WSIO Reference Pages

This chapter contains reference pages describing routines and data structures used by drivers to communicate with the WSIO CDIO.
**NAME**

dma_cleanup (WSIO3) – Clean up from a DMA transfer

**SYNOPSIS**

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

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 to recover from a DMA transfer.

See eisa_dma_cleanup(EISA3) for further details.

**RETURN VALUES**

dma_cleanup() returns an indeterminate value. Treat it as a void functions.

**CONSTRAINTS**

**SEE ALSO**

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; /* host offset address */
  space_t spaddr; /* host space address */
  int count;
};

#include <sys/esa.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 DMA_CASCADE 0x80
#define DMA_8BYTE 0x100
#define DMA_16WORD 0x200
WSIO Reference Pages
Functions, Macros and Structures

#define DMA_16BYTE 0x400
#define DMA_32BYTE 0x800
#define DMA_READ 0x1000
#define DMA_WRITE 0x2000

/* bit definitions for flags field - bits cleared */
/* are defaults */
#define ADDR_CHAIN 0x1
/* addr is pointer to chain of I/O’s */
#define NO_CHECK 0x2
/* don’t perform error checking */
#define NO_ALLOC_CHAIN 0x4
/* allocate chain for addr/count’s */

/* errors returned by dma_setup */
#define UNSUPPORTED_FLAG -1
#define RESOURCE_UNAVAILABLE -2
#define BUF_ALIGN_ERROR -3
#define MEMORY_ALLOC_FAILED -4
#define TRANSFER_SIZE -5
#define INVALID_OPTIONS_FLAGS -6
#define ILLEGAL_CHANNEL -7

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

dma_setup(WSIO3) – Set up and return information for a DMA transfer

SYNOPSIS

#include <wsio/wsio.h>

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

PARAMETERS

isc Pointer to an ISC structure.

dma_parms Pointer to a dma_parms structure.

DESCRIPTION

The dma_setup() WSIO function performs the required setup and returns the necessary information for a DMA transfer.

RETURN VALUES

dma_setup() returns the following values:

  0 Successful completion.

<>0 Error. Values depend upon the underlying CDIO function that is called.

CONSTRAINTS

SEE ALSO

dma_cleanup(WSIO3)
NAME

driver_addr_probe(WSIO_DRV) – Provide an interface driver specific probing function.

SYNOPSIS

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

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);

PARAMETERS

this_node A pointer to an io_tree_node struct.
dev_probe Probe function registered by device driver to be called by driver_addr_probe().
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 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 `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**

The `driver_addr_probe` routine is expected to return the following 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**

**WARNINGS**

The parameters shown in the call to `driver_addr_probe` are only valid if this routine was previously registered with `wsio_register_addr_probe`.

**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

Of course, if you call the function with a bad card_id it might not return anything, so you should do something to check for a failure and handle it properly.
returned by the next `driver_attach` routine in the chain. The end-of-chain function returns a unique completion code.

LP64 Considerations

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);
  }
}
```

CONSTRAINTS

SEE ALSO

`get_new_isc(WSIO3), isc_table_type(KER4)`
NAME

driver_class_probe (WSIO_DRV) - Provide a driver-specific probing function.

SYNOPSIS

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

driver_class_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 * dev_class, char * desc);

PARAMETERS

this_node A pointer to an io_tree_node struct.
drv_info The drv_info_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.
dev_class 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 `driver_class_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 `wsio_register_probe_func()` routine as part of your `driver_attach()` routine. Commonly, `driver` is replaced by your driver's name.

See HP-UX Driver Development Guide for details;

RETURN VALUES

The `driver_probe()` routine is expected to return the following values:

- **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.

CONSTRAINTS

SEE ALSO

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

driver_close(WSIO_DRV) - Close a device

SYNOPSIS

int driver_close (dev_t dev, int flag);

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).

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

driver_dev_init(WSIO_DRV) - Initialize a device driver

SYNOPSIS

int driver_dev_init();

PARAMETERS

None.

DESCRIPTION

The 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, dev_init(), with the driver_install() routine. Commonly, driver is replaced by your driver's name.

See HP-UX Driver Development Guide for details;

RETURN VALUES

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

CONSTRAINTS

SEE ALSO

driver_install(WSIO_DRV)
NAME

driver_if_init(WSIO_DRV) - Initialize interface driver

SYNOPSIS

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();

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

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

PARAMETERS

dev The device number of the associated device.

cmd The command word described in DESCRIPTION.

arg_ptr Pointer to the command's arguments, if any.

flag The file access flags. Most drivers ignore this parameter.

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

driver_ioctl() is expected to return the following 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
SEE ALSO

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

`driver_isr(WSIO_DRV)` - Execute device interrupt in interrupt context

SYNOPSIS

