LINUX DEVICE DRIVER FOR THE IBM CHARM-LITE ASYNCHRONOUS TRANSFER MODE ADAPTER

---

by

Ishraq Ahmed

Submitted to

the graduate faculty

of the Department of Computer Science

In Partial Fulfillment

of the Requirements for the Degree of

Master of Science

in

Computer Science

May 12, 1999

Clemson University

Clemson, SC 29634-1906

Abstract

Asynchronous Transfer Mode (ATM) delivers important advantages over existing LAN and WAN technologies, including the promise of scalable bandwidths at unprecedented price and performance points and Quality of Service (QoS) guarantees, which facilitate new classes of applications such as multimedia. For a long time, ATM remained a long-haul technology with core backbone routers using it for high speed transfer over fibre optic cables. But recently, ATM technology has started to play a role in the evolution of current workgroup, campus and enterprise networks. True end-to-end ATM networks are possible with the help of reliable, high-performance ATM Network Interface Cards that will bring ATM to the desktop. The CHARM-lite chip developed by IBM is capable of delivering up to 622 Mbps with suitable configurations.

This paper describes the Linux Device Driver that is being developed at the Department of Computer Science, Clemson University for the CHARM-lite chipset based ATM adapter from IBM Corporation.

Acknowledgments

First and foremost, I thank Dr. Westall for accepting my plea to drop my network performance evaluation related project and work on what I thought would be a more exciting and programming intensive research project. From then on, his invaluable guidance and support helped me at every stage of this project. He gave me insightful suggestions just when I needed them most. His knowledge of the Linux kernel and experience in writing device drivers proved to be extremely useful to me.

I would also like to thank Dr. Geist for being my second advisor, but prior to that, for being the person who really got me interested in the Linux kernel with his excellent teaching in the Operating Systems Case Study course in Spring 1998.

I would like to express my gratitude to my parents and my elder brother for their unflinching support and to my twin brother whose 20-year-long competition in academics was probably a good reason for me being what I am today.

Contents

List of Figures

Chapter 1
Introduction

1.1 Overview of ATM

Asynchronous Transfer Mode (ATM) is an evolving network technology designed to carry a heterogeneous traffic stream that includes voice, video and data. ATM provides connection oriented service with best-effort delivery and can provide quality of service (QoS) guarantees with regard to throughput and jitter control[1].

An ATM connection between two network end-points is called a Virtual Channel Connection (VCC). Two distinct connection types are supported. A Permanent Virtual Channel (PVC) is analogous to a leased line in the telephone network. It must be manually configured by a system administrator and persists until a system administrator manually destroys it. A Switched Virtual Channel (SVC) is analogous to a dialed connection in the telephone network. It is created in response to a request by one of the end-points and persists only until one of the end-points terminates it.

The 53-byte ATM cell is the basic unit of data transfer in an ATM network. To adapt the raw cell pipe provided by the ATM network to the specific requirements of various real-time and non-real-time applications, some standard transport-like protocols commonly called *ATM adaptation layers (AALs)* have been defined. These adaptation layers are present only in end-point nodes of an ATM network. Seven AALs have been specified to date. The most significant is AAL5 which is the de facto standard for all Variable Bit Rate (VBR) traffic in ATM based computer networks. AAL6 improves on AAL5 by adding header cut-through support. AAL7 adds header and packet cut-through. Cut through support allows software to be written that minimizes latency and the number of interrupts (events) that occur.

[1] This section is based on the ATM overview presented in [2]

ATM adaptation layers are often viewed as consisting of two sublayers: the convergence sublayer (CS) and the segmentation and reassembly sublayer (SAR). In AAL5 the function of the CS is to perform framing and error detection while SAR's job is to segment AAL5 frames into ATM cells for transmission and to reassemble AAL5 frames from ATM cells on reception.

Development of ATM protocol support in Linux is being coordinated by Werner Almesberger (the author of LILO) at the LRC (*Laboratoire de Reseaux de Communication*) in Lusanne, Switzerland. Although development is ongoing, ATM in Linux now provides most of the functionality expected in a state-of-the-art ATM network. Both PVCs and SVCs are supported. ATM services are delivered to applications via a socket based API.

The Linux ATM protocol support is completely independent of the characteristics of any underlying network adapter. To achieve this independence, the protocol depends upon the existence of a low-level, adapter-dependent device driver to implement a small set of functions like receive-frame, send-frame etc.

1.2 The IBM CHARM-lite 155Mbps ATM adapter

The IBM CHARM-lite chipset based ATM adapter provides a highly functional way to deliver ATM service to a user workstation. The CHARM-lite chip is an ATM support chip. It acts as an interface between a Peripheral Component Interconnect (PCI) bus and an ATM Physical Layer device. This chip supports an integrated packet/frame memory and performs the SAR functions for several of the ATM adaptation layers. A detailed functional description of this chip is given in chapter 3.

1.3 Organization of the paper

Chapter 1 gave a brief introduction to ATM and the CHARM-lite chip. Chapter 2 discusses some general concepts about writing device drivers in Linux. It describes the features of Linux that are useful for a device driver writer: installable modules, character device interface, PCI device support, memory mapping and interrupt handling. Chapter 3 gives a detailed functional description of the CHARM-lite chip. Chapter 4 explains the initialization and shutdown of the device driver. It concentrates on the Linux side of the house and doesn't delve into the device specific initialization which forms the core of chapter 5. Chapter 6 deals with the user-level control of the device driver. It describes the tools that were developed for using and testing the driver. The paper concludes with a brief note on the possible future enhancements to the driver.

Chapter 2
Device Driver Support in Linux

2.1 Installable Device Drivers

Linux supports kernel extensions through dynamically loadable modules. This facility significantly reduces driver development time by reducing the number of kernel rebuilds and allowing installation and testing without system reboots. A module is a collection of C functions containing at least the following two specific functions *init_module()* and *cleanup_module()*. The collection is compiled and linked as a single object file, *charm.o* in our case, and then loaded with the command */sbin/insmod charm.o*. The *init_module()* function is called when the module is loaded, and the *cleanup_module()* is called when the module is removed, using the command */sbin/rmmod charm*.

The *init_module* function performs two distinct tasks. First, it must establish bindings between the device driver and the kernel. These bindings are established via various *registration* calls. This driver registers itself as a character device driver using the *register_chrdev* kernel call. Second, the driver must properly initialize the hardware device. A device driver must also contain a *cleanup_module* function. This function must unregister every item that was registered by the module, free any memory allocated in the module, and properly quiesce the hardware.

The */sbin/lsmod* command is used to display the modules currently installed. For example, on the machine glint2.cs.clemson.edu:

```
[ahmed@glint2]$ /sbin/lsmod
Module                  Size  Used by
charm                  18420   0  (unused)
fw                       448   0  (unused)
```

In the output shown above, *charm* is our device driver module and *fw* is another module that we configured to be dynamically loaded when the system boots. It happens to be a firewall to keep out malicious hackers.

2.2 Character Device Interface

The character device interface is a simple mechanism to provide user level access to the device driver. A character device can be accessed like a file, and its device driver is responsible for implementing this behavior. Such a driver usually implements the *open, close, ioctl, read, and write* system calls. Char devices are accessed by means of file system nodes usually located in the */dev* directory. Device files are special files and are identified by a "c" in the first column of the output of *ls -l*, indicating that they are char nodes. If you issue the *ls* command, you'll see two numbers (separated by a comma) on the device file entries before the date of last modification, where the file length normally appears. These numbers are the "major" and "minor" numbers for the particular device. The following listing shows a few devices as they appear on glint2.cs.clemson.edu. The major major numbers are 1, 10, 2 and 4, while the minor numbers are 0, 1, 3, 5 and 64-65.

```
crw-rw-rw-   1 root     root       1,   3 May  5  1998 null
crw-rw-r--   1 root     root      10,   1 May  5  1998 psaux
crw-rw-rw-   1 root     root       2,   0 Mar 29 22:03 ptyp0
crw-rw-rw-   1 root     root       2,   1 Mar 30 00:15 ptyp1
crw-r--r--   1 root     root       4,  64 May  5  1998 ttyS0
crw-r--r--   1 root     root       4,  65 May  5  1998 ttyS1
crw-rw-rw-   1 root     root       1,   5 May  5  1998 zero
```

The major number identifies the driver associated with the device. For example, */dev/null* and */dev/zero* are both managed by driver 1, while all the tty's and are managed by driver 4. The kernel uses the major number to associate the appropriate driver with its device.

The minor number is only used by the device driver; other parts of the kernel don't use it, and merely pass it along to the driver. It is not unusual for a driver to control several devices (as in the example above)–the minor number provides a way to differentiate among them.

Adding a new device to the system requires assigning a major number to it. The assignment should be made at driver (module) initialization by calling the following function defined in `<linux/fs.h>`:

```
int register_chrdev(unsigned int major, const char *name,
                    struct file_operations *fops);
```

The return value is the error code. A negative return code indicates an error; a zero or positive return code means successful completion. The `major` argument is the major number being requested, `name` is the name of the device, which will appear in */proc/devices*, and `fops` is the pointer to a jump table used to invoke the driver functions. The table looks like this:

```
struct file_operations {
   loff_t (*llseek) (struct file *, loff_t, int);
   ssize_t (*read) (struct file *, char *, size_t, loff_t *);
   ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
   int (*readdir) (struct file *, void *, filldir_t);
   unsigned int (*poll) (struct file *, struct poll_table_struct *);
   int (*ioctl) (struct inode *, struct file *,
                                    unsigned int, unsigned long);
   int (*mmap) (struct file *, struct vm_area_struct *);
   int (*open) (struct inode *, struct file *);
   int (*release) (struct inode *, struct file *);
   int (*fsync) (struct file *, struct dentry *);
   int (*fasync) (int, struct file *, int);
   int (*check_media_change) (kdev_t dev);
   int (*revalidate) (kdev_t dev);
   int (*lock) (struct file *, int, struct file_lock *);
};
```

Of all these function pointers, we are most interested in the `open`, `ioctl` and `close`.

Applications need a name by which to refer to the driver. So a name must be inserted into the */dev* directory and associated with the device's major and minor numbers.

The command to create a device node on a filesystem is *mknod*. It requires superuser privileges. The command takes three arguments in addition to the name of the node being created. For instance, the command:

```
mknod /dev/charm155 c 79 1
```

creates a char device (`c`) whose major number is 79 and whose minor number is 1. And the name `charm155` will appear in */proc/devices*.

To access the driver, an application will first have to issue the *open* system call and then use *ioctl* to communicate with the device. When finished with the device, it should call *close* on the device.

When a module is unloaded from the system, the major number should be released. This is accomplished with the following function, which is called from *cleanup_module*:

```
int unregister_chrdev(unsigned int major, const char *name);
```

The arguments are the major number being released and the name of the associated device. The kernel compares the name to the registered name for that number. It returns an error code if there is a mismatch in the names or if the major number is invalid.

2.3 PCI Device Support

2.3.1 Introduction to PCI

The Peripheral Component Interconnect (PCI) Bus is present in a wide variety of microcomputers ranging from Intel-based PC architectures to DEC-Alpha-based workstations. The PCI architecture was designed as a replacement for the ISA standard, with three main goals: to get better performance when transferring data between the computer and its peripherals, to be as platform-independent as possible, and to simplify adding and removing peripherals to the system.

2.3.2 PCI Addressing

Each peripheral is identified by a *bus* number, a *device* number, and a *function* number. While the PCI specification permits a system to host up to 256 buses, PCs have only one. Each bus hosts up to 32 devices, and each device can be a multi-function board. The hardware circuitry of the board is capable of responding to queries pertaining to three address spaces: memory locations, I/O ports, and configuration registers. The memory and I/O spaces are shared by all the devices on a PCI bus while the configuration space exploits "geographical addressing" i.e., each slot has a private enable wire for configuration transactions, and the PCI controller accesses one board at a time with no address collision. As far as the driver is concerned, memory and I/O are accessed in the usual ways via *inb*, *memcpy*, etc. Configuration transactions, on the other hand, are performed by calling specific kernel functions to access configuration registers.

