

# Generic SCSI Command Generator

A Major Qualifying Project Report

submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

---

**Alex Kinney**

---

**Minh Huynh**

Date: March 12, 2004

APPROVED:

---

**Professor Bob Kinicki, Major Advisor**

---

**Professor Mark Claypool, Co-Advisor**

## Table of Contents

<b>1</b>	<b>Introduction .....</b>	<b>7</b>
<b>2</b>	<b>SAN Features and Benefits .....</b>	<b>9</b>
2.1	Disaster Recovery .....	9
2.2	Fault Tolerance.....	10
2.3	Scalability.....	10
2.4	Transfer Performance.....	11
2.5	Manageability.....	11
2.6	Network Performance Enhancements .....	12
2.7	Cost Effectiveness .....	12
<b>3</b>	<b>SCSI.....</b>	<b>13</b>
3.1	Bus Control Signals.....	13
3.2	Bus Phases.....	14
3.2.1	Bus Free.....	14
3.2.2	Arbitration.....	14
3.2.3	Selection.....	14
3.2.4	Reselection .....	14
3.2.5	Command .....	15
3.2.6	Data .....	15
3.2.7	Status.....	15
3.2.8	Message.....	15
3.3	Generic SCSI Interface.....	15
<b>4</b>	<b>Fibre Channel .....</b>	<b>18</b>
4.1	Fibre Channel Topology.....	19
4.1.1	Point to Point.....	19
4.1.2	Arbitrated Loop.....	20
4.1.3	Fabric Switch.....	21
4.2	Fibre Channel Layers .....	22
4.2.1	FC-0.....	23
4.2.2	FC-1.....	23
4.2.3	FC-2.....	24
4.2.4	FC-3 and FC-4.....	26
4.3	Flow Control .....	27
4.3.1	Buffer to Buffer.....	27
4.3.2	End to End.....	28
4.4	Classes of Service.....	28
<b>5</b>	<b>FC Analyzer .....</b>	<b>31</b>
5.1	Recording a Trace .....	32
5.2	Conventional Recording.....	33
5.3	Spooled Recording.....	34

5.4	Recording Rules .....	35
5.5	Reading and Interpreting Trace Data .....	38
<b>6</b>	<b>The SCSI Command Generator .....</b>	<b>41</b>
6.1	Requirement Elicitation.....	41
6.2	Selected Commands .....	41
6.2.1	INQUIRY .....	42
6.2.2	READ .....	43
6.2.3	WRITE .....	45
6.2.4	READ CAPACITY .....	46
6.2.5	MODE SENSE .....	47
6.2.6	TEST UNIT READY .....	48
6.2.7	READ DEFECT DATA.....	49
6.2.8	REPORT LUNS .....	50
6.2.9	VERIFY .....	51
6.2.10	SEND DIAGNOSTIC .....	52
6.3	SCSI Command Generator Features .....	54
6.3.1	The Interface .....	54
6.3.2	Pre-Defined Commands .....	54
6.3.3	Customized Commands.....	55
6.3.4	Scripting Commands.....	55
<b>7</b>	<b>Interpreting and Decoding SCSI Data .....</b>	<b>58</b>
7.1	The Exchange Level.....	58
7.2	The Sequence Level .....	59
7.3	The Frame Level .....	61
<b>8</b>	<b>Conclusions .....</b>	<b>66</b>
<b>9</b>	<b>Future Work .....</b>	<b>67</b>
	<b>References.....</b>	<b>68</b>
	<b>Appendix A.....</b>	<b>70</b>
	The sg_io_hdr Structure .....	70
	<b>Appendix B.....</b>	<b>71</b>
	SCG Documentation .....	71

## Table of Figures

Figure 4-1: Point to Point Topology [FIB95].....	20
Figure 4-2: Arbitrated Loop Topology [FIB95].....	21
Figure 4-3: Fabric Switch Topology .....	22
Figure 4-4: Frame Structure [FIB94] .....	26
Figure 5-1: FCAnalyzer .....	31
Figure 6-1: INQUIRY Command Structure.....	43
Figure 6-2: READ(6) Command Structure .....	44
Figure 6-3: READ(10) Command Structure .....	44
Figure 6-4: WRITE(6) Command Structure .....	45
Figure 6-5: WRITE(10) Command Structure .....	46
Figure 6-6: READ CAPACITY Command Structure.....	47
Figure 6-7: MODE SENSE(6) Command Structure .....	48
Figure 6-8: MODE SENSE(10) Command Structure .....	48
Figure 6-9: TEST UNIT READY Command Structure.....	49
Figure 6-10: READ DEFECT DATA Command Structure.....	50
Figure 6-11: REPORT LUNS Command Structure.....	51
Figure 6-12: VERIFY Command Structure .....	52
Figure 6-13: SEND DIAGNOSTIC Command Structure.....	53
Figure 6-14: Example SCG usage .....	54
Figure 6-15: Sample Script .....	57
Figure 7-1: INQUIRY Exchange .....	58
Figure 7-2: TEST UNIT READY Exchange .....	58
Figure 7-3: READ DEFECT DATA Exchange .....	59
Figure 7-4: INQUIRY Sequence.....	60
Figure 7-5: INQUIRY Sequence Collapsed.....	61
Figure 7-6: Inquiry Frame.....	62
Figure 7-7: INQUIRY Frame Collapsed.....	63
Figure 7-8: INQUIRY Data Block.....	65

## Executive Summary

The need for a faster and more efficient network is a growing need in today's technology field. In particular, the company General Dynamics (GD) has interests in implementing secure, streaming, high-speed communication networks within highly classified government projects. A Storage Area Network (SAN) using Fibre Channel, a fast connection allowing transfer rates of up to two gigabits per second, is a popular solution. Because SAN and Fibre Channel are relatively new technologies, this project focuses on helping GD to learn and understand more about the traffic that utilizes this advanced network medium.

In this project, the chosen approach to learning more about SAN and Fibre Channel was the development of a test tool dubbed the SCSI Command Generator (SCG) that generates basic traffic on demand. This tool generates a number of SCSI commands and sends them to devices attached to a SAN. Scripting capabilities allow a user to pre-record a limitless series of these commands, and then to execute them all at once, one after the other in rapid succession. The SCG also provides the ability for users to configure customized commands that may not be specifically implemented at the time, meaning it actually supports every SCSI command in existence. With the help of the SCG, GD developers will be able to generate streams of traffic that can later be examined and analyzed.

In particular, the analyzer tool used by GD is known as the FCTracer. Its capabilities and features include displaying recorded traffic using a graphical interface and multi-port recording with advanced triggering and decoding features. It also clarifies the higher level significance of Fibre Channel traffic (SCSI for example), and not just the raw data. Automatic decoding and the use of color-coded packets to represent frames, sequences, and exchanges help to show the meaning of Fibre Channel traffic so that potential problems can be found more quickly and efficiently.

Unfortunately, because of General Dynamics' security needs, the precise role for the SCG remains somewhat of a mystery. All that is known externally is that it will have something to do with classified Fibre Channel encryption methods. In any case, the SCG should prove helpful to GD, and could be easily modified to meet any potential changing needs.

## **Abstract**

This project resulted in the development of a tool called the SCSI Command Generator to allow General Dynamics to create strictly controlled traffic over a Fibre Channel Storage Area Network (SAN). A wide array of SCSI commands can be strung together and sent to devices in succession. The resulting traffic can be analyzed in real time to help developers test and debug changes to a Fibre Channel SAN.

# 1 Introduction

Recently, the efficient storage and transfer of large amounts of data have become increasingly important. As data storage capabilities and transfer speeds advance, tasks like streaming large media files from single or distributed sources are emerging as important applications to support. Storage Area Networks (SANs) and Fibre Channel are potential solutions to support the modern generation of network applications.

A SAN is a high-speed network that connects shared storage devices, such as disk, tape, or CD-ROM drives, to computing devices like PCs or file servers. The goal of an SAN is to provide high-speed, fault-tolerant access to data for multiple client/server and host computers, which may range from single computers to large mainframe systems. Most importantly, however, the specifics of the storage devices on the SAN remain transparent to its users. This means that to a client, a SAN appears as a single contiguous data source, while in reality this is likely not the case.

Physically, a SAN will likely consist of multiple separate data sources that share storage responsibilities. When separating the data storage units, the need arises to centrally connect the devices in a fast, reliable manner. One possible technology is the use of Fibre Channel, which can offer data rates of one or two gigabits per second. However, connecting the devices with Fibre Channel is not sufficient to ensure the optimal performance of a SAN. Other factors, such as the configuration of the SAN and its individual devices, can also affect the success of the network. Fortunately, there exist tools that can test and debug Fibre Channel traffic and assist in optimizing the conditions on a SAN.

Computer Access Technology Corporation's FCTracer is one such product. By thoroughly analyzing real-time 2-gigabit Fibre Channel traffic, it is capable of offering detailed insight into conditions surrounding the data being transferred over a storage area network. CATC's Tracer software is able to provide users with a powerful tool for specifically dissecting and managing its SAN traffic.

In particular, General Dynamics, a leading supplier of sophisticated defense systems to the United States government, is especially interested in SAN technology.

Although details of their plans are guarded as secrets, GD is currently developing methods of Fibre Channel encryption to ensure secure data transfer in sensitive circumstances. The company requested that a group of Computer Science students from WPI develop a tool to assist them in the development of this new technology.

In order to help them effectively test and understand encrypted SAN traffic, we set out to develop a SCSI Command Generator to create user-defined SCSI traffic, and therefore provide its users with better control of SAN traffic at a low level. The application would issue SCSI level commands directly over Fibre Channel to storage devices on the attached SAN. Commands performing tasks like reading data, writing data, performing diagnostic tests, or requesting information regarding a device's status could be sent individually or as a sequence of commands strung together. The result would be the ability to analyze the consequent traffic with the FCTracer and its associated FC Analyzer software, which processes the traffic at the Fibre Channel level and displays it meaningfully to the user. With the help of the SCSI Command Generator, developers should be able to find and correct problems faster within a Fibre Channel SAN.

## **2 SAN Features and Benefits**

Storage networking is a technology that is of great importance to the field of information technology and computer science today. It is a consistently growing field because of growing needs large quantities of data and faster transfer speeds. SAN technology is a primary solution for satisfying these storage networking needs.

A SAN is a high-speed LAN (Local Area Network) that connects shared storage devices, such as disk, tape and CD ROM drives to all types of computing devices. The goal of a SAN is to give high-speed, fault-tolerant access to data for many different client/server and host computers. These computing devices can be as simple as just a single PC or consist of a large mainframe system.

