Project Number: MLC-RL99

Development of a Linux Device Driver for an Emulex Fibre Channel Interface

A Major Qualifying Project

submitted to the Faculty

of

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

____

John V. Baboval

____

Josh Huber

May 2, 2000

Approved:

1. Linux
2. Fibre Channel
3. RAID

____

Professor Mark Claypool

Abstract

This project continues development of a Linux device driver for an Emulex Light Pulse Fibre Channel interface. This document discusses hardware specifications, basic driver design, and implementation details. During the course of this project, we implemented all the low-level functionality of the LP6/7/8000 card, and started work on SCSI encapsulation code. All card initialization is performed up to fabric login. Future work includes implementing fabric login and extensive performance testing.

Contents

List of Figures

Chapter 1

Introduction

Modern enterprise computing becomes more demanding everyday. As more powerful servers serve more and more users, the need for flexible high performance storage and networking becomes greater. At the same time, the convenience for shared storage among multiple servers, as well as remote storage is desired. Along with these new requirements, classic reliability issues still exist.

Traditionally, reliability and performance issues were solved by combining SCSI with RAID. SCSI provided a moderately high performance data channel that was both flexible and inexpensive. RAID provided fault tolerance and increased performance. The disadvantage to this is that SCSI has approached it's upper limits of speed and distance. For this reason traditional SCSI can no longer keep up in higher-end servers. Fibre Channel has been developed as a SCSI replacement with high performance and additional features, such as redundancy and multiple simultanious protocols, that make it more useful in server clusters.

1.1 CLARiiON

CLARiiON Advanced Storage Solutions, a division of EMC^2, develops high-end RAID storage and end-to-end Fibre Channel solutions [2]. Many of their current storage solutions are based on their AViiON line of servers running Data General UNIX (DG/UX), which is no longer being developed, and Windows NT. CLARiiON sponsored this project with WPI to develop a Fibre Channel driver for Linux, to be used as a possible alternative to Windows NT on their AViiON servers. To this end, CLARiiON lent WPI a 10-slot Fibre Channel RAID, an AViiON server, and an Emulex LP8000 Fibre Channel PCI card for this project. We have used this equipment, with Linux running on the AViiON server, to develop a device driver for the Emulex LP8000 card.

1.2 Motivation

There were two primary motivations for this project, both on our part and CLARiiON's, to write this device driver. The first was to replace DG/UX, Data General's in-house version of UNIX. CLARiiON currently uses DG/UX as a development platform for their RAID development. They also ship complete storage solutions based on DG/UX, among other platforms. Over the years, it has become too expensive for Data General to continue maintaining DG/UX—CLARiiON's business is RAID, not UNIX. By using Linux, CLARiiON gains many advantages. Linux is freely available at no licensing charges, even for a full source license. It is available to anyone including CLARiiON and all its customers, allowing CLARiiON to subtract the cost of DG/UX from the price tag. Linux is being constantly developed, at no cost to CLARiiON. In addition, the customer will have a wide range of Linux distributions to choose from, any one of which will work with exactly the same kernel and LP6/8000 drivers.

Our personal motivation for this project was to help fill a gap in Linux device support. Fibre Channel drivers are almost non-existent for Linux on platforms other than Sparc. There is a notable exception to this; Qlogic has a driver for their Fibre Channel interface, however the card emulates a SCSI interface thus limiting the performance and feature set. The Emulex LP6/7/8000 series of interfaces are some of the most feature rich available, and Linux drivers for the device have been long awaited.

1.3 Technologies

There are three main technologies used in this project: the Linux operating system, Fibre Channel, and RAID. Linux is a clone of the Unix operating system originally written by Linux Torvalds and is openly maintained by a diverse collection of developers worldwide. Fibre Channel is a high speed serial bus. FCP, an implementation of the SCSI command set on Fibre Channel, is the primary use of Fibre Channel. RAID is a series of technologies that provide ways of combining several physical disks into a single logical disk providing a combination of performance increase and fault tolerance. By using SCSI commands over Fibre Channel this new, high performance data bus can be more easily integrated into existing environments.

1.3.1 The Linux Operating System

Linux[3] is a freely available, freely distributable UNIX-like operating system. The original kernel was developed by Linus Torvalds while he was a Computer Science student at the University of Helsinki, Finland, while the other core components of the operating system, including system libraries, compiler, and other system utilities were taken from the work of the Free Software Foundation[4](FSF). The Linux kernel, and all software released by the FSF, is released under the GNU General Public License[5](GPL).

While most licenses restrict the redistribution or modification of software, the GPL instead ensures that anyone who obtains a copy of GPL-licensed software has the right to freely modify and redistribute the software as much as they like, with two restrictions. First, any redistributions must also be released under the GPL. Second, you must release all source code modifications you have made, thereby giving the next person down the line an equal chance to view and modify the software. As a result, hundreds of people have read, debugged, and enhanced the popular GPL software packages, making them typically more stable and full-featured than their commercial counterparts.

For CLARiiON, the social and moral reasons for releasing and using software developed under the GPL are secondary. Rather, the fact that Linux is a stable, efficient, rapidly developing operating system means that they will be able to deliver products to their customers for less total cost than maintaining DG/UX, the in-house UNIX they currently ship on all servers. The liberal redistribution policy of Linux also makes it an ideal platform for projects such as ours. The full source code can be easily obtained with few worries about restrictive licensing, red tape, or fees. And, best of all, all work done by this project may be freely published without getting further agreement from any of the authors, as this is explicitly allowed by the GPL.

1.3.2 Fibre Channel

Fibre Channel [12] is an I/O channel and network technology that was developed and standardized by ANSI in 1994 to provide a high-performance, flexible architecture to interconnect storage systems, computers, and networks together over the same physical medium. Fibre Channel combines the best aspects of both networking and I/O channel technologies, and provides compatibility by allowing existing network (IP, IEEE 802) and I/O (SCSI, IPI) protocols to run over it. The Fibre Channel standard allows for performance ranging from 100 megabits/sec to 4 gigabits/sec over copper or fiber optic interconnects. Fibre Channel also supports multiple topologies allowing arbitrated loop (FC-AL), point-to-point, and switched environments. These features make it easily adaptable to nearly all high-performance, high-reliability applications.

1.3.3 RAID

There are two main reasons for using RAID: increased performance and increased reliability. RAID provides two main features to achieve this. The first of these features is striping. Striping interleaves data across devices in an array to take advantage of the devices speed in parallel. The second feature is mirroring. Mirroring provides redundant copies of all data across devices in an array so that after device failure, the device contents can be rebuilt on a replacement device. RAID has been implemented with various bus technologies including IDE, SCSI, and Fibre Channel.

Chapter 2

Background

Before we delve into the details of our project, we present a little background information on the core technologies that we have dealt with. In this chapter, we will discuss RAID, SCSI, Fibre Channel, and programming the Linux kernel.

2.1 RAID

The basic idea of RAID, which was developed in 1987 by Patterson, Gibson and Katz at the University of California Berkeley[9], is to combine multiple inexpensive disk drives into an array in order to increase the performance and/or reliability of the storage device. The different methods of combining the individual drives are called RAID *levels*. Patterson[9] originally defined RAID levels 1 through 5, described below. Since that, other configurations have been defined by various vendors. A popular addition is RAID-0, also described below.

2.1.1 Mirroring and Striping

Before discussing the specific RAID levels in detail, it is helpful to understand the basic building blocks upon which the RAID levels are built. The basic idea of RAID is to combine multiple physical disks into one logical or virtual disk. This is done by *mirroring* and *striping*.

Mirroring is just what it sounds like—maintaining several copies of the data, with one copy per physical disk. Mirroring increases the overall reliability of the data storage system, since if one disk fails, there are other copies of the data to recover from.

With striping, the data is split into fixed sized chunks that are distributed among all of the disks. Striping increases performance, since the I/O is split across multiple separate devices, but it decreases reliability, since if one disk fails, all the data is lost.

Each RAID level uses various combinations of these two techniques to increase both the fault tolerance and performance of the array.

2.1.2 RAID-0

RAID-0 is just simple striping as described above. The size of each stripe will affect the performance of the array. Typically, large stripes are used for I/O intensive applications, such as Usenet news servers. Smaller stripes (such as sector-sized stripes) may be used, but in that case the spindles of the individual disk drives usually must be synchronized to maintain adequate performance. Since no redundant storage of data is done in RAID-0, a loss of one drive will cause all data on the array to be lost. The capacity of a RAID-0 array is equal to the sum of the capacities of each drive in the array.

2.1.3 RAID-1

RAID-1 is plain mirroring. The read performance in this case will be n times that of a single disk, where n is the number of drives in the RAID-1 array. Since the data needs to be written to all disks simultaneously, the write performance remains unchanged. Reliability of a RAID-1 array is also n times that of a single disk, since there are always n copies of the data. The capacity of a RAID-1 array, however, is only as much as a single disk.