2.3.3 Boot Time

When power is applied to a PCI device, the hardware is placed in the *reset* state. In this state, the device will only respond to configuration transactions. Memory and I/O ports are not mapped in the computer's address space and every other device specific feature, such as the interrupt lines, is disabled as well. Every PCI motherboard is equipped with PCI-aware firmware, called the BIOS. The firmware offers access to the device configuration address space, even if the processor's instruction set doesn't offer such a capability.

At system boot, the BIOS performs configuration transactions with every PCI peripheral in order to allocate a safe place for any address region it offers. By the time a device driver accesses the device, its memory and I/O regions have already been mapped into the processor's address space. The device driver doesn't usually have to change this default assignment. The user can look at the PCI device list by reading */proc/pci*, which is a text file that has an entry for each PCI board in the system. Here is how our adapter appears in this file:

```
Bus  0, device  13, function  0:
  ATM network controller: IBM Unknown device (rev 3).
    Vendor id=1014. Device id=4f.
    Medium devsel.  Fast back-to-back capable.  BIST capable.
              IRQ a.  Master Capable.  Latency=64.
    Non-prefetchable 32 bit memory at 0xfedfc000 [0xfedfc000].
    Prefetchable 32 bit memory at 0xfede0000 [0xfede0008].
```

This tells us that our adapter has on-board memory that has been mapped to address 0xfede0000. There is an other address space at 0xfedfc000 that can be used to access the registers.

2.3.4 Detecting the Device

Each PCI device features a 256-byte configuration space. The first 64 bytes are standardized while the rest are device-dependent. Three PCI registers identify a device: *vendor*, *deviceID*, and *class.* Every PCI peripheral puts its own values in these read-only registers, and the driver can use them to look for the device.

vendor:

This 16-bit register identifies a hardware manufacturer. For instance, every IBM device is marked with the same vendor number 0x1014. There is a global registry of such numbers, and manufacturers must apply to have a unique number assigned.

deviceID:

This is another 16-bit register, selected by the manufacturer; no official registration is required for the *deviceID*. The *deviceID* for our adapter is 0x004f as mentioned in its hardware documentation.

class:

Every peripheral device belongs to a *class*. The *class* register is a 16-bit value whose top eight bits identify the "base class" (or *group*). For example, "ethernet" and "token-ring" are two classes belonging to the "network" group, while "serial" and "parallel" classes belong to the "communication" group.

Linux offers the following two functions for detecting the BIOS and identifying a PCI device.

```
int pcibios_present(void);
```

Since the PCI-related functions don't make sense on non-PCI computers, the *pcibios_present* function tells the driver if the computer supports PCI; it returns a non-zero value if the BIOS is PCI-aware.

```
struct pci_dev *pci_find_device (unsigned int vendor,
                      unsigned int device, struct pci_dev *from);
```

If `CONFIG_PCI` is defined and *pcibios_present* is true, this function is used to request information about the device from the BIOS. The `vendor/device` pair identifies the device. The call to *pci_find_device* returns a pointer to the following structure defined in `<linux/pci.h>`.

```
struct pci_dev {
   struct pci_bus *bus;           /* bus this device is on           */
   struct pci_dev *sibling;       /* next device on this bus         */
   struct pci_dev *next;          /* chain of all devices            */
   void     *sysdata;             /* hook for sys-specific extension */
   struct proc_dir_entry *procent;/* device entry in /proc/bus/pci   */
   unsigned int   devfn;          /* encoded device & function index */
   unsigned short vendor;
   unsigned short device;
   unsigned int   class;          /* 3 bytes: (base,sub,prog-if)     */
   unsigned int   hdr_type;       /* PCI header type                 */
   unsigned int   master : 1;     /* set if device is master capable */
   unsigned int   irq;            /* irq generated by this device    */
   unsigned long  base_address[6];/* address spaces on the device    */
   unsigned long  rom_address;
};
```

2.3.5 Accessing the Configuration Space

After the driver has detected the device, it usually needs to read from (or write to the three address spaces: memory, port, and configuration. The configuration space can be accessed through 8-bit, 16-bit, or 32-bit data transfers. The relevant functions, *pcibios_read_config_byte*, *pcibios_read_config_word*, *pcibios_read_config_dword*, *pcibios_write_config_byte*, *pcibios_write_config_word*, and *pcibios_write_config_dword* are all prototyped in `<linux/bios32.h>`. The multi-byte read functions convert the value just read from little-endian to the native byte order of the processor. The multi-byte write functions convert the value to little-endian before writing to the device. So endianness is not an issue with these functions.

A pitfall:

It is possible that memory, port addresses and interrupt numbers of PCI devices might have been remapped by the Linux kernel. So these must be read from the `pci_dev` structure returned by *pci_find_device* and not from the config space. We had this problem initially with our adapter when we first tried to read the config space on the board for addressing information. So we now read these numbers from the `pci_dev` structure instead.

2.4 Memory Mapping

Memory on a PCI device has to be mapped to the processor's address space before it can be accessed. This can be done using *ioremap*:

```
void * ioremap(unsigned long offset, unsigned long size);
```

System initialization does not probe for PCI buffers but leaves each driver responsible for managing buffers on its own device. For example, if the frame buffer of a VGA device has been mapped to the address 0xf0000000 (a typical value), *ioremap* can be used to build the correct page tables for the processor to access it. Thus *ioremap* will not remap a physical address that doesn't start at a page boundary. The return value of *ioremap* is a virtual address that can be used to access the specified physical range; the virtual address obtained must be eventually released by calling *iounmap*.

```
void iounmap(void *addr);
```

readX/writeX() [1] are used to access memory mapped devices. On the x86 architecture, these functions are implemented as macros that read or write the memory location directly. These functions are defined in `<asm/io.h>`:

```
#define readl(addr) (*(volatile unsigned int *)__io_virt(addr))
#define writel(b,addr) (*(volatile unsigned int *)__io_virt(addr)=(b))
```

Based on these functions, we defined macros that can conveniently access the registers on the board:

```
/* macros for accessing the registers on the board.  */
#define RD_REGL(reg_id)  (readl(charm155.vregbase + (reg_id)/4))
#define WT_REGL(value,reg_id) (writel((value), \
                               (charm155.vregbase + (reg_id)/4))
```

where `charm155.vregbase` is the *ioremap*ed register space; and `reg_id` is the offset address of the register on the board.

2.5 Interrupt Handling

Since our driver has extensive interrupt handling, a brief description of interrupt handling in Linux is in order. The kernel keeps a registry of interrupt lines, similar to the registry of I/O ports. A module is allowed to request an interrupt channel (or IRQ, for Interrupt ReQuest) and release it when it is done. The following functions defined in `<linux/sched.h>` implement the interface:

```
int request_irq(unsigned int irq,
                void (*handler)(int, void *, struct pt_regs *),
                unsigned long flags,
                const char *device,
                void *dev_id);
void free_irq(unsigned int irq, void *dev_id);
```

[1] This notation refers generically to the family of macros: *readb()*, *writeb()*, *readw()*, *writew()*, *readl()* and *writel()*

Chapter 3
CHARM-lite Functional Description

The IBM CHARM-lite is an extremely complicated Asynchronous Transfer Mode support chip used to build ATM adapters capable of very high data rates. It is internally organized into a set of logical entities that interact with each other to provide a wide array of features. This chapter makes an attempt to explain the major functions of the device and also discusses the interaction of the driver with the hardware. We will henceforth refer to the "IBM CHARM-lite based ATM adapter", simply, as CHARM.

3.1 Logical Entities in CHARM

This section briefly describes the logical entities that comprise CHARM. Each of these entities has a set of registers that can be modified to alter the behavior of the entity. Most of the entities are capable of raising interrupts to the processor and can be configured to do so when certain events occur. Some entities require the device driver to initialize certain data structures located at specific memory areas in CHARM before they are ready to perform certain tasks. The role of some of these entities will become clear when we discuss the specific features of CHARM in different sections of this chapter. The details of how each entity is initialized by the device driver are given in chapter 5.

3.1.1 Control Processor Bus Interface

These entities manage the hardware interaction of the device with the processor, providing the driver writer with a relatively simple interface to work with.

- PCINT: PCI Interface entity

This entity provides PCI [1] specific interfacing between the external connection and the internal entities. The various configuration registers such as the IRQ line and the base addresses are setup by the BIOS upon system start-up. These are further modified by the Linux kernel as it boots up and finds any inconsistencies.

- INTST: Interrupt Status entity

  This entity contains the masking registers that will choose which interrupt/status source can generate an interrupt. It also has high-level controls such as Master-chip-enable and LCD [2] size selector. It also features a general purpose timer.

- GPDMA: General Purpose DMA entity

  This provides DMA (Direct Memory Access) control between system memory and CHARM packet memory. There are two ways to initiate DMA transfers. The first is by directly writing the source address, destination address and transfer count and flag registers. The second is by writing descriptors [3] in system memory that directly map to the three registers and issuing an Enqueue DMA Descriptor primitive which is part of the interface provided by GPDMA to the device driver. The current version of the device driver doesn't support DMA.

3.1.2 Memory Control

These entities manage the on-board control and packet memory on CHARM. Control memory contains all the major data structures used by CHARM whereas the

[1] *Peripheral Component Interconnect* - See section 2.3 for a detailed explanation of how PCI devices are handled in Linux.

[2] *Logical Channel Descriptor* - See section 3.2 for an explanation of Logical Channels in CHARM.

[3] A DMA descriptor is a structure containing the source address, destination address and byte count along with some other optional information.

packet memory contains the data from the cells that have been received and those that are to be transmitted.

- COMET/PAKIT: DRAM Controlling entity

  This entity generates the signals necessary for DRAM accesses. These include RAS, CAS, Write enable and Output enable. The device driver doesn't need any direct interaction with COMET; it is used internally by CHARM.

- VIMEM: Virtual Memory controller

  This entity manages the Virtual Address Space on the adapter. Packet sizes up to 64K are supported by forming virtual buffers from several (up to 16) real buffers of (up to 5) different sizes. It keeps track of the real buffers that make up a virtual buffer with the help of buffer maps (similar to page tables). Virtual memory support in CHARM is discussed in detail in the next section.

- BCACH: Bus Cache entity

  This entity is also used internally by CHARM and the device driver has minimal interaction with it. It provides the caching function for data transfers on the Control Processor Bus. The array is organized as four logically separate 32-byte cache lines, any of which can be used for processor accesses or master/slave DMA accesses.

- POOLS: Memory Pool manager

  This is the primary memory manager of CHARM. A very user-friendly interface hides the extremely complex memory management that goes on internally. Memory buffers are checked out and checked in via two primitives supported by POOLS: the *get pointer primitive* and the *free pointer primitive*. POOLS also contains mechanisms to control resource utilization and supports a real memory mode and a virtual memory mode.

3.1.3 Transmit Data Path

- SEGBF: Cell Segmentation entity

  This entity accepts frames either from the device driver or internally, and then generates cells to be sent out over the external physical interface.

- CSKED: Cell Scheduler

  This transmit cell scheduling entity is responsible for receiving a packet from the device driver, determining when cells from the packets need to be transmitted, passing this information to the segmentation buffer (SEGBF) entity, and handling congestion control. CSKED and SEGBF together play a major role in transmitting data. A more detailed explanation of the transmit cycle is given in a later section.

3.1.4 Receive Data Path

The following entities are responsible for processing the cells received from the network and reassembling them into frames according to the rules of the AAL [4] being used.

- REASM: Cell Re-assembly entity

  This entity collects bytes from the physical interface into cells, while discarding null cells and idle symbols. It also checks and corrects HEC errors. It looks up the control memory for the proper Logical Channel Descriptor address and then passes it to the Receive AAL processing entity.

- RAALL AAL processor

[4] ATM Adaptation Layer - See section 1.1 for a brief discussion of AALs.

This implements all the receive functions of the ATM Adaptation Layers (AAL) viz. Protocol Verification, Cell Reassembly into Packets, Cell CRC Verification, Packet Reassembly Timeouts and Errors. The following AALs are supported:

  - AAL 0 - Raw Cell Mode
  - AAL 0 - Receive FIFO
  - AAL 5 - High Speed Data Transfer with Variable Bit Rate
  - NULL AAL 5 - AAL 5 without protocol error checking
  - AAL 6 - AAL 5 with header cut through
  - AAL 7 - AAL 5 with header and packet cut through.

