DESIGN, IMPLEMENTATION AND EVALUATION OF A FIBRE CHANNEL DRIVER FOR IP ON LINUX

by

Vineet M. Abraham

B.E., Anna University, India (1997)

THESIS

Submitted to the University of New Hampshire

in partial fulfillment of

the requirements for the degree of

Master of Science

in

Computer Science

September, 1999

This thesis has been examined and approved.

Thesis Director, Dr. Robert D. Russell
Associate Professor of Computer Science

Dr. Philip J. Hatcher
Professor of Computer Science

Barry B. Reinhold
Director, InterOperability Laboratory

Date

DEDICATION

To my mother and father.

ACKNOWLEDGEMENTS

I would like to thank all my committee members for their support. Prof. Russell for guiding me through the frustrating times, Barry for first giving me the chance to work at the IOL and for the confidence he had in me and Prof. Hatcher for his constant encouragement. A big “Thank You” to all the guys who worked with me at the IOL’s Fibre Channel consortium. I would also like to thank Interphase Corporation for their support. Last but not the least, a special thanks to Chris Loveland for introducing me to the world of Linux kernel programming and also for his patience.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

DESIGN, IMPLEMENTATION AND EVALUATION OF A FIBRE CHANNEL DRIVER FOR IP ON LINUX

by

Vineet M Abraham

University of New Hampshire, September 1999

Fibre Channel, which is used for high-speed data transfers, supports several higher layer protocols including Small Computer System Interface (SCSI) and Internet Protocol (IP). Until now, SCSI has been the only widely used protocol over Fibre Channel. IP over Fibre Channel had not been successful mainly due to inadequate specification in the standards. Currently IP specifications have reached a stage where interoperable implementations are possible.

Although some support does exist for SCSI on Linux, there is no support for IP on Linux. This thesis aims at designing, developing, testing and evaluating a Fibre Channel driver for IP on Linux.

CHAPTER 1

INTRODUCTION

1.1 Overview and Related Work

In recent years the need for extremely fast data links has increased as a result of various technical developments. The rapid growth of data-intensive and high-speed networking applications continues to fuel the need for faster technologies. However, the network interconnection technologies that currently exist between computers and I/O devices are unable to run at the desired speeds. The purpose of Fibre Channel is to develop a means of quickly transferring data between computers, storage devices and other peripherals. Fibre Channel is a generic name for a set of standards developed by the American National Standards Institute (ANSI).

Currently, several companies like Compaq, Emulex, Hewlett-Packard, Interphase, Jaycor, LSI Logic, Prisa, Qlogic and Sun are involved in the production of Fibre Channel products for various platforms, the most common being Windows NT, Solaris and IRIX. One of the first protocols to be implemented over Fibre Channel was IP (Internet Protocol). But the efforts did not succeed due to poorly written specifications. The focus then shifted towards running SCSI (Small Computer System Interface) over Fibre Channel. Until recently SCSI has been the primary protocol for which Fibre Channel solutions were provided. Fibre Channel was predominantly used for communication between storage devices and servers. Now more and more companies have started developing Fibre Channel solutions for networking protocols like IP. Some of the companies that have Fibre Channel solutions for SCSI on Windows NT are Compaq, Emulex, Hewlett-Packard, Interphase, LSI Logic and Qlogic. Jaycor and Sun have SCSI solutions for Solaris while Prisa's solution runs on IRIX. Jaycor, Interphase and Prisa are the companies that have implemented IP on Fibre Channel for the operating systems as mentioned above.

1.2 Goals of this thesis

The primary goal of this thesis is to design, develop, test and evaluate a Fibre Channel driver for IP on Linux. Although a lot of work has been done on operating systems like Windows NT, Solaris and IRIX, the support for Fibre Channel on Linux has been minimal. The only known support for Fibre Channel on Linux has been the implementation of a SCSI driver for a Qlogic Fibre Channel card at the University of New Hampshire's InterOperability Lab. This thesis involved writing an IP driver for the Interphase 5526 Fibre Channel PCI card. The driver was built based on the specifications documented in [13]. The driver supports Fibre Channel Point-to-Point, Loop and Switched topologies (discussed in section 2.2).

1.3 Hardware and Software Platforms

The Interphase 5526 Fibre Channel PCI card was used for implementing the driver. The card contains a chip called "Tachyon" manufactured by Hewlett-Packard. The Tachyon chip implements the low level Fibre Channel protocol. Also present on the Interphase 5526 card is the Interphase (i)chipTPI which provides the interface between the Tachyon and the PCI (Peripheral Component Interconnect) bus. The implementation will be made using the Linux operating system, version 2.2.5. The driver will be built for both Intel and Alpha-based platforms. Testing and evaluation of the driver will be done at the University of New Hampshire's InterOperability Lab using the Finisar Gigabit Traffic Generator and Analyzer.

1.4 Organization of the Thesis

Chapter 2 gives a general introduction of Fibre Channel. It talks about the different Fibre Channel topologies and layers as well as how addressing is done in Fibre Channel. Chapter 3 talks about the pieces of hardware with which the device driver interacts and provides a functional overview. Chapter 4 gives a brief summary of the Internet Draft "IP and ARP over Fibre Channel". The driver is implemented based on this draft. Chapter 5 discusses some of the basic functions that the driver performs. Chapters 6, 7 and 8 describes the design, implementation and evaluation of the driver respectively. Chapter 9 summarizes the work done and also discusses the work that can be done in future.

CHAPTER 2

FIBRE CHANNEL

2.1 Introduction

Fibre Channel is an efficient, standard, high-speed data transport interface whose primary task is to transport data at the fastest speeds currently available with the least latency. It is a flexible, scalable method for achieving high-speed interconnection, communication, and data transfer among heterogeneous systems and peripherals, including workstations, mainframes, supercomputers, desktop computers, and storage devices. It allows many well known existing channel and networking protocols to run over the same physical interface and media. This chapter, which gives an introduction to Fibre Channel, is based on [14] and [15].

Logically, Fibre Channel is a bi-directional point-to-point serial data channel that has been structured such that it provides high performance. Physically, Fibre Channel can be an interconnection of multiple communication points, called *N_Ports*, interconnected by a switching network called a *Fabric*, a point-to-point link or a ring configuration. Fibre Channel is structured as a set of hierarchical functions as illustrated in Figure 2-1. Fibre Channel supports a number of existing protocols, such as SCSI (Small Computer System Interface) and IP (Internet Protocol). It is not a high-level protocol like SCSI, but it does contain a low-level protocol for managing link operations. Fibre Channel is not aware of, nor is it concerned with, the content of the user data being transported. Networking and I/O protocols, such as SCSI commands, are mapped to Fibre Channel constructs and encapsulated and transported within Fibre Channel frames. The main purpose of Fibre Channel is to be able to run multiple protocols in a heterogeneous environment.

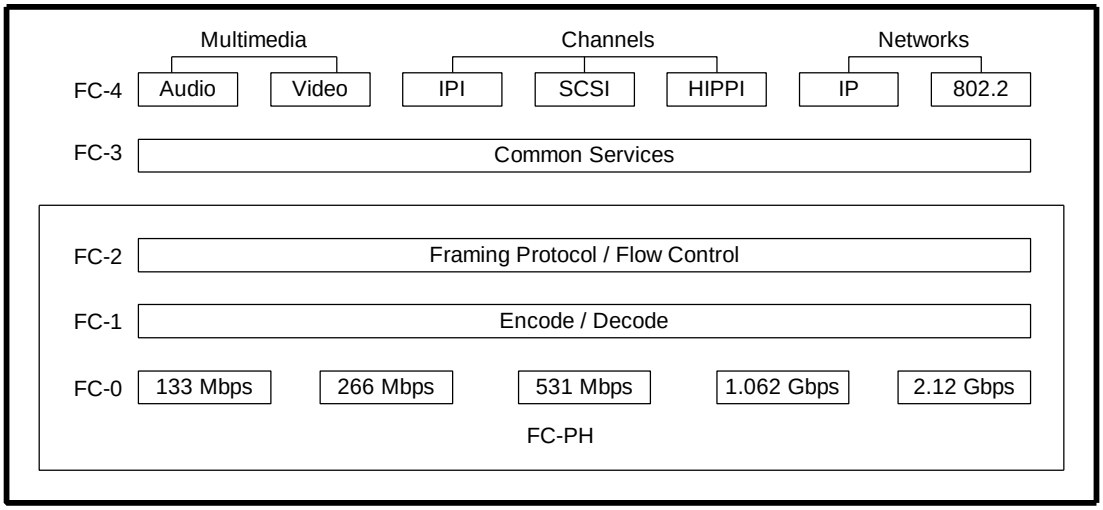

Figure 2-1 Fibre Channel layers

2.2 Fibre Channel Topologies

Fibre Channel defines three topologies, namely Point-to-Point, Arbitrated Loop and Fabric. These topologies are used to interconnect varying number of devices, called *nodes* in Fibre Channel terminology. An N_Port that can operate on an Arbitrated Loop is called an *NL_Port.* An NL_Port could be either a *public* or *private* NL_Port. A public NL_Port is one that is capable of logging into a Fabric and to communicate with devices connected to the different ports of a Fabric. On the other hand, a private NL_Port is one that does not log into the Fabric and hence can communicate only with those that are on the same Loop as it is on. A port on a Fabric switch to which N_Ports may be directly connected is called an *F_Port.* An F_Port which is capable of supporting an attached Arbitrated Loop is called an *FL_Port.*

Point-to-Point. This is the simplest of the three topologies. It consists of two and only two Fibre Channel devices connected directly together. The two devices can utilize the entire bandwidth of the link. A simple initialization is required between the two devices before communication can begin.

Arbitrated Loop. It is a cost-effective way of connecting up to 127 ports in a single network without the need for a Fabric switch. In this topology, the media is shared among the devices. Before a Loop is usable, it must be initialized so that each port obtains an Arbitrated Loop Physical Address (AL_PA). An AL_PA is a dynamically assigned value that is used by the ports to communicate.

Fabric. The Fabric topology is used to connect devices in a cross-point switched configuration. The media is not shared and more than one device can communicate at the same time. When the N_Ports are attached to a Fabric, they are required to log into the Fabric. The Fabric will then assign a unique address. Fabrics also have FL_Ports that allows attachment of an Arbitrated Loop to a Fabric.

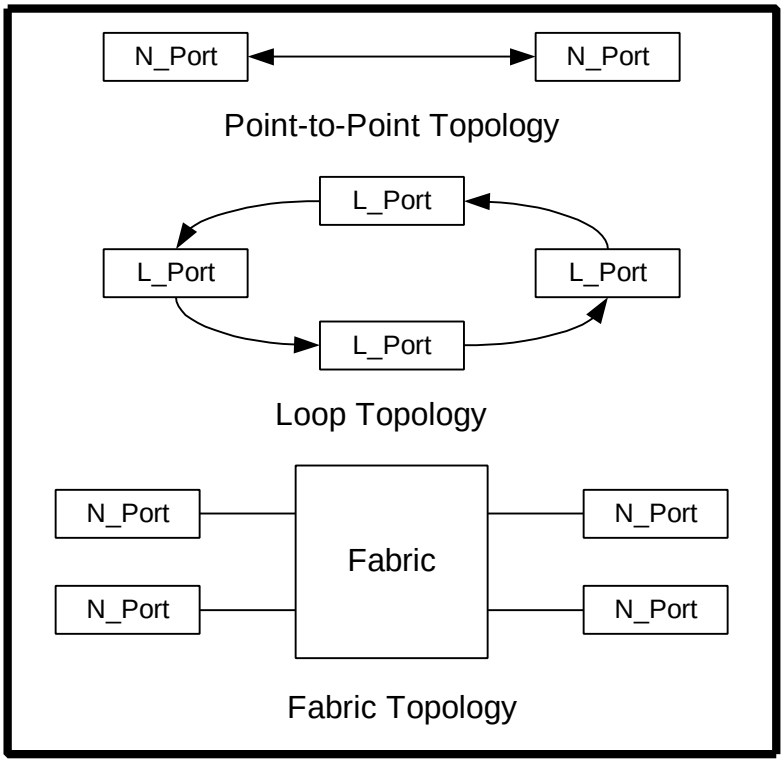

Figure 2-2 Fibre Channel Topologies

2.3 Fibre Channel Layers

As shown in Figure 2-1, Fibre Channel is structured with independent layers. The five layers of Fibre Channel define the physical and transmission rates, encoding scheme, framing protocol, common services and the upper-layer protocol interfaces. The user protocol being transported over the Fibre Channel – SCSI or IP, for example – is known as the *upper level protocol* (ULP).

2.3.1 FC-4: The Protocol Mappings Layer

This topmost level in the Fibre Channel hierarchy defines the application interfaces that can execute over Fibre Channel. Fibre Channel supports networking and channel protocols, which include SCSI, IP and IPI. Each ULP that is supported by Fibre Channel requires a separate FC-4 mapping. These mappings are specified in separate FC-4 documents. For example, the Fibre Channel protocol for SCSI, which is known as FCP, defines a Fibre Channel mapping layer that uses the services of the lowest three Fibre Channel layers to transmit SCSI command, data, and status information between a SCSI initiator and a SCSI target.

2.3.2 FC-3: The Common Services Layer

The FC-3 level defines a set of services that are common across multiple ports of a node. This layer is not well defined in the standards and is not generally implemented.

2.3.3 FC-2: The Framing Protocol Layer

This layer defines the rules for the signaling protocol and it serves as the transport mechanism of Fibre Channel. Included in this layer are the framing rules that define how data is transferred between ports. It also defines the various mechanisms used for controlling the different classes of service. To aid in the transport of data across the link, [2] defines the following building blocks: Ordered Sets, Frame, Sequence, Exchange and Protocol. *Ordered Sets* are used for handling such functions as configuration management, error recovery, frame demarcation, and signaling between two ends of a link. A *Primitive Sequence* is an Ordered Set that is transmitted and repeated continuously until a specific response is received. *Frames* are the smallest indivisible units of user data that can be sent on the Fibre Channel link. Frames can be variable in length, up to a maximum of 2148 bytes long. Each frame begins and ends with a Frame Delimiter as shown in Figure 2-3. Immediately following the four-byte *Start of Frame* (SOF) is the 24-byte Frame Header. Each frame can carry up to 2112 bytes of FC-4 payload. The payload includes zero to 64 bytes of *Optional Headers* and zero to 2048 bytes of ULP data. A Fibre Channel Optional Header allows up to 4 optional header fields – Expiration Security Header (16 bytes), Network (16 bytes), Association (32 bytes), Device (64 bytes). The four byte Cyclic Redundancy Check (CRC) precedes the four-byte *End of Frame* (EOF) delimiter.

4 bytes	24 bytes	2112 byte Data Field		4 bytes	4bytes
Start of Frame	Frame Header	64 byte Optional Header	2048 byte Payload	CRC	End of Frame

Figure 2-3 Fibre Channel Frame Format

A *Sequence* is a set of one or more related frames transmitted unidirectionally from one N_Port to another. A sequence count is used to uniquely distinguish frames within a sequence. An *Exchange* contains one or more Sequences. The Exchanges may be unidirectional or bi-directional between two N_Ports. Fibre Channel provides *Protocols* to manage its operating environment for data transfer. FC-PH specifies the following protocols: Primitive Sequence protocol, Fabric Login protocol, N_Port Login protocol, Data transfer protocol and N_Port Logout protocol. Primitive Sequence protocols are based on Primitive Sequences and are specified for Link Failure, Link Initialization, Link Reset, and Online to Offline transition. If a Fabric is present, Fabric Login protocol specifies how an N_Port interchanges parameters with the Fabric. The N_Port Login protocol specifies how an N_Port interchanges its parameters with another N_Port before performing data transfer. The Data transfer protocol describes the methods of transferring ULP data using Flow Control mechanisms in Fibre Channel. The N_Port Logout protocol is performed when an N_Port requests removal of its parameters from another N_Port. This request may be used to free up resources at the other N_Port. The Login protocols are typically performed each time the Link gets reinitialized. Some of the cases where the Link gets reinitialized are powering on of a new device, removal and reinsertion of a device, upper layer time-outs, etc.

Though the underlying hardware might provide some support for performing the above-mentioned functions, it is the responsibility of the driver software to establish and maintain connections and also to implement the protocols mentioned above.

2.3.3.1 Flow Control

Flow control is the FC-2 control process to pace the flow of frames between N_Ports and between an N_Port and the Fabric to prevent overrun at the receiver. A device can transmit frames to another device only when the other device is ready to accept them. Before the devices can send data to each other, they

must login to each other. One of the things accomplished in login is establishing credit. Credit refers to the number of frames a device can receive at a time. This value is exchanged with another device during login. After login has been performed successfully, each device knows how many frames the other can receive. After enough frames have been transmitted and credit runs out, no more frames can be transmitted until the destination device indicates it has processed one or more frames and is ready to receive new ones. Thus, no device should ever be overrun with frames. Fibre Channel uses two types of flow control, *end-to-end* and *buffer-to-buffer*.

End-to-End

End-to-End flow control is not concerned with individual links, but rather the source and destination N_Ports. When the two N_Ports log into each other, they report how many receive buffers are available for the other port. This value becomes EE_Credit. EE_Credit_CNT is set to 0 after login and increments by 1 for each frame transmitted to the other port. It is decremented upon reception of an ACK Link Control frame from that port. ACK frames can indicate the port has received and processed 1 frame, N frames, or an entire Sequence of frames.

Buffer-to-Buffer

This type of flow control deals only with the link between an N_Port and an F_Port or between two N_Ports. Each port on the link exchanges values of how many frames it is willing to receive at a time from the other port. This value becomes the other port's BB_Credit value and remains constant as long as the ports are logged in. Each port also keeps track of BB_Credit_CNT, which is initialized to 0. For each frame transmitted, BB_Credit_CNT is incremented by 1. The value is decremented by 1 for each R_RDY primitive Signal received from the other port. Transmission of an R_RDY indicates the port has processed a frame, freed a receive buffer, and is ready for one more. If BB_Credit_CNT reaches BB_Credit, the port can not transmit another frame until it receives an R_RDY.

2.3.3.2 Class of Services

Fibre Channel defines six classes of service. The Class used greatly depends on the type of data that is being transmitted. The major difference between the various Classes of service is the flow control mechanism that is used.

Class 1 is a service that provides dedicated connections. This Class of service guarantees delivery of frames in the order in which they were transmitted. Confirmation of delivery is also provided. Since the connection is dedicated, there is no need for buffer-to-buffer flow control. Thus, only end-to-end flow control is used in Class 1. Class 1 is best suited for sustained high-throughput transactions.

Class 2 is referred to as multiplex due to the fact that it is a connectionless Class of service with notification of delivery and non-delivery of frames. Since no dedicated connection needs to be established, a port can transmit frames to and receive frames from more than one N_Port. As a result, the N_Ports share the bandwidth of the links with other network traffic. Frames are not guaranteed to arrive in the order in which they were transmitted, except in the point-to-point or Loop topologies. Also, the media speeds may vary for different links that make up the path. Both buffer-to-buffer and end-to-end flow control are used in Class 2. Class 2 is more like typical LAN traffic, such as IP or FTP, where the order and timeliness of delivery is not so important.

Class 3 is identical to Class 2, except that the Frame delivery is not confirmed. It only uses buffer-to-buffer flow control. It is referred to as datagram service. Class 3 would be used when order and timeliness are not so important, and when the ULP itself handles lost frames efficiently. Class 3 is the choice for SCSI.

FC-PH also defines an optional service mode called *Intermix*. Intermix is an option of Class 1 which allows interleaving of Class 2 and Class 3 frames during an established Class 1 connection between two N_Ports. The Class 2 and Class 3 frames may or may not be destined to the same N_Port as the Class 1 frames. Both N_Ports as well as the Fabric must support Intermix for it to be used.

