which calls your driver through the bdevsw or
cdevsw tables always returns success (0) to the higher level file system
function which called it, ignoring the return value it gets from your
driver.
Therefore, the driver_close() routine need not return a valid value. However, to avoid problems (as with strict compiler return value checking), the driver_close() routine should return some integer value.

CONSTRAINTS

SEE ALSO

close(2), driver_open (WSIO_DRV), drv_ops (CDIO4), open(2)
NAME

\texttt{driver\_dev\_init(WSIO\_DRV)} – Initialize a device driver

SYNOPSIS

\texttt{int driver\_dev\_init(void);}

PARAMETERS

None.

DESCRIPTION

The \texttt{driver\_dev\_init()} WSIO function is provided by the driver writer. It can have any unique name. You pass the name to WSIO Services by including it in the init chain, \texttt{dev\_init()}, with the \texttt{driver\_install()} routine. Commonly, \texttt{driver} is replaced by your driver's name.

See \textit{HP-UX Driver Development Guide} for details;

RETURN VALUES

Each \texttt{driver\_dev\_init()} routine is expected to return the return value returned by the next \texttt{driver\_dev\_init()} routine in the chain. The end-of-chain function returns a unique completion code.

CONSTRAINTS

SEE ALSO

\texttt{driver\_install(WSIO\_DRV)}
**NAME**

`driver_dev_probe(WSIO_DRV)` – Interface driver dev probing function.

**SYNOPSIS**

```
#include <wsio/wsio.h>
#include <sys/ioparams.h>

int driver_dev_probe (void *this_node, drv_info_t *drv_info,
                      void *probe_id, hw_path_t *hw_path,
                      struct ics_table_type *isc, int probe_type,
                      char *name, char *desc);
```

**PARAMETERS**

- `this_node` A pointer to an `io_tree_node` struct.
- `drv_info` The `drv_into_t` struct registered with `wsio_install()`.
- `probe_id` A unique identifier (for example, first 4 bytes of SCSI Inquiry data).
- `hw_path` A pointer to a structure containing the hardware path information of the module being probed.
- `isc` A pointer to the ISC structure assigned to the interface node that is being probed.
- `probe_type` The type of hardware probe to perform.
  Defined types are:
  - `PROBE_FIRST` Start at first available address.
  - `PROBE_NEXT` Increment the last address and start looking from there.
  - `PROBE_ADDRESS` Look only for this address.
- `name` A string describing the class of the device.
- `desc` A string describing the device found by the probe (usually 8 bytes of Vendor Id followed by 16 bytes of Product Id).
DESCRIPTION

The \texttt{driver\_dev\_probe()} WSIO function is provided by the driver writer. It can have any unique name. If you need one, you register it with WSIO by executing the \texttt{wsio\_register\_dev\_probe()} routine as part of your \texttt{driver\_install()} routine. Commonly, \texttt{driver} is replaced by your driver's name.

See \textit{HP-UX Driver Development Guide} for details;

RETURN VALUES

\begin{verbatim}
PROBE\_SUCCESS Successfully found something and can identify it.
PROBE\_UNSUCCESSFUL Got to end of the appropriate address range without finding anything, or something went wrong with the probe.
\end{verbatim}

CONSTRAINTS

SEE ALSO

\begin{verbatim}
driver\_addr\_probe(WSIO\_DRV), wsio\_register\_addr\_probe(WSIO3),
wsio\_probe\_dev\_info(WSIO4), wsio\_register\_dev\_func(WSIO3)
\end{verbatim}
NAME

driver_if_init(WSIO_DRV) – Initialize interface driver

SYNOPSIS

#include<sys/io.h>

int driver_if_init (struct isc_table_type * isc);

PARAMETERS

isc Pointer to an ISC structure for an interface this driver controls.

DESCRIPTION

The driver_if_init() WSIO function is provided by the driver writer. It can have any unique name. You pass the name to WSIO Services by specifying it in the isc->gfsw->init of the ISC structure with your driver's driver_attach() routine. Commonly, driver is replaced by your driver's name.

See HP-UX Driver Development Guide for details;

RETURN VALUES

0 Successful completion.

-1 Error.

CONSTRAINTS

SEE ALSO

driver_attach(WSIO_DRV), isc_table_type(KER4), isrlink(WSIO3)
NAME
driver_install(WSIO_DRV) – Register a driver with the system

SYNOPSIS

int driver_install(void);

PARAMETERS
None.

DESCRIPTION
The driver_install() WSIO function is provided by the driver writer. The name must be in the format shown, with driver replaced by the name of your driver as you specify it in the system file (defaults to /stand/system) and in the $DRIVER_INSTALL section of a master file in the /usr/conf/master.d directory.
See HP-UX Driver Development Guide for details;

RETURN VALUES

driver_install() is expected to return the value returned by
wsio_install_driver() or install_driver(). Those values are:

0 Failure. The driver was not installed.
1 Success.

If it fails, the appropriate message below appears on the system console and in the system's error-log file. driver is the name of your driver.

wsio_install_driver: Install of driver driver failed.
install_driver: Install of driver driver failed.

CONSTRAINTS
SEE ALSO

config(1M), driver_attach(WSIO_DRV),
driver_dev_init(WSIO_DRV), driver_install(WSIO_DRV),
driver_probe(WSIO_DRV), install_driver(CDIO3), master(4),
wsio_install_driver(WSIO3), wsio_register_addr_probe(WSIO3),
wsio_register_probe_func(WSIO3)
NAME

driver_ioctl(WSIO_DRV) – Execute driver-specific control functions

SYNOPSIS

#include<sys/conf.h>

int driver_ioctl (dev_t dev, int cmd, caddr_t arg_ptr, int flag);

PARAMETERS

dev Device number.

cmd Command word.

arg_ptr Pointer to the command word arguments, if any.

flag File access flags.

DESCRIPTION

The driver_ioctl() WSIO function is provided by the driver writer. It can have any unique name. You pass the name to WSIO Services by specifying it in the d_ioctl field of the drv_ops structure. Commonly, driver is replaced by your driver's name.

See HP-UX Driver Development Guide for details;

RETURN VALUES

0 Successful completion

<> 0 Error. The value is expected to be an error value.

CONSTRAINTS

LP64 CONSIDERATIONS

Pay particular attention to the cmd argument which has different values depending upon the calling program environment. HP-UX Driver Development Guide
See Also

drv_ops(CDIO4), errno(2), ioctl(2), ioctl(5)
NAME

driver_isr(WSIO_DRV) – Execute device interrupt in interrupt context

SYNOPSIS

```c
int driver_isr (isc_table_type isc, int arg1, int arg2);
```

PARAMETERS

- `isc` A pointer to the `isc_table_type` structure passed in with your `isrlink()` call.
- `arg1` First optional parameter passed in with your `isrlink()` call.
- `arg2` Second optional parameter passed in with your `isrlink()` call.

DESCRIPTION

The `driver_isr()` WSIO function is provided by the driver writer. It can have any unique name. You pass the name to WSIO Services by specifying it as a parameter of the `isrlink()` function, executed in your `driver_attach()` or `driver_if_init()` routine. Commonly, `driver` is replaced by your driver's name.

See *HP-UX Driver Development Guide* for details;

RETURN VALUES

- 0 The card does not belong to this driver.
- 1 This routine handled the interrupt.

CONSTRAINTS

SEE ALSO

`isrlink(WSIO3)`
NAME

driver_minor_build(WSIO_DRV) – Build a minor number

SYNOPSIS

#include <wsio/wsio.h>
#include <sys/ioparams.h>

int driver_minor_build (isc_table_type *isc, hw_path_t *dev_path,
                       char *option);

PARAMETERS

isc A pointer to the ISC structure associated with the
     interface card for the device.

dev_path A pointer to a structure containing device hardware
            path information relative to the interface card.

option A null-terminated string indicating device-specific
        options.

DESCRIPTION

The driver_minor_build() WSIO function is provided by the driver
writer. It can have any unique name. You pass the name to WSIO
Services by specifying it in the drv_minor_build field of the
wsio_drv_data_t structure. Commonly, driver is replaced by your
driver's name.

See HP-UX Driver Development Guide for details;

RETURN VALUES

>0 Successful completion. The value is the minor number.

-1 Error. The minor number could not be built.

CONSTRAINTS
SEE ALSO

wsio_drv_data_t(WSIO4)
NAME

\texttt{driver\_minphys(WSIO\_DRV)} \textendash{} Driver specific transfer size adjustment

SYNOPSIS

\begin{verbatim}
#include<sys/buf.h>

void \texttt{driver\_minphys (struct buf *bp);
\end{verbatim}

PARAMETERS

\begin{verbatim}
bp Transfer information structure;
\end{verbatim}

DESCRIPTION

The \texttt{driver\_minphys()} WSIO function is provided by the driver writer. It can have any unique name. You pass the name to \texttt{physio()} by specifying it in the \texttt{mincnt} parameter in the call to \texttt{physio()}. Commonly, \texttt{driver} is replaced by your driver's name.

The \texttt{driver\_minphys()} WSIO function adjusts the \texttt{bp->b\_bcount} field of the \texttt{buf} structure passed in.

RETURN VALUES

\texttt{driver\_minphys()} is a \texttt{void} function.

EXAMPLES

The following example illustrates a \texttt{minphys()} routine for a device that can handle at most \texttt{NBPG} size transfers.

\begin{verbatim}
void mydriver_minphys(struct buf *bp) {
    if (bp->b\_bcount > NBPG)
        bp->b\_bcount = NBPG;
}
\end{verbatim}

CONSTRAINTS
SEE ALSO

minphys(KER2), physio(KER2)
NAME

driver_open(WSIO_DRV) – Open a device

SYNOPSIS

#include<sys/conf.h>

int driver_open (dev_t dev, int oflags intptr_t dev, int mode);

PARAMETERS

dev

The dev_t device number of the file to be opened. The
driver_open() routine can extract the major and
minor numbers from the device number. See major
(WSIO3) and minor (WSIO3).

oflags

A value corresponding to the oflag parameter of the
open() system call. The kernel executes the oflag
functions (described in fcntl (5) and open (2)) before it
calls your driver. Your driver, therefore, can usually
ignore these flags.

Nevertheless, the kernel translates the O_xxxx values
into corresponding F_xxxx values, which it passes to the
driver_open() routine. The flags of possible interest
to your driver include: FREAD, FWRITE, FNDELAY, and
FEXCL.

mode

Whether this is a call to a block or char driver. This
parameter is not accessible from an open (2) call.

dummy

A parameter used as by some drivers, though it is not
accessible from an open (2) call.

DESCRIPTION

The driver_open() WSIO function is provided by the driver writer. It
can have any unique name. You pass the name to WSIO Services by
specifying it in the driver_open field of the drv_ops structure.
Commonly, driver is replaced by your driver's name.

See HP-UX Driver Development Guide for details;
RETURN VALUES

driver_open() is expected to return the following values:

0  Successful completion.
<>0  Error. The value is expected to be an errno value.

If the driver_open() routine is successful, the kernel's open() call returns a file descriptor to the user. If it is unsuccessful, the kernel returns -1 to the user and sets errno to the value returned by the driver_open() routine. The user's process can check the returned value and errno to determine whether an error occurred. See the <errno.h> header file for possible values for errno.

The driver_open() routine should return an error under these conditions. See open(2) for the expected error names.

- The device is off line.
- The device does not exist.
- The device was never configured into the system.
- The initialization of the device failed.
- The device is an exclusive-open device, and it is already open.

CONSTRAINTS

SEE ALSO

drv_ops(CDIO4), open(2)
NAME

driver_psize(WSIO_DRV) – Get swap partition size of a device

SYNOPSIS

#include<sys/conf.h>

int driver_psize (dev_t dev);

PARAMETERS

dev Contains encoded major and minor numbers;

DESCRIPTION

The driver_psize() WSIO function is provided by the driver writer. It can have any unique name. You pass the name to WSIO Services by specifying it in the d_psize field of the drv_ops structure. Commonly, driver is replaced by your driver’s name.

The driver_psize() WSIO function should return the size of the swap partition on a block swapping device, It is called by the kernel. Consider writing this routine only if your device is used for swapping.

See HP-UX Driver Development Guide for details;

RETURN VALUES

>0 Successful completion. The value is the swap partition size.

-1 Error.

CONSTRAINTS

SEE ALSO

drv_ops (CDIO4)
NAME

driver_read(WSIO_DRV) – Read data from/to a character device

SYNOPSIS

#include<sys/conf.h>

int driver_read (dev_t dev, struct uio * uio);

PARAMETERS

dev The device number of the associated device file. The routine can extract the major and minor numbers from the device number. Your driver_open() routine should verify that the minor number is valid.

uio A pointer to a uio structure. The uio structure contains information about the data being read or written.

DESCRIPTION

When a user process issues a read(), readv(), write(), or writev() system call for a character device, the kernel puts information about the request in the uio and iovec structures and dispatches control to the driver_read() or driver_write() routine for that device, passing the uio structure to the driver as a parameter.

See HP-UX Driver Development Guide for details;

RETURN VALUES

0 Successful completion.

<>0 Error. The value is expected to be an errno value.

CONSTRAINTS
EXAMPLES

See physio (KER2) and uiomove (KER2).

SEE ALSO

drv_ops(CDIO4), physio(KER2), uiomove(KER2)
NAME
driver_select(WSIO_DRV) – Test I/O completion on a device

SYNOPSIS
#include<sys/conf.h>

int driver_select (dev_t dev, int flag);

PARAMETERS

    dev  The device number.

    flag The type of readiness to test, according to the following values:

      FREAD       Read
      FWRITE      Write
      0           Exception conditions

DESCRIPTION

The driver_select() WSIO function is provided by the driver writer. It can have any unique name. You pass the name to WSIO Services by specifying it in the d_select field of the drv_ops structure. Commonly, driver is replaced by your driver's name.

See HP-UX Driver Development Guide for details;

RETURN VALUES

<>0 True. The device or driver is ready for read or write or an exception condition was found. The kernel sets the corresponding bit in the bit-mask field that select() returns to the user.

0 False. The device or driver is not ready for read or write or no exception condition was found. select() puts the calling process to sleep until the condition becomes true. The driver must inform the system when this condition becomes true.
If the `driver_select()` routine detects an error while selecting for read or write, it should return false and set an error in `u.u_error`. If it detects an error while selecting for an exception condition, it should return true and set an error in `u.u_error`.

**CONSTRAINTS**

**SEE ALSO**

`drv_ops(CDIO4), selwakeup(KER2), select(2)`
**NAME**

driver_strategy(WSIO_DRV) – Execute block read or write for character or block devices

**SYNOPSIS**

```c
#include<sys/conf.h>

void driver_strategy (struct buf *bp);
```

**PARAMETERS**

- `bp`: Pointer to a `buf` structure.

**DESCRIPTION**

The `driver_strategy()` WSIO function is provided by the driver writer. It can have any unique name. For a block device, you pass the name to WSIO Services by specifying it in the `driver_strategy` field of the `drv_ops` structure. For a character device, you pass the name as a parameter to `physio()`. Commonly, `driver` is replaced by your driver's name.

See *HP-UX Driver Development Guide* for details;

**RETURN VALUES**

- None.

**CONSTRAINTS**

**SEE ALSO**

- `physio(KER2)`
NAME

driver_write(WSIO_DRV) – Write data from/to a character device

SYNOPSIS

#include<sys/conf.h>

int driver_write (dev_t dev, struct uio * uio);

PARAMETERS

dev The device number of the associated device file. The routine can extract the major and minor numbers from the device number. Your driver_open() routine should verify that the minor number is valid.

uio A pointer to a uio structure. The uio structure contains information about the data being read or written.

DESCRIPTION

When a user process issues a read(), readv(), write(), or writev() system call for a character device, the kernel puts information about the request in the uio and iovec structures and dispatches control to the driver_read() or driver_write() routine for that device, passing the uio structure to the driver as a parameter.

See HP-UX Driver Development Guide for details;

RETURN VALUES

0 Successful completion.

<>0 Error. The value is expected to be an errno value.

CONSTRAINTS
SEE ALSO

drv_ops(CDIO4), physio(KER2), uiomove(KER2)
NAME

`free_isc(WSIO3)` – Free a driver's ISC entry

SYNOPSIS

```c
#include <wsio/wsio.h>

int free_isc (struct isc_table_type *isc);
```

PARAMETERS

`isc` Pointer to an ISC entry.

DESCRIPTION

The `free_isc()` kernel function frees an ISC entry that was obtained explicitly by a driver by using `get_new_isc()` or another similar service. `free_isc()` should be called after a severe driver-disabling error or before a driver is unloaded.

RETURN VALUES

- 0 Successful completion.
- -1 Error.

CONSTRAINTS

SEE ALSO

`get_new_isc(WSIO3)`
NAME

get_new_isc(WSIO3) – Allocate a new ISC structure for this card function

SYNOPSIS

#include <sys/io.h>

struct isc_table_type * get_new_isc (struct isc_table_type * dd_isc);

PARAMETERS

dd_isc Pointer to a currently allocated ISC structure.

RETURN VALUES

CONSTRAINTS

DESCRIPTION

The get_new_isc() WSIO function allocates a new ISC structure when you need more than one, as for a multifunction card.

If the isc->ftn_no field is not -1 in an entry for a multifunction card, the driver_attach() routine should call get_new_isc() to allocate a new ISC structure for the driver's functions and set the isc->ftn_no field of the new ISC structure to the function number for its portion of the card, and then continue its normal power-on initializations, using the new ISC structure that was returned from get_new_isc().

Pass the new ISC on to the next driver in the attach chain.

The get_new_isc() function allocates and zeros out a new ISC structure and then does the following:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>bus_type</td>
<td>Copied from old_isc</td>
</tr>
<tr>
<td>my_isc</td>
<td>Copied from old_isc</td>
</tr>
</tbody>
</table>
get_new_isc(WSIO3)

if_reg_ptr  Copied from old_isc
bus_info    Copied from old_isc
ftn_no      Set to -1, the caller should correctly set this field after call
old_isc->next_ftn  Set to the new isc
if_info     Allocated and then copied from old_isc
new->next_ftn Set to NULL
ifsw        Copied from old_isc
if_drv_data Copied from old_isc
gfsw        Allocated and copied from old_isc if old_isc->gfsw is not NULL

RETURN VALUES
<>NULL     Success. The value is a pointer to a new ISC structure.
NULL       Failure. get_new_isc() was unable to allocate memory for the new ISC structure.
NAME
iodone(WSIO3) – Complete the buffer I/O transaction

SYNOPSIS

```
#include <sys/buf.h>

int iodone (struct buf * bp);
```

PARAMETERS

bp Pointer to a buf structure.

DESCRIPTION

The iodone() WSIO function is used by legacy drivers as an alias for biodone(). New drivers should call biodone() directly instead of calling iodone().

The biodone() kernel function completes the buffer I/O transaction. There should be a corresponding call to biowait() for the same bp.

If B_CALL is set in bp->b_flags, biodone() calls the callback function specified in bp->b_iiodone. The callback function is expected to set the B_DONE flag in bp->b_flags.

If B_CALL is not set in bp->b_flags, biodone() marks the buffer I/O as completed by setting the B_DONE flag in bp->b_flags. If B_ASYNC is set, biodone() releases the buf structure and associated buffer pointed to by bp, else it resumes the thread waiting on the corresponding call to biowait().

RETURN VALUES

None.

CONSTRAINTS

Must not be called while holding a spinlock of order >=
BUF_HASH_LOCK_ORDER.
WARNINGS

biodone() calls panic() if B_DONE is set in bp->b_flags upon entry.

SEE ALSO

biodone(KER2), biowait(KER2), iowait(WSI03)
NAME

iowait(WSIO3) – Wait for the buffer I/O to complete.

SYNOPSIS

#include <sys/buf.h>

int iowait (struct buf * bp);

PARAMETERS

bp Pointer to a buf structure.

DESCRIPTION

The iowait() WSIO function is used by legacy drivers as an alias for biowait(). New drivers should call biowait() directly instead of calling iowait().

The biowait() kernel function waits for the completion of the buffer I/O specified by bp. A corresponding call to biodone() is required to resume the waiting thread.

RETURN VALUES

0 Must not be called in an interrupt context.
<>0 Error.

CONSTRAINTS

Must not be called in an interrupt context.
Must not be called while holding a spinlock.
EXAMPLES

```c
int error;
struct buf *bp;

error = iowait(bp);

/*
 * iowait() returns 0 if the IO completes successfully.
 * A non-zero value is returned if an error has been
 * encountered, however, the error value returned is not
 * always for the IO completion. To get the IO
 * completion error that is returned with the buf, we
 * need to call geterror().
 */
if (error) {
    error = geterror(bp);
}
```

SEE ALSO

biodone (KER2), biowait (KER2), geterror (KER2), iodone (WSIO3)
NAME

isc_claim(WSIO3) – Marks an ISC entry as claimed by the driver.

SYNOPSIS

```
#include <sys/io.h>
#include <wsio/wsio.h>

void isc_claim (struct isc_table_type *isc, wsio_drv_info_t drv_info);
```

PARAMETERS

- **isc**
  Pointer to the ISC entry associated with an interface card or device.

- **drv_info**
  Pointer to the `wsio_drv_info_t` structure of the driver that is claiming the ISC structure.

DESCRIPTION

The `isc_claim()` WSIO function marks an ISC entry as claimed by the driver. `isc_claim()` is called in the `driver_attach` function when the driver wants to be assigned to the device represented by the ISC entry.

If `drv_info` is `NULL`, the driver is indicating the ISC entry should be discarded. An example of this situation is when the PS2 keyboard driver encounters its second ISC entry. Since the driver only uses the first ISC entry, it can claim and discard the second ISC entry by passing `NULL` as `drv_info`.

RETURN VALUES

None.

CONSTRAINTS
EXAMPLE

static int
mydrv.attach(uint32_t id, struct isc_table_type *isc)
{
    ...  
    if (id == MY_DEVICE_HW_ID) {
        /*
         * Specify the interface init function that is
         * called for each claimed ISC entry after the
         * attach chain processing has completed.
         */
        isc->gfsw->init = mydrv_if_init;
    }

    /*
    * Claim the ISC entry representing the device.
    */
    isc_claim(isc, &mydrv_info);
}

/*
 * Call the next driver on the attach chain.
 */
return (*mydrv_saved_attach)(id, isc);

SEE ALSO

driver_attach(WSIO_DRV)
NAME

isc_table_type(WSIO4) – ISC table entry structure

SYNOPSIS

#include <sys/io.h>

DESCRIPTION

Each interface card (each device claimed by an interface driver) has an associated Interface Select Code (ISC) entry. The contents of an ISC entry are declared as the isc_table_type structure in <sys/io.h>.

WSIO uses the pointer to an ISC entry as the handle to an interface card. Many WSIO functions require the pointer to an ISC as an argument. WSIO also passes card-specific information to an interface driver through fields in the ISC entry.