- RXQUE: Receive Queue manager

  This entity manages the receive packet queues for software by providing an easy to use primitive interface. A group of 8 receive queues are available. The receive queues hold events or user specified data. Each entry (an event) is 32 bits long and contains an event-type field and some event information. The events are obtained by the device driver by executing a dequeue operation. The device driver can be notified of new events by setting up the appropriate registers or the queue length can be polled to check for new events.

3.1.5 PHY Level Interfaces

- LINKC: Asynchronous Physical Layer interface

  LINKC provides the interface between the CHARM-lite and the ATM Physical Layer (PHY) device. The current version of the device driver disables the PHY and uses internal loopback instead.

- NPBUS: Nodal Processor Bus interface

  This entity controls the signals of the Nodal Processor Bus, which connects CHARM with the ATM PHY. The registers of the PHY are accessible through this entity from the address space of CHARM. Thus any PHY specific commands can be issued using the NPBUS registers.

3.1.6 Hardware Protocol Assist

- CHKSM: TCP/IP Checksum Logic

  The CHKSM entity is capable of initializing and testing packet and control memory and can perform 2's complement, 16-bit sum with end-around-carry. When the device driver is installed, it uses this entity to initialize the memory on CHARM.

3.1.7 Base Device Functions

- CRSET: Hardware and Software Reset Controlling entity

  This entity will perform built-in-self-test, flush and unit reset operation. Chip software resets can be controlled by this entity.

3.2 Logical Channel Support

The Logical Channel is the unit of resource allocation in CHARM. It describes the characteristics of an end-station to end-station connection. CHARM maintains data structures in its control memory that describe the logical channels. These are called LCDs (Logical Channel Descriptors).

An LCD consists of two sections: a transmit descriptor and a receive descriptor. The complete LCD data structure is given in Appendix B. The transmit portion of an LCD *must* be initialized before any transmission can take place and the receive portion must be initialized before any received cells can be processed. The LCD is thus the basis of all transmission and reception occurring in the device. For transmitting a packet, the lcd-address field in the packet header has to be written before the packet is queued to CSKED for transmission. The control memory contains a

VPI/VCI to LCD translation table which is used to determine the LCD to be used for processing each of the cells received. More details about the use of the LCD in transmitting and receiving data are given in later sections of this chapter.

3.3 Memory Pools

As mentioned briefly before, the POOLS entity manages the memory buffers resident on CHARM. Memory buffers are checked out and checked in by the device driver via two primitives supported by POOLS: the get-pointer primitive and the free-pointer primitive. POOLS also contains mechanisms to control resource utilization and supports a real memory mode and a virtual memory mode.

3.3.1 Real Mode Operation

POOLS maintains a circular list of available buffers. There are pointers (head and tail) to the start and end of the list. When a get pointer is executed, the buffer at the head of the list is checked out and the head pointer is advanced. When a free pointer is executed, the freed buffer is checked in at the end of the list and the tail pointer is advanced.

3.3.2 Virtual Mode Operation

With the addition of virtual memory, POOLS now maintains 5 sets of head and tail pointers; one for the virtual buffers themselves and the rest for the four regions of real buffers that constitute the virtual buffers. In this case, the base virtual address of the buffer is the item returned from a get pointer operation and returned during a free pointer operation.

3.3.3 Resource Controls

POOLS regulates resource utilization by providing 16 different pools of buffers. For each pool, a maximum number of allowable buffers is specified. The idea is to make it possible for several applications to use CHARM at once without starving each other for memory buffers. A particular pool's buffers are divided into *guaranteed* and *common* buffers. *Common* buffers are the buffers remaining after all the *guaranteed* buffers are subtracted from the total buffers. Two thresholds are maintained per pool per buffer size - the *guaranteed_threshold* and the *total_threshold*. An *active_count* is maintained for each pool for each buffer size. A request for a buffer is granted only if one of the following is true:

- $active\_count \leq guaranteed\_threshold$
- $guaranteed\_threshold < active\_count < total\_threshold$ and a common buffer is available.

3.4 Virtual Memory Support

This section gives a detailed explanation of how virtual memory is implemented in CHARM. To the device driver, the packet memory space appears as if it is up to 64K buffers each of which can (architecturally) appear to be 64K bytes long. A level of indirection has been added to the addressing of packet memory to provide these large frame buffers without requiring memory behind all of them at the same time. This has been done for a number of reasons: The frames on the network can be up to 64KB long. The receiver does not know how long a frame will be until it is completely received. Software generally has a much easier time of dealing with contiguous memory than dealing with scattered memory on the board.

The memory does not page or swap. There two major efficiencies used internally: The first N bytes of memory in a buffer are directly referenced. The blocks that make up the buffers are of multiple sizes.

Each virtual buffer consists of a number of real buffers. For each virtual buffer there is a buffer map that defines the size and number of real buffers that may be allocated to the virtual buffer (See figure 3.1). Each map is built from a common template (the VIMEM Virtual Buffer Segment Size Register) that associates 1 to n buffer indexes in the map to a real buffer in 1 of the 4 real buffer regions defines in VIMEM. In VIMEM, the Buffer Map Base Address Register defines the size of the map and therefore also the number of buffer indexes in the virtual buffer map. Each 8-byte entry of the map contains the pool ID of the pool to which the buffer is allocated plus space for 3 real buffer segment indexes. This implies the smallest map yields a virtual buffer of 1-4 real segments (3 real buffer segments plus the implicit real buffer that all virtual buffers are allocated). The biggest map defines a virtual buffer of 1-16 real buffer segments (15 plus implicit).

The idea behind this structure is to allow the user to customize the value in the Virtual Buffer Segment Size Register to utilize memory in an efficient manner relative

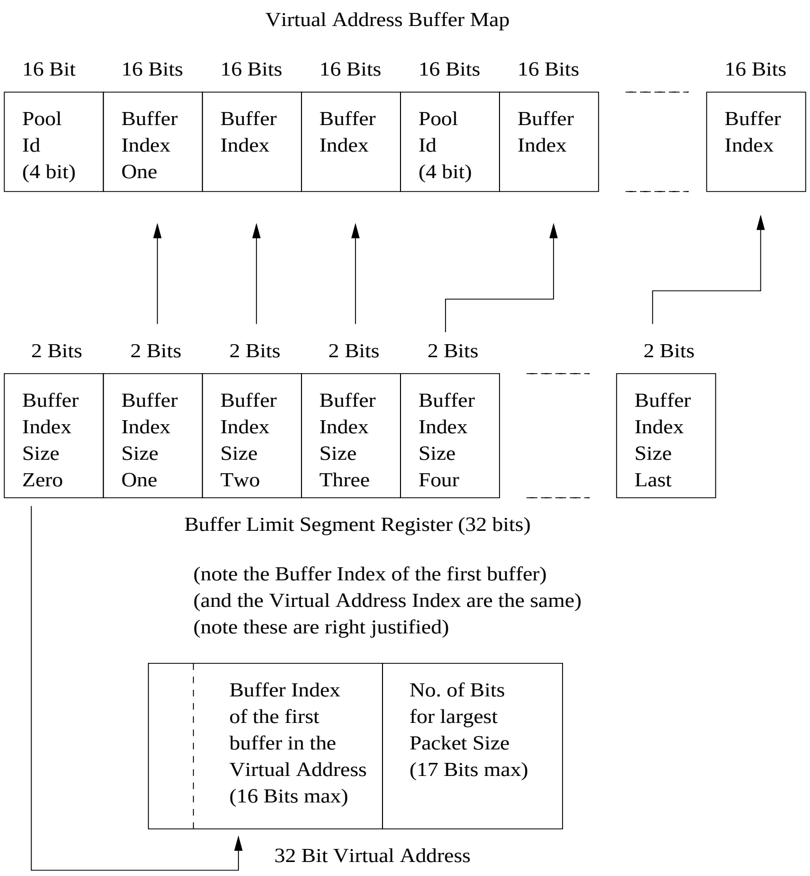

Figure 3.1 Virtual Address Buffer Map

to network data traffic. For example, if network traffic contained 50% packets of less than 512 bytes, 35% packets of less than 1K bytes, and the rest was less than 5K bytes, the user could set up virtual memory to use 3 real segments of 512 bytes, 512 bytes and 4K bytes. The incoming data would neatly fit into the segments and minimize wasted memory.

POOLS and VIMEM maintain the maps for the virtual buffers. On a write that crosses a real buffer boundary, into an as yet unresolved region of a virtual buffer, a page fault occurs. When a page fault occurs, POOLS determines whether or not a real buffer can be assigned. If it can be assigned, the index of the real buffer relative to the base address of the particular buffer size is placed by VIMEM into the buffer map. The first buffer is implicitly associated with the virtual memory address for a particular virtual buffer and enough real memory must be available to support the first real buffer of each virtual buffer at initialization time. There is not necessarily enough real storage for all the possible real buffers associated with a virtual buffer, which means that a non-recoverable page-fault can occur. In such a situation, VIMEM raises an interrupt for the device driver to handle. The current version of the driver doesn't handle this interrupt.

3.4.1 An Example

The following example illustrates the above concept with the help of arbitrary values for the various buffer sizes. Let us assume the following buffer sizes:

Size 0	512 bytes
Size 1	1024 bytes
Size 2	2048 bytes
Size 3	8192 bytes
Size 4 (Implicit Buffers)	256 bytes

Note that the Implicit Buffer size is 256 bytes. This means that every virtual buffer will be at least 256 bytes long. As the virtual buffer grows, it can add on real buffers of sizes 512 bytes, 1K, 2K and 8K depending on how the VIMEM Virtual

Buffer Segment Size register is set up. Let us assume that the register is set up as follows:

Virtual Buffer Segment Size register:

11	11	11	11	11	11	11	10	01	01	01	00	01	01	00	00

This register will establish how VIMEM will select a buffer from the available real buffers of various sizes, to be added to a virtual buffer whenever a page fault occurs while accessing the virtual buffer. Each pair of bits in the figure above refers to a real buffer size according to the following table:

00	this real buffer segment is a Size 0 Real Buffer (512-byte buffer)
01	this real buffer segment is a Size 1 Real Buffer (1K buffer)
10	this real buffer segment is a Size 2 Real Buffer (2K buffer)
11	this real buffer segment is a Size 3 Real Buffer (8K buffer)

Let us consider the following situation: The device driver has to transmit a 10K sized packet. It requests a (virtual) buffer from CHARM using the get-pointer primitive provided by POOLS. The POOLS entity returns a virtual buffer address to the device driver. The driver now starts transferring the packet data on to this buffer. But since the virtual buffer initially has only one (Implicit) real buffer of size 256 bytes associated with it, a page fault occurs (inside CHARM) as soon as 256 bytes have been transferred from the driver to the board. Now the VIMEM entity has to service this page fault. A second real buffer has to be added to this virtual buffer. The VIMEM entity looks at the second pair of bits in the Virtual Buffer Segment Size register bits (00) and learns that a real buffer of size 512 bytes has to be combined with the virtual buffer. It appropriately updates the virtual buffer map. All this happens internally and the device driver doesn't know about it. The driver continues transferring the remaining packet data into the virtual buffer. Now another page fault occurs after 768 bytes (256 + 512) have been transferred. This time, VIMEM looks at the third pair of bits in the Virtual Buffer Segment Size register (01) and figures out that a buffer of size 1K has to be combined into this virtual buffer. It appropriately updates the virtual buffer map. Now the total size of this virtual buffer is 1792 bytes, still short of 10K packet size. This means that another page fault will occur and another real buffer will be added to this virtual buffer in a similar manner. This real buffer is also of size 1K as can be noticed by looking at the Virtual Buffer Segment Size register. This goes on until the memory requirement of the device driver is satisfied. Looking at the value of the Virtual buffer segment register, we can see that real buffers are added to the virtual buffer in the following order: 256 bytes (implicit), 512 bytes, 1K, 1K, 512 bytes, 1K, 1K, 1K, 2K, and finally 8K. This will satisfy the 10K virtual buffer size requirement of the device driver.