Storage Area Technology architecture makes all storage devices available to all the current or developing network systems. These systems may include LANs or even WANs (Wide Area Networks). As more storage devices such as Network Attached Servers (NAS) are developed, it will be necessary to make them accessible from any system in the network. Because stored data is not located directly on any of a network's servers, server power is utilized for business applications, and network capacity is released to the end user. A SAN is different from a traditional network, which is commonly the LAN or WAN. A LAN / WAN is used to connect the various types of computing devices together, whether it is client to server, user-to-user or corporate network to the Internet. A LAN / WAN is the means of communications used to travel to the Internet, log on to a corporate network or just to print a document from a printer.

SANs have greatly improved in the past few years. Because of past research, SANs provide several key benefits to the enhancement of networking technology in the form of disaster recovery, scalability, fault tolerance, better data transferring performance, manageability, network performance enhancements, and better cost effectiveness.

### **2.1 Disaster Recovery**

SANs give more flexibility in Disaster Recovery. They provide a higher data transfer rate over greater distances than conventional LAN/WAN technology. So

therefore backups or recovery to and from remote locations, can be done during a fairly short period of time. Because storage devices are easy to get to by any server attached to the SAN, a secondary data center could immediately recover from a failure if a primary data center goes offline [FAR01].

## **2.2 Fault Tolerance**

Every server in a traditional network can be thought of as an “island” of data. If the server were not available, then access to the data on that server would not be possible. In traditional networks, most storage devices are physically connected to servers using a Small Computer Standard Interface connection (SCSI). SCSI is a hardware interface that enables a single expansion board in a computer to connect multiple peripheral devices such as disk drives, CD ROMs, tape drives, scanners, ect. Since access to data attached in this manner is only available if the server is up and running, a potential for a single point of failure could occur. In a SAN environment, if a server were to fail, access to the data could still be possible. This is because other servers will have access to that data through the SAN. In LAN/WAN environments, completely fault-tolerant access to data requires mirrored servers. This is an expensive approach for businesses and organizations. Also, a mirror server approach puts a large amount of traffic on the network, or needs a proprietary approach for data replication [KUM99].

## **2.3 Scalability**

Scalability within this technology is another key benefit. In today's standards of computing, the need for a great amount of high-speed storage is increasing at a significant rate. This demand creates new problems in Information Technology departments [FAR01]. A major concern is the actual placement of the physical storage devices. The older approach to connecting a storage device to a processing unit was through SCSI connections. However, SCSI has physical restrictions that would make it difficult or rather impossible to connect the many necessary storage devices to the servers at once. SAN technology breaks this physical distance barrier by letting someone locate their own storage that could be miles away as opposed to only a few feet away.

## **2.4 Transfer Performance**

SANs are an attractive alternative to current data transfer methodologies in place today [FAR01]. For example, modern SCSI can achieve data throughput rates of 160 MBps. SANs improve upon this rate at an astonishing 100 MBps full duplex. That translates into a data throughput rate of 200 MB/s when considering both directions. Additionally, parallel connections can be used with SANs further increasing performance. Due to this promising result, further efforts are being taken to increase throughput. Without a doubt, SANs are the next generation in the business of data transfer.

## **2.5 Manageability**

Many organizations have people whose duties are dedicated to specific functions. It is very common to find Novell Administrators, NT Administrators, or Unix Administrators all at one company. All of these Administrators have two things in common. One key is that they all use a network to communicate to the clients they serve. Another key is that they all require disk storage. For an organization's networking needs, you will often find a Network Manager or Network Group. They maintain the installed base of hubs, switches and routers [FAR01]. The Network Manager makes sure the network is operating effectively and makes plans for future growth.

Few organizations have groups whose duties include managing the storage resources. It is typical that a company's most typical resource, which is data storage, generally may have no formal group to manage it effectively and efficiently. Each type of system administrator is required to monitor the storage attached to their servers, perform backups and plan for growth. Storage Management in a SAN environment could offload the responsibility of maintaining the storage devices to a dedicated group. This group can perform backups over the SAN, lighten the LAN/WAN traffic for all type of servers. The group could allocate disk space to any server regardless of the type. The SAN managers could actively monitor the storage systems of all platforms and take immediate corrective action whenever needed.

## **2.6 Network Performance Enhancements**

In order to access data from a network that does not employ SAN attached storage demands, the only current solution would be the use of a traditional network. This is an inefficient method of accessing data. For example, consider an ordinary corporate intranet. Many of these intranets are organized into a multi-level structure. Now, if the user requests data from a storage device that is connected to the server from which the user has access, data must travel twice on the network. First data must be transferred from the storage device to the server. Next, the data must travel from the server to the user's workstation. This needlessly increases network traffic and could ultimately result in the slowing of the network.

Alternatively, if the data storage device were connected to a SAN, the data would travel over the user accessible network only once. What a SAN would accomplish is the prospect of deterring the traffic between the server and the Network Attached Storage (or NAS) device from the LAN network [KUM99]. Therefore, the data requested only travels across the LAN once, leading to diminished network traffic and ultimately improving performance.

## **2.7 Cost Effectiveness**

Each server requires its own equipment for storage devices. The storage cost for environments with multiple servers running either the same or different operating systems can be massive. SAN technology allows an organization or business to lower this cost through economies of scale. Multiple servers with different operating systems can access storage in RAID clusters. SAN technology allows the total capacity of storage to be allocated where it is needed. If requirements change, storage can be reallocated from devices with an excess of storage to those with too little storage. Storage devices are no longer connected individually to a server; they are connected to the SAN, from which all devices gain access to the data [KUM99].

## 3 SCSI

SCSI, or Small Computer System Interface, is a widely used I/O bus for connecting peripherals and computers, and is the foundation on which our Command Generator will operate. SCSI is often used to connect devices such as disk drives, scanners, and printers, and can operate over various types of physical media [SCS03], including Fibre Channel. The SCSI command set, mapped to Fibre Channel via SCSI-FCP, can perform various operations on connected devices, such as reading data, writing data, and requesting status information. While up to eight of these devices can share a single SCSI bus, only two of them (a target and an initiator) can communicate at once. In a nutshell, an initiator, a computer in most cases, will send a command and any associated data to a SCSI device known as the target. The target will then process the command, perform the required tasks, and respond to the initiator.

### 3.1 Bus Control Signals

At a low level, there are 47 different SCSI bus signals generated by the devices [SCS00b], each of which can take on a value of either true or false. 11 of these signals deal with bus control, while the other 36 are used for data transfer. Below is a list of the bus control signals.

- BSY (BUSY) indicates that the bus is being used
- SEL (SELECT) is used by an initiator to select a target, or vice versa
- C/D (CONTROL/DATA) indicates whether control (true) or data information is on the bus
- I/O (INPUT/OUTPUT) indicates the direction of the data transfer
- MSG (MESSAGE) signifies the MESSAGE phase
- REQ (REQUEST) indicates a target's request for an ACK handshake
- REQB (REQUEST) indicates a target's request for an ACK handshake on the optional B cable
- ACK (ACKNOWLEDGE) indicates an initiator's acknowledgment of a REQ
- ACKB (ACKNOWLEDGE) indicates an initiator's acknowledgment of a REQB

- ATN (ATTENTION) indicates the ATTENTION condition
- RST (RESET) indicates the RESET condition

## **3.2 Bus Phases**

At a given time, the SCSI bus is said to be in one of eight phases that describes its current state. Those phases are the Bus Free, Arbitration, Selection, Reselection, Command, Data, Status, and Message phases, and are noted by the values of the various signals present.

### **3.2.1 Bus Free**

When in the Bus Free phase, the SCSI bus is not busy with any I/O processes and is available for use. This phase is detected when both the SEL and BSY signals are false, meaning that no devices are busy and no devices are currently being selected.

### **3.2.2 Arbitration**

The Arbitration phase allows a SCSI device to seize control of the bus in order to proceed with an I/O process. To accomplish this, the device will wait for a Bus Free phase, and then assert the BSY signal. Subsequently, if a higher priority SCSI ID bit is true on the data bus, this means the device has lost the arbitration. Otherwise, it has won control of the bus.

### **3.2.3 Selection**

After winning the arbitration, the initiator asserts both the BSY and SEL signals, with the I/O signal false to distinguish the phase from Reselection. It also sets the data bus to the or-ed value of its SCSI ID and the desired target's ID, and asserts the ATN signal and releasing the BSY signal to false. The target will then recognize the request, and assert the BSY signal.

### **3.2.4 Reselection**

The Reselection phase occurs when a target is to reconnect to an initiator to continue a suspended process. This phase is very similar to Selection, except the roles of initiator and target are reversed. In addition, the I/O signal must be true for the initiator to recognize the phase.

### 3.2.5 Command

During the Command phase, the target will request command information from the initiator. The target sets the C/D signal to true while setting I/O and MSG to false while completing a REQ/ACK signal handshake.

### 3.2.6 Data

The Data In phase allows a target to request that the initiator receive data that it sends. The target will set the I/O signal to true, and the C/D and MSG signals to false while completing the REQ/ACK signal handshake. The Data Out phase allows the target to request that the initiator send data. In this case, the target will set I/O, C/D, and MSG to false.

### 3.2.7 Status

In the Status phase the initiator requests to send status information to the initiator. Setting the C/D and I/O signals to true and setting MSG to false during the REQ/ACK handshake signifies this phase.

### 3.2.8 Message

The Message In phase lets the target send messages to the initiator. The I/O, C/D, and MSG signals will be set to true. The Message Out phase lets the target request that the initiator send it messages, and is signified by true C/D and MSG signals and a false I/O signal. The target shall continue in this phase until the ATN signal is negated.

## 3.3 Generic SCSI Interface

The Linux operating system includes four main SCSI drivers: *sd*, *st*, *sr*, and *sg*. *Sd* interfaces with direct access devices such as hard disks; *st* interfaces with tape drives; and *sr* interfaces with CD-ROM drives. *Sg* refers to Generic SCSI and is a more low-level driver that allows the additional use of a broader spectrum of devices like scanners, printers, or re-writeable CD [SCS99]. In a potentially heterogeneous setting such as a SAN, the *sg* driver may be the best access choice.

Interfacing with a generic SCSI device is very similar to standard file I/O. Each device will have a corresponding device file in the `/dev/` directory on a Linux system. The `open()` system call creates a descriptor to the device, while `close()` will destroy it.

While the device is open, commands and data are sent to the device with the `write()` call, and data is received by using the `read()` call. In general, the data provided to the driver will consist of a command, data, and various other parameters. Data read from the driver will often include additional data, along with a status code and other information.

SCSI commands generally can measure 6, 10, 12, or 16 bytes in length. Usually a command will consist of a command code byte, logical address information, a transfer length, and various other command-specific flags. In addition, variable length data may accompany the command, such as with a WRITE command.

