<u>Cyclone Microsystems Linux Host to Linux</u>
<u>Programmable Network Adapter Messaging Driver</u>

The Cyclone Microsystems Messaging Driver allows DMA-driven data movement between an x86-based host processor running Linux version 2.6 and a Cyclone Microsystems XScale-based Intelligent I/O Processor (IOP) running Linux version 2.6. The Cyclone Messaging Driver follows a split driver model. In a split driver model, the driver has two components: a host driver, running on the host processor, and an IOP driver, running on the XScale processor. The driver makes use of the XScale Messaging Unit (MU) which is contained within the Address Translation Unit (ATU) of the XScale processor. The MU provides a mechanism for data to be transferred between the host processor and the XScale core and notifies the respective processor of the arrival of new data through an interrupt. The MU can be used to both send a receive messages. The Cyclone Messaging Driver makes use of Messaging Registers and the Circular Queues within the MU. The Messaging Registers are used to exchange information during driver initialization. The Circular Queues are used to exchange messages (via message pointers) during driver operation. There are four queues in the MU. Two of the queues (Inbound Free and Inbound Post) are used to send messages from the host to the XScale core. The other two queues (Outbound Free and Outbound Post) are used to send messages from the XScale core to the host.

<u>IOP Driver</u>

The IOP-based portion of the driver is contained within the **files *arch/arm/mach-iop3xx/cyc_msg.c*** and ***arch/arm/mach-iop3xx/cyc_msg.h*.** At its most basic level, the driver responds to request messages from the host processor, sent via the MU queues. The two main requests are HOST_TO_CYC_DMA_READ and HOST_TO_CYC_DMA_WRITE. These messages are supported using two queues which are maintained by the driver: the read buffer queue and the write buffer queue. The driver is interrupt driven and responds to inbound message post interrupts.

Write buffers are empty buffers which are posted to the write buffer queue and are used by the driver when the host issues a HOST_TO_CYC_DMA_WRITE command. When the driver receives a HOST_TO_CYC_DMA_WRITE message from the host, it pulls an empty buffer from the write buffer queue and kicks off a DMA Read operation to pull data from the host's buffer into the empty buffer. Upon completion of the DMA operation, the application layer is notified using a callback routine which is setup when the buffer is posted to the write buffer queue. The host processor is also notified at the successful completion of the DMA operation using an outbound message of type CYC_TO_HOST_DMA_WR_DONE. An unsuccessful transfer results in an outbound message of type CYC_TO_HOST_DMA_ERROR being sent to the host.

Read buffers are filled buffers which are posted to the read buffer queue and are used by the driver when the host issues a HOST_TO_CYC_DMA_READ command. When the driver receives a HOST_TO_CYC_DMA_READ message from the host, it pulls a filled buffer from the read buffer queue and kicks off a DMA Write operation to push data into the host's buffer from the filled buffer. Upon completion of the DMA operation, the application layer is notified using a callback routine which is setup when the buffer is posted to the read buffer queue. The host processor is also notified at the completion of the DMA operation using an outbound message of type CYC_TO_HOST_DMA_RD_DONE. An unsuccessful transfer results in an outbound message of type CYC_TO_HOST_DMA_ERROR being sent to the host.

To minimize the amount of copying that needs to be done, the driver allocates physically contiguous buffer memory during initialization. This buffer area is made accessible to the application in its address space via an ***mmap*** call to the driver. In the current version, the number and size of the buffers is set at driver compile time. The application and the driver must be consistent with respect to the number and size of the buffers.

To support the posting of empty write buffers for use by the driver, the system call ***cyc_post_write_message*** has been added. The function prototype for the system call is as follows:

int cyc_post_write_message (void *p_message, int nbytes,
void (*callback)(void *, int), void *arg)

The arguments to the call are a pointer to the empty message buffer, the size of the buffer, and a callback routine function pointer.

To support the posting of filled read buffers for use by the driver, the system call ***cyc_post_read_message*** has been added. The function prototype for the system call is as follows:

int cyc_post_read_message (void *p_message, int nbytes,
void (*callback)(void *, int), void *arg)

The arguments to the call are a pointer to a filled message buffer, the size of the data in the buffer, and a callback routine function pointer.