It is possible that a page fault occurs when there are no real buffers available. In this case an error bit is set in the header of the packet that is being transferred which is later checked by the CSKED entity as described in section 3.8.1.

So, finally the device driver ends up using a virtual buffer of size more than 16K for a 10K packet. Clearly, there is internal fragmentation. This can be avoided by more aggressive management of CHARM memory by the device driver. The device driver could, for instance, gather network traffic statistics and dynamically change the buffer sizes to minimize wastage of memory.

We can see that this virtual memory support is very convenient as far as the device driver is concerned. The driver does not have to deal with the complex scheme of things that happen inside the device. The manner in which CHARM memory is organized into buffers of various sizes is highly flexible. The current version of the driver actually tends to make things simple by using only two different real buffer sizes, apart from the implicit buffers. It organizes the packet memory of CHARM into 2280 implicit buffers of size 512 bytes each, 728 buffers of size 4K each and 88 buffers of size 8K each.

3.5 Receive Queues

The RXQUE entity has two functions - managing the receive event queues and providing an easy-to-use interface to the device driver. A group of eight receive queues are available for the device driver to use. The receive queues hold events or user-specified data.

Each queue entry (event) is 32 bits, and contains both an event type field (6 low order bits) and some event information (26 high order bits). The event information field typically contains a pointer (with low order bits zeroed) to a packet buffer, a cell buffer, or an LCD. It can also contain user-specified data. There are various types of events defined. The two most frequently used events are the **Transmit Complete Buffer Freed** event and the **Normal AAL5 Packet Received** event. Many other events are defined for various conditions occurring in the adapter.

The receive queues are maintained by RXQUE with the following operations being available to the device driver:

dequeue - Remove the entry at the head of the queue

enqueue - Add arbitrary entry at the tail of the queue

Events fall into different categories: Normal Events, Error Events, Counter Events, Transmit Complete Events, and DMA Events. These events are routed to one of the eight available queues as configured by the device driver. It is then the responsibility of the device driver to read the event queues and take appropriate action based on the type of event. The events are obtained by the driver by executing the dequeue operation on a particular queue. RXQUE can be configured such that the device driver will be interrupted when the number of events in a queue reaches a certain threshold. Otherwise, the device driver can poll the queue length to check for events. The following pseudo code illustrates how a queue is processed by the device driver:

```
// rxq was polled or interrupt occurred to get here

Event = RXQUE->Deq;              // read an event from rxq
while (Event != 0) {
```

```
    EventType = Event & 0x0000003f;  // calc event type
    EventInfo = Event & 0xffffffc0;  // calc LC or buffer ptr

    switch (EventType) {
        case(Event1):
            ProcessSimpleEvent1(EventInfo);
            break;
        case(Event2):
            ProcessSimpleEvent2(EventInfo);
            break;
            .
            .
        case(EventX):
            ProcessSimpleEventX(EventInfo);
            break;
    }

    Event = RXQUE->Deq;
}
```

When a receive queue is full, the appropriate status bit is set. When a queue is full, all subsequent events are flushed until room is available in the receive queue. If a buffer was associated with the event, it is freed back to POOLS. Therefore, it is not good to let a receive queue become full. For each one of the eight available queues, there is a threshold that can be set, such that whenever the length of the queue is greater than its threshold, RXQUE raises an interrupt. Currently, the device driver sets all thresholds to one, which means the processor is interrupted every time there is an item in the queue.

3.6 Transmit Scheduling

There is extensive support for transmit scheduling. The transmit cell scheduler entity, CSKED, is responsible for receiving a packet from the processor, determining when cells from the packets need to be transmitted, passing this information to the SEGBF entity and handling congestion control. An LCD containing scheduling paramenters must be initialized by the device driver before cell scheduling and segmentation can be started. The parameters that are important to the scheduling operation are the average interval, peak interval, transmission priority, maximum credits that can be accumulated, and congestion recovery data. Further information on these LCD fields can be found in appendix B. When congestion is detected on a channel, CHARM will recover from it according to the parameters set up in the LCD. To implement scheduling CHARM maintains data structures called *Time Wheels* in its control memory. LCDs are queued to these time wheels in an order that depends on their scheduling parameters. There are three time wheels for each of the three levels of traffic priority: low, medium, and high.

3.7 Adapter Memory Map

Before we go on to a detailed explanation of the Transmit and Receive Data Flows, here is a snapshot of how the packet and control memory of CHARM is organized by the device driver. The various data structures mentioned in the previous sections can be located in figures 3.2 and 3.3.

3.7.1 Control Memory Map

Figure 3.2 shows how the control memory is organized by the device driver when it is installed.

3.7.2 Packet Memory Map

The device driver currently uses 6 MB of packet store. The memory map is as shown in figure 3.2. Based on network traffic characteristics, a more aggressive memory usage could be implemented. For example, the four real-buffer sizes can be altered to suit particular requirements. The order in which real buffers of various sizes are combined to form virtual buffers can also be altered. From the figure 3.3, it is clear that the device driver currently does not utilize the full potential of having five different real-buffer sizes; it only uses three sizes including the implicit buffers.

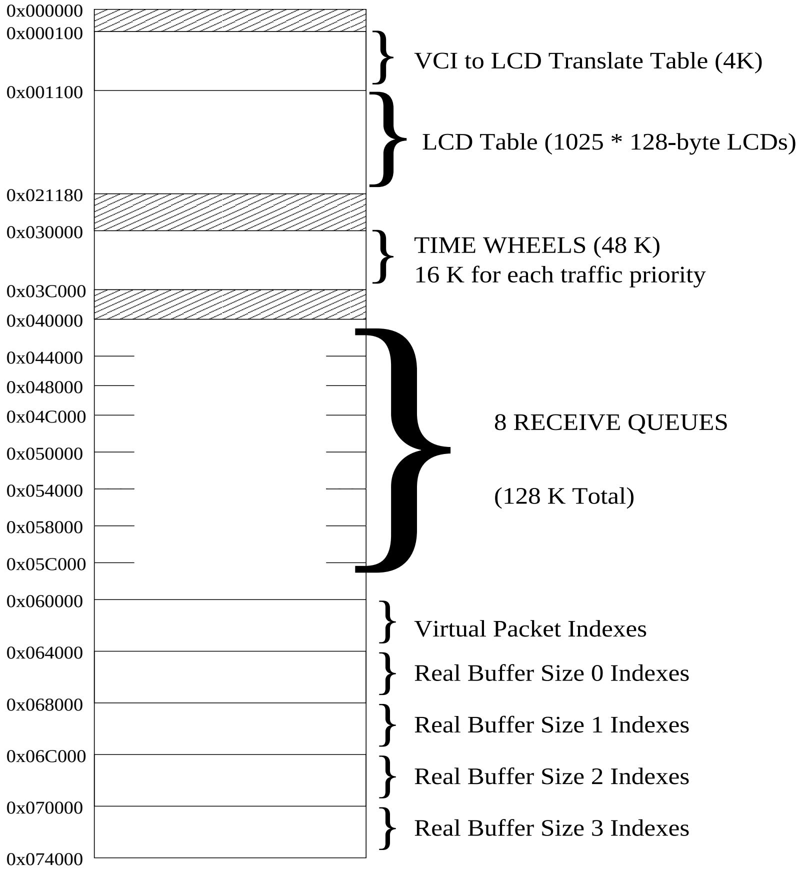

Figure 3.2 Control Memory Map

3.8 Data Flows

This section describes the data and control flow to and through CHARM. As mentioned before, in order for cell traffic to flow through CHARM, the cells require that Logical Channels be allocated.

3.8.1 Transmit Path

A typical transmit operation begins with the device driver requesting a buffer from POOLS and filling it with data via slave DMA, master DMA, or processor writes. The current version of the device driver uses processor writes. If virtual buffers are being used, the data write operation can fail due to lack of physical buffers. In the event of a failure, the header of the packet is updated to indicate the failure. The

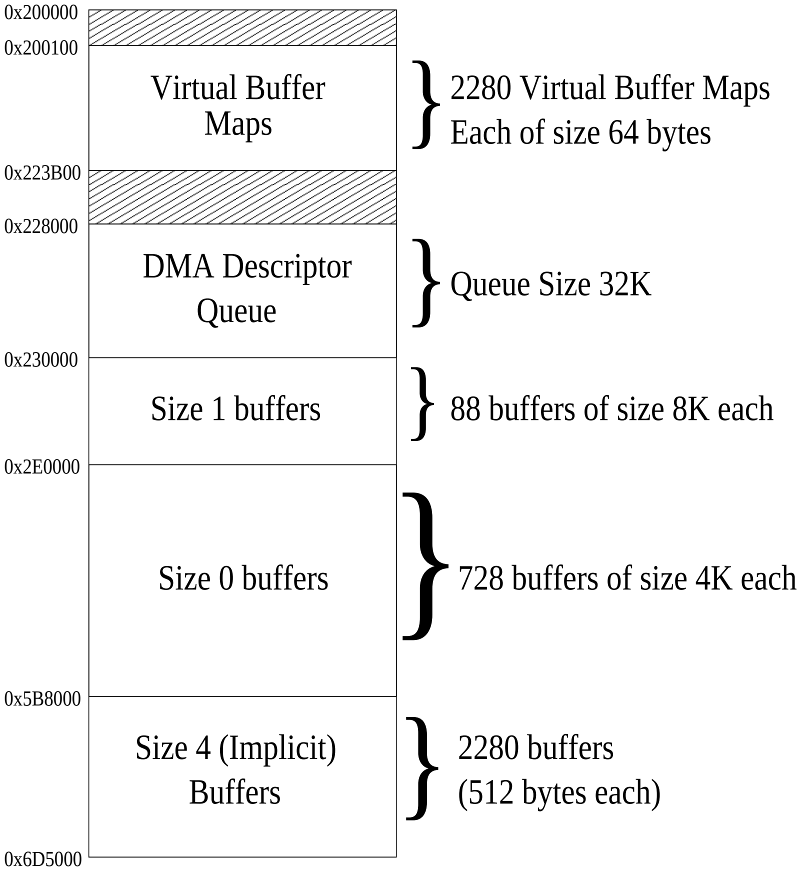

Figure 3.3 Packet Memory Map

---

device driver can audit the header after the buffer has been completely transferred, and either take action to recover the data immediately, or allow CSKED to generate an event later in the transmit cycle for any buffers that have had a data write failure.

Before the data can be transmitted, the buffer header must be updated by the device driver to contain information required for correct transmission. Information such as data length, starting offset, and Logical Channel (LC) address are just a few of the fields that must be correctly reflected in the buffer header. For a complete list of the fields in the buffer header refer to Packet Header in appendix A.

In addition to the fields in the buffer header, the scheduling and segmentation sections of the Logical Channel Descriptor (LCD) such as peak rate, average rate, and

AAL type must also be set up correctly prior to transmission. For a complete list of the fields in the LCD, refer to Transmit Descriptor Data Structures in appendix B.

After the data has been transferred into packet storage and both the buffer header and the LCD structure have been correctly initialized, the device driver enqueues the buffer address to CSKED. When it receives a buffer, CSKED checks the buffer header (packet memory) to make sure that the data transfer operation that filled the buffer completed without error[5]. If it finds an error, CSKED posts an event to device driver and does nothing further with this buffer. If it does not find an error, CSKED fetches several fields from the LCD (control memory) indicated in the buffer header to determine the current state of that LCD. If the LCD is busy sending another buffer, the new buffer is queued to this LCD and will be processed when all previously enqueued buffers have been transmitted. If the LCD is not busy, CSKED updates the LCD based on several fields in the buffer header and queues the LCD to the next time slot on the time wheel (control memory).

When CSKED detects a previously enqueued LCD on the time wheel, several fields are retrieved from the LCD. Among other things, these fields are used by CSKED to determine where on the time wheel to reschedule this LCD. The LCD address is then provided to SEGBF for processing.