Class 4 is a connection-oriented service for use with a Fabric. It provides fractional bandwidth allocation of the resources of a path through a Fabric that connects two N_Ports. One N_Port will set up a Virtual Circuit (VC) by sending a request to the Fabric indicating the remote N_Port as well as quality of service parameters. The resulting Class 4 circuit will consist of two unidirectional VCs between the two N_Ports. The VCs need not be the same speed. Like a Class 1 dedicated connection, Class 4 circuits will guarantee that frames arrive in the order they were transmitted and will provide acknowledgement of delivered frames. The main difference is that an N_Port may have more than one Class 4 circuit, possibly with more than one other N_Port at the same time. In a Class 1 connection, all resources are dedicated to the two N_Ports. In Class 4, the resources are divided up into potentially many circuits. The Fabric regulates traffic and manages buffer-to-buffer flow control for each VC separately using the VC_RDY Primitive Signal. Intermixing of Class 2 and 3 frames is mandatory for devices supporting Class 4.

Class 5 involves isochronous, just-in-time service. However, it is still undefined and is not mentioned in any of the FC-PH documents.

Class 6 provides dedicated connections for reliable multicast for a Fabric topology. Basically, a device wishing to transmit frames to more than one N_Port at a time sets up a Class 1 dedicated connection with the multicast server within the Fabric at the well-known address of hex "FFFFF5". The multicast server sets up individual dedicated connections between the original N_Port and all the destination N_Ports. The multicast server is responsible for replicating and forwarding the frame to all other N_Ports in the multicast group. N_Ports become members of a multicast group by registering with the Alias Server at the well-know address of hex “FFFFF8”. The Class 6 is very similar to Class 1. Class 6 SOF delimiters are the same as used in Class 1. Also, end-to end flow control is used between the N_Ports and the multicast server.

2.3.4 FC-1: The Encode/Decode Layer

This layer, which is the transmission encode/decode layer, defines serial physical transport, timing recovery, and serial line balance. Fibre Channel uses the 8B/10B encode/decode scheme. In the 8B/10B scheme, 8-bit internal bytes are encoded and transmitted on the Fibre Channel link as 10-bit Transmission

Characters. The transmission characters are converted back into 8-bit bytes at the receiver. Two types of Transmission Characters (Data and Special) are defined. FC-PH designates special meaning to certain combinations of Transmission Characters, referred to as Ordered Sets. Ordered Sets are used to identify frame boundaries, transmit primitive function requests, and maintain proper link transmission characteristics during periods of inactivity. One special character called a *comma* is used for byte synchronization.

2.3.5 FC-0: The Physical Layer

FC-0, the lowest of the five levels, covers the physical characteristics of the interface and media, including the cables, connectors, transmitters, and receivers. The FC-0 level describes the link between two Ports. Essentially this consists of a pair of either optical fiber or electrical cables along with transmitter and receiver circuitry which work together to convert a stream of bits at one end of the link to a stream of bits at the other end.

2.4 Addressing

Fibre Channel uses a three-byte address identifier, which is dynamically assigned during Login (as mentioned in 2.2). N_Ports transmit frames from their own Source_ID (S_ID) to a Destination_ID (D_ID). Addresses in the range of hex "FFFFF0" to hex "FFFFFE" are reserved. These are called *well-known addresses* and are used for such things as the Fabric Name Server, Alias Server, or the Multicast Server. Hex "FFFFFF" is reserved for broadcast. Before Fabric Login, the N_Port's S_ID is undefined: hex "000000". In a point-to-point topology, Fabric Login will fail, and the two ports will simply chose two unique addresses. Arbitrated Loop devices still use the three byte address identifiers, but also use an Arbitrated Loop Physical Address (AL_PA). AL_PAs are one byte values dynamically assigned each time the Loop is initialized. Once the Loop is initialized and each L_Port has selected an AL_PA, public NL_Ports will attempt Fabric Login if they are connected to an FL_Port. If there is an FL_Port, the Fabric will assign the upper two bytes of the NL_Port's address identifier and usually allow the low byte to be the NL_Port's AL_PA. If not, the Loop will need to be re-initialized so the NL_Port can select the Fabric assigned AL_PA. If no Fabric exists or if an NL_Port is a private NL_Port, the upper two bytes of the

address identifier will remain '0000', and the lower byte will simply be the NL_Port's AL_PA. Also for performing the Loop Initialization, a port needs to be identified uniquely. This is accomplished using Name_Identifiers, a fixed 64-bit value. Name_Identifiers are used to uniquely identify nodes (Node_Name), a Port (Port_Name), and a Fabric (Fabric_Name). Name Identifiers are not used to route frames, but are used in mapping to upper layer protocols. Port Names are also called World Wide Names. These names could be one of the different formats like IEEE, IEEE extended, CCITT, etc. The first 4 bits of the World Wide Name specifies the Network Address Authority (NAA) to indicate the format of the name used. The next 12 bits are usually set to 0 followed by 48 bits that contain the address in the format specified by the first 4 bits.

CHAPTER 3

TACHYON AND (i)CHIP

3.1 Introduction

The two pieces of hardware that form the core of the Interphase 5526 PCI based Fibre Channel Host Bus Adapter are:

- Hewlett-Packard's Tachyon chip, and,
- Interphase's (i)chipTPI.

This chapter provides a functional overview of the above-mentioned chips. Figure 3-1 gives an idea about the organization of a Fibre Channel Adapter.

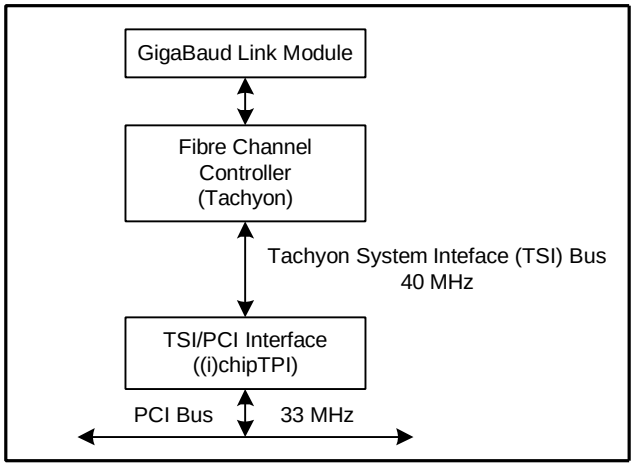

Figure 3-1 PCI Fibre Channel Adapter

3.2 Tachyon: A Gigabit Fibre Channel Protocol Chip

This section summarizes the relevant contents of the Tachyon User's Manual[11]. The Tachyon chip implements the FC-1 and FC-2 layers of the five-layer Fibre Channel standard. Some of the features of Tachyon are:

- supports Fibre Channel Class 1, 2 and 3 services
- supports Fibre Channel Arbitrated Loop (FC-AL), Point-to-Point and Fabric topologies
- automatically generates acknowledgement (ACK) frames for inbound data frames for class 1 and class 2
- handles NL_Port and N_Port initialization entirely in hardware
- manages concurrent inbound and outbound sequences
- uses a messaging queue to notify the host of all completion messages
- assists networking protocols by supporting IP checksums
- supports up to 2K-byte frame payload size for all Fibre Channel classes of service
- supports broadcast transmission and reception of frames in an Arbitrated Loop topology
- supports SCSI encapsulation over Fibre Channel

3.2.1 Tachyon Functional Overview

The interface of the Tachyon chip to the device driver is a set of registers used for initialization, configuration, and control. There is also a set of data structures that are used for sending and receiving data and for event notification. The driver and Tachyon use five host-based circular queues to pass messages and memory descriptors. The circular queues that the driver needs to maintain are:

- Outbound Command Queue (OCQ)
- High-Priority Command Queue (HPCQ)
- Inbound Message Queue (IMQ)
- Single Frame Sequence Buffer Queue (SFSBQ)
- Multi Frame Sequence buffer Queue (MFSBQ)

Each circular queue consists of four main components:

- The queue itself
- The queue length
- The producer index
- The consumer index

Each circular queue is a contiguous area in the host's virtual memory that can be thought of as an array of 0 to (n-1) entries (if there are 'n' entries). All entries in all of the queues are 32 bytes in length. Except for the IMQ, all the other queues must contain a minimum of two entries. The IMQ must have a minimum of four entries. Base addresses of the queues must be aligned on a sizeof (queue) boundary, i.e., 32 multiplied by the number of entries in the queue. The device driver creates the queues in its host's virtual memory and writes the base address and size of the queues to the corresponding registers on the Tachyon. The queues should be contiguous in physical memory. This is typically done initially when the driver gets loaded. The queues need to be set up before Tachyon can be used for transmission/reception of data.

3.2.1.1 Transmit Process Overview

For Tachyon to transmit a frame, the driver has to provide the data to Tachyon in a fixed format. Initially, the driver has to construct the Tachyon Header structure. The driver then creates the data buffers that need to be transmitted. The addresses of these structures are put into Outbound Descriptor Blocks (ODB), which in turn are present inside the OCQs. Each OCQ corresponds to a Fibre Channel Exchange that needs to be transmitted. The details of the transmit process are given below.

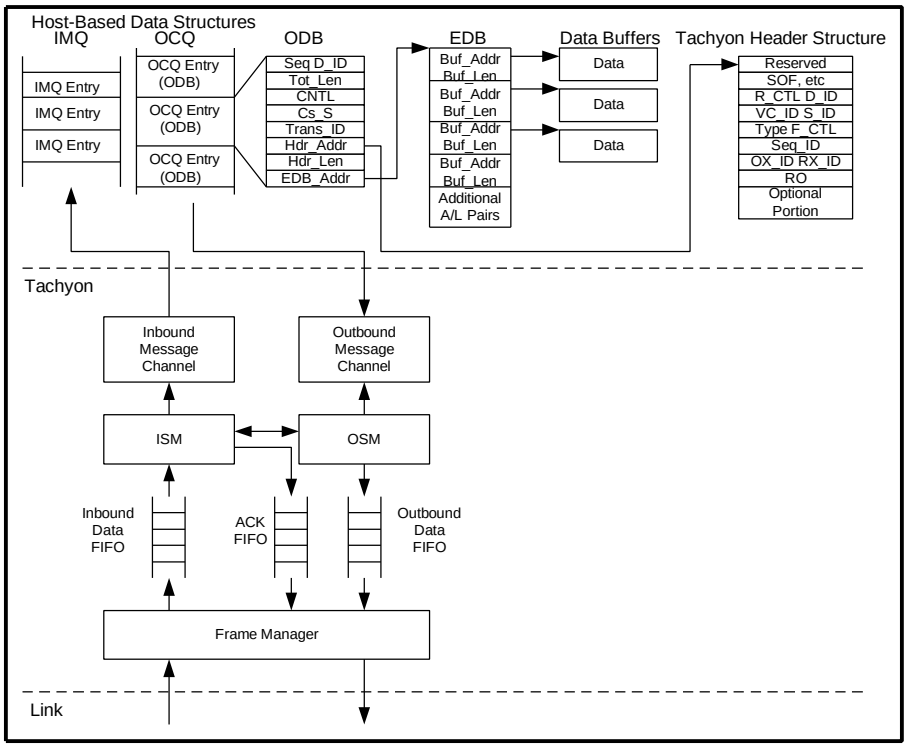

Figure 3-2 Transmit Process Overview

For transmission, the driver creates buffers of data to be transmitted. The driver stores these data buffers in its host's virtual memory. Initially the driver creates a Tachyon Header Structure that contains the Fibre Channel frame header information. This structure is also stored in the host memory. The driver then creates another data structure called the Outbound Descriptor Block (ODB) that defines the sequence of data to be transmitted. The ODB contains Fibre Channel information like maximum frame size, class of service, etc., that is obtained by the driver during the login procedures it had performed using the protocols mentioned in 2.3.3. The ODB contains a pointer (Hdr_Addr) to the Tachyon Header Structure. The ODB also contains a pointer (EDB_Addr) to the Extended Descriptor Block (EDB). The EDB contains Address/Length pairs (A/L pairs). These Address/Length pairs define the location and length of data buffers stored in host memory. Tachyon uses the Fibre Channel information in the ODB and the Tachyon Header Structure to construct Fibre Channel frame headers. The Tachyon header structure is copied by Tachyon to its internal registers so that it can be used for generating Fibre Channel headers for subsequent frames in that sequence.

Each OCQ entry consists of one ODB. The producer index of the OCQ, known as the OCQ Producer Index, points to the next available OCQ entry in which an ODB can be created. Whenever the driver creates a new ODB, it updates the OCQ Producer Index to inform Tachyon that a new valid ODB exists. If Tachyon's Outbound Sequence Manager (OSM) is not currently transmitting a sequence, the OSM performs a DMA operation to move the ODB via the Outbound Message Channel from the host memory into an internal Tachyon resource. When the OSM receives the ODB, the OSM has all the information needed to transmit a sequence.

The OSM retrieves data from the Outbound Message Channel in frame size packets for transmission. The OSM DMAs the first frame from the data buffer in host memory to the Outbound Frame FIFO. Once the entire frame is in the Outbound Frame FIFO, the OSM notifies the Frame Manager to begin transmitting the frame onto the link. When the first word of the frame is transmitted onto the link, the OSM is notified. The OSM then begins to move the second frame from host memory to the Outbound Frame FIFO. This operation continues until the entire sequence has been transmitted.

When the OSM transmits the sequence successfully, it notifies the Inbound Sequence Manager (ISM) to generate an outbound-completion message. The ISM sends this completion message to the driver as an entry in the Inbound Message Queue (IMQ). Completion messages contain information about the status of the transfer and any information needed by the driver for maintaining the queues.

3.2.1.2 Receive Process Overview

Upon receiving a frame, Tachyon puts the frame in the SFS or MFS Buffer Queue depending on if it is a single or multi-frame sequence. The buffer contains the Tachyon Header and the payload of the frame. The driver has to strip off the Tachyon Headers before it gives it to the upper layers for processing. The details of the receive process are given below.

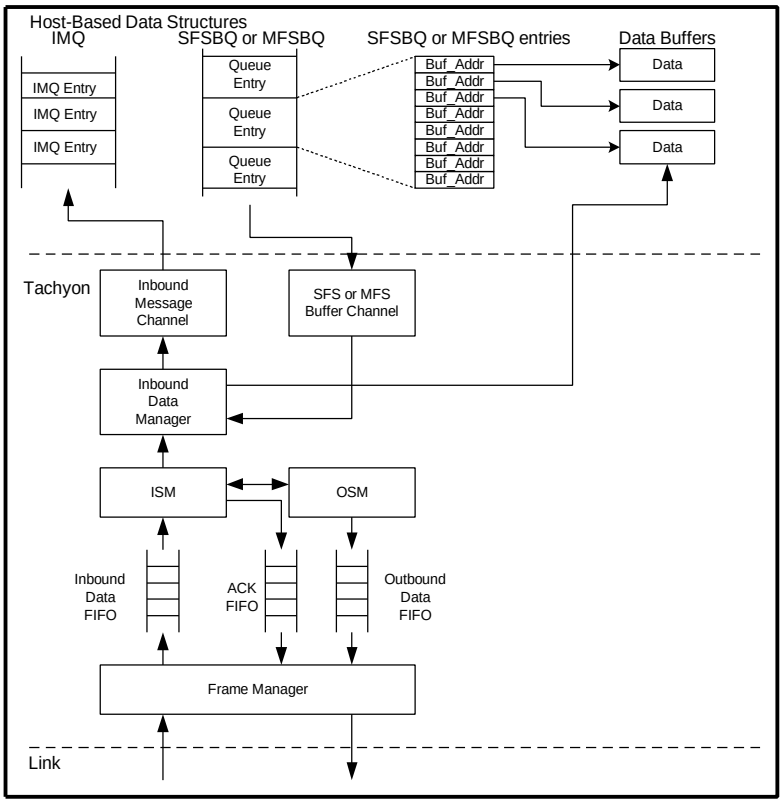

Figure 3-3 Receive Process Overview

As Tachyon receives a frame, the frame is stored in the Inbound Data FIFO while it verifies the Cyclic Redundancy Check (CRC). If the CRC fails, the frame is discarded. If the CRC passes, the ISM is notified of the received frame. The ISM then checks to see if the frame received is a frame of a single-frame sequence (SFS). If it is an SFS, Tachyon uses the next available buffer in the Single Frame Sequence Buffer Queue (SFSBQ), which was created by the driver, to store the received frame. Tachyon does this by DMAing the SFS (Fibre Channel Header and data payload) to the SFSBQ in the driver via the Inbound Data Manager and the SFS Buffer Channel. On the other hand, if the ISM determines that the received frame is part of a multi-frame sequence (MFS), Tachyon uses the next available buffer in the Multi Frame Sequence Buffer Queue (MFSBQ), which was created by the driver, to store the received frame. When the complete sequence has been received, the ISM generates an inbound-sfs-completion message or an inbound-mfs-completion message and passes the message to the driver through the Inbound Data and Inbound Message Channel as an entry in the IMQ. After Tachyon sends the completion message, Tachyon generates an interrupt to signal the host to process the received sequence.

3.3 Interphase (i)ChipTPI

The (i)chipTPI directly connects the Hewlett-Packard Tachyon Fibre Channel Controller chip to the industry standard PCI local bus. The Tachyon does not have a PCI bus interface of its own. The (i)chipTPI provides a transparent, high-performance PCI local bus interface for the Tachyon chip.

3.3.1 (i)chipTPI Functional Overview

The primary function of the (i)chipTPI is to bridge between the PCI bus and the TSI (Tachyon System Interface) bus. All PCI specific functions that are required by the PCI specifications are implemented inside the (i)chipTPI. The PCI bus interface provided by the (i)chipTPI fully supports 32-bit initiator and target modes and it includes a full set of configuration registers. The TSI bus interface connects to the Tachyon chip. The interface fully supports TSI bus master and slave cycles and manages arbitration. The (i)chipTPI also provides a flexible address translation facility to control byte-swap conversion of little-endian to big-endian data. This feature is necessary because Tachyon is big-endian. For platforms like x86 which is little-endian, the conversion to big-endian has to be made before passing the data to Tachyon. The (i)chipTPI also generates PCI interrupts depending on PCI and TSI bus conditions.

CHAPTER 4

IP AND ARP OVER FIBRE CHANNEL

This chapter summarizes the contents of [13].

4.1 Introduction

The Internet Draft "IP and ARP over Fibre Channel" specifies how IP and Address Resolution Protocol (ARP) packets are encapsulated over Fibre Channel. The document also describes mechanisms for IP address resolution.

4.2 Objective

The primary objective of [13] is to promote interoperable implementations of IP over Fibre Channel. Although the Fibre Channel standard [5] architecturally defines support for IP encapsulation and address resolution, it is inadequately specified. Two other documents that specify IP encapsulation are [9] and [10]. Each document had its own drawbacks. Since [5] prohibits broadcasts, loops are not covered. [10] has no support for Class 3 services. [9] was put together by the Fibre Channel Association, which is a industry consortium of Fibre Channel vendor companies and not a standards body. [13] is largely derived from [9].

4.3 Summary

The Internet Draft specifies how to encapsulate IPv4 and ARP packets over Fibre Channel. This specification supports all the Fibre Channel topologies, namely, loop, fabric and point-to-point. It also supports any Fibre Channel class of service. Figure 2-3 shows the structure of a Fibre Channel frame.

4.3.1 IP/ARP Encapsulation

The IP and ARP Sequences are required to carry a Network_Header optional header field that is 16 bytes long. When the Fibre Channel payload contains either an IP or ARP payload, only the first frame of the sequence will include the Fibre Channel Network_Header. There is a one-to-one mapping between an IP packet and a Fibre Channel sequence. The Maximum Transmission Unit (MTU) for IP is the maximum length of the IP packet, which is theoretically 65,535 bytes. However, [13] sets the IP MTU that can be used to 65,280. The remaining 255 bytes have been reserved for future use. As mentioned in section 2.4, Fibre Channel devices are identified by a 64-bit unique nonvolatile World Wide Name (WWN) and a 24-bit volatile address identifier (Port_ID). The first 4 bits of the WWN, which specifies the Network Address Authority (NAA), is set to "0001" to indicate that IEEE 48-bit MAC is used. The next 12 bits are set to 0, which is then followed by an IEEE 48-bit MAC address. The payload of a Fibre Channel sequence carrying an IP packet uses the format shown in Figure 4-1. Figure 4-2 shows the format when the payload is an ARP packet. Both formats use IEEE 802.2 LLC/SNAP encapsulation.