Interface drivers use the ISC to store driver-specific information. Some fields in the ISC entry are defined by system and drivers are expected to use these fields as intended by the system. Other fields are available to the driver for its internal use.

The I/O Switch Tables

The I/O system supports two I/O interface switch tables through fields in the ISC structure. isc->gfsw is intended for use by the system. isc->ifsw is available to specify communication between device and interface drivers.

Generic Function Switch

The generic function switch, isc->gfsw, is intended for system-to-interface driver communication. The table consists of pointers to two function routines:

- isc->gfsw->init points to a driver-defined interface initialization routine that is called after all calls to driver_attach() functions have been made.
- isc->gfsw->diag points to a driver-defined interface diagnostic routine. Its usage is currently not implemented, and drivers must set this pointer to NULL.
Interface Function Switch

The interface function switch, `isc->ifsw`, is intended for device-to-interface driver communication. It is through this table that a device driver calls its corresponding interface driver. The table is defined and initialized by the interface driver.

For an example of an interface switch table, see the `scsi_ifsw` structure in `<wsio/scsi_ctl.h>`.

STRUCTURE MEMBERS

The following is a list of driver accessible fields in the ISC structure that are initialized by the system.

Table 4-1 Driver Relevant `isc_table_type` Structure Fields

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td><code>bus_type</code></td>
</tr>
<tr>
<td>int</td>
<td><code>if_id</code></td>
</tr>
<tr>
<td><code>caddr_t</code></td>
<td><code>if_info</code></td>
</tr>
<tr>
<td>int</td>
<td><code>if_info-&gt;flags</code></td>
</tr>
<tr>
<td><code>caddr_t</code></td>
<td><code>if_reg_ptr</code></td>
</tr>
</tbody>
</table>

- **`bus_type`**: Type of I/O bus for the interface card. For example, `PCI_BUS` for PCI interface cards.
- **`if_id`**: Hardware ID of the interface card. The contents of this field are `bus_type` dependent.
- **`if_info`**: Pointer to a `wsio_if_info` structure declared in `<sys/wsio.h>`. 
if_info->flags

Flags indicating the result of an interface driver attempting to claim an interface card. Currently defined values are:

HAS_IOCHKERR  The card has an I/O check error.
INITIALIZED  An interface driver's attach routine has successfully initialized the card. This flag is set by the isc_claim() function.
INIT_ERROR  An interface driver's attach routine tried to claim this card, but failed.
IS_ISA_CARD  This card is an ISA card.
NO_ALLOC_GDD  Reserved.
SLOT_ADDR_VALID  Reserved.

if_reg_ptr

Pointer to the base of the interface card’s memory mapped registers. After initial configuration, i.e., when your interface driver's attach routine is called, if your driver claims the card you may use this field as desired. For example, if your PCI driver memory space is not mapped due to size constraints, you can call map_mem_to_host() and store the returned virtual address from that call in this field.

The following is a list of fields in the ISC structure that are initialized by drivers.

Table 4-2  Driver Initialized ISC Fields

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>volatile int *</td>
<td>card_ptr</td>
</tr>
<tr>
<td>int (*)(struct isc_table_type *)</td>
<td>gfsw-&gt;init</td>
</tr>
<tr>
<td>caddr_t</td>
<td>if_drv_data</td>
</tr>
<tr>
<td>void *</td>
<td>if_isc</td>
</tr>
<tr>
<td>caddr_t</td>
<td>ifsw</td>
</tr>
</tbody>
</table>
### Table 4-2  Driver Initialized ISC Fields (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>my_address</td>
</tr>
<tr>
<td>struct buf *</td>
<td>owner</td>
</tr>
<tr>
<td>unsigned int</td>
<td>state</td>
</tr>
</tbody>
</table>

- **card_ptr**: Pointer to a range of memory mapped interface card registers.
- **gfsw->init**: Pointer to the `init` function for the interface driver. The system initializes `gfsw` to point to a generic function switch table. The interface driver is responsible for updating the table.
- **if_drv_data**: Pointer to a driver specified object.
- **if_isc**: Pointer to a driver specified object.
- **ifsw**: Interface driver entry-point switch. It is set by the interface driver and is intended to be an operational interface between a device driver and its interface driver.
- **my_address**: Can be used as desired. Usually contains the interface card's bus address.
- **owner**: Can be used as desired. Usually contains a pointer to the active `buf` or I/O request.
- **state**: Can be used as desired. Usually contains the device state information.
NAME

m_wsio_funcnum(WSIO3) – Get the number of an interface card function

SYNOPSIS

#include<sys/wsio.h>

int m_wsio_funcnum (dev_t dev, wsio_drv_info_t * drv_hdr_ptr);

PARAMETERS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dev</td>
<td>The dev_t number of a device.</td>
</tr>
<tr>
<td>drv_hdr_ptr</td>
<td>Pointer to the wsio_drv_info_t structure for the device.</td>
</tr>
</tbody>
</table>

DESCRIPTION

The m_wsio_funcnum() WSIO function returns the number of the interface card function associated with device number dev.

RETURN VALUES

CONSTRANTS

SEE ALSO
NAME

m_wsio_selcode(WSIO3) – Get the select code for a device

SYNOPSIS

#include<sys/wsio.h>

int m_wsio_selcode (dev_t dev, wsio_drv_info_t * drv_hdr_ptr);

PARAMETERS

dev                  The dev_t number of a device.
drv_hdr_ptr          Pointer to the wsio_drv_info_t structure for the device.

DESCRIPTION

The m_wsio_selcode() WSIO function returns the select code associated with device number dev.

RETURN VALUES

CONSTRAINTS

SEE ALSO
NAME

m_wsio_vsc(WSIO3) – Return the system bus module number for a device number

SYNOPSIS

#include<sys/wsio.h>

int m_wsio_vsc (dev_t dev, wsio_drv_info_t * drv_hdr_ptr);

PARAMETERS

dev The dev_t number of a device.
drv_hdr_ptr Pointer to the wsio_drv_info_t structure for the device.

DESCRIPTION

The m_wsio_vsc() WSIO function returns the system bus module number for device number dev.

RETURN VALUES

CONSTRAINTS

SEE ALSO
mod_wsio_attach_list_add(WSIO3) – Add the driver attach function pointer to the specified WSIO attach list.

SYNOPSIS

```c
int mod_wsio_attach_list_add (int type, void *attach_func);
```

PARAMETERS

- `type` Type of WSIO attach list.
- `attach_func` Pointer to the driver attach function.

DESCRIPTION

The `mod_wsio_attach_list_add()` WSIO function adds the driver attach function pointer `attach_func` to the WSIO attach list specified by `type`.

Dynamically loadable drivers call `mod_wsio_attach_list_add()` when they are loaded into the kernel.

The `type` parameter specifies the attach list to use. Valid values are:

- `MOD_WSIO_CORE` for Core I/O attach list.
- `MOD_WSIO_EISA` for EISA I/O attach list.
- `MOD_WSIO_PCI` for PCI I/O attach list.

The `attach_func` parameter points to the driver attach function that will be called by the WSIO environment to see if the driver wants to claim a device.

RETURN VALUES

- 0 Successful completion.
- 1 Error.

CONSTRAINTS
EXAMPLE

/*
 * Add my driver attach function to the WSIO attach list
 * for claiming PCI devices.
 */
if (mod_wsio_attach_list_add(MOD_WSIO_PCI,
                            &mydrv_attach)) {
    return ENXIO; /* attach add failed! */
}

SEE ALSO

mod_wsio_attach_list_remove(WSIO3).
NAME

`mod_wsio_attach_list_remove` (WSIO3) – Remove the driver attach function pointer to the specified WSIO attach list.

SYNOPSIS

```c
int mod_wsio_attach_list_remove (int type, void *attach_func);
```

PARAMETERS

- **type**: Type of WSIO attach list.
- **attach_func**: Pointer to the driver attach function.

DESCRIPTION

The `mod_wsio_attach_list_remove` WSIO function removes the driver attach function pointer `attach_func` from the WSIO attach list specified by `type`. The `attach_func` and `type` parameters must match the parameters passed to `mod_wsio_attach_list_add()`.

Dynamically loadable drivers call `mod_wsio_attach_list_remove()` when they are unloaded from the kernel.

The `type` parameter specifies the attach list to use. Valid values are:

- `MOD_WSIO_CORE` for Core I/O attach list.
- `MOD_WSIO_EISA` for EISA I/O attach list.
- `MOD_WSIO_PCI` for PCI I/O attach list.

The `attach_func` parameter points to the driver attach function.

RETURN VALUES

- **0**: Successful completion.
- **1**: Error.

CONSTRAINTS
EXAMPLE

/*
 * Remove my driver attach function from the WSIO attach list
 * for claiming PCI devices.
 */
if (mod_wsio_attach_list_remove(MOD_WSIO_PCI,
    &mydrv_attach)) {
    return ENXIO;    /* attach remove failed! */
}

SEE ALSO

mod_wsio_attach_list_add(WSIO3).
NAME

wsio_activate_probe(WSIO3) – Activate the probe function for a driver.

SYNOPSIS

void wsio_activate_probe (char * probe_name,
                        struct drv_info * drv_infop);

PARAMETERS

probe_name Name of the device probe function as registered by
            wsio_register_dev_probe()

drv_infop Pointer to the driver drv_info structure

DESCRIPTION

The wsio_activate_probe() WSIO function connects the probe
function for a dynamically loadable interface driver to the driver
drv_info structure. wsio_activate_probe() is called in the driver’s
load entry point after its device probe function has been registered with
the WSIO CDIO.

RETURN VALUES

None

CONSTRAINTS
EXAMPLES

static wsio_drv_info_t mydrv_info = { ... };

int mydrv_load(void * arg)
{
    /*
    * Use the drv_info passed to to the driver as arg
    * instead of using the static version.
    */
    mydrv_info.drv_info = (drv_info_t *)arg;

    /*
    * Register the driver with WSIO.
    * Note: returns 0 on failure.
    */
    if (!wsio_install_driver(&mydrv_info)) {
        return ENXIO;    /* Install driver failed! */
    }

    /*
    * Add my driver attach function to the WSIO attach list
    * for claiming PCI devices.
    */
    if (mod_wsio_attach_list_add(MOD_WSIO_PCI,
                                  &mydrv_attach)) {
        /*
        * Attach list add failed! Uninstall the driver
        * and return.
        */
        (void)wsio_uninstall_driver(&mydrv_info);
        return ENXIO;
    }

    /*
    * Register the device probe function for the driver.
    */
    if (wsio_register_dev_probe(IF_CLASS, mydrv_probe_func,
                                "mydrv_probe")) {
        /*
        * Register device probe failed! Remove driver from
        * the attach list, uninstall the driver and return.
        */
        (void)mod_wsio_attach_list_remove(MOD_WSIO_PCI,
                                           &mydrv_attach);
        (void)wsio_uninstall_driver(&mydrv_info);
return ENXIO;
}

/*
 * The following step is only required for dynamically
 * loadable drivers: connect the probe function.
 */
wsio_activate_probe("mydrv_probe", mydrv_info.drv_info);

return 0;

SEE ALSO

wsio_register_dev_probe(WSIO3),
wsio_unregister_dev_probe(WSIO3)
NAME

wsio_alloc_mem(WSIO3) – Service to allocate memory for DMA buffers or control structures.

SYNOPSIS

wsio_alloc_status_t
wsio_alloc_mem ( wsio_mem_handle_t mem_handle,
size_t size,
wsio_vaddr_t *vaddr,
wsio_alloc_flags_t flags)

PARAMETERS

mem_handle A handle allocated by a call to
wsio_alloc_mem_handle
size The size of the buffer.
vaddr A pointer to the allocated buffer if successful else NULL
flags Flags which describe restrictions

WSIO_SLEEP_OK Flag to indicate service can sleep if
waiting for resources

DESCRIPTION

This WSIO service can be called by drivers to allocate memory for DMA buffers or control structures. The first parameter to the service must be a mem_handle that was allocated by the driver using
wsio_alloc_mem_handle.

RETURN VALUES

WSIO_ALLOC_OK Indicates that the buffer was allocated
WSIO_ALLOC_OUT_OF_RESOURCES Unable to allocate the specified resources
WSIO Reference Pages

wsio_alloc_mem(WSIO3)

CONSTRAINTS

EXAMPLES

SEE ALSO
NAME

wsio_alloc_mem_handle(WSIO3) – Service to specify the type of memory to allocate.

SYNOPSIS

wsio_alloc_status_t
wsio_alloc_mem_handle ( struct isc_table_type *isc,
wsio_mem_handle_t *mem_handle,
wsio_mem_alloc_attrib_t attribs)

PARAMETERS

isc A pointer to the device's isc structure
mem_handle A pointer to a variable where the handle will be stored upon completion.
attribs Attributes describing the criteria for the type of memory to allocate.

WSIO_OPTIMIZE_FOR_DEVICE
Allocate memory close to the device. On Half-Dome systems, memory will be allocated on the same cell as the device.

WSIO_OPTIMIZE_FOR_CPU
Allocate memory close to the current CPU. On Half-Dome systems, memory will be allocated on the same cell as the CPU. This is the default behavior.

WSIO_32BIT_MEMORY
The buffer must be allocated below 4G.

WSIO_IO_CONTIGUOUS
On platforms without an IOpdir physically contiguous memory will be allocated.
DESCRIPTION

Drivers will call this service to specify the type of memory they want to allocate. The service will return a `mem_handle` which can be passed into the WSIO memory alloc and free routines. Drivers can allocate more than one `mem_handle` to specify different criteria for memory allocation.

While it is not a requirement, it is recommended that drivers call this routine early in their initialization sequence. This is due to the high overhead of the routine. Buffer alignment is as follows:

<table>
<thead>
<tr>
<th>Allocation Size</th>
<th>Buffer Aligned On</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less Than Cacheline Size</td>
<td>32 Byte Boundary</td>
</tr>
<tr>
<td>Greater Than or Equal to Cacheline Size</td>
<td>Cacheline Boundary</td>
</tr>
<tr>
<td>Greater Than or Equal to I/O Page Size (4K)</td>
<td>4K Boundary</td>
</tr>
</tbody>
</table>

RETURN VALUES

- **WSIO_ALLOC_OK**
  - Indicates that the buffer was allocated
- **WSIO_ALLOC_OUT_OF_RESOURCES**
  - Unable to allocate the specified resources
- **WSIOINVAL_PARAM**
  - A parameter was not valid

CONSTRAINTS

None

EXAMPLES
SEE ALSO
NAME

wsio_allocate_dma_handle(WSIO3) – Obtain a handle used to setup DMA

SYNOPSIS

#include <wsio/wsio.h>

void *wsio_allocate_dma_handle (struct isc_table_type *isc);

PARAMETERS

isc Pointer to the driver's isc_table entry.

DESCRIPTION

The wsio_allocate_dma_handle() WSIO function is called by device drivers to obtain a DMA handle. This handle, which is passed to all DMA services, can be associated with various DMA hints, and is used to control DMA. Multiple handles can be allocated, allowing a device driver to associate different hints with each handle.

RETURN VALUES

A void pointer to the handle. If NULL is returned, a handle count could not be allocated.

CONSTRAINTS

EXAMPLE

void *dma_handle;

DMA handle = wsio_allocate_dma_handle(isc_entry);
if (dma_handle == NULL {
   /* No handle allocated. */
   return (ERROR);
});
/* The DMA handle is now in the dma_handle variable */
SEE ALSO

wsio_allocate_shared_mem(WSIO3), wsio_dma_pass_thru(WSIO3),
wsio_fastmap_dma_buffer(WSIO3), wsio_free_dma_handle(WSIO3),
wsio_free_shared_mem(WSIO3), wsio_flush_shared_mem(WSIO3),
wsio_init_map_context(WSIO3), wsio_iowa_to_phys(WSIO3),
wsio_map_dma_buffer(WSIO3), wsio_remap_dma_buffer(WSIO3),
wsio_set_device_attributes(WSIO3),
wsio_set_dma_attributes(WSIO3), wsio_unmap_dma_buffer(WSIO3)
NAME

wsio_allocate_shared_mem(WSIO3) – Set up an I/O virtually contiguous DMA buffer.

SYNOPSIS

#include <wsio/wsio.h>

wsio_map_status_t wsio_allocate_shared_mem (
   struct isc_table_type *isc, void *dma_handle,
   size_t size, wsio_iova_t *iova, wsio_vaddr_t *vaddr,
   wsio_shared_mem_attr_t shared_mem_attr);

PARAMETERS

isc          Pointer to the driver’s isc_table entry.
dma_handle   DMA handle allocated using
             wsio_allocate_dma_handle().
size         Size of buffer to allocate.
iova         Pointer that contains the I/O virtual address upon
             completion. A wsio_iova_t must be allocated by the
             driver, and the pointer to this is what should be passed
             into the macro.
vaddr         Pointer that contains the host virtual address upon
              completion. A wsio_vaddr_t must be allocated by the
              driver, and the pointer to this is what should be passed
              into the macro.
shared_mem_attr
             Bitmask that indicates how to allocate the memory.
             The acceptable values are described in the following
             list. If a type of 0 is used, the default behavior of
             WSIO_IO_SHMEM_OPTIMIZE_DEVICE_LATENCY is used.
             The behavior of the allocation is also affected by
             attributes set using
             wsio_dma_set_device_attributes(), and
             wsio_set_dma_attributes().
The following are the `wsio_shared_mem_attr_t` allowable bitmask values.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>WSIO_IO_SHMEM_OPTIMIZE_DEVICE_LATENCY</code></td>
<td>Allocation should optimize for device access latency. If possible, allocate object in memory local to a bus bridge.</td>
</tr>
<tr>
<td><code>WSIO_IO_SHMEM_OPTIMIZE_HOST_LATENCY</code></td>
<td>Allocation should optimize for host access latency. If possible, allocate in host memory.</td>
</tr>
<tr>
<td><code>WSIO_IO_SHMEM_DMA_ALLOC_COMPATIBLE</code></td>
<td>Behave exactly as the 10.X <code>dma_alloc</code> service. This type is for compatibility with 10.X only.</td>
</tr>
<tr>
<td><code>WSIO_IO_SHMEM_INBOUND</code></td>
<td>This attribute can be OR'ed with the other attributes to indicate the buffer is used exclusively for inbound DMA.</td>
</tr>
<tr>
<td><code>WSIO_IO_SHMEM_OUTBOUND</code></td>
<td>This attribute can be OR'ed with the other attributes to indicate the buffer is used exclusively for outbound DMA.</td>
</tr>
<tr>
<td><code>WSIO_IO_SHMEM_DEV_WEAK_OK</code></td>
<td>This attribute can be OR'ed with the other attributes to indicate the accesses to the buffer can be weakly ordered. The default ordering is the strongest that can be provided for the given I/O bus.</td>
</tr>
<tr>
<td><code>WSIO_IO_SHMEM_ALIGN_ON_SIZE</code></td>
<td>This attribute can be OR'ed with the other attributes to specify <code>size</code> also indicates the alignment boundary for the allocation.</td>
</tr>
</tbody>
</table>
DESCRIPTION

The `wsio_allocate_shared_mem()` WSIO function is called by a device driver to allocate an I/O virtually contiguous DMA buffer that is to be used for continuous DMA. Continuous DMA means that the memory appears contiguous to the I/O device and can be read or written by the I/O device on a continuous basis. For packet DMA or DMA that is used for temporary mappings, `wsio_map_dma_buffer()` should be used.

If a callback function is set up (see `wsio_set_dma_callback()`), and no resources are available when the call is made, `WSIO_MAP_W_CALLBACK` will be returned, and the callback will be triggered when resources become available.

This macro may be called in a non-blocking context.

RETURN VALUES

- **WSIO_MAP_OK**  
  Success.

- **WSIO_MAP_W_CALLBACK**  
  Returned if no resources are available and a callback is registered.

- **WSIO_MAP_E_NO_RESOURCES**  
  Returned if no resources are available and no callback is registered.

- **WSIO_MAP_E_RESOURCE_ERROR**  
  Returned if cannot allocate resources. If this is returned, the allocation will never succeed.

- **WSIO_MAP_E_PARAMETER_ERROR**  
  Returned on bad parameter (Software bug).

- **WSIO_MAP_E_UNKNOWN_ERROR**  
  Returned if there is an unknown error.

CONSTRAINTS
EXAMPLE

```c
void *dma_handle = NULL;
wsio_iova_t io_virtual_addr;
wsio_vaddr_t host_virtual_addr;

dma_handle = wsio_allocate_dma_handle(isc_entry);
if (dma_handle == NULL) {
    /* No handle allocated. */
    return ERROR;
}

/* The DMA handle is now in the dma_handle variable */

if (wsio_allocate_shared_mem(isc_entry, dma_handle, buf_size, &io_virtual_addr, &host_virtual_addr, 0) != WSIO_MAP_OK) {
    /* Unable to allocate shared memory, so return an error */
    return ERROR;
}

/* A buffer of size 'buf_size' is now allocated and mapped
* in both host virtual memory and I/O virtual memory space.
*/
```

SEE ALSO

wsio_allocate_dma_handle(WSIO3), wsio_dma_pass_thru(WSIO3),
wsio_free_dma_handle(WSIO3), wsio_free_shared_mem(WSIO3),
wsio_flush_shared_mem(WSIO3), wsio_iova_to_phys(WSIO3),
wsio_set_device_attributes(WSIO3),
wsio_set_dma_attributes(WSIO3)
NAME

WSIO_BIG_ENDIAN(WSIO3) – Function to return true (1) if the local bus is big-endian

SYNOPSIS

#include <wsio/wsio.h>

int WSIO_BIG_ENDIAN (struct isc_table_type *isc);

PARAMETERS

isc Pointer to the driver's isc_table entry.

DESCRIPTION

The WSIO_BIG_ENDIAN() macro is called by a device driver to report whether the local bus is big-endian. If it is, true is returned, otherwise it returns false. This can be used by a driver along with the known endianness of the host processor to decide whether endian swapping should be performed. Endian swapping might be necessary for any data transfers between the I/O bus and local host memory.

WSIO_BIG_ENDIAN() can be called in a non-blocking context.

RETURN VALUES

1 Local bus is big-endian.
0 Not big-endian

CONSTRAINTS

EXAMPLE

if (WSIO_BIG_ENDIAN(isc_entry)) {
    /* No endian swapping necessary */
} else {
    /* Endian swapping must be performed */
}
SEE ALSO

WSIO_LITTLE_ENDIAN(WSIO3)
NAME

`wsio_cfg_inXX()` (WSIO3) – Macros to read from configuration space.

SYNOPSIS

```c
#include <wsio/wsio.h>

void wsio_cfg_inXX (struct isc_table_type *isc,
                    wsio_addr_handle_t cfg_handle,
                    uint32_t offset, uintXX_t *data);
```

PARAMETERS

- `isc` Pointer to the driver's isc_table entry.
- `cfg_handle` Configuration space handle.
- `offset` Byte offset into the configuration space.
- `data` Pointer to an appropriately sized and aligned memory space for the returned data.