When CSKED provides an LCD address to SEGBF, the segmentation portion of the LCD is retrieved from control memory by SEGBF to determine both the current address at which to continue buffer segmentation and the type of cell to construct. Depending on the AAL type bits in the segmentation portion of the LCD, the cell is constructed in an internal array using data from the LCD as well as data fetched from packet memory. When the cell construction is complete, status is raised to LINKC indicating that a new cell is available for transmission.

[5] See section 3.4.1 for a description of a possible error

Transmit opportunities are repeatedly provided to SEGBF by CSKED at the desired rate until all the data in the buffer has been passed to LINKC via the cell buffer array. When SEGBF detects that no more data exists for a buffer, the LCD address is passed back to CSKED, indicating buffer completion. If no more buffers are queued, CSKED removes the LCD from the time wheel. If more buffers are queued, the LCD is updated and the segmentation process continues until all buffers on the LCD queue are used. A bit in the buffer header generates a transmit complete event when no buffers remain in the queue. This event is appropriately handled by the device driver.

3.8.2 Receive Path

As cells arrive, they pass from LINKC to REASM. REASM uses a portion of the ATM header to look up the LCD address for this cell. The LCD address is then passed to RAALL. RAALL reads the receive portion of the LCD, and then processes the cell based on the LCD information. For example, the LCD specifies what AAL to use and maintains the current reassembly state. Using the current reassembly state, the cell data is written to packet memory. While the data is written to packet memory, other functions such as CRC generation and verification are performed in parallel. If a packet is complete, all trailer verification is performed. If the packet is good, an event is placed on a receive queue in the RXQUE entity. At this point, the device driver can dequeue the packet event from RXQUE using the dequeue operation. It can then examine headers, transfer the data into user space, and perform TCP checksums. When these actions are complete, the device driver must return the buffer to CHARM by performing a POOLS free-buffer operation.

Chapter 4

Standalone Driver Initialization/Shutdown

When the driver is installed using */sbin/insmod charm.o*, the *init_module* function gets called as we noted in chapter 2. This function works as follows:

It first checks whether the BIOS is PCI-aware, by calling the *pcibios_present* function:

```
if (!pcibios_present()) {
    printk("IBM CHARM: No PCI Bios!\n");
    return(-1);
}
```

Then, it calls *pci_find_device* using the appropriate `vendor` and `deviceID`.

```
pci_dev = NULL;
while (pci_dev = pci_find_device(PCI_VENDOR_ID_IBM,
                                      PCI_DEVICE_ID_ATM_CHARM, pci_dev)) {
   ibm_dev = pci_dev;
   found++;
}
if (found != 1) {
   printk("IBM CHARM: Found %d adapters \n" , found);
   printk("IBM CHARM: Exactly ONE adapter is required \n");
   return(-1);
}
```

After getting hold of the `pci_dev` structure, it saves the IRQ and address information. Using this address information, it calls *ioremap* to map the physical register and memory addresses on the board to the processor's address space.

```
/* ioremap the PCI registers so we can access them */
charm->vregbase = (unsigned int *)ioremap((charm->pci_membase &
                                  PCI_BASE_ADDRESS_MEM_MASK), 1024 * 64);
/* ioremap the memory on the board.
charm->vmembase = (unsigned int *)ioremap((charm->pci_membase2 &
                                  PCI_BASE_ADDRESS_MEM_MASK), 1024 * 64);
```

After successful initialization of the hardware, *init_module* calls *register_chrdev* using the appropriate major number and a `file_operations` structure, to register the device as a character device.

```
struct file_operations charm_fops = {
   NULL,           /* lseek              */
   NULL,           /* read               */
   NULL,           /* write              */
   NULL,           /* readdir            */
   NULL,           /* select             */
   charm_ioctl,    /* ioctl              */
   NULL,           /* mmap               */
   charm_open,     /* open               */
   charm_release,  /* release            */
   NULL,           /* fsync              */
   NULL,           /* fasync             */
   NULL,           /* check_media_change */
   NULL            /* revalidate         */
};
    . . .
if ((rc = register_chrdev(CHARM155_MAJOR, "charm155", &charm_fops)) < 0) {
   printk("IBM CHARM: Error: Cannot register as a char device!\n");
   return rc;
}
```

In the function-table above, only three entries are used. We are most interested in the `ioctl` aspect of the character device interface. The other two functions are called when the device is opened or closed. In the current version of the driver, the `open` and `release` functions do nothing.

To install the interrupt handler, *request_irq* function is used as follows:

```
rc = request_irq(charm->pci_irq, charm_irq,
                           SA_INTERRUPT | SA_SHIRQ, "charm", charm);
if (rc) {
   printk("IBM CHARM: Can't allocate IRQ %d. Code was %d \n",
                                          charm->pci_irq, rc);
   return -EIO;
}
```

When the user uninstalls the driver using the command */sbin/rmmod charm*, the *cleanup_module* function is called which deallocates all the resources allocated by *init_module*.

cleanup_module calls *free_irq* to release the interrupt line:

```
/* free the interrupt line */
free_irq(charm->pci_irq, charm);
```

It then calls *unregister_chrdev* to unregister the character device:

```
/* unregister our device */
if (unregister_chrdev(CHARM155_MAJOR, "charm155")<0)
{
   printk("IBM CHARM: Error: Cannot unregister char device!");
}
```

Now it has to make sure that the page tables are updated as the *ioremap*ed PCI memory is no longer accessible. This it does using *iounmap*:

```
/* unmap the ioremaped memory */

/* register address space */
iounmap((void*)(charm->vregbase));
/* PCI memory space */
iounmap((void*)(charm->vmembase));
```

Chapter 5
Initialization of CHARM

Chapter 3 gave a detailed explanation of how the various hardware entities in CHARM work together to provide a wide array of features for the device driver to use. This chapter briefly describes the function of each hardware entity and explains how it needs to be initialized by the device driver for the proper functioning of the device.

5.1 Control Processor Bus Interface Entities

5.1.1 PCINT - The IOP Bus Specific Interface Controller

This entity provides PCI specific interfacing between the external connection and the internal entities. The various configuration registers such as the IRQ line and the base addresses are setup by the BIOS upon system start-up. These are further modified by the Linux kernel as it boots up and finds any inconsistencies. None of the PCINT registers need to be altered while initializing the chip at module load time.

5.1.2 INTST - Interrupt Vector and Status/Control

This entity contains the masking registers that will choose which interrupt/status source can generate an interrupt. It also has high-level controls like Master chip enables and LCD (Logical Channel Descriptor) size selector. It also features a general purpose timer.

Initialization:

- Clear all CPB (Control Processor Bus) Status bits,
- Enable all CPB status bits,
- Enable all CPB errors to halt the chip,

- Clear the control register,
- Set the LCD size to 128 bytes,
- Enable the master chip enables for receive and transmit.

5.1.3 GPDMA - General Purpose DMA

This provides DMA control between System memory and CHARM-lite packet memory. There are two ways to initiate DMA transfers. The first is by directly writing the source address, destination address and transfer count and flag registers. The second is by writing descriptors in system memory that directly map to the three registers and issuing an Enqueue DMA Descriptor primitive. Although the current version of the driver configures CHARM memory to reserve space for DMA descriptors, at this time, it doesn't use DMA to transfer data.

Initialization:

- Clear all status bits,
- Enable DMA transfers to be initiated by Descriptors,
- Enable TX and RX DMA,
- Enable logging of error events in case of DMA errors.
- Limit the DMA maximum burst time to 64 cycles,
- Configure the DMA Descriptor Queue Base Address and Queue Size.

5.2 Memory Controlling Entities

5.2.1 VIMEM - ATM Virtual Buffer Logic

This entity manages the Virtual Address Space on the Adapter. Packet sizes up to 64K are supported by forming virtual buffers from several (up to 16) real buffers of (up to 5) different sizes. It keeps track of the real buffers that make up a virtual buffer with the help of buffer maps (similar to page tables).

Initialization:

- Set the start of virtual memory, control memory and packet memory,
- Set the virtual memory size,
- Set the virtual buffer size to 128K (64K + overhead),
- Set the base address of the Virtual Buffer Maps,
- Set the Maximum Buffer Size,
- Configure the 5 real buffer base addresses,
- Prevent a VM Recoverable Page Fault from causing an interrupt or locking the memory,
- Configure the Virtual Buffer Segment Size register specifying the sizes of the 16 real buffers that can make up a virtual buffer,

5.2.2 BCACH - The Bus DRAM Cache Controller

This entity provides the caching function for data transfers on the Control Processor Bus. The array is organized as four logically separate 32-byte cache lines, any of which can be used for processor accesses or master/slave DMA accesses.

Initialization:

- Clear the control register
- Enable packet memory caching with prefill (for efficiency)
- Clear the status register
- Disable cache line collisions from causing interrupts or locking the cache

5.2.3 COMET - The DRAM Controller

This generates the signals necessary for DRAM accesses. These include RAS, CAS, Write enable and Output enable.

Initialization:

This entity is setup by the EPROM when powered up, so not much software setup is required.

- Clear the Status register,
- Enable all interrupts from this entity except single-bit ECC

5.2.4 POOLS - Buffer Pool Management

This is the memory manager for CHARM-lite. Memory buffers are checked out and checked in via two primitives supported by POOLS: the get pointer primitive and the free pointer primitive. POOLS also contains mechanisms to control resource utilization and supports a real memory mode and a virtual memory mode.

Initialization:

- For each of the 5 queues of buffers, setup the lower and upper bound addresses, head and tail pointers, queue lengths, common pools counts and the thresholds.
- Setup all 16 pools to have a total threshold of the total number of buffers,
- Enter initialization mode, leave diagnostic mode and enable virtual memory,
- Use the free-pointer primitive to free all the buffers.
- Leave initialization mode
- Clear status register
- Enable interrupt due to failure of a get-pointer primitive

5.3 Transmit Data Path Entities

5.3.1 CSKED - Transmit Scheduling

The transmit cell scheduler entity is responsible for receiving a packet from the processor, determining when cells from the packets need to be transmitted, passing this information to the segmentation buffer entity, and handling congestion control.

Initialization:

- Configure the base address of the timing data
- Enable traffic of all three priorities
- Set up to queue the LCD address upon transmit complete.

5.3.2 SEGBF - ATM Transmit Buffer Segmentation

This entity accepts frames from CSKED or software, and then generates ATM cells to send out over the external physical interface.

Initialization:

- Clear status and control registers
- Set up to interrupt if the total cells transmitted counters overflow, an invalid VCD is detected or cell generation for a VCD is complete.

5.4 Receive Data Path Entities

5.4.1 REASM - Cell Re-Assembly

This entity collects bytes from the physical interface into cells, while discarding null cells and idle symbols. It also checks and corrects HEC errors. It looks up the control memory for the proper Logical Channel Descriptor address and then passes it to the Receive AAL processing entity.

Initialization:

- Set the number of bits to use for the VPI and VCI,
- Disable the VCI to LCD translate cache,
- Setup the base addresses for the LCD table and the VCI to LCD translate Table,
- Clear the status register,
- Set up to interrupt when a HEC error occurs or the total user cell counter overflows or an overrun is detected.

5.4.2 RAALL - Receive AAL Processing

This implements all the receive functions of the ATM Adaptation Layers (AAL) viz. Protocol Verification, Cell Reassembly into Packets, Cell CRC Verification, Packet Reassembly Timeouts and Errors. It has a 4-stage FIFO buffer where it stores incoming cells for processing. This buffer can be read by the device driver when RAALL is configured to work in diagnostic mode. This feature is exploited by a debugging utility, *celldump* mentioned in section 6.3.5.

Initialization:

- Configure the lower and upper bounds of the LCD Table,
- Set up the reassembly timeout value,
- Set up the Reassembly Timeout and Pre-Scaler registers,
- Set up the receive Queue to which all Non-User Data is routed to,
- Specify the receive Buffer offset for Non-User Data Cells,
- Clear Status register
- Enable Host Data Word in LCD/Packet Header; the 8th word of the rx LCD is used as host data and written in to the 5th word of the packet header.
- Set up the threshold number of events dropped after which a status bit is set; make it zero.

5.4.3 RXQUE - Receive Queues

This entity manages the receive packet queues for software by providing an easy to use primitive interface. A detailed description of receive queues in CHARM is given in section 3.5.