Data written to and read from the `sg` device takes the form of a structure called `sg_io_hdr_t`. The structure is 64 bytes long and consists of 22 fields, several of which are particularly important:

```
int interface_id: always must be set to 'S'
int dxfer_direction: specifies the direction of the data transfer (to or from the device)
unsigned char *dxferp: pointer to the data to transfer (if any)
unsigned int dxfer_len: length of data to transfer, in bytes
unsigned char *cmdp: pointer to the command
unsigned char cmd_len: the length of the command (usually 6, 10, or 12 bytes)
unsigned char *sbp: pointer to the sense buffer, which will contain any error information
unsigned char mx_sb_len: the maximum length of the sense buffer data
unsigned int timeout: time to wait for command to complete
unsigned char masked_status: status (GOOD, CHECK CONDITION, etc) of the target
```

Other fields allow further control over the behavior of the driver. The full structure definition can be found attached to this document in Appendix A.

After receiving a command, the target device responds with a single byte value called the SCSI status. GOOD is the status indicating everything has gone well. The most common other status is CHECK CONDITION, which signifies an unexpected condition that should be checked. Further information about exactly what went wrong can be found in the sense buffer; if the sense buffer size is insufficient, a "REQUEST SENSE" SCSI command can retrieve additional condition data. It is important to realize that a CHECK CONDITION may be a critical warning or just being informative (e.g. command

needed to be retried before succeeding) to fatal (e.g. "medium error" which often indicates it is time to replace the disk).

So in all cases a user application should check the various status values. If necessary the sense buffer will be copied back to the user application. SCSI commands like READ convey data back to the user application (if they succeed). The `sg` driver arranges for this data transfer from the device to the user space, if necessary. The interface makes special device handling possible from user level applications. Therefore, kernel driver development, which is more risky and difficult to debug, is not necessary.

However, if the driver is not programmed properly, it is possible to create serious errors affecting the SCSI bus, the driver, or the kernel. Therefore, it is important to properly program the generic driver and to first back up all files to avoid losing data. Another useful precaution before running a program that uses the `sg` driver is to issue a `sync` command to ensure that any buffers are flushed to disk, minimizing data loss if the system hangs. Another advantage of the generic driver is that as long as the interface itself does not change, all applications are independent of new kernel development. In comparison, other low-level kernel drivers have to be synchronized with other internal kernel changes. Typically, the generic driver is used to communicate with new SCSI hardware devices that require special user applications to be written to take advantage of their features. The generic interface allows these to be written quickly.

## 4 Fibre Channel

Currently, most SAN implementations are built with Fibre Channel architecture. Fibre Channel is a technology standard that is used to transfer data from one node to another in a network at very high-speeds. Current implementations transfer data at one Gbps or two Gbps. 10 Gbps data rates have already been tested and many companies have products in development that will support this [FIB94]. Aside from its high speed, it is also very reliable, very scalable and very flexible. Fibre Channel is usually used in attaching servers with storage device, such as RAID array or tape backup device. Fibre Channel is very helpful because it enables us to add more data storage without disrupting the system operations. Fibre Channels are designed to work with either copper wires or with fibre optic cables. Copper wires are used for only short distances. Fibre optic cables are used for longer distances. They can support transferring of data up to six miles away with a minimal loss of speed. Fibre Channel is capable of supporting multiple protocols and a variety of topologies; making it the most versatile data transfer technology available [FIB94]. Fibre Channel combines storage I/O channels with networking to create a high-bandwidth networking technology that is appropriate for today's expanding data needs.

Fibre Channel is a high performance serial link with its own protocol, and it also supports other higher-level protocols such as the FDDI, SCSI, HIPPI and IPI. The Fibre Channel standard addresses the need for very fast transfers of large amounts of information. The fast (up to 1 Gbps) technology can even be converted for LAN. Another advantage of Fibre Channel is that it gives users one port that supports both channel and network interfaces, relieving the computers from large number of I/O ports. FC provides control and complete error checking [FIB94].

There are two basic types of data communication between processors and between processors and peripheral devices. They are channels and networks. A channel provides a direct or switched point-to-point connection between the communicating devices. A channel is typically hardware-intensive and transfers data at a high speed with very little overhead. In contrast, a network is a collection of distributed nodes (like

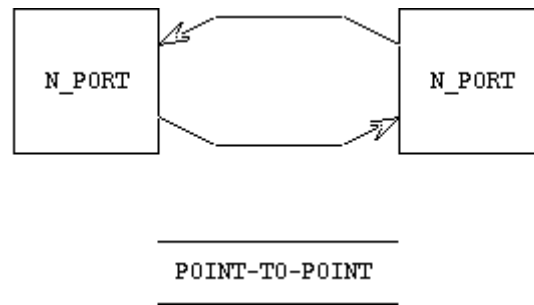
workstations, file servers or peripherals) with its own protocol that supports interaction among these nodes. A network has relatively high overhead since it is software-intensive, and is therefore slower than a channel. Networks can handle a more extensive range of tasks than channels, as they are designed to operate under less rigid conditions than channels. Fibre Channel attempts to combine the better of these two methods of communication into a new I/O interface that meets the needs of channel users and also networks users.

## **4.1 Fibre Channel Topology**

One of Fibre Channel's best qualities is its flexible topology. Flexibility is a feature that is of great importance to any kind of network. The topology can be selected depending on system performance requirements or packaging options. There are currently three different types of Fibre Channel topologies: Point-to-Point, Arbitrated Loop, and Fabric.

### **4.1.1 Point to Point**

A Point-to-Point topology (Figure 4-1: Point to Point Topology [FIB95]) is the simplest of the three choices. It contains two Fibre Channel devices connected directly together. There is no sharing of the channel, which permits the devices to benefit from the total bandwidth of the link. Point-to-point offers total bandwidth between the two connected systems. Data may flow directly between two connected endpoints, or multiple devices may share a switch that is configured for several point-to-point configurations. This connection may be established for the duration of an operation period, or when there is a need to communicate.



**Figure 4-1: Point to Point Topology [FIB95]**

### 4.1.2 Arbitrated Loop

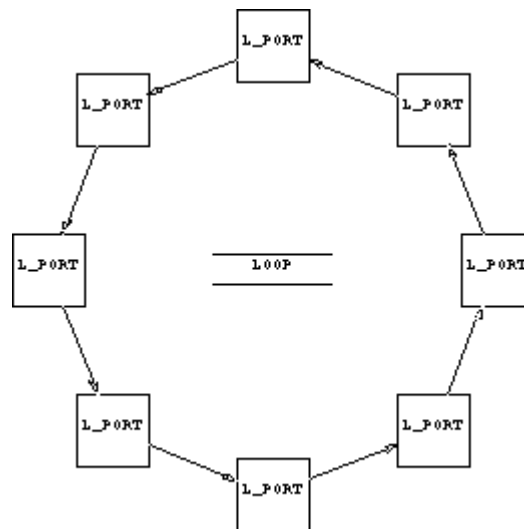
Arbitrated loop (Figure 4-2: Arbitrated Loop Topology [FIB95]) is the most common Fibre Channel topology. It is used for storage interconnection, the interconnection of small numbers of devices, and intermittent high bandwidth situations. With all these network benefits, it is no surprise that it is the most complicated structure. However, an arbitrated loop topology is the most cost-effective means of connecting up to 127 ports in a single network without the need of a switch. Unlike the other two topologies, the channels are shared between all of the devices. For a loop to operate, all devices must specifically support the architecture.

Another disadvantage of this topology is that they are inefficient in large configurations. This is due to the fact that every node in the loop must look at the data regardless of the destination. Since each node is logically connected in a ring, cable induced delays may occur with larger configurations. It is suggested that arbitrated loop be used when the total number of connected nodes is less than 30 and the maximum distances between nodes is 100 meters or less [FIB95].

An arbitrated loop scheme has a simple but yet an effective way of communication between nodes. When a node or device is ready and capable of sending data, it must first arbitrate and gain total authorization throughout the loop. It accomplishes this by sending the target node an OPEN (OPN) signal to the target node. From that point on, the two nodes carry on a Point-to-Point type of relationship.

However, if more than two nodes or device try to gain control at the same time, the situation gets treated in a prioritized manner. Before a node even sends out a (OPN)

signal to a destination device, it had first sent out and received its own primitive signal for authorization purposes within the loop. This primitive signal is represented as (ARBx) where x is equal to the physical address of the device. It is represented as (AL\_PA). The x values of the ARB Primitive Signals are compared. When an arbitrating device receives another device's ARBx, the ARBx with the numerically lower AL\_PA is forwarded, while the ARBx with the numerically higher AL\_PA is blocked. Thus, the device with the lower AL\_PA will gain control of the loop first. Once that device relinquishes control of the loop, the other device can have a chance.



**Figure 4-2: Arbitrated Loop Topology [FIB95]**

### 4.1.3 Fabric Switch

The Fabric topology (Figure 4-3: Fabric Switch Topology) is used to connect many devices in a cross-point switched configuration. The switched fabric connection can interconnect large numbers of devices, sustain high bandwidth requirements, connect devices that run at different speeds, and provide cable matching. The main benefit of this topology is that many devices can communicate at the same time since the channels are not shared. This connection may be established for the duration of an operation period, or when there is a need to communicate.

The fabric or switched topology consists of several switches that support the end system nodes. When the fabric from a transmitting port receives data, the frame header of the incoming data determines the destination. The fabric will then transfer the data to an

adjacent node to be delivered, or deliver the appropriate information to the receiving port. This topology type allows for error detection and is easily expandable. The drawback however is that it requires a specific Fibre Channel switch. The switched fabric topology gives the greatest connection capability and largest total combined throughput. Each device is connected to a switch and receives a non-blocking data path to any other connection on the switch. This would be equivalent to a dedicated connection to every device. As the number of devices increases to occupy multiple switches, the switches are in turn connected together.

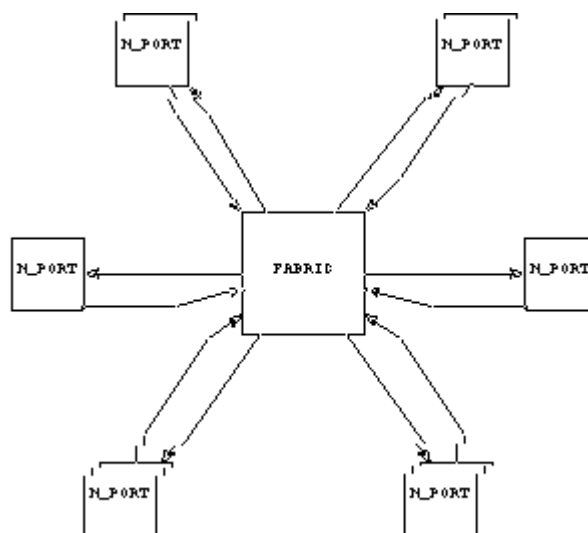


Figure 4-3: Fabric Switch Topology

## 4.2 Fibre Channel Layers

Fibre Channel allows simultaneous transmission of different protocols over a single optical-fibre pair and it can allow a number of existing services, such as network, point-to-point, and peripheral interfaces, to be accessed over a single medium using the same hardware connection. Fibre Channel also provides control and complete error checking. What allows a Fibre Channel network this flexibility is its structure made up of multiple layers.