```
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

- `driver_isr()` is expected to return the following 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

```c
#include <sys/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.


RETURN VALUES

- `driver_minor_build()` is expected to return the following 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(WSIO4)
NAME

driver_minphys(WSIO_DRV) – Driver specific transfer size adjustment

SYNOPSIS

void driver_minphys (struct buf *buf);

PARAMETERS

bp Transfer information structure;

DESCRIPTION

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

The driver_minphys() WSIO function adjusts the bp->b_bcount field of the buf structure passed in.

RETURN VALUES

driver_minphys() is a void function.

EXAMPLES

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

void mydriver_minphys(struct buf *bp) {
    if (bp->b_bcount > NBPG)
        bp->b_bcount = NBPG;
}

CONSTRAINTS

SEE ALSO

minphys(KER2)
NAME

driver_open(WSIO_DRV) – Open a device

SYNOPSIS

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.

CONSTRANTS

SEE ALSO

`drv_ops(CDIO4), open(2)`
NAME

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

SYNOPSIS

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

driver_psize() returns the following 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

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

driver_read() and driver_write() are executed to return the following 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(CDOI4), physio(KER2), uiomove(KER2)
NAME

\texttt{driver\_select(WSIO\_DRV)} - Test I/O completion on a device

SYNOPSIS

\begin{verbatim}
int driver\_select (dev_t dev, int flag);
\end{verbatim}

PARAMETERS

\begin{itemize}
  \item \texttt{dev} \hspace{1cm} The device number.
  \item \texttt{flag} \hspace{1cm} The type of readiness to test, according to the following values:
    \begin{itemize}
      \item \texttt{FREAD} \hspace{1cm} Read
      \item \texttt{FWRITE} \hspace{1cm} Write
      \item 0 \hspace{1cm} Exception conditions
    \end{itemize}
\end{itemize}

DESCRIPTION

The \texttt{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 \texttt{d\_select} field of the \texttt{drv\_ops} structure. Commonly, \texttt{driver} is replaced by your driver's name.

See HP-UX Driver Development Guide for details;

RETURN VALUES

\texttt{driver\_select()} is expected to return the following values:

\begin{itemize}
  \item \texttt{<>0} \hspace{1cm} 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 \texttt{select()} returns to the user.
  \item \texttt{0} \hspace{1cm} False. The device or driver is not ready for read or write or no exception condition was found. \texttt{select()} puts the calling process to sleep until the condition becomes true. The driver must inform the system when this condition becomes true.
\end{itemize}

If the \texttt{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

\texttt{driver\_strategy} (WSIO\_DRV - Execute block read or write for character or block devices

SYNOPSIS

\texttt{void driver\_strategy (struct buf * bp);} 

PARAMETERS

\texttt{bp} \hspace{1cm} A pointer to a \texttt{buf} structure, which contains all the information that the \texttt{driver\_strategy()} routine needs to process the request.

DESCRIPTION

The \texttt{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 \texttt{driver\_strategy} field of the \texttt{drv\_ops} structure. For a character device, you pass the name as a parameter to \texttt{physio()}. Commonly, \texttt{driver} is replaced by your driver's name.

See HP-UX Driver Development Guide for details;

RETURN VALUES

None.

CONSTRAINTS

SEE ALSO

\texttt{physio(KER2)}
NAME

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

SYNOPSIS

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

driver_read() and driver_write() are executed to return the following 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

_free_isc(WSIO3) - Free a driver's ISC entry

SYNOPSIS

#include <sys/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

free_isc() returns the following 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>
Functions, Macros and Structures

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

get_new_isc() returns the following 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_iodone. 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.
FUNCTIONS, MACROS AND STRUCTURES

WARNINGS

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

SEE ALSO

biodone(KER2), biowait(KER2), iowait(WS103)
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

iowait() returns the following 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

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 <sys/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
type, 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 == MYDEVICE_HW_ID) {
        /*
         * Specify the interface init function that is
         * called for each claimed ISC entry after the
         * attach chain processing has completed.
         */
        isc->gfw->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

isrlink(WSIO3) - Register an interrupt service routine

SYNOPSIS

```c
int isrlink (struct isc_table_type *isc, int (*isr)(),
            int irq_line, long arg1, long arg2);
```

PARAMETERS

- `isc`  Pointer to the ISC structure for the driver.
- `isr`  Pointer to the driver's interrupt service routine.
- `irq_line`  Interrupt request line asserted by the device. For PCI devices this should be -1 allowing WSIO services to determine the interrupt request line being used.
- `arg1`  Driver defined parameter passed as the first parameter to `isr`. Typically, `isc` is passed as `arg1`.
- `arg2`  Driver defined parameter passed as the second parameter to `isr`.

DESCRIPTION

The `isrlink()` WSIO function registers an interrupt service routine (ISR). `isrlink()` is typically called in the `driver_if_init()` function, if specified by `driver_attach()` or in `driver_attach()`.

When `isrlink()` returns, interrupts for the assigned `irq_line` are enabled. The driver should be prepared to handle an interrupt from its device or another device sharing the `irq_line`.

RETURN VALUES

`isrlink()` returns the following values:

- 0  Successful completion.
- WSIO_ERROR  Error.
CONSTRAINTS

EXAMPLE

```c
static int
mydrv_if_init(struct isc_table_type *isc)
{
    ...
    mydrv_reset_hw(isc);
    return isrlnk(isc, mydrv_isr, -1, (long)isc, 0L);
}
```

SEE ALSO

`driver_attach(WSIO_DRV), driver_if_init(WSIO_DRV), driver_isr(WSIO_DRV), isrunlink(WSIO3).`
NAME

`isrunlink(WSIO3)` - Remove the ISR registered by `isrlink()`

SYNOPSIS

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

int isrunlink (struct isc_table_type *isc, int (*isr)(), int irq_line, long arg1, long arg2);
```

PARAMETERS

- **isc**: Pointer to the ISC structure for the driver.
- **isr**: Pointer to the driver’s interrupt service routine.
- **irq_line**: Interrupt request line asserted by the device. For PCI devices this should be -1, allowing WSIO services to determine the interrupt request line used.
- **arg1**: Driver defined parameter passed as the first parameter to ISR.
- **arg2**: Driver defined parameter passed as the second parameter to ISR.

DESCRIPTION

The `isrunlink()` WSIO function removes the `isr` registered by `isrlink()`. This function should be called before a driver is unloaded but after the device has been quiesced.

RETURN VALUES

- `isrunlink()` returns the following values:
  - **0**: Successful completion.
  - **WSIO_ERROR**: Error.

CONSTRAINTS
SEE ALSO

isrlink(WSIO3)
NAME

_m_instance(WSIO3) – Get the driver instance field from the device number

SYNOPSIS

#include <sys/io.h>

#define m_instance(dev) ((int)(((unsigned) (dev)) >> 16) & 0xff)

PARAMETERS

- **dev**
  - The device number of a device.

DESCRIPTION

The _m_instance() WSIO macro returns the driver instance field from the device number.

RETURN VALUES

CONSTRAINTS

SEE ALSO
NAME

m_wsio_funcnum(WSI03) – Get the number of an interface card function

SYNOPSIS

int m_wsio_funcnum (dev_t dev, wsio_drv_info_t * drv_hdr_ptr);

PARAMETERS

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

DESCRIPTION

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

RETURN VALUES

CONSTRAINTS

SEE ALSO
NAME

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

SYNOPSIS

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 A 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

Chapter 4
NAME

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

SYNOPSIS

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 A 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
NAME

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

SYNOPSIS

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

mod_wsio_attach_list_add() returns the following 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(WSIO) – Remove the driver attach function pointer to the specified WSIO attach list.

SYNOPSIS

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

mod_wsio_attach_list_remove() returns the following values:

0 Successful completion.
1 Error.

CONSTRAINTS
/* 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_allocate_shared_memory (WSIO) – Allocate and map contiguous memory used for continuous DMA.