Initialization:

- Clear the control and all Interrupt enable registers and enter diagnostic mode
- For each of the 8 queues, set up the lower and upper bounds, queue lengths and thresholds
- Leave diagnostic mode

- Assign all inbound queues
- Enable interrupts for any events dropped or queues that are full.
- Set the latency timer to approximately 30 microseconds
- Set the events dropped threshold

5.5 Physical Level Interfaces

5.5.1 LINKC - The PHY interface

LINKC provides the interface between the CHARM-lite and the ATM PHY device. The PHY device on the board we used is PMC SIERRA PM5346 SUNI LITE FOR SONET STS-3c 155.52 Mbps.

Initialization:

- Set the PHY type to SUNI-lite, with 8-bit data bus and Odd parity
- configure to use the internal 60ns clock
- set loopback mode
- Raise reset to the PHY and after some delay, lower it.

5.5.2 NPBUS - Nodal Processor Bus Interface

This entity controls the signals of the Nodal Processor Bus. The registers of the PHY are accessible through this entity from the address space of the CHARM-lite chip. Also, the operation of the front end is affected by the NPBUS Status register.

5.6 Hardware Protocol Assist Entities

5.6.1 CHKSM - Checksum and DRAM Test Support

The CHKSM entity is capable of initializing and/or testing packet and control memory and can perform 2's complement, 16-bit sum with end-around-carry. While initializing the chip, this entity is used to set up the packet and control memory.

5.7 Base Chip Function Entities

5.7.1 CRSET - Reset and Power-On Logic

This entity will perform built-in-self-test, flush and unit reset operation. Chip software resets can be controlled by this entity. This entity doesn't require any initialization by the device driver.

Chapter 6
User-Level Control/Testing Support

6.1 IOCTL Access

The character device interface described in section 2.2 provides a convenient mechanism for `ioctl` access. The *ioctl* function call in the user space corresponds to the following prototype:

```
int ioctl(int fd, int cmd, ...);
```

This prototype stands out in the list of Unix system calls because of the dots, which usually represent a variable number of arguments. In a real system, however, a system call can't actually have a variable number of arguments. Therefore, the third argument of ioctl is actually a single optional argument, and the dots are simply there to prevent type checking during compilation. The actual nature of the third argument depends on the specific control command being issued (the second argument). Some commands take no arguments, some take an integer value, and some take a pointer to other data. Using a pointer is the way to pass arbitrary data to the *ioctl* call; the device will be able to retrieve any amount of data from the user space. This feature is fully exploited by the tools that we developed to control and test the driver.

Arguments to the system call are passed to the driver method according to the method declaration:

```
int (ioctl*) (struct inode *inode, struct file *filp,
              unsigned int cmd, unsigned long arg);
```

The `inode` and `filp` pointers are the values corresponding to the file descriptor `fd` passed on by the application. The `cmd` argument is passed unchanged, and the optional `arg` argument is passed in the form of an `unsigned long`, regardless of whether it was passed as an integer or a pointer. If the invoking program doesn't pass a third argument, the `arg` value received by the driver function will not be meaningful.

Like most *ioctl* implementations, our IOCTL function consists of a `switch` statement that selects the correct behavior according to the `cmd` argument. Different commands have different numeric values, which are usually given symbolic names to simplify coding.

The following sub-sections show the IOCTL commands implemented in the device driver. Code snippets are included to better illustrate what they do:

6.1.1 AIO_RDCREG

This is used to read a register on the board. The third argument is a pointer to a user buffer. The first word in the buffer is the register address. Upon completion, the second word will be the value read from that register. It uses the `RD_REGL` macro described in section 2.4.

```
case AIO_RDCREG:
   get_user(reg_id, word);
   /* read the specific register and transfer its value */
   val = RD_REGL(reg_id);
   put_user(val, word + 1);
   break;
```

6.1.2 AIO_WTCREG

This is used to write to a register on the board. The third argument is a pointer to a user buffer. The first word in the buffer is the register address. And the second word is the value to be written to the register. It uses the `WT_REGL` macro described in section 2.4.

```
case AIO_WTCREG:
   get_user(reg_id,  word);
   get_user(reg_val, word + 1);
   /* write the data to the specified register */
   WT_REGL(reg_val, reg_id);
   break;
```

6.1.3 AIO_CREATELC

This is used to set up a logical channel descriptor in the control memory of the adapter. The third argument is a pointer to a user buffer. The first word in the buffer is the VCI. Upon completion, the second word will be the logical channel descriptor handle.

```
case AIO_CREATELC:
   get_user(id, word);               /* Get the VCI */
   val = charm_setup_lcd(0, id);     /* set up the LCD */
   put_user(val, word+1);
   break;
```

charm_setup_lcd is a module function that takes the VPI as the first parameter and the VCI as the second parameter. Currently the driver only supports a VPI value of zero. This command uses the `WRITE_MEM` macro defined in `"charm155.h"`.

```
/* macro for accessing memory on the board */

#define SETWINDOW(addr) \
{ \
     WT_REGL((bit32)(addr), PCINT_WINDOW_BASE3); \
     udelay(200); /* short delay to allow things to settle */ \
} \

#define WRITE_MEM(board, sys, len) \
{ \
   SETWINDOW((board)); \
   memcpy_toio((bit32)charm155.vmembase + \
                          (bit32)((board)&0xffff),(sys),(len));\
}
```

6.1.4 AIO_FREELC

This is used to shut down a logical channel descriptor in the control memory of CHARM. The third argument is a pointer to a user buffer. The first word in the buffer is the VCI.

```
case AIO_FREELC:
   get_user(id, word);                 /* Get the VCI */
   val = charm_shutdown_lcd(0, id);    /* Shut down the LCD */
   put_user(val, word+1);
   break;
```

charm_shutdown_lcd is a module function that takes the VPI as the first parameter and the VCI as the second parameter. Currently the driver only supports a VPI value of zero. It invalidates the appropriate LCD data structure in control memory and also updates the VPI/VCI to LCD translation table.

6.1.5 AIO_SENDDATA

This is used to send user data through the device. The third argument is a pointer to a user buffer. The first word in the buffer is the VCI number, the second word is the address of the user data and the third word is the length of the user data.

```
get_user(id, word);
get_user(loc, word + 1);
get_user(len, word + 2);
rc = verify_area(VERIFY_WRITE, (char *)loc, len);
if (rc)
{
   printk("Bad code from verify \n");
   return(rc);
}
rc = charm_send(id, (char*)loc, len);
if (rc < 0)
   printk("Bad return code from charm_send: %d\n",rc);
break;
```

6.1.6 AIO_RECVDATA

This is used to get the received packets from the driver to the user-space. The third argument is a pointer to a user buffer. The first word in the buffer is the VCI number, the second word is the address of the user buffer where the data is to be written and the third word is the buffer length.

```
case AIO_RECVDATA:
   get_user(id, word);
   get_user(loc, word + 1);
   get_user(len, word + 2);
   rc = verify_area(VERIFY_WRITE, (char *)loc, len);
   if (rc)
      return(rc);
   printk("RECVDATA queue = %d \n", id);
```

```
    rc = charm_recv(id, loc, &len);
    printk("charm_recv() returned %d\n", rc);
    return(len);
```

6.1.7 AIO_DUMP_STATS

This is a debug support feature. The third `ioctl` argument is not used. It calls the module function *DumpStatus* which prints out the values of all the status registers of the various hardware entities in the device (to the `/var/log/messages` file).

```
  case AIO_DUMP_STATS:
    printk("Dumping Status Registers...\n");
    DumpStatus();
    break;
```

6.1.8 AIO_MEM_DUMP

This is also a debug support feature. The third argument is a pointer to a user buffer. The first word in the buffer is the starting address (virtual) of memory that needs to be read from CHARM. The second word is the length of the memory that is being read. Upon completion, the user buffer is filled with the values read from the board memory.

```
  case AIO_MEM_DUMP:
    get_user(board, word);
    get_user(len, word + 1);
    DumpMemory(word, board, len);
    break;
```

This command uses the READ_MEM macro defined in `"charm155.h"`. The `SET_WINDOW` macro is shown in sub-section 6.1.3.

```
/* macro for accessing memory on the board */

#define READ_MEM(board, sys, len) \
{ \
  SETWINDOW((board)); \
  memcpy_fromio((sys),(bit32)(charm155.vmembase)+ \
                                        (bit32)((board)&0xffff),(len)); \
}
```

6.2 Transmitting/Receiving Utilities

The following programs allow the user to send and receive data from the board. They use the IOCTL facilities outlined above.

6.2.1 charmsend

This program reads `stdin` and sends out the data on a particular logical channel. The first parameter is the VCI number and the optional second parameter is the frame size.

6.2.2 charmrecv

This program reads the packets available on a particular logical channel and dumps them to `stdout`. It takes the VCI number as a parameter.

6.2.3 charmgen

This program is used to transmit large amounts of traffic through the adapter. The first parameter is the VCI number, the optional second parameter is the `blocksize` and the optional third parameter is the total number of bytes to transmit (`bytecount`). It continuously transmits frames of size `blocksize` until `bytecount` bytes have been transmitted.

6.2.4 charmgobl

This is the corresponding frame gobbler. It consumes the traffic generated by *charmgen*. It takes the VCI number as a parameter.

6.3 Testing/Debugging Utilities

6.3.1 readreg

This program reads the value of any register on the board. It takes the register address as a parameter.

6.3.2 writereg

This program writes to any register on the device. The first parameter is the register address. The second parameter is the value to be written.

6.3.3 dumpstats

This program causes the kernel to dump the values of all the status registers of all the hardware entities on the board into the file `/var/log/messages`.

6.3.4 memdump

This program dumps out the contents of the memory on the board. The first parameter is the physical address of the memory on the board. The optional second parameter is the number of words to dump (default being 1024).

6.3.5 celldump

This dumps out the contents of the 4-stage FIFO cell buffer in the RAALL entity (See section 5.4.2). For this, the RAALL entity has to be configured to work in diagnostic mode. It takes the cell stage # (1-4) as a parameter.

6.4 Device Driver Performance

The device driver currently achieves a transmit rate of 61Mbps. We tested the driver by transmitting varying amounts of data, ranging from 5 MB to several GB. The resulting throughput values remained consistent. Figure 6.1 shows the results when 5 MB of data was trasnmitted with different packet sizes. From the figure, we can see that throughput is tied to the frame size and is increasing as we increase the frame size. This is understandable, as a higher frame size leads to fewer kernel calls for the same amount of data; which in turn implies lesser interrupt overhead for the kernel. This might lead to the question: why do we limit the frame size to 4K? We tried implementing support for larger frames in the device driver and had some success in raising the transmit rate to 143 Mbps, but not without encountering problems with the device. At certain frame sizes, the board would report internal page faults and lock up CHARM memory. An investigation into why this happens would certainly be worth the effort considering the immense gain in throughput.

In an attempt to improve performance, we tried to tinker with the fields in the LCD data structure[1]. In the LCD data structure there is a `max_resolution` bit which can be used to specify the peak and average rates in finer resolution than "one cell time". Modifying this bit and the peak and average rates resulted in a very small increase in the rate (by about 4 Mbps).

In figure 6.1, the **TX Only** column shows the throughput when reception of packets is disabled in the driver. The **TX+RX** column shows the throughput values when the driver is transmitting and receiving simultaneously. The **TX Only** values are more relevant because in a typical scenario, one machine is transmitting while another machine is receiving packets.

The performance of the device driver seems to be CPU limited as can be seen by the output of the `charmgen` command. Observe the user time and system time shown below:

```
[ahmed@glint2$] charmgen 22 4000 50000000

Sending to vci 22
Frame size 4000 bytes
User time    =      0.01
System time  =      6.50
Elapsed time =      6.51
MBits / sec  =    61.430
Frames sent  =     12500
Frames/usec  =     0.002
```

Clearly, a large amount of time is spent in the kernel. The reason for this is that DMA is currently not supported by the device driver. For every frame, the kernel has

[1]See appendix B for a description of the various fields in a Logical Channel Descriptor data structure.