DESCRIPTION

The `wsio_cfg_inXX()` macros are called by device drivers to read from configuration space. The `cfg_handle` and the `offset` are used to specify the correct location to read from. The value ‘XX’ refers to 8, 16, 32, or 64 and indicates the amount of data to read from configuration space. Endian translation is performed automatically if the host memory and local bus have different endianness.

RETURN VALUES

None

CONSTRAINTS
EXAMPLE

```c
wsio_addr_handle_t handle;
uint32_t data;

if (wsio_map_cfg_handle(isc_entry,&handle) != WSIO_OK) {
    return(ERROR);
}
wsio_cfg_in32(isc_entry,handle,offset,&data);

/* 'data' will now contain whatever was at 'offset' in
 * configuration space
 */
```

SEE ALSO

wsio_cfg_outXX(WSIO3), wsio_map_cfg_handle(WSIO3),
wsio_unmap_cfg(WSIO3)
NAME

`wsio_cfg_outXX` (WSIO3) – Macros for writing to configuration space

SYNOPSIS

```c
#include <wsio/wsio.h>

void wsio_cfg_outXX (struct isc_table_type *isc,
    wsio_addr_handle_t cfg_handle,
    uint32_t offset, uintXX_t data)
```

PARAMETERS

- `isc`  
  Pointer to the driver's `isc_table` entry.
- `cfg_handle`  
  Configuration handle.
- `offset`  
  Byte offset into the configuration space.
- `data`  
  Pointer to an appropriately sized and aligned memory space for the returned data.

DESCRIPTION

The `wsio_cfg_outXX()` macros are called by device drivers to write to configuration space. The `cfg_handle` and the `offset` are used to specify the correct location to write to.

The value ‘XX’ refers to 8, 16, 32, or 64 and indicates the amount of data to write to configuration space. Endian translation is performed automatically if host memory and the local bus have different endianness.

RETURN VALUES

None

CONSTRAINTS
EXAMPLE

wsio_addr_handle_t handle;
uint32_t data = 0x5a;

if (wsio_map_cfg_handle(isc_entry,&handle) != WSIO_OK) {
    return(ERROR);
}
wsio_cfg_out32(isc_entry,handle,offset,data);

/* 0x5a will now be at 'offset' in configuration space */

SEE ALSO

wsio_cfg_inXX(WSIO3), wsio_map_cfg_handle(WSIO3),
wsio_unmap_cfg(WSIO3),
NAME

wsio_dma_pass_thru(WSIO3) – Call a DMA pass-thru function that might not otherwise be accessible.

SYNOPSIS

#include <wsio/wsio.h>

intptr_t wsio_dma_pass_thru (struct isc_table_type *isc,
   void *dma_handle,
   wsio_pt_type_t pass_thru_type,
   intptr_t pass_thru_param);

PARAMETERS

isc Pointer to the driver’s isc_table entry.

dma_handle DMA handle allocated using
   wsio_allocate_dma_handle().

pass_thru_type Indicates which pass-thru function to call. Two types are defined as shown in the following items:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSIO_MAP_PT_DEVICE_LOCK</td>
<td>Prevents a device’s access to shared memory. This can be used so that processors can access host memory atomically, and can be used for synchronization. The parameter’s pass_thru_param indicates whether shared memory should be locked (1) or unlocked (0). Implementation of this function is not required, so the return code is zero (0) if the function is implemented or non-zero if not implemented.</td>
</tr>
</tbody>
</table>
WSIO Reference Pages

wsio_dma_pass_thru(WSIO3)

WSIO_MAP_PT_SYNC_BUS
Causes any FIFOs, buffers, or I/O caches associated with a device to be synchronized with memory. The parameter must be zero (0). This function returns zero (0) if the function is implemented and non-zero if it is not.

pass_thru_param
Parameter that will get passed into the pass-thru function as an argument.

DESCRIPTION
The wsio_dma_pass_thru() WSIO function is present to allow new DMA interfaces to be added without breaking binary compatibility. It allows such interfaces to be accessed if they are present.

RETURN VALUES
Dependent on the specific pass-thru function being called.

CONSTRAINTS

EXAMPLE

if (wsio_dma_pass_thru(isc_entry,dma_handle, WSIO_MAP_PT_SYNC_BUSSES,0) !=0) {
    /* The function isn’t implemented */
    return(ERROR);
} else {
    /* The function is implemented, and completed correctly */
    return(0);
}
/* This code will attempt to sync memory associated with * dma_handle */
SEE ALSO

wsio_allocate_dma_handle(WSIO3),
wsio_allocate_shared_mem(WSIO3),
wsio_fastmap_dma_buffer(WSIO3), wsio_flush_shared_mem(WSIO3),
wsio_free_dma_handle(WSIO3), wsio_free_shared_mem(WSIO3),
wsio_init_map_context(WSIO3), wsio_ioga_to_phys(WSIO3),
wsio_map_dma_buffer(WSIO3), wsio_remap_dma_buffer(WSIO3),
wsio_set_device_attributes(WSIO3),
wsio_set_dma_attributes(WSIO3), wsio_unmap_dma_buffer(WSIO3)
NAME

wsio_dma_set_device_attributes(WSIO3) – Associate DMA hints with a device

SYNOPSIS

#include <wsio/wsio.h>

wsio_map_status_t wsio_dma_set_device_attributes
    (struct isc_table_type *isc,
     void *dma_handle,
     wsio_dma_attribute_t attribute
     wsio_dma_attr_param_t param);

PARAMETERS

isc Pointer to the driver's isc_table entry.
dma_handle DMA handle allocated using
wsio_allocate_dma_handle().
attribute Indicates which hint to set for the device associated
with dma_handle. The possible attributes follow:

- WSIO_DMA_ATTR_ADDR_WIDTH
  Bits of addressing supported by the
device. This is used to determine
whether a device can DMA directly to
memory buffers.
  Default value = 32

- WSIO_DMA_ATTR_ALIGNMENT
  Byte alignment of DMA buffer
required for device.
  Default value = HW Dep.
WSIO Reference Pages
wsio_dma_set_device_attributes(WSIO3)

WSIO_DMA_ATTR_ATM
ATM hint. Used by hardware in some implementations.

0 = not ATM
1 = ATM48 (optimize for 48-byte transfers)
2 = ATM192 (optimize for 192-byte transfers)

Default value = 0

WSIO_DMA_ATTR_CALLBACK
Specifies a function to call when resources become available.

Default value = NULL

WSIO_DMA_ATTR_CALLBACK_ARG
Specifies an argument to the callback function.

Default value = 0

WSIO_DMA_ATTR_FLUSH_ON_USE
Specifies the cacheline should be flushed from any intermediate buffers as soon as it is referenced. This inhibits any coalescing of data by bus bridges.

Default value = 0

WSIO_DMA_ATTR_IGN_ALIGN
Specifies the mapping service should not handle cacheline fragments in a special way.

Default value = 0
WSIO_DMA_ATTR_INTERLEAVE
   Iova allocation model
   
   0 = DMA streams are normally interleaved (mass-storage).
   1 = DMA streams are normally not interleaved (networking)
   2 = DMA buffers are static and accessed randomly (low fat).

   Default value = 0

WSIO_DMA_ATTR_PREFETCH
   Specifies how aggressively hardware should prefetch for outbound DMA.
   
   0 = no prefetch
   1 = moderate prefetch
   2 = aggressive prefetch

   Default value = 1

WSIO_DMA_ATTR_SAFE
   Specifies the most conservative coherency model should be used for inbound DMA. Inhibits semicohorent transactions such as WRITE_PURGE unless it is guaranteed that no data in processor caches will be lost.
   
   1 = ON
   2 = OFF

   Default value = 0

WSIO_DMA_ATTR_TXN_SIZE
   Specifies the default transaction size used by the device. This is used by hardware to optimize conversion of transactions between buses.

   Default value = HW Dep.

WSIO_DMA_ATTR_INBOUND
   DMA buffers will be used exclusively for inbound DMA.

   Default value = 0
The `wsio_dma_set_device_attributes()` WSIO function is used to associate DMA transaction hints and attributes with a specific device. These hints are overridden by any hints set for a specific DMA handle via `wsio_set_dma_attributes()`, or some hints passed in as parameters to `wsio_map_dma_buffer()`.

`wsio_dma_set_device_attributes()` can be called in a non-blocking context. If `WSIO_DMA_ATTR_INTERLEAVE` is set to 1, a subsequent `wsio_allocate_shared_mem()` or `wsio_map_dma_buffer()` can only successfully request a buffer with a maximum of one page (4K) in length and this buffer cannot cross a page boundary. This is the limitation placed by the underlying platform. If a larger buffer is desirable, use the default value of 0. This larger buffer can be used for control structures rather than packet DMAs.

Do not call `wsio_dma_set_device_attributes()` to set the `WSIO_DMA_ATTR_INTERLEAVE` to the default value of 0. The call will fail.

**RETURN VALUES**

- `WSIO_MAP_OK` Success.
- `WSIO_MAP_E_PARAMETER_ERROR` Returned if an invalid parameter has caused failure of the call.
CONSTRAINTS

EXAMPLE

```c
if (wsio_dma_set_device_attributes(isc_entry, dma_handle,
    WSIO_DMA_ATTR_INTERLEAVE, 1) != WSIO_MAP_OK) {
    /* There was a parameter error */
    return(ERROR);
} else {
    /* DMA streams are now not normally interleaved for all DMA
        * associated with this device
    */
    return(0);
}
```

SEE ALSO

wsio_allocate_dma_handle(WSIO3),
wsio_allocate_shared_mem(WSIO3), wsio_dma_pass_thru(WSIO3),
wsio_fastmap_dma_buffer(WSIO3), wsio_free_dma_handle(WSIO3),
wsio_free_shared_mem(WSIO3), wsio_flush_shared_mem(WSIO3),
wsio_init_map_context(WSIO3), wsio_iova_to_phys(WSIO3),
wsio_map_dma_buffer(WSIO3), wsio_remap_dma_buffer(WSIO3),
wsio_set_dma_attributes(WSIO3), wsio_unmap_dma_buffer(WSIO3),
NAME

*wsio_drv_data_t*(WSIO4) – Driver-specific fields for WSIO drivers

SYNOPSIS

```
#include<sys/wsio.h>
```

DESCRIPTION

The *wsio_drv_data_t* WSIO structure type, defined in `<wsio/wsio.h>`, contains driver-specific fields for WSIO drivers.

STRUCTURE MEMBERS

<table>
<thead>
<tr>
<th>Field</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>drv_path</strong></td>
<td>Follow these guidelines:</td>
</tr>
<tr>
<td></td>
<td>❑ For device drivers, <em>drv_path</em> is typically a string that contain the interface card's type and the device's class. For example, <code>scsi_disk</code>.</td>
</tr>
<tr>
<td></td>
<td>❑ For interface drivers, <em>drv_path</em> should match the card's type. For example, <code>scsi</code>.</td>
</tr>
<tr>
<td></td>
<td>❑ For pseudo drivers, <em>drv_path</em> should match the card's class. For example, <code>graphics</code>.</td>
</tr>
<tr>
<td><strong>drv_type</strong></td>
<td>One of the following values:</td>
</tr>
<tr>
<td>T_INTERFACE</td>
<td>The driver controls an interface card.</td>
</tr>
<tr>
<td>T_DEVICE</td>
<td>The driver controls a hardware device.</td>
</tr>
<tr>
<td><strong>drv_flags</strong></td>
<td>One of the following values:</td>
</tr>
<tr>
<td>DRV_CONVERGED</td>
<td>The driver meets the HP-UX Release 10.0 Converged I/O specifications. All new drivers should meet these specifications.</td>
</tr>
<tr>
<td>NOT_CONVERGED</td>
<td>The driver conforms to the pre-Release 10.0 unconverged specifications.</td>
</tr>
</tbody>
</table>


**WSIO Reference Pages**

**wsio_drv_data_t(WSIO4)**

**drv_minor_build**

   Pointer to your minor number formatter. Use **NULL** if you don’t provide one.

**drv_minor_decode**

   Pointer to your minor number interpreter. Use **NULL** if you don’t provide one.

**EXAMPLES**

```c
static wsio_drv_data_t sdisk_data = {
    "scsi_disk",
    T_DEVICE,
    DRV_CONVERGED,
    NULL,
    NULL,
};
```

**SEE ALSO**
NAME

wsio_drv_info(WSIO4) – Structure containing pointers to other CDIO and WSIO data structures

SYNOPSIS

#include <wsio/wsio.h>

DESCRIPTION

The wsio_drv_info_t WSIO structure type, defined in <wsio/wsio.h>, contains pointers to three other data structures.

STRUCTURE MEMBERS

drv_info Pointer to a drv_info_t CDIO structure.
drv_ops Pointer to a drv_ops_t CDIO structure.
drv_data Pointer to a wsio_drv_data_t structure.
driver_version

Set to WSIO_DRV_CURRENT_VERSION.

SEE ALSO

drv_info(CDIO4), drv_ops(CDIO4), wsio_drv_data_t(WSIO4)
NAME

wsio_event_t(WSIO5) – An enumeration of WSIO associated events

SYNOPSIS

#include <wsio/wsio.h>

DESCRIPTION

This structure enumerates the WSIO associated events. When the driver
handler is invoked for a WSIO event, the argument passed to the driver
handler is wsio_generic_event_t type. The event field in the
argument structure indicates the event for which the driver handler is
invoked, as described in the Example section.

STRUCTURE MEMBERS

typedef enum {
    WSIO_NO_EVENT,
    WSIO_EVENT_SUSPEND,
    WSIO_EVENT_RESUME,
    WSIO_EVENT_REMOVE,
    WSIO_EVENT_DEV_ERROR,
    WSIO_EVENT_BUS_ERROR,
    WSIO_EVENT_SELF_TEST,
    WSIO_EVENT_LBI_INTR_MIGR,
    WSIO_EVENT_OFFLINE_CPU,
    WSIO_EVENT_ONLINE_CPU
} wsio_event_t;

EXAMPLE

If the driver has a transaction based interrupt, the driver is notified if
that interrupt is being reassigned to a new CPU. The
wsio_generic_event_t structure is filled with the following
information:

    event: WSIO_EVENT_OFFLINE_CPU
    event_id: WSIO provided event_id
isc Pointer to the instance of the associated driver \texttt{isc\_table\_type} structure

\texttt{wsio\_completion\_cb} WSIO provided call back

\texttt{arg} Pointer to a structure of type \texttt{wsio\_intr\_migr\_t}

Refer to the Interrupt Migration chapter of the \textit{Driver Development Guide} for relevant information.

\textbf{SEE ALSO}

\texttt{wsio\_drv\_event\_t(WSIO5)},
\texttt{wsio\_install\_drv\_event\_handler(WSIO3)},
\texttt{wsio\_reg\_drv\_capability\_mask(WSIO5)}
NAME

wsio_fastmap_dma_buffer (WSIO3) – Function to map an existing memory object for packet DMA.

SYNOPSIS

#include <wsio/wsio.h>

wsio_map_status_t wsio_fastmap_dma_buffer (  
  struct isc_table_type *isc, void *dma_handle,  
  wsio_range_type_t range_type,  
  wsio_dma_map_t *host_range,  
  wsio_dma_map_t *io_range);

PARAMETERS

isc Pointer to the driver’s isc_table entry.

dma_handle DMA handle allocated using  
  wsio_allocate_dma_handle().

range_type Indicates the type of host memory being mapped. It can be:

  KERNELSPACE Indicates host_range is a kernel virtual buffer.

  PHYSICAL Indicates host_range is a physical buffer.

  > 0 Indicates host_range is in user space, and this will be the space ID of the virtual address.

host_range Pointer to an address/length structure that contains information about the host space to map. If the mapping was only partially completed, this will contain information about the remaining space to be mapped when the call completes.

io_range Pointer to an address/length structure that will contain information about the I/O space that was mapped.
DESCRIPTION

The `wsio_fastmap_dma_buffer()` macro is called by a device driver to map an existing memory object for packet DMA. It operates in the same way as `wsio_map_dma_buffer()` except the entire host address range must reside on a single physical page. If the range cannot be mapped in a single call, an error will be returned. Cacheline fragments are ignored (the same behavior as `WSIO_DMA_IGN_ALIGNMENT`).

The size of a buffer can have a maximum length of one page (4K) and this buffer can not cross a page boundary.

All mappings will remain in effect until `wsio_unmap_dma_buffer()` or `wsio_remap_dma_buffer()` are called to remove or change them.

Device drivers can set up a callback routine that will come into play if resources are not available at the time a mapping is attempted. If this callback is set up, `WSIO_MAP_W_CALLBACK` will be returned instead of a no resource error. When resources become available, the callback routine will be called to indicate this to the device driver. For more information on how to setup and use a callback, see the `wsio_set_dma_callback()` manpage. `wsio_fastmap_dma_buffer()` can be called in a non-blocking context.

RETURN VALUES

`WSIO_MAP_OK` Returned if the entire buffer has been mapped.

`WSIO_MAP_W_CALLBACK` Returned if no resources are available and a callback function exists.

`WSIO_MAP_E_NO_RESOURCES` Returned if no resources are available and no callback function exists.

`WSIO_MAP_E_RESOURCE_ERROR` Returned if the request cannot and will never succeed.

`WSIO_MAP_E_HIGH_ADDR` Returned if the call failed because the device cannot reach the destination address.

`WSIO_MAP_E_PARAMETER_ERROR` Returned if an invalid parameter has caused failure of the call.
WSIO_MAP_E_UNKNOWN_ERROR
Returned for hardware or other errors.

CONSTRAINTS

EXAMPLE

void *dma_handle;
wsio_dma_map_t host_range, io_range;

dma_handle = wsio_allocate_dma_handle(isc_entry);

host_range.iov_base = host_virtual_address;
host_range.iov_len = dma_buffer_length;

if (wsio_fastmap_dma_buffer(isc_entry,dma_handle,
KERNELSPACE,&host_range, &io_range)!= WSIO_MAP_OK) {
    /* Unable to map the range, so return an error */
    return(ERROR);
}

/* The host virtual buffer represented in the above code by
* 'host_virtual_address' is now mapped. */

SEE ALSO

wsio_allocate_dma_handle(WSIO3),
wsio_allocate_shared_mem(WSIO3), wsio_dma_pass_thru(WSIO3),
wsio_free_dma_handle(WSIO3), wsio_free_shared_mem(WSIO3),
wsio_flush_shared_mem(WSIO3), wsio_init_map_context(WSIO3),
wsio_iova_to_phys(WSIO3), wsio_map_dma_buffer(WSIO3),
wsio_remap_dma_buffer(WSIO3),
wsio_set_device_attributes(WSIO3),
wsio_set_dma_attributes(WSIO3), wsio_unmap_dma_buffer(WSIO3),
NAME

wsio_flush_shared_mem(WSIO3) – Flush an I/O virtually contiguous DMA buffer.

SYNOPSIS

#include <wsio/wsio.h>

wsio_map_status_t wsio_flush_shared_mem ( 
    struct isc_table_type *isc, void *dma_handle, 
    size_t size, wsio_iova_t iova, wsio_vaddr_t vaddr, 
    wsio_shared_mem_attr_t shared_mem_attr);

PARAMETERS

isc Pointer to the driver's isc_table entry.

dma_handle DMA handle allocated using wsio_allocate_dma_handle.

size Size of buffer to be flushed.

iova I/O virtual address of the shared memory.

vaddr Host virtual address of the shared memory.

shared_mem_attr Bitmask that was used to allocate the shared memory.

DESCRIPTION

The wsio_flush_shared_mem() WSIO function is called by a device driver to guarantee the consistency of the memory object allocated via wsio_allocate_shared_mem(). Any non-coherent buffers associated with the memory object are flushed. All parameters to the function should be the same as those passed to the call that allocated the memory. wsio_flush_shared_mem() can be called in a non-blocking context.

RETURN VALUES

WSIO_MAP_OK Success.

WSIO_W_NOP The call has no effect. The caller need not call it again.
**CONSTRAINTS**

**EXAMPLE**

```c
void *dma_handle = NULL;
wsio_iova_t io_virtual_addr;
wsio_vaddr_t host_virtual_addr;

dma_handle = wsio_allocate_dma_handle(isc_entry);
if (dma_handle == NULL) {
    /* No handle allocated. */
    return(ERROR);
}
/* The DMA handle is now in the dma_handle variable */

if (wsio_allocate_shared_mem(
    isc_entry, dma_handle, buf_size, &io_virtual_addr,
    &host_virtual_addr, 0) != WSIO_MAP_OK) {
    /* Unable to allocate the shared memory,
     * so return an error
     */
    return(ERROR);
}
/* A buffer of size 'buf_size' is now allocated and mapped in
 * both host virtual memory and I/O virtual memory space.
 */
wsio_flush_shared_mem(isc_entry, dma_handle, buf_size,
    io_virtual_addr, host_virtual_addr, 0);
```

**SEE ALSO**

- wsio_allocate_dma_handle(WSIO3),
- wsio_allocate_shared_mem(WSIO3), wsio_dma_pass_thru(WSIO3),
- wsio_free_dma_handle(WSIO3), wsio_free_shared_mem(WSIO3),
- wsio_iova_to_phys(WSIO3), wsio_set_device_attributes(WSIO3),
- wsio_set_dma_attributes(WSIO3)
NAME

\texttt{wsio\_free\_dma\_handle} (WSIO3) – Release a DMA handle.

SYNOPSIS

\begin{verbatim}
#include <wsio/wsio.h>

void wsio_free_dma_handle (struct isc_table_type *isc,
                          void *dma_handle);
\end{verbatim}

PARAMETERS

- \texttt{isc} Pointer to the driver’s \texttt{isc\_table\_entry}.
- \texttt{dma\_handle} Pointer to the DMA handle to free.

DESCRIPTION

The \texttt{wsio\_free\_dma\_handle}() WSIO function is called by device drivers
to release a handle that has been allocated by
\texttt{wsio\_allocate\_dma\_handle}(). It should be called anytime a handle is
no longer needed.

RETURN VALUES

None.

CONSTRAINTS
EXAMPLE

```c
void *dma_handle;

dma_handle = wsio_allocate_dma_handle(isc_entry);
if (dma_handle == NULL) {
    /* No handle allocated. */
    return(ERROR);
}
/* The DMA handle is now in the dma_handle variable */

wsio_free_dma_handle(isc_entry, dma_handle);
/* The DMA handle has now been released */
```

SEE ALSO

wsio_allocate_dma_handle(WSIO3),
wsio_allocate_shared_mem(WSIO3), wsio_dma_pass_thru(WSIO3),
wsio_fastmap_dma_buffer(WSIO3), wsio_free_shared_mem(WSIO3),
wsio_flush_shared_mem(WSIO3), wsio_init_map_context(WSIO3),
wsi0_i6o v a_t o p h y s (WSIO3), wsio_map_dma_buffer(WSIO3),
wsi0_remap_dma_buffer(WSIO3),
wsi0_set_device_attributes(WSIO3),
wsi0_set_dma_attributes(WSIO3), wsio_unmap_dma_buffer(WSIO3),

NAME

`wsio_free_mem(WSIO3)` – Frees memory allocated by `wsio_alloc_mem`.