The Fibre Channel structure is defined as a multi-layered stack of functional levels, not unlike those used to represent network protocols, although not mapping directly to OSI layers. The layers of the Fibre Channel standard define the physical media

and transmission rates, encoding scheme, framing protocol and flow control, common service, and the upper-level applications interfaces [FIB94]. The five layers are FC-0, FC-1, FC-2, FC-3, and FC-4.

#### **4.2.1 FC-0**

The lowest layer is the FC-0 layer. It specifies the physical link in the system, including the media, transmitters, receivers, and connectors that can be used with Fibre Channel. This also includes electrical and optical characteristics, transmission rates, and other physical components of the standard. The physical level is designed to be able to use a large number of technologies to meet the widest range of system requirements. An end-to-end route can use different link technologies for increased performance and decreased cost, and systems integrators can tailor an installation to meet the specific needs of their customers [FIB95].

FC-0 also specifies the (OFC) Open Fibre Control system. It is a safety system used to control the optical power level of SW laser data links when or if an open fibre condition occurs. This safety feature is required because the optical power levels in this kind of system exceed the limits defined by the laser safety standards [FIB95]. Whenever this open fibre condition occurs in the link from the sending port, the receiver port detects it and pulses its laser at a low duty cycle within the laser safety requirements. The receiver at the other port detects the pulsing signal and itself sends pulses within the specified laser safety range. If the open fibre condition is restored, receivers from both ports receive the pulsing signals that will result in a double handshaking procedure to restore normal transmission after a couple of seconds.

#### **4.2.2 FC-1**

FC-1 defines all transmission protocols including serial encoding and decoding rules, special characters and error control. Every 8 bits of data are encoded into 10 bit Transmission Character. A Transmission Word is composed of four contiguous Transmission Characters. The Transmission character is used to ensure that clock recovery is possible by having enough transitions present in the serial bit stream.

Character conversion is accomplished by taking an uuencoded information byte that is made up of eight information bits logically labeled A,B,C,D,E,F,G,H and a control character Z. E, D, C, B, and A represent each binary value. It is converted into the form

xx, which is a decimal representation of the binary value, since  $2^5-1=31$  requires at most two decimal digits. H, G similarly is converted into the form y, which is a one digit decimal value. D represents the control character Z for data-type or K for special-type. The resulting combination of this information forms a name in the form Zxx.y that represents a valid Transmission Character. After transmission the D-type Transmission Characters are decoded into one of 256 eight-bit combinations. Any K-type Transmission Characters are used for protocol management functions. All other codes besides D or K types are invalid.

The physical layer uses a Running Disparity (RD). This is a binary number that is calculated based on the number of 0's and 1's in the two sub-blocks. The first sub-block is the first six bits of the Transmission Character and the second sub-block is the last four bits of the Transmission Character [FIB94]. A new RD is calculated at the transmitter and receiver and if the RD values are not the same, then a disparity violation condition is indicated.

### 4.2.3 FC-2

FC-2 is called the signaling protocol level. It is responsible for breaking the data to be transmitted into, and reassemble the frames after transport. It specifies the framing rules of the data to be sent from one port to another, ways for controlling the three service classes, and controlling the sequence in which the data is transferred. All frames in a transfer have a sequence number from 0 to N so that the receiver is able to tell if a frame is missing and exactly which frame(s) is missing. There are five different building blocks defined to provide efficient means of data transport across a link. They are Ordered Sets, Frames, Sequences, Exchanges, and Protocols.

Ordered Sets are four byte transmission words containing data and special characters which have a special meaning. Ordered Sets provide the availability to obtain bit and word synchronization, which also establishes word boundary alignment. An Ordered Set always begins with the special character K28.5. Three major types of Ordered Sets are defined by the signaling protocol. They are Frame delimiters, Primitive signals, and Primitive sequences.

The Frame delimiters, the Start-of-Frame (SOF) and End-of-Frame (EOF) Ordered Sets are Ordered Sets that immediately begin or follow the contents of a Frame.

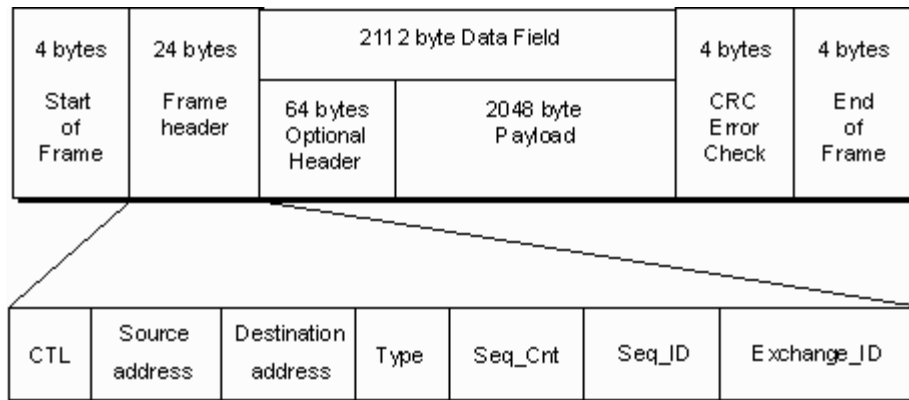
There are multiple SOF and EOF delimiters defined for the Fabric and N\_Port Sequence control.

The two Primitive Signals: Idle and Receiver Ready (R\_RDY) are Ordered Sets designated by the standard to have a special meaning. An Idle is a Primitive Signal transmitted on the link to indicate an operational Port facility ready for Frame transmission and reception. The R\_RDY Primitive Signal indicates that the interface buffer is available for receiving further Frames [FIB00].

A Primitive Sequence is an Ordered Set that is transmitted and repeated continuously to indicate specific conditions within a Port or conditions encountered by the receiver logic of a Port [FIB00]. When a Primitive Sequence is received and recognized, a corresponding Primitive Sequence or Idle is transmitted in response. Recognition of a Primitive Sequence requires consecutive detection of three instances of the same Ordered Set. The Primitive Sequences supported by the standard are Offline (OLS), Not Operational (NOS), Link Reset (LR) and Link Reset Response (LRR)

The basic building blocks of an FC connection are the Frames. The Frames contain the information to be transmitted (Payload), the address of the source and destination ports and link control information. Frames are broadly categorized as Data frames and Link control frames. Data frames may be used as Link Data frames and Device Data frames, link control frames are classified as Acknowledge (ACK) and Link Response (Busy and Reject) frames. The primary function of the Fabric is, to receive the Frames from the source port and route them to the destination port. It is the FC-2 layer's responsibility to break the data to be transmitted into Frame size, and reassemble the Frames.

Each Frame begins and ends with a Frame Delimiter. The Frame Header immediately follows the SOF delimiter. The Frame Header is used to control link applications, control device protocol transfers, and detect missing or out of order Frames. An optional header may contain further link control information. A maximum 2112 byte long field (payload) contains the information to be transferred from a source N\_Port to a destination N\_Port. The 4 bytes Cyclic Redundancy Check (CRC) precede the EOF delimiter. The CRC is used to detect transmission errors [FIB94].



**Figure 4-4: Frame Structure [FIB94]**

A set of one or more related Frames transmitted in a unidirectional way from one N\_Port to another is known as a sequence. Each Frame within a sequence is uniquely numbered with a Sequence Count. Error recovery, controlled by an upper protocol layer is usually performed at Sequence boundaries [FIB00]. An Exchange is composed of one or more nonconcurring sequences for a single operation. The Exchanges may be unidirectional or bidirectional between two N\_Ports. Within a single Exchange, only one sequence may be active at any one time, but Sequences of different Exchanges may be concurrently active [FIB00]. The Protocols are related to the services offered by Fibre Channel. Protocols may be specific to higher-layer services, although Fibre Channel provides its own set of protocols to manage its operating environment for data transfer.

#### **4.2.4 FC-3 and FC-4**

The FC-3 level of the FC standard is intended to provide services for multiple ports on one node. The common services required for advanced features are striping, hunting group, and multicasting. Striping, multiplies bandwidth using multiple N\_ports in parallel to transmit a single information unit across multiple links. Hunting for groups gives the ability for more than one Port to respond to the same alias address. This improves efficiency by decreasing the chance of reaching a busy N\_Port. Multicasting delivers a single transmission to multiple destination ports. This includes sending to all N\_Ports on a Fabric (broadcast) or to only a subset of the N\_Ports on a Fabric.

The FC-4 layer is the highest level in the FC structure defines the application interfaces that can execute over Fibre Channel. It specifies the mapping rules of upper

layer protocols using the FC levels below. Fibre Channel is equally adept at transporting both network and channel information and allows both protocol types to be concurrently transported over the same physical interface [FIB94].

The following network and channel protocols that are currently supported on this layer are:

- Small Computer System Interface (SCSI)
- Intelligent Peripheral Interface (IPI)
- High Performance Parallel Interface (HIPPI) Framing Protocol
- Internet Protocol (IP)
- ATM Adaptation Layer for computer data (AAL5)
- Link Encapsulation (FC-LE)
- Single Byte Command Code Set Mapping (SBCCS)
- IEEE 802.2

### **4.3 Flow Control**

The concept of flow control deals with the problem where a device receives frames faster than it can process them. When this happens, the result is that the device is forced to drop some of the frames. Fibre Channel has a built-in flow control solution to this problem. The idea is simple enough. 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, so each 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, buffer-to-buffer and end-to-end.

#### **4.3.1 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. Both ports on the link exchange 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. For example, when ports A and B log into each other, A may report that it is willing to handle 4 frames from B; B might report that it will accept 8 frames from A. Thus, B's BB\_Credit is set to 4, and A's is set to 8.

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 cannot transmit another frame until it receives an R\_RDY [REC03].

#### **4.3.2 End to End**

End-to-End flow control is not concerned with individual links, but rather the source and destination N\_Ports. The concept is very similar to buffer-to-buffer flow control. 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 [REC03].