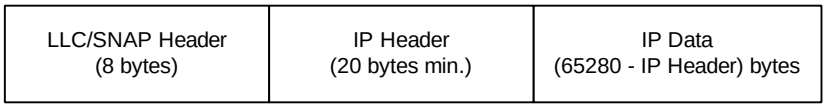

Figure 4-1 Format of Fibre Channel Sequence Payload carrying IP

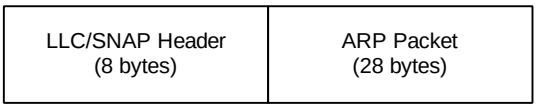

Figure 4-2 Format of Fibre Channel Sequence Payload carrying ARP

The Logical Link Control (LLC) header is 3 bytes long and consists of a 1-byte Destination Service Access Point (DSAP) field, a 1-byte Source Service Access Point (SSAP) field, and a 1-byte Control field as shown in Figure 4-3. The LLC header value is always hex 'AA-AA-03'. The LLC's DSAP and SSAP values of 0xAA indicate that an IEEE 802.2 SNAP header follows. The LLC's CTRL value of 0x03 specifies an Unnumbered Information Command PDU.

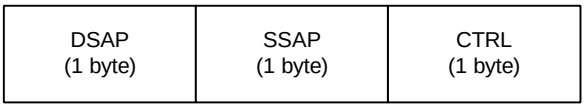

Figure 4-3 LLC Format

The Sub Network Access Protocol (SNAP) header is 5 bytes long and consists of a 3-byte Organizationally Unique Identifier (OUI) field and a 2-byte Protocol Identifier (PID) as shown in Figure 4-4.

Figure 4-4 SNAP Format

The SNAP OUI value is 0x00-00-00, which means that the PID is a routed non-OSI protocol. With the OUI value set to 0x00-00-00, a SNAP PID value of 0x08-00 indicates IP and a SNAP PID value of 0x08-06 indicates ARP.

The format of the ARP packet that is encapsulated is as shown in Figure 4-4.

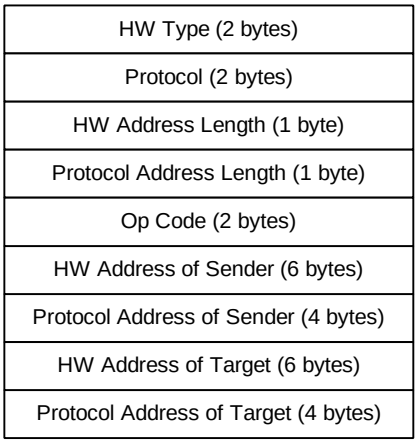

Figure 4-5 ARP Packet Format

The "HW Type" field is set to either 0x00-06 or 0x00-01. Technically, the correct HW Type value is 0x00-06 to indicate IEEE 802 networks. The support for HW Type 0x00-01 is for practical reasons. When Fibre Channel is bridged to Ethernet, some Ethernet end stations are known to cause rejections when the HW Type is set to 0x00-06. To overcome this, the specification supports both HW Types. The "Protocol" is set to 0x08-00 to indicate IP. The "HW Address Length" is set to 0x06 to indicate that there are 6 bytes in the Hardware Address. The 48-bit IEEE address in the WWN is used as the Hardware Address. The "Protocol Address Length" is set to 0x04 to indicate that there are 4 bytes in the IP address. The "Op Code" is set to either 0x00-01 for ARP Request or 0x00-02 for ARP Reply. The "HW Address of Sender" is set to the 6-byte IEEE MAC address of the sender. The "Protocol Address of Sender" is set to the 4-byte IP address of the sender. The "HW Address of the Target" is set to zero if the "Op Code" field is set to 0x00-01. Otherwise it is set to the 6-byte IEEE MAC address of the original sender of the ARP request. The "Protocol Address of Target" is set to the 4-byte IP address of the target.

4.3.2 Address Resolution

Address Resolution deals with associating IP addresses with Fibre Channel Port addresses. As explained in section 2.4, Fibre Channel devices have two types of addresses. One of them is the 64-bit static address called the World Wide Name (or World Wide Port_Name). The other address called the Port_ID is dynamically assigned. Therefore the Address Resolution mechanism needs two levels of mapping:

1. A mapping from the 4-byte IP address to the last 6 bytes of the World Wide Port_Name (i.e., the 48-bit IEEE MAC address).
2. A mapping from the 6-byte MAC address to the 3-byte Fibre Channel address identifier (Port_ID).

The first mapping is done at the operating system level while the second mapping is done in the driver. The World Wide Port_Names are static, but the Port_ID is volatile. That means that the second mapping has to be validated by the driver before use.

The first mapping, namely the mapping of the IP address to the World Wide Port_Name, is handled by the Address Resolution Protocol (ARP). It occurs when an ARP response is received for an ARP request that was sent out. This mapping is maintained by the operating system in its ARP table. When an IP packet needs to be transmitted, along with the IP packet the driver also receives the destination MAC address from the upper layers. This MAC address is provided by operating system by looking up its ARP table. Since the MAC address is the last 6 bytes of the World Wide Port_Name, the driver maps the MAC address that was received from the upper layers to a Fibre Channel Port_ID. Once this resolution has been done, the packet is ready to be transmitted by the driver. The second mapping is performed by the driver for every packet that needs to be transmitted.

CHAPTER 5

LINUX DEVICE DRIVERS

This chapter gives an introduction to writing device drivers on Linux for PCI based network interface cards. The Interphase 5526 PCI Fibre Channel card has been used as an example for explaining the various concepts. [1] has been used as the primary source for this chapter.

5.1 Introduction

When writing a device driver for a PCI based card, the functions that need to be performed by the device driver are two-fold. First, it needs to establish the interface with the underlying hardware and then with the operating system. Establishing the interface with the card requires the driver to:

- detect the PCI card,
- read/write values from/to the PCI configuration registers on the card,
- initialize registers (of Tachyon and (i)chipTPI) on the card.

Establishing the interface with the operating system involves providing functions for:

- registering the card,
- requesting an interrupt line for the card,
- handling interrupts from the card,
- managing the device,
- constructing frames using the data provided by the upper layers (like IP) and transmitting them,
- receiving frames from the card and providing the data to upper layers (like IP).

5.2 Hardware Interface

This section discusses how the PCI registers are organized and how the device driver accesses them. It then talks about ways to read/write to the registers (of Tachyon and (i)chipTPI) on the card.

5.2.1 PCI Configuration Registers

A typical PCI register layout is given in Figure 5-1. It lists the PCI registers and the offsets at which they are located.

PCI Register				Offset
Byte 3	Byte 2	Byte 1	Byte 0	
DEVICE_ID		VENDOR_ID		0x00
STATUS		COMMAND		0x04
CLASS_CODE			REVISION_ID	0x08
BIST	HEADER_TYPE	LATENCY_TIMER	CACHE_LINE_SIZE	0x0C
Memory Base Address Register				0x10
IOBASEL				0x14
IOBASEU				0x18
Reserved				0x1C
Reserved				0x20
Reserved				0x24
Reserved				0x28
Subsystem ID		Subsystem Vendor ID		0x2C
Expansion ROM Base Address				0x30
Reserved				0x34
Reserved				0x38
Max_Lat	Min_Gnt	Interrupt Pin	Interrupt Line	0x3C
Reserved				0x40-0xFF

Figure 5-1 PCI Configuration Register Map

A *bus* number, a *device* number, and a *function* number identify each peripheral. The PCI specification permits up to 256 buses. PCs normally have only one PCI bus. Each bus hosts up to 32 devices and each device can perform up to 8 functions.

As shown in Figure 5-1, the PCI devices have a 256-byte address space. The first 64 bytes are standardized, while the rest are device-dependent. Of the first 64 bytes, the registers that have been shaded are required and the others are optional. Every PCI device must contain meaningful values in the registers that are mandatory. Three PCI registers identify a device: VENDOR_ID, DEVICE_ID and CLASS_CODE. Every

PCI device puts its own values in these read-only registers. The device driver looks up these values to identify a device.

The first step taken by the driver is to check if there is a PCI bus in the system. If the motherboard supports PCI, it will then be equipped with PCI-aware firmware called the BIOS. The BIOS offers access to the device's PCI configuration registers. The BIOS is probed to see if a specific device (Fibre Channel card) of a particular vendor (Interphase) is present. If these operations are successful, then the next logical step is to allocate system resources like memory for the card. The information required to perform the above mentioned operations are obtained by accessing the PCI configuration registers on the card. Some of the kernel functions used while dealing with PCI based cards are:

- *pcibios_present()*
- *pcibios_find_class()*
- *pcibios_find_device()*
- *pcibios_read_config_byte()*
- *pcibios_read_config_word()*
- *pcibios_read_config_dword()*
- *pcibios_write_config_byte()*
- *pcibios_write_config_word()*
- *pcibios_write_config_dword()*

pcibios_present() checks to see if the computer supports PCI. If the BIOS is PCI-aware, it returns a non-zero value. It takes no arguments.

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

pcibios_find_class() requests information about the class of the device from the BIOS. It returns the bus number and device number of the particular device on the PCI bus. A return value of 0 indicates success and non-zero values indicate failure.

```
int pcibios_find_class(unsigned int class_code, unsigned short index, unsigned char* bus, unsigned char * device);
```

where:

- `class_code` is the specific class to which the device belongs (for example, the class could be Fibre Channel).
- `index` is used to identify multiple devices with the same VENDOR_ID /DEVICE_ID identifier.
- `bus` corresponds to the PCI bus number. On a single system it is conceivable to have multiple PCI buses. Normally there is only one PCI bus in a system, and `bus` is set to 0.
- `device` corresponds to the index of the device in the PCI bus.

pcibios_find_device() requests information about the device from the BIOS. It is used to check if a particular device of a particular vendor is present. It returns the bus number and device number of the particular device on the PCI bus. A return value of 0 indicates success and non-zero values indicate failure.

```
int pcibios_find_device(unsigned short VENDOR_ID, unsigned short DEVICE_ID, unsigned short index, unsigned char* bus, unsigned char* device);
```

where:

- `VENDOR_ID` is a two-byte number used to uniquely identify a vendor. This information can be obtained from the PCI register at offset hex ‘00’.
- `DEVICE_ID` is a two-byte number used to uniquely identify a device of a particular vendor. This information can be obtained from the PCI register at offset hex ‘02’.
- `index` is used to identify multiple devices with the same VENDOR_ID /DEVICE_ID identifier.
- `bus` corresponds to the PCI bus number. On a single system it is conceivable to have multiple PCI buses. Normally there is only one PCI bus in a system, and `bus` is set to 0.
- `device` corresponds to the index of the device in the PCI bus.

pcibios_read_config_byte() is used to read a byte from the configuration space of the device identified by *bus* and *device*.

```
int pcibios_read_config_byte(unsigned char bus, unsigned char device, unsigned char where, unsigned char *value);
```

where:

- `bus` and `device` are the values returned from the *pcibios_find_device()* function call.
- `where` is the offset into the PCI registers. For example, an offset of hex '3C' returns the interrupt line used by the PCI card into `value.`

Other variants of the above function are: pcibios_read_word() and pcibios_read_dword() which are used to read 2 and 4 bytes respectively.

pcibios_write_config_byte() is used to write a byte to the PCI configuration space of the device identified by *bus* and *device*.

```
pcibios_write_config_byte(unsigned char bus, unsigned char device, unsigned char where, unsigned char value);
```

where:

- `bus` and `device` are the values returned from the *pcibios_find_device()* function call.
- `where` is the offset into the PCI registers.
- `value` is the value to be written at the offset given by `where.`

Other variants of the above function are: pcibios_write_word() and pcibios_write_dword() which are used to write 2 and 4 bytes respectively.

5.2.2 Accessing the registers on the card

There are two ways of controlling I/O. One is to have a separate I/O address space and the other is memory-mapped I/O. In this section, memory mapped I/O is discussed. Several terminologies exist when

referring to PCI memory. There are terms like “shared memory”, “I/O memory” or simply “PCI memory”. To access these memory locations, kernel functions that are used are:

- *readb()*
- *readw()*
- *readl()*
- *writeb()*
- *writew()*
- *writel()*

readb() is used to read a byte from an address in memory.

```
unsigned long readb(unsigned long addr);
```

where:

- `addr` is the address from which a byte should be read.

Other variants of the above function, namely readw() and readl()are used to read 2 and 4 bytes respectively.

writeb () is used to write a byte into the address in memory that is specified.

```
writeb(unsigned char c, unsigned long addr);
```

where:

- `c` is the byte value that should be written into the address specified.
- `addr` is the address to which a byte should be written.

Other variants of the above function, namely writew() and writel() are used to write 2 and 4 bytes respectively.

The type of I/O that a card performs can be determined by looking at /proc/pci. A sample listing of /proc/pci is shown below. The information about two devices on a PCI bus is shown below.

```
Bus  0, device  14, function  0:
Network controller: Compaq Unknown device (rev 1).
Vendor id=e11. Device id=a0ec.
```

```
Medium devsel. Fast back-to-back capable. IRQ 3.  Master Capable.
Latency=64.
I/O at 0xf000 [0xf001].
I/O at 0xf400 [0xf401].
Non-prefetchable 32 bit memory at 0xfedfec00 [0xfedfec00].

Bus  0, device  15, function  0:
Fiber Channel: Unknown vendor Unknown device (rev 0).
Vendor id=107e. Device id=4.
Slow devsel.  Fast back-to-back capable.  BIST capable.  IRQ 9.
Master Capable.  Latency=64.
Non-prefetchable 32 bit memory at 0xfedfe800 [0xfedfe800].
```

The listing basically interprets the various fields in the PCI registers. The first device, namely device 14, has a separate I/O address space as well as memory mapped I/O. The second device, namely device 15, does only memory mapped I/O. This is indicated by the absence of a separate I/O address space in the listing. The memory listed in the listing cannot be used directly. This is an arbitrary physical address that needs to be mapped to kernel address space. This needs to be done when the kernel wants to access the high addresses directly. The kernel function that is used to remap the address is called *ioremap().* For example, for device 15 whose listing is given above, the function *ioremap()* may be used as follows:

```
char * baseptr = ioremap(0xfedfe800, size_of_memory_needed);
```

This is done as part of the initialization of the driver for the card and is done typically when the driver gets loaded. So, for writing a byte value, say, hex ‘10’ into the offset 20 of the area, the following is done:

```
writeb(0x10, baseptr + 20);
```

When the driver is unloaded, memory is umapped as shown below:

```
iounmap(baseptr);
```

Thus by using *readb/writeb* and similar functions, we can read/write to the registers on a PCI card from a driver.

5.3 Operating System Interface

Each device is represented in the operating system by a `struct device.` The `device` structure is the principal data structure used by network drivers. `struct device` can be conceptually divided into two parts: "visible" and "invisible". The visible part of each device is made up of fields most of which are explicitly assigned in the driver. The hidden fields comprise several additional fields that are assigned at device initialization. It also has pointers to functions that are part of the interface to the kernel. Most of the functions are implemented in the driver itself. If the functionality of a function is common for all devices of a particular technology, then those functions are included as part of the kernel itself. These function pointers are filled in typically when the device registers itself with the operating system.

5.3.1 Registering the card

Functions for registering the card are generally built into the kernel. If the driver is written for a card that is based on older technologies like Ethernet or Token Ring, then the existing functions can be used for registering the device. For technologies (like Fibre Channel) for which support is not present in the kernel, the functions will have to be added to the networking code of the kernel. Registration of devices is done when the driver gets loaded. The devices get unregistered when the driver gets unloaded. The two functions that need to be implemented are *register_device()* and *unregister_device()*. During registration, the device is assigned a unique name, say *fc0*, and the operating system adds the device to its list of networking devices. Also, the various function pointers and parameters, like the MTU of the device, size of its address field and size of the hardware header for the device, are specified during registration.

5.3.2 Requesting an IRQ

Once the device has registered itself, the next step is to request an interrupt line for the card. Interrupts are an asynchronous means by which the device tells the device driver that it ready to be acted upon. Whenever an event occurs that requires the driver to perform some action, the card sends an interrupt on the IRQ line. On seeing the interrupt from the card, the driver is required to perform appropriate action. The kernel function used to request an IRQ is *request_irq().* The interface provided for the function *request_irq()* is shown below:

```
int request_irq(unsigned int pci_irq_line, void (*handler) (int,
void*, struct pt_regs*), unsigned long flags, const char
*card_name, void *dev_id);
```

where:

- `pci_irq_line` is the interrupt number retrieved from the PCI registers using the function call `pcibios_read_config_byte(bus, device, 0x3C, &pci_int_line)` as explained in section 5.2.1.
- The second parameter is the pointer to the function that will handle interrupts from the card. The interrupt handler gets installed at this point.
- `flags` are related to interrupt management. For example, the flag could be set to `SA_SHIRQ` to show that the interrupt number can be shared with other devices.
- `card_name` is a string passed to *request_irq()* to show the owner of the interrupt when displayed by /proc/interrupts. The string could be the device name itself (say *fc0*) or a string like "Interphase 5526 Fibre Channel HBA".
- `dev_id` is used for shared interrupt lines to uniquely identify a device and is set to NULL if the interrupts are not shared.

The following is a sample listing of /proc/interrupts.

```
 0:    1867787          XT-PIC  timer
```

```
 1:       8138          XT-PIC  keyboard
 2:          0          XT-PIC  cascade
11:      14458          XT-PIC  eth0
12:      32539          XT-PIC  PS/2 Mouse
13:          1          XT-PIC  fpu
14:     117963          XT-PIC  ide0
15:          4          XT-PIC  ide1
```

On executing the following piece of code in a driver,

```
pci_irq_line = 9;
card_name = "Interphase 5526 Fibre Channel HBA";
request_irq
      (pci_irq_line,&interrupt_handler,SA_SHIRQ,card_name,dev);
```

the listing of /proc/interrupts changes as shown below. Notice that the interrupt number 9 has been associated with the “Interphase 5526 Fibre Channel HBA”.

```
 0:    1867787          XT-PIC  timer
 1:       8138          XT-PIC  keyboard
 2:          0          XT-PIC  cascade
 9:         31          XT-PIC  Interphase 5526 Fibre Channel HBA
11:      14458          XT-PIC  eth0
12:      32539          XT-PIC  PS/2 Mouse
13:          1          XT-PIC  fpu
14:     117963          XT-PIC  ide0
15:          4          XT-PIC  ide1
```

At this stage, the device has an IRQ and its interrupt handler has been installed.

5.3.3 Handling Interrupts

The card under various circumstances sends interrupts. For example, the card might send interrupts in conditions like “received_new_packet”, “transmission_of_packet_complete”, etc. Appropriate bits in the *status_register* of the card are set to reflect what caused the interrupt. When the driver receives an interrupt, the interrupt handler that was installed while requesting the IRQ gets invoked. In the interrupt handler, the driver checks the *status_register* of the card to find out the cause for the interrupt. It then performs the appropriate action in the interrupt handler. A sample *status_register* is shown in Figure 5-2.