SYNOPSIS

```c
void wsio_free_mem (wsio_mem_handle_t mem_handle,
                    wsio_vaddr_t vaddr)
```

PARAMETERS

- `mem_handle` A handle allocated by a call to `wsio_alloc_mem_handle`.
- `vaddr` A pointer to the allocated buffer.

DESCRIPTION

This WSIO service is called to free memory allocated by the service `wsio_alloc_mem()`.

RETURN VALUES

None.

CONSTRAINTS


EXAMPLE


SEE ALSO
NAME

wsio_free_mem_handle(WSIO3) – Destroy handle previously allocated by
wsio_alloc_mem_handle.

SYNOPSIS

    void wsio_free_mem_handle (wsio_mem_handle_t mem_handle)

PARAMETERS

    mem_handle A handle allocated by a call to
                wsio_alloc_mem_handle.

DESCRIPTION

    Drivers call this service to destroy a mem_handle that was allocated by a
    previous call to wsio_alloc_mem_handle.

RETURN VALUES

    None.

CONSTRAINTS

EXAMPLE

SEE ALSO
NAME

wsio_free_shared_mem(WSIO3) – Release an I/O virtually contiguous DMA buffer.

SYNOPSIS

#include <wsio/wsio.h>

void wsio_free_shared_mem (struct isc_table_type *isc,
    void *dma_handle, size_t size,
    wsio_iova_t iova, wsio_vaddr_t vaddr,
    wsio_shared_mem_attr_t shared_mem_attr);

PARAMETERS

isc Pointer to the driver's isc_table entry.
dma_handle DMA handle allocated using
    wsio_allocate_dma_handle().
size Size of buffer to be released.
iova I/O virtual address of the shared memory.
vaddr Host virtual address of the shared memory.
shared_mem_attr Bit mask that was used to allocate the shared memory.

DESCRIPTION

The wsio_free_shared_mem() WSIO function is called by a device
driver to release an I/O virtually contiguous DMA buffer that was
allocated by wsio_allocate_shared_mem(). All parameters to the
macro should be the same as those passed to the call that allocated the
memory.

wsio_free_shared_mem() can be called in a non-blocking context.

RETURN VALUES

None
CONSTRANTS

EXAMPLE

```c
void *dma_handle = NULL;
wsio_iova_t io_virtual_addr;
wsio_vaddr_t host_virtual_addr;

dma_handle = wsio_allocate_dma_handle(isc_entry);
if (dma_handle == NULL) {
    /* No handle allocated. */
    return(ERROR);
}

/* The DMA handle is now in the dma_handle variable */

if (wsio_allocate_shared_mem(isc_entry, dma_handle, buf_size,
               &io_virtual_addr,
               &host_virtual_addr, 0)
    != WSIO_MAP_OK) {
    /* Unable to allocate shared memory,
     * so return an error */
    return(ERROR);
}

/* A buffer of size 'buf_size' is now allocated and mapped 
* in both host virtual memory and I/O virtual memory space. 
*/
wsio_free_shared_mem(isc_entry, dma_handle, buf_size,
               io_virtual_addr, host_virtual_addr, 0);
```

SEE ALSO

wsio_allocate_dma_handle(WSIO3),
wsio_allocate_shared_mem(WSIO3), wsio_dma_pass_thru(WSIO3),
wsi_free_dma_handle(WSIO3), wsio_flush_shared_mem(WSIO3),
wsi_iova_to_phys(WSIO3), wsio_set_device_attributes(WSIO3),
wsi_set_dma_attributes(WSIO3)
NAME

wsio_generic_event_t(WSIO) – Generic WSIO event information

SYNOPSIS

#include <wsio/wsio.h>

DESCRIPTION

This is a generic WSIO event information data structure. Any event in the system which is of relevance to a driver is managed through this data structure.

STRUCTURE MEMBERS

struct wsio_generic {
    wsio_event_t event;
    wsio_event_id_t event_id;
    struct isc_table_type *isc;
    generic_complete_callback_t wsio_completion_cb;
    void *arg;
} wsio_generic_event_t;

The fields in the structure are:

- **event**
  - A structure of type `wsio_event_t`, which indicates the event.

- **event_id**
  - A WSIO provided `event_id`.

- **isc**
  - Pointer to the `isc(struct isc_table_type)` structure for the instance of the driver.

- **wsio_completion_cb**
  - A WSIO provided completion callback routine. Once the driver processes the event, this is the completion callback to WSIO.

- **arg**
  - An event related argument. This could be used in the context of the event and might provide further information relevant to the event.
EXAMPLE

If the driver has a transaction based interrupt, the driver is notified if that interrupt is being reassigned to a new CPU. The wsio_generic_event_t structure is filled with the following information:

- **event**: WSIO_EVENT_OFFLINE_CPU
- **event_id**: WSIO provided event_id
- **isc**: Pointer to the instance of the associated driver isc_table_type structure.
- **wsio_completion_cb**: WSIO provided call back.
- **arg**: Pointer to a structure of type wsio_intr_migt_t

If a driver is using a transaction based interrupt, the driver must register for the WSIO_EVENT_OFFLINE_CPU event. Refer to the Interrupt Migration chapter in the *Driver Development Guide* for related information.

SEE ALSO

- wsio_drv_event_t(WSIO5),
- wsio_install_drv_event_handler(WSIO3),
- wsio_reg_drv_capability_mask(WSIO5)
NAME

wsio_get_active_processor_count(WSIO3) – Get the number of active CPUs in the system.

SYNOPSIS

#include <wsio/wsio.h>

int wsio_get_active_processors_count(void);

PARAMETERS

DESCRIPTION

The wsio_get_active_processor_count() WSIO function returns the number of CPUs currently active. On a running system the active CPU number may change. For example, if a processor is deallocated or allocated.

RETURN VALUES

The number of active processors.

CONSTRAINTS

EXAMPLE

int active_cpus;

active_cpus = wsio_get_active_processor_count();
printf("number of active CPUs %d\n", active_cpus);

SEE ALSO

wsio_get_processor_count(WSIO3)
NAME

wsio_get_all_registers(WSIO3) – Get an array of all available device registers

SYNOPSIS

#include <wsio/wsio.h>

wsio_reg_info_t * wsio_get_all_registers (struct isc_table_type * isc);

PARAMETERS

isc

Pointer to the driver’s isc_table entry.

DESCRIPTION

The wsio_get_all_registers() WSIO function obtains an array of all of the registers for the device associated with the isc_table_type entry that is passed in. This function assumes that the device driver will know how many registers will be returned in the array and what their uses are.

RETURN VALUES

This function returns the address of an array of structures of type wsio_reg_info_t. If no registers exist, NULL will be returned.

CONSTRAINTS
EXAMPLE

```c
wsio_reg_info_t *registers;

registers = wsio_get_all_registers(isc_entry);
if (registers == NULL) {
    /* No registers exist. Return an error. */
    return(ERROR);
}

/* All of the devices registers are now contained in the
 * 'registers' variable. They may be mapped as follows:
 * /
if (wsio_map_reg(isc_entry,&registers[1]) != WSIO_OK) {
    return(ERROR);
}

/* The second device register (index 1 into the array) will
 * now be mapped.
 * /
```

SEE ALSO

wsio_map_reg(WSIO3), wsio_read_regXX(WSIO3),
wsio_unmap_reg(WSIO3), wsio_write_regXX(WSIO3)
NAME

`wsio_get_ioports(WSIO)` – Obtain the addresses and sizes of I/O ports.

SYNOPSIS

```c
#include <wsio/wsio.h>

int wsio_get_ioports (struct isc_table_type *isc,
                      int cnt, wsio_iop_t port_array[]);
```

PARAMETERS

- `isc`  
  Pointer to the driver’s `isc_table` entry.
- `cnt`  
  Indicates the maximum number of ports to return.
- `port_array`  
  Array to store the ports in.

DESCRIPTION

The `wsio_get_ioports()` WSIO function gets up to `cnt` I/O ports and stores them in an array. All ports will be obtained if the `cnt` variable is large enough. The driver must allocate enough space for all ports inside the `port_array`. Once these ports have been obtained, they can be mapped using `wsio_map_port()` and accessed using `wsio_port_inXX()`, and `wsio_port_outXX()`.

RETURN VALUES

- `WSIO_OK`  
  Successful completion.
- `WSIO_ERROR`  
  There was a parameter error.

CONSTRAINTS
EXAMPLE

```c
wsio_addr_handle_t port_handle;
wsio_iop_t ioports_array[10];
/* An array with enough space for all ports needs to
 * be allocated */

if (wsio_get_ioports(isc_entry,10,ioports_array
     ) != WSIO_OK) {
    /* There was a problem obtaining the ports */
    return(ERROR);
}

if (wsio_map_port(isc_entry,ioports_array[0].addr,
     ioports_array[0].size, &port_handle
     ) != WSIO_OK) {
    /* There was an error mapping the port */
    return(ERROR);
}

/* Now unmap the port */
if (wsio_unmap_port(isc_entry,ioports_array[0].addr,
     ioports_array[0].size, port_handle
     ) != WSIO_OK) {
    /* There was an error unmapping the port */
    return(ERROR);
}
```

SEE ALSO

wsio_map_port(WSIO3), wsio_port_inXX(WSIO3),
wsio_port_outXX(WSIO3)
NAME

*wsio_get_isc*(WSIO3) – Get the ISC structure pointer for a device file.

SYNOPSIS

```c
#include <sys/io.h>
#include <wsio/wsio.h>

int wsio_get_isc (dev_t dev, struct isc_table_type **isc_ptr,
                  wsio_drv_info_t *wsio_drv_info);
```

PARAMETERS

- `dev` The device file of the hardware for which we want an ISC pointer.
- `isc_ptr` A pointer to the location for the routine to put a pointer to the ISC structure.
- `wsio_drv_info` A pointer to the `wsio_drv_info_t` header structure of the driver, used to decode `dev`. If a NULL value is passed in this field, `wsio_get_isc()` will use the `wsio_drv_info_t` structure of the character device (not block) whose major number matches that of the `dev` argument.

DESCRIPTION

The `wsio_get_isc()` WSIO function finds an ISC table entry associated with the nearest interface ancestor of the device specified by `dev`, assuming that `dev` uses the HP-UX Converged I/O minor number format.

NOTE

Drivers modified to match the Converged I/O driver guidelines will automatically have the old `get_isc()` call mapped to this one when they include the `wsio.h` header file. Drivers that have not been modified for Release 10.0 and do not include `wsio.h` will be assumed to use the old minor number format and the old `get_isc()`.
Modified drivers that still call the old `get_isc()` will work only for drivers that pass their `character dev` values. If a driver has only its `block dev`, it must call `wsio_get_isc()` directly.

### RETURN VALUES

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Successful completion. The ISC pointer found is returned in <code>isc_ptr</code>.</td>
</tr>
<tr>
<td>0</td>
<td>Failure. The ISC could not be found.</td>
</tr>
</tbody>
</table>

### CONSTRAINTS

### SEE ALSO
NAME

wsio_get_processor_count(WSIO3) – Get the number of CPUs in the system.

SYNOPSIS

#include <wsio/wsio.h>

int wsio_get_processor_count(void)

PARAMETERS

DESCRIPTION

The wsio_get_processor_count() WSIO function returns the number of CPUs configured into the system. On a running system the number of CPUs may change. For example, if a processor is deallocated or allocated.

RETURN VALUES

The number of processors.

CONSTRAINTS

EXAMPLE

int num_cpus;

num_cpus = wsio_get_processor_count();

printf("number of CPUs %d\n", num_cpus);

SEE ALSO

wsio_get_active_processor_count(WSIO3)
NAME
wsio_get_system_params(WSIO3) – Get information about the system.

SYNOPSIS

#include <wsio/wsio.h>

int wsio_get_system_params_t * wsio_get_system_params(
    struct isc_table_type *isc, wsio_get_sys_parm_t id,
    wsio_unintptr_t *parm);

PARAMETERS

isc Pointer to the driver's isc_table entry.
id An identifier indicating what parameter to get.
parm Value returned will be written to a memory location pointed to by parm.

id parm

WSIO_CACHELINE_SIZE Indicates the cacheline size in bytes

WSIO_DEFAULT_PAGE_SIZE Indicates the default page size in bytes.

WSIO_DMA_64BIT_ADDRESSING Indicates 64-bit addressing capability (1), or not capable (0).

WSIO_DMA_COHERENT_IO Indicates IO coherent (1), or not coherent (0).

WSIO_DMA_IOPDIR_PRESENT Indicates IOPDIR is present (1) or not present (0).

WSIO_NUM_CPUS Indicates the number of CPUs on the system.
DESCRIPTION

The `wsio_get_system_params()` WSIO function obtains all system parameters that are currently defined. The `id` is to identify what parameter to retrieve. The value that the caller is interested in will be stored into the memory location pointed to by `parm`.

RETURN VALUES

- **WSIO_OK**  
  Successful completion.
- **WSIO_ERROR**  
  Error.

CONSTRAINTS

EXAMPLE

```c
int ret;
wsio_uintptr_t value;
ret = wsio_get_system_params(isc_entry,
                             WSIO_CACHELINE_SIZE, &value);

/* The cacheline size will be obtained and put into value */
```

SEE ALSO
NAME

\texttt{wsio\_init\_map\_context(WSIO3)} – Initialize the context used for DMA mapping.

SYNOPSIS

\begin{verbatim}
#include <wsio/wsio.h>

void wsio_init_map_context (wsio_map_context_t *context);
\end{verbatim}

PARAMETERS

context \hspace{1cm} Pointer to the context to be initialized.

DESCRIPTION

The \texttt{wsio\_init\_map\_context()} WSIO macro is called by device drivers to initialize a context that is needed by \texttt{wsio\_map\_dma\_buffer()}, and other map related functions. A context is used internally by mapping services so that system resources can be used efficiently. The same context should be used for a set of DMA mappings that are all going to be unmapped at the same time. This context should be initialized before its first use. If the mapping that is being performed is independent of all other mappings, then a context is not necessary and \texttt{NULL} should be passed to the mapping function in the context field.

RETURN VALUES

None

CONSTRAINTS
EXAMPLE

```c
wsio_map_context_t dma_context;

wsio_init_map_context(&dma_context);
/* dma_context is now ready to be used for mapping. It can
 * be used for any number of mappings as long as they are
 * all unmapped at the same time
 */
```

SEE ALSO

wsio_allocate_dma_handle(WSIO3), wsio_dma_pass_thru(WSIO3),
wsio_fastmap_dma_buffer(WSIO3), wsio_free_dma_handle(WSIO3),
wsio_iova_to_phys(WSIO3), wsio_map_dma_buffer(WSIO3),
wsio_remap_dma_buffer(WSIO3),
wsio_set_device_attributes(WSIO3),
wsio_set_dma_attributes(WSIO3), wsio_unmap_dma_buffer(WSIO3).
NAME

wsio_install_driver(WSIO3) – Install a driver's header structure into the WSIO CDIO.

SYNOPSIS

#include<sys/wsio.h>

int wsio_install_driver (void * wsio_drv_info);

PARAMETERS

wsio_drv_info Pointer to the driver's wsio_info_t structure.

DESCRIPTION

The wsio_install_driver() WSIO function installs a driver's header structure into the WSIO CDIO.

RETURN VALUES

1 Successful completion.

0 Error. The major number specified for the driver is already in use. The following message is displayed on the system console and in the error-log file:

wsio_install_driver:
Install of driver driver failed.

CONSTRAINTS


EXAMPLES

/* Declare the driver entry points */
static drv_ops_t beep_ops = {
    beep_open, /* open */
    beep_close, /* close */
    NULL, /* strategy */
    NULL, /* dump */
    NULL, /* psize */
    NULL, /* mount */
    NULL, /* read */
    NULL, /* write */
    beep_ioctl, /* ioctl */
    NULL, /* select */
    NULL, /* option_1 */
    NULL, /* reserved1 */
    NULL, /* reserved2 */
    NULL, /* reserved3 */
    NULL, /* reserved4 */
    0 /* flag */
};

/* Declare the CDIO driver-specific fields */
/* Flags DRV_CHAR/DRV_BLOCK/DRV_PSEUDO/DRV_SCAN/DRV_MP_SAFE/
   DRV_SAFE_CONF */
static drv_info_t beep_info = {
    "beep", /* char *name for device type */
    "graf_pseudo", /* char *name for device class */
    DRV_PSEUDO|DRV_CHAR,/*ubit32 flags pseudo? block? char? scan? */
    -1, /* int b_major maj dev# if block type */
    168, /* int c_major maj dev# if char type */
    NULL, /* struct cdio *cdio drivers set to NULL */
    NULL, /* void *gio_private drivers set to NULL */
    NULL /* void *cdio_private drivers set to NULL */
};

/* Declare the WSIO driver-specific fields */
static wsio_drv_data_t beep_data = {
    "hil", /* char *drv_path match probes-drivers */
    T_DEVICE, /* sbit8 drv_type type of H/W: dev or IF */
    DRV_CONVERGED,
    /* ubit32 drv_flags DRV_CONV... or NOT_C... */
    NULL,
    /* int (*drv_minor_build)() minor# formatter */
    NULL
/* int (*drv_minor_decode)() interpreter */
}
static wsio_drv_info_t beep_wsio_info = {
    &beep_info, /* drv_info_t *drv_info */
    &beep_ops, /* drv_ops_t *drv_ops driver entry points */
    &beep_data /* wsio_drv_data_t *drv_data */
};

beep_install()
{
    /* register driver with WSIO and return any error */
    return( wsio_install_driver( beep_wsio_info ) );
}

SEE ALSO

install_driver(CDIO3)
NAME

wsio_install_drv_event_handler(WSIO3) – Install a driver's event handler

SYNOPSIS

#include <wsio/wsio.h>

int wsio_install_drv_event_handler (wsio_drv_info_t * drv_info,
                                   void (* drv_handler)(wsio_generic_event_t * generic_ptr);

PARAMETERS

drv_info Pointer to the driver's wsio_drv_info_t structure.
drv_handler Function pointer to a driver's event handler

DESCRIPTION

The wsio_install_drv_event_handler() WSIO function is called by a device driver to register an event handler to deal with events. When an event occurs, such as a suspend request, WSIO will invoke this handler. This function should be called in a driver's installation routine after wsio_install_driver() is executed.

RETURN VALUES

WSIO_OK Returned on success.
WSIO_DRV_NOT_FOUND An earlier call to wsio_install_driver() was not successful.
WSIO_HANDLER_NULL Drv_handler is a NULL pointer.
WSIO_INFO_NULL Drv_info is a NULL pointer.

CONSTRAINTS
EXAMPLE

```c
static wsio_drv_info_t my_drv_info {
    &my_info,
    &my_ops,
    &my_data,
    WSIO_DRV_CURRENT_VERSION,
    /* This is a macro defined in wsio.h, bearing a version
    * stamp */
}

my_driver_install() {
    if (wsio_install_driver(&my_drv_info) != 1) {
        return (0); /* 0 means error */
    }
    /* Register the event handler with WSIO */
    if (wsio_install_drv_event_handler(
            &my_drv_info, my_handler) != WSIO_OK) {
        /* The example given shows that wsio_uninstall_driver()
        * is called. In this case, the driver
        * will not be brought up. If a driver chooses to
        * ignore the error and proceed, it will
        * run without being able to handle events */
        wsio_unregister_driver(&my_drv_info);
        return (0); /* 0 means error */
    }
}
```

SEE ALSO

- wsio_query_supported_function(WSIO_DRV),
- wsio_reg_drv_capability_mask(WSIO5),
- wsio_unregister_drv_event_handler(WSIO3)
NAME

wsio_intr_activate(WSIO5) – Enable an interrupt object.

SYNOPSIS

#include <wsio/wsio.h>

int wsio_intr_activate (struct isc_table_type *isc, 
                      wsio_intr_object_t obj);

PARAMETERS

isc Pointer to the driver's isc_table entry.

obj Interrupt object to enable.

DESCRIPTION

The wsio_intr_activate() WSIO function activates an interrupt object 
that was allocated with wsio_intr_alloc(). The interrupt object must 
be activated before the system will call the device driver's ISR (as 
specified in wsio_intr_alloc()). It is assumed that (if possible) the 
device will not generate interrupts until after this function is called. The 
wsio_intr_deactivate() or wsio_intr_deactivate_nowait() service 
undoes the effects of this function.

Attempting to activate an interrupt object that is already active is an 
error condition that returns WSIO_INTR_ACTIVATED, without modifying 
the interrupt object.

If interrupt migration software is present in the system, the following 
scenarios relate to a wsio_intr_activate() caller.

If a driver invokes this routine as part of a non-WSIO event to activate a 
line based card interrupt (for instance, a card reset) and if interrupt 
migration is in progress, the invocation fails with a WSIO_ERROR.

When using transaction based interrupts as part of interrupt migration 
operation, the drivers must invoke wsio_intr_activate(), 
wsio_intr_set_cpu_spec(), and wsio_intr_deactivate() routines. 
Therefore, the driver must take care of the synchronization of any two 
driver threads invoking the routines at the same time.
See the Interrupt Migration chapter of the Driver Development Guide for related information.

**RETURN VALUES**

- **WSIO_OK**  
  Operation succeeded.

- **WSIO_ERROR**  
  Failure; no interrupt services available or interrupt migration might be in progress.

- **WSIO_INTR_INV_OBJ**  
  Must call `wsio_intr_set_cpu_spec()` or `wsio_intr_set_irq_line()` first.

- **WSIO_INTR_ACTIVATED**  
  `obj` already active.

- **WSIO_PARM_ERROR**  
  Invalid parameters.

**CONSTRAINTS**

**EXAMPLE**

```c
/* Allocate a line based interrupt and activate it */
wsio_intr_object_t obj;
int status;

/* allocate an interrupt object */
status = wsio_intr_alloc(isc, driver_isr,
                        (uintptr_t)isc, 0, &obj);
if (status != WSIO_OK) {
    return(ERROR);
}

/* Get a Level Sensitive IRQ */
status = wsio_intr_set_irq_line(isc, obj,
                                WSIO_IRQ_LINE_AUTO, 0);
if (status != WSIO_OK) {
    return(ERROR);
}

/* activate the interrupt */
status = wsio_intr_activate(isc, obj);
```

310 Chapter 4
if (status != WSIO_OK) {
    return (ERROR);
}

SEE ALSO

wsio_intr_alloc(WSIO3), wsio_intr_deactivate(WSIO5),
wsio_intr_deactivate_nowait(WSIO3),
wsio_intr_set_cpu_spec(WSIO3), wsio_intr_set_irq_line(WSIO3)
NAME

wsio_intr_alloc (WSIO3) – Allocate an interrupt object.