2.1.4 RAID-1/0

RAID-1/0 combines striping and mirroring together. This yields the performance of a pure stripe, along with the reliability of mirroring. This is the best case for both performance and reliability. It is also the most expensive, since it only gives the user n/2 disk space. For high availability situations, where there is typically not a large amount of data, it is a cost effective solution.

2.1.5 RAID-2

RAID-2 is similar to RAID-0 with sector-sized stripes, except some of the disks are used to store ECC (Error Correcting Code) information. This level is rarely used anymore due to most modern drives having built in ECC.

2.1.6 RAID-3

RAID-3, like RAID-2, uses striping with sector-sized stripes and drive spindle synchronization between the disks of the array. However, instead of storing ECC information, one drive is used to store parity information. This parity information can be used to recover the data on a single failed disk by exclusive-ORing it to the data on the other disks. Since each write operation requires reads to every disk in the array in order to calculate the parity information, write performance is not much improved over single disks. The capacity of a RAID-3 array is n-1 times the capacity of each disk in the array.

RAID-3 provides the best read throughput of all RAID levels while still providing some degree of protection against lost data. The best use for RAID-3 is reading large contiguous disk blocks, where RAID-3 avoids the drive head seek bottleneck. Small writes are not good for RAID-3, since the parity disk must be written to at the same time as the data disks. This makes RAID-3 more suited to high throughput tasks such as multimedia streaming.

2.1.7 RAID-4

RAID-4 is identical to RAID-3, except large stripe sizes are used to increase performance. Due to the larger stripes, an entire read operation will usually be from one disk, allowing multiple I/O operations to be overlapped. Writes, however, are not overlapped since the parity disk still needs to be updated. RAID-4 has been pretty much replaced by RAID-5.

2.1.8 RAID-5

RAID-5 removes the parity disk bottleneck of RAID-4 and RAID-3. Instead of storing parity information on a separate disk, RAID-5 rotates the parity among all the data disks. In this way, multiple reads and writes can be overlapped, since each operation will at most access two drives—the drive holding the data to be read or written, and the drive holding the corresponding parity information. Performance of RAID-5 arrays is not quite as good as RAID-1; however, they provide much better storage capacity for the same number of disks, as well as much better fault tolerance.

2.2 SCSI

SCSI, the Small Computer System Interface, is an I/O channel designed for connecting up to eight (and in later versions, sixteen) devices to a computer in a bus configuration. The original SCSI specification (now called SCSI-1), specified a synchronous 8-bit parallel interface running at 5 MHz, for a total throughput of 5 megabytes/sec. This was extended in several ways by increasing the bus speed, utilizing asynchronous transfers, and widening the bus, such that the latest SCSI systems are able to achieve a throughput of up to 80 megabytes/sec. All SCSI systems use basically the same communications protocol, with the later versions adding various extensions. This protocol can also be used over other physical interfaces, such as Fibre Channel.

2.3 Fibre Channel

Fibre Channel is popular as a simple replacement for the highly complicated proprietary disk channels and storage area networks currently used in high-end computing environments. Fibre Channel uses several new concepts in the area of disk interconnections. Flexibility of configuration, cable topology, administration simplification, and cable length are all improved over SCSI.

The Fibre Channel specifications are structured into five independent layers that describe various levels of functionality. These layers are Physical Interface and Media (FC-0), Transmission Protocol (FC-1), Framing and Signaling Protocol (FC-2), Common Services (FC-3), and Upper Layer Protocol Mapping (FC-4). Below, we discuss the lowest three layers together as the Physical Layer, since more detail at those layers is not in the scope of this paper.

2.3.1 Physical Layer

As SCSI speeds have increased over the years with the introduction of technologies such as Ultra Wide SCSI, cabling and termination issues have become more and more critical. By using large numbers of parallel connections in a ribbon cable (often over 50 physical connections), crosstalk becomes a severe problem, demanding that nearly half of the connections are grounds. While SCSI may support as many as 15 devices on a single physical set of cabling, this is often difficult due to the distance limitations of the cable. This limitation is somewhat helped by requiring termination on each end of the chain, but this in turn introduces another element that must be carefully configured as well as another point of failure.

Fibre Channel is considerably more flexible. The physical layer specifications FC-0, FC-1, and FC-2 are defined by the FC-PH document. In a typical configuration, a 4 wire serial connection is used. Fibre Channel can run at speeds from 133 Mbaud to 4.25 Gbaud. Typical configurations run at 1.06 Gbaud. When the overhead due to the 8B/10B encoding is figured in, this has a theoretical bandwidth of about 100 MB/s. This is considerably better than the 40 MB/s that Ultra Wide SCSI can do. Also, future increases in speed up to 4.25 Gbaud are being developed, with a total throughput of about 400 MB/s [12].

Copper Media

The more popular way to wire Fibre Channel devices is with either twin-axial copper cable or specially made twisted pair serial cables. For short lengths, such as the connections between disks, copper cable is much easier to use. Copper cable is more common, due to fiber optics being significantly more expensive and more sensitive to physical damage. Typical configurations with twin-axial cable can support distances of up to 100 meters. Television coaxial cable is another supported copper physical medium.

Optical Media

Fibre Channel can use an optical carrier too, thus making campus wide networks, remote replication, and storage area networks a reality for data centers and distributed computing sites. The reader should not infer that Fibre Channel means fibre optic carrier; Fibre Channel is more or less independent of the physical layer. Singlemode and multimode fiber optics are supported at distances of several hundred meters to over 2 kilometers.

2.3.2 Topologies

Fibre Channel disks and arrays can be connected by point to point, arbitrated loop, or switched fabric type connections. This allows for more advanced configurations, such as multiple servers connected to a single disk. While such a configuration is very difficult with SCSI, it is routine with Fibre Channel.

Point-to-Point

Fibre Channel ports can be connected to each other in a point-to-point configuration, either in a manner similar to a crossover cable used for ordinary RS-232 serial ports, or via a switch which sets up the point-to-point connection. In a point-to-point configuration, the two devices have a dedicated bandwidth, full-duplex connection to each other.

Arbitrated Loop

Arbitrated Loop is a shared bandwidth connection that allows up to 127 devices to be connected in a daisy-chain fashion. Each device arbitrates for access to the loop, and then sets up a logical connection to the target device. Only one connection on the loop can be open at any one time. During this connection, the source and target devices have full access to the total bandwidth available on the loop. Arbitrated Loop is the preferred connection topology for connecting disk arrays; it is designed as a direct replacement for SCSI.

To alleviate the problems with broken loops when a device in the loop is powered off or removed, a Fibre Channel hub can be used to create a physical star topology. In this case, if the loop is broken, the hub will complete the loop, bypassing the disconnected or powered off device. This is similar in operation in token ring concentrators.

Switch Fabric

If an arbitrated loop does not provide enough bandwidth for the devices using it, it is also possible to use a switch to increase capacity. This provides what is physically a star topology, as well as many simultaneous virtual point to point connections. A switch will also allow a device to disconnect from a loop without taking down every other device. A Fibre Channel switch can provide circuit switched service, frame switched service, or both. In addition, a switch can be used to interconnect individual devices (N_Ports), arbitrated loops (NL_Ports), and other switches (E_Ports). When many switches are connected together in this manner, the entire collection is referred to as a *fabric*. This flexible functionality is configured completely automatically upon startup of the switches, and allows for great scalability of the Fibre Channel system.

2.3.3 Classes of Service

Fibre Channel FC-2 provides several classes of service for connections made between two devices. These classes are described below:

Class 1 - Acknowledged connection-oriented service

This class of service is a dedicated bandwidth connection with guaranteed quality of service (QoS). The connection must be set up with a handshake sequence before it can be used. Each frame is delivered in-order and receipt is acknowledged. Once the connection is no longer needed, it must be disconnected with another handshake sequence. While two devices have a Class 1 connection open, no other device can talk to them (except when Intermix mode is used—see below).

Class 2 - Acknowledged connectionless service

This class of service is similar to Class 1, except no connection set up sequence is needed. A device simply transmits Class 2 frames whenever it wants, and receipt of the frames is acknowledged by the destination. Each frame may follow a different path in the switch fabric, since no path is dedicated for this class of service.

Class 3 - Unacknowledged connectionless service

Class 3 is just like Class 2, except no acknowledgement of frames is done.

Class 4 - Fractional bandwidth connection-oriented service

Class 4 is a connection-oriented protocol similar to Class 1. In the case of Class 4, however, only a fraction of the total available bandwidth is dedicated for the connection. Quality of service parameters are used to set up the desired bandwidth characteristics for each direction of the connection independently. Delivery is guaranteed in-order with acknowledgements.