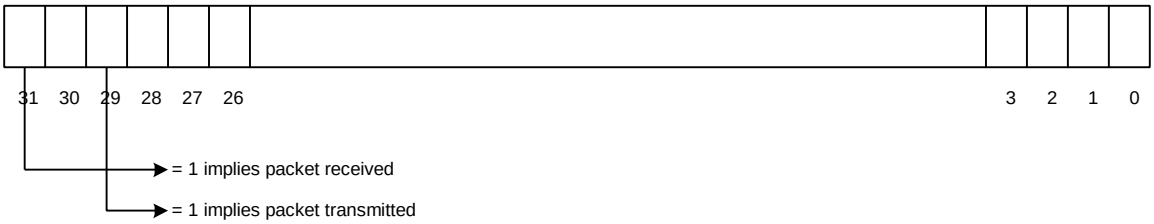

Figure 5-2 Example of a Register

While handling interrupts, some devices may require that a certain bit on the interface board be cleared to prevent other interrupts from being generated while an interrupt is being processed. But generally devices will not generate additional interrupts if there is an interrupt pending which will be reflected by the *interrrupt_pending* bit. The *interrupt_pending* bit gets cleared after the interrupt gets serviced. A simple interrupt handler might then look like this:

```
void interrupt_handler(int irq, void *dev_id, struct pt_regs *regs)
{
      switch(readb(status_register_in_card))
      {
            case PACKET_RECEIVED:
                  .
                  .
                  break;
            case PACKET_TRANSMITTED:
                  .
```

```
                .
                break;
            .
            .
        }
        writeb(interrupt_register_in card, interrupt_pending = FALSE);
}
```

5.3.4 Device Methods

The `device` structure has pointers to device methods that are part of the interface to the operating system. Each network device needs to specify these functions so that they can be acted upon. Device methods for a network interface fall into two categories: fundamental and optional. Fundamental methods are those that are needed to be able to use the interface. Optional methods are not strictly required and they implement more advanced functionalities. Some of the fundamental methods are:

```
int (*open)(struct device *dev);
```

This method *open*s the interface. Before a network device can be used, it should be *opened*. This is done when *ifconfig* activates it and associates an IP address with it.

```
int (*stop)(struct device *dev);
```

This method *stops* the interface. An interface is stopped when it is brought down. All operations that were performed as part of *open* should be reversed.

```
int (*hard_start_xmit)(struct sk_buff *skb, struct device *dev);
```

This method requests the transmission of a packet. The packet is contained in a socket buffer (`sk_buff`) structure.

5.3.5 Packet Transmission and Reception

When the kernel has a packet that needs to be transmitted, it calls the method *hard_start_xmit*. Each packet that is handled by the kernel is contained in a socket buffer structure called `struct sk_buff`, the definition of which is found in `<linux/skbuff.h>`. The buffer that is passed to *hard_start_xmit* is the actual physical data that needs to be transmitted. It has all the headers in place and is given to the driver to be transmitted across the link. After transmission the driver updates the appropriate statistics.

When a packet is received, the driver copies it into a `sk_buff`. The driver then sets the protocol field in the `sk_buff` to that of the packet received and passes it to the upper layers for processing. The driver does this by calling the kernel function *netif_rx()* with the `sk_buff` structure as its parameter. The driver then updates the appropriate statistics.

CHAPTER 6

DESIGN OF THE DRIVER

6.1 Introduction

As explained in Chapter 5, a device driver has two parts – an interface to the operating system, and, an interface to the card for which the driver is being written. The interface to the operating system is established after the operating system detects the card successfully. The interface to the card involves setting up the registers on the Tachyon and the Interphase (i)chipTPI chips. Once the interfaces have been set up, the Fibre Channel protocol, as described by the Internet Draft, can be implemented. As mentioned earlier, the Tachyon Fibre Channel controller performs the functionality up to the FC-2 layer. All other Fibre Channel functions like performing login procedures, error control mechanisms and node-validation mechanisms have to be implemented in the driver.

6.2 Functions of the Driver

6.2.1 Operating System Interface

When the system boots up, the cards are detected and chained together in a list of networking devices. For this to happen, the card should be registered as a network device. Prior to registration, the device should select a unique name, say *fc0*, for itself. During registration, it should specify certain parameters like its MTU, size of its frame header, size of its address field and the maximum queue size needed at the IP level. Once registration is complete, the device should be part of the operating system's list of networking devices. The network code of the Linux Operating System does not have native support for Fibre Channel. Currently, it supports technologies like Ethernet, FDDI and Token Ring. So functions to register a Fibre

Channel network card have to be added to the networking code in the Linux kernel. As part of the initialization process the driver also has to be able to detect multiple Fibre Channel cards of the same type. The driver then uses the existing operating system interface to request an Interrupt line for each device. Interrupts are the means by which the device communicates with the driver. While requesting an interrupt line, the driver also provides a function that will be called each time an interrupt occurs. As part of the initialization procedures, the driver also provides functions that can be called by operating system when it needs to open/close the device, transmit a packet through the device or get statistics about the device.

6.2.2 Hardware Interface

As mentioned in Chapter 3, the two pieces of hardware that the driver has to interact with are the Tachyon Fibre Channel controller and the Interphase (i)chipTPI.

At system boot time, the driver needs to build the queues for Tachyon (discussed in chapter 3) in the host's virtual memory and write the values of the size and location of the queues to the registers on the Tachyon. All entries in each of the queues are 32 bytes in length. Except for the IMQ, all the other queues must contain a minimum of two entries. The IMQ must have a minimum of four entries. Base addresses in host memory of the queues must be aligned on a sizeof (queue) boundary, i.e., 32 multiplied by the number of entries in the queue. The complexity of this task is compounded by the fact that most of these registers on the Tachyon are Write-only. The queues that need to be built are the OCQ, IMQ, SFSBQ and the MFSBQ. HPCQ, which is used for passing high priority commands, is not needed as all commands are treated with equal priority in this implementation of the driver. The MFSBQ is used for holding Multi-frame Sequences. If a frame that is larger than 2048 (which is the maximum number of bytes that Tachyon can send/receive in one frame) is received, Tachyon transfers it into the MFSBQ in chunks of 2048 bytes each. The driver then has to coalesce the 2048-byte buffers to create one large frame and pass it to the upper layers for processing. To make this process of coalescing buffers efficient, the MFSBQ is designed such that it occupies a huge chuck of contiguous physical memory. This minimizes the number of *memcpy()*s that are needed while coalescing the buffers.

6.2.3 Login/Logout Protocols

Tachyon can be programmed such that it generates an interrupt after a frame is transmitted. Generating an interrupt for each frame is inefficient. So frames are transmitted with the interrupt bit set only if the frame is a Fibre Channel Extended Link Service (ELS) frame. This helps the driver to map the responses that it receives to the ELS frames that were transmitted. This is essential for managing the Login/Logout protocols (as mentioned in 2.3.3). Each frame that is transmitted has a field called the *transaction_id* in the ODB structure (Figure 3-2). When a frame has been successfully transmitted, Tachyon generates an interrupt (if enabled) and returns the transaction_id in the IMQ. The driver can use the transaction_id to keep track of frames that were sent out.

6.2.4 Port Discovery

Before an ARP request or an IP frame can be transmitted, the destination device should be logged in with the source. In Fibre Channel, login is performed by transmitting a frame called the PLOGI (Port LOGIn). The source and destination devices are considered to have logged into each other if one of the device transits a PLOGI and the recipient of the PLOGI transmits a PLOGI ACC (PLOGI Accept) in response to the PLOGI. The frame formats of these frames are defined in [2]. For the device to detect the presence of other devices, it needs to go through a process of *Port Discovery*.

6.2.4.1 Private Loop

Port Discovery can be performed in several different ways depending on the topology. In a simple Arbitrated Loop (with no FL_Port, i.e., in a non-switched environment), the device can poll for each valid AL_PA on the Loop. If it detects that there is a device with a particular AL_PA on the Loop, the device can go ahead and transmit a PLOGI to that device and get logged in. The Port Discovery process may be performed each time the Link gets reset.

6.2.4.2 Public Loop

In a switched environment, the discovery process is a bit simpler. But the device needs to transmit frames of different types to initially log into the Fabric (section 2.2). Fabrics are required to provide various

services at specialized addresses. The most important ones are the *Directory Server* at address hex "FFFFFC", *Fabric Controller* at address hex "FFFFFD" and *Fabric F_Port* at address hex "FFFFFE". When the Link comes up, if a device detects that an FL_Port is present on the Loop, the device should then transmit a FLOGI (Fabric LOGIn) to the Fabric F_Port at address hex "FFFFFE". When the device receives an ACC to the FLOGI, the device is considered to have logged into the Fabric. The device can then go ahead and transmit a PLOGI to the Directory Server at address hex "FFFFFC". The Directory Server maintains information about all the devices that have logged into the Fabric. After logging into the Directory Server, any device can probe the Directory Server for information about the other devices that are logged in to the Fabric. Once the result of the probe to the Directory Server is received, the device can then perform PLOGI to those devices. The target devices may be connected to different ports on the Fabric. Some of them might be on the same local loop as the source device and some of them might be connected to remote ports. If the local Loop gets interrupted, the source device can perform the Port discovery procedure as mentioned above for the switched environment and reestablish login. There is the case where the state of the remote devices can change. The source device needs to know whenever the state of the Fabric changes. This can be accomplished by performing a *State Change Registration* (SCR) with the Fabric Controller at address hex "FFFFFD". Once SCR has been transmitted and it has been accepted by the Fabric Controller, then the device that transmitted the SCR will be notified if there is any change in the state of the Fabric. The recipient device can then perform the login validation with the devices whose states have changed.

CHAPTER 7

IMPLEMENTATION OF THE DRIVER

7.1 Address Maps

The (i)chipTPI can access all components on the board through memory-mapped I/O. The (i)chipTPI requires 1KB of memory space. The base address is written into the PCI Configuration Memory Base Address register (Figure 5-1) as part of the initialization process. The driver should retrieve this base address from the PCI Configuration space and then map it into a virtual memory address (as explained in section 5.2.2). The map of the memory space looks like this:

Function	Start Address	End Address
(i)chip registers	0x000	0x0FF
Repeat of (i)chip registers	0x100	0x1FF
Tachyon	0x200	0x3FF

Figure 7-1 Memory Space Map

The actual address in the host system of the driver would be the base address in virtual memory that is returned by *ioremap()* (section 5.2.2) plus the addresses shown in Figure 7-1.

7.2 Register Structure

This section lists the registers relevant to the driver for the Tachyon and (i)chipTPI chips as described in the corresponding user manuals ([11] and [12]). All registers are 32-bit registers.

7.2.1 Tachyon Registers

Tachyon has three different sets of registers: Queue registers, Frame Manager registers and Tachyon registers. Each queue has a set of 5 registers.

Registers for the Outbound Command Queue (OCQ):

- OCQ Base Address register - stores the base address of the OCQ.
- OCQ Length register - stores the number of entries in the OCQ.
- OCQ Producer Index register - points to the next empty entry in the OCQ that the driver can use to provide data to Tachyon.
- OCQ Consumer Index Address register - points to the next OCQ entry that Tachyon will process.
- Host's Copy of Tachyon's OCQ Consumer Index - this resides in host memory.

Registers for the Inbound Message Queue (IMQ):

- IMQ Base Address register - stores the base address of the IMQ.
- IMQ Length register - stores the number of entries in the OCQ.
- IMQ Consumer Index register - points to the next completion message that needs to be processed by the driver.
- IMQ Producer Index Address register - points to the next empty entry that Tachyon can use to post a completion message.
- Host's Copy of Tachyon's IMQ Producer Index - resides in host memory.

Single Frame Sequence Buffer Queue (SFSBQ):

- SFSBQ Base Address register - stores the base address of the SFSBQ.
- SFSBQ Length register - stores the number of entries in the SFSBQ.
- SFSBQ Producer Index register - points to the current end of the SFSBQ.
- SFSBQ Consumer Index register - used by Tachyon to store incoming SFS.
- SFS Buffer Length register - stores the length of each SFS buffer.

Multi Frame Sequence Buffer Queue (MFSBQ):

- MFSBQ Base Address register - stores the base address of the MFSBQ.

- MFSBQ Length register - stores the number of entries in the MFSBQ.
- MFSBQ Producer Index register - points to the current end of the MFSBQ.
- MFSBQ Consumer Index register - used by Tachyon to store incoming MFS.
- MFS Buffer Length register - stores the length of each MFS buffer.

The Tachyon registers include:

- Tachyon Configuration register
- Tachyon Control register
- Tachyon Status register

The Frame Manager registers include:

- Frame Manager Configuration register
- Frame Manager Control register
- Frame Manager Status register
- Frame Manager Timeout register
- Frame Manager World Wide Name High register
- Frame Manager World Wide Name Low register
- Frame Manager Received AL_PA register

7.2.2 (i)chipTPI Registers

The (i)TPI consists of several registers, the most important of which are:

- Hardware Control Register
- Hardware Status Register
- Hardware Address Translate Base Register

7.3 Programming (i)chipTPI

The first step in using the (i)chipTPI is to reset it. This is performed by asserting the reset bit on the Hardware Control Register and then de-asserting it. The Hardware Control Register also has bits that enable or disable interrupts that come from Tachyon. The bits that correspond to the Tachyon interrupts

should be enabled. Each time Tachyon sends an interrupt, the Interrupt Latch on the Interphase (i)chipTPI gets set. The bit corresponding to the Interrupt Latch is present on Hardware Status Register. The Interrupt Latch should be cleared each time an interrupt gets serviced. If the latch is not cleared, further interrupts from Tachyon will not be passed on to the driver. The Tachyon is a Big-Endian device while the Interphase (i)chipTPI is natively Little-Endian. By default, the Interphase (i)chipTPI performs a Little-Endian to Big-Endian conversion before handing out the data to Tachyon. So the bit that enables the byte-swap capability of the (i)chipTPI should be asserted in the Hardware Address Translate Base register.

7.4 Reading the NOVRAM

The World Wide Name of the device is stored in a Non-Volatile RAM (NOVRAM) on the card. The NOVRAM stores 1024 bits. The value that is of immediate concern to the driver is the World Wide Name that is stored in the NOVRAM. The NOVRAM is read and the World Wide Name is retrieved. The retrieved value then needs to be written to the Frame Manager World Wide Name High and Low registers as described in section 7.4.3. The driver then sets the MAC address of the device to the last 6 bytes of the World Wide Name that was read from the NOVRAM. The base address of the NOVRAM is the same as the address of the Hardware Control register of the (i)chipTPI.

7.5 Programming Tachyon

Before Tachyon can be used, the registers on Tachyon must be programmed correctly. The first step is to perform a "software reset" on the Tachyon Configuration register. This initializes the registers to their default power-on values. Tachyon is taken to an "offline" state before programming its registers. The following section describes how each of the other relevant registers should be programmed.

7.5.1 Taking Tachyon Offline

Tachyon must be in the "offline" state (power-up state) before the registers of Tachyon are programmed. To take Tachyon reliably into the "offline" state, the following needs to be performed.

If Tachyon is on a Loop and the Loop is up, then,

- Clear the Frame Manager Configuration register
- Write the "Initialize" command into the Frame Manager Control register
- Write the "Offline" command into the Frame Manager Control register.

If Tachyon is configured for a Loop topology and if the Loop is down, then,

- Write the "Host Control" command into the Frame Manager Control register
- Write the "Offline" command into the Frame Manager Control register.
- Write the "Exit Host Control" command into the Frame Manager Control register.

Once Tachyon has been programmed to the "Offline" state, the other registers can be written into.

7.5.2 Building the Queues

As mentioned in section 6.2.2, at least four of the five queues need to be built by the driver at system boot time. Each queue should be contiguous in host's physical memory. Each queue entry is 32 bytes in length. The base addresses of the queues should be aligned on sizeof (queue) boundary, i.e., 32 multiplied by the number of entries of the queue. This is achieved by using the kernel function *__get_dma_pages()* which returns contiguous parts of physical memory on the host system that can be used for Direct Memory Access (DMA).

__get_dma_pages() returns a contiguous part of host memory that can be used for DMA.

```
unsigned long __get_dma_pages(int priority, unsigned long order)
```

where:

- `priority` could be flags like `GFP_KERNEL` or `GFP_ATOMIC`. The former indicates that the allocation is performed in kernel space and the later is used when the allocation is done outside the context of the process (like in interrupt handlers).
- `order` is the power of two of the number of pages that is being requested. Current versions of Linux restricts the value to be less than or equal to 5 which means that the maximum amount of contiguous memory that can be allocated is 256KB on the Alpha and 128KB on the Intel (Page size is 8KB on Alpha and 4KB on Intel).

The OCQ, IMQ, SFSBQ and MFSBQ are allocated using the above function call. As described in section 3.2.1, each queue has four components. The queue itself, the queue length, the producer index and the consumer index. The base address of each queues returned by the __get_dma_pages() function is written to the Base Address register corresponding to that queue. The number of entries present in each queue is written to the Length register of the corresponding queue.

For the OCQ, the driver on the host system is the producer and Tachyon is the consumer. The driver, as the producer, creates ODBs and copies them into the OCQ. After filling in each OCQ entry, the driver increments the OCQ Producer Index register. The producer index points to the next empty entry that the driver can use to create ODBs. Tachyon, being the consumer, processes the ODB and then increments the OCQ Consumer Index. The OCQ Consumer Index resides in host memory. The address in host memory where the OCQ Consumer Index should reside is written to the OCQ Consumer Index Address register.

For the IMQ, the Tachyon is the producer and the driver on the host system is the consumer. Tachyon, as the producer, fills in IMQ entries. IMQ entries are used up whenever Tachyon posts a completion message. After filling in each IMQ entry, the Tachyon increments the IMQ Producer Index register. The producer index points to the next empty entry that Tachyon can use to post completion messages. The driver, being the consumer, processes the completion message in the IMQ and then increments the IMQ Consumer Index. The IMQ Producer Index resides in host memory. The address in host memory where the IMQ Producer Index should reside is written to the IMQ Producer Index Address register.

For the SFSBQ and MFSBQ, the driver on the host system is the producer and Tachyon is the consumer. The driver, as the producer, fills in entries with pointers to empty data buffers. These addresses are the addresses of buffers into which Tachyon will DMA a received Single-Frame or Multi-Frame Sequence. The SFSBQ and MFSBQ, like all the other queues has 32-byte entries in it. Each 32-byte entry holds eight 32-bit addresses. The driver has to allocate 8*(number_of_sfsbq_entry or number_of_mfsbq_entry) buffers each of size sfsbq_buffer_length or mfsbq_buffer_length and write the addresses of these buffers into the page allocated for the queue. After the driver writes in 8 addresses, it increments the SFSBQ or MFSBQ

Producer Index register. The Producer Index points to the current end of queue. At initialization time, the driver fills in all the entries and updates the corresponding Producer Index. The size of the buffers that are allocated is written to the corresponding Buffer Length register. Tachyon, as the consumer, uses the pointers into the queue to store incoming SFS or MFS. When Tachyon completes storing an SFS or an MFS, it writes the index of the SFSBQ or MFSBQ entry that is currently being used to store the SFS or the MFS into the SFSBQ or MFSBQ Consumer Index register.

Once the queue registers on Tachyon are initialized with the appropriate values, Tachyon DMAs data directly to the addresses that were handed down to Tachyon. So the addresses that are handed to Tachyon should not be virtual memory addresses. Instead they must correspond to the real physical address (bus address) on the system. Also, while retrieving data from address that were handed to Tachyon, the reverse function, i.e., conversion from bus address to virtual memory address, should be performed. This is achieved by using the kernel functions:

- *bus_to_virt()*
- *virt_to_bus()*

bus_to_virt() converts a bus address to a virtual address in host memory.

```
unsigned long * bus_to_virt(unsigned long addr);
```

where:

`addr` is the bus address whose virtual address needs to be found.

virt_to_bus() converts a virtual address in host memory to a bus address.

```
unsigned long virt_to_bus(unsigned long *addr);
```

where:

`addr` is the virtual address whose bus address needs to be found.

While passing values to Tachyon or while retrieving data given by Tachyon, it should be kept in mind that Tachyon expects the data to be in network byte order (big-endian). Appropriate conversions are made using the functions:

- *ntohs()*
- *ntohl()*
- *htons()*
- *htonl()*

ntohs() converts a 2-byte value from the network byte-ordering to the byte-ordering used by the host.

```
unsigned short ntohs(unsigned short value);
```

where:

`value` is the 2-byte number whose byte-ordering needs to be changed.

Similarly, ntohl()converts a 4-byte value from the network byte-ordering to the byte-ordering used by the host.

htons() converts a 2-byte value from the byte-order used by the host to the network byte-ordering.

```
unsigned short htons(unsigned short value);
```

where:

`value` is the 2-byte number whose byte-ordering needs to be changed.