SYNOPSIS

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

shmemp_status_t wsio_allocate_shared_memory (
    struct isc_table_type* isc, size_t size,
    caddr_t* iova, caddr_t* vaddr,
    wsio_shmem_attr_t type);
```

PARAMETERS

isc Pointer to an ISC table entry.
size Size in bytes or memory to be allocated.
iova Pointer to the returned I/O virtual address.
vaddr Pointer to the returned virtual address.
type Bit mask of the requested memory attributes.

DESCRIPTION

The `wsio_allocate_shared_memory()` function allocates and maps contiguous memory used for continuous DMA. For packet DMA (short lived DMA typical of I/O transactions), temporary mappings should be done using `wsio_map()`. Continuous DMA is intended for accesses by a device on a continuous basis, typically for device control and status.

Platforms that implement I/O virtual addressing will allocate memory contiguous in I/O virtual address space; platforms that do not will allocate memory contiguous in physical address space. Some platforms may allocate memory that is local to an I/O adapter (aka a “bus bridge”), and such memory may be severely limited in size.

The `isc` parameter is a pointer to the ISC table entry assigned to the driver’s interface card. It is the handle for the driver instance.

The `size` parameter is the size in bytes of memory to be allocated.
The `iova` parameter is a pointer to the returned I/O virtual address. It is the base address of the allocated memory from the view of the interface card.

The `vaddr` parameter is a pointer to the returned virtual address corresponding to the I/O virtual address. It is the base address of the allocated memory from the view of the processor.

The `type` parameter is a bit mask of the requested memory attributes. Valid memory attributes are the following:

- **WSIO_SHMEM_OPTIMIZE_DEVICE_LATENCY** - allocation should optimize for device access latency. If the platform allows, allocated memory should be local to the I/O adapter connecting the interface card. This is the default attribute if none are specified.

- **WSIO_SHMEM_OPTIMIZE_HOST_LATENCY** - allocation should optimize for host access latency. If the platform allows, allocated memory should be in host memory.

- **WSIO_SHMEM_INBOUND** - hint indicating the allocated memory will be used exclusively for inbound (device to memory) DMA only

- **WSIO_SHMEM_OUTBOUND** - hint indicating the allocated memory will be used exclusively for outbound (memory to device) DMA only

- **WSIO_SHMEM_DEV_WEAK_OK** - hint indicating accesses to the allocated memory can be weakly ordered.

- **WSIO_SHMEM_ALIGN_ON_SIZE** - allocation must align the memory on the size specified. If, for example, size is a power of 2, the base address of the memory allocated must be aligned to the same power of 2 or a multiple of that value.

**RETURN VALUES**

`wsio_allocate_shared_memory()` returns the following values:

- **SHMEM_OK**  
  Successful completion

- **SHMEM_NO_RESOURCES**  
  Memory not allocated

**CONSTRAINTS**
EXAMPLES

caddr_t my_iova;
caddr_t my_vaddr;

/*
 * Allocate contiguous memory that is page bytes in size
 * and aligned to a page size boundary. If the platform
 * allows, allocate memory that is local to the I/O
 * adapter (bus bridge) connecting the interface card.
 */
if (wsio_allocate_shared_memory(
   isc, NBPG, &my_iova, &my_vaddr,
   WSIO_SHMEM_OPTIMIZE_DEVICE_LATENCY |
   WSIO_SHMEM_ALIGN_ON_SIZE) != SHMEM_OK) {

  /*
   * Failed to allocate memory for continuous DMA.
   */
}

SEE ALSO

wsio_flush_shared_memory(WSIO4),
wsio_free_shared_memory(WSIO4)
NAME

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

SYNOPSIS

DESCRIPTION

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

STRUCTURE MEMBERS

Field  Purpose

drv_path  Follow these guidelines:

• For device drivers, drv_path is typically a string that contain the interface card's type and the device's class. For example, scsi_disk.

• For interface drivers, drv_path should match the card's type. For example, scsi.

• For pseudo drivers, drv_path should match the card's class. For example, graphics.

drv_type  One of the following values:

T_INTERFACE  The driver controls an interface card.

T_DEVICE  The driver controls a hardware device.

drv_flags  One of the following values:

DRV_CONVERGED  The driver meets the HP-UX Release 10.0 Converged I/O specifications. All new drivers should meet these specifications.

NOT_CONVERGED  The driver conforms to the pre-Release 10.0 unconverged specifications.
**WSIO Reference Pages**

**Functions, Macros and Structures**

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

**RETURN VALUES**

**CONSTRAINTS**

**SEE ALSO**
NAME

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

SYNOPSIS

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

typedef struct wsio_drv_info {drv_info_t *drv_info;
    drv_ops_t *drv_ops; wsio_drv_data_t *drv_data;}
    wsio_drv_info_t;
```

DESCRIPTION

The wsio_drv_info_t WSIO structure type, defined in <sys/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.

RETURN VALUES

CONSTRAINTS

SEE ALSO

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

wsio_fastmap(WSIO) – Map all or part of a host address range into an I/O virtual address range

SYNOPSIS

#include <sys/dma.h>

int wsio_fastmap (struct isc_table_type *isc, int range_type,
               struct iovec *host_range,
               struct iovec *io_range);

PARAMETERS

isc Pointer to an ISC structure.
range_type The type of host address for host_range. If range_type > 0 then this is the space ID of the address range. Other values are:
            KERNELSPACE Kernel virtual buffer (same as space ID = 0)
host_range A pointer/length pair indicating the host address range of type range_type. The length will be modified by the service to indicate the bytes remaining to be mapped. This length is also returned by wsio_fastmap().
io_range A pointer/length pair filled by wsio_fastmap() with the I/O virtual address range or EISA address range.

DESCRIPTION

The wsio_fastmap() WSIO function provides the same functionality as wsio_map() when the entire host address range resides on a single physical page. This condition is not checked by wsio_fastmap(). Cache-line fragments are ignored (as in wsio_map() with the IO_IGN_ALIGNMENT hint).

Since the behavior and side effects are different for coherent and noncoherent systems, these are discussed separately below. Although the behaviors are slightly different, drivers should not have dependencies on
these differences. In all cases, the programming model is the same. All mappings remain valid until the \texttt{io\_range} is unmapped by calls to \texttt{wsio\_unmap()}. If the \texttt{io\_range} is reused, \texttt{dma\_sync()} must be used to resynchronize it.

**Behavior on Noncoherent Systems**

On noncoherent systems, the I/O virtual address is equivalent to the host physical address. \texttt{wsio\_map()} will return an \texttt{io\_range} at each page break in the \texttt{host\_range}. If the mapping is for an EISA module, then the \texttt{io\_range} is filled with EISA addresses and the EISA map will contain the host physical address.