Class 6 - Simplex connection service

Class 6 is similar to Class 1 in simplex mode, providing unidirectional and multicast connections. This is ideal for sending the same data to many devices at the same time.

Intermix

Intermix is a special mode of Class 1 connections. In this mode, the full Class 1 bandwidth is dedicated for the connection, but other Class 2 and Class 3 connectionless traffic can be sent to the two devices when the bandwidth is not otherwise being used for the Class 1 connection.

2.3.4 Upper Layer Protocols

Mapping Upper Layer Protocols (ULP's) to Fibre Channel is defined in the FC-4 documents. Several protocols are supported simultaneously over a single Fibre Channel system. Each protocol is defined in its own FC-4 document. A few of the more important protocols for which FC-4 mapping specifications have been written include SCSI (FCP), Intelligent Peripheral Interface (IPI), Internet Protocol (IP), and IEEE 802 (FC-LE).

2.4 Programming the Linux Kernel

While nearly all of the Linux Kernel is written in C, including all of our work, there are still a few programming difficulties not present in a typical program. The first of these is that the standard C library functions, such as string handling, network and file I/O, and advanced memory management are not available. In some cases, such as string handling, some of the simpler functions have been re-implemented, though they are much simpler than the more advanced versions found in a full blown C library. Other functions, such as simple memory management, have Application Programmer Interfaces (API's) very closely related to the standard C version (i.e., `kmalloc()` instead of `malloc()`), and still others, such as network and file IO, are sufficiently different and difficult to handle directly that they are rarely if ever used directly.

A kernel programmer must also be more aware of the resources used, as the limitations placed upon user space programs do not apply to the kernel. Any memory allocated by the kernel must be held in physical RAM at all times, meaning that it cannot ever be swapped out. Any kernel code can disable interrupts if it so desires, or spin in an infinite busy loop, effectively locking up the entire system. And, with full access to any and all system and device I/O ranges, the driver must be careful only to access resources known to be safe.

Luckily, there are standard API's for nearly all of these problems. There are routines to query if resources have been claimed by any other drivers, ways to schedule large portions of work without locking up the entire system, and enough information is readily available directly from the kernel such that, beyond data buffers, it should rarely be necessary to allocate large chunks of memory. As long as the API's are properly used and a little thought and care is put into the driver, none of these are overly large problems.

2.4.1 IRQ Handling

Setting up interrupt handling under Linux is fairly straightforward. Once the IRQ used by the device is determined[1], all that you need to do is call the kernel `request_irq` function and pass it the IRQ number to be used and the function to be called whenever that IRQ is raised. From that point on, the kernel will call the handler function whenever it detects that IRQ. The kernel will also prevent any other device drivers from registering that IRQ, unless both drivers have been written to allow IRQ sharing. Note that the hardware must also support IRQ sharing; it is assumed that the device driver will have knowledge of this and pass the appropriate flags to `request_irq`. Likewise, the `free_irq` function may be called to release the IRQ handler. This can be useful for loadable modules, or for devices that will only use the IRQ when the device is opened and in use, for example.

While the IRQ handler is running, interrupts should be disabled. Since no other interrupts can be detected during this time, it is in the best interest of overall system stability and performance for the interrupt handler to finish and re-enable interrupts as soon as possible. Since there often is lengthy work to be done, such as computing checksums on packets or copying

[1] This can be difficult with some devices, particularly ISA ones where auto probing can be dangerous - luckily the LP6000 card is a PCI device, which greatly simplifies matters.

large amounts of data into and out of buffers, the *bottom half handlers* are frequently used.

2.4.2 Bottom Half Handlers

Often times, there are situations where a kernel author wants to defer a block of work until later. The most common situation is in the interrupt handlers, where the actual handler wants to do its work as quickly as possible and exit so that interrupts may be re-enabled. To meet this need, the bottom half handler code was created.

To register an instance of a bottom half handler, the `queue_task()` function must be used. The three most important pieces of information passed are the function to be called, a single argument passed to the function (typically a pointer to a struct containing more information), and which bottom half queue is to be used. The function `mark_bh` must then be called on the queue to which the function was just added.

Exactly when the function gets called depends on which queue was used. The maximum number of bottom half handlers is currently a fixed number, 32. Since adding new bottom half queues requires modifying core kernel code directly, and there is no API to do so, it is very rare for device drivers (which are often distributed as a stand alone source code file separate from the main kernel) to create their own queues. There are several task specific queues that may be used, including one for network activity, and one for console I/O activity. The most commonly used generic queue is `IMMEDIATE_BH`. This queue runs immediately on return from interrupt handlers and system calls.

2.4.3 PCI Handling

One of the greatest advantages of PCI devices is intelligent auto-configuration. With older ISA devices, the user must manually configure each device, ensuring that there are no resource conflicts. With PCI devices, the PCI BIOS ensures that all PCI devices are configured properly at each bootup. This allows for virtually trouble free automatic configuration, including easy handling of added and removed devices with no configuration changes required by the user. In addition, the fact that all configuration is done by the PCI BIOS allows the operating system to easily obtain all configuration values by simply querying the PCI BIOS itself, rather than having to probe all present PCI devices separately. Once the generic resources, vendor ID, and card ID in use by each card is known, then further device specific probing and configuration may safely be done without danger of interfering with an unknown device.

The Linux kernel currently provides two PCI API's. The first one, the `pcibios_` class of functions, is provided for backwards compatibility. It provides a function to find a PCI device based upon the vendor and/or card ID, and a pair of functions to read and write to the PCI configuration space of that particular card. Device specific information is kept in a `struct pci_dev`. Values of offsets to commonly used information, such as I/O regions and IRQ's, are provided as `#defined` values.

The second API, the `pci_` class, is much newer. It is only present in the development kernels 2.1.90 and newer. It is much simpler to use, however. The primary difference is that much of the information which previously could only be obtained by accessing the PCI configuration space for a particular card is now available in the `struct pci_dev` for that device. Also new is the `pci_set_master` function. With the previous API, to configure a card for bus mastering operation, the author would simply rewrite the PCI command word. With newer kernels, this step is much easier—one simply calls the `pci_set_master` function. In addition to writing out the appropriate word, this function will adjust PCI latency timings, and perform any other kernel and PCI BIOS configurations needed for proper bus mastering operation. More documentation on the PCI API, in particular the `pcibios_` one, can be found in the Linux kernel source tree under `Documentation/pci.txt` and `include/linux/pci.h`.

2.4.4 Memory Management

Basic memory management is, from the point of view of a device driver, fairly simple. For most small to medium sized memory requirements, `kmalloc` and `kfree` can typically be used. The `kmalloc` function is almost identical to the standard C library `malloc` function, the biggest difference being that it takes two arguments instead of just one. The first argument for `kmalloc`, like `malloc`, is the amount of memory in bytes. The second argument is the priority. Two priorities are typically used, either `GFP_KERNEL` or `GFP_ATOMIC`.

The difference between the two is that `GFP_ATOMIC` allows access to a special reserved pool of memory that is set aside for situations where waiting to swap out a page is not an option, such as during an interrupt. When `GFP_KERNEL` is used, on the other hand, the calling routing must be prepared to block while the system attempts to free up the needed memory.

When larger amounts of memory are needed, however, it is more efficient to use a page oriented scheme. There are two sets of functions for doing this. The first set is used for grabbing pages directly. They are `get_free_page`, which returns a pointer to a new page, `__get_free_pages`, which returns a pointer to a multiple page-sized contiguous memory area, and `__get_dma_pages`, which returns a pointer to a region suitably located for ISA DMA. In order to free memory allocated by these functions, either `free_page` or `free_pages` should be called. All of the allocation functions take a priority, with semantics identical to the priority passed to `kmalloc`. The multiple page versions (`__get_free_pages` and `__get_dma_pages`) also take an order argument. An order of n indicates that 2^n pages should be allocated.

The other method that may be used is the `vmalloc` function. This function takes one argument, the size of the memory block requested. It works by requesting enough pages to make up the needed amount individually with `get_free_page`, and then modifying the page tables to make the newly requested pages appear to be contiguous in memory. In reality, though, the pages may be highly fragmented across memory, and the addresses returned are always beyond the actual highest physical memory address. This fragmentation makes `vmalloc` unsuitable for allocating buffers to be passed to devices, as they nearly always expect the buffer to be contiguous. In order to free the memory, the `vfree` function should be used.

In certain extreme cases, even more contiguous memory than this may be needed. In particular, note that `get_free_pages` will only take a maximum order of 5, for a maximum of 32 consecutive pages. The only method left then is by grabbing the memory at boot time. During the boot sequence, each driver has a chance to modify the upper bound of memory. The driver can lie about the new upper bound by lowering it to give itself enough room to work in. Since the kernel no longer believes that region of memory exists, the driver is free to work with that range as it will. Note that this memory can never be returned to the kernel, and this method will only work for compiled in drivers, not for loadable modules.