Similarly, htonl()converts a 4-byte value from the byte-order used by the host to the network byte-ordering.

7.5.3 Configuring Frame Manager and Tachyon Registers

Once the queue registers have been programmed, the Frame Manager and Tachyon registers need to be set to the appropriate values. The Frame Manager Configuration register is configured to select an AL_PA in the desired phase of Loop Initialization. The Frame Manager Timeout registers are then programmed with the values that should be used for E_D_TOV (Error Detect Time Out) and R_T_TOV (Receiver Transmitter Time Out). These timers and their use are specified by the Fibre Channel standards [6] and [7]. The World Wide Name retrieved from the NOVRAM is then written to the Frame Manager World Wide Name High and Low registers. Once the above mentioned registers have been programmed, Tachyon is "initialized" by writing the "Initialize" command into the Frame Manager Control register. Tachyon uses the values written into its registers to initialize the Loop. When the Loop comes up, Tachyon sends an interrupt to the driver informing it that the Loop has been initialized. When the driver receives the "Loop Up" interrupt, it can go ahead and perform the Port Discovery procedures described in section 6.2.4.

7.6 Completion Messages

Tachyon generates completion messages for various events. All completion messages are 32 bytes long. Tachyon passes the completion message to the driver by writing it as an entry into the IMQ. The first 4 bytes of each IMQ entry indicate the type of the completion message. The remaining 28 bytes contain additional information or pad words depending on the type of the completion message. The different types of completion messages that are relevant at this point are:

- Outbound completion message with interrupt (outbound_i)
- Inbound SFS completion message (inbound_sfs)
- Inbound MFS completion message (inbound_mfs)
- SFS buffer warning completion message (sfs_buf_warn)
- MFS buffer warning completion message (mfs_buf_warn)
- IMQ buffer warning completion message (imq_buf_warn)
- Frame Manager Interrupt completion message (frame_mgr_interrupt)

The Outbound completion message indicates that an entry from the outbound queue OCQ has been processed or has been interrupted by an error. The Inbound completion message indicates that Tachyon has received either a Single or Multi-Frame Sequence. The Buffer warning completion messages are generated by Tachyon to warn the driver that the resources available are on the verge of being exhausted. An sfs_buf_warn or an mfs_buf_warn completion message is generated when only one SFSBQ or MFSBQ entry is available to receive data. Tachyon generates an imq_buf_warn completion message when only 2 empty entries are left in the IMQ. The Frame Manager may generate a frame_mgr_interrupt completion message to inform the driver that the configuration of the link has changed or that an error has occurred. The driver should then read the Frame Manager Status register to determine the cause of the completion message and take necessary actions.

7.7 Transmitting a Frame

For transmitting a frame, the driver needs to follow the procedures documented in section 3.2.1.1 (Transmit Process Overview). The Tachyon Header structure (shown in Figure 3-2) has a one-byte field called "AL_PA". This is the last byte of the dynamically assigned 3-byte Fibre Channel address of the target device. When Tachyon has a frame to be transmitted, it arbitrates for access to the medium and then opens a connection with the target device. It then transmits the frame to the device with which it had opened a connection. Once the frame has been transmitted, Tachyon closes the active connection and relinquishes control of the medium. This procedure is carried out for each frame that needs to be transmitted.

The value of the "AL_PA" field in the Tachyon Header structure depends on the topology.

- In a Private Loop (section 2.2), i.e., in a topology with no FL_Port, the dynamically assigned Fibre Channel address of each device is of the form hex "0000XX", where XX = AL_PA of the device. When the driver needs to transmit a frame to an AL_PA "YY", it puts the value "YY" into the "AL_PA" field in the Tachyon Header structure and sets the destination address to the address of the target device. Tachyon then takes care of opening a connection with the device with AL_PA "YY" and then transmits the frame. If a frame needs to be broadcast (as in the case of an ARP request), the driver fills in the "AL_PA" field of the Tachyon Header structure with value hex "FF" and sets the destination address to hex "FFFFFF".
- In a Public Loop (section 2.2), i.e., in a topology where there is an FL_Port, the target device can be either on the local Loop or it could be connected to a remote port on the switch. If the target device is on the same local Loop, the driver then fills in the "AL_PA" field of the Tachyon Header structure in the same manner as it did as in the case of the Private Loop scenario. If the target device is not on the local Loop, the task of passing on the frame to the target device rests with the switch. For achieving this, the driver fills in the value of the "AL_PA" field in the Tachyon Header structure with hex "00", which is the AL_PA reserved for the FL_Port. Tachyon then opens a connection with the FL_Port and transmits the frame to the FL_Port with the destination address set to the address of the target device. It is then the task of the FL_Port to route the received frame to the target device. If a frame needs to be broadcast (as in the case of an ARP request), the driver fills in the "AL_PA" field of the Tachyon

Header structure with value hex "00" and sets the destination address to hex "FFFFFF". It is the responsibility of the switch to broadcast the received frame both to the Loop on which the frame originated and also to all other remotely connected devices.

7.8 Login/Logout Protocols

After the driver has written the "Initialize" command into the Frame Manager Control register, Tachyon generates a frame_mgr_interrupt completion message and fills it in as an entry in the IMQ. The driver can then read the Frame Manager Status register to see the status of the Loop. If the "Loop Up" bit is set, it means that Loop initialization has been completed and that the device has obtained a unique AL_PA. The next task of the driver is to detect if it is on a Private or Public Loop. This can be determined by trying to transmit a FLOGI. If the FLOGI fails, it implies that an FL_Port does not exist on the local Loop and the driver can perform the Port Discovery procedure described in section 6.2.4.1. If the FLOGI completes successfully, it implies that an FL_Port is present on the local Loop. The driver can then perform the Port Discovery procedure described in section 6.2.4.2. Once the Port Discovery process has been completed, the driver has logged in with all the devices that are known to it through the Port Discovery process. Any device can log out with any other device by transmitting a Fibre Channel Extended Link Service (ELS) frame called the LOGO. The format of the frame is defined in [2]. Upon receiving a LOGO frame, the driver frees up all resources and information associated with the device that transmitted the LOGO. Before transmitting an ARP response or any IP frame, the devices must be logged in with each other. If an ARP response or an IP frame needs to be transmitted and if the target device is not logged in, then the driver performs a PLOGI with the target device. Upon receiving an ACC in response to the PLOGI, the ARP response or IP frame can be transmitted.

7.9 Things to Keep in Mind

There are several minor details that need to kept in mind while using the Tachyon. The most important ones are listed below:

- If the driver tries to transmit a frame to an AL_PA that does not exist on the local Loop, Tachyon posts a frame_mgr_interrupt completion message in the IMQ informing the driver that the AL_PA does not exist. Also, the Outbound Sequence Manager (OSM) (Figure 3-2) gets frozen. If the OSM is in a frozen state, Tachyon will not be able to transmit further frames. It is the responsibility of the driver to unfreeze the OSM. The driver does this by asserting the "Error Release" bit and the "OCQ Reset" bit in the Tachyon Control register.
- The OSM also gets frozen when a link that was "up" goes "down". The recovery procedure is the same as the one documented above.
- While performing an OCQ reset, the reset might not occur the instant the driver asserts the "OCQ Reset" bit of the Tachyon Control register. It is the responsibility of the driver to check the Tachyon Status register to check if the reset has been completed before proceeding. The bit in the Tachyon Status register that informs the driver about the state of the reset is called the "OCQ Reset Status" bit.
- Whenever the driver hands Tachyon an address in host memory, make sure that the driver is handing Tachyon the physical address on the host system. Handing out a virtual memory address is asking for trouble!
- While performing Port Discovery in a Private Loop, the driver will try to transmit PLOGI frames to all valid AL_PAs sequentially. During the process, whenever Tachyon detects that there is no device on the local Loop with the AL_PA that is used, it sets a bit in its internal registers to indicate that the AL_PA used was bad. But it erroneously does not clear that bit after processing has been completed. It clears the bit only after it has successfully transmitted a frame to a valid AL_PA. As a result, the driver should take some action to clear up the mess. The technique that this driver uses is to transmit a frame to itself whenever Tachyon informs the driver that an AL_PA used is invalid. When Tachyon succeeds in transmitting a frame to itself, the bad bits are cleared and sanity is restored.

- For an incoming frame, if Tachyon detects an incorrect CRC, it discards the frame.
- If the driver hands out a frame to Tachyon with a payload size exceeding 2048 bytes (maximum payload that Tachyon can transmit in one frame), Tachyon splits the frame into multiple frames and then transmits them as a Multi-Framed Sequence by correctly setting the appropriate bits in the frame header.

7.10 Clones

There are other companies in the market that uses Tachyon as their Fibre Channel controller on their Fibre Channel cards. The only major difference between those cards and the Interphase 5526 PCI Fibre Channel card is the interface to the PCI bus. This driver can be made to work for those cards by changing the functions that interact with the PCI interface.

CHAPTER 8

TESTING AND EVALUATION

The driver was run through the test suites developed by the University of New Hampshire's InterOperability Lab's Fibre Channel consortium. The test suites that were run on this driver were the FC-AL (Arbitrated Loop), FC-FLA (Fabric Loop Attachment) and FC-PLDA (Private Loop Direct Attach) test suites. Apart from the Fibre Channel conformance-like test suites, tests to measure the performance of the driver from an IP standpoint were also made.

8.1 Fibre Channel Conformance Tests

8.1.1 FC-AL Testing

The FC-AL test suite tests the conformance of a Device Under Test (DUT) to the Fibre Channel Arbitrated Loop standard (ANSI X3.272-1996). The test suite essentially tests the Fibre Channel protocol chip that is used. This driver uses the Tachyon Fibre Channel Controller as its protocol chip. Therefore the “passes” and “fails” in the test report are attributed mainly due to the features provided or not provided by the protocol chip. Since the test report is very detailed and voluminous, it is not included here. Copies of the report can be obtained from the University of New Hampshire’s InterOperability Lab’s web page at www.iol.unh.edu/consortiums/fc/fc_linux.html.

8.1.2 FC-FLA Testing

The FC-FLA test suite tests the conformance of the DUT to the Fabric Loop Attachment 2.7 standard. The test suite tests the features provided by the driver for working in a switched environment. The driver has passed most sections of the test suite. The features that have not been implemented in the driver include

Fabric Address Notification (FAN) as documented in [8]. Those have been left for future improvements. Since the test report is very detailed and voluminous, it is not included here. Copies of the report can be obtained from the University of New Hampshire's InterOperability Lab's web page at www.iol.unh.edu/consortiums/fc/fc_linux.html.

8.1.3 FC-PLDA Testing

The FC-PLDA test suite tests the conformance of the DUT to the Private Loop Direct Attach Profile. The test suite tests the features provided by the driver for working in a non-switched environment. The driver has passed most sections of the test suite. The features that have not been implemented in the driver include the ABTS protocol as documented in [7]. Those have been left for future improvements. Since the test report is very detailed and voluminous, it is not included here. Copies of the report can be obtained from the University of New Hampshire's InterOperability Lab's web page at www.iol.unh.edu/consortiums/fc/fc_linux.html.

8.2 Performance Testing

The driver was set up on two platforms. Each machine was a 200MHz Pentium with 64MB RAM running Linux 2.2.5. The configuration for the tests is shown below:

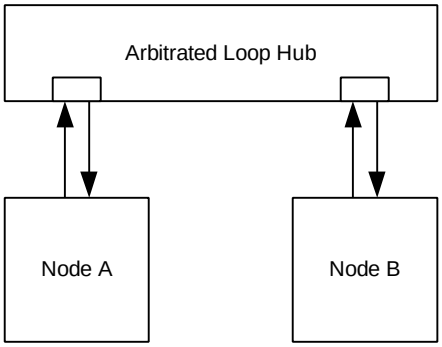

Figure 8-1 Setup for Performance Tests

Two different types of tests were performed on the driver. One of them was called the "ping" test and the other was called the "blast" test. In the "ping" test, one of the implementations acted as the server and the

other as the client. One message of fixed size was transferred back and forth between the server and client 10000 times. The purpose of this test was to measure the Round Trip Time (RTT) for messages of different sizes. The “ping” test was run using both UDP and TCP/IP. In the “blast” test, the client generated 10000 data messages and transmitted these to the server. The server just discarded the data. The purpose of this test was to measure the actual throughput achieved in MBps. TCP was used to perform the “blast” test.

The tests were run with the MTU of the driver set to various values. The different MTUs that were used were 2024, 4072, 8168, 16360 and 32744 bytes. The reason behind selecting these values as MTUs is described below. The maximum payload that Tachyon can transmit in a Fibre Channel frame is 2048 bytes. If more than 2048 bytes are handed to Tachyon, it will then fragment the data into chunks of 2048 bytes and transmit them. As described in section 4.3.1, for each IP/ARP frame, an eight-byte LLC/SNAP header and a sixteen-byte Network Header are added. Each IP frame is transmitted as a single Fibre Channel Exchange. So the maximum useful data that a single Fibre Channel Exchange can contain is (payload - 24) bytes. So when the MTU is set to 2024, each full IP packet is transmitted as one full sized (2048 bytes) Fibre Channel frame. In this case, fragmentation is done by the IP code. Whereas when the MTU is set to anything higher than 2024, fragmentation is done in hardware by Tachyon. As MTU size increases, the degree of fragmentation done by the IP layer decreases and the degree of fragmentation done by the hardware increases.

While using TCP, the factors that come into play when determining performance are:

- Nagle Algorithm
- Delayed Acknowledgements

TCP being a streaming protocol uses the Nagle algorithm. The Nagle algorithm reduces the number of small packets sent by a host. The Nagle algorithm works as follows:

- The first message always gets transmitted.
- For subsequent messages, if the messages do not form full-sized packets, they are buffered at the sender until either a full-sized packet can be transmitted or the acknowledgement for all previously transmitted data is received.

Thus the Nagle algorithm helps in avoiding the transmission of numerous very small packets and improves the ratio of useful data in a packet.

The purpose of Delayed Acknowledgements is to avoid transmission of explicit acknowledgements with the hope that acknowledgements can be piggybacked on outgoing data packets. There are rules that determine when the receiver should transmit an acknowledgement to the sender for the data received. The receiver will delay sending an acknowledgement until one or more of the following becomes true:

- It receives two full-sized frames.
- It receives out-of-order data, or,
- It receives a window update.

When it receives a packet and if none of the above conditions is true, the receiver then starts a timer. It sends an acknowledgement after the timer expires or when any of the conditions become true. The aim of delaying acknowledgements is to minimize traffic caused by transmitting explicit acknowledgement.

The tests were performed for different MTUs with the four possible conditions:

- Nagle Enabled, Delayed Acknowledgements Enabled
- Nagle Enabled, Delayed Acknowledgements Disabled
- Nagle Disabled, Delayed Acknowledgements Enabled
- Nagle Disabled, Delayed Acknowledgements Disabled

The user program can control enabling and disabling Nagle. This is done by setting the TCP_NODELAY option ON or OFF while setting the socket options. On the other hand, disabling Delayed Acknowledgements involves changing the kernel source code and recompiling it.

In a high-speed network like Fibre Channel, it might not be very effective to delay the transmission of acknowledgements or to try to minimize the number of packets transmitted. The time spent waiting for timers to expire before transmitting could be potentially used to transmit useful data. This is the rationale behind trying out the above-mentioned combinations.

8.2.1 Ping Test

As mentioned in section 8.2, the "ping" tests were performed using both TCP and UDP.

8.2.1.1 Ping Tests with TCP

The graphs for the "ping" test results using TCP are shown in Figures 8.2 - 8.5. The Round Trip Time in each case is shown. For each test, one message was sent back and forth 10000 times. The values reported are the average from 10 runs of each test.

When both Nagle and Delayed Acknowledgements are enabled, the graph in Figure 8-2 depicts sharp spikes when the message sizes hit MTU boundaries. When the graphs in Figure 8-2 and Figure 8-3 are compared, it can be observed that disabling Delayed Acknowledgements avoids the spikes that are seen when Delayed Acknowledgements are enabled. In the kernel, the implementation of Delayed Acknowledgements is such that the acknowledgements are delayed until the receiver receives at least two full frames. When the message size is just above the MTU size, the receiver waits for the second frame to arrive before sending an acknowledgement, while the sender waits for the acknowledgement of the first frame before transmitting the second frame. The receiver finally transmits an acknowledgement after a timer expires. This happens for each iteration of the message and hence the spikes in the graph shown in Figure 8-2.

In the other cases shown in Figure 8-4 and Figure 8-5, the performance is almost identical to that shown in Figure 8-3. In the cases where Delayed Acknowledgements are disabled, explicit acknowledgements are sent by the receiver for each packet. The performance in these cases does not seem to degrade. This is due to the fact that the available bandwidth is not utilized completely.

Therefore the conclusion is that Delayed Acknowledgments must be disabled for better performance in the “ping” tests. The problem with this is that enabling or disabling Delayed Acknowledgements is not under user control. In order to disable it, the source code of the kernel has to be changed.