Packet Size (bytes)	TX Only (Mbps)	TX+RX (Mbps)
200	3.818	1.893
400	7.543	3.674
600	11.180	5.367
800	14.690	6.947
1000	18.177	8.672
1200	21.548	18.177
1400	24.840	21.548
1600	27.996	12.736
1800	31.126	13.829
2000	34.251	15.533
2200	37.250	16.135
2400	40.162	17.043
2600	42.937	18.144
2800	45.745	20.206
3000	48.579	20.951
3200	51.111	21.331
3400	53.795	21.981
3600	56.460	24.668
3800	58.914	25.044
4000	**61.334**	25.619

Figure 6.1 Device Driver Performance Results

to spend time copying packet data to hardware buffers when transmitting and should copy data back into system memory while receiving. We believe that implementing DMA support would be an important enhancement for the device driver and will result in a marked decrease in system time spent while transmitting and receiving data.

Chapter 7
Future Research and Concluding Remarks

The paper discussed the implementation of the device driver that is being developed for the IBM CHARM-lite based ATM NIC. The driver is reasonably stable in its standalone form. The transmit rate is around 61 Mbps. A user level library was developed comprising functions that will create a logical channel, transmit and receive data and close a logical channel. There is a lot more to be done in terms of making efficient use of the facilities offered by the hardware.

- A more efficient virtual memory management scheme can be implemented by making use of the hardware capabilities to dynamically change the way real buffers are allocated to a virtual buffer by keeping track of changes in network traffic.

- We hope that using DMA to transfer data in and out of the board will significantly enhance its performance.

- The hardware is capable of supporting other AALs, but the driver currently only supports AAL5. Although we don't see the need for incorporating support for older adaptation layers, it might be useful to add support for AAL6 and AAL7.

- An interesting area of study might be the MPEG support offered by the hardware and how it can be used in the driver.

- The driver has to be integrated with the Linux ATM stack. There is a more or less standard way to do this.

- Most importantly, testing has been quite limited. The driver has to be tested with a real network. Currently it configures the hardware to work in an internal loopback mode, which means there is no interaction with the ATM switch. Testing on other platforms is also required.

Building a device driver for a complex ATM NIC has been a very educating experience and has helped us extend our knowledge of Linux kernel programming.

References

1. Westall, J.M., Geist, R.M., *Bringing the High End to the Low End: High Performance Device Drivers for Linux PC*. In *Proceedings - 36th Annual ACM Southeast Conference.* ACM, 1998.

2. Westall, J.M., Geist, R.M., Flower, A.L., *ATM Device Driver Development in Linux*, Clemson University, 1998.

3. *ATM Functional Specification: The CHARM-lite Chip 1.5*, IBM Corporation, August 1998.

4. Rubini, A. (1998), *Linux Device Drivers*, Sebastopol, CA: O'reilly and Associates.

5. *ATM Resource Manager Version 2.1 DataBook*, IBM Corporation, 1998

6. Documentation of the Linux kernel version 2.1.117 available under the source tree `/usr/src/linux/Documentation`.

APPENDIX A
Packet Header Data Structures

Each packet buffer is comprised of two parts. The first part is the control information used by CHARM. The second part is used to hold actual packet data. The following code shows the structure of the tx/rx packet headers:

```
struct TxPacketHeader {
     bit32 next_buffer;
     bit8  AAL5_user_byte1;
     bit8  buffer_offset;
     bit16 buffer_length;
     bit26 lcd_address;
     bit2  reserved;
     bit1  generate_CRC10;
     bit1  free_on_xmit;
     bit1  queue_on_xmit;
     bit1  cell_loss_priority;
     bit32 reserved;
     bit16 AAL5_user_byte2;
}

struct RxPacketHeader {
     bit16 rx_label;
     bit14 reserved;
     bit1  error_status;
     bit1  done_status;
     bit8  AAL5_user_byte;
     bit8  buffer_offset;
     bit16 buffer_length;
     bit26 lcd_address;
     bit5  reserved;
     bit1  cell_loss_priority;
     bit32 rx_atm_header;
     bit24 host_data;
     bit8  AAL5_user_byte2;
     bit32 cut_thru_sys_buff_desc_addr;
}
```

The following is a description of the above fields:

- **next_buffer** This field is used by hardware to chain buffers together on queues. It contains the address of the next buffer if one exists.

- **AAL5_user_byte1** This field contains the value to be sent in the user byte in the last cell of an AAL5 packet if INTST is configured for one user byte.

- **generate_CRC10** If this bit is set CRC 10 will be generated over the cell(s) in this packet.

- **free_on_xmit** If this bit is set the buffer will be freed after the transmission completes.
- **queue_on_xmit** If this bit is set the buffer will be queued on the transmit complete queue after the transmission completes.
- **cell_loss_priority** This bit is used on both transmit and receive:
- **tx**: If this bit is set the cell loss priority bit in the ATM cell header will be set for each cell in this packet.
- **rx**: This bit contains the or-ed cell loss priority bits across all the cells that comprised this packet if this lcd is using AAL5.
- **buffer_offset** this field contains the offset into the buffer where the data starts.
- **buffer_length** This field contains the length of the packet.
- **lcd_address** This is the address of the LCD that this packet was received on.
- **rx_atm_header** On reception, the 4-byte ATM header (no HEC) is copied from the first and last cell into this area.
- **AAL5_user_byte2 (tx)** This field contains the value to be sent in the user bytes in the last cell of an AAL5 packet if INTST is configured for two user bytes.
- **AAL5_user_byte2 (rx)** This field contains the 2nd AAL5 user byte in the last cell of an AAL5 packet if INTST is configured for two user bytes.
- **rx_label** This field is written with "RA" in ASCII (0x5421) to signal that this buffer was used by RAALL.
- **reserved** This field is zeroed.
- **error_status** This bit is written when the packet is completed, if an error condition occurs.
- **done_status** This bit is written when the packet is completed.
- **host_data** If host data is enabled in RAALL, then the 24 bits of host data is read from the LCD and written to this area for each packet.
- **cut_thru_sys_buff_desc_addr** This field is only used in one of the cut-thru modes.

APPENDIX B

Logical Channel Data Structure

The LCD data structure has a transmit and a receive portion. The following figure shows the general layout:

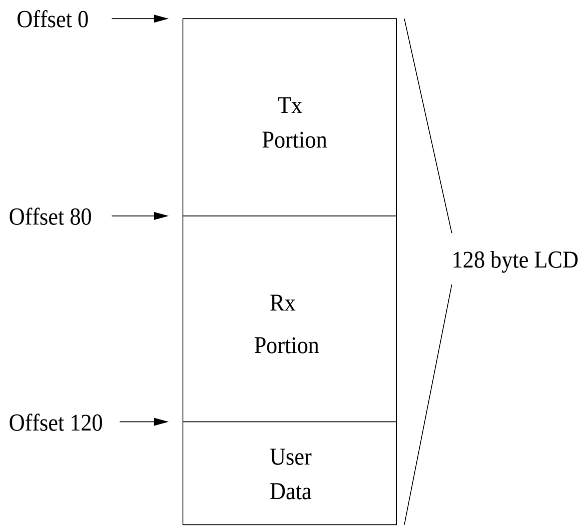

Figure B-1 General LCD Layout

Transmit LC Descriptor Data Structure

The following code shows the layout of the transmit LCD. When initializing an LCD, any locations that are not written to a specific value, should be initialized to zeros. The linkage between the various transmit data structures is shown in figure B-2.

```
struct CharmTxLcd {
        bit32   next_lcd;
        bit16   peak_interval;
        bit16   average_interval;

        bit32   timestamp;
        bit4    no_congestion_counter;
        bit1    flush_on_congestion;
        bit3    cong_recovery_value;
        bit1    remove_lcd;
        bit1    lc_on_timewheel;
        bit1    VP_fairness;
        bit2    alter_sched;
```

```
bit2    transmit_priority;
bit1    max_resolution;
bit3    max_burst_mult;
bit10   max_burst_value;

bit26   head_packet_pointer;
bit1    free_on_xmit;
bit1    queue_on_xmit;
bit4    reserved;
bit26   tail_packet_pointer;
bit6    reserved;

bit16   transmit_length;
bit8    buffer_offset;
bit1    enable_blocking;
bit1    enable_statistics;
bit1    CRC_sent;
bit1    generate_CRC10;
bit1    OAM_CLP_source;
bit3    AAL_type;
bit32   ATM_header;

bit32   segmentation_pointer;
bit32   current_CRC;
bit16   previous_PCR_bits15_0;  // MPEG Support
bit11   PID_bits_10_0;          // MPEG Support
bit1    PCR_delta_valid;        // MPEG Support
bit36   PCR_delta;              // MPEG Support

bit16   BIP-16;
bit8    Monitor_sequence_number;// MPEG Support
bit1    OAM_cell_transmitted;
bit2    previous_PCR_bits17_16; // MPEG Support
bit1    OAM_PTI_bit0;
bit1    previous_packet_contained_PCR;  // MPEG Support
bit1    OAM_CLP_value;
bit2    OAM_block_size;
bit8    Current_Blocking_Count (4 bytes)
bit8    Blocking_size (4 bytes x'47' for MPEG-2)
bit1    PID_field_valid;        // MPEG Support
bit2    PID_bits_12_11;         // MPEG Support
bit5    Current_transport_stream_packet;
bit1    Check_PCR;              // MPEG Support
bit1    PCR_present;            // MPEG Support
bit1    PID_matches;            // MPEG Support
bit5    Packets_per_AAL5_frame;
```

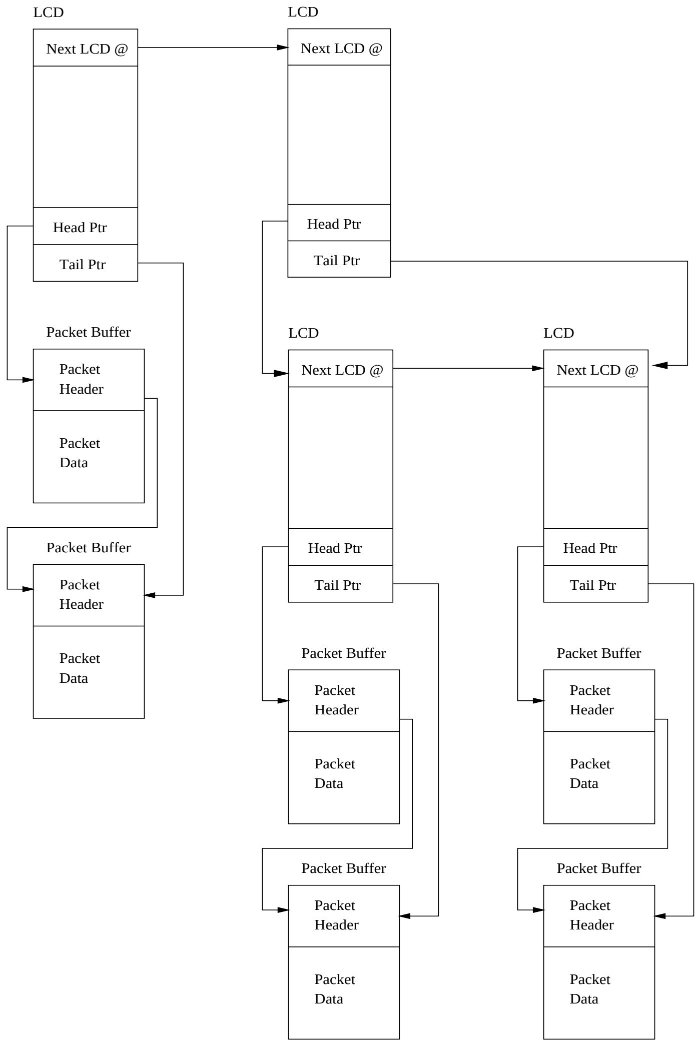

Figure B-2 Tx Data Structure Linkage

```
        bit32   total_user_cells;
        bit32   total_user_cells_CLP0;
}
```

The following is a description of the fields above:

- **next_lcd** This field is used by CHARM to chain LCDs together on queues. It contains the address of the next LCD if one exists.