2.4.5 Accessing I/O Addresses

Typically, PCI devices will have one or more sections of memory to be mapped out for the purpose of transferring data to and from the device. The PCI BIOS determines the exact address this memory is to be mapped out to, typically in a BIOS dependent range near the top of the address space. On some architectures, this may be mapped into the main memory address space; on others, the I/O buses may have their own separate address space.

In either case, this memory needs to be made accessible before it can be used. At boot time, the page tables are only set up with an entry for each page of physical or virtual memory. Therefore, the `ioremap(address, size)` function must be called. Very simply, it adds the needed entries into the page tables for the mapped I/O range. If this is not done before the memory is accessed, then the Memory Management Unit (MMU) will indicate a request for an invalid memory address, causing an invalid page fault and kernel panic.

Once the memory has been remapped with `ioremap`, a bit of care must still be taken accessing it. On some architectures, the address returned by `ioremap` is not directly usable by the CPU. The driver cannot directly dereference the address, but rather must go through the right address space. Therefore, to be portable, the `readX` and `writeX` functions (X equals b for 8 bit byte, w for 16 bit word, and l for 32 bit long) must be used for all access to the I/O addresses.

2.4.6 The SCSI Layer

The only portion of the Linux SCSI layer which we concerned ourselves with is that of writing a low-level SCSI driver. The bulk of the SCSI logic is contained in the middle and upper SCSI layers. When a SCSI driver is initialized at boot time, or loaded later during run time, it registers itself with the upper layers by passing a struct containing various flags indicating card and driver properties and abilities, and several functions. There are four key sections that the driver presents to the upper layers: initialization, the SCSI command handler, the interrupt handler, and the abort function.

The initialization code is called during boot time or module load time. It is expected to find all cards handled by that driver, configure them into a known ready state including reserving IRQ and I/O ranges, and register each one with the upper SCSI layers.

The SCSI command handler takes SCSI commands from the upper layers that have been issued as a result of events such as disk reads and writes. The typical method is for the device driver to add the SCSI command to a driver-specific queue for processing later. Our driver uses a bottom half handler to defer processing of the command to a later time. In addition to the SCSI command itself, a pointer to a function is also passed. Once the driver has finished processing the given SCSI command, this function must be called on the command to signal to the upper SCSI layers that this command has been fully processed. If this function is not called within a given timeout period, then the upper layers assume that an error has occurred and attempt to abort the command.

The interrupt handler is invoked whenever the card raises an interrupt. It must determine which section of the card raised the interrupt, copy any pertinent information from the card into host memory for later processing, and clear the relevant interrupt bit on the card. Like the SCSI command handler, the interrupt handler defers the bulk of the processing work via a bottom half handler.

The abort function should only be called when an unrecoverable error has occurred. It is called upon a particular outstanding SCSI command when the error is detected. For example, if the upper layers detect a timeout, they will call this function. The abort function is expected to dequeue the command entirely.

Chapter 3

Goals

We had the following goals at the start of this project:

1. Continue work on the partially completed Emulex LP 6/7/8000 Fibre Channel (SCSI only) driver for Linux.

2. If time permitted, conduct performance/configuration testing.

3. Examine platform specific issues with the driver.

3.1 Continue work on Emulex driver for Linux

When the development of the *emulexfc* driver was started the latest series of the Linux kernel was 2.1. Linux 2.1 is a development series (all the odd minor version numbers designate development work, while all the even number designate released work) and thus had many changes being done. Finer grained kernel locking, and bottom half handlers allowed for better performance and scalability, especially on MP systems. Alternative architecture support was also greatly increased.

At the time that we picked up development of the *emulexfc* driver the kernel was well into the 2.3 series. We chose to do our primary development work with the 2.2 series because that gave us a stable KPI (kernel programming interface) to work with.

A primary area of work was more robust handling of IRQs. It was our goal to move the *emulexfc* driver from it's polled state to a faster, more robust interrupt driven state. The reason the previous group (and us) chose to go this route was because writing the polled driver is simpler and makes it easier to debug. After the required functionality is present, converting to using hardware interrupts is a semi-trivial task. [1]

We worked with the SCSI layer of the Linux kernel in development. The SCSI layer provides the framework to translate user-space reading and writing to the raw SCSI commands to be delegated to the specific hardware driver. In our case, to goal was to use the Linux SCSI driver as a method of controlling, managing, and exporting the functionality of our disk array to user space. We will be encapsulating the SCSI commands provided to us from the kernel in FCP packets, and sending the resultant packets to the array or fabric.

[1] I say semi-trivial, because there's bound to be something that goes wrong!

3.2 Performance and Configuration Testing

As with the group prior to ours, an important goal of this project was performance testing. Once the driver is in a working state, performance testing will be possible. Using various disk intensive methods (see the future work section) a formal test of performance could be made. Not only could we examine the RAID array, and the card itself, but the Linux SCSI layer would be a prime area to analyze for bottlenecks.

Testing with all the possible configurations is also important. To aid us in this task we have a fibre channel switch, which can be connected, not unlike an Ethernet switch, between the host computer and the array (or another computer) on the other end.

3.3 Alternate Platform Support

Another goal for the future is better cross platform support. Because PCI is guaranteed to be little-endian across all platforms, this greatly simplifies cross platform driver development. According to the chipset documentation, the Emulex LP series of cards support a byte-swapping mode (via a programmable PCI register), so that no extra work is necessary to support big-endian machines. Even without this feature, all that would be required would be simple changes to the read/write macros.

As for supporting 64 bit architectures (such as the Alpha), it is possible that more work would be necessary. For the driver to support 64 bit architectures might not be much of a chore, but for it to really take advantage (from a performance point of view) would take much more effort. This would be a goal for the more distant future.

Chapter 4

Methodology

This chapter details how we researched the project, how we did our development, what resources were used, what difficulties we encountered, and how we attempted to overcome them.

4.1 Linux Emulex Driver

Given a partially written driver, and an unfamiliar code base, our initial research was entirely on how the existing code worked along with learning the kernel/driver interface. Fortunately, the Linux kernel was well documented early on, so we were able to take advantage of online[13] and published[6] documentation. Unfortunately, much of this documentation is out of date, and information about the new interface had to be derived from the kernel changes documentation, which documents kernel changes from version to version. To learn about the existing driver code, we followed the driver's procedures through along with the flowcharts that came with the Emulex "documentation" from Clariion.[1]

The next task on our list was to clean up the code we were given by the previous group. Though there was nothing seriously wrong with the code, it had the feel of a project that was worked on by frustrated developers. The cleanup involved splitting the distinct sections of card initialization up into separate functions, removing non-descriptive and random debugging code, commenting sections, and removing GOTOs where they were unnecessary.

After learning the ins and outs of the existing code and performing spring cleaning, we felt rather confident to begin development with alacrity. We were presented with new documentation that was to be far more accurate than the documentation that the previous group had to work with. The existing code provided a complete layout for the architecture of the driver, so all we really had to do was fix all of the parts of the driver that didn't work.

From here, our methods became rather dynamic. Each problem we solved or encountered determined which section of the driver we worked on next.

The first problem we chose to investigate was the inability to reset the card. Though this was not a necessary function to be able to use the card, as it resets itself properly on power up, having to power cycle the machine after every test run was a major time hindrance. To solve this problem we first wrote lots of debug code to assure ourselves that we were actually doing everything we thought we were. After determining that we weren't crazy, we attempted setting combinations of undocumented bits in the control register, and sending seemingly promising but undocumented mailbox commands to wake the card from it's post reset coma. We eventually gave up, choosing to spend our time on other problems the new documentation promised to alleviate.

Next, we moved on to fixing the initialization process. Each of the major components on the card has it's own initialization procedures. After the card is initialized, the link needs to be configured and remote ports need to be discovered and logged into. Working initialization required a working command mailbox for the chipset, so first we tweaked the mailbox code to our liking. (This was the most complete section of the existing code, so this was not too much trouble.) We then stepped through each of seven initialization phases insuring that each step in the chipset initialization worked perfectly.

When we had reached the point of link initialization, we received new documentation for the card which included explanations of the previously undocumented mailbox commands. We proceeded to use this information to make the card reset work. From this point development was much smoother.

The last segment of initialization is port discovery and login. Our documentation included blank sections on how to do this, so we postponed development on this section and worked on the FCP section of the code. After significant progress in this section of the code, we also worked on the interrupt handler. We still await documentation about how to perform port discovery and login.

4.2 Stability and Performance Testing

Unfortunately, due to several setbacks, we were unable to produce a driver capable of giving any meaningful test results. However, future groups should plan to perform and document as much stability and performance testing as possible.