**Behavior on Coherent Systems**

On coherent systems, the I/O virtual address is obtained by creating a mapping in the I/O PDIR. For cache-line fragments, the SAFE bit will be set in the I/O PDIR, unless inhibited by \texttt{IO\_IGN\_ALIGNMENT}. However, it is expected that buflets will be used by the caller to handle them. If the mapping is for an EISA module, then the \texttt{io\_range} is filled with EISA addresses and the EISA map will contain the corresponding IOVAs.

**RETURN VALUES**

\texttt{wsio\_fastmap()} always returns 0 on a Noncoherent system.

\texttt{wsio\_fastmap()} returns the following values on a Coherent system:

- 0 \hspace{1cm} The range was fully mapped.
- -1 \hspace{1cm} The necessary resources could not be obtained.

**CONSTRAINTS**

**WARNING**

The \texttt{IO\_NO\_SEQ} flag is NOT set with \texttt{wsio\_fastmap()}, and if the mapping is cache line aligned the \texttt{IO\_SAFE} bit is NOT set. Under certain conditions this call MUST NOT be used for PCI. See pci-errata(PCI5) for details.

**EXAMPLE**

The following function maps a single page of memory (\texttt{virt\_page} is aligned on a page boundary):
caddr_t my_page_map(port_num, caddr_t virt_page)
    port_num_type port_num;
    caddr_t virt_page;
    {
        struct iovec host, io;
        host->iov_base = virt_page; /* virt_page is page-aligned */
        host->iov_len = NBPG;
        wsio_fastmap(port_num,KERNELSPACE,&host,&io);
        return io->iov_base;
    }

WARNINGS

It is up to the caller to ensure that the space to be mapped does not cross a page boundary. On a Coherent system this will be indicated by a return of -1, on a Noncoherent system the call will succeed.

SEE ALSO

dma_sync(CDIO3), init_map_context(CDIO3),
isc_table_type(KER4), pci_errata(PCI5), wsio_fastmap(WSIO3),
wsio_remap(WSIO3), wsio_set_attributes(WSIO3),
wsio_unmap(WSIO3)
NAME

wsio_flush_shared_memory(WSIO) - Flush the memory previously allocated and mapped by wsio_allocate_shared_memory().

SYNOPSIS

#include <sys/wsio.h>

shmem_status_t wsio_flush_shared_memory (  
    struct isc_table_type* isc, size_t size,  
    caddr_t iova, caddr_t vaddr,  
    wsio_shmem_attr_t type);

PARAMETERS

isc Pointer to an ISC table entry.
size Size in bytes or memory to be flushed.
iova I/O virtual address.
vaddr Virtual address corresponding to iova.
type Bit mask of the memory attributes.

DESCRIPTION

The wsio_flush_shared_memory() function flushes the memory previously allocated and mapped by wsio_allocate_shared_memory(). This ensures that data in the allocated memory is viewed consistently by the device and processors. All parameters passed to wsio_flush_shared_memory() must match the parameters passed to the corresponding call to wsio_allocate_shared_memory().

RETURN VALUES

wsio_flush_shared_shared_memory() returns the following values:

SHMEM_OK Successful completion
SHMEM_NO_RESOURCES Memory not flushed
CONSTRAINTS

SEE ALSO

wsio_allocate_shared_memory(WSIO4),
wsio_free_shared_memory(WSIO4)
NAME

wsio_free_shared_memory(WSIO4) - Release and unmap contiguous memory previously allocated and mapped by wsio_allocate_shared_memory().

SYNOPSIS

```c
#include <sys/wsio.h>
void wsio_free_shared_memory (struct isc_table_type *isc,
                           size_t size, caddr_t iova, caddr_t vaddr,
                           wsio_shmem_attr_t type);
```

PARAMETERS

- **isc** Pointer to an ISC table entry.
- **size** Size in bytes or memory to be released.
- **iova** I/O virtual address.
- **vaddr** Virtual address corresponding to `iova`.
- **type** Bit mask of the memory attributes.

DESCRIPTION

The `wsio_free_shared_memory()` function releases and unmaps contiguous memory previously allocated and mapped by `wsio_allocate_shared_memory()`. All parameters passed to `wsio_free_shared_memory()` must match the parameters passed to the corresponding call to `wsio_allocate_shared_memory()`.

RETURN VALUES

None

CONSTRAINTS
SEE ALSO

wsio_alocate_shared_memory(WSIO4),
wsio_flush_shared_memory(WSIO4)
NAME

wsio_get_interrupts(WSIO3) - Determine which interrupt has been assigned to a card

SYNOPSIS

#include <sys/io.h>

int wsio_get_interrupts (struct isc_table_type *isc);

PARAMETERS

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

DESCRIPTION

The \texttt{wsio_get_interrupts()} WSIO function determines the IRQ of an interface card hardware module associated with the given ISC structure. It expects the "interrupt" property of the iotree node to have been appropriately assigned during I/O configuration. It can be used by drivers for getting the information needed to set up their \texttt{isc->eim} values and for setting up their \texttt{isrlink()} service calls. PCI bus drivers must use \texttt{-1} for their \texttt{isrlink()} calls, allowing the PCI services to determine the actual interrupt line assigned.

RETURN VALUES

\texttt{wsio_get_interrupts()} returns the following values:

\begin{itemize}
  \item \texttt{>0} The IRQ for the card.
  \item \texttt{-1} \texttt{isc} is NULL or there was a problem retrieving the node's interrupt property.
\end{itemize}

CONSTRAINTS

EXAMPLES

mydriver_init(isc)
struct isc_table_type *isc;
{
    int wsio_eim;
    ...

    if (NOT_PCI) {
        mydriver_reset(isc);
        isc->eim = wsio_eim = wsio_get_interrupts(isc);
        isc->eim_control = 0;
    } else {
        mydriver_reset(isc);
        wsio_eim = -1;
    }
    isrlink(isc, mydriver_isr, wsio_eim, isc, isc->if_drv_data);
    ...
}

SEE ALSO

isc_table_type(KER4)
NAME

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

SYNOPSIS

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

1  Successful completion. The ISC pointer found is returned in isc_ptr.

0  Failure. The ISC could not be found.

CONSTRAINTS

SEE ALSO
NAME

wsio_get_pva(WSIO3) - Translate an I/O virtual address to its Processor virtual address

SYNOPSIS

#include <sys/dma.h>
caddr_t wsio_get_pva (struct isc_table_type *isc, caddr_t iova);

PARAMETERS

isc 
Pointer to an ISC structure.

iova 
An I/O Virtual Address.

DESCRIPTION

The wsio_get_pva() WSIO function translates I/O virtual addresses into processor virtual addresses. The translation is more efficient if the memory object is equivalently mapped (ProcVA == Physical address). This function is restricted to virtual buffers in kernel space.