SYNOPSIS

#include <wsio/wsio.h>

int wsio_intr_alloc (struct isc_table_type *isc,
                     wsio_drv_isr_t isr,
                     uintptr_t arg,
                     uint64_t flags,
                     wsio_intr_object_t *obj);

PARAMETERS

isc Pointer to the driver's isc_table entry.

isr Address of the interrupt service routine.

arg Argument to be passed to the driver_isr.

flags Shared or exclusive flag.

obj Interrupt object.

DESCRIPTION

The wsio_intr_alloc() function allocates and initializes an interrupt object that will field interrupts generated by the given device associated with the isc. This routine may sleep and thus must be called from a thread context. This routine might allocate hardware resource, so it should be used with care.

The flags parameter should be either WSIO_INTR_EXCLUSIVE if the device driver's ISR cannot be shared, or zero if the ISR can be shared. If the WSIO_INTR_EXCLUSIVE flag is not present, the driver's ISR may be called even if the device did not generate an interrupt. If the device driver has no way of determining if the card needs servicing, the device driver's ISR cannot be shared. For such a card, running the device driver's ISR indicates the card needs servicing.

The function may block, so it must be called in the thread context.
RETURN VALUES

WSIO_OK  Successful completion.
WSIO_ERROR Failure; no interrupt services available.
WSIO_INTR_INV_FLAG Must specify 0 (shared) or WSIO_INTR_EXCLUSIVE only.
WSIO_MEM_ALLOC_FAILED Interrupt services failed to allocate memory.
WSIO_PARM_ERROR Invalid parameters

CONSTRAINTS

EXAMPLE

/* Allocate a line based interrupt */
wsio_intr_object_t obj;
int status;

/* allocate an interrupt object for a shared interrupt */
status = wsio_intr_alloc(isc, isr,
    (uintptr_t)isc, 0, &obj);
if (status != WSIO_OK) {
    return(ERROR);
}

SEE ALSO

wsio_intr_free(WSIO3)
NAME

wsio_intr_deactivate(WSIO5) – Disable an interrupt object.

SYNOPSIS

#include <wsio/wsio.h>

int wsio_intr_deactivate (struct isc_table_type *isc, wsio_intr_object_t obj);

PARAMETERS

isc Pointer to the driver’s isc_table entry.

obj Interrupt object.

DESCRIPTION

The wsio_intr_deactivate() WSIO function deactivates an interrupt object that was previously activated with wsio_intr_activate(). By deactivating the interrupt object the system will stop calling the device driver’s ISR (as specified in wsio_intr_alloc()). It is assumed that (if possible) the device’s interrupts will be disabled before this function is called. This function will sleep.

Attempting to deactivate an interrupt object that has not been activated is an error condition that returns WSIO_INTR_DEACTIVATED, without modifying the interrupt object.

If interrupt migration software is present in the system, the following scenarios are relevant to a wsio_intr_deactivate() caller.

If a driver invokes this routine as part of a non-WSIO event to activate a line based card interrupt (such as a card reset) and if interrupt migration operation is in progress, the invocation fails with a WSIO_ERROR.

When drivers use transaction based interrupts as part of the interrupt migration operation, they must invoke wsio_intr_activate(), wsio_intr_set_cpu_spec() and wsio_intr_deactivate() routines. Therefore, the driver must take care of the synchronization of any two driver threads invoking the routines at the same time.
See the Interrupt Migration chapter of the *Driver Development Guide* for related information.

## RETURN VALUES

- **WSIO_OK**  
  Successful completion.

- **WSIO_INTR_DEACTIVATED**  
  obj not active.

- **WSIO_PARM_ERROR**  
  Invalid parameters.

- **WSIO_ERROR**  
  Failure; no interrupt services available or interrupt migration might be in progress.

## CONSTRAINTS

Must not be called in an interrupt context.

## EXAMPLE

```c
/* Allocate a line based interrupt and activate it */
wsio_intr_object_t obj;
int status;

/* allocate an interrupt object */
status = wsio_intr_alloc(isc, driver_isr,(uintptr_t)isc, 0, &obj);
if (status != WSIO_OK) {
    return(ERROR);
}

/* Get a Level Sensitive IRQ */
status = wsio_intr_set_irq_line(isc, obj,
                                WSIO_IRQ_LINE_AUTO, 0);
if (status != WSIO_OK) {
    return(ERROR);
}

/* activate the interrupt */
status = wsio_intr_activate(isc, obj);
if (status != WSIO_OK) {
    return(ERROR);
} 
```
/* deactivate the interrupt */
status = wsio_intr_deactivate(isc, obj);
if (status != WSIO_OK) {
    return(ERROR);
}

SEE ALSO

wsio_intr_activate(WSIO5), wsio_intr_alloc(WSIO3),
wsio_intr_deactivate(WSIO5),
wsio_intr_deactivate_nowait(WSIO3),
wsio_intr_set_cpu_spec(WSIO3), wsio_intr_set_irq_line(WSIO3),
NAME

wsio_intr_deactivate_nowait(WSIO3) – Disable an interrupt object with callback.

SYNOPSIS

#include <wsio/wsio.h>

int wsio_intr_deactivate_nowait (struct isc_table_type *isc,
                                 wsio_intr_object_t obj,
                                 wsio_intr_deact_cb_t callback_func,
                                 uintptr_t arg);

PARAMETERS

isc Pointer to the driver’s isc_table entry.
obj Interrupt object.
callback_func A driver callback function that will be executed.
arg Parameter passed back to a driver in the callback.

DESCRIPTION

The wsio_intr_deactivate_nowait() WSIO function is a mirror of
wsio_intr_deactivate() except that this one does not sleep. It
deactivates an interrupt object that was previously activated with
wsio_intr_activate(). By deactivating the interrupt object the system
will stop calling the device driver’s ISR (as specified in
wsio_intr_alloc()). It is assumed that, if possible, the device’s
interrupts will be disabled before this function is called.

Attempting to deactivate an interrupt object that has not been activated
is an error condition that returns WSIO_INTR_DEACTIVATED, without
modifying the interrupt object.

This function will return immediately. Upon receiving confirmation that
the interrupts have been disabled, the callback_func will be executed
passing the arg back to the driver. Since this call will not sleep, it can be
called in a non-blocking context.
RETURN VALUES

- **WSIO_OK** Successful completion.
- **WSIO_INTR_DEACTIVATED** obj not active.
- **WSIO_ERROR** Operation failed.
- **WSIO_INTR_INV_OBJ** Must call wsio_intr_set_cpu_spec() or wsio_intr_set_irq_line() first.
- **WSIO_NO_INTR_CB** No call back function passed in.
- **WSIO_PARM_ERROR** Invalid parameters.

CONSTRAINTS

EXAMPLE

```c
/* Driver's callback function for wsio_intr_deactivate_nowait() */
void my_callback(struct isc_table_type *isc)
{
    ........
}

/* Allocate a line based interrupt and activate it */
wsio_intr_object_t obj;
int status;

/* allocate an interrupt object */
status = wsio_intr_alloc(isc, driver_isr,
                         (uintptr_t)isc, 0, &obj);
if (status != WSIO_OK) {
    return(ERROR);
}

/* Get a Level Sensitive IRQ */
status = wsio_intr_set_irq_line(isc, obj,
                                WSIO_IRQ_LINE_AUTO, 0);
if (status != WSIO_OK) {
    // Handle error
}
```
return(ERROR);
}

/* activate the interrupt */
status = wsio_intr_activate(isc, obj);
if (status != WSIO_OK) {
    return(ERROR);
}

............

/* deactivate the interrupt */
status = wsio_intr_deactivate_nowait(isc, obj, my_callback, isc);
if (status != WSIO_OK) {
    return(ERROR);
}

SEE ALSO

wsio_intr_activate(WSIO5), wsio_intr_alloc(WSIO3),
wsio_intr_deactivate(WSIO5), wsio_intr_set_cpu_spec(WSIO3),
wsio_intr_set_line(WSIO3)
NAME

wsio_intr_free(WSIO3) – Free an interrupt object.

SYNOPSIS

#include <wsio/wsio.h>

int wsio_intr_free (struct isc_table_type *isc, wsio_intr_object_t obj);

PARAMETERS

isc Pointer to the driver's isc_table entry.

obj Interrupt object.

DESCRIPTION

Before this function returns, the services guarantee that all outstanding ISRs have been run to completion. The obj object must not be active (either never activated, or deactivated with wsio_intr_deactivate(), or wsio_intr_deactivate_nowait()) when this function is called.

This function may block, so it must be called in a thread context.

RETURN VALUES

WSIO_OK Successful completion.

WSIO_INTR_ACTIVATED Must call wsio_intr_deactivate or wsio_intr_deactivate_nowait first.

WSIO_PARM_ERROR Invalid parameters.

CONSTRAINTS

Must not be called in an interrupt context.
EXAMPLE

/* Allocate a line based interrupt and activate it */
wsio_intr_object_t obj;
int status;

/* allocate an interrupt object */
status = wsio_intr_alloc(isc, driver_isr,
            (uintptr_t)isc, 0, &obj);
if (status != WSIO_OK) {
    return(ERROR);
}

/* Get a Level Sensitive IRQ */
status = wsio_intr_set_irq_line(isc, obj,
            WSIO_IRQ_LINE_AUTO, 0);
if (status != WSIO_OK) {
    return(ERROR);
}

/* activate the interrupt */
status = wsio_intr_activate(isc, obj);
if (status != WSIO_OK) {
    return(ERROR);
}

/* deactivate the interrupt */
status = wsio_intr_deactivate(isc, obj);
if (status != WSIO_OK) {
    return(ERROR);
}

/* free the interrupt obj */
status = wsio_intr_free(isc, obj);
if (status != WSIO_OK) {
    return(ERROR);
}
obj = NULL;

SEE ALSO

wsio_intr_alloc(WSIO3)
NAME

`wsio_intr_get_assigned_cpu` (WSIO3) – Get the currently assigned CPU for the interrupt object.

SYNOPSIS

```c
#include <wsio/wsio.h>

int wsio_intr_assigned_cpu (wsio_intr obj, intptr_t *cpu_spec);
```

PARAMETERS

- `obj`  
  Interrupt object.
- `cpu_spec`  
  CPU specification.

DESCRIPTION

The `wsio_intr_get_assigned_cpu()` WSIO function returns the currently assigned CPU for the passed interrupt object. The return type for `cpu_spec` will depend on platform. On PA, it will return assigned CPU in the location pointed by `cpu_spec`

`INTR_ATTR_ASSIGNED_CPU` attribute is not currently defined in BN-CDIO spec, but will be added as it is a necessary functionality.

RETURN VALUES

- **WSIO_OK**  
  Successful completion.
- **WSIO_ERROR**  
  `INTR_ATTR_ASSIGNED_CPU` not implemented in CDIO.
- **WSIO_PARM_ERROR**  
  Invalid parameters.

CONSTRAINTS
EXAMPLE

/* Allocate a transaction based interrupt */
wsio_intr_object_t obj;
int status;
intptr_t cpu_spec;

/* allocate an interrupt object */
status = wsio_intr_alloc(isc, driver_isr,
    (uintptr_t)isc, 0, &obj);
if (status != WSIO_OK) {
    return(ERROR);
}

/* Get a TXN based interrupt */
status = wsio_intr_set_cpu_spec(isc, obj, WSIO_INTR_CPU_ANY);
if (status != WSIO_OK) {
    return(ERROR);
}

/* activate the interrupt */
status = wsio_intr_activate(isc, obj);
if (status != WSIO_OK) {
    return(ERROR);
}

/* get the CPU spec */
status = wsio_intr_get_assigned_cpu(obj, &cpu_spec);
if (status != WSIO_OK) {
    return(ERROR);
}

printf("assigned cpu for the interrupt object is = %d\n",
    cpu_spec);

SEE ALSO

wsio_intr_activate(WSIO5), wsio_intr_alloc(WSIO3),
wsio_intr_set_cpu_spec(WSIO3)
NAME

wsio_intr_get_irq_line(WSIO3) – Get the interrupt line number.

SYNOPSIS

#include <wsio/wsio.h>

int wsio_intr_get_irq_line (struct isc_table_type *isc,
    wsio_intr_object_t obj,
    intptr_t *irq_line_num);

PARAMETERS

isc Pointer to the driver's isc_table entry.
obj Interrupt object.
irq_line_num The interrupt line number.

DESCRIPTION

The wsio_intr_get_irq_line() WSIO function returns the line number that the given obj is currently using. In most cases this routine is not necessary, as WSIO_IRQ_LINE_AUTO, can be given to wsio_intr_set_irq_line(). However, if for some reason a device driver needs to know the interrupt line that a card should use, this function provides the necessary data. On a running system the line number assigned to an obj may change. For example, when a processor is deallocated, the driver services will reassign the interrupt line on behalf of the driver.

RETURN VALUES

WSIO_OK Successful completion.
WSIO_INTR_INV_OBJ Must call wsio_intr_set_irq_line() first.
WSIO_INTR_ACTIVATED Object not active, call wsio_intr_activate() first.
WSIO_ERROR Failed to get line number.
/* Allocate a line based interrupt and activate it */
wsio_intr_object_t obj;
int status;
uintptr_t irq;

/* allocate an interrupt object */
status = wsio_intr_alloc(isc, driver_isr,
    (uintptr_t)isc, 0, &obj);
if (status != WSIO_OK) {
    return(ERROR);
}
/* Get a Level Sensitive IRQ */
status = wsio_intr_set_irq_line(isc, obj,
    WSIO_IRQ_LINE_AUTO, 0);
if (status != WSIO_OK) {
    return(ERROR);
}
/* activate the interrupt */
status = wsio_intr_activate(isc, obj);
if (status != WSIO_OK) {
    return(ERROR);
}
/* get the line number */
status = wsio_intr_get_irq_line(isc, obj, &irq);
if (status != WSIO_OK) {
    return(ERROR);
}
printf ("my line number is %ld\n", irq);

SEE ALSO
wsio_intr_activate(WSIO5), wsio_intr_alloc(WSIO3),
wsio_intr_set_line(WSIO3)
NAME

wsio_intr_get_txn_info(WSIO3) – Get the transaction address and data value.

SYNOPSIS

```c
#include <wsio/wsio.h>

int wsio_intr_get_txn_info (struct isc_table_type *isc, 
                           wsio_intr_object_t obj, 
                           intptr_t *txn_addr, 
                           intptr_t *txn_data);
```

PARAMETERS

- `isc` Pointer to the driver's `isc_table` entry.
- `obj` Interrupt object.
- `txn_addr` Transaction address value.
- `txn_data` Transaction data value.

DESCRIPTION

The `wsio_intr_get_txn_info()` WSIO function returns the transaction address and transaction data associated with a transaction based `obj`.

RETURN VALUES

- `WSIO_OK` Successful completion.
- `WSIO_INTR_INV_OBJ` Must be a transaction based `obj`; call `wsio_intr_set_cpu_spec()` first.
- `WSIO_ERROR` Failed to get values.
- `WSIO_PARM_ERROR` Invalid parameters.

CONSTRAINTS
EXAMPLE

/* Allocate a transaction based interrupt */
wsio_intr_object_t obj;
int status;
intptr_t txn_addr, txn_data;

/* allocate an interrupt object */
status = wsio_intr_alloc(isc, driver_isr,
                      (uintptr_t)isc, 0, &obj);
if (status != WSIO_OK) {
    return(ERROR);
}

/* Get a TXN based interrupt */
status = wsio_intr_set_cpu_spec(isc, obj, WSIO_INTR_CPU_ANY);
if (status != WSIO_OK) {
    return(ERROR);
}

/* activate the interrupt */
status = wsio_intr_activate(isc, obj);
if (status != WSIO_OK) {
    return(ERROR);
}

/* get the TXN values */
status = wsio_intr_get_txn_info(isc, obj,
                         &txn_addr, &txn_data);
if (status != WSIO_OK) {
    return(ERROR);
}
printf ("txn_addr = %ld\n", txn_addr);
printf ("txn_data = %ld\n", txn_data);

SEE ALSO

wsio_intr_activate(WSIO5), wsio_intr_alloc(WSIO3),
wsio_intr_set_cpu_spec(WSIO3)
NAME

wsio_intr_migr_t (WSIO5) – Driver - WSIO communication structure during interrupt migration.

SYNOPSIS

#include <wsio/wsio.h>

DESCRIPTION

This structure is used for communication between drivers and WSIO when interrupt migration is being processed.

See the Interrupt Migration chapter in the Driver Development Guide for relevant information.

STRUCTURE MEMBERS

<table>
<thead>
<tr>
<th>Table 4-3 Driver Relevant wsio_intr_migr_t Structure Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td>wsio_intr_object_t</td>
</tr>
<tr>
<td>intptr_t</td>
</tr>
<tr>
<td>wsio_intr_migr_info_t</td>
</tr>
<tr>
<td>wsio_ret_code_t</td>
</tr>
<tr>
<td>void*</td>
</tr>
</tbody>
</table>

intr_obj Interrupt object of the interrupt being moved.

dest_spu CPU ID of the CPU to which the interrupt is to be moved.
The value of dest_spu depends on the event and can be as indicated in the following table:

### Table 4-4 dest_spu values

<table>
<thead>
<tr>
<th>dest_spu</th>
<th>migr_info</th>
<th>Event</th>
<th>Descr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/A</td>
<td>WSIO_LBI_INTR_MIGR_NOTIFY</td>
<td>WSIO_LBI_INTR_MIGR</td>
<td>The notify event is sent to all LBI drivers which have registered for the WSIO_LBI_INTR_MIGR event. dest_spu is not valid here.</td>
</tr>
<tr>
<td>spu_id</td>
<td>WSIO_LBI_INTR_MIGR_COMPLETE</td>
<td>WSIO_LBI_INTR_MIGR</td>
<td>The LBI drivers, which have registered for the WSIO_LBIINTR_MIGR event are notified after the interrupt migration has completed. The dest_spu is the “new” CPU to which the interrupt has migrated.</td>
</tr>
<tr>
<td>-1/spu_id</td>
<td>N/A</td>
<td>WSIO_OFF_LINE_CPU</td>
<td>If the dest_cpu is not -1, the spu_id should be used by drivers in the wsio_intr_set_cpu_spec() call to migrate the interrupt to this CPU. If it is -1, drivers can pass in WSIO_INTR_CPU_ANY, WSIO_INTR_CPU_ANY_UNIQUE, or a spu_id (see wsio_intr_set_cpu_spec()).</td>
</tr>
</tbody>
</table>

migr_info  More information about the migration event; see wsio_intr_migr_info_t.
ret_val    Return value of migration.
resvd      Reserved field.
SEE ALSO

wsio_intr_migr_info(WSIO5), wsio_drv_event_t(WSIO5),
wsio_intr_set_cpu_spec(WSIO3)
NAME

wsio_intr_migr_info_t(WSIO5) – Event notification for migration of line based interrupts

SYNOPSIS

#include <wsio/wsio.h>

DESCRIPTION

This enumerator is used to notify the LBI drivers, which have registered with WSIO, about an interrupt migration event. The wsio_intr_migr_t structure is used for communication between WSIO and the driver. This structure is not utilized with TBI drivers.

Refer to the Interrupt Migration chapter of the Driver Development Guide for relevant information.

STRUCTURE MEMBERS

typedef enum wsio_intr_migr_info {
    WSIO_LBI_INTR_MIGR_NOTIFY = 1,
    WSIO_LBI_INTR_MIGR_COMPLETE
} wsio_intr_migr_info_t;

WSIO_LBI_INTR_MIGR_NOTIFY    WSIO sends a notification to the LBI driver that the interrupt will be migrated.

WSIO_LBI_INTR_MIGR_COMPLETE   WSIO notifies the driver of completion of the interrupt migration. The spu_id field of wsio_intr_migr_t is set to cpu_id of the CPU to which the interrupt has been migrated.

SEE ALSO

wsio_intr_migr_t(WSIO5), wsio_drv_event_t(WSIO5),
wsio_intr_set_cpu_spec(WSIO3)
NAME

`wsio_intr_set_cpu_spec`(WSIO3) – Initialize and distribute transaction based interrupts.

SYNOPSIS

```c
#include <wsio/wsio.h>

int wsio_intr_set_cpu_spec (struct isc_table_type *isc,
                       wsio_intr_object_t obj,
                       intptr_t cpu_spec);
```

PARAMETERS

- **isc**
  Pointer to the driver's isc_table entry.

- **obj**
  Interrupt object.

- **cpu_spec**
  CPU load balancing specification. If `cpu_spec` is not one of the two flags described below but is a `txn_addr`, the interrupt will be bound to this CPU.

<table>
<thead>
<tr>
<th><code>cpu_spec</code></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSIO_INTR_CPU_ANY</td>
<td>The services will select any processor.</td>
</tr>
<tr>
<td></td>
<td>The device driver does not care which processor it is bound to.</td>
</tr>
<tr>
<td>WSIO_INTR_CPU_ANY_UNIQUE</td>
<td>The services will select a processor that this adapter card does not already send interrupts to; this is not currently implemented. The services will return a <code>WSIO_NOT_IMPLEMENTED</code> error if the driver specifies <code>WSIO_INTR_CPU_ANY_UNIQUE</code> for <code>cpu_spec</code>.</td>
</tr>
</tbody>
</table>

- **txn_addr**
  Bind to this CPU address. If a processor number is specified, it should be a value between zero and the number of CPUs minus one.
DESCRIPTION

The `wsio_intr_set_cpu_spec()` WSIO function is used to initialize and distribute transaction based interrupts. All drivers using transaction based interrupts must register to the `WSIO_EVENT_OFFLINE_CPU` before calling this function. The drivers can register for this mandatory event through the `wsio_reg_drv_capability_mask()` call. The registration routine is invoked in the attach routine after the completion of `isc_claim()`. The Interrupt Migration chapter of the Driver Development Guide has relevant information.

The function can block and should be called in the thread context. If the function is called multiple times each successive call overrides the value of the previous call.

RETURN VALUES

- **WSIO_OK**
  - Successful completion.
- **WSIO_INTR_INV_OBJ**
  - Must be a transaction based `obj`; call `wsio_intr_set_cpu_spec()` first.
- **WSIO_ERROR**
  - Failed to set `cpu_spec`.
- **WSIO_EXCLUSIVE_FAILED**
  - Can not get an exclusive interrupt.
- **WSIO_INTR_ACTIVATED**
  - `obj` is currently active; must call `wsio_intr_deactivate()` first.
- **WSIO_NOT_IMPLEMENTED**
  - `cpu_spec` algorithm not implemented.
- **WSIO_INTR_INV_CPU_NUM**
  - Invalid CPU number specified.
- **WSIO_PARM_ERROR**
  - Invalid parameters.