4.3 Development Environment

For coding, all members of the group opted to use GNU Emacs running on Linux machines. GNU Emacs provides a very nice environment for C coding, including such features as keyword highlighting, automatic indentation, and automatic interaction with the CVS and RCS source control systems.

For source control, CVS was used. With a small group such as ours, CVS combined with plenty of communication to avoid conflicts proved efficient for storing all work, including the code, project time-lines, documentation, and the final writeup. The CVS server was stored on our development machine and regularly backed up to prevent a disaster. Additionally, we set up the CVSWeb system, so that it was easy to follow our progress from the web, and download a current snapshot of our code.

4.4 Driver Implementation

There are two major parts to the *emulexfc* driver implementation. The first is the *card initialization* piece, the second is the *runtime* piece.

Card discovery is performed according to figure 4.1, by looping, and examining the PCI device information. After the driver knows which PCI cards present are supported Emulex cards, chipset initialization is performed according to figure 4.2. Each function is called in turn to set up different parts of the card, eventually, performing an actual login to the fibre channel device.

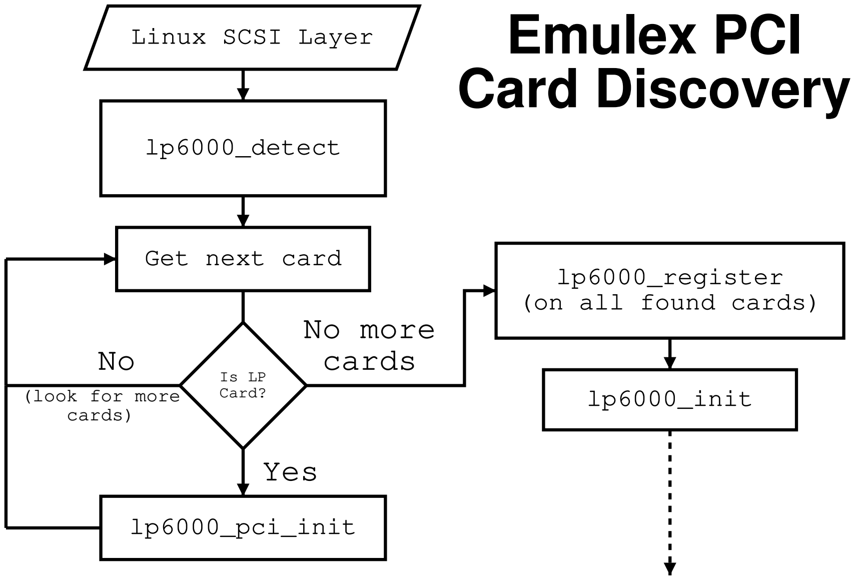

Figure 4.1: Emulex LP 6/8000 Card Discovery

Figure 4.3 shows the flow of data from user space (i.e. a process running, accessing the disk, through filesystem code) to device level. Additionally, the

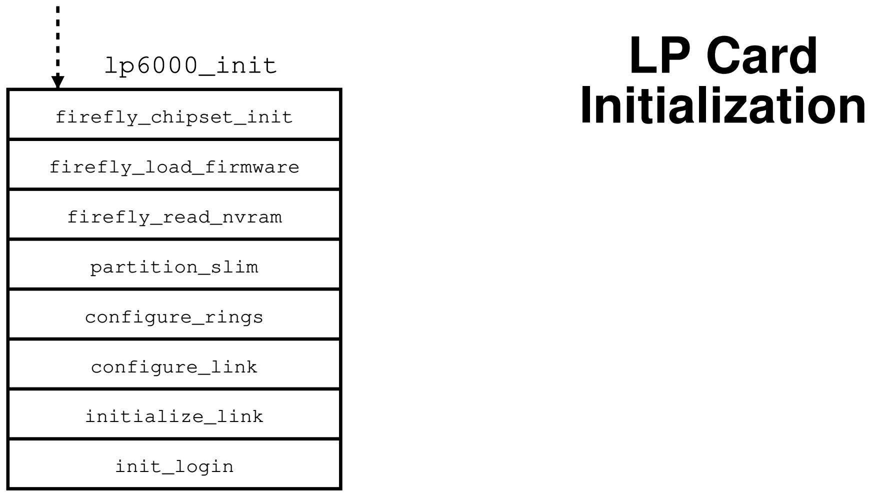

Figure 4.2: Emulex LP 6/8000 Card Initialization

rightmost part of said figure (4.3) details the Fibre Channel frame structure. Some information is left out for clarity.

Finally, figure 4.4 shows the interaction of each subsystem that makes the *emulexfc* driver work. After the card is initialized, and IRQ handlers are registered, the entire functionality is interrupt driven. See the Future Work chapter for more details.

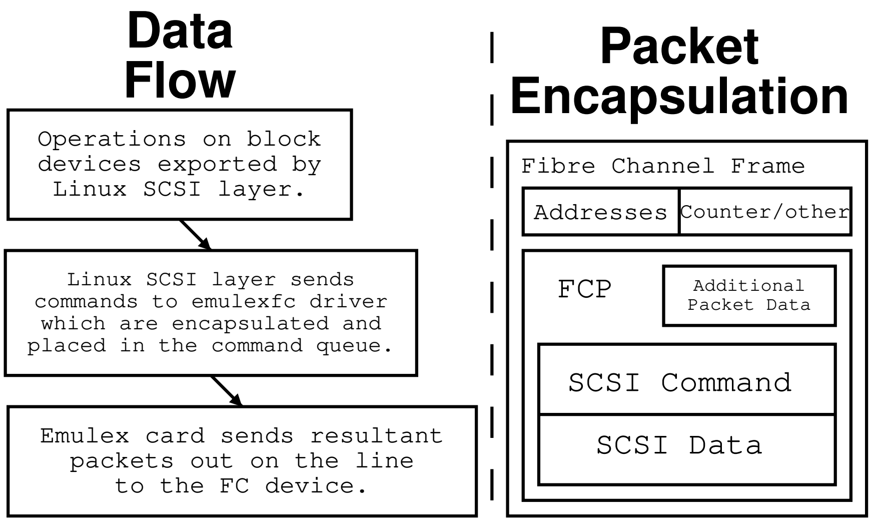

Figure 4.3: Fibre Channel Data Flow

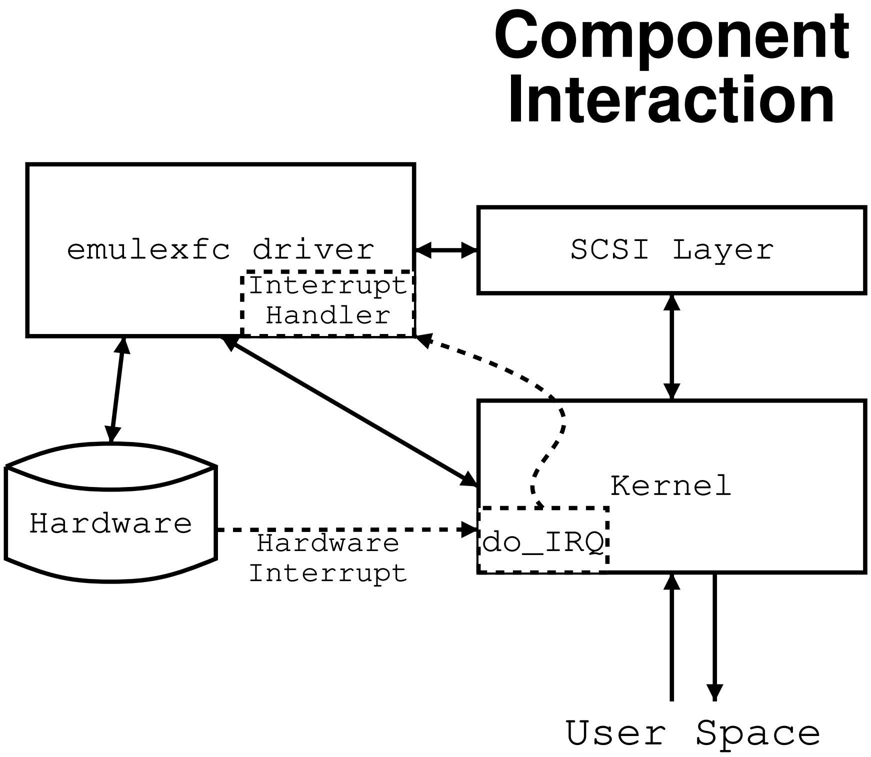

Figure 4.4: Driver/Kernel/Hardware Interaction

Chapter 5

The Emulex LP6/7/8000 Driver

The Emulex Light Pulse card is a high performance PCI Fibre Channel host adapter. It supports all Fibre Channel topologies (point to point, arbitrated loop, switch fabric), multiple Upper Layer Protocols (ULP), including both network (such as IP) and storage (such as FCP) loaded simultaneously. It has a maximum transfer rate of 1.0625 Gb/s at full duplex over either fiber optics or copper serial wire. The card is driven by the FireFly chipset.