RETURN VALUES

<>NULL 
The processor virtual address, if the I/O virtual address exists and there is a corresponding processor virtual address in KERNELSPACE.

NULL 
Otherwise.

CONSTRAINTS

EXAMPLES

#include <sys/dma.h>
#define HOST_RAM_SIZE 0x8000
extern int coherent_io_enabled;
my_attach(id,isc)
PCI_ID id;
struct isc_table_type *isc;
{
    caddr_t host_ram;
    int i, pages_mapped;
    caddr_t io_tmp, proc_tmp;
    ...
    MALLOC(hostram,cadr_t, HOST_RAM_SIZE,M_DYNAMIC,M_NOWAIT);
    bzero(hostram,HOST_RAM_SIZE);
    pages_mapped = HOST_RAM_SIZE/NBPG;
    if (HOST_RAM_SIZE % NBPG)
        pages_mapped++;
    if (!coherent_io_enabled) {
        /*
        ** need to ensure a contiguous
        ** buffer on processors that do
        ** not have an IO TLB, because
        ** our card expects contiguous
        ** space for task lists
        */
        proc_tmp = host ram;
        io_tmp = wsio_get_pva(isc,proc_tmp);
        for (i=0;i < pages_mapped;
            i++,io_tmp += NBPG,proc_tmp += NBPG) {
            if (io_tmp != wsio_get_pva(isc,proc_tmp)) {
                msg_printf("my_attach IO buffer not contiguous\n");
                FREE(hostram,M_DYNAMIC);
                return(*my_pci_saved_attach)(id,isc)
            }
        }
    }
    ...
    return(*my_pci_saved_attach)(id,isc)
}

SEE ALSO
wsio_fastmap(WSIO3), wsio_map(WSIO3), wsio_remap(WSIO3),
wsio_unmap(WSIO3)
NAME

wsio_get_registers (WSIO3) – Get the register addresses of an interface card

SYNOPSIS

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

caddr_t * wsio_get_registers (struct isc_table_type * isc);
```

PARAMETERS

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

DESCRIPTION

The wsio_get_registers() WSIO function retrieves the register addresses of the interface card associated with the given ISC entry.

Any module that has additional SPA space (for example, graphics) will need access to both of the register-property addresses. Normally, only the first register is retrieved and held in the ISC structure. This routine will allow drivers to get both register pointers if they need them.

wsio_get_registers() is not supported for PCI interface cards.

RETURN VALUES

<>-1 Successful completion. The value is a pointer to an array of register sets for this module (for most modules, this will be an array of 1 or 2 elements, corresponding to HPA and SPA).

-1 Failure. isc is NULL or there was a problem retrieving the node's registers property.

CONSTRAINTS
SEE ALSO
NAME

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

SYNOPSIS

```c
int wsio_install_driver (wsio_drv_info_t * 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

`wsio_install_driver()` returns the following 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

```c
/* 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_isc_to_instance(WSIO3) – Retrieve an instance number of an iotree node

SYNOPSIS

#include <sys/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.

• 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

wsio_isc_to_instance() returns the following values:

>=0  Successful completion. The value is the matching instance number.

-1   Error.

CONSTRAINTS

SEE ALSO
NAME

wsio_map(WSIO3) - Map all or part of a host address range into an I/O virtual address range

SYNOPSIS

#include <sys/dma.h>

int wsio_map (struct isc_table_type *isc, io_map_t *map_cb, int hints, int range_type, struct iovec *host_range, struct iovec *io_range);

PARAMETERS

isc
A pointer to the isc struct for this device which contains information that is bus specific used by the underlying mapping services to correctly set up the mapping hardware and other bus specific details.

map_cb
A control structure which is private to the mapping service. This structure stores the mapping context across multiple calls to wsio_map(). map_cb can be a local variable. If used, it must be initialized before the first call to wsio_map() via a call to init_map_context(). A non-NULL value causes the default IOVA allocation scheme to be used regardless of previous calls to wsio_set_attributes().

hints
Hints which change the behavior of wsio_map():

IO_CONTIGUOUS Indicates that wsio_map() must allocate a single contiguous I/O virtual range. If wsio_map() is unable to do this, it will return -1. Of course, on noncoherent systems, the object must by physically contiguous. This hint implies IO_IGN_ALIGNMENT.

IO_IGN_ALIGNMENT Indicates that wsio_map() should not set the SAFE bit for cache-line.
fragments and that `wsio_map()` should not return separate `io_ranges` for cache-line fragments.

**IO_LOCK**

Forces the LOCK bit to be set in the I/O PDIR for this mapping. This hint should be used for devices which can request exclusive access to memory. For instance, an EISA card can assert the EISA LOCK signal to request exclusive access to memory. Memory objects used in this way must be mapped with the `IO_LOCK` hint specified.

**IO_NO_SEQ**

Turns off the SEQUENTIAL bit in the I/O PDIR for this mapping. This inhibits prefetching of data for this object by the I/O Adapter. This hint has no effect for modules which do not prefetch data.

**IO_SAFE**

Forces the SAFE bit to be set in the I/O PDIR for this mapping. This causes the I/O subsystem to perform read-modify-write bus transactions for this mapping. This hint should be specified if sub-cacheline sized DMA will be used for the buffer.

**IO_SEMA**

Provides a hint that this memory object will be used as a semaphore.

**IO_UPDATE**

Forces the UPDATE bit to be set in the I/O PDIR for this mapping.

**range_type**

The type of host address for `host_range`. If `range_type > 0`, then this is the space ID of the address range. Other values are:

- KERNELSPACE Kernel virtual buffer (same as space ID = 0)

**host_range**

A pointer/length pair indicating the host address range of type `range_type`. The length will be modified by the service to indicate bytes remaining to be mapped. This
 Functions, Macros and Structures

length will also be returned by wsio_map().

io_range A pointer/length pair filled by wsio_map() with the I/O virtual address range or the EISA address range.

DESCRIPTION

The wsio_map() WSIO function maps the host address range into an I/O virtual address range. For EISA busses, the range is also mapped into EISA space. It may take multiple calls to wsio_map() to map the entire host range due to cache-line and page alignment restrictions. If the host_range is not aligned on a cache-line, then the first io_range will also not be cache-line aligned and will contain only the bytes in the same cache-line. Likewise, if the host_range does not end on a cache-line, then the last io_range will represent a cache-line fragment (but will be cache-line aligned). Buflets for the cache-line fragments must be managed by the caller.

Since the behavior and side effects are different for coherent and noncoherent systems, these are discussed separately below. Although the behaviors are slightly different, drivers should not have dependencies on these differences. In all cases, the programming model is the same. For each mapping:

• Call init_map_context() for the map_cb if used.
• Repeatedly call wsio_map() until the return value is less than or equal to 0.

All mappings remain valid until the io_range is unmapped via calls to wsio_unmap(). If the io_range is reused, dma_sync() must be used to resynchronize it.

Behavior on Noncoherent Systems

On noncoherent systems, the I/O virtual address is equivalent to the host physical address. wsio_map() will return an io_range at each page break in the host_range. If the mapping is for an EISA module, then the io_range is filled with EISA addresses and the EISA map will contain the host physical address.

Behavior on Coherent Systems

On coherent systems, the I/O virtual address is obtained by creating a mapping in the I/O PDIR. For cache-line fragments, the SAFE bit will be set in the I/O PDIR unless inhibited by IO_IGN_ALIGNMENT. However, it is expected that buflets will be used by the caller to handle them. If the mapping is for an EISA module, then the io_range is filled with EISA
addresses and the EISA map will contain the corresponding IOVAs.

**RETURN VALUES**

`wsio_map()` returns one of the following values:

- **>0** The number of bytes remaining in the `host_range`.
- **0** The range was fully mapped.
- **-1** The necessary resources could not be obtained.

**CONSTRAINTS**

**WARNING**

Under certain conditions a PCI master MUST have the `IO_SAFE` and `IO_NO_SEQ` flag bits set to ensure coherency. See `pci-errata(PCI5)`.

**EXAMPLES**

The following function maps a set of host pointer/length pairs given by `host_vec` (WSIO driver):