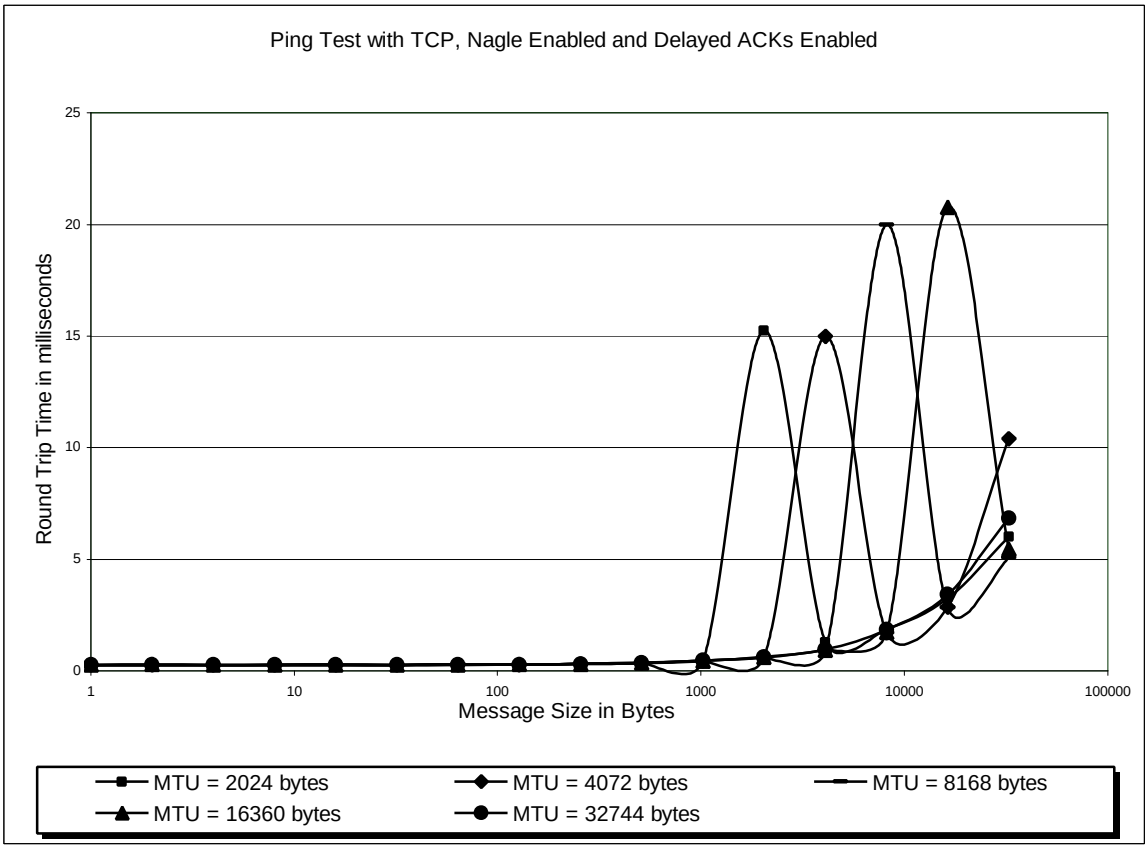

Figure 8-2 TCP Ping with Nagle Enabled and Delayed Acknowledgements Enabled

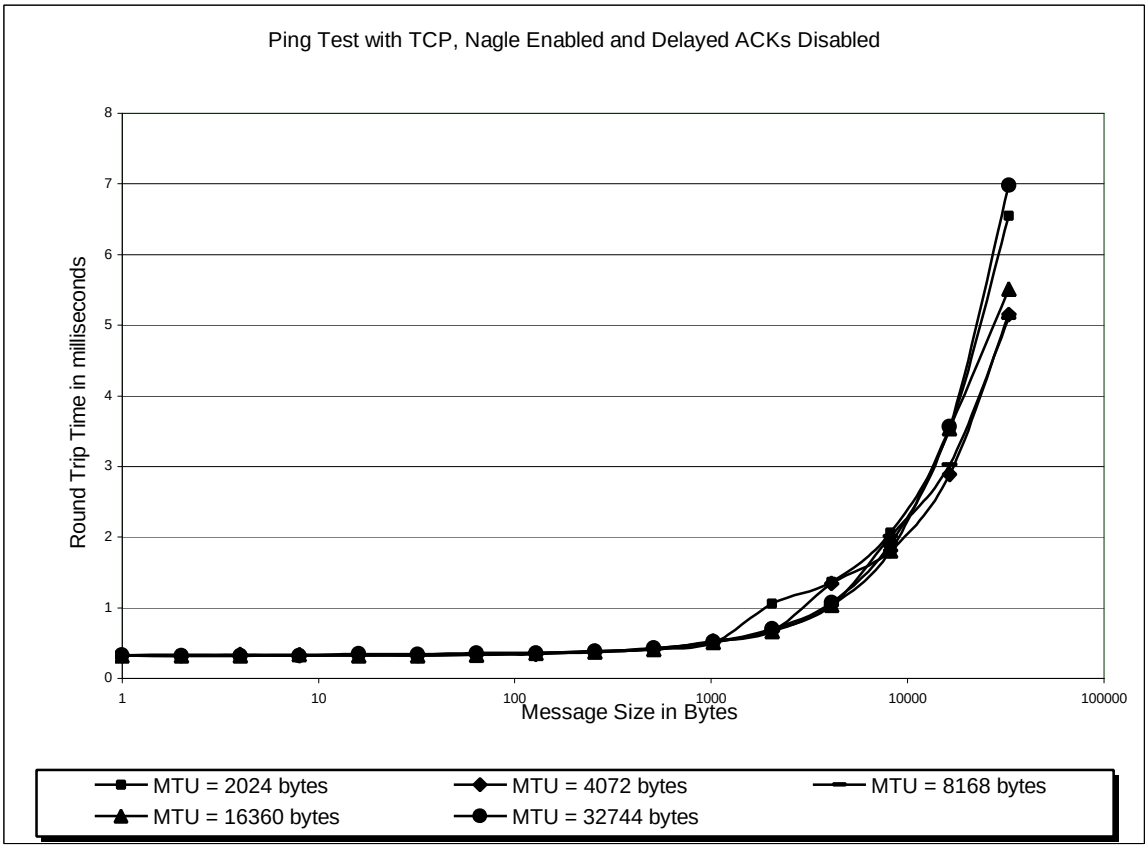

Figure 8-3 TCP Ping with Nagle Enabled and Delayed Acknowledgements Disabled

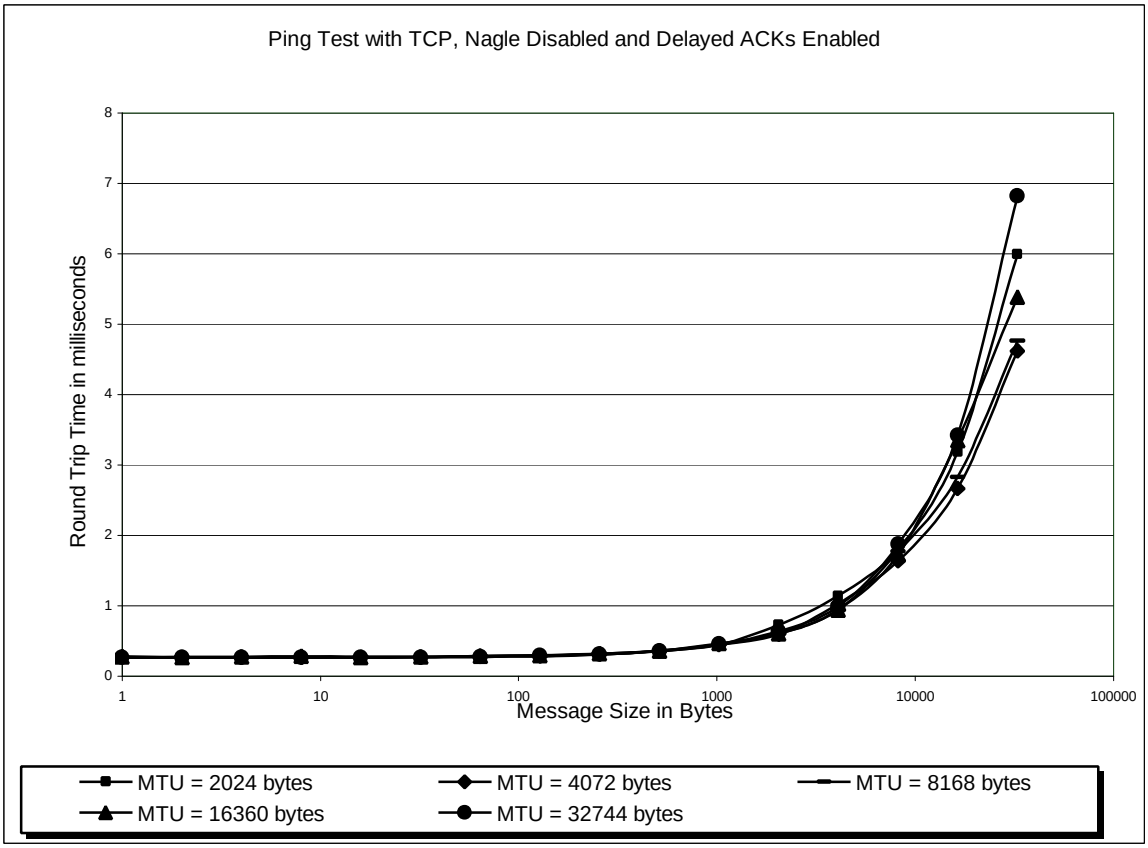

Figure 8-4 TCP Ping with Nagle Disabled and Delayed Acknowledgements Enabled

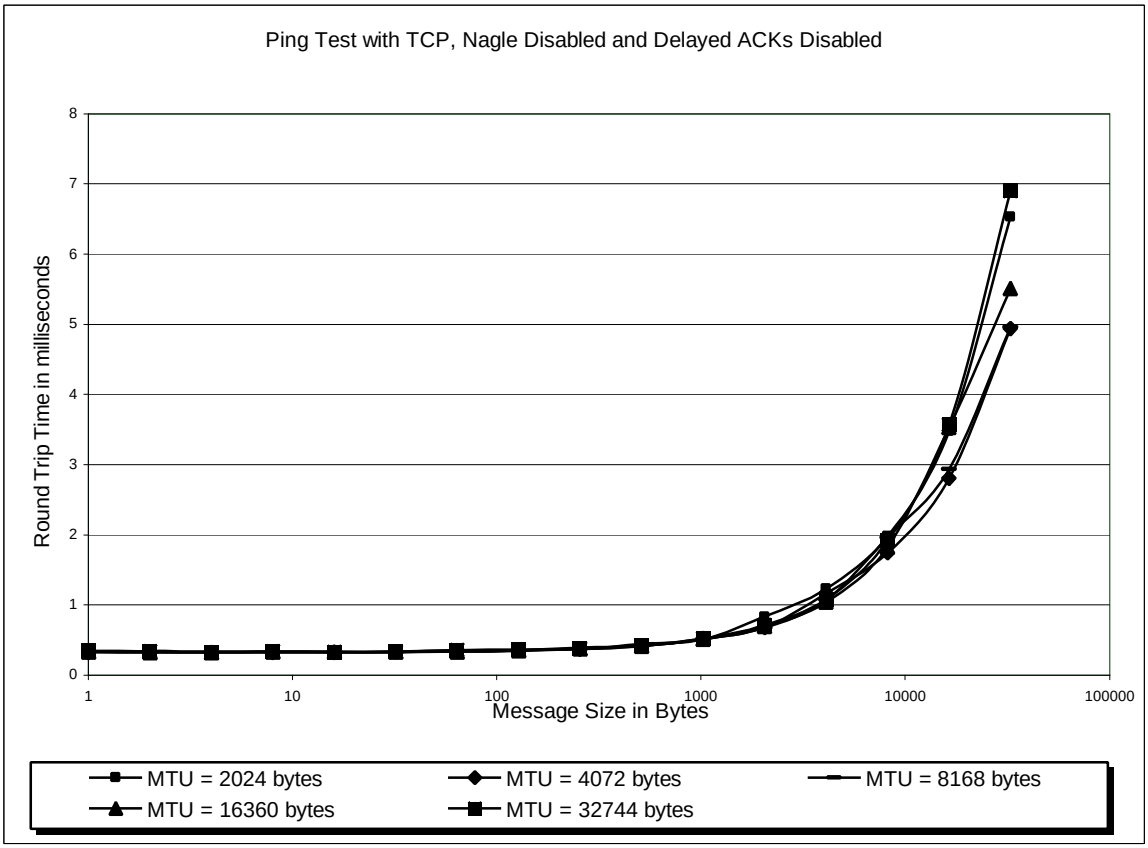

Figure 8-5 TCP Ping with Nagle Disabled and Delayed Acknowledgements Disabled

8.2.1.2 Ping Tests with UDP

The graph for the "ping" test results using UDP is shown in Figure 8-6.

For each test, one message was sent back and forth 10000 times. The values reported are the average from 10 runs of each test. The Round Trip Time (RTT) for messages of different sizes when the MTU was set to different values is shown below.

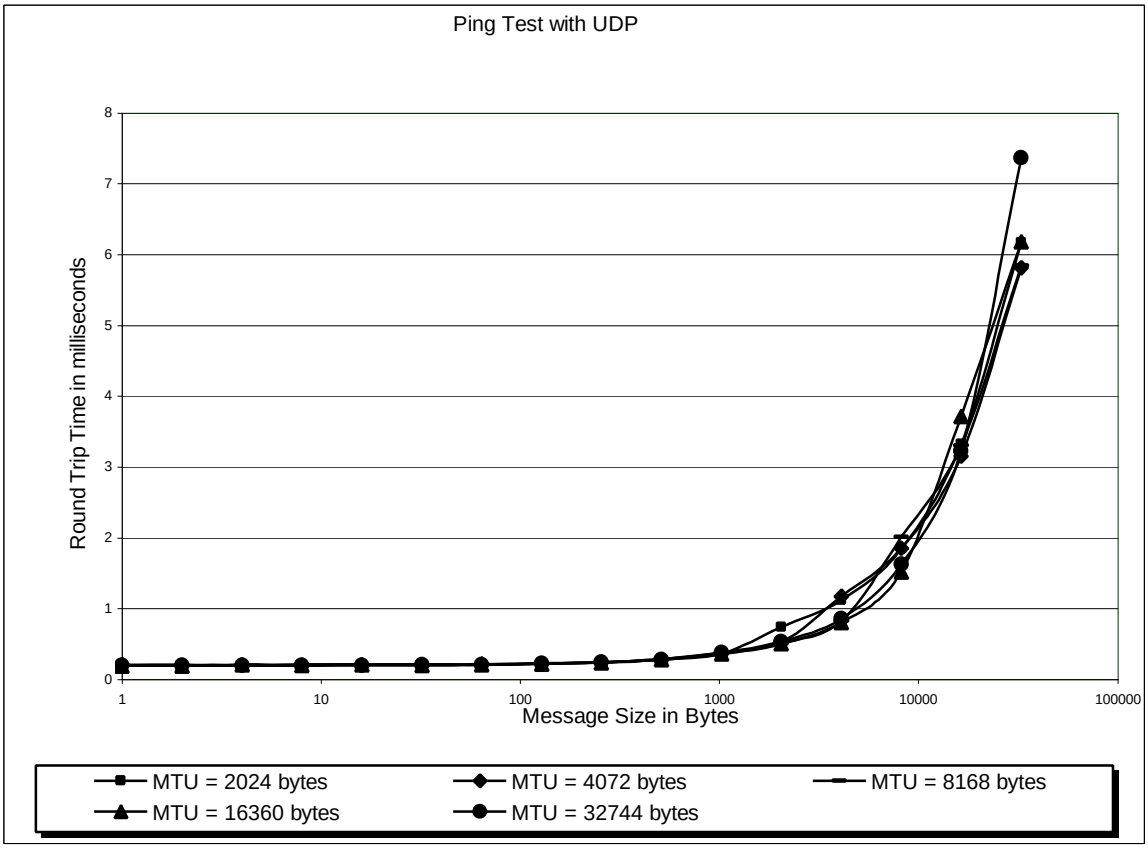

Figure 8-6 Ping Test with UDP

UDP does not have the options (Nagle and Delayed Acknowledgements) that were tested with TCP. The number of packets transmitted is relatively higher than TCP (because of the absence of optimizing algorithms like Nagle) and hence the poorer performance when compared to TCP. For example, comparing graphs in Figure 8-3 and Figure 8-6, the maximum round-trip-time for MTU = 32744 bytes is 7 msecs using TCP while it is about 7.5 msecs for UDP.

8.2.2 Blast Test

The graphs for the "blast" test results using TCP are shown in Figures 8.7 - 8.10.

The throughputs for different MTUs under different conditions are shown below. The different conditions are:

- Nagle Enabled and Delayed Acknowledgements Enabled
- Nagle Enabled and Delayed Acknowledgements Disabled
- Nagle Disabled and Delayed Acknowledgements Enabled
- Nagle Disabled and Delayed Acknowledgements Disabled

For one test, 10000 messages were sent from the client to the server. The values reported are the average from 10 runs of each test.

For the case where both Nagle and Delayed Acknowledgements are enabled (Figure 8-7), the throughput increases steadily as the message size increases. In the case where Nagle is disabled and Delayed Acknowledgement is enabled (Figure 8-8), the graph depicts bad performance. This is attributed to a bug in the implementation in the Linux kernel related to the code used when the Nagle algorithm is disabled. When sending a bunch of small messages, Linux allocates full MTU sized buffers for each message even though only part of the buffer is actually used when Nagle is disabled. The sender does not free them until they have been acknowledged. But there is a limit (set by the SO_SNDBUF option) on the buffer space that can be allocated to a socket connection. When a lot of small messages are sent, this limit will be exhausted rapidly and the sending process will stop transmitting. However, the receiver does not know this, and will transmit an acknowledgement only after some delay because the TCP window has not been exhausted.

Comparing Figure 8-9 and Figure 8-10, it can be observed that when Nagle is enabled, the throughput for lower message sizes is significantly higher when compared to that when Nagle is disabled. The effect of Nagle is not significant as the message sizes increase beyond 1000 bytes. Also, Delayed Acknowledgements does not seem to affect performance in the “blast” tests.

From this we conclude that Nagle should be enabled for better performance for the “blast” tests.

All the graphs depict an asymptotic behavior as the MTU increases beyond 1000 bytes. This is due to the fact that the CPU on the machines that were running the tests was running at 100% utilization. Once the CPU utilization reaches 100%, the throughput remains the same no matter what the message size is.