The FireFly chipset is controlled entirely through a memory mapped region containing registers and command buffers. This region, called the SLIM (Service Level Interface Memory), contains four 32 bit control registers, a command mailbox, and a configurable number of Fibre Channel command and and response buffers called the I/O control block (IOCB). The registers

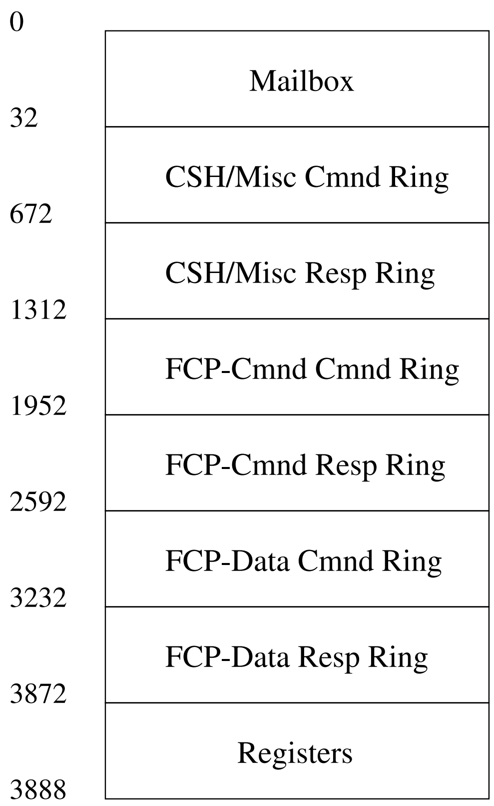

Figure 5.1: SLIM Partitioning Scheme

are used for status messages to the host, and controlling the card prior to mailbox initialization. The mailbox handles all post initialization commands to the chipset such as configuring the IOCB rings (partitioning the SLIM), reading card feature support information, and saving/restoring NVRAM values. The IOCB handles the sending and receiving of Fibre Channel data.

5.1 Registers

The FireFly chipset has four 32 bit registers in the SLIM. The registers are used for configuring, controlling, and checking the status of the card.

The Host Attention Register (HATT) is used by the FireFly chipset to notify the host of a condition that requires attention. The sources can be any one of the IOCB rings, the mailbox, link attention, or an error.

The Chip Attention Register (CATT) is used by the host to notify the FireFly chipset of a condition that requires it's attention, typically of a command that requires processing. The source can be any one of the IOCB rings, or the mailbox.

The Control Register is used to initialize chipset restart, and indicate which events should generate interrupts. The potential interrupts sources can be the mailbox, any of the IOCB rings, link attention, or errors. Each of these sources can be configured individually to generate interrupts or not. If interrupts are not enabled for any sources, then the host is expected to poll the status of that source periodically.

The Status Register is used by the FireFly chipset to indicate the status of the chipset. The conditions reported include the readiness of the chipset, readiness of the mailbox, and the various error conditions.

5.2 The Mailbox

The mailbox is used for sending commands to the FireFly chipset. The mailbox consists of eight 32 bit words mapped into host memory as part of the SLIM. It is primarily used during the initialization sequence, including partitioning the IOCB rings within the SLIM, and performing the fabric login.

The mailbox contains several fields. The first 32 bit word contains the return code, the command code, and the host contention (HC) and OWN bits. The HC bit is used as a mutex to exclude other processes on the host machine from accessing the host adapter when it is in use. The OWN bit is set to signal to the FireFly chipset that it may read the command and execute it. The chipset will clear the OWN bit when the command is complete. The remaining part of the mailbox contains any parameters necessary for the command being executed, as well as any result values.

5.3 I/O Control Block Rings

The IOCB rings are used for the actual transfer of data between the host and the bus. The SLIM can contain up to 4 IOCB rings. Each ring is configured to receive a specific type of Fibre Channel message. Since our driver is oriented towards FCP only, we used 3 sets of rings: one for Fibre Channel command packets, one for Fibre Channel data packets, and one for all other types of data. Each set contains two individual ring buffers, one for

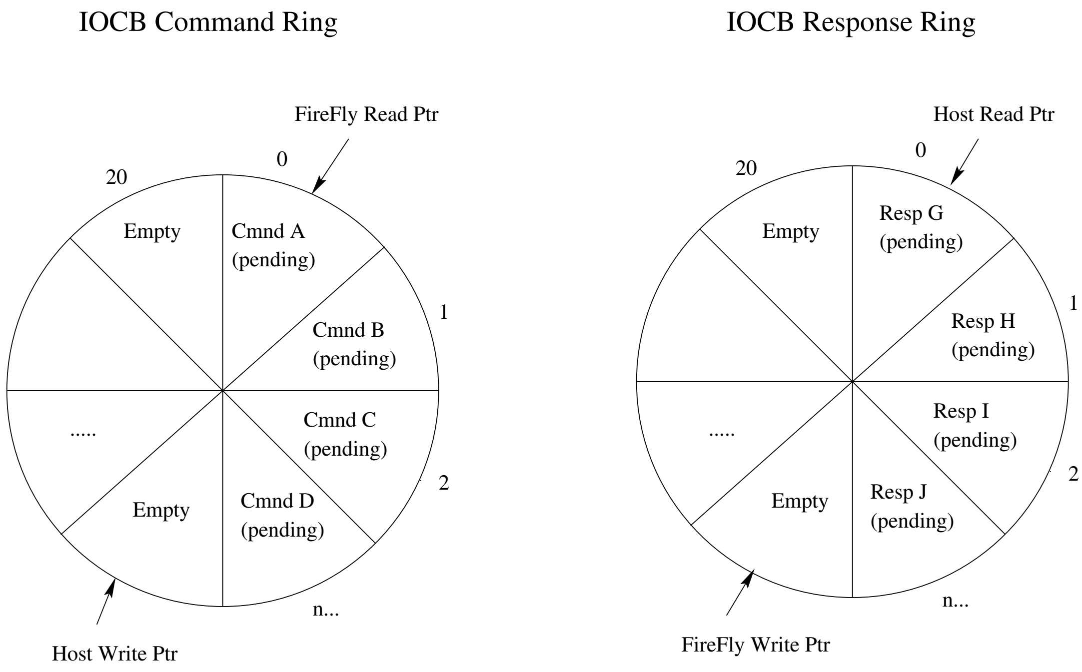

Figure 5.2: IOCB Ring Set

sending IOCB's, and one for receiving response IOCB's.

Each ring is logically presented as a circular queue. To send a command, the driver must wait for the current entry in the buffer is clear. This can be done by enabling interrupts, which means that the host will receive an interrupt whenever a ring entry is freed, or by polling the OWN bit on the IOCB entry. The IOCB is then written into the ring, and the OWN bit is set, indicating that the transfer is complete and the chipset may start reading it.

A similar procedure is used for receiving IOCB response entries. If IOCB response interrupts have been enabled, then the host will receive an interrupt when a new response is available, otherwise the host must poll the OWN bit on the current entry. Once it is determined that the response entry can be read, the entire entry should be read from the SLIM into host memory, and then the OWN bit should be set, indicating to the chipset that it may re-use the entry.

Each IOCB entry consists of 8 32 bit words. Words 6 and 7 contain the various status bits, including a unique tag used to associate the IOCB command and it's corresponding response. The host is responsible for ensuring that each tag is unique. Words 0 through 6 are command dependent. Most typically, for data transfers, it will contain one or more Buffer Descriptor Entries (BDE). The BDE's are used to point to the specific region of memory that the card is to transfer data to or from.

IOCB commands for Upper Level Protocols (ULPs) are highly optimized for their particular task. There are IOCB commands to handle all of the protocol specific tasks of both FCP and FC4 (IPv4 over Fibre Channel). This frees the host from processing and encapsulating data in software. Only Fibre Channel specific packets, such as Extended Link Services (ELS) requests, must be assembled by the host.

Chapter 6

Conclusions and Future Work

Our goal for this project was to take the work from the previous group and complete a working driver for the Emulex LightPulse series of cards. The previous group was working with a LP6000 card, and we were working with a LP8000 card. The two cards provide the same interface, so very little changes were required to get the driver to use the new card. The LP 8000 card is quite complex, and because of this, and certain documentation issues, we were unable to complete a working driver. Because of this, and because there are many features left to be implemented in our driver, there is sufficient work for one, possibly two more projects.

6.1 Current Driver Status

Currently, all low-level card functions are working, including card reset. Getting the card to reset without resorting to a cold restart of the system was essential, because without this feature, we would have wasted significant amounts of time waiting for a machine restart.