To allow the driver to handle application-level callback functions for write buffers, the system call ***cyc_wait_wr_complete*** has been added. This function does not return until a semaphore within the driver, indicating that a write operation has completed, is incremented. Upon completion of the operation, the parameters associated with the previously-specified callback function (set when the buffer was posted) are returned to the calling application allowing the callback function to be executed at the application level. The function prototype for the system call is as follows:

int cyc_wait_wr_complete (unsigned long *callback,
unsigned long *arg,
unsigned long *status)

See the example application for further details on the usage of this function.

To allow the driver to handle application-level callback functions for read buffers, the system call ***cyc_wait_rd_complete*** has been added. This function does not return until a semaphore within the driver, indicating that a read operation has completed, is incremented. Upon completion of the operation, the parameters associated with the previously-specified callback function (set when the buffer was posted) are returned to the calling application allowing the callback function to be executed at the application level. The function prototype for the system call is as follows:

```
int cyc_wait_rd_complete (unsigned long *callback,
                          unsigned long *arg,
                         unsigned long *status)
```

See the example application for further details on the usage of this function.

In addition, kernel level code can post physically contiguous buffers to the driver using the exported functions:

```
EXPORT_SYMBOL(cyc_post_read_message);
EXPORT_SYMBOL(cyc_post_write_message);
```

Functionally, these functions are identical to the system calls. One difference however is that since the buffers are posted from the kernel level, the callback functions can be called directly from the driver upon completion of the host read or write transfer.

In order to allow an application to access the mmap functionality of the XScale driver, a device node must be created in the root filesystem's /dev directory. After starting the kernel, perform the following:

```
grep mmap /proc/devices
```

The result of this command should be something similar to:

```
254 mmap
```

Create the device node file (assuming major number 254):

```
mknod /dev/cycmsgmem c 254 0
```

The resulting device node is as follows:

```
crw-r--r--    1 root     root     254,   0 May  9 16:05 cycmsgmem
```

Host Driver

The host-based portion of the driver is contained within the files ***cyclhdd.c*** and ***cyclhdd.h***. At its most basic level, the block storage style driver issues request messages to the XScale IOP processor via the MU queues. The two main requests are HOST_TO_CYC_DMA_READ and HOST_TO_CYC_DMA_WRITE. These messages are issued based on the **read()** and **write()** system calls from an application.

When an application issues a **write()** call, the driver de-queues a free kernel write buffer and then copies the user data into the buffer using **copy_from_user()**. After the copy is completed, the driver retrieves a free inbound message buffer from the IOP using the MU. Using the message buffer, the driver composes a HOST_TO_CYC_DMA_WRITE message and then writes the message pointer to MU Inbound Queue Port thereby interrupting the IOP XScale core. The driver then sleeps on a semaphore which is not incremented until the driver receives a DMA completion message from the IOP. At this point the write operation is completed and control is returned to the application.

When an application issues a **read ()** call, the driver de-queues a free kernel read buffer and retrieves a free inbound message buffer from the IOP using the MU. Using the message buffer, the driver composes a HOST_TO_CYC_DMA_READ message and then writes the message pointer to MU Inbound Queue Port thereby interrupting the IOP XScale core. The driver then sleeps on a semaphore which is not incremented until the driver receives a DMA completion message from the IOP. At this point, the driver copies the data from the kernel buffer into the user buffer using **copy_to_user()**. After the copy is completed, the read operation is completed and control is returned to the application.

In order to allow an application to access the messaging driver, a device node must be created in the root filesystem's /dev directory. To create the device node, perform the following:

```
mknod /dev/cyclhdd c 42 0
chmod 0666 /dev/cyclhdd
```

The resulting device node is as follows:

```
crw-rw-rw-    1 root     root      42,   0 May 21 12:44 cyclhdd
```

Future Enhancements

In order to allow users to take advantage of the network interfaces supported by the Cyclone IOP, some additional functionality is planned. The message set between the host and the XScale IOP will be expanded to allow the host to create, read from, and write to sockets on the IOP. All IP processing will be done on the IOP with the DMA engine moving the socket data in and out of the host memory.