- **peak_interval** This field contains the minimum spacing allowed between consecutive cells on this logical channel. The spacing is expressed in cell times. A channel that can transmit every cell time would have a value of 1 for this field.

- **average_interval** This field contains the minimum average spacing allowed between cells transmitted on this channel. It is the inverse of the Sustainable Cell Rate. The value for this field is expressed in cell times.
- **timestamp** This field contains a timestamp used by the hardware to determine if transmit opportunity credits exist and if the Burst Tolerance has been exceeded. It should be initialized at connection setup time to the value of the current timeslot counter.
- **no_congestion_counter** This field is used by CHARM to recover from congestion.
- **flush_on_congestion** If this bit is set and congestion is detected, the packet currently being sent on this channel will be flushed.
- **cong_recovery_value** This field is used to specify if congestion control is to be used and the rate of recovery from congestion control.
- **cong_received_count** This field is used to determine how many times congestion has been received and not been recovered on this channel. This field should be initialized to zero.
- **remove_lcd** If this bit is set, the LCD will be removed from the timewheel at the next transmission opportunity. It should be initialized to zero.
- **lc_on_timewheel** This field indicates if the LCD is currently queued on the timewheel. It should be initialized to zero.
- **VP_fairness** This bit is used to specify that if this VC is on a Virtual Path, the bandwidth is to be shared on a cell basis rather than on a packet basis.
- **alter_sched** These encoded bits will alter the scheduling of cells on a VC.
- **transmit_priority** This field specifies the priority of transmission on this channel. 0=high, 1=medium, 2=low.
- **max_resolution** If this bit is set, the lower 8 bits of the average interval and the peak interval contain a fractional component. This allows a finer resolution for scheduling. For example, for a peak interval of 1.5 cell times, the value written to the peak_interval field should be 0x0180.
- **max_burst_mult** The values in this field and the next field are used to limit the number of cells that can be transferred at the peak rate. The max_burst_value will be multiplied by 4 to the power of the value in this field to yield the maximum credit time. This time is expressed in cell times and represents the time it would take to acquire the maximum number of cell credits. This maximum credit time should equal the average interval times the maximum number of cells that can be transferred at the peak rate.

- **max_burst_value** The value in this field will be multiplied by 4 to the power of the value in the max_burst_mult field to yield the maximum credit time.
- **head_packet_pointer** This field is used to chain buffers to LCDs.
- **tail_packet_pointer** This field is used to chain buffers to LCDs.
- **transmit_length** This field contains the length of the currently transmitted packet.
- **free_on_xmit** This bit is set if the header of the currently transmitted packet has specified that the packet is to be freed after transmission. See appendix A for Packet Header fields.
- **queue_on_xmit** This bit is set if the header of the currently transmitted packet has specified that the packet is to be queued after transmission.
- **buffer_offset** This field contains the offset into the buffer that the transmit data starts.
- **enable_blocking** This bit when set will enable OAM blocking cells to be sent on the associated VC by the segmentation logic. Other fields in the LCD define the content and frequency of these frames. Setting this bit also forces statistics to be kept for the associated LCD regardless of the state of the enable_statistics bit.
- **enable_statistics** This bit when set will enable statistics keeping for the associated LCD. Another 8 bytes of the associated LCD will be used to maintain the counts of the total number of user cells and the total number of user cells with CLP [1] = 0 that have been sent over this VC.
- **CRC_sent** This bit when set, indicates that the segmentation logic has completed processing the current frame, and that no other frames have been queued by the cell scheduling logic. This field is initialized to zero by CHARM every time the scheduling logic queues a new frame for segmentation.
- **generate_CRC10** This bit is set if the header of the currently transmitted packet has specified that the cells in this packet should have CRC-10 generated.
- **AAL_type** This field specifies the AAL type to be used by this logical channel.
- **ATM_header** This field contains the first four bytes of the ATM header.
- **segmentation_pointer** This field will contain a pointer to the next data to be transmitted. This is initialized internally by CHARM.
- **current_CRC** This field will contain the CRC as it is being built.

[1] Cell Loss Priority

- **total_user_cells** This 32 bit field will contain a count of the total number of user cells that have been sent on this LCD. This field is zeroed when the channel is initialized. An event will be generated when this count overflows.
- **total_user_cells_CLP0** This 32 bit field will contain a count of the total number of user cells that have been sent on this LCD with CLP=0. This field is zeroed when the channel is initialized. An event will be generated when this count overflows.
- **BIP-16** This 16-bit field will contain the Bit Interleaved Parity accumulated over the last block of data. This is used internally by CHARM.
- **Monitor_sequence_number** This 8-bit field will contain the Monitor Sequence Number that is used to construct the hardware generated OAM cells. This is only used internally by CHARM.
- **OAM_cell_transmitted** This 1-bit field is set when an OAM cell has been sent; used internally by CHARM.
- **OAM_PTI_bit0** This 1 bit field is copied directly to the low bit of the Payload-type field in the ATM header as the segmentation logic is building an OAM cell; used internally by CHARM.
- **OAM_CLP_value** This 1-bit field is can be used to provide the value for the CLP bit in the hardware generated OAM cells.
- **OAM_block_size** This 2-bit field defines the number of user cells that are sent between OAM cells.
- **Current_transport_stream_packet** This 5-bit field contains the number of the current transport stream packet that is being segmented; used internally by CHARM.
- **Packets_per_AAL5_frame** This 5-bit field is initialized by the device driver to indicate how many packets should be concatenated into an AAL5 frame.

The other fields that haven't been described are related to MPEG support which is not currently supported by the device driver.

Receive LC Descriptor Data Structure

The receive data in the LCD structure is dependent on which AAL is being run. Since we are only concerned with AAL5, the following code shows the layout of the receive LCD for AAL5. The linkage between the various receive data structures is shown in figure B-3.

```
struct CharmAAL5RxLcd {
        bit32   packed_lcd_info;
        bit16   header_threshold;
```

```
        bit16   max_packet_length;
        bit32   curr_rx_buff_addr;
        bit32   current_crc;
        bit32   total_user_cells;
        bit32   total_CLP0_user_cells;
        bit16   oam_tuc;
        bit16   oam_bip16;
        bit32   reserved;
}
```

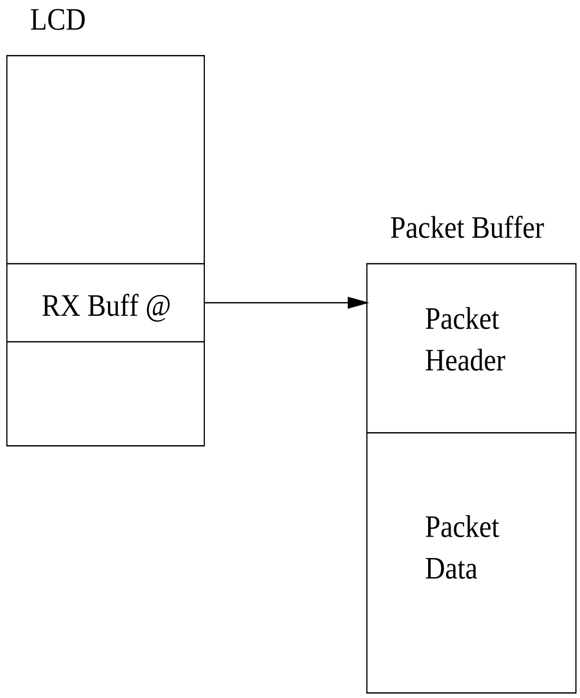

Figure B-3   Rx Data Structure Linkage (AAL5)

A brief description of the fields follows:

- **packed_lcd_info** The following code shows the layout of this 32-bit field:

```
struct packed_lcd_info {
   bit3  aal_type;        // which AAL is this LC running.
   bit3  lc_state;        // current rx state of LC
   bit4  proto_bits;      // bits affecting AAL5 behavior
   bit1  desc_state;      // used internally
   bit1  enable_status;   // if on, LCD status events are
                          //         surfaced to the rxq
```

```
    bit1  enable_stats;    // if on, LCD statistics are enabled
    bit1  enable_blocking; // if on, blocking is enabled
                           //         for this LCD
    bit1  RTO_TestSet;     // used internally for reassembly timeout
    bit1  RTO_Enable;      // reassembly timeout enable
    bit1  ored_clp;        // used internally to track ored-clp
                           //         value of packet
    bit3  rxq_num;         // rxq number that events
                           //         should be queued to
    bit4  rx_pool_id;      // pool id to get buffers from
    bit8  rx_offset;       // rx offset into packet buffer
}
```

- **header_threshold** Used in cut-thru support not currently implemented in the driver.
- **max_packet_length** Specifies the maximum amount of data that can be received per packet on this LC. If set to zero, the normal 64K-1 value for AAL5 will be used. This allows the maximum packet size to be set on an LC basis.
- **curr_rx_buff_addr** This field is used internally by CHARM, but is initialized to zero by the device driver. This field is used to track the current packet under reassembly.
- **current_crc** This field is used internally by CHARM to maintain the CRC residue as the current packet is reassembled. This field doesn't have to be initialized by the device driver.
- **total_user_cells** An LC statistic that counts the total cells received on this LC. For accurate counts, the device driver should initialize this to zero.
- **total_CLP0_user_cells** An LC statistic that counts the total cells with CLP=0 received on this LC. For accurate counts, the device driver should initialize this to zero.
- **oam_tuc** Total OAM blocking cells received on this LC.
- **oam_bip16** Used internally; OAM blocking 16-bit Bit-Interleaved-Parity. This is the calculated BIP over the current block of cells for this LC.

APPENDIX C
CHARM Register Space Map

The register addresses for the different entities in CHARM are shown in the figure below. The file `"charm155.h"` consists of symbolic names for the various registers used by the driver.

Address	**Entity**	**Elements Accessed**
XXXX 0000 - FF	PCINT	Registers
XXXX 0400 - FF	INTST	Registers
XXXX 0500 - FF	CRSET	Registers
XXXX 0800 - FF	GPDMA	Registers
XXXX 0900 - FF	COMET/PAKIT	Registers
XXXX 0A00 - FF	CHKSM	Registers
XXXX 0B00 - FF	LINKC	Registers
XXXX 0C00 - FF	RAALL	Registers
XXXX 0D00 - FF	VIMEM	Registers
XXXX 1000 - 1FF	BCACH	Registers & Array
XXXX 1200 - 3FF	CSKED	Registers & Array
XXXX 1400 - 5FF	SEGBF	Registers & Array
XXXX 1600 - 7FF	REASM	Registers & Array
XXXX 1800 - FFF	RXQUE	Registers
XXXX 2000 - FFF	NPBUS	Registers & External EPROM
XXXX 3000 - FFF	POOLS	Registers & Arrays

Figure C-1 CHARM-lite Memory Map for Registers and Arrays

APPENDIX D
List of Abbreviations

The following is a list of abbreviations used in the paper:

AAL	ATM Adaptation Layer
ACM	Association for Computing Machinery
ATM	Asynchronous Transfer Mode
BIOS	Basic Input Output System
BIP	Bit Interleaved Parity
CAS	Column Address Strobe
CLP	Cell Loss Priority
CRC	Cyclic Redundancy Check
CS	Convergence Sublayer
DMA	Direct Memory Access
DEC	Digital Equipment Corporation
DRAM	Dynamic Random Access Memory
FIFO	First-In-First-Out
HEC	Header Error Checksum
IBM	International Business Machines Corporation
IOP	IO Processor
IRQ	Interrupt Request
LAN	Local Area Network
LC	Logical Channel
LCD	Logical Channel Descriptor
LILO	Linux Loader
MPEG	Moving Picture Experts Group
OAM	Operations Administration and Management
PCI	Peripheral Component Interconnect
PHY	Physical Layer Device
PVC	Permanent Virtual Circuit
QoS	Quality of Service
RAS	Row Address Strobe
SAR	Segmentation and Reassembly Layer
SVC	Switched Virtual Circuit
VBR	Variable Bit Rate
VCC	Virtual Channel Connection
VCD	Virtual Channel Descriptor
VCI	Virtual Circuit Identifier
VM	Virtual Memory
VPI	Virtual Path Identifier
WAN	Wide Area Network