Most of the SLI-1[1] related mailbox commands have been implemented, as have some of the IOCB commands. We were able to bring up a fibre channel link, get status, card parameters, and were able to send arbitrary data across the fibre link to the target device.

Because important pieces of documentation were missing, we were unable to have a complete and functioning driver. Fortunately we had much more quality documentation than the previous group, so more of the card's functionality and initialization could be implemented.

We were able to test the *emulexfc* driver in its current state with a Fibre Channel switch, and verify that the basic functionality was operating properly with that configuration. (as much as we expected it to, anyway.)

As it stands, the driver is able to perform all low-level functions required. The next step is to get FC Login implemented, which is a high-level protocol issue.

[1] The Emulex LP cards support two programming interfaces. We chose to use SLI-1 as it is simpler to implement, and will be easy to extend to SLI-2 when a working driver is complete.

6.2 Login

Because of holes in our documentation, we were unable to get fibre login to work. The procedure was performed according to the documentation, and we were able to get a response from the RAID array – unfortunately, the response was an error, reporting that we had essentially sent it a garbage command.

This would be an essential first step for any followup work. After this, more SCSI commands could be implemented. We started with TEST_UNIT_READY, which the Linux SCSI layer sends to all registered SCSI drives.

6.3 Stability

Fibre Channel is often chosen for its fault-tolerant nature: it has the ability to have fail-over servers and mirroring disks. Because Linux is not a fault tolerant OS, using the strengths of Linux and Fibre Channel together to build servers could provide a cost effective and stable solution.

Linux is an excellent overall choice for stability. On the right hardware, Linux has high recorded uptimes – some reported cases on the Linux Kernel mailing list claim uptimes in excess of 500 days! Because Linux is well known for its stability, our Emulex driver must meet or exceed those standards to be useful.

We were unable to test stability issues with our driver, but it's important to note, that even in our development stages a hung machine was a rare sight.

6.4 Performance

Fibre channel is often chosen because of its performance, especially in database related applications. Because of this, the code driving the fibre channel devices (the *emulexfc* driver) must be as streamlined as possible.

Because a fully functional driver was not completed, we were unable to complete performance testing. As it stands, the driver concentrates on functionality, and not performance, which results in debug code and delays to aid in development.

After stability and functionality is tested, optimizing the driver for performance would be the next step. Testing and comparing with other complete drivers, such as the Windows NT driver would be a step in the right direction – a good goal would be to meet or exceed the performance of the NT driver. Additionally, testing with database software, such as Oracle, which is available for both Linux and Windows would be a good method of comparison. Databases are very I/O intensive, and would stress the driver and hardware to its limits.

Another prime market for a RAID setup (especially with Fibre Channel) would be multimedia delivery – applications such as web based video and audio streaming are becoming more ane more common.

6.5 Other Upper Level Protocols

Farther into the future, a possible extension to this project would be into other protocols, such as IP. As previously discussed, Fibre Channel can be used for arbitrary protocols, but the LightPulse series is optimized for handling both SCSI and IP.

Using FC for IP would give you great flexibility for network topology while at the same time providing extremely good performance.

6.6 Hardware Variety

As discussed in the goals section, another important area to look into would be support for non-Intel architecture. Supporting Both 64-bit systems such as Alphas, and big-endian systems such as PowerPCs and Sparcs is important.

A good majority of the drivers in Linux are supported across the wide variety of hardware that Linux runs on. It's strongly encouraged that anyone writing drivers[2] think about cross platform issues and try to code in a platform independent manner.

[2] Especially for PCI card drivers, as many systems share the PCI bus.

Bibliography

[1] *Emulex FireFly Chipset Driver Development Documentation.* Emulex Corporation, http://www.emulex.com/.

[2] *CLARiiON Advanced Storage Solutions.* http://www.clariion.com/

[3] *Linux Homepage.* http://www.linux.org/

[4] *Free Software Foundation.* http://www.fsf.org/

[5] *GNU General Public License.* http://www.gnu.org/copyleft/gpl.html

[6] Rubini, Alessandro. *Linux Device Drivers.* Sebastopol, CA: O'Reilly & Associates, Inc., 1998.

[7] Sweetser, Frank and Anderson, Charles. *Development of a CLARiiON Fibre Channel Device Driver for Linux.* http://angus.ind.wpi.edu/fp/, Worcester, MA: Worcester Polytechnic Institute, 1998.

[8] Morano, Michael and Savela, Russ. *I/O Subsystems Technical Report.* http://angus.ind.wpi.edu/fp/docs/io-report/io-report/index.html, Worcester, MA: Worcester Polytechnic Institute, 1998.

[9] Patterson, Gibson and Katz, Randy. *RAID: High-Performance, Reliable Secondary Storage.* http://sunsite.berkeley.edu/Dienst/Repository/2.0/Body/ncstrl.ucb/CSD-93-778/postscript. University of California Berkeley, 1993.

[10] Coveney, Christopher and Kaufmann, Aviv. *RAID System Queuing Model.* Worcester, MA: Worcester Polytechnic Institute, MQP 98D415M, 1998.

[11] *Fibre Channel Linux Driver.* http://www.iol.unh.edu/consortiums/fc/fc_linux.html

[12] *Fibre Channel Association.* http://www.fibrechannel.com/

[13] *Linux Kernel Hacker's Guide.* http://www.redhat.com:8080/

Appendix A

Annotated Driver Output

The following is the raw output from the driver after you run `insmod emulexfc.o`. The hope is that the followup group will have a better understanding of the debug output and the general working of the driver due to this section. A line ending with a \ signifies a continuation. Also, some of the output has been trimmed for readability.

A.1 PCI Card Detection

The following debug output is from the lp6000_detect function (see figure 4.2). This shows the vital PCI information that has been set up for us. We also call lp6000_register which initializes the driver data structure, checks that the memory I/O region is available, and registers our interrupt handler with the kernel.

```
kernel: emulexfc: entering lp6000_detect()
kernel: emulexfc: lp6000 at PCI 10/0
kernel: emulexfc: Found 1 lp6000 cards
kernel: emulexfc: entering lp6000_register()
kernel: emulexfc-0: using irq 17, io 0xf000-0xf010, \
 memory 0xc8820000-0xc8821000, reg mem 0xc8822c00-0xc8822c10
kernel: (scsi2) <LP6000> found on PCI bus 0 slot 10 function 0
```

A.2 Card Reset

We thought about conditionally resetting the card if it were in an unknown state, but it proved to be easier (as soon as we had documentation that didn't *lie* about how to do it) to just always perform the card reset. The following debug output just verifies that the card returned success from the restart mailbox command.

```
kernel: emulexfc: entering lp6000_init()
kernel: 0000: 00000000 00000000 00c00000 00000000
kernel: emulexfc: Setting INITFF
kernel: ctl register: ffffffff
kernel: checking for STATUS_MB
kernel: 0000: 00000000 00000000 00000000 00000000
kernel: emulexfc: status reg = 00000000
kernel: emulexfc: mbox_restart returned 0x00000000
kernel: status register: 00c00000
kernel: checking for STATUS_RDY
kernel: emulexfc: card reset successful
kernel: emulexfc: success with 00000000
kernel: HSTCTL Bits 0x00000000
kernel: HSTATT bits 0x00000000
kernel: HSTSTAT: 0x00c00000
```

A.3 NVRAM Reading

Next, we read the NV parameters stored on the card, first reading the default device addresses (much like an ethernet hardware address), then reading the revision information. The LP 8000 *should* support SLI-2, but when we tried to query for SLI-2 information, it always returned information indicating otherwise. This is an area worth looking into, because SLI-2 is a more efficient interface to run the card in.

```
kernel: emulexfc: N_PORT name    = 0x10000000c9210808
kernel: emulexfc: NODE name      = 0x10000000c9210808
kernel: emulexfc: Preferred D_ID = 0x0
kernel: emulexfc: Hard AL_PA     = 0x0
kernel: Card supports SLI level: 1
```

```
kernel: BIU rev: 2
kernel: Sequence Manager rev: 0
kernel: Version 2.8.1 , S/N: 1
kernel: ENDEC+ rev: 909
kernel: FC-PH level: 0-0
kernel: POST-Kernel version 1.0.0 , S/N: 1
kernel: Operational firmware rev: 44050544
kernel: Desc: .2SD  18
```

A.4 IOCB Ring Configuration

The next step in card initialization is IOCB ring configuration. We had originally had three pairs of rings that contained 20 buffers each, but finally switched over to using one pair of 50 entires each for ease of implementation. In each pair, one ring is for sending commands, and one is for reading the responses. Multiple rings are useful for supporting multiple simultanious protocols, which we weren't interested in yet.

```
kernel: emulexfc: partitioned 50 CMD IOCBs at 0x100
kernel: emulexfc: partitioned 50 RSP IOCBs at 0x380
kernel: emulexfc: configure_rings succesful
```