### **4.4 Classes of Service**

To make certain of efficient transmission of different types of traffic, Fibre Channel defines six classes of service. The class greatly depends on the type of data to be transmitted. The major difference between the classes is the types of flow control used. Users select service classes based on the characteristics of their applications, like packet length and transmission duration, and allocate the services by the Fabric Login protocol.

Class 1 is a service that provides dedicated connections. Once established, a Class 1 connection is retained and guaranteed by the Fabric. This service guarantees the maximum bandwidth between two N\_Ports. Thus, this is the best class for sustained,

high throughput transactions. In class 1, frames are delivered to the destination Port in the same order as they are transmitted [SEA03].

Class 2 is a frame-switched, connectionless service that allows bandwidth to be shared by multiplexing frames from multiple sources onto the same channel or channels [SEA03]. The Fabric may not guarantee the order of the delivery and frames may be delivered out of order. This service class can be used, when the connection setup time is greater than the latency of a short message. Both Class 1 and Class 2 send acknowledgment frames confirming frame delivery. If delivery cannot be made due to congestion, a Busy frame is returned and the sender tries again.

Class 3 is connectionless without delivery acknowledgement [SEA03]. The Class 3 service is identical to Class 2, except that the frame delivery is not confirmed. Flow control is managed only on buffer level. This type of transfer, known as datagram provides the quickest transmission by not sending confirmation. This service is useful for real-time broadcasts where timeliness is crucial and information not received in time is valueless.

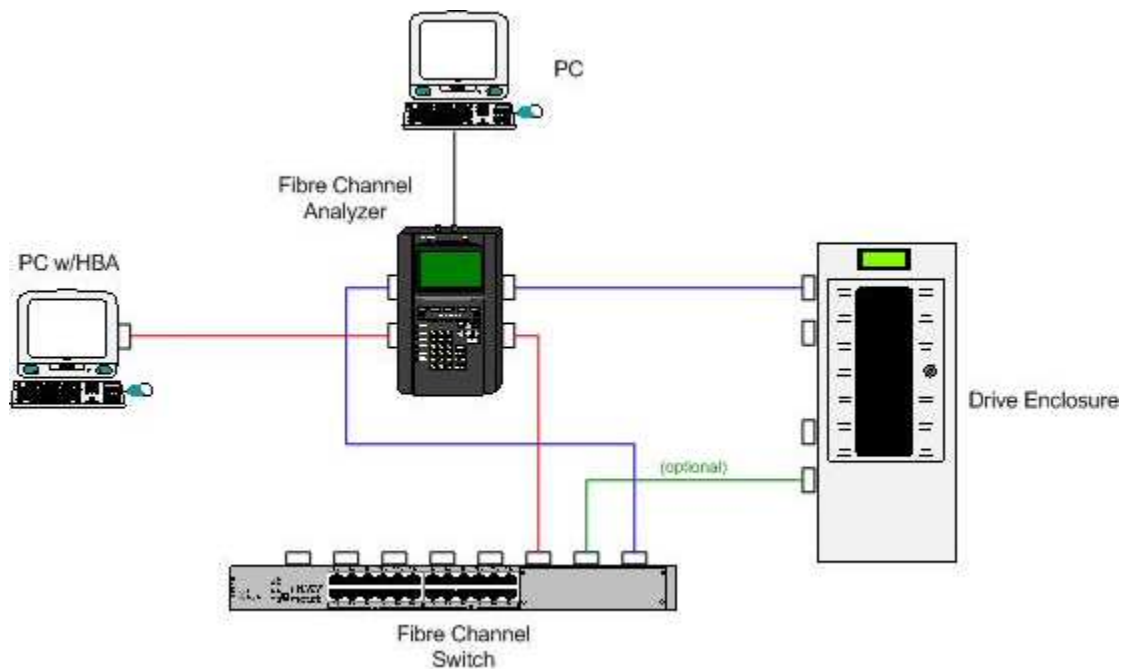
Class 4 provides fractional bandwidth allocation of the resources of a path through a Fabric that connects two N\_Ports [SEA03]. Class 4 can be used only with the pure Fabric topology. One N\_Port will set up a Virtual Circuit (VC) by sending a request to the Fabric indicating the remote N\_Port as well 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 [SEA03]. 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 FC\_RDY Primitive Signal. Intermixing of Class 2 and 3 frames is mandatory for devices supporting Class 4.

Class 5 was developed for “just-in-time service” [FIB94]. However, it is still undefined, and maybe even scrapped altogether. It is not mentioned in any of the FC-PH documents.

Class 6 provides support for multicast service through a Fabric. 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. The multicast server sets up individual dedicated connections between the original N\_Port and the entire 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 also a well-know address. 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 [SEA03].

## 5 FC Analyzer

The FCTracer is a Fibre Channel test and debug platform manufactured by Computer Access Technology Corporation. With the FCTracer module plugged in as one of two modules, CATC's Universal Protocol Analyzer System (UPAS), a unit which records and reports various types of traffic, can record up to two gigabytes of traffic at a time. This data can be uploaded via USB to a host PC [CAT04]. Figure 5-1: FCAnalzer shows the configuration of the SAN at General Dynamics:



**Figure 5-1: FCAnalzer**

Despite the high speeds of one and two-gigabit Fibre Channel, UPAS is able to record transmitted frames transparently at full wire speed using its Bus-Engine technology and active port bypass circuitry. Bus-Engine also employs various forms of

filtering and triggering that permits UPAS to selectively record traffic. Hence, the two gigabytes of recorded data need not be contiguous.

Once a recording session has completed, the UPAS will upload the data to a PC via a USB connection. Subsequently, CATC's Trace software is used to analyze the traffic. The tool allows the user to view specific information at multiple levels, including invalid CRCs, running disparity errors, and invalid 10-bit codes at FC-1; frame, sequence, and exchange violations at FC-2; and FCP mapping and SCSI errors at FC-4 [CAT04]. The software can display individual primitives, data frames, or decoded sequences on separate rows using colors and graphics to show the various fields composing each unit of data.

## **5.1 Recording a Trace**

The FCTracer has the responsibility of recording Fibre Channel frames. Using the FC analyzer software, the user can pick and choose certain data that we consider important enough to be recorded. We can do this by configuring the application to create event-triggers or by increasing or decreasing the amount of memory allocation for recording. Also by changing and manipulating the configurations of the FCTracer, we are able to interact with other Fibre Channel devices in different ways.

There are three different recording type options that allow the user to set how FCTracer begins and ends a recording. The options are Manual Trigger, Snapshot, and Event Trigger.

A Manual Trigger is a recording whose Trigger point is activated by pressing the Trigger button on the front panel. Pressing the Start button on the tool bar begins recording. Recording continues in a circular manner within the limits set by the buffer size. Recording ends when the Stop button is clicked on the Tool Bar or when the Trigger button is pressed on the analyzer's front panel. If the user presses the Trigger button, recording will continue until the post-trigger memory has been filled. For example, if the user set the Recording Buffer to 10 MB and the Trigger Point to 50%, pressing the Manual Trigger button will cause the analyzer to preserve 5 MB of pre-trigger and 5 MB of post-trigger.

A Snapshot is a fixed-length recording [CAT04]. The user is able to set the size in two different ways. Adjusting the Buffer Size slide bar in the Recording Options General dialog box is one method. Another way is by clicking the Stop button any time during the recording session. Recording starts by clicking the Start button on the Tool Bar and ends when either the selected buffer size is filled or when the Stop button is pressed.

Another recording option is the Event Trigger. An Event Trigger is a recording defined by a specific event or events. Before recording begins, the user would have to define the event trigger in the Recording Rules page in the Recording Options dialog box. The recording then starts by clicking the Start button on the Tool Bar. Recording continues in a circular manner within the limits set by the buffer size. Once the trigger event occurs, some post-trigger recording occurs, and the recording ends.

## **5.2 Conventional Recording**

In a conventional recording, the entire trace is recorded and stored in the analyzer buffer before it is saved onto the host PC. Because of this, recordings are limited to the size of the analyzer buffer. This is a good option to select if the user has the intention of creating a moderate size recording. The user sets the buffer size and the trigger position then soon starts the recording. When the analyzer's buffer reaches the two gigabyte capacity, the traffic is uploaded to the host PC. Spooled recording is the best option for larger recordings [CAT04]. Normally, this is selected when the recording records for days at a time. In a Spooled Recording, traffic is uploaded every so often to the host PC. This frees up memory in the analyzer buffer and allows the recording to continue. The Buffer Size box has a slide bar for adjusting the recording buffer size from 1.6 megabytes to 2048 MB. This option is used for setting the memory for a Conventional recording. The Recording Type option determines how this buffer is used. Although there are 2048 MB of physical memory in the Analyzer, the efficiency of the recording ranges from 2:1 to 4:1 ratios of physical memory to actual Fibre Channel traffic [SEA03]. Shorter Fibre Channel packets generate a less efficient recording. The non-traffic portion of physical memory is used for control and timing information.

Multi-segmenting allows these larger spooled recordings to be split up into smaller traces. This option tells the analyzer to segment the trace into multiple files and create an index file called data.mlt. This file summarizes the starting and finishing frame for each segment. The multi-segmenting option is useful for very large recordings and for host PCs with not that much memory. Multi-segmenting gives a PC with limited memory a way to open recordings that would otherwise be too large to open. The only downside to multi-segmenting is that it limits the scope of reports such as Traffic Summary, Bus Utilization, and Error Summary to each of the segments. The user will not be able to perform summary statistics on the full recording. The default value for this option is 64 MB.

The multi-segment option creates an index file and segmented trace files. The default name of the index file is dataXYZ.mlt, where XYZ is the last three digits of the analyzer's serial number. Thus, for example, if the analyzer had the serial number 111, the index file would be called data111.mlt. The index file and the segmented trace files are stored in a directory named after the index file. The directory is named `indexfilename_mlt_files`. Below this directory are additional sequentially numbered sub-directories that house the segmented trace files. There can be up to 100,000 trace files in this directory. These sub-directories have simple numerical names which are 00000 - 00999. Each of these subdirectories can hold up to 100 sequentially numbered segment files. In all, the entire directory structure can hold up to 10 million files [CAT04].

### **5.3 Spooled Recording**

In a Spooled recording, uploading begins from the analyzer to the host PC when the recording is begun. As traffic is uploaded, the analyzer memory gets freed, creating space for recording additional traffic. Thus, recording can continue for long periods of time, and create file lengths that are larger than two gigabytes.