CONSTRAINTS
EXAMPLE

The following is an example of a driver registering for a WSIO_EVENT_OFFLINE_CPU event:

driver_attach(...) {
    wsio_event_mask_t newmask;
    .
    .
    isc_claim(isc);
    newmask = oldmask | WSIO_EVENT_OFFLINE_CPU;
    ret = wsio_reg_drv_capability_mask(isc, newmask);
    .
    .
}

The following is an example of a driver calling wsio_intr_set_cpu_spec().

/* Allocate a transaction based interrupt */
wsio_intr_object_t obj;
int status;
uintptr_t txn_addr, txn_data;

/* allocate an interrupt object */
status = wsio_intr_alloc(isc, driver_isr,
                          (uintptr_t)isc, 0, &obj);
if (status != WSIO_OK) {
    return(ERROR);
}

/* Get a TXN based interrupt */
status = wsio_intr_set_cpu_spec(isc, obj, WSIO_INTR_CPU_ANY);
if (status != WSIO_OK) {
    return(ERROR);
}

/* activate the interrupt */
status = wsio_intrActivate(isc, obj);
if (status != WSIO_OK) {
    return(ERROR);
}

/* get the TXN values */
status = wsio_intr_get_txn_info(isc, obj,
                                &txn_addr, &txn_data);
if (status != WSIO_OK) {
    return(ERROR);
}
printf("txn_addr = %ld\n", txn_addr);
printf("txn_data = %ld\n", txn_data);

SEE ALSO

wsio_intr_activate(WSIO5), wsio_intr_alloc(WSIO3),
wsio_intr_set_cpu_spec(WSIO3),
wsio_intr_get_assigned_cpu(WSIO3)
NAME

\texttt{wsio\_intr\_set\_irq\_line}(\texttt{WSIO3}) – Set the interrupt line number.

SYNOPSIS

```c
#include <wsio/wsio.h>

int wsio\_intr\_set\_irq\_line (struct isc\_table\_type *isc,
                               wsio\_intr\_object\_t obj,
                               intptr\_t *irq\_line\_num,
                               uint64\_t flags);
```

PARAMETERS

- \textit{isc} Pointer to the driver's isc\_table entry.
- \textit{obj} Interrupt object.
- \textit{irq\_line\_num} The interrupt line number, or \texttt{WSIO\_IO\_INT\_LINE\_AUTO}.
- \textit{flags} Zero (level sensitive) or \texttt{WSIO\_INTR\_EDGE\_SENSITIVE}.

DESCRIPTION

The \texttt{wsio\_intr\_set\_irq\_line()} \texttt{WSIO} function is used to setup a line based \textit{obj}. The \texttt{WSIO\_INTR\_ACTIVATED} error code will be returned if this function is called on an active interrupt object. Most drivers will use \texttt{WSIO\_IRQ\_LINE\_AUTO} for the value of the \textit{irq\_line\_num} parameter which forces the services to determine the interrupt line value for the particular device.

If the device generates level-sensitive interrupts, the \textit{flags} parameter should be zero. If the device generates edge-sensitive interrupts, the \textit{flags} parameter should be \texttt{WSIO\_INTR\_EDGE\_SENSITIVE}.

Calling this interface multiple times will not move this interrupt from one processor to another in a round-robin manner. Refer to the Interrupt Migration chapter of the \textit{Driver Development Guide} for driver related changes and impact.
RETURN VALUES

WSIO_OK Operation succeeded.
WSIO_INTR_INV_OBJ Must not be a transaction based interrupt.
WSIO_INTR_ACTIVATED Object is active; do not call wsio_intr_activate() first.
WSIO_ERROR Failed to set line number.
WSIO_INTR_INV_FLAG Must be zero (level) or WSIO_INTR_EDGE_SENSITIVE.
WSIO_PARM_ERROR Invalid parameters.

CONSTRAINTS

EXAMPLE

/* Allocate a line based interrupt */
wsio_intr_object_t obj;
int status;

/* allocate an interrupt object */
status = wsio_intr_alloc(isc, driver_isr, 
(uintptr_t)isc, 0, &obj);
if (status != WSIO_OK) {
    return(ERROR);
}

/* Get a Level Sensitive IRQ */
status = wsio_intr_set_irq_line(isc, obj, 
    WSIO_IRQ_LINE_AUTO, 0);
if (status != WSIO_OK) {
    return(ERROR);
}

SEE ALSO

wsio_intr_activate(WSIO5), wsio_intr_alloc(WSIO3),
wsio_intr_set_line(WSIO3)
NAME

wsio_io_sync(WSIO3) – Perform a sync of shared memory if necessary.

SYNOPSIS

#include <wsio/wsio.h>

void wsio_io_sync (struct isc_table_type * isc);

PARAMETERS

isc Pointer to the driver's isc_table entry.

DESCRIPTION

The wsio_io_sync() function synchronizes a device's view and the host's view of memory. This functionality can also be achieved by doing a PIO read of a card register, but in some implementations this function may provide a lower latency mechanism.

wsio_io_sync() can be called in a non-blocking context.

RETURN VALUES

None

CONSTRAINTS

EXAMPLE

wsio_io_sync(isc_entry);

SEE ALSO

WSIO_ORDERED_INTERRUPTS(WSIO3)
NAME
wsio_iova_to_phys(WSIO3) – Translate an I/O virtual address to a physical address.

SYNOPSIS

```c
#include <wsio/wsio.h>

caddr_t wsio_iova_to_phys (struct isc_table_type *isc,
                      void *dma_handle, wsio_iova_t iova);
```

PARAMETERS

- `isc` Pointer to the driver’s isc_table entry.
- `dma_handle` DMA handle allocated using `wsio_allocate_dma_handle()`.
- `iova` I/O virtual address to be translated.

DESCRIPTION

The `wsio_iova_to_phys()` WSIO function is called by a device driver to translate an I/O virtual address to a physical address.

`wsio_iova_to_phys()` can be called in a non-blocking context.

RETURN VALUES

The physical address corresponding to `iova` or NULL if a translation does not exist.

CONSTRAINTS
EXAMPLE

caddr_t phys_address = NULL;

phys_address =
wsio_iova_to_phys(isc_entry,dma_handle,io_address);

/* The 'phys_address' variable will now contain the physical
 * address that 'iova' translates to.
 */

SEE ALSO

wsio_allocate_dma_handle(WSIO3),
wsio_allocate_shared_mem(WSIO3), wsio_dma_pass_thru(WSIO3),
wsio_fastmap_dma_buffer(WSIO3), wsio_free_dma_handle(WSIO3),
wsio_free_shared_mem(WSIO3), wsio_flush_shared_mem(WSIO3),
wsio_init_map_context(WSIO3), wsio_map_dma_buffer(WSIO3),
wsio_remap_dma_buffer(WSIO3),
wsio_set_device_attributes(WSIO3),
wsio_unmap_dma_buffer(WSIO3).
NAME

wsio_isc_to_instance(WSIO3) – Retrieve an instance number of an iotree node

SYNOPSIS

```c
#include <wsio/wsio.h>
#include <sys/ioparams.h>

int wsio_isc_to_instance(struct isc_table_type *isc, 
                         hw_path_t *dev_hw_path);
```

PARAMETERS

isc
A pointer to the ISC structure associated with the interface card for the device.

dev_hw_path
A pointer to a structure containing device hardware path information relative to the interface card, or NULL if the card instance is desired.

DESCRIPTION

The wsio_isc_to_instance() WSIO function provides an instance number of an iotree node that is a descendant of the card or device node specified by the `isc` and `dev_hw_path` parameters, according to the following rules:

- If `isc` is valid and `dev_hw_path` is NULL, the instance number of an iotree node corresponding to the interface card is returned. SCSI interface drivers must check the instance number returned by `wsio_isc_to_instance()`. If the returned value is greater than `SCSI_MAX_BUS_ID`, the driver should return `WSIO_ERROR`.

- If `isc` is valid and `dev_hw_path` contains a path to a valid device (relative to the device’s interface card), the instance number of the iotree node corresponding to the device is returned.

This function will not provide valid instance numbers if it is accessed before the driver has actually claimed a device (that is, before the driver’s attach routine has successfully claimed and initialized an ISC structure). Until that point, the driver is not associated with an iotree
node and will not have a valid instance number. Using this service within a driver's `driver_if_init()` or `driver_dev_init()` routines and later will yield a valid result.

**RETURN VALUES**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>≥0</td>
<td>Successful completion. The value is the matching instance number.</td>
</tr>
<tr>
<td>-1</td>
<td>Error.</td>
</tr>
</tbody>
</table>

**CONSTRAINTS**

**SEE ALSO**
NAME

`WSIO_LITTLE_ENDIAN` (WSIO3) – Macro to return true (1) if the local bus is little-endian.

SYNOPSIS

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

int WSIO_LITTLE_ENDIAN(struct isc_table_type *isc);
```

PARAMETERS

`isc` Pointer to the driver's `isc_table` entry.

DESCRIPTION

The `WSIO_LITTLE_ENDIAN` () macro is called by a device driver to report whether the local bus is little-endian. If it is, true is returned; otherwise it returns false. This can be used by a driver along with the known endianness of the host processor to decide whether endian swapping should be performed. Endian swapping might be necessary for any data transfers between the I/O bus and local host memory.

`WSIO_LITTLE_ENDIAN()` can be called in a non-blocking context.

RETURN VALUES

Returns a one (1) if the local bus is little-endian, and a zero (0) otherwise.

CONSTRAINTS

EXAMPLE

```
if (WSIO_LITTLE_ENDIAN(isc_entry)) {
    /* Endian swapping must be performed
} else {
    /* No endian swapping necessary */
```
WSIO Reference Pages

WSIO_LITTLE_ENDIAN(WSIO3)

SEE ALSO

WSIO_BIG_ENDIAN(WSIO3)
NAME

wsio_map_cfg_handle(WSIO3) – Obtain a configuration space access handle.

SYNOPSIS

#include <wsio/wsio.h>

int wsio_map_cfg_handle (struct isc_table_type *isc,
                        wsio_addr_handle_t *cfg_handle);

PARAMETERS

isc Pointer to the driver's isc_table entry.

cfg_handle Pointer to contain the configuration handle upon completion.

DESCRIPTION

The wsio_map_cfg_handle() WSIO function is called by device drivers to obtain a handle to access configuration space. wsio_map_cfg_handle() must not be called in a non-blocking context.

RETURN VALUES

WSIO_OK Indicates a handle was successfully returned in cfg_handle.

WSIO_ERROR Indicates there was an error obtaining a handle.

CONSTRAINTS

Must not be called in an interrupt context.
EXAMPLE

wsio_addr_handle_t handle;

if (wsio_map_cfg_handle(isc_entry,&handle) != WSIO_OK) {
    /* Error obtaining configuration space handle. Return an
     * error code */
    return(ERROR);
}
/* The configuration space handle is now in the handle
 * variable */

SEE ALSO

wsio_cfg_inXX(WSIO3), wsio_cfg_outXX(WSIO3),
wsio_unmap_cfg_handle(WSIO3)
NAME

wsio_map_dma_buffer(WSIO3) – Map an existing memory object for packet DMA.

SYNOPSIS

#include <wsio/wsio.h>

wsio_map_status_t wsio_map_dma_buffer (  
    struct isc_table_type * isc,  
    void * dma_handle,  
    wsio_map_context_t * context,  
    wsio_dma_buffer_hints_t hints,  
    wsio_range_type_t range_type,  
    wsio_dma_map_t * host_range,  
    wsio_dma_map_t * io_range);

PARAMETERS

isc                  Pointer to the driver's isc_table entry.

dma_handle          DMA handle allocated using
                    wsioAllocate_dma_handle().

context             Pointer to the context used for mapping.

hints               Bitmask that provides mapping hints. The allowable
                    hints are as follows:

<table>
<thead>
<tr>
<th>hints</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSIO_DMA_SAFE</td>
<td>Forces coherent transactions to be used even for</td>
</tr>
<tr>
<td></td>
<td>full-cacheline transactions. In some implementations</td>
</tr>
<tr>
<td></td>
<td>semi-coherent transactions are used to enhance</td>
</tr>
<tr>
<td></td>
<td>performance when it is known that the entire</td>
</tr>
<tr>
<td></td>
<td>cacheline will be modified. Affects inbound DMA only.</td>
</tr>
</tbody>
</table>


WSIO_DMA_LOCK
In some implementations this allows atomic access to memory for devices using bus-lock primitives.

WSIO_DMA_FLUSH_ON_USE
In some implementations this hint tells the hardware to flush resources associated with this buffer after they are used. This inhibits coalescing transactions into larger transfers for cases where it is not beneficial.

WSIO_DMA_IGN_ALIGNMENT
Indicates map_dma_buffer should not automatically set the IO_SAFE hint for partial cacheline buffers.

WSIO_DMA_CONTIGUOUS
Indicates map_dma_buffer must allocate a single contiguous IOVA. If wsio_map_dma_buffer is unable to do this, it will return WSIO_MAP_E_PARAMETER_ERROR. This hint implies WSIO_IO_IGN_ALIGNMENT.

WSIO_DMA_NO_SEQ
Inhibits hardware prefetching for outbound DMA in some implementations.

WSIO_DMA_INBOUND
Indicates the buffer will be used exclusively for inbound DMA.

WSIO_DMA_OUTBOUND
Indicates the buffer will be used exclusively for outbound DMA.

WSIO_DMA_NULL
Forces all hint values to zero. A hint value of zero tells the BN-CDIO to take hint values from the DMA object.
range_type  Indicates the type of host memory being mapped. It can be:

KERNELSPACE  Indicates host_range is a kernel virtual buffer.

PHYSICAL  Indicates host_range is a physical buffer.

> 0  Indicates host_range is in user space, and this will be the space ID of the virtual address.

host_range  Pointer to an address/length structure that contains information about the host space to map. If the mapping was only partially completed, this will contain information about the remaining space to be mapped when the call completes.

io_range  Pointer to an address/length structure that will contain information about the I/O space that was mapped.

**DESCRIPTION**

The `wsio_map_dma_buffer()` WSIO function is called by a device driver to map an existing memory object for packet DMA. If continuous DMA is required `wsio_allocate_shared_mem()` should be used. Continuous DMA should be used for control structures, circular buffers, or any kind of buffer that needs to be accessed on a continuous basis by the I/O device. Packet DMA should be used when the mappings are temporary, or when pre-existing memory objects must be mapped for DMA.

If the entire host range cannot be mapped, the buffer may only be partially mapped. This will be indicated by a return value of `WSIO_MAP_W_PARTIAL`. In this case, `wsio_map_dma_buffer()` will need to be called again to map more of the host range. Using this method of calling `wsio_map_dma_buffer()` multiple times, the entire host range can be mapped into multiple I/O virtual ranges.

All mappings remain in effect until `wsio_unmap_dma_buffer()` or `wsio_remap_dma_buffer()` are called to remove or change them.

Callers are guaranteed buffers of at least 4K Bytes can be mapped into a contiguous range of I/O virtual address.
The `context` parameter is used to insure I/O resources are used efficiently. This structure should be initialized via a call to `wsio_init_map_context()`. The same context should be used for any group of mappings that will all be unmapped at the same time. If multiple buffers will not be mapped into one context, `NULL` can be passed in instead of a valid context.

Device drivers can set up a callback routine that will come into play if resources are not available at the time a mapping is attempted. If this callback is set up, `WSIO_MAP_W_CALLBACK` will be returned instead of a no resource error. When resources become available, the callback routine will be called to indicate this to the device driver. For more information on how to setup and use a callback, see the `wsio_set_dma_callback()` manpage.

`wsio_map_dma_buffer()` can be called in a non-blocking context.

**RETURN VALUES**

- **WSIO_MAP_OK**     Returned if the entire buffer has been mapped.
- **WSIO_MAP_W_PARTIAL**     Returned if only part of the buffer has been mapped.
- **WSIO_MAP_W_CALLBACK**     Returned if no resources are available and a callback function exists.
- **WSIO_MAP_E_NO_RESOURCES**     Returned if no resources are available and no callback function exists.
- **WSIO_MAP_ERESOURCE_ERROR**     Returned if the request cannot and will never succeed.
- **WSIO_MAP_E_HIGH_ADDR**     Returned if the call failed because the device cannot reach the destination address.
- **WSIO_MAP_E_PARAMETER_ERROR**     Returned if an invalid parameter has caused failure of the call.
- **WSIO_MAP_E_UNKNOWN_ERROR**     Returned for hardware or other errors.
CONSTRAINTS

EXAMPLE

```c
void *dma_handle;
wsio_map_context_t dma_con;
wsio_dma_map_t host_range, io_range;

dma_handle = wsio_allocate_dma_handle(isc_entry);
wsio_init_map_context(&dma_con);

host_range.iov_base = host_virtual_address;
host_range.iov_len = dma_buffer_length;

if (wsio_map_dma_buffer(isc_entry,dma_handle,dma_con,
    WSIO_DMA_OUTBOUND,KERNELSPACE,&host_range,&io_range
)! = WSIO_MAP_OK) {
    /* Unable to perform the mapping so return an error */
    return(ERROR);
}

/* The host virtual buffer represented in the above
 * code by host_virtual_address’ is now mapped. Note that
 * this code does not handle the case where the buffer is
 * only partially mapped. In that case, a more
 * complete example would call wsio_map_dma_buffer() again
 * each time WSIO_MAP_PARTIAL was returned and save each
 * io_range that was returned.
 */
```

SEE ALSO

wsio_allocate_dma_handle(WSIO3),
wsio_allocate_shared_mem(WSIO3), wsio_dma_pass_thru(WSIO3),
wsio_fastmap_dma_buffer(WSIO3), wsio_free_dma_handle(WSIO3),
wsio_free_shared_mem(WSIO3), wsio_flush_shared_mem(WSIO3),
wsio_init_map_context(WSIO3), wsio_iova_to_phys(WSIO3),
wsio_remap_dma_buffer(WSIO3),
wsio_map_dma_buffer(WSIO3)

wsio_set_device_attributes(WSIO3),
wsio_set_dma_attributes(WSIO3),
wsio_unmap_dma_buffer(WSIO3),
NAME

\texttt{wsio\_map\_port}(WSIO3) – Obtain an I/O port handle.

SYNOPSIS

```c
#include <wsio/wsio.h>

int wsio_map_port (struct isc_table_type *isc, 
                  int32_t port_addr, 
                  size_t size, 
                  wsio_addr_handle_t *port_handle);
```

PARAMETERS

\begin{itemize}
\item \textit{isc} \hspace{1cm} Pointer to the driver's isc_table entry.
\item \textit{port_addr} \hspace{1cm} Address of the port obtained from \texttt{wsio_get_iports()}
\item \textit{size} \hspace{1cm} Size of the port to be mapped.
\item \textit{port_handle} \hspace{1cm} The port handle upon completion.
\end{itemize}

DESCRIPTION

The \texttt{wsio_map_port()} WSIO function will attempt to map an I/O port. After a port has been mapped, it can be read from and written to using the \texttt{wsio_port_inXX()} and \texttt{wsio_port_outXX()} accessor functions. \texttt{wsio_map_port()} must not be called in a non-blocking context.

RETURN VALUES

\begin{itemize}
\item \texttt{WSIO\_OK} \hspace{1cm} Successful completion.
\item \texttt{WSIO\_ERROR} \hspace{1cm} There was a parameter error.
\end{itemize}

CONSTRAINTS

Must not be called in an interrupt context.
EXAMPLE

```c
wsio_addr_handle_t port_handle;
wsio_iop_t iports_array[10];
/* An array with enough space for all ports needs to be
 * allocated */

if (wsio_get_ioports(isc_entry,10,iports_array) != WSIO_OK) {
    /* There was a problem obtaining the ports */
    return(ERROR);
}

if (wsio_map_port(isc_entry,iports_array[0].addr,
             iports_array[0].size, &port_handle) != WSIO_OK) {
    /* There was an error mapping the port */
    return(ERROR);
}

/* The first I/O port will now be mapped and can be
 * accessed via 'port_handle'. */
```

SEE ALSO

wsio_get_ioports(WSIO3), wsio_port_inXX(WSIO3),
wsio_port_outXX(WSIO3), wsio_unmap_port(WSIO3)
NAME

wsio_map_reg(WSIO3) – Map device registers to host memory space.

SYNOPSIS

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

int wsio_map_reg (struct isc_table_type *isc,
                wsio_reg_info_t* reg_info);
```

PARAMETERS

isc Pointer to the driver’s isc_table entry.

reg_info Pointer to information about the register to be mapped. This structure should be obtained from wsio_get_all_registers(). It will also be used after the mapping to access the register.

DESCRIPTION

The wsio_map_reg() WSIO function, given information about a register to be mapped, will attempt to map a register so that it is accessible via the wsio_read_regXX(), and the wsio_write_regXX() functions. After the mapping has been performed, the reg_info variable can be used to access the register. Prior to the mapping, reg_info must be obtained from a call to wsio_get_all_registers(). wsio_map_reg() must not be called in a non-blocking context.

RETURN VALUES

WSIO_OK Successful completion.

WSIO_ERROR Could not create the mapping.

CONSTRAINTS

Must not be called in an interrupt context.
EXAMPLE

wsio_reg_info_t *registers;

registers = wsio_get_all_registers(isc_entry);
if (registers == NULL) {
    /* No registers exist. Return an error */
    return(ERROR);
}

if (wsio_map_reg(isc_entry,&registers[1]) != WSIO_OK) {
    return(ERROR);
}

/* The second device register (index 1 into the array) will
* now be mapped. */

SEE ALSO

wsio_get_all_registers(WSIO3), wsio_read_regXX(WSIO3),
wsio_write_regXX(WSIO3), wsio_unmap_reg(WSIO3),
NAME

`WSIO_ORDERED_INTERRUPTS(WSIO3)` – Macro to indicate whether interrupts are ordered with respect to DMA transactions.

SYNOPSIS

```c
#include <wsio/wsio.h>

int WSIO_ORDERED_INTERRUPTS(struct isc_table_type * isc);
```

PARAMETERS

`isc` Pointer to the driver's `isc_table` entry.

DESCRIPTION

The `WSIO_ORDERED_INTERRUPTS()` macro reports to a device driver about whether interrupts are ordered with respect to DMA transactions. If true (1) is returned, nothing needs to be done. However, if false (0) is returned, interrupts are not ordered, and drivers must ensure DMA transactions have completed by reading a status register, or by calling `wsio_io_sync()`.

`WSIO_ORDERED_INTERRUPTS()` can be called in a non-blocking context.

RETURN VALUES

Returns a one (1) if interrupts are ordered with respect to DMA transactions, and a zero (0) otherwise.