A.5 Link Intitialization

The next step is to initialize the link. After this, link parameters are read from the card using the READ_LA mailbox command and printed for debugging. As you can see, there is a (too) tight loop polling the host attention register to wait for the card to respond to the mailbox command (*last message repeated 159 times*).

```
kernel: emulexfc: Link initialized
kernel: emulexfc: waiting for link attention
kernel: emulexfc: HOST_ATT_REG = 00000000
last message repeated 159 times
kernel: emulexfc: Link parameters:
kernel: emulexfc: Event Tag            = 0x1
kernel: emulexfc: FC-AL Port Bypass    = False
```

```
kernel: emulexfc: FC-AL Implicit Logout = False
kernel: emulexfc: Attention Type        = Link up.
kernel: emulexfc: AL_PA Granted:          0x1
kernel: emulexfc: LIP AL_PS             = 0x0
kernel: emulexfc: LIP Type              = 0x0
kernel: emulexfc: Topology              = Arbitrated Loop
kernel: emulexfc: LU=0, TF=0
kernel: emulexfc: buffer contains 0
last message repeated 3 times
```

A.6 Fabric Login

The next, and final, step is to perform a fabric login. This step is required, even if you are not using a fabric, for the express purpose to determine wether or not there is a fabric present. If a fabric is *not* present, instead of performing a fabric login, you then perform an N_PORT login.

```
kernel: emulexfc: Acquired FLOGI payload:
kernel: emulexfc: bdl1 contains 0x00000809
kernel: emulexfc: bdl1 contains 0x00000010
kernel: emulexfc: bdl1 contains 0x000000c0
kernel: emulexfc: bdl1 contains 0x00000080
kernel: emulexfc: Array RPI = 0x0001
kernel: emulexfc: leaving lp6000_register()
kernel: emulexfc: Registered 1 lp6000 hosts
kernel: emulexfc: leaving lp6000_detect()
kernel: emulexfc: entering lp6000_info()
kernel: emulexfc: leaving lp6000_info()
```

A.7 SCSI Layer Work

After the `init_login` function is complete, in theory you should have a link up with a fibre channel device, and be logged into that device. At this point, card initialization is complete – the rest of the work is performed by the Linux SCSI layer and interrupt handlers. The SCSI layer queries each of the drives registered with the following sequence of commands:

1. TEST_UNIT_READY

2. SCSI_INQUIRY
3. SCSI_INQUIRT_CONT

After this sequence of commands, the SCSI layer should have enough information to register the individual drives[1] as block devices. More work needs to be put into implementing SCSI commands to enable reading and writing to the block devices. Of course, login needs to be fixed first.

```
kernel: scsi2 : Emulex LP6000 SCSI on PCI bus 00 device 50 irq \
 17 mem base 0xc8820000 reg base 0xc8822c00 io base 0xf000
kernel: scsi : 3 hosts.
kernel: emulexfc: entering lp6000_queuecommand()
kernel: emulexfc: target = 0x00, lun = 0x00, cmd_len = 0x06
kernel: emulexfc: command = 0x00 0x00 0x00 0x00 0x00 0x00
kernel: emulexfc: leaving lp6000_queuecommand()
kernel: emulexfc: entering lp6000_queuecommand()
kernel: emulexfc: target = 0x01, lun = 0x00, cmd_len = 0x06
kernel: emulexfc: command = 0x00 0x00 0x00 0x00 0x00 0x00
kernel: emulexfc: leaving lp6000_queuecommand()
kernel: emulexfc: entering lp6000_queuecommand()
kernel: emulexfc: target = 0x02, lun = 0x00, cmd_len = 0x06
kernel: emulexfc: command = 0x00 0x00 0x00 0x00 0x00 0x00
kernel: emulexfc: leaving lp6000_queuecommand()
kernel: emulexfc: sending IOCB TEST_UNIT_READY command.
kernel: emulexfc: entering lp6000_iocb_command()
kernel: emulexfc: IOCB command
kernel: emulexfc: leaving lp6000_iocb_command()
kernel: emulexfc: entering lp6000_iocb_getrsp()
kernel: emulexfc: host-att-reg = 0x00000080
kernel: emulexfc: leaving lp6000_iocb_getrsp()
kernel: emulexfc: cmd is now 0x00
kernel: emulexfc: entering lp6000_queuecommand()
kernel: emulexfc: target = 0x02, lun = 0x00, cmd_len = 0x06
kernel: emulexfc: command = 0x12 0x00 0x00 0x00 0xff 0x00
kernel: emulexfc: leaving lp6000_queuecommand()
```

[1]Because of the nature of Fibre Channel, the drives that the SCSI layer registers are probably not individual physical drives, but multiple drives bound together in a RAID

```
kernel: emulexfc: sending SCSI INQUIRY command.
kernel: emulexfc: entering lp6000_iocb_command()
kernel: emulexfc: IOCB command
kernel: emulexfc: leaving lp6000_iocb_command()
kernel: emulexfc: sending SCSI INQUIRY CONT command.
kernel: emulexfc: entering lp6000_iocb_command()
kernel: emulexfc: IOCB command
kernel: emulexfc: leaving lp6000_iocb_command()
kernel: emulexfc: entering lp6000_iocb_getrsp()
kernel: emulexfc: host-att-reg = 0x00000800
kernel: emulexfc: leaving lp6000_iocb_getrsp()
kernel: emulexfc: cmd is now 0x00
kernel: scsi: unknown type 27
kernel:   Vendor: <garbage>  Model: <garbage>  Rev: <garbage>
kernel:   Type:   Unknown                      ANSI SCSI revision: 02
kernel: emulexfc: entering lp6000_queuecommand()
kernel: emulexfc: target = 0x02, lun = 0x01, cmd_len = 0x06
kernel: emulexfc: command = 0x00 0x20 0x00 0x00 0x00 0x00
kernel: emulexfc: leaving lp6000_queuecommand()

[snip same output repeated many more times]
[as the Linux SCSI layer scans all devices]
```

Appendix B

Project Continuation Notes

The following is a short document for whomever continues this project, written by Josh Huber.

B.1 Development Machine Setup

Important things to know about the development machine (fibre-penguin.wpi.edu) are CVS configuration and web server configuration.

B.1.1 CVS

The cvs repository is located at /usr/local/cvs. To check out the code, add a user for yourself on the system, and add yourself to the 'cvs' group. Then, from another machine:

```
$ cvs -d <serverpath> login
$ cvs -d <serverpath> co fp
```

Where <serverpath> in the above commands is:
:pserver:<use>@fibre-penguin.wpi.edu:/usr/local/cvs

This will create a directory called 'fp' which contains the emulexfc driver source code and makefile (see section 2)

You might want to get familiar with cvs at the point. (check out http://www.loria.fr/ molli/cvs-index.html), but basically, what you do is the following (after you have logged in and checked out the source, you needn't use the -d <serverpath> syntax any more):

```
cd <source directory>
cvs up                      # updates your source with any changes
....hack...hack...hack...
cvs commit
# cvs will now ask you for a text description of your changes.
```

There is also a `fp-doc` repository with all sorts of fibre channel, Emulex, and CLARiiON documentation. I highly recommend checking it out.

B.1.2 Apache/cvsweb

There is a web server set up, with HTML in `/var/www` (default Debian location), with project information. Also, I installed cvsweb, so that you can view the driver progress from the web page. You can also get diffs between any two versions via the web page.

It might be a good idea to move the CVS and web server off the development machine.

B.2 Files

There are really only 2 files that contain the emulexfc driver, the c file (*emulexfc.c*) and the header file (*emulexfc.h*). Pretty standard for a Linux driver.

The supplied Makefile is a total hack (tip: read it for a good laugh!), and should be fixed, so as to support building outside of a kernel tree using a `-I/path/to/kernel/includes` and the appropriate defines. Adding the required changes to `Documentation/Configure.help`, `drivers/scsi/Config.in`, and `drivers/scsi/Makefile` would also be a good idea if you were instead move the source files into the kernel tree itself. It's up to you.

B.3 The Driver and the Array

I recommend reading the conclusions chapter of our MQP report to find out where we left off. Also, take a look at the methodology and driver chapters for some slides that may help you out.

Something you're going to want to do is hook a serial cable up with the the RAID array to access configuration and view the log. Important: To access the configuration menu while in the serial console type the following sequence of characters (who thought this up, I don't know): `@$@$`. Yes, that's shift-2424. Very odd, but important to know. Without this information, you would have to resort to using the NT configuration utility, which is most inconvienent.

B.4 Additional Help

Feel free to email myself, or my partner John at:
Josh Huber <huber@mclinux.com>
John Baboval <baboval@mclinux.com>

Good luck!