With Spooled Recordings, there is an option for setting the recording length based on time or on the recording size. The 'Record for' button enters the duration of the recording in days, hours, minutes, and seconds. The 'Record' button enters the amount of traffic in megabytes that user wants the analyzer to record. Selecting this option will create a fixed length recording that begins as soon as the REC button on the menu bar is

clicked. The ‘Until Triggered, plus’ button enters the amount of traffic in time that the user wants the analyzer to record following an event trigger. Recording time units are days, hours, minutes, and seconds. The ‘Until Triggered, plus’ enters the amount of traffic in megabytes to record following an event trigger. Performance issues occur in a spooled recording if the analyzer buffer fills faster than the trace can be uploaded to the host PC. If this happens, the analyzer will briefly suspend recording until some of the buffer is free [CAT04]. When recording is suspended, gaps will appear in the trace. These gaps will appear as entries in the trace.

During the recording, it is possible to see if gaps are likely to happen by reading the status bar at the bottom of the screen. Gaps are caused by a number of factors. Some factors are the number of channels being recorded, the absence of filtering, the performance of the host PC and also the amount of traffic produced by the devices under test. There are several ways to prevent gaps. The user can experiment with the setup configuration to determine the best scheme. This may include filtering out primitives, turning on data truncation and filtering out loop initializations in Fibre Channel. The Status bar has two fields that the user can use to determine if gaps are occurring or are about to occur [SEA03].

## **5.4 Recording Rules**

Recording Rules lets the user set triggers and filters. Triggers and filters are conditions created to determine the data to be or not to be recorded. The steps to creating trigger and filter conditions can be found in the main display area of the FCTracer. The user creates conditions by dragging buttons onto the Main display area from the Available Events area. He could then create additional conditions by right clicking on a button and selecting options from a pop-up menu.

The Recording Rules page can be thought of as a chalkboard in creating a graphical model of the events and actions. Basically, the user is in charge of creating a visual representation of the rules that the analyzer must follow during a recording when it encounters events that are specified. Rules can be either simple or very complex. Creating a rule involves four steps:

- Step 1** Create event buttons
- Step 2** Click the small buttons on the Event buttons to select the channel that the analyzer should record
- Step 3** Move the Event buttons to the Main Display area
- Step 4** Assign an action such as Trigger to the events buttons.

To create a rule, one or more Event buttons must be created. After creating Event buttons, they are automatically placed in the Available Events area. The following steps show the process.

- Step 1** Click the New Event button.
- Step 2** Select an Event from the menu. The selected Event will appear in the Available Events area.
- Step 3** To move the button to the main display area, drag the event button.

At the bottom of each event button are eight small buttons that represent analyzer channels. Selecting a channel button tells the analyzer which channel it should watch. When the user drags an event button to the main display area, there are two places he can put the button: in a cell marked "Global State" and in another marked "Drag an event here to add an event sequence."

The Global State cell is a placeholder for rules that are active throughout the recording [CAT04]. If an event button such as Error is placed here and is then assigned an action such as Trigger, the analyzer will always look for Errors and trigger when an error is found. Another main display area is the Drag an event here to add an event sequence. This cell is used to create conditional rules called Event Sequences. An Event Sequence is a chain of events leading to some outcome such as triggering - such as "If x is followed by y, trigger."

SCSI Command Properties Dialog lets the user specify parameters for the selected SCSI Op-Code. To enter a parameter under this recording option, the user must enter a value into the Mask-Match fields. The "XXX" value means that the particular bit is being ignored by the hardware when trying to match a data frame. The user can run the mouse

over blue field captions to get a tool-tip string. If the blue caption is clicked then the user will be given either a text box to enter a hex value for this field or a combo box to select one of the values or type in an arbitrary hex value. If the value of the Op-Code field is changed, the rest of the fields will be reformatted according to the new Op-Code layout. If NOT is specified an event match will be generated in one of two cases: (a) when a SCSI Command with the Op-Code other than the selected one occurs, and (b) when SCSI Command with the selected Op-Code occurs but other fields do not satisfy the specified Mask-Match conditions [SEA03].

The Global State cell is used to create conditions that are active at all times.

These conditions can be considered as default conditions. The Global State cell is capable of creating simple event triggers and filters. For example, "Trigger when you see an *xxx* error," or "filter out all *xyz* primitives." For most simple trigger conditions, this is the most appropriate cell to use. To create a simple condition that is active at all times, place an event button in the Global State cell:

- Step 1** Click the New Event button. An Events menu opens.
- Step 2** Select an event from the menu. The event will appear as a button in the Available Events area on the left.
- Step 3** Drag the event button to the cell marked "Global State."
- Step 4** Right-click on the button (i.e., not the cell). A pop-up menu appears.
- Step 5** Select Trigger from the menu. An arrow will project from the error button and point to a cell marked Trigger.

Another useful action in recording is Creating Multiple Event Conditions in the Global State Cell. When multiple buttons are placed in the Global Cell, it creates an "AND" condition or an "OR" condition depending on the actions selected. In creating an OR Condition when two or more buttons in the Global State cell are assigned the same action, the analyzer will search for all of the events and perform the action on whichever event it sees first. In creating an AND condition when multiple events are placed in the Global State cell and assigned different actions, an AND statement is created [SEA03].

Filter In and Filter Out is also a commonly used condition for recording. A filter causes the analyzer to filter in or out specified events from the recording. If events are filtered out of the recording, they are excluded and not simply hidden from the trace. The purpose of filtering is to preserve recording memory so it can conduct longer recording sessions and exclude events that are not of interest. To Filter In or Out traffic:

**Step 1** Click the New Event button. The New Event menu opens.

**Step 2** Select an event from the menu.

**Step 3** Drag the event into the Global State cell.

**Step 4** Right-click on the button. A pop-up menu opens.

**Step 5** Select Specify Action(s)

**Step 6** Select Filter Out (for example).

The analyzer is now configured to filter out the selected event.

## ***5.5 Reading and Interpreting Trace Data***

The FCTracer software displays color and graphics extensively to fully document the captured traffic. Colors scheme is customized according to the users. Sequences are shown on separate rows, with their individual fields both labeled and color-coded. Frames are time-stamped, and highlighted to show the device status such as master or slave.

Fields that has small triangles in their top left corners can be expanded to display greater detail or collapsed to a compact view. One way to do this is through the trace navigator tool. The Navigator's primary function is to give control over the amount of trace displayed in the main window. The Navigator is a bar that can be displayed on the right side of the window. The bar represents the entire trace. By dragging the top and bottom of the Navigator bar the user can restrict how much traffic is displayed in the main window. Navigator also shows where errors occur in the trace. Enabling the Errors to be displayed can be done by right clicking in the Navigator and selecting Errors from the pop-up menu. Errors will then display as horizontal lines running across the width of

the Navigator. The Navigator bar is made up of three parts: Pre-Trigger traffic, Post-Trigger traffic, and errors.

FCTracer has three decode levels: Frames, Sequences, and Exchanges, The FCTracer default decode/display level is the Frame. FCTracer automatically decodes FC operations and helps developers to focus on the information that is most important to their own specific environment. At the Sequence level, FCTracer decodes Basic and Extended Link Services, GS-3 management server transactions, and FCP operations. The Exchange level intelligently group sequences that are part of a common exchange for easy identification by the user. By accurately interpreting upper level protocols, users can focus on solving problems instead of spending time looking up information in the Fibre Channel Specification. By grouping Frames logically within their related sequences and exchanges, users are free from reconstructing FC-4 level operations in their head [CAT04].

The user may decode and display traffic through the toolbar or by selecting a decode level from the Display Options dialog box. The decode buttons on the toolbar perform the following functions: Fra (Display Frames) Seq (Display Sequences) Xch and (Display Exchanges). Fibre Channel's high data rate makes the ability to selectively capture the specified traffic important. With real-time filtering of primitives or user definable frame types, FCTracer preserves recording memory and allows users to focus their analysis on the essential Fibre Channel traffic. FCTracer also uses a compression algorithm to store primitives for even further reduction of trace memory requirements [CAT04]. Powerful search options allow users to find any primitive, or header field in the trace. Many of the most useful searches have already been indexed during upload. "Point and click" tools allow hiding of almost every element within a trace including primitives or traffic from individual channels. As display options are customized, it's easy to save them to a configuration file and share within a development team [CAT04].

FCTracer includes many mechanisms to measure and report on Fibre Channel traffic. For each sequence, the user can see an absolute time-stamp, time-delta between sequential time-stamps, or idle time between packets on the same link. The user may specify whether the timestamp is displayed in nanoseconds, microseconds, milliseconds, or seconds. The FCTracer's Traffic Summaries provide statistics on the occurrences of

errors, primitives, frames, sequences, and exchanges. Users can evaluate these metrics at a glance or use them to find the way through the trace. FCTracer also features graphical bus utilization and throughput reports that provide a histogram of activity dynamically linked to packet level details [CAT04]. The FCTracer Timing and Bus Usage Calculator provides various throughput and performance metrics on a user-specified subset of the trace.

## 6 The SCSI Command Generator

The design process of the SCSI application is a significant part of this project. The design and development phase is responsible for producing a tool that is practical, versatile, and useful with respect to the focus of the overall project. The requirements of the application and the basic key functions that make up this application are extensively discussed throughout this section. The interface design is also discussed in detail.

### 6.1 Requirement Elicitation

The requirements phase of this project is vital in that it defined its contributions to the overall project: learning and understanding more about the state of the art of SAN and Fibre Channel networks.

There were several issues involved in which direction to take this project and therefore lead to many changes in direction. The final direction chosen was to create a tool to generate various SCSI commands to analyze on a Fibre Channel medium. It was agreed that control over individual SCSI commands would generate interesting Fibre Channel traffic. SCSI commands are commands issued by a protocol known as SCSI, or Simple Computer System Interface. It is a protocol for connecting peripherals and computers. SCSI is often used to connect devices such as disk drives, scanners, and printers, and can operate over various types of buses [SCS03], including Fibre Channel. The SCSI command set, mapped to Fibre Channel via SCSI-FCP, can perform various operations on connected devices, such as reading data, writing data, and requesting status information. The SCSI Command Generator (SCG) will in essence issue low-level SCSI commands to devices attached to the SAN in order to observe respective SCSI responses with the FCTracer.

### 6.2 Selected Commands

In a potentially heterogeneous setting such as a SAN, the *sg* driver is the best access choice. Therefore it was most convenient to choose types of commands that utilized this *sg* driver. The list of SCSI commands that were selected includes (the

numbers in parentheses represent the size of the command in bytes when the command can take different sizes):

- Inquiry
- Read(6)
- Read(10)
- Write(6)
- Write(10)
- Read Capacity
- Mode Sense(6)
- Mode Sense(10)
- Test Unit Ready
- Read Defect Data(10)
- Read Defect Data(12)
- Report LUNS
- Verify
- Send Diagnostic

These commands are compatible with Direct Access Device models and Removable Mediums because *sg* is a more generic low-level SCSI driver and allows the additional use of a broader spectrum of devices like scanners, printers, or re-writeable CD [SCS99]. Similarly, in the General Dynamics configuration we are accessing two different Seagate hard drives placed within a JMT Fortra 6 bay drive enclosure. Following are descriptions of the selected commands [DEM00].