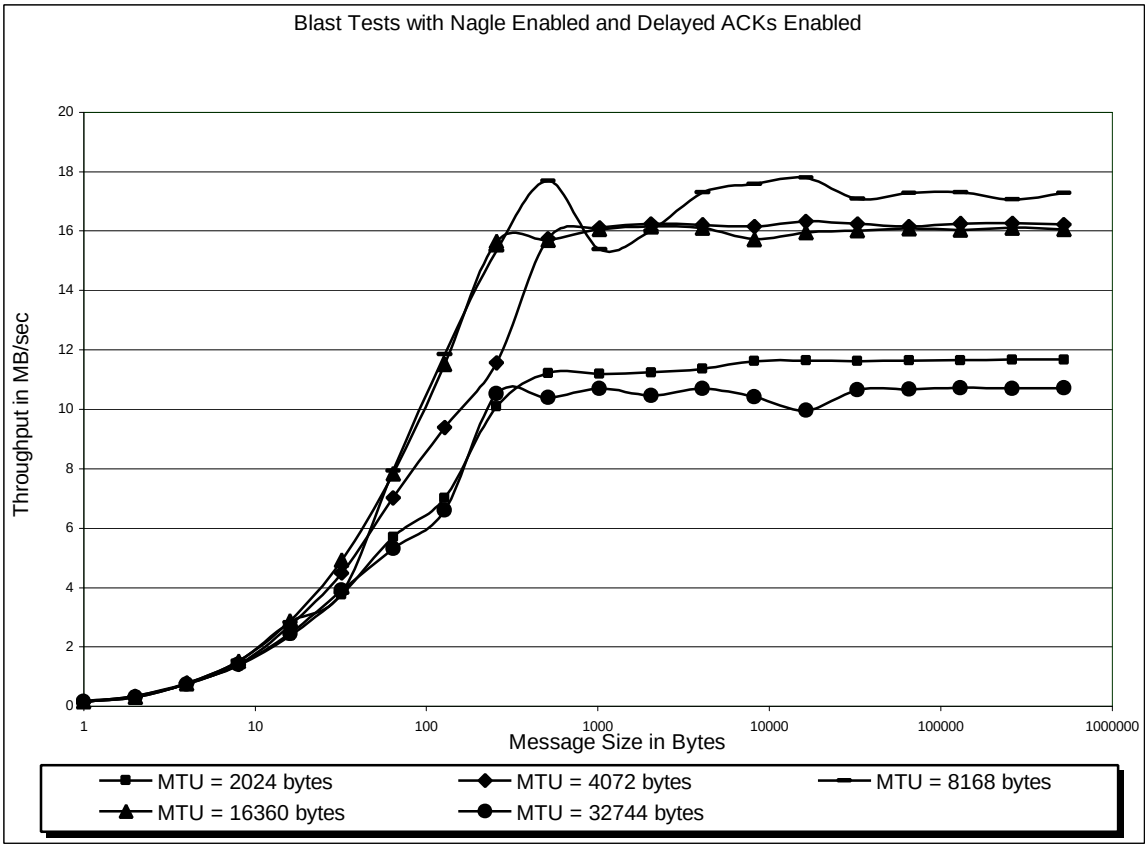

Figure 8-7 Blast Test with Nagle Enabled and Delayed Acknowledgements Enabled

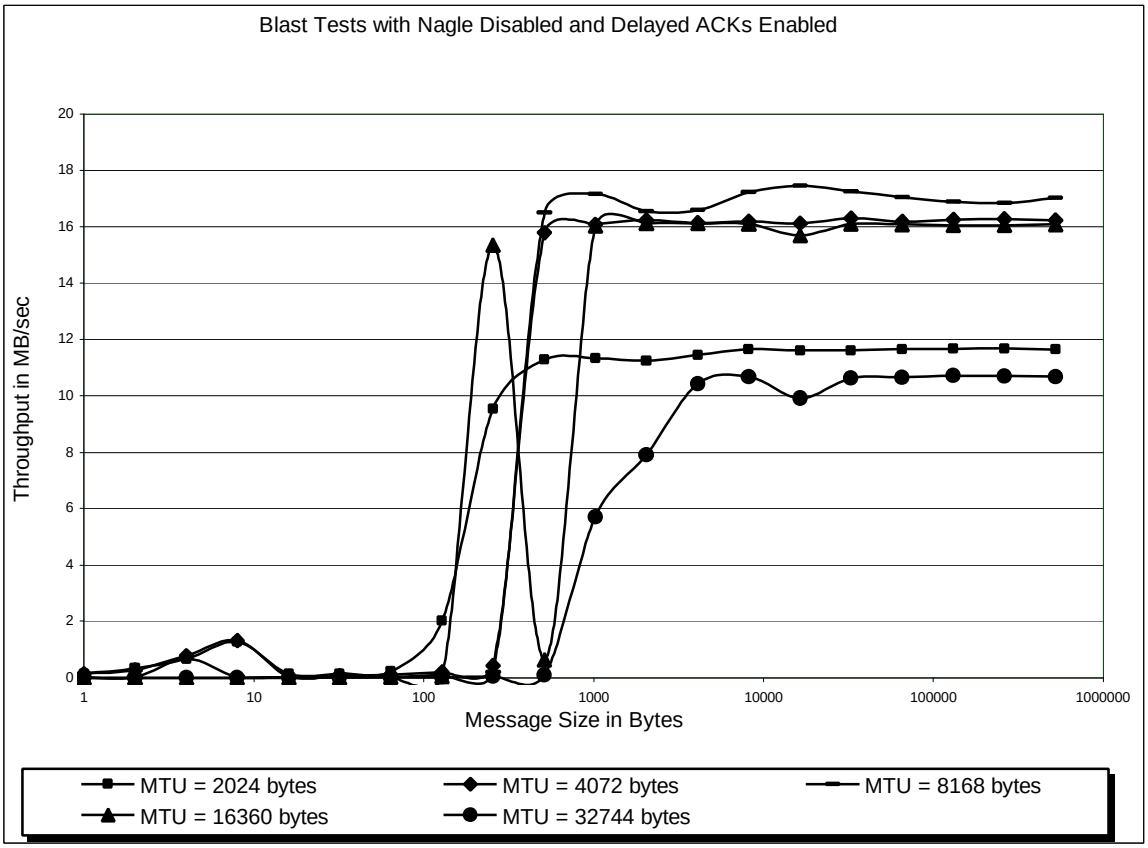

Figure 8-8 Blast Test with Nagle Disabled and Delayed Acknowledgements Enabled

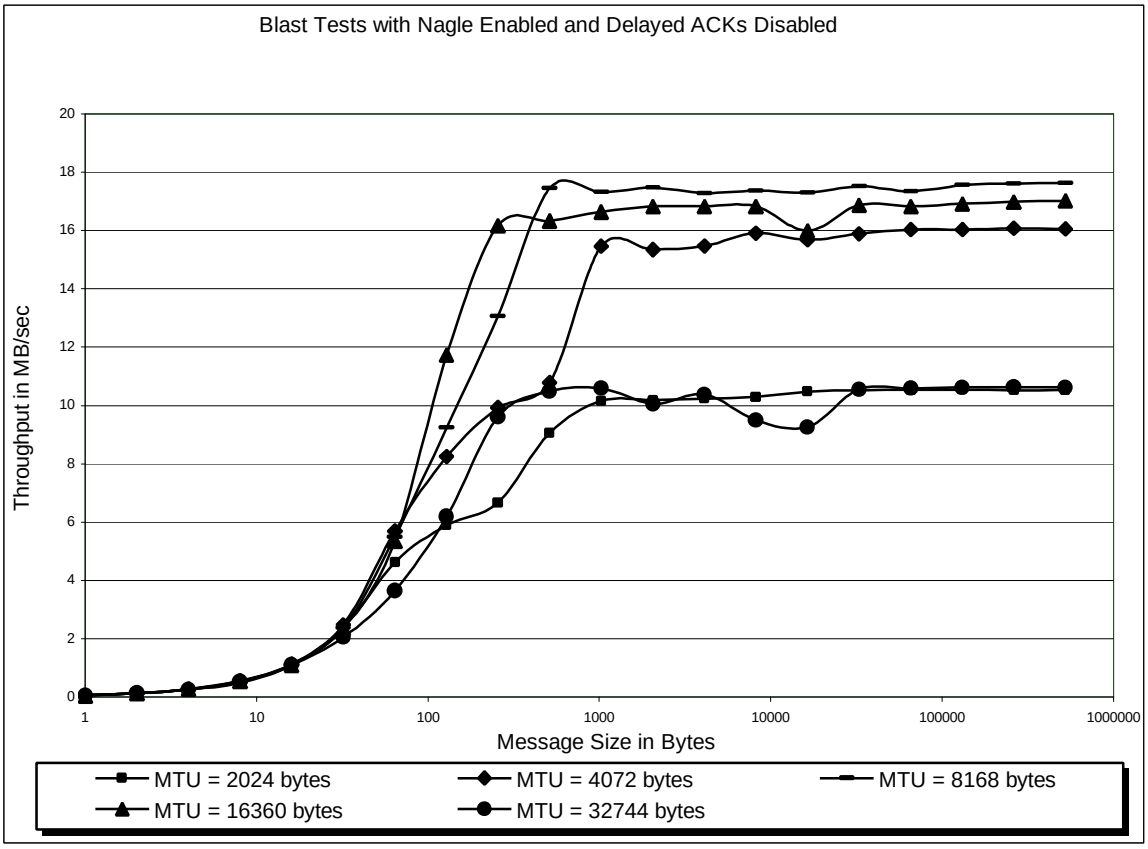

Figure 8-9 Blast Test with Nagle Enabled and Delayed Acknowledgements Disabled

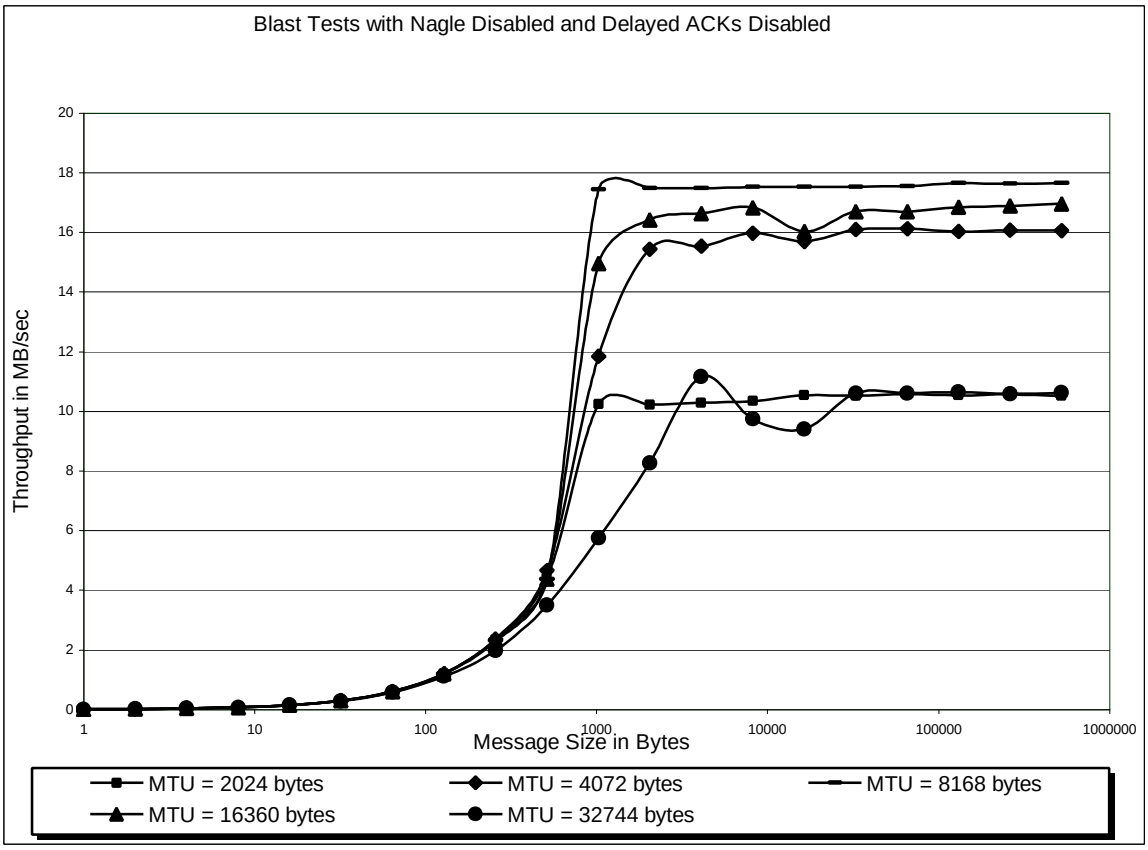

Figure 8-10 Blast Test with Nagle Disabled and Delayed Acknowledgements Disabled

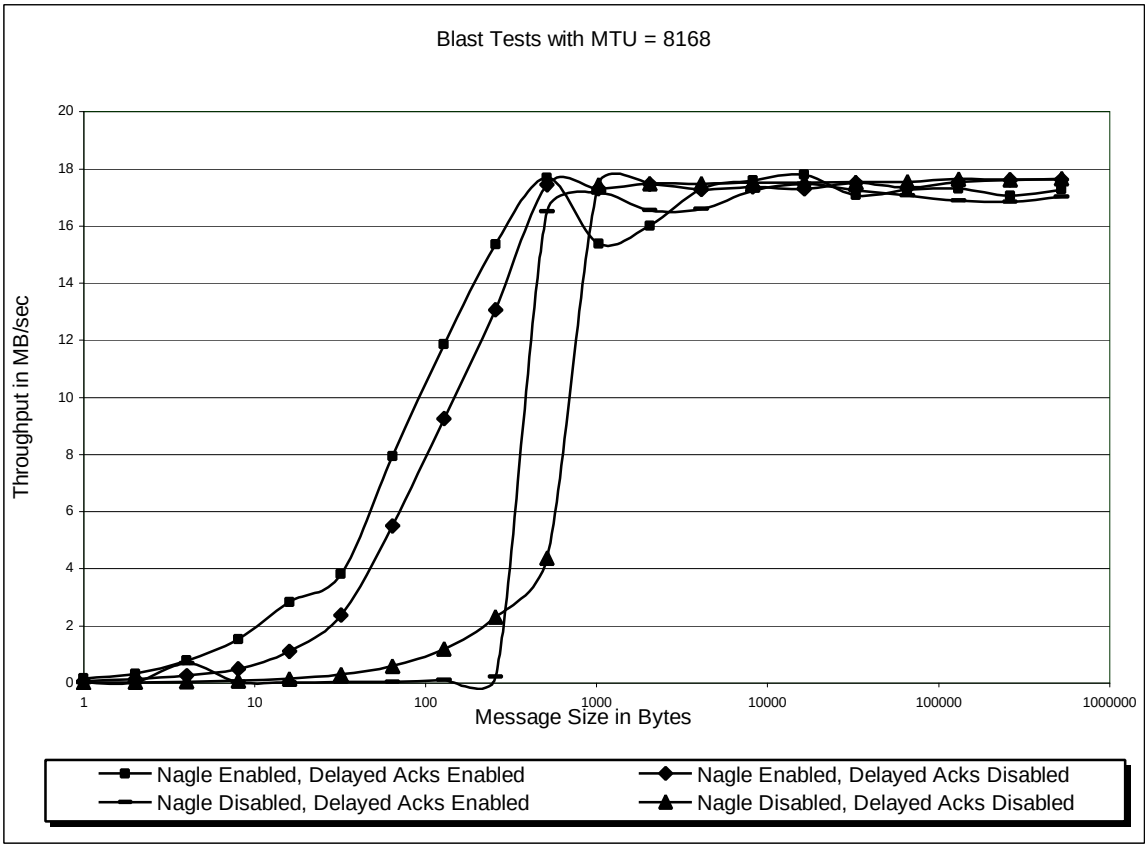

Figure 8-11 Blast Test with MTU = 8168 bytes

Although the MTU size did not significantly affect performance in the "ping" test, the graphs in Figure 8-7 through 8-10 show that the best performance in all four cases of the "blast" test is obtained when the MTU is set to 8168 bytes. The graph in Figure 8-11 shows the throughput when MTU is set to 8168 bytes and the other factors are varied. It can be seen that when both Nagle and Delayed Acknowledgements are enabled, the performance is the best. However, we have already concluded that enabling Delayed Acknowledgements affects the "ping" tests in a bad manner. In order to get the best performance for both "ping" and "blast" tests, it is necessary to enable Nagle and disable Delayed Acknowledgements. As shown in Figure 8-11, this combination reduces the performance in the "blast" test only slightly from the best combination.

8.3 Ideal Throughput

The ideal throughput can be calculated by first calculating the total time taken to transmit a message and then determining the number of bytes that can be transmitted in one second.

Total time to transmit a message = time to transmit a data frame + time due to overhead.

Time due to overhead = delay introduced by components (cable, node, hub) + time to transmit signaling data.

On a Fibre Channel link running at 1.0625 Gbps, the time to transmit a byte (i.e., 10 bits – due to the 8B/10B encoding as described in section 2.3.4) = 9.375 nanoseconds.

Therefore, the time to transmit a word (4 bytes, i.e., 40 bits) on a Fibre Channel link = 9.375 nanoseconds/byte * 4 bytes/word = 37.5 nanoseconds/word.

Delay due to components

- Cable delays

  According to [2], the maximum transmission delay caused by the fibre used = 5 nanoseconds/meter.

  For the tests performed in this thesis, the distance between each node and the hub = 5 meters.

  The one-way delay due to one cable = (5 nanoseconds/meter)*(5 meter) = 25 nanoseconds.

  Therefore, the one-way delay caused by two cables in terms of words = (2*25 nanoseconds)/(37.5 nanoseconds/word) = *1.33 words*.

- Node delays

  According to [2], the maximum delay at each node during transmission in terms of words = 6 words.

  Therefore the maximum delay caused on a two-node network during transmission in terms of words = 2*6 words = *12 words*.

- Hub delays

  The ports on the hub through which the nodes were connected cause a delay in transmission. The delay caused on each port on the hub in terms of words = 2 words/node.

  Therefore, the delay caused by connecting two nodes through a hub = 2*2 words = *4 words*.

The total one-way delay caused by:

- cable = *1.33 words*
- node delays = *12 words*
- hub delays = *4 words*

Therefore, the Round Trip Time (RTT) due to components on a two node network using a 5 meter cable through a hub = 2*(1.33 + 12 + 4) words = *34.66 words*.

Time to transmit signaling data

In an Arbitrated Loop, there are three Round Trip Times involved in transmitting a frame.

- First, the node that wishes to transmit needs to gain control of the Loop (which is a shared medium) by a process called *arbitrating*. To gain access to the Loop, the node transmits a 4-byte word called an ARB. When the ARB goes around the Loop and returns to the node that transmitted the ARB, it means that there is no other node that wants control of the Loop and that it can go to the next step towards transmitting a frame.
- After the source node has won arbitration, it *opens* the destination node. A source node opens a destination node by transmitting a 4-byte word called an OPN addressed to the destination node. Upon receiving the OPN addressed to it, the destination node indicates to the source node that it is prepared to receive frames. This is indicated by the transmission of a 4-byte word called R-RDY from the destination node. Once the source receives an R-RDY from the destination node, the source transmits the frame.
- After successful transmission of the frame, the source then *closes* the Loop so that other nodes can gain access to the Loop. This is performed by transmitting a 4-byte word called CLS. When the CLS goes round the Loop and returns to the node that transmitted it, the frame has been successfully transmitted.

Therefore, to transmit a frame, the total overhead = 3*RTT + time to transmit signaling data (ARB, OPN, R-RDY and CLS).

3*RTT = 3*34.66 words = 103.98 words.

Time to transmit ARB, OPN, R-RDY and CLS in terms of words = 4 words.

Therefore, total overhead time involved in transmitting a frame = 103.98 + 4 words ≅*108 words*.

Total time to transmit a message

Each Fibre Channel frame has the Start of Frame (SOF) delimiter, Fibre Channel header, a maximum of 2048 bytes of payload, Cyclic Redundancy Check (CRC), End of Frame (EOF) delimiter and Inter-frame gap. The optimal throughput values were calculated taking into consideration the transmission time only (assuming that there is no processing overhead). For example the time to transmit a 1024 byte message on a Fibre Channel Loop can be calculated as shown in Figure 8-12.

Field	Number of words	Number of bytes
SOF	1	4
Fibre Channel header	6	24
FC Network header	4	16
LLC/SNAP header	2	8
IP header	5	20
TCP header	5	20
User data	256	1024
CRC	1	4
EOF	1	4
Inter-frame gap	6	24
Overhead Round Trip Delays	108	432
TOTAL	395	1580

Figure 8-12 Total time Calculation for a user payload of 1024 bytes of data

Ideal throughput

The best possible time to transmit a frame containing 1024 bytes of payload = 1580*9.375 nanoseconds = 14812.5 nanoseconds. The throughput for a message size of 1024 bytes = 1024 bytes / 14812.5 nanoseconds = 69.13 MBps. Using the above calculations, the ideal throughput for a Fibre Channel Arbitrated Loop was calculated for different message sizes and is shown in Figure 8-13.

Message Size (bytes)	Ideal Throughput (MBps)	Actual Throughput as a % of Ideal Throughput
1	0.19	89.5
2	0.38	94.7
4	0.76	98.7
8	1.51	92.1
16	2.98	79.9
32	5.80	65.0
64	11.01	51.8
128	19.96	35.2
256	33.63	30.0
512	51.14	21.9
1024	69.13	16.2
2048	69.13	16.3
4096	75.80	15.0
8192	79.64	14.6
16386	81.71	14.2
32768	82.79	14.0
65536	82.79	14.1
131072	82.06	14.0
262144	83.20	14.0
524288	83.27	14.0

Figure 8-13 Ideal Throughput vs. Actual Throughput measured by 'blast' test

The performance numbers obtained from the "blast" tests compare poorly to that of Figure 8-13. The major limiting factor seems to be the processing capacity of the CPU since the CPUs of the machines used for testing were running at 100% utilization when the tests were performed.

8.4 Conclusion

The "ping" and "blast" tests measure two totally different aspects of network operation. The "ping" tests measure the round-trip-time while the "blast" tests measure throughput. In the "ping" tests we have seen that Delayed Acknowledgements should be disabled for better performance while for "blast" tests we have seen that Nagle should be enabled for better performance. The effect of Nagle in the "ping" tests and the effect of Delayed Acknowledgements in the "blast" tests are minimal. Also, both tests show that the performance is the best when the MTU is set to 8168 bytes. Performance increases steadily when the MTU is increased from 2024 to 4072 to 8168. This is due to the fact that for MTUs greater than 2024, fragmentation is done in hardware by Tachyon whereas when the MTU is 2024, fragmentation is done by the IP layer. The performance peaks at MTU = 8168 and then, for some unknown reason, drops for any

further MTU increases. Therefore it has been concluded that for getting the best performance under both types of tests, Nagle should be enabled, Delayed Acknowledgements should be disabled and the MTU should be set to 8168 bytes.

CHAPTER 9

SUMMARY AND FUTURE WORK

9.1 Conclusion

A Fibre Channel driver for IP on Linux has been designed, implemented and tested. The driver complies with most sections of the Internet draft "IP and ARP over Fibre Channel". Various Fibre Channel conformance and general performance tests were performed. The driver has been found to be compliant to most of the test suites that were run on the driver. The throughput achieved during performance testing has been satisfactory. The performance for both types of tests ("ping" and "blast") were found to be the best when Nagle was enabled, Delayed Acknowledgements were disabled and MTU was set to 8168 bytes.

The driver (which was initially developed on an x86 platform) has also been ported to work on the Alpha platform as well. The driver has also been tested on SMP machines and was found to work satisfactorily. The driver also can run with multiple cards on the same machine.

9.2 Future Work

- The implementation could be improved to make it fully compliant with the Internet draft and with other UNH-IOL test suites.
- A large chunk of the driver deals with Fibre Channel protocol related issues. This part could be modularized so that all Fibre Channel drivers that will be written in the future for Linux can use it.
- The performance of this Fibre Channel driver could be compared with drivers of other gigabit technologies like Gigabit Ethernet and also with other Fibre Channel drivers written for various other operating systems.