```c
int
my_driver_output(struct isc_table_type *isc, int vec_cnt, struct iovec *host_vec)
{
  io_map_t context;
  struct iovec *io_vec;
  struct my_dma_type *dma_desc;
  int resid;

  init_map_context(&context);

  /* Allocate a DMA structure for my DMA model */
  MALLOC( dma_desc, sizeof(struct my_dma_type) );

  /* Synchronize DMA buffer (outbound DMA) */
  dma_sync_IO(KERNELSPACE, host_vec->iov_base, host_vec->iov_len, IO_SYNC_FORDEV);

  /* Point to the beginning of the DMA Vector area */
```
io_vec = &dma_desc->iov;

/* Do mapping for each host vector */
for (; vec_cnt; host_vec++)
    do {
        resid = wsio_map(my_isc, context, 0, KERNELSPACE,
                         host_vec, io_vec);
        if (resid < 0)
            /* Handle Error condition */
    }
    /* Point to next DMA vector */
    io_vec++;
} while (resid > 0);

dma_desc->iov_cnt = io_vec - &dma_desc->iov;

/* Kick off the DMA */
return my_start_output(isc, dma_desc);

The following example attempts to allocate 32K bytes of contiguous memory for a PCI SCSI bus master’s task lists. Because there is no API for contiguous memory on a Noncoherent system the routine checks for this. This scheme has the highest chance of success if it is done during PCI attach time because memory is generally not fragmented at that time. For 10.20 coherent_io_enabled indicates whether this is a Coherent system or a Noncoherent system.

#define HOST_RAM_SIZE 0x8000
#define TMP_BUF_SIZE 0x200

extern int coherent_io_enabled;

sample_pci_attach(id, isc)
PCI_ID id;
struct isc_table_type *isc;
{
    caddr_t tmp_buf, host_ram;
    caddr_t tmp_buf_phys, host_ram_phys;
    struct iovec host_vec, io_vec;
    struct iovec *io_vec_ptr *workptr;
    io_map_t map_cb;

    /* code to check that it’s our card */
/* set up memory */
MALLOC(tmp_buf, caddr_t, TMP_BUF_SIZE, M_DYNAMIC, M_NOWAIT);
bzero(tmp_buf, TMP_BUF_SIZE);
MALLOC(host_ram, caddr_t, HOST_RAM_SIZE, M_DYNAMIC, M_NOWAIT);
bzero(host_ram, HOST_RAM_SIZE);

/*
** WSIO mapping services do different things
** on coherent IO systems (C-Class) and
** non-coherent IO systems (B-Class).
**
** In both cases the mapping call manipulates
** the host iovec base and length fields so
** that wsio_map can be called repeatedly.
**
** ------------------------------------------
** coherent IO systems behave as follows:
** A single call attempts to map all
** pages in host.iov_len. The IO_CONTIGUOUS
** flag forces a call failure if the mapping
** crosses a ‘range’ boundary (currently 32K)
** AND unmaps all of the pages. If the
** IO_CONTIGUOUS flag is not set, the call
** will return with 0, but with host.iov_len > 0,
** indicating that you need to call wsio_map
** again, n.b., the next mapping may NOT be
** contiguous from the point of view of PCI.
**
** ------------------------------------------
** non-coherent IO systems behave as follows:
** The IO_CONTIGUOUS flag is ignored, and
** at most, each call will map a single
** page (4K), there is no guarantee that
** malloc will have given you a contiguous
** buffer. Basically, you need to check
** each page and call wsio_map for each page.
**
** The unmap call is a no-op on B-Class
**
** n.b., don’t depend upon the ‘range’ boundary
** for future releases remaining at 32K. I went
** ahead and commented this because we have seen a
** failure mode due to a mapping order of:
** tmpbuf = 200 bytes = a page, and then
** hostram = 0x7a4 bytes = 8 pages
** i.e., a total of 9 pages crossing a  
** range boundary -> the map call failed.  
** */

```c
int pages_mapped = HOST_RAM_SIZE / NBPG;
int i, map_ret;

if (HOST_RAM_SIZE % NBPG)  
    pages_mapped ++;
/* see if we have a physically contiguous buffer  
* on B-Class  
*/
if (!coherent_io_enabled) {
    caddr_t phys_tmp,virt_tmp;