### **6.2.1 INQUIRY**

One of the most basic SCSI commands is the INQUIRY command (Figure 6-1: INQUIRY Command Structure). It is used to identify the type and make of the SCSI devices. The command along with its special operation code, get sent to the device.

Bit/Byte	7	6	5	4	3	2	1	0
0	Operation Code (12h)							
1	Logical Unit Number			Reserved			EVPD	
2	Page Code							
3	Reserved							
4	Allocation Length							
5	Control							

**Figure 6-1: INQUIRY Command Structure**

The device then responds with information that includes type, standard level, and often the vendor's identification, model number, and other useful information. This command is implemented in the `do_inquiry()` function of the program.

### 6.2.2 READ

There are two different types of READ commands implemented in SCSI. The first is the READ(6) command (Figure 6-2: READ(6) Command Structure). It has the task of requesting that the device server transfer data to the application client. The most recent data value written in the addressed logical block is returned. The cache control bits are not provided for the READ(6) command. These cache control bits offer explicit control that can be found in the READ(10) command (Figure 6-3: READ(10) Command Structure). Block devices with cache memory may have values for the cache control bits which may affect the READ(6) command. Therefore, if precise control is required, the READ(10) should be used.

Bit/Byte	7	6	5	4	3	2	1	0
0	Operation Code 08h							
1	Logical Unit Number							
2	Logical Block Address							
3								
4	Transfer Length							
5	Control							

**Figure 6-2: READ(6) Command Structure**

Bit/Byte	7	6	5	4	3	2	1	0
0	Operation Code 28h							
1	Logical Unit Number			DPO	FUA	Reserved		RelAdr
2	Logical Block Address							
3								
4								
5	Reserved							
6								
7	Transfer Length							
8	Control							
9								

**Figure 6-3: READ(10) Command Structure**

These commands are implemented in the `do_read()` function of the SCG source.

### 6.2.3 WRITE

Similar to the READ command, there are two types of WRITE commands. The WRITE(6) command (Figure 6-4: WRITE(6) Command Structure) requests that the device server write the data transferred by the application client to the medium. The cache control bits are not provided for this command. Also similar to the READ commands is that block devices with cache memory may have values for the cache control bits which may affect the WRITE(6) command. Therefore for precise and explicit control, the WRITE(10) command (Figure 6-5: WRITE(10) Command Structure) should be used to send data.

Bit/Byte	7	6	5	4	3	2	1	0
0	Operation Code 0Ah							
1	Logical Unit Number							
2	Logical Block Address							
3								
4	Transfer Length							
5	Control							

**Figure 6-4: WRITE(6) Command Structure**

Bit/Byte	7	6	5	4	3	2	1	0
0	Operation Code 28h							
1	Logical Unit Number			DPO	FUA	Reserved		RelAdr
2	Logical Block Address							
3								
4								
5								
6								
7	Transfer Length							
8	Control							
9								

**Figure 6-5: WRITE(10) Command Structure**

These commands are implemented in the `do_write()` function of the SCG.

#### 6.2.4 READ CAPACITY

The READ CAPACITY command (Figure 6-6: READ CAPACITY Command Structure) provides a means for the application client to request information regarding the capacity of the logical unit. It is intended to assist storage management software in determining whether there is sufficient space on the current track, cylinder, etc., to contain a frequently accessed data structure, such as a file directory or file index, without suffering access delays.

The Partial Medium Indicator (PMI) located in the 0 bit of the eighth byte of the command structure informs the program of any delays. A PMI bit of one indicates that the returned logical block address and block length in bytes are those of the logical block address after which a substantial delay in data transfer will be encountered. This returned logical block address shall be greater than or equal to the logical block address specified by the RelAdr and logical block address fields in the command descriptor block. A PMI bit of zero indicates that the returned logical block address and the block length in bytes are those of the last logical block on the logical unit.

Bit/Byte	7	6	5	4	3	2	1	0
0	Operation Code 25h							
1	Logical Unit Number			Reserved				RelAdr
2	Logical Block Address							
3								
4								
5								
6								
7	Reserved							
8	Reserved							PMI
9	Control							

**Figure 6-6: READ CAPACITY Command Structure**

The READ CAPACITY command is implemented in the `do_readcapacity()` function of the SCG.

### 6.2.5 MODE SENSE

The SCSI MODE SENSE command requests information about operating parameters. It is a way for a device server to report parameters to the application client. There are six-byte and 10-byte versions of this command (Figure 6-7: MODE SENSE(6) Command Structure and Figure 6-8: MODE SENSE(10) Command Structure), each of which has a Disable Block Descriptor (DBD) bit in byte 1. A DBD of 0 indicates that zero or more block descriptors are returned in the `MODE_SENSE` data. A DBD of 1 indicates that the command will not return any block descriptors to the returning `MODE_SENSE` response.

Bit/Byte	7	6	5	4	3	2	1	0
0	Operation Code 1Ah							
1	Logical Unit Number			Resvd	DBD	Reserved		
2	PC		Page Code					
3	Reserved							
4	Allocation Length							
5	Control							

**Figure 6-7: MODE SENSE(6) Command Structure**

Bit/Byte	7	6	5	4	3	2	1	0
0	Operation Code 5Ah							
1	Reserved				DBD	Reserved		
2	PC		Page Code					
3	Reserved							
4	Reserved							
5	Reserved							
6	Reserved							
7	Allocation Length							
8								
9	Control							

**Figure 6-8: MODE SENSE(10) Command Structure**

The MODE SENSE command is implemented in the `do_modesense()` function of the SCG.

### 6.2.6 TEST UNIT READY

The TEST UNIT READY command (Figure 6-9: TEST UNIT READY Command Structure) provides a means for the application client to assess whether a device is ready without the need to allocate space for returned data. It is not a request for

a self-test. If the logical unit accepts a medium-access command without returning CHECK CONDITION status, this command shall return a GOOD status. If the logical unit cannot become operational or is in a state that an initiator action is required to make the unit ready, the target should return a CHECK CONDITION status with a sense key of NOT READY.

Bit/Byte	7	6	5	4	3	2	1	0
0	Operation Code 00h							
1	Logical Unit Number			Reserved				
2	Reserved							
3	Reserved							
4	Reserved							
5	Control							

**Figure 6-9: TEST UNIT READY Command Structure**

The TEST UNIT READY command is implemented in the `do_testunitready()` function of the SCG.

### 6.2.7 READ DEFECT DATA

The READ DEFECT DATA command (Figure 6-10: READ DEFECT DATA Command Structure) has the task of requesting the device server to transfer the medium defect data to the application client. In byte 2 of the command structure, there is a “primary defect list” (Plist) and a “grown defect list” (Glist). A Plist bit of 1 indicates the application is requesting that the target return the primary list of defects. A Plist bit of 0 is requesting that the target not return the primary list of defects. Similarly, a Glist bit of one request that the target return the grown defect list. A Glist bit of zero requests that the target not return the grown defect list. A Plist bit of one and a Glist bit of one indicates that it is requesting that the target return the primary and the grown defect lists. The order in which the lists are returned is vendor-specific. Whether the lists are merged or not is vendor-specific. A Plist bit of zero and a Glist bit of zero requests that the target return only the defect list header. The initiator to indicate the preferred format for the defect list

uses the defect list format field. This field is intended for those targets capable of returning more than one format.

Bit/Byte	7	6	5	4	3	2	1	0
0	Operation Code 37h							
1	Logical Unit Number			Reserved				
2	Reserved			Plist	Glist	Defect list for format		
3	Reserved							
4	Reserved							
5	Reserved							
6	Reserved							
7	Allocation Length							
8								
9	Control							

**Figure 6-10: READ DEFECT DATA Command Structure**

The READ DEFECT DATA command is implemented in the `do_readdefectdata()` function of the program.

### 6.2.8 REPORT LUNS

The REPORT LUNS command (Figure 6-11: REPORT LUNS Command Structure) requests that the peripheral device logical unit inventory be sent to the application client. A logical unit inventory is a list that has the logical unit numbers of all the logical units having a peripheral qualifier of 000b. Changes to the logical unit inventory may result from completion of initialization, removal of a logical unit, or creation of a logical unit. If the logical unit inventory changes for those or any other reasons, then the device server would send a unit attention command for all initiators. The allocation length in this command must be at least 16 bytes. If it is not, the device server will return a CHECK CONDITION status. The sense key will be set to ILLEGAL REQUEST and the additional sense data will be set to INVALID FIELD IN CDB. If the allocation length is not enough to contain the logical number values for all configured

logical units, the device server will report as many logical unit number values that would fit in the specified Allocation length. This is not considered an error.

Bit/Byte	7	6	5	4	3	2	1	0
0	Operation Code A0h							
1	Reserved							
2								
3								
4								
5								
6	Allocation Length							
7								
8								
9	Reserved							
10								
11	Control							

**Figure 6-11: REPORT LUNS Command Structure**

The REPORT LUNS command is implemented in the `do_reportluns()` function of the SCG.

### 6.2.9 VERIFY

The VERIFY command (Figure 6-12: VERIFY Command Structure) has the task of requesting that the target verify the data written on the medium. A byte check (BytCh) bit of zero causes a medium verification to be performed with no data comparison. A BytCh bit of one causes a byte-by-byte compare of data written on the medium and the data transferred from the initiator. If the compare is unsuccessful for any reason, the target shall return CHECK CONDITION status with the sense key set to MISCOMPARE. The verification length field specifies the number of contiguous logical

blocks of data that shall be verified. A transfer length of zero indicates that no logical blocks shall be verified. This condition shall not be considered as an error. Any other value indicates the number of logical blocks that shall be verified.

Bit/Byte	7	6	5	4	3	2	1	0
0	Operation Code 2Fh							
1	Logical Unit Number			DPO	Reserved		BytCh	RelAdr
2	Logical Block Address							
3								
4								
5								
6	Reserved							
7	Verification Length							
8								
9	Control							

**Figure 6-12: VERIFY Command Structure**

The VERIFY command is implemented in the `do_verify()` function of the SCG.

#### **6.2.10 SEND DIAGNOSTIC**

The SEND DIAGNOSTIC command (Figure 6-13: SEND DIAGNOSTIC Command Structure) requests the target to perform diagnostic operations on itself, on the logical unit, or on both. The only mandatory implementation of this command is the self-test feature with the parameter list length of zero. Except when the self-test bit is one, this command is usually followed by a RECEIVE DIAGNOSTIC RESULTS command.