CONSTRAINTS

EXAMPLE

```c
if (WSIO_ORDERED_INTERRUPTS(isc_entry)) {
    wsio_io_sync(isc_entry);
} else {
    /* No syncing necessary */
}
```
WSIO Reference Pages

WSIO_ORDERED_INTERRUPTS(WSIO3)

SEE ALSO

wsio_io_sync(WSIO3)
NAME

wsio_port_inXX(WSIO3) – Read XX bits from an I/O port.

SYNOPSIS

#include <wsio/wsio.h>

void wsio_port_inXX (struct isc_table_type *isc,
        wsio_addr_handle_t handle,
        uint32_t addr,
        uintXX_t *data);

PARAMETERS

isc Pointer to the driver's isc_table entry.

handle Handle obtained from wsio_map_port().

addr Offset into the port to read from.

data Contains the data read upon completion.

DESCRIPTION

The wsio_port_inXX() WSIO function is used to read XX from an I/O port. The port must have been mapped prior to the read by using the wsio_map_port() macro. No endian swapping is performed for port access, so if the local bus and the host bus are of opposite endianness, the driver must perform an endian swap. The XX refers to the size of the transfer to perform and must be either 8, 16, 32, or 64.

RETURN VALUES

None.

CONSTRAINTS
EXAMPLE

```c
wsio_addr_handle_t port_handle;
wsio_iop_t iports_array[10];
uint32_t data;

/* An array with enough space for all ports needs to be
 * allocated. We have just made it a local variable in this
 * example */
if (wsio_get_ioports(isc_entry,10,iports_array) != WSIO_OK) {
    /* There was a problem obtaining the ports */
    return(ERROR);
}

if (wsio_map_port(isc_entry,iports_array[0].addr,
                   iports_array[0].size, &port_handle
                 ) != WSIO_OK) {
    /* There was an error mapping the port */
    return(ERROR);
}

/* Now read 32-bits from the port */
wsio_port_in32(isc_entry,port_handle,0,&data);

/* 'data' will now contain whatever was at offset 0 of I/O
 * port 0 */
```

SEE ALSO

- wsio_get_ioports(WSIO3)
- wsio_map_port(WSIO3)
- wsio_port_outXX(WSIO3)
- wsio_unmap_port(WSIO3)
NAME

wsio_port_outXX(WSIO3) – Write XX bits to an I/O port.

SYNOPSIS

#include <wsio/wsio.h>

void wsio_port_outXX (struct isc_table_type *isc,
                      wsio_addr_handle_t handle,
                      uint32_t addr,
                      uintXX_t data);

PARAMETERS

isc Pointer to the driver's isc_table entry.
handle Handle obtained from wsio_map_port().
addr Offset into the port to read from.
data Data to be written.

DESCRIPTION

The wsio_port_outXX() WSIO function is used to write XX bits to an I/O port. The port must have been mapped prior to the write by using the wsio_map_port() macro. No endian swapping is performed for port access, so if the local bus and the host bus are of opposite endianness, the driver must perform an endian swap. The XX refers to the size of the transfer to perform, and must be either 8, 16, 32, or 64.

RETURN VALUES

None.

CONSTRAINTS
EXAMPLE

```c
wsio_addr_handle_t port_handle;
wsio_iop_t iports_array[10];
uint32_t data;

/* An array with enough space for all ports needs to be * allocated. We have just made it a local variable in this * example */
if (wsio_get_ioports(isc_entry,10,iports_array) != WSIO_OK) {
    /* There was a problem obtaining the ports */
    return(ERROR);
}
if (wsio_map_port(isc_entry,iports_array[0].addr,
        iports_array[0].size,
        &port_handle) != WSIO_OK) {
    /* There was an error mapping the port */
    return(ERROR);
}
/* Now write 32-bits to the port */
wsio_port_out32(isc_entry,port_handle,0,0x5a);
/* This example wrote the 32 bits 0x5a to I/O port 0 at * offset 0 */
```

SEE ALSO

wsio_get_ioports(WSIO3), wsio_map_port(WSIO3),
wsio_port_outXX(WSIO3), wsio_unmap_port(WSIO3)
NAME

wsio_probe_dev_info(WSIO4) – WSIO device probe information

SYNOPSIS

#include <wsio/wsio.h>

struct wsio_probe_dev_info
{
    unsigned short instance;
    unsigned short target;
    unsigned short opt_1;
    unsigned short opt_2;
    unsigned short opt_3;
};

DESCRIPTION

The wsio_probe_dev_info structure contains some of the device file information needed for WSIO driver probe routines. This structure communicates hardware path information within driver probe routines (especially for those cases where drivers may split their probe routines into two routines: one for determining the next address to probe and one to actually try to build and open a device file for that address). The hardware addressing information in this format can easily be passed between two routines and is needed for building special device files for the devices being probed.

STRUCTURE MEMBERS

instance Instance number of the nearest interface card ancestor.

target Relative hardware address of first layer to be probed.

opt_1 Optional. Sometimes used for the hardware address of the second layer to be probed (e.g., LUN).

opt_2 Driver-discretionary element.

opt_3 Driver-discretionary element.
RETURN VALUES

CONSTRAINTS

EXAMPLE

A SCSI probe example might consist of two routines.
scsi_probe_function() determines the next address to be probed and
scsi_probe() builds device files and actually tries to open devices.
Addressing information about the current node we’re trying to probe is
passed between the two routines via this structure. Elements of the
structure are used by scsi_probe() to build device files as follows:

```c
dev = ((major_num << 24) & 0xff000000);
dev |= ((probe_dev->instance << 16) & 0x00ff0000);
dev |= ((probe_dev->target << 12) & 0x0000f000);
dev |= ((probe_dev->opt_1 << 8) & 0x00000f00);
if ( (sctl_open(dev)) == 0 )
{
    /* Do an ioctl() on the device to get the ID information
    * for building the name, description, and id strings.
    */
}
```

SEE ALSO

wsio_register_probe_func(WSIO3)
NAME

wsio_query_supported_function(WSIO_DRV) – Return a pointer to a function supported by WSIO.

SYNOPSIS

#include <wsio/wsio.h>

void * wsio_query_supported_function (wsio_func_ptr_type_t func_id);

PARAMETERS

func_id  Identifies what function pointer a driver is interested in.

func_id  Function pointers returned

WSIO_GET_HANDLER_REG_FUNC  A WSIO function to register a driver’s event handler.

WSIO_GET_HANDLER_UNREG_FUNC  A WSIO function to unregister a driver’s event handler.

WSIO_GET_INSTALL_DRV_FUNC  A WSIO function to register a driver’s function.

WSIO_GET_MASK_REG_FUNC  A WSIO function to register a driver’s supported event mask.

DESCRIPTION

The wsio_query_supported_function() WSIO function queries WSIO to see if certain functions are supported. The purpose is that a driver can be installed on systems with different versions of wsio(11i or newer.) By doing this query, a driver can determine what functionality is supported and tailor its capabilities. If a function is supported, a pointer will be returned (otherwise a NULL); a driver can then call this function pointer. This mechanism will allow a driver to be able to link on multiple versions of wsio(11i or newer.)
RETURN VALUES

Returns a function pointer if supported, otherwise a NULL.

CONSTRAINTS

EXAMPLE

```c
static wsio_drv_info_t my_drv_info {
    ....
} 
void my_event_handler()
{
    ....
}
int my_install(void)
{
    wsio_install_handler_func_ptr_t handler_reg_ptr;

    if (wsio_install_driver(&my_drv_info) {
        /* Query wsio to get a pointer to its registration function */
        handler_reg_ptr = wsio_query_supported_function
                          (WSIO_GET_HANDLER_REG_FUNC);
        if (handler_reg_ptr) {
            /* Now register our event handler with wsio */
            ret = handler_reg_ptr(&my_drv_info,
                                  my_event_handler);
                ....
        }
    }
    ....
}
```

SEE ALSO

wsio_install_drv_event_handler(WSIO3),
wsio_install_drv_func(WSIO3),
wsio_reg_drv_capability_mask(WSIO5),
wsio_uninstall_drv_event_handler(WSIO3)
NAME

wsio_read_regXX(WSIO3) – Read XX bits from a mapped device register.

SYNOPSIS

#include <wsio/wsio.h>

void wsio_read_regXX (struct isc_table_type *isc,
                      wsio_reg_info_t *reg_info,
                      uint32_t offset,
                      uintXX_t *data);

PARAMETERS

isc Pointer to the driver's isc_table entry.

reg_info Pointer to information about the register to be read
           from. This register must have been successfully
           mapped by a call to wsio_map_reg().

offset The offset into the register to read from.

data Pointer to an appropriately sized data structure. This
       will contain the data upon completion of the read. The
       size should be selected based upon what size read is
       desired.

DESCRIPTION

The wsio_read_regXX() WSIO functions are called by device drivers to
read XX bits from a device register. The reg_info variable specifies
which register to read from, and offset indicates the correct location to
read from. The value XX refers to 8, 16, 32, or 64 and indicates the
amount of data to read. No endian swapping is performed for register
reads, so if the local bus and the host bus are of opposite endianness, the
driver must perform an endian swap.

RETURN VALUES

None.
CONSTRAINTS

EXAMPLE

wsio_reg_info_t *registers;
uint32_t data;

registers = wsio_get_all_registers(isc_entry);
if (registers == NULL) {
    /* No registers exist. Return an error */
    return(ERROR);
}

/* All of the devices registers are now contained in*
 * the 'registers' variable. They may be mapped as follows:
 */
if (wsio_map_reg(isc_entry,&registers[1]) != WSIO_OK) {
    return(ERROR);
}

/* The second device register (index 1 into the array) will*
 * now be mapped. */
wsio_read_reg32(isc_entry, &registers[1],
            (uint32_t)0x10, &data);
/* 'data' will now contain 32 bytes obtained from offset*
 * 0x10 into the second register. */

SEE ALSO

wsio_get_all_registers(WSIO3), wsio_map_reg(WSIO3),
wsio_unmap_reg(WSIO3), wsio_write_regXX(WSIO3)
NAME

wsio_reg_drv_capability_mask(WSIO) – Register a driver's capability mask.

SYNOPSIS

#include <wsio/wsio.h>

int wsio_reg_drv_capability_mask (struct isc_table_type *isc,
                                  wsio_event_mask_t event_mask);

PARAMETERS

isc Pointer to the driver's isc_table entry.

event_mask A mask representing the operations the driver will deal
            with in its event handler.

DESCRIPTION

The wsio_reg_drv_capability_mask() WSIO function is called by a
device driver to register with WSIO for operations the driver is capable of
handling. The mask is an OR of wsio_event_t and is for an instance of
the driver. This function should be called in a driver's attach routine
after isc_claim() is executed. The driver should register for the
different events in one single call. A new mask will supercede the
previous one.

If a platform does not support event handling, WSIO_HA_NA is returned
and the driver should ignore the error.

See the Interrupt Migration chapter in the Driver Development Guide for
relevant information.

RETURN VALUES

WSIO_OK Returned on success.

WSIO_INVALID_COMBIN_EVENTS
        Invalid combination of events.

WSIO_HA_NA Platform does not support HA events.
WSIO Reference Pages

tsio_reg_drv_capability_mask(WSIO5)

<table>
<thead>
<tr>
<th>WSIO_INVALID_EVENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid event.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>WSIO_INVALID_ISC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid isc.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>WSIO_NO_DRV_HANDLER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A driver’s event handler has not been installed.</td>
</tr>
</tbody>
</table>

CONSTRANTS

EXAMPLE

my_driver_attach() {
    int ret;
    wsio_event_mask_t my_mask = WSIO_EVENT_SUSPEND |
       WSIO_EVENT_RESUME | WSIO_EVENT_LBI_INTR_MIGR;
    isc_claim(isc, &my_drv_info);
    ret = wsio_reg_drv_capability_mask(isc, my_mask);
    if ( (ret!=WSIO_OK) && (ret!=WSIO_HA_NA)) {
        isc_unclaim(isc, &my_drv_info);
        //free resource as needed
    }
    // the rest of the attach chain
}

SEE ALSO

wsio_event_t(WSIO5), wsio_install_drv_event_handler(WSIO3),
wsio_query_supported_function(WSIO_DRV)
NAME

wsio_register_addr_probe(WSIO3) – Register a driver probe function.

SYNOPSIS

#include<sys/wsio.h>

void wsio_register_addr_probe (int (*) func(), char *drv_name);

PARAMETERS

func A pointer to the driver probe function.

drv_name An ASCII string indicating the name of the driver.

DESCRIPTION

The WSIO service wsio_register_addr_probe() is used to register an interface driver's probe function. The probe function is used by WSIO SCAN to look for I/O devices underneath interface cards claimed by the driver. The drv_name parameter must match the name field of the driver's drv_info_t structure.

The driver probe function must have the following calling syntax:

drv_addr_probe( void *handle,
    int (*dev_probe)(),
    drv_info_t *drv_info,
    void *probe_id,
    hw_path_t *hw_path,
    struct isc_table_type *isc,
    int probe_type,
    char *name,
    char *desc )

handle A pointer to a GIO structure. Drivers should not touch this structure.

dev_probe A pointer to a probe function registered via the WSIO service wsio_register_dev_probe() if one exists, else NULL.

drv_info A pointer to the driver's drv_info_t structure.
wsio_register_addr_probe(WSIO3)

probe_id
A unique identifier for the device found.

hw_path
When an input, the hardware path of the last device found. When an output, the hardware path of the next device to be found.

isc
A pointer to the isc_table_type structure of the interface card being probed.

probe_type
One of three types of probe, which are:

PROBE_FIRST  Find the first device underneath the interface card.
PROBE_NEXT  Find the next device after the previous one found.
PROBE_ADDRESS  Look for a device at the hardware address specified in hw_path.

name
A pointer to a string initialized with the device’s name such as scsi_disk. This information is used to match the device to a driver based on the information in the drv_path field of the wsio_drv_data_t structure.

desc
A pointer to a string with a description of the device.

When the driver probe function is called with a probe_type of PROBE_FIRST the function should find the first device underneath the interface card specified by the isc parameter. The hw_path parameter has the address of the interface card. When the driver probe function is called with a probe_type of PROBE_NEXT, the driver should find the next device after the last device found. The address of the last device is passed in the hw_path parameter. The driver then updates the hw_path parameter with the address of the new device. Each time the probe function reports a device it should return the additional information of probe_id, name and desc. The probe_id is a unique identifier. The name string should match the drv_path field in the wsio_drv_data_t structure of the driver that controls the device. desc is an ASCII string describing the device.
A probe function registered via `wsio_register_addr_probe()` can be used as a standalone probe function or in conjunction with another probe function registered by the service `wsio_register_dev_probe()`. In the latter case, the probe function registered via `wsio_registered_dev_probe()` is passed in as the second parameter to the first.

This method can be used by a driver stack where a device driver knows the syntax to talk to certain devices and an interface driver would know the range of addresses for a given I/O bus. The interactive driver would register an address probe function via `wsio_register_addr_probe()` and the device driver would register its probe function via `wsio_register_dev_probe()`.

If the probe function is used as a standalone probe function then a NULL value is passed in as the second parameter. Most drivers need only register a single probe function using the WSIO service `wsio_register_dev_probe()`.

**RETURN VALUES**

None

**EXAMPLE**

```c
int mydrv_install()
{
    (void)wsio_register_addr_probe(mydrv_probe, "mydrv")
    return(wsio_install_driver(&mydrv_wsio_info));
}
```

**CONSTRAINTS**

**SEE ALSO**

`drv_info(CDIO4), wsio_drv_data_t(WSIO4), wsio_register_dev_probe(WSIO3),`
NAME

wsio_register_dev_probe(WSIO3) – Register a driver probe function

SYNOPSIS

#include<sys/wsio.h>

int wsio_register_dev_probe (int type, int (*func)(), char *drv_name);

PARAMETERS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>Indicates what driver data the third parameter should match to. Valid values are:</td>
</tr>
<tr>
<td>IF_CLASS</td>
<td>The third argument drv_name is to be matched with the drv_path field of the wsio_drv_data_t structure.</td>
</tr>
<tr>
<td>DRV_NAME</td>
<td>The third argument, drv_name is to be matched with the name field of the drv_info_t structure.</td>
</tr>
<tr>
<td>func</td>
<td>A pointer to the driver probe function.</td>
</tr>
<tr>
<td>drv_name</td>
<td>An ASCII string indicating the name or class of the driver.</td>
</tr>
</tbody>
</table>

DESCRIPTION

The WSIO service wsio_register_dev_probe() is used to register a driver probe function. The driver probe function is used by WSIO SCAN to look for I/O devices beneath specific interface cards. Which cards to scan depend on the values of the first and third parameters. The third parameter, drv_name, is an ASCII string that is used to match the probe function to specific driver/interfaces cards. The first parameter, type, is used to indicate what driver information the ASCII string is to be matched to. If the parameter has the value IF_CLASS, it indicates the string should be matched to the drv_path field of the driver's wsio_drv_data_t structure. If the type parameter is set to the value of DRV_NAME, the third argument is matched with the name field of the driver's drv_info_t structure.
A value of `DRV_NAME` causes a tight pairing of the probe function to a particular driver since the probe is matched to the driver's name. A value of `IF_CLASS` is more general since several drivers may have the same `drv_path`. Probe functions registered via the service `wsio_register_dev_probe()` should have the following calling syntax:

```c
void *handle,
drv_info_t *drv_info,
void *probe_id,
hw_path_t *hw_path,
struct isc_table_type *isc,
int probe_type,
char *name,
char *desc
```

- **handle**: A pointer to an internal GIO structure. Drivers should not attempt to access it.
- **drv_info**: A pointer to the `drv_info_t` structure.
- **probe_id**: A unique identifier for the device found.
- **hw_path**: When an input, the hardware path of the last device found. When an output, the hardware path of the next device to be found.
- **isc**: A pointer to the `isc_table_type` structure of the interface card being probed.
- **probe_type**: The type of probe. The following types are supported:
  - `PROBE_FIRST`: Find the first device underneath the interface card.
  - `PROBE_NEXT`: Find the next device after the previous one found as indicated by the `hw_path` parameter.
  - `PROBE_ADDRESS`: Look for a device at the specific hardware address.
- **name**: A pointer to a string initialized with the device's name such as `scsi_disk`. This information is used to match the device to a driver on the information in the `drv_path`. 
desc  

A pointer to a string with the device description. This is driver dependent.

When the driver probe function is called with a probe_type of PROBE_FIRST the function should find the first device underneath the interface card specified by the isc parameter. The hw_path parameter has the address of the interface card. When the driver probe function is called with a probe_type of PROBE_NEXT the driver should find the next device after the last device found. The address of the last device is the last element of the hw_path parameter. The driver then updates the hw_path with the address of the new device. Each time the probe function reports a device it should return the additional information of probe_id, name and desc. The probe_id is a unique identifier. The name string should match the drv_path field in the wsio_drv_data_t structure of the driver that controls the device. desc is an ASCII string describing the device.

RETURN VALUES

0  
Successful completion.

-1  
Error.

EXAMPLE

```c
int mydrv_install()
{
    (void)wsio_register_dev_probe(DRV_NAME,
                                 mydrv_probe, "mydrv");
    return(wsio_install_driver(&mydrv_wsio_info));
}
```

CONSTRAINTS

SEE ALSO

`drv_info(CDIO4), wsio_drv_data_t(WSIO4), wsio_register_addr_probe(WSIO3),`
NAME

wsio_remap_dma_buffer(WSIO3) – Map pre-allocated IOVAs to new host ranges.

SYNOPSIS

```c
#include <wsio/wsio.h>

wsio_map_status_t wsio_remap_dma_buffer (  
  struct isc_table_type *isc,  
  void *dma_handle,  
  wsio_range_type_t range_type,  
  wsio_dma_map_t *host_range,  
  wsio_dma_map_t *io_range);
```

PARAMETERS

isc Pointer to the driver’s isc_table entry.

dma_handle DMA handle allocated using
  wsioAllocate_dma_handle().

range_type Indicates the type of host memory being mapped. It can be:
  KERNELSPACE Indicates host_range is a kernel virtual buffer.
  PHYSICAL Indicates host_range is a physical buffer.
  > 0 Indicates host_range is in user space, and this will be the space ID of the virtual address.

host_range Pointer to an address/length structure that contains information about the host space to map. If the mapping was only partially completed, this will contain information about the remaining space to be mapped when the call completes.

io_range Pointer to an address/length structure that will contain information about the I/O space that was mapped.
DESCRIPTION

The `wsio_remap_dma_buffer()` WSIO function is called by a device driver to map a new host memory address to existing I/O virtual addresses (IOVA). The IOVAs must have been previously allocated via a call to `wsio_map_dma_buffer()`, `wsio_remap_dma_buffer()`, or `wsio_fastmap_dma_buffer()`.

The `io_range` must use exactly the same number of mapping resources as the previous mapping. This can be ensured by making sure the buffers are page-aligned and of equal sizes.

All mappings will remain in effect until `wsio_unmap_dma_buffer()` or `wsio_remap_dma_buffer()` are called to remove or change them.

A callback function is not necessary for `wsio_remap_dma_buffer()` because DMA resources were allocated when the initial mapping took place.

`wsio_remap_dma_buffer()` can be called in a non-blocking context.

RETURN VALUES

- `WSIO_MAP_OK` Returned if the entire buffer has been mapped.
- `WSIO_MAP_E_HIGH_ADDR` Returned if the call failed because the device cannot reach the destination address.
- `WSIO_MAP_E_PARAMETER_ERROR` Returned if an invalid parameter has caused failure of the call.
- `WSIO_MAP_E_UNKNOWN_ERROR` Returned for hardware or other errors.

CONSTRAINTS
EXAMPLE

```c
void *dma_handle;
wsio_map_context_t dma_con;
wsio_dma_map_t host_range, new_host_range, io_range;

dma_handle = wsio_allocate_dma_handle(isc_entry);
wsio_init_map_context(&dma_con);

host_range.iov_base = host_virtual_address;
host_range.iov_len = dma_buffer_length;

if (wsio_map_dma_buffer(isc_entry,dma_handle,
                        dma_con,WSIO_DMA_OUTBOUND KERNELSPACE,
                        &host_range,&io_range)!= WSIO_MAP_OK) {
    return(ERROR);
}

new_host_range.iov_base = new_host_virtual_address;
new_host_range.iov_len = dma_buffer_length;

if (wsio_remap_dma_buffer(isc_entry,dma_handle,
                           KERNELSPACE,&new_host_range, &io_range
                            ) != WSIO_MAP_OK) {
    return(ERROR);
}

/* The host virtual buffer represented in the above
   code by ‘host_virtual_address’ was mapped. After that,
   a second host virtual buffer represented by
   ‘new_host_virtual_address’ was mapped using the
   same I/O virtual address as the initial mapping.
   This example doesn’t take into account a
   WSIO_MAP_W_PARTIAL return code being returned by the
   call to wsio_map_dma_buffer(). */
```
SEE ALSO

wsio_allocate_dma_handle(WSIO3),
wsio_allocate_shared_mem(WSIO3), wsio_dma_pass_thru(WSIO3),
wsio_fastmap_dma_buffer(WSIO3), wsio_free_dma_handle(WSIO3),
wsio_free_shared_mem(WSIO3), wsio_flush_shared_mem(WSIO3),
wsio_init_map_context(WSIO3), wsio_iova_to_phys(WSIO3),
wsio_map_dma_buffer(WSIO3),
wsio_set_dma_buffer_attributes(WSIO3),
wsio_set_dma_attributes(WSIO3), wsio_unmap_dma_buffer(WSIO3)
NAME

\texttt{wsio\_set\_description} (WSIO3) – Set the I/O tree node description for this driver.