    virt_tmp = host_ram;
    phys_tmp = wsio_get_pva(isc,virt_tmp);
    for (i=0;i<pages_mapped;i++,phys_tmp += NBPG,virt_tmp += NBP  
G) {
        if (phys_tmp != wsio_get_pva(isc,virt_tmp)) {  
            msg_printf("sample attach B-Class buffer not contiguous\n"  
        );
        FREE(tmp_buf, M_DYNAMIC);
        FREE(host_ram, M_DYNAMIC);
        return (*sample_pci_saved_attach)(id, isc);
    }
}
/*  
** now do the mappings -  
** do hostram first because  
** it fills a C-Class ‘range’  
*/
MALLOC(io_vec_ptr, struct iovec *,  
    sizeof(struct iovec) * pages_mapped), M_DYNAMIC, M_NOWAIT);
if (io_vec_ptr == NULL) {  
    FREE(tmp_buf, M_DYNAMIC);
    FREE(host_ram, M_DYNAMIC);
    return (*sample_pci_saved_attach)(idc, isc);
}
workptr = io_vec_ptr;
host_vec.iov_base = host_ram;
host_vec.iov_len = HOST_RAM_SIZE;
init_map_context(&map_cb);
for (i=0;i<pages_mapped;i++,workptr++) {
```
map_ret = wsio_map(isc, &map_cb,
   IO_CONTIGUOUS | IO_SAFE | IO_LOCK,
   KERNELSPACE, &host_vec, workptr);
if (map_ret < 0) {
   msg_printf("sample attach: could not map hostram pointer\n"");
   FREE(tmp_buf, M_DYNAMIC);
   FREE(host_ram, M_DYNAMIC);
   FREE(iov_ec_ptr, M_DYNAMIC);
   return (*sample_pci_savedAttach)(id, isc);
}
if (i==0)
   host_ram_phys = io_vec_ptr->iov_base; /* base addr for PCI */
*/
if (map_ret == 0)
   break;
} /* now map the tmp_buf **
** Note that we map it after hostram **
** because 10.20 uses a 32 K range and **
** if we did it before hostram the **
** contiguous call would fail on a C class **
** because we would have used up TMP_BUF_SIZE **
** of the 32 K range.
*/
init_map_context(&map_cb);
host_vec.iov_base = tmp_buf;
host_vec.iov_len = TMP_BUF_SIZE;
   if ( wsio_map ( isc , &map_cb ,
      IO_CONTIGUOUS | IO_SAFE | IO_LOCK,
      KERNELSPACE , &host_vec , &io_vec ) ) {
      printf("sample attach Could not map() tmp_buf pointer.\n"");
      io_vec.iov_base = host_ram_phy_addr;
      io_vec.iov_len = HOST_RAM_SIZE;
      wsio_unmap(isc, io_vec);
      FREE(tmp_buf, M_DYNAMIC);
      FREE(host_ram, M_DYNAMIC);
      FREE(iov_ec_ptr, M_DYNAMIC);
      return (*sample_pci_savedAttach)(id, isc);
   } /* set up rest of stuff e.g., isrlink
isc_claim( isc , &wsio_sample_drv_info );
return (*sample_pci_savedAttach)(id, isc);
SEE ALSO

dma_sync(CDIO3), init_map_context(CDIO3), pci_errata(PCI5),
wsio_fastmap(WSIO3), wsio_remap(WSIO3),
wsio_set_attributes(WSIO3), wsio_unmap(WSIO3)
NAME

wsio_probe_dev_info(WSIO4) - WSIO device probe information

SYNOPSIS

#include <sys/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 consists 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_register_addr_probe(WSIO3) - Register a driver probe function.

SYNOPSIS

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.
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(WSIO4),
wsio_register_dev_probe(WSIO_DRV),
NAME

\texttt{wsio\_register\_dev\_probe} (WSIO3) - Register a driver probe function

SYNOPSIS

\begin{verbatim}
int wsio_register_dev_probe (int type, int (*func)(), char *drv_name);
\end{verbatim}

PARAMETERS

- \textit{type}\hspace{1cm} Indicates what driver data the third parameter should match to. Valid values are:
  - \texttt{IF\_CLASS}\hspace{1cm} The third argument \texttt{drv\_name} is to be matched with the \texttt{drv\_path} field of the \texttt{wsio\_drv\_data\_t} structure.
  - \texttt{DRV\_NAME}\hspace{1cm} The third argument, \texttt{drv\_name} is to be matched with the name field of the \texttt{drv\_info\_t} structure.

- \textit{func}\hspace{1cm} A pointer to the driver probe function.

- \textit{drv\_name}\hspace{1cm} An ASCII string indicating the name or class of the driver.

DESCRIPTION

The WSIO service \texttt{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, \texttt{drv\_name}, is an ASCII string that is used to match the probe function to specific driver/interfaces cards. The first parameter, \texttt{type}, is used to indicate what driver information the ASCII string is to be matched to. If the parameter has the value \texttt{IF\_CLASS}, it indicates the string should be matched to the \texttt{drv\_path} field of the driver's \texttt{wsio\_drv\_data\_t} structure. If the \texttt{type} parameter is set to the value of \texttt{DRV\_NAME}, the third argument is matched with the name field of the driver's \texttt{drv\_info\_t} structure.

A value of \texttt{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 \texttt{IF\_CLASS} is more general since several drivers may have the same
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 `driv_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 structure of the driver that controls the device. desc is an ASCII string describing the device.

RETURN VALUES

wsio_register_dev_probe() returns the following values:

0 Successful completion.
-1 Error.

EXAMPLE

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(WSIO4),
wsio_register_addr_probe(WSIO_DRV),
NAME

`wsio_register_probe_func` (WSIO3) - Insert a driver-specified probe function into the global probe list.

SYNOPSIS