- The bandwidth utilization for IP over Fibre Channel could be compared with that of SCSI over Fibre Channel and the overhead involved in the two protocols could be measured.
- The performance for both “ping” and “blast” tests were found to be the best when the MTU was set to 8168 bytes. More research could be done to find out why the performance drops off when the MTU is greater than 8168 bytes.
- The maximum throughput achieved by the driver under the environment in which it was tested was around 18 MBps. As mentioned in section 8.3, the CPU utilization was 100% on the machines that were used for testing. When one of the 200MHz Pentiums was replaced by an Alpha running at 450MHz, the CPU utilization on the Alpha was around 5% and the throughput was much higher. Future performance tests can be tried on high-end machines to see if throughput gets closer to the ideal.

BIBLIOGRAPHY

[1] Alessandro Rubini. *Linux Device Drivers.* O'Reilly and Associates, Inc. First Edition.

[2] ANSI X3T11/Project 755D/Rev 4.3, Fibre Channel Physical and Signaling Interface (FC-PH)

[3] ANSI X3T11/Project 901D/Rev 7.3, Fibre Channel Physical and Signaling Interface – 2 (FC-PH-2)

[4] ANSI X3T11/Project 1119D/Rev 9.2, Fibre Channel Physical and Signaling Interface – 3 (FC-PH-3)

[5] ANSI X3T11/Project 955M/Rev 1.1, Fibre Channel Link Encapsulation (FC-LE)

[6] ANSI X3T11/Project 960D/Rev 4.5, Fibre Channel Arbitrated Loop (FC-AL)

[7] ANSI X3T11/Project 1162DT/Rev 2.1, Fibre Channel Private Loop SCSI Direct Attach (FC-PLDA)

[8] ANSI X3T11/Project 1235DT/Rev 2.7, Fibre Channel Fabric Loop Attachment (FC-FLA)

[9] FCA IP Profile, Revision 2.3, May 15, 1997

[10] FCSI IP Profile, FCSI-202, Revision 2.1 September 8, 1995

[11] Hewlett-Packard Company. *TACHYON User's Manual*, First Edition, Revision 5.0

[12] Interphase Corporation. *(i) Chip TPI User's Guide.*

[13] Murali Rajgopal, Raj Bhagwat and Wayne Richard. *Internet Draft - IP and ARP over Fibre Channel*.

[14] Meryem Primmer. *An Introduction to Fibre Channel.* Hewlett-Packard Journal. October 1996.

[15] University of New Hampshire's InterOperability Lab. *Fibre Channel Tutorial*. http://www.iol.unh.edu

APPENDIX A

Data for Performance Tests

Blast Test Results - MTU = 2024 bytes

Message Size (bytes)	T1 (sec)	T2 (sec)	T3 (sec)	T4 (sec)
1	0.06	0.14	0.06	0.98
2	0.05	0.15	0.06	0.99
4	0.05	0.14	0.06	0.99
8	0.06	0.14	0.06	0.99
16	0.06	0.14	1.03	0.98
32	0.08	0.13	2.00	0.99
64	0.11	0.13	2.71	1.00
128	0.17	0.21	0.60	1.01
256	0.24	0.37	0.26	1.04
512	0.44	0.54	0.43	1.11
1024	0.87	0.96	0.86	0.95
2048	1.74	1.92	1.74	1.91
4096	3.44	3.82	3.41	3.80
8192	6.73	7.59	6.70	7.56
16386	13.42	14.91	13.46	14.83
32768	26.88	29.71	26.89	29.69
65536	53.72	59.35	53.60	59.07
131072	107.31	118.56	107.11	118.63
262144	214.31	237.74	213.97	236.61
524288	428.31	474.75	429.14	475.50

T1 - Time taken for 10000 iterations with Nagle Enabled and Delayed Acknowledgements Enabled.

T2 - Time taken for 10000 iterations with Nagle Enabled and Delayed Acknowledgements Disabled.

T3 - Time taken for 10000 iterations with Nagle Disabled and Delayed Acknowledgements Enabled.

T4 - Time taken for 10000 iterations with Nagle Disabled and Delayed Acknowledgements Disabled.

Blast Test Results - MTU = 4072 bytes

Message Size (bytes)	T1 (sec)	T2 (sec)	T3 (sec)	T4 (sec)
1	0.06	0.14	0.06	0.98
2	0.06	0.14	0.07	0.98
4	0.05	0.14	0.05	0.98
8	0.05	0.14	0.06	0.98
16	0.06	0.13	2.86	0.98
32	0.07	0.12	4.03	0.99
64	0.09	0.11	5.29	1.00
128	0.13	0.15	5.73	1.01
256	0.21	0.25	5.59	1.04
512	0.31	0.45	0.31	1.05
1024	0.61	0.63	0.61	0.82
2048	1.20	1.27	1.20	1.26
4096	2.41	2.53	2.42	2.51
8192	4.84	4.91	4.82	4.89
16386	9.57	9.96	9.69	9.95
32768	19.24	19.68	19.16	19.42
65536	38.68	39.00	38.60	38.76
131072	76.90	77.98	76.86	77.99
262144	153.77	155.60	153.59	155.60
524288	308.19	311.44	308.13	311.43

T1 - Time taken for 10000 iterations with Nagle Enabled and Delayed Acknowledgements Enabled.

T2 - Time taken for 10000 iterations with Nagle Enabled and Delayed Acknowledgements Disabled.

T3 - Time taken for 10000 iterations with Nagle Disabled and Delayed Acknowledgements Enabled.

T4 - Time taken for 10000 iterations with Nagle Disabled and Delayed Acknowledgements Disabled.

Blast Test Results - MTU = 8168 bytes

Message Size (bytes)	T1 (sec)	T2 (sec)	T3 (sec)	T4 (sec)
1	0.06	0.15	1.11	1.01
2	0.06	0.15	0.87	1.01
4	0.05	0.14	0.06	1.02
8	0.05	0.16	2.86	1.01
16	0.05	0.14	7.63	1.02
32	0.08	0.13	10.64	1.02
64	0.08	0.11	11.12	1.03
128	0.10	0.13	11.58	1.03
256	0.16	0.19	11.39	1.06
512	0.28	0.28	0.30	1.12
1024	0.64	0.56	0.57	0.56
2048	1.22	1.12	1.18	1.12
4096	2.26	2.26	2.35	2.23
8192	4.44	4.50	4.54	4.46
16386	8.78	9.03	8.94	8.92
32768	18.30	17.85	18.11	17.83
65536	36.20	36.04	36.63	35.61
131072	72.21	71.23	74.06	70.84
262144	146.51	141.97	148.38	141.80
524288	289.38	283.76	293.71	283.17

T1 - Time taken for 10000 iterations with Nagle Enabled and Delayed Acknowledgements Enabled.

T2 - Time taken for 10000 iterations with Nagle Enabled and Delayed Acknowledgements Disabled.

T3 - Time taken for 10000 iterations with Nagle Disabled and Delayed Acknowledgements Enabled.

T4 - Time taken for 10000 iterations with Nagle Disabled and Delayed Acknowledgements Disabled.

Blast Test Results - MTU = 16360 bytes

Message Size (bytes)	T1 (sec)	T2 (sec)	T3 (sec)	T4 (sec)
1	0.06	0.15	18.85	0.99
2	0.06	0.15	17.86	0.99
4	0.05	0.13	15.65	0.99
8	0.05	0.14	20.81	0.99
16	0.05	0.14	22.09	0.99
32	0.06	0.13	23.41	1.00
64	0.08	0.11	24.04	1.00
128	0.11	0.10	0.16	1.02
256	0.16	0.15	0.31	1.04
512	0.31	0.30	0.61	1.12
1024	0.61	0.59	1.21	0.65
2048	1.21	1.16	2.42	1.19
4096	2.43	2.32	4.85	2.35
8192	4.97	4.64	9.95	4.64
16386	9.80	9.78	19.41	9.75
32768	19.51	18.52	38.84	18.70
65536	38.84	37.14	77.83	37.45
131072	77.84	73.93	77.83	74.20
262144	155.95	147.26	155.68	147.99
524288	311.35	294.00	310.76	294.89

T1 - Time taken for 10000 iterations with Nagle Enabled and Delayed Acknowledgements Enabled.

T2 - Time taken for 10000 iterations with Nagle Enabled and Delayed Acknowledgements Disabled.

T3 - Time taken for 10000 iterations with Nagle Disabled and Delayed Acknowledgements Enabled.

T4 - Time taken for 10000 iterations with Nagle Disabled and Delayed Acknowledgements Disabled.

Blast Test Results - MTU = 32744 bytes

Message Size (bytes)	T1 (sec)	T2 (sec)	T3 (sec)	T4 (sec)
1	0.06	0.14	51.57	1.03
2	0.06	0.14	59.06	1.03
4	0.05	0.14	55.06	1.03
8	0.05	0.14	49.91	1.03
16	0.06	0.14	49.91	1.03
32	0.08	0.15	53.75	1.04
64	0.11	0.17	49.91	1.06
128	0.19	0.20	49.91	1.09
256	0.23	0.25	52.43	1.24
512	0.47	0.47	49.94	1.40
1024	0.91	0.92	1.71	1.70
2048	1.87	1.94	2.46	2.36
4096	3.65	3.76	3.75	3.50
8192	7.50	8.22	7.31	8.02
16386	15.69	16.88	15.75	16.61
32768	29.34	29.61	29.40	29.46
65536	58.56	59.01	58.59	58.88
131072	116.54	117.66	116.59	117.45
262144	233.40	235.28	233.51	236.04
524288	466.53	470.83	467.71	470.74

T1 - Time taken for 10000 iterations with Nagle Enabled and Delayed Acknowledgements Enabled.

T2 - Time taken for 10000 iterations with Nagle Enabled and Delayed Acknowledgements Disabled.

T3 - Time taken for 10000 iterations with Nagle Disabled and Delayed Acknowledgements Enabled.

T4 - Time taken for 10000 iterations with Nagle Disabled and Delayed Acknowledgements Disabled.

Ping Test Results using TCP - MTU = 2024 bytes

Message Size (bytes)	T1 (msec)	T2 (msec)	T3 (msec)	T4 (msec)
1	2.70	3.32	2.66	3.23
2	2.74	3.39	2.65	3.44
4	2.69	3.25	2.66	3.35
8	2.69	3.25	2.67	3.38
16	3.08	3.25	2.70	3.35
32	2.75	3.27	2.73	3.30
64	2.79	3.42	2.78	3.46
128	2.93	3.58	2.88	3.57
256	3.24	3.91	3.10	3.68
512	3.63	4.16	3.59	4.41
1024	4.59	5.09	4.49	5.11
2048	152.58	10.59	7.33	8.35
4096	12.86	13.69	11.45	12.30
8192	18.67	20.64	18.10	19.78
16386	32.47	35.35	31.86	34.77
32768	60.04	65.43	59.92	65.35

T1 - Time taken for 10000 iterations with Nagle Enabled and Delayed Acknowledgements Enabled.

T2 - Time taken for 10000 iterations with Nagle Enabled and Delayed Acknowledgements Disabled.

T3 - Time taken for 10000 iterations with Nagle Disabled and Delayed Acknowledgements Enabled.

T4 - Time taken for 10000 iterations with Nagle Disabled and Delayed Acknowledgements Disabled.

Ping Test Results using TCP - MTU = 4072 bytes

Message Size (bytes)	T1 (msec)	T2 (msec)	T3 (msec)	T4 (msec)
1	2.66	3.26	2.77	3.24
2	2.73	3.23	2.66	3.27
4	2.67	3.41	2.67	3.23
8	2.68	3.23	2.68	3.25
16	2.72	3.39	2.74	3.26
32	2.73	3.28	2.73	3.30
64	2.85	3.44	2.91	3.52
128	2.87	3.45	2.89	3.55
256	3.13	3.83	3.15	3.69
512	3.55	4.17	3.58	4.11
1024	4.47	5.29	4.49	5.22
2048	6.18	6.73	6.00	6.78
4096	150.01	13.44	10.29	11.67
8192	17.26	18.11	16.40	17.42
16386	28.60	28.88	26.68	28.03
32768	104.10	51.51	46.19	49.41

T1 - Time taken for 10000 iterations with Nagle Enabled and Delayed Acknowledgements Enabled.

T2 - Time taken for 10000 iterations with Nagle Enabled and Delayed Acknowledgements Disabled.

T3 - Time taken for 10000 iterations with Nagle Disabled and Delayed Acknowledgements Enabled.

T4 - Time taken for 10000 iterations with Nagle Disabled and Delayed Acknowledgements Disabled.

Ping Test Results using TCP - MTU = 8168 bytes

Message Size (bytes)	T1 (msec)	T2 (msec)	T3 (msec)	T4 (msec)
1	2.68	3.24	2.69	3.34
2	2.69	3.23	2.71	3.25
4	2.78	3.23	2.77	3.26
8	2.70	3.24	2.69	3.26
16	2.79	3.24	2.78	3.29
32	2.73	3.39	2.73	3.33
64	2.82	3.36	2.78	3.34
128	2.90	3.65	2.88	3.49
256	3.18	3.73	3.17	3.90
512	3.58	4.25	3.58	4.15
1024	4.52	5.12	4.50	5.12
2048	6.01	6.84	6.09	6.73
4096	9.38	10.35	9.42	10.74
8192	200.01	20.10	17.73	19.59
16386	28.42	30.29	28.29	29.38
32768	50.74	50.97	47.59	49.61

T1 - Time taken for 10000 iterations with Nagle Enabled and Delayed Acknowledgements Enabled.

T2 - Time taken for 10000 iterations with Nagle Enabled and Delayed Acknowledgements Disabled.

T3 - Time taken for 10000 iterations with Nagle Disabled and Delayed Acknowledgements Enabled.

T4 - Time taken for 10000 iterations with Nagle Disabled and Delayed Acknowledgements Disabled.

Ping Test Results using TCP - MTU = 16360 bytes

Message Size (bytes)	T1 (msec)	T2 (msec)	T3 (msec)	T4 (msec)
1	2.77	3.26	2.80	3.35
2	2.94	3.27	2.72	3.24
4	2.75	3.25	2.75	3.24
8	2.87	3.38	2.87	3.28
16	2.81	3.24	2.73	3.27
32	2.79	3.27	2.78	3.30
64	2.82	3.37	2.87	3.35
128	3.04	3.56	2.97	3.46
256	3.12	3.80	3.23	3.75
512	3.55	4.12	3.62	4.16
1024	4.52	5.17	4.67	5.12
2048	6.10	6.68	6.05	6.96
4096	9.50	10.35	9.46	10.39
8192	17.38	18.07	17.37	18.26
16386	207.70	35.39	33.53	35.31
32768	54.70	55.09	53.89	55.08

T1 - Time taken for 10000 iterations with Nagle Enabled and Delayed Acknowledgements Enabled.

T2 - Time taken for 10000 iterations with Nagle Enabled and Delayed Acknowledgements Disabled.

T3 - Time taken for 10000 iterations with Nagle Disabled and Delayed Acknowledgements Enabled.

T4 - Time taken for 10000 iterations with Nagle Disabled and Delayed Acknowledgements Disabled.

Ping Test Results using TCP - MTU = 32744 bytes

Message Size (bytes)	T1 (msec)	T2 (msec)	T3 (msec)	T4 (msec)
1	2.72	3.31	2.69	3.41
2	2.71	3.26	2.69	3.36
4	2.72	3.25	2.70	3.23
8	2.74	3.27	2.71	3.37
16	2.75	3.48	2.73	3.28
32	2.73	3.45	2.74	3.32
64	2.92	3.64	2.80	3.58
128	2.92	3.63	3.01	3.62
256	3.17	3.87	3.15	3.78
512	3.68	4.28	3.65	4.22
1024	4.69	5.21	4.61	5.22
2048	6.29	7.02	6.41	7.10
4096	9.73	10.73	9.91	10.81
8192	18.56	19.09	18.76	19.22
16386	34.22	35.62	34.25	35.71
32768	68.52	69.78	68.21	69.03

T1 - Time taken for 10000 iterations with Nagle Enabled and Delayed Acknowledgements Enabled.

T2 - Time taken for 10000 iterations with Nagle Enabled and Delayed Acknowledgements Disabled.

T3 - Time taken for 10000 iterations with Nagle Disabled and Delayed Acknowledgements Enabled.

T4 - Time taken for 10000 iterations with Nagle Disabled and Delayed Acknowledgements Disabled.

Ping Test Results using UDP

Message Size (bytes)	T1 (msec)	T2 (msec)	T3 (msec)	T4 (msec)	T5 (msec)
1	2.05	2.08	1.99	1.93	1.97
2	2.03	2.02	2.10	1.93	2.00
4	1.99	1.99	1.98	2.14	2.01
8	2.13	2.05	2.04	2.01	1.98
16	2.11	2.04	2.05	2.10	2.03
32	2.15	2.05	2.07	2.02	2.07
64	2.08	2.17	2.11	2.09	2.14
128	2.30	2.26	2.26	2.21	2.26
256	2.41	2.43	2.46	2.40	2.45
512	2.82	2.84	2.89	2.79	2.87
1024	3.66	3.70	3.80	3.63	3.78
2048	7.39	5.26	5.17	5.08	5.38
4096	11.19	11.74	8.27	8.10	8.57
8192	18.46	18.51	20.10	15.25	16.27
16386	33.25	31.54	33.05	37.18	32.31
32768	61.77	58.17	58.47	61.85	73.73

T1 - Time taken for 10000 iterations with MTU = 2024 bytes.

T2 - Time taken for 10000 iterations with MTU = 4072 bytes.

T3 - Time taken for 10000 iterations with MTU = 8168 bytes.

T4 - Time taken for 10000 iterations with MTU = 16360 bytes.

T5 - Time taken for 10000 iterations with MTU = 32744 bytes.

APPENDIX B

Installation Guide

Steps Involved in installing the driver:

- Create a clean Linux 2.2.5 source tree.
- From the directory /usr/src/linux, run "patch –p1 < *patch_file*" to apply the patch for the driver.
- From the directory /usr/src/linux, run "make config" and configure the driver to be loaded as a module or as part of the kernel.
- From the directory /usr/src/linux, run "make dep; make bzImage". Once the kernel has been compiled, copy to the bootable image to the /boot/ directory. Run LILO.
- In the directory /etc/sysconfig/network-scripts/, create an entry for the Fibre Channel interface.

Files that were added/modified:

The patch includes several files that were either newly added or modified. The contents of the patch are given below.

New Files

The core driver specific files are present in a separate directory called "fc" under the /usr/src/linux/drivers/net directory. The files present in the directory /usr/src/linux/drivers/net/fc are:

- Makefile - Makefile for the "fc" directory.
- iph5526.c - The main driver file.
- iph5526_ip.h - It has definitions specific to IP.
- iph5526_novrom.c - It has routines to read the NOVROM on the card.
- tach.h - It has definitions for Tachyon and (i)chip TPI.
- tach_structs.h - It has the definition for the structures used by the driver.

The other newly added files include:

- /usr/src/linux/include/linux/fcdevice.h - It has definitions for Fibre Channel kernel functions.
- /usr/src/linux/include/linux/if_fc.h - It has global definitions for Fibre Channel.
- /usr/src/linux/net/802/fc.c - It has routines used to build headers for Fibre Channel network devices.

Modified Files

- /usr/src/linux/Makefile - Added entry for the /usr/src/linux/drivers/net/fc directory.
- /usr/src/linux/drivers/net/Config.in - Added entry for configuring Fibre Channel network devices.
- /usr/src/linux/drivers/net/Makefile - Added entry to include the directory /usr/src/linux/drivers/net/fc.

- /usr/src/linux/drivers/net/Space.c - Added routines to detect Fibre Channel network devices.
- /usr/src/linux/drivers/net/net_init.c - Added routines to support Fibre Channel network devices.
- /usr/src/linux/net/netsyms.c - Added entries to export Fibre Channel kernel functions.
- /usr/src/linux/include/linux/netdevice.h - Added entries for supporting bootable and loadable versions of Fibre Channel network drivers.
- /usr/src/linux/include/linux/pci.h - Added entries for vendor and device identifiers for the Interphase 5526 card.
- /usr/src/linux/net/802/Makefile - Added entry to include Fibre Channel specific files.