SYNOPSIS

\begin{verbatim}
#include <wsio/wsio.h>

void wsio_set_description (struct isc_table_type *isc, char *description);
\end{verbatim}

PARAMETERS

\begin{itemize}
\item \textit{isc} Pointer to the driver's isc_table entry.
\item \textit{description} String containing the description.
\end{itemize}

DESCRIPTION

The \texttt{wsio_set_description}() WSIO function sets the I/O tree node description of a driver. This functionality is provided because at times, WSIO can not make appropriate decisions as to how to describe a driver at driver install time. This \textit{description} is used by the ioscan command that is provided to system users, and therefore is essential to describing the system.

Many drivers may not need to call this function. To decide if it is necessary, a driver writer should make a decision based upon whether the ioscan output for that device driver is understandable.

RETURN VALUES

None.

CONSTRAINTS

EXAMPLE

\begin{verbatim}
wsio_set_description (isc_entry,"My Driver Description");
\end{verbatim}
WSIO Reference Pages

wsio_set_description(WSIO3)

SEE ALSO
NAME

wsio_set_dma_attributes(WSIO3) – Associate DMA hints with a DMA handle.

SYNOPSIS

#include <wsio/wsio.h>

wsio_map_status_t wsio_set_dma_attributes(
    struct isc_table_type *isc,
    void *dma_handle,
    wsio_dma_attribute_t attribute,
    wsio_dma_attr_param_t param);

PARAMETERS

isc Pointer to the driver’s isc_table entry.
dma_handle DMA handle allocated using
    wsio_allocate_dma_handle().
attribute Indicates which hint to set for the device associated
    with dma_handle. The possible attributes are:

<table>
<thead>
<tr>
<th>Hint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSIO_DMA_ATTR_ADDR_WIDTH</td>
<td>Bits of addressing supported by the device. This is used to determine whether a device can DMA directly to memory buffers. Default = 32</td>
</tr>
<tr>
<td>WSIO_DMA_ATTR_ALIGNMENT</td>
<td>Byte alignment of DMA buffer required for device. Default = H/W Dep.</td>
</tr>
</tbody>
</table>
WSIO_DMA_ATTR_ATM
ATM hint; used by hardware in some implementations.

- 0 = not ATM
- 1 = ATM48 (optimize for 48-byte transfers)
- 2 = ATM192 (optimize for 192-byte transfers)

Default = 0

WSIO_DMA_ATTR_CALLBACK
Specifies a function to call when resources become available.

Default = NULL

WSIO_DMA_ATTR_CALLBACK_ARG
Specifies an argument to the callback function.

Default = 0

WSIO_DMA_ATTR_FLUSH_ON_USE
Specifies the cacheline should be flushed from any intermediate buffers as soon as it is referenced. This inhibits any coalescing of data by bus bridges.

Default = 0

WSIO_DMA_ATTR_IGN_ALIGN
Specifies the mapping service should not handle cacheline fragments in a special way.

Default = 0
**WSIO_DMA_ATTR_INTERLEAVE**

IOVA allocation model

- 0 = DMA streams are normally interleaved (mass storage)
- 1 = DMA streams are normally not interleaved (networking)
- 3 = DMA buffers are static and accessed randomly (lowfat)

Default = 0

**WSIO_DMA_ATTR_PREFETCH**

Specifies how aggressively hardware should prefetch for outbound DMA.

- 0 = no prefetch
- 1 = moderate prefetch
- 2 = aggressive prefetch

Default = 1

**WSIO_DMA_ATTR_SAFE**

Specifies that the most conservative coherency model should be used for inbound DMA. Inhibits semi-coherent transactions such as WRITE_PURGE unless it is guaranteed that no data in processor caches will be lost.

- 1 = ON
- 2 = OFF

Default = 0

**WSIO_DMA_ATTR_TXN_SIZE**

Specifies the default transaction size used by the device. This is used by hardware to optimize conversion of transactions between busses.

Default = H/W Dep.
WSIO_DMA_ATTR_INBOUND
   DMA buffers will be used exclusively for inbound DMA.
   Default = 0

WSIO_DMA_ATTR_OUTBOUND
   DMA buffers will be used exclusively for outbound DMA.
   Default = 0

WSIO_DMA_ATTR_STABLE
   Indicates that the buffer will not be modified by another entry while mapped for DMA.
   
   1 = true for data buffers
   0 = false for control structures

   Default = 0

param
   Driver defined parameter passed as the first parameter to isr. Typically, isc is passed as arg1.

DESCRIPTION

The wsio_set_dma_attributes() WSIO function is used to associate DMA transaction hints and attributes with a specific DMA handle. These hints override any hints set via wsio_dma_set_device_attributes(), and are overridden by some hints passed in as parameters to wsio_map_dma_buffer(). wsio_set_dma_attributes() can be called in a non-blocking context.

If WSIO_DMA_ATTR_INTERLEAVE is set to 1, a subsequent wsio_allocate_shared_mem() or wsio_map_dma_buffer() can only successfully request a buffer of a maximum of one page (4K) in length and this buffer cannot cross a page boundary. This is the limitation placed by the underlying platform. If a larger buffer is desirable, use the default value of 0. This larger buffer can be used for control structures rather than packet DMAs.

Do not call wsio_set_dma_attributes() to set the WSIO_DMA_ATTR_INTERLEAVE to the default value of 0. The call will fail.
RETURN VALUES

WSIO_MAP_OK      Successful completion.
WSIO_MAP_E_PARAMETER_ERROR
                    Returned if an invalid parameter has caused failure of
                    the call.

CONSTRAINTS

EXAMPLE

if (wsio_set_dma_attributes(isc_entry,dma_handle,
                            WSIO_DMA_ATTR_INTERLEAVE,1) != WSIO_MAP_OK) {
    /* There was a parameter error */
    return(ERROR);
} else {
    /* DMA streams are now not normally interleaved for
     * all DMA associated with dma_handle
     */
    return(0);
}

SEE ALSO

wsio_allocate_dma_handle(WSIO3),
wsio_allocate_shared_mem(WSIO3), wsio_dma_pass_thru(WSIO3),
wsio_dma_set_device_attributes(WSIO3),
wsio_fastmap_dma_buffer(WSIO3), wsio_free_dma_handle(WSIO3),
wsio_free_shared_mem(WSIO3), wsio_flush_shared_mem(WSIO3),
wsio_init_map_context(WSIO3), wsio_iova_to_phys(WSIO3),
wsio_map_dma_buffer(WSIO3), wsio_remap_dma_buffer(WSIO3),
wsio_unmap_dma_buffer(WSIO3)
NAME

*wsio_set_dma_callback*(WSIO3) – Set the callback function and argument for DMA

SYNOPSIS

```c
#include <wsio/wsio.h>

wsio_map_status_t wsio_set_dma_callback (struct isc_table_type *isc,
                                        void *dma_handle,
                                        void *func,
                                        void *arg);
```

PARAMETERS

- *isc* Pointer to the driver’s *isc_table* entry.
- *dma_handle* DMA handle allocated using `wsio_allocate_dma_handle`.
- *func* Function pointer to be used as a callback.
- *arg* Argument to be passed to the callback when it is called.

DESCRIPTION

The `wsio_set_dma_callback()` WSIO function is called by a device driver to set up the callback function for certain DMA transactions. If resources are not available when `wsio_map_dma_buffer()`, `wsio_fastmap_dma_buffer()`, or `wsio_allocate_shared_memory()` are called, and a callback function is setup, `WSIO_MAP_W_CALLBACK` is returned to the caller, and the callback function will be called when resources become available. This eliminates the need to continuously loop to attempt to obtain DMA resources.

RETURN VALUES

- `WSIO_MAP_OK` Successful completion.
- `WSIO_MAP_E_PARAMETER_ERROR` Returned if an invalid parameter has caused failure of the call.
CONRAINTS

EXAMPLE

if (wsio_set_dma_callback(isc_entry, dma_handle,
        callback_func, callback_arg)
    != WSIO_MAP_OK) {
    /* There was a parameter error */
    return(ERROR);
}

SEE ALSO

wsio_allocate_dma_handle(WSIO3),
wsio_allocate_shared_mem(WSIO3), wsio_dma_pass_thru(WSIO3),
wsio_fastmap_dma_buffer(WSIO3), wsio_free_dma_handle(WSIO3),
wsio_free_shared_mem(WSIO3), wsio_flush_shared_mem(WSIO3),
wsio_init_map_context (WSIO3), wsio_map_dma_buffer(WSIO3),
wsio_remap_dma_buffer(WSIO3),
wsio_set_device_attributes(WSIO3),
wsio_set_dma_attributes (WSIO3) wsio_unmap_dma_buffer(WSIO3),
NAME

`wsio_uninstall_driver` (WSIO3) – Uninstall a driver's header structure from the WSIO CDIO.

SYNOPSIS

```c
int wsio_uninstall_driver (wsio_drv_info_t * wsio_drv_info);
```

PARAMETERS

`wsio_drv_info` Pointer to the driver's `wsio_info_t` structure.

DESCRIPTION

The `wsio_uninstall_driver()` WSIO function uninstalls a driver's header structure from the WSIO CDIO. `wsio_uninstall_driver()` is called by the driver prior to unloading.

RETURN VALUES

0 Successful completion.
<>0 Error.

CONSTRAINTS

SEE ALSO

`wsio_install_driver` (WSIO3)
NAME

wsio_uninstall_drv_event_handler(WSIO3) – Uninstall a driver’s event handler

SYNOPSIS

#include <wsio/wsio.h>

int wsio_uninstall_drv_event_handler (wsio_drv_t * drv_info,
                                          wsio_drv_event_handler_t drv_handler);

PARAMETERS

drv_info Pointer to the driver’s wsio_drv_info_t structure.
drv_handler Function pointer to the driver’s event handler

DESCRIPTION

The wsio_uninstall_drv_event_handler() WSIO function is called by a driver to uninstall its event handler. The call is made if a driver needs to clean up after an init failure, online deletion, or DLKM.

RETURN VALUES

WSIO_OK Returned on success.
WSIO_DRV_NOT_FOUND Could not find the driver (driver has not registered yet).
WSIO_ERROR Returned on failures.
WSIO_INFO_NULL Drv_info is NULL.
WSIO_NO_DRV_HANDLER Invalid drive handler.

CONSTRAINTS
EXAMPLE

```c
static wsio_drv_info_t my_drv_info {
    &my_info,
    &my_ops,
    &my_data,
    WSIO_DRV_CURRENT_VERSION,
}
my_cleanup() {
    int ret;
    ret = wsio_uninstall_drv_event_handler(
        &my_drv_info, my_handler);
    ...
    ret = wsio_uninstall_driver(&my_drv_info);
}
```

SEE ALSO

wsio_install_drv_event_handler(WSIO3)
NAME

wsio_unmap_cfg_handle(WSIO3) – Release a configuration space handle.

SYNOPSIS

#include <wsio/wsio.h>

int wsio_unmap_cfg (struct isc_table_type *isc,
                     wsio_addr_handle_t *cfg_handle);

PARAMETERS

isc Pointer to the driver’s isc_handle entry.

cfg_handle Pointer to the configuration handle.

DESCRIPTION

The wsio_map_cfg_handle() WSIO function is called by device drivers to release a configuration space handle.

RETURN VALUES

WSIO_OK Indicates a handle was successfully returned in cfg_handle.

WSIO_ERROR Indicates there was a parameter error and the handle has not been released.

CONSTRAINTS
EXAMPLE

```c
wsio_addr_handle_t handle;

if (wsio_map_cfg_handle(isc_entry,&handle) != WSIO_OK) {
    /* Error obtaining configuration space handle. Return
    * an error code */
    return(ERROR);
}

if (wsio_unmap_cfg(isc_entry,&handle) != WSIO_OK) {
    /* Error releasing configuration space handle. Return an
    * error code */
    return(ERROR);
}

/* The configuration space handle is no longer valid */
```

SEE ALSO

wsio_map_cfg_handle(WSIO3), wsio_cfg_inXX(WSIO3),
wsio_cfg_outXX(WSIO3)
NAME

wsio_unmap_dma_buffer(WSIO3) – Remove a DMA packet mapping.

SYNOPSIS

#include <wsio/wsio.h>

wsio_map_status_t wsio_unmap_dma_buffer (
    struct isc_table_type *isc,
    void *dma_handle,
    wsio_dma_map_t *io_range);

PARAMETERS

isc Pointer to the driver's isc_table entry.
dma_handle DMA handle allocated using
    wsio_allocate_dma_handle().
io_range Pointer to an address/length structure that contains
    the information about the mapping to remove.

DESCRIPTION

The wsio_unmap_dma_buffer() WSIO function is called by a device
driver to remove a packet DMA memory mapping, and to free all
resources associated with such a mapping. io_range must have been
obtained via a previous call to wsio_map_dma_buffer(),
wsio_remap_dma_buffer(), or wsio_fastmap_dma_buffer().

wsio_unmap_dma_buffer() can be called in a non-blocking context.

RETURN VALUES

WSIO_MAP_OK Successful completion.

WSIO_MAP_E_PARAMETER_ERROR
    Returned if an invalid parameter has caused failure of
    the call. The buffer will not be unmapped.
EXAMPLE

```c
void *dma_handle;
wsio_map_context_t dma_con;
wsio_dma_map_t host_range, new_host_range, io_range;

dma_handle = wsio_allocate_dma_handle(isc_entry);
wsio_init_map_context(&dma_con);

host_range.iov_base = host_virtual_address;
host_range.iov_len = dma_buffer_length;

if (wsio_map_dma_buffer(isc_entry,dma_handle,
                      dma_con,WSIO_DMA_OUTBOUND,KERNELSPACE,&host_range,
                      &io_range)!= WSIO_MAP_OK) {
    return(ERROR);
}

if (wsio_unmap_dma_buffer(isc_entry,dma_handle,&io_range
                      ) != WSIO_MAP_OK) {
    /* There must have been a parameter error. */
    return(ERROR);
}

/* The host virtual buffer represented in the above
 * code by 'host_virtual_address' was mapped and then
 * immediately unmapped. This example doesn’t take into
 * account a WSIO_MAP_W_PARTIAL return code being
 * returned by the call to wsio_map_dma_buffer().
 */
```
SEE ALSO

wsio_allocate_dma_handle(WSIO3),
wsio_allocate_shared_mem(WSIO3), wsio_dma_pass_thru(WSIO3),
wsio_fastmap_dma_buffer(WSIO3), wsio_free_dma_handle(WSIO3),
wsio_free_shared_mem(WSIO3), wsio_flush_shared_mem(WSIO3),
wsio_init_map_context(WSIO3), wsio_iova_to_phys(WSIO3),
wsio_map_dma_buffer(WSIO3), wsio_remap_dma_buffer(WSIO3),
wsio_set_device_attributes(WSIO3),
wsio_set_dma_attributes(WSIO3)
NAME

wsio_unmap_port (WSIO) – Unmap an I/O port.

SYNOPSIS

#include <wsio/wsio.h>

int wsio_unmap_port (struct isc_table_type *isc,
    int32_t port_addr,
    size_t size,
    wsio_addr_handle_t port_handle);

PARAMETERS

isc Pointer to the driver’s isc_table entry.
port_addr Address of the port obtained from
    wsio_get_ioports().
size Size of the port.
port_handle Handle obtained from wsio_map_port().

DESCRIPTION

The wsio_unmap_port() WSIO function removes the mapping done by
wsio_map_port(). After this is called, the port should not be accessed by
the driver anymore.

RETURN VALUES

WSIO_OK Successful completion.
WSIO_ERROR Parameter error.

CONSTRAINTS
EXAMPLE

wsio_addr_handle_t port_handle;
wsio_iop_t ioports_array[10];
/* An array with enough space for all ports needs to be allocated */

if (wsio_get_ioports(isc_entry,10,ioports_array) != WSIO_OK) {
    /* There was a problem obtaining the ports */
    return(ERROR);
}

if (wsio_map_port(isc_entry,ioports_array[0].addr,
                     ioports_array[0].size, &port_handle) != WSIO_OK) {
    /* There was an error mapping the port */
    return(ERROR);
}

/* Now unmap the port */
if (wsio_unmap_port(isc_entry,ioports_array[0].addr,
                     ioports_array[0].size, port_handle) != WSIO_OK) {
    /* There was an error unmapping the port */
    return(ERROR);
}

SEE ALSO

wsio_get_ioports(WSIO3), wsio_map_port(WSIO3),
wsio_port_inXX(WSIO3), wsio_port_outXX(WSIO3)
NAME

wsio_unmap_reg(WSIO3) – Unmap a device register.

SYNOPSIS

#include <wsio/wsio.h>

int wsio_unmap_reg (struct isc_table_type *isc,
                    wsio_reg_info_t *reg_info);

PARAMETERS

isc Pointer to the driver's isc_table entry.
reg_info Pointer to information about the register to be unmapped. This should be the same structure used to map the register.

DESCRIPTION

The wsio_unmap_reg() WSIO function removes the mapping done by wsio_map_reg().

RETURN VALUES

WSIO_OK Successful completion.
WSIO_ERROR Could not unmap the register.

CONSTRAINTS
EXAMPLE

wsio_reg_info_t *registers;

registers = wsio_get_all_registers(isc_entry);
if (registers == NULL) {
    /* No registers exist. Return an error */
    return(ERROR);
}

if (wsio_map_reg(isc_entry,&registers[1]) != WSIO_OK) {
    return(ERROR);
}

/* The second device register (index 1 into the array) will now
 * be mapped.
 */

if (wsio_unmap_reg(isc_entry,&registers[1]) != WSIO_OK) {
    return(ERROR);
}

/* The second device register will now be unmapped */

SEE ALSO

wsio_get_all_registers(WSIO3), wsio_map_reg(WSIO3),
wsio_read_regXX(WSIO3), wsio_write_regXX(WSIO3)
NAME

`wsio_unregister_dev_probe` (WSIO3) – Unregisters a driver probe function.

SYNOPSIS

```c
int wsio_unregister_dev_probe (int type, char * name);
```

PARAMETERS

- **type**
  - Indicates what driver data the second parameter should be matched to. Valid values are:
  - `IF_CLASS`
    - The second argument, `name` is to be matched with the `drv_path` field of the `wsio_drv_data_t` structure.
  - `DRV_NAME`
    - The second argument, `name` is to be matched with the `name` field of the `drv_info_t` structure.
- **name**
  - An ASCII string indicating the name or class of the driver.

DESCRIPTION

The WSIO service `wsio_unregister_dev_probe()` is used to unregister a driver probe function that was previously registered by a call to `wsio_register_dev_probe()`. The `type` and `name` parameters passed to `wsio_unregister_dev_probe()` should be the same as the first and third arguments passed to `wsio_register_dev_probe()` when the driver registered the probe function.

The first parameter, `type`, is used to indicate what driver information the ASCII string is to be matched to. If the parameter has the value `IF_CLASS`, it indicates the string should be matched to the `drv_path` field of the driver's `wsio_drv_data_t` structure. If the `type` parameter is set to the value `DRV_NAME`, the second argument is matched with the `name` field of the driver's `drv_info_t` structure. The second parameter, `name`, is an ASCII string with the driver's name or path.

The service is used primarily by DLKM type drivers in their unload routines.
RETURN VALUES

0  Successfully found and deleted the driver
-1  Not found

CONSTRAINTS

EXAMPLE

```c
int mydrv_unload( void *arg)
{
    int ret;
    struct isc_table_type *isc;
    void (token, *priv_ptr);

    /*******************************************************************************
    * Remove the attach function from the DLKM attach list
    *******************************************************************************
    if (mod_wsio_attach_list_remove (MOD_WSIO_CORE,
                        &module_name_core_attach))
        return (ENXIO);
    /*******************************************************************************
    * Unregister the device probe
    *******************************************************************************
    (void) wsio_unregister_dev_probe(IF_CLASS,"mydrv_path");
        "probe_name";
    /*******************************************************************************
    * Uninstall the driver. If it fails, go back to the
    * load state and undo what has been done in the
    * unload routine.
    *******************************************************************************
    if(wsio_uninstall_driver(&module_name_wsio_info)) {
        return (ENXIO);
    }
    return(0);
}
```
SEE ALSO

wsio_drv_data_t(WSIO4), wsio_drv_info(WSIO4),
wsio_register_dev_probe(WSIO3)
NAME

`wsio_write_regXX(WSIO3)` – Write XX bits to a mapped register.

SYNOPSIS

```c
#include <wsio/wsio.h>

void wsio_write_regXX (struct isc_table_type *isc,
                        wsio_reg_info_t *reg_info,
                        uint32_t offset,
                        uintXX_t data);
```

PARAMETERS

- **isc**
  Pointer to the driver's `isc_table` entry.

- **reg_info**
  Pointer to information about the register to be written to. This register must have been successfully mapped by a call to `wsio_map_reg()`.

- **offset**
  The offset into the register to write to.

- **data**
  Appropriately sized piece of data to be written. The size should be selected based upon what size write is desired.

DESCRIPTION

The `wsio_write_regXX()` WSIO functions are called by device drivers to write XX bits to a device register. The `reg_info` variable specifies which register to write to, and `offset` indicates the correct location to write to. The value XX refers to 8, 16, 32, or 64 and indicates the amount of `data` to write. No endian swapping is performed for register writes, so if the local bus and the host bus are of opposite endianness, the driver must perform an endian swap.

RETURN VALUES

None.
CONSTRANTS

EXAMPLE

wsio_reg_info_t *registers;
uint32_t data = 0x5a;

registers = wsio_get_all_registers(isc_entry);
if (registers == NULL) {
    /* No registers exist. Return an error */
    return(ERROR);
}

/* All of the devices registers are now contained in the
 * 'registers' variable. They may be mapped as follows:
 */
if (wsio_map_reg(isc_entry, &registers[1]) != WSIO_OK) {
    return(ERROR);
}

/* The second device register (index 1 into the array) will now
 * be mapped. */
wsio_write_reg32(isc_entry, &registers[1],
    (uint32_t)0x10, data);

/* The 32 bytes consisting of 0x5a will now be written to
 * offset 0x10 into the second register. */

SEE ALSO

wsio_get_all_registers(WSIO3), wsio_map_reg(WSIO3),
wsio_read_regXX(WSIO3) wsio_unmap_reg(WSIO3)