```c
int wsio_register_probe_func (int (* func)(), char * if_class);
```

PARAMETERS

- `func` - The name of your probe function.
- `if_class` - The string that will be matched with the `drv_path` field of driver `wsio_drv_data` structs registered during `driver_install()`.

DESCRIPTION

The `wsio_register_probe_func()` WSIO function inserts a driver-specified probe function into the global probe list. The underlying probe routines match your `if_class` string with driver `drv_path` strings to determine which probe to use. These routines match, character by character, up to a space or underline character, at which point the match succeeds. Note that the the maximum string length is limited to 16 characters. For example, if `scsi` was passed in as the `if_class` parameter when registering your probe function, it would successfully match `scsi_ctl` in the `drv_path` field of the `scsi_ctl` driver's `wsio_drv_data` struct.

RETURN VALUES

`wsio_register_probe_func()` returns an indeterminate value. Treat it as a `void` function.

CONSTRAINTS

SEE ALSO

`driver_class_probe (WSIO_DRV)`, `driver_addr_probe (WSIO_DRV)`,
wsio_drv_data(WSIO4)
NAME

_WSIO_REMAP (WSIO3) - Map a host range into a pre-mapped I/O Virtual Address range

SYNOPSIS

#include <sys/dma.h>

int wsio_remap (struct isc_table_type *isc, int range_type,
                struct iovec *host_range, struct iovec *io_range);

PARAMETERS

isc             An isc_table_type structure used to get interface and CDIO information.
range_type      The type of host address for host_range. If range_type > 0 then this is the space ID of the address range. Other values are:
                KERNELSPACE Kernel virtual buffer (same as space ID = 0)
host_range      A pointer/length pair, indicating the host address range of type range_type. The length will be modified by the service to indicate bytes remaining to be mapped.
io_range        A pointer/length pair, which was filled by a previous call to wsio_map(), wsio_fastmap(), or wsio_remap(). wsio_remap() will map the new host_range into this range.

DESCRIPTION

The wsio_remap() WSIO function is normally used by CDIOs such as EISA which have to manage CDIO specific map registers. It is generally not used by driver writers.

The wsio_remap() WSIO function maps a pre-allocated I/O virtual address to new host_ranges. 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 equal sizes.
Cache-line fragments are ignored (IO_IGN_ALIGNMENT is assumed).

Since the behavior and side effects are different for coherent and noncoherent systems, these are discussed separately below. Although the behaviors are slightly different, drivers should not have dependencies on these differences.

All mappings remain valid until the io_range is unmapped via calls to wsio_unmap(). If the io_range is reused, dma_sync_IO() must be used to resynchronize it.

**Behavior on Noncoherent Systems**

On noncoherent systems, the I/O virtual address is equivalent to the host physical address. This will likely not be the same as the I/O virtual address provided by the caller. Therefore, wsio_remap() will fill io_range with the new address range. If the mapping is for an EISA module, then the io_range is filled with EISA addresses and the EISA map will contain the host physical address.

**Behavior on Coherent Systems**

On coherent systems, the I/O virtual address is remapped to point to the new host_range. The page type bits are not modified.

**RETURN VALUES**

- 0 The range was fully mapped.
- -1 The necessary resources could not be obtained.

**CONSTRAINTS**

**SEE ALSO**

dma_sync(CDIO3), wsio_fastmap(WSIO3), wsio_map(WSIO3), wsio_remap(WSIO3), wsio_unmap(WSIO3)
NAME

wsio_set_attributes(WSIO3) – Set map function attributes

SYNOPSIS

#include <sys/dma.h>

void wsio_set_attributes (struct isc_table_type *isc, int attributes);

PARAMETERS

isc
Attributes which change the default behavior of
wsio_map(), wsio_fastmap(), wsio_remap(), and
wsio_unmap(). More than one attribute can be set by
ORing them together. The following attributes are
defined:

IO_DEBUG_DMA Turns on additional checks in
mapping services. This should be
used for debugging only. The services
will call panic() if any problems are
detected.

IO_INTERLEAVED_DMA The device is likely to
interleave many I/O requests. Mass
storage devices are an example of this
type of device. This is the default
behavior.

IO_NONINTERLEAVED_DMA The device is likely to
satisfy a single I/O request at a time.
This is typical of networking devices.
This attribute cannot be specified with IO_INTERLEAVED_DMA.

Even if this attribute is set, the
services will behave as if they are
IO_INTERLEAVED_DMA if the map cb
argument is non-NULL in calls to
wsio_map(). Networking cards are
typical of noninterleaved devices.
DESCRIPTION

The `wsio_set_attributes()` WSIO function alters the default behavior of `wsio_fastmap()`, `wsio_map()`, `wsio_remap()`, and `wsio_unmap()`.

RETURN VALUES

`wsio_set_attributes()` is a `void` function.

CONSTRAINTS

SEE ALSO

panic(KER2), `wsio_fastmap(WSIO3)`, `wsio_map(WSIO3)`, `wsio_remap(WSIO3)`, `wsio_unmap(WSIO3)`
NAME

wsio_uninstall_driver(WSIO3) - Uninstall a driver's header structure from the WSIO CDIO.

SYNOPSIS

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

wsio_uninstall_driver() returns the following values

0 Successful completion.
<>0 Error.

CONSTRAINTS

SEE ALSO

wsio_install_driver(WSIO3)
NAME

wsio_unmap (WSIO) – Function to unmap an I/O virtual address range

SYNOPSIS

#include <sys/dma.h>

void wsio_unmap (struct isc_table_type *isc, struct iovec *io_range);

PARAMETERS

isc       A pointer to the isc struct holding bus-specific
           information used by the mapping services.

io_range  A pointer/length pair, representing the I/O virtual
           range to be unmapped.

DESCRIPTION

On coherent systems, the resources associated with the mapping are
released. On noncoherent systems, this function does nothing. In
addition to the wsio_unmap() WSIO function, the caller must call
dma_sync() during post-DMA cleanup for inbound data.

When multiple objects (I/Os) are mapped with a single map context,
wsio_unmap() must not be called for ANY of the mapped objects until
ALL the I/Os for that context have completed. Failure to ensure that all
I/Os have completed may result in data corruption.

RETURN VALUES

wsio_unmap() does not return a value.

CONSTRAINTS

EXAMPLE

The following function cleans up after an inbound DMA:

#define SYNC() dma_sync(0,0,0,0)
void my_inbound_dma_cleanup(isc, vec_cnt, host_vec, dma_desc)
    struct isc_table_type *isc;
    int vec_cnt;
    struct iovec *host_vec;
    struct my_dma_type *dma_desc;
{
    struct iovec *io_vec;
    int dma_cnt;

    /* Point to the beginning of the DMA Vector area */
    io_vec = &dma_desc->iov;

    /* Unmap each DMA vector */
    for (dma_cnt=dma_desc->iov_cnt; dma_cnt>0; dma_cnt, 
        io_vec++) wsio_unmap(my_isc,io_vec);

    /* Inbound data so synchronize each host range */
    for (; vec_cnt; host_vec++)
        dma_sync(KERNELSPACE,host_vec->iov_base,
                host_vec->iov_len,
                IO_SYNC_FORCPU|IO_NO_SYNC|IO_PREFETCHED);

    SYNC();
    FREE( dma_desc );
}

SEE ALSO

dma_sync(CDIO3), init_map_context(CDIO3),
wsio_fastmap(WSIO3), wsio_map(WSIO3), wsio_remap(WSIO3)
NAME

wsio_unregister_dev_probe(WSIO3) - Unregisters a driver probe function.

SYNOPSIS

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_dev_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_dev_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

wsio_unregister_dev_probe() returns the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Successfully found and deleted the driver</td>
</tr>
<tr>
<td>-1</td>
<td>Not found</td>
</tr>
</tbody>
</table>

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(WSIO4), wsio_drv_info(CDI04),
wsio_register_dev_probe(WSIO_DRV)