Bit/Byte	7	6	5	4	3	2	1	0
0	Operation Code 1Dh							
1	Logical Unit Number			PF	Resvd	SelfTst	DevOfL	UnitL
2	Reserved							
3	Parameter List Length							
4								
5	Control							

**Figure 6-13: SEND DIAGNOSTIC Command Structure**

A page format (PF) bit of one specifies that the SEND DIAGNOSTIC parameters match up to the page structure as specified in this International Standard. The implementation of the PF bit is optional. A PF bit of zero indicates that the SEND DIAGNOSTIC parameters are as specified in SCSI-1 (i.e. all parameters are vendor-specific). A self-test (SelfTst) bit of one directs the target to complete its default self-test. If the self-test successfully passes, the command would be terminated with GOOD status; otherwise, the command would be terminated with a CHECK CONDITION status and the sense key would be set to HARDWARE ERROR. A self-test bit of zero requests that the target perform the diagnostic operation specified in the parameter list. The diagnostic operation might or might not require a target to return data that contains diagnostic results. If the return of data is not required, the return of GOOD status indicates successful completion of the diagnostic operation. If the return of data is required, the target would do either of the following:

- Perform the requested diagnostic operation, prepare the data to be returned and indicate completion by returning GOOD status. The initiator issues a RECEIVE DIAGNOSTIC RESULTS command to recover the data;
- Accept the parameter list, and if no errors are detected in the parameter list, and return GOOD status. The requested diagnostic operation and the preparation of the data to be returned are performed upon receipt of a RECEIVE DIAGNOSTIC RESULTS command.

To insure that the diagnostic command information is not destroyed by a command sent from another initiator, either the SEND DIAGNOSTIC command should be linked to the RECEIVE DIAGNOSTIC RESULTS command or the logical unit should be reserved. The SEND DIAGNOSTIC command is implemented in the `do_senddiagnostic()` function of the SC.

## 6.3 SCSI Command Generator Features

The final program can be compiled and executed from a Linux terminal window. It requires no kernel modifications or special configuration; all that is required is a recent version of the generic SCSI driver along with standard C libraries.

### 6.3.1 The Interface

At present, the program uses a simple command line interface. The prompt specifies the current working device, such as `/dev/sg0`. Typing “help” brings up a list of available options, with explanations on how to use them. Using the interface is as simple as entering these commands, and following subsequent prompts. After sending a command, the user is notified of the hexadecimal bytes defining the command being sent. Additional data may be displayed depending on the command issued, as well as any error resulting from a non-GOOD status byte. Figure 6-14: Example SCG usage depicts the interface when executing a simple Test Unit Ready command.

```
sg0> testunitready
  Sending command: 00 00 00 00 00 00
  Test unit ready: completed
sg0>
```

**Figure 6-14: Example SCG usage**

### 6.3.2 Pre-Defined Commands

Several simple commands require no input from the user. Executing a command like “test unit ready” is as simple as entering “testunitready” at the prompt. In these cases, the commands have no configurable parameters.

Other commands do require input from the user to determine certain values relevant to the command. In these cases, after entering the desired command name, the user is prompted individually for each configurable parameter. Most values are assumed to be decimal, but some, such as addresses, are expected to be in hexadecimal format. Many parameters have default value specified, speeding up the process in many cases. In future versions, the command line may accept arguments for these parameters to further speed things up for experienced users, rather than being prompted separately for each value.

### **6.3.3 Customized Commands**

In the SCG, 14 SCSI commands have been implemented. There are, however, many more commands in existence, all of which are supported. Typing “custom” allows a user to enter a customized command to send: the size of the command, the command itself (in hexadecimal), additional data to send, and the direction of transfer. The program blindly packages the `sg_io_header_t` structure and applies it to the target device. This is a useful tool for intentionally crafting invalid commands to observe the results.

### **6.3.4 Scripting Commands**

Finally, the ability to script commands was added to the program. The first step in this process is to begin recording a script to a file, accomplished by “rec <filename>”. Once in record mode, every command issued is noted in the script file instead of being sent to the target device. Loops are supported as well, allowing a command or complex series of commands to be repeated a specified number of times. To begin a loop, type “startloop #,” where # represents the number of iterations. To close the loop, type “endloop.” Starting a new loop before ending a previous loop results in nested loops. To end the recording session, type “stoprec.” Quitting the program while still in a recording session automatically ends the session before exiting.

While recording and in a loop, the program also allows certain bytes in the command to be changed incrementally after each iteration of the loop. For example, the Read, Write, and Verify commands prompt the user for an optional value to increment or decrement address. Entering “2” results in the address bytes(s) being increased by 2 each iteration; “-5” results in a decrease of 5. The custom command allows any of the bytes to

be treated this way by specifying the bytes which are to be changed. The user should remember that overflow is possible, but should not harm the operation of the program.

To execute the recorded commands, simply type “script <filename>.” The file’s contents are decoded, and each command is processed in rapid succession. No data is displayed on the screen in this event, because there could likely be hundreds or thousands of commands issued. The consequent traffic generated would likely be much more interesting and useful than just a single command when analyzed using an attached FCTracer unit.

The script file itself is in ASCII format. Leading white spaces (tabs, space, new lines) are ignored, and can be used freely for formatting. Each line represents a new action. The types of actions are the following:

- repeat:#** Begins a loop of # iterations.
- loop** Jumps to the top of the associated loop.
- cmd:<cmd>** Sets the command to <cmd>, in hex.

Within <cmd>, a + or - sign signifies a value that is decremented/incremented by a set amount each iteration. Following the + or - should be an integer specifying the amount to increment/decrement. The portion of the <cmd> to change should be contained in brackets []. The program prompts for this "iterative change" for certain commands.

Example: 0000+4[00]00 behaves the same as 00000000 => 00000400 => 00000800 => etc.

- data:#:<data>** Sets the data to whatever is specified by <data>. # must specify the exact length of <data> (to allow ‘\n’ and special characters to appear here without disrupting the file)
- xlen:#** Sets the data transfer length, if applicable

<b>xdir:[t/f]</b>	Sets the transfer direction. 't' for TO_DEV, 'f' for FROM_DEV, any other for NONE.
<b>send</b>	Sends the command as prepared.
<b>end</b>	Signifies that the script is done

Using this format, it is possible to create a scripted series of commands without using the utility's 'rec' function. Unrecognized actions are simply bypassed. The script format would be expandable for the future, as different parameters could be specified in the file and processed with little extra programming.

Example: the following script first sends an Inquiry command, and retrieves 96 bytes of inquiry data. Then it enters a 10-iteration loop. During each iteration, it writes two 512 byte blocks to the specified, and reads the data back. The first time, it writes to block 0004; the second to 0006, third to 0008, etc. After 10 repetitions the loop ends, and then the script ends as well.

```

cmd:080000000000
data:0:
xlen:96
xdir:f
send
repeat:10
  cmd:0a00+2[0004]0200
  data:5:hello
  xlen:1024
  xdir:t
  send
  cmd:0800+2[0004]0200
  THIS LINE IS TOTALLY IGNORED::1::!!!
  data:0:
  xlen:1024
  xdir:f
  send
loop
end

```

**Figure 6-15: Sample Script**

## 7 Interpreting and Decoding SCSI Data

Every SCSI command can be viewed, interpreted, and or decoded in three main ways under the FCTracer. It can be viewed at the Exchange level, Sequence level, or at the Frame level.

### 7.1 The Exchange Level

A Fibre Channel Exchange is defined as one or more non-concurrent Sequences between two ports. An Exchange may occur in either direction. All the commands in the application have a total of two exchanges, since the program both sends data and it receives data. The command structures are the data that is sent (Exchange1) and the output responses are the data that is received (Exchange2). Therefore, it is possible to identify if an error has occurred in transaction by counting the number of exchanges at the Exchange level. Figure 7-1: INQUIRY Exchange, Figure 7-2: TEST UNIT READY Exchange, and Figure 7-3: READ DEFECT DATA Exchange are examples of exchanges viewed from the tracer program.

1	Exchange	Originator	Responder	OX_ID	RX_ID	FCP SCSI	LUN	00000000 00000000	CDB	INQUIRY	Data-In	STATUS	GOOD
2	0	0x010100	0x0102E4	0x3408	0xFFFF						144 bytes		
		Metrics	#Seq.	TimeDelta									
			3	2.105 $\mu$ s									
3	Exchange	Originator	Responder	OX_ID	RX_ID	FCP SCSI	LUN	00000000 00000000	CDB	INQUIRY	Data-In	STATUS	GOOD
4	1	0x010100	0x0102E4	0x3408	0xFFFF						144 bytes		
		Metrics	#Seq.										
			3										

Figure 7-1: INQUIRY Exchange

1	Exchange	Originator	Responder	OX_ID	RX_ID	FCP SCSI	LUN	00000000 00000000	CDB	TEST UNIT READY	STATUS	GOOD	Metrics
2	0	0x010100	0x0102E4	0x3468	0xFFFF								
		#Seq.	TimeDelta										
		2	2.123 $\mu$ s										
3	Exchange	Originator	Responder	OX_ID	RX_ID	FCP SCSI	LUN	00000000 00000000	CDB	TEST UNIT READY	STATUS	GOOD	Metrics
4	1	0x010100	0x0102E4	0x3468	0xFFFF								
		#Seq.											
		2											

Figure 7-2: TEST UNIT READY Exchange

1	Exchange	Originator	Responder	OX_ID	RX_ID	FCP SCSI	LUN	CDB	Data-In
2	0	0x010100	0x0102E8	0x34B0	0xFFFF		00000000 00000000	READ DEFECT DATA(10)	13988 bytes
STATUS		GOOD	Metrics	#Seq	TimeDelta				
				3	2.105 µs				
3	Exchange	Originator	Responder	OX_ID	RX_ID	FCP SCSI	LUN	CDB	Data-In
4	1	0x010100	0x0102E8	0x34B0	0xFFFF		00000000 00000000	READ DEFECT DATA(10)	13988 bytes
STATUS		GOOD	Metrics	#Seq					
				3					

**Figure 7-3: READ DEFECT DATA Exchange**

The Fibre Channel Analyzer has four channels for which data can be exchanged. For all of the preceding trace examples, Exchange 0 occurred on the first and second channel and Exchange 1 had occurred on the third and fourth channel. Channel selection is shown on the left most side of the two Exchange boxes.

## 7.2 The Sequence Level

The red colored blocks in the Exchange trace is giving the address identifier of the sender and the receiver. The green box is identifying the type of command that is being sent or received. For all these commands, the tracer should display FCP SCSI within this block. The logical unit number is also given. The Command Block (CDB) found after the LUN specifically states the exact type of SCSI command that is being sent or received. The number of data in bytes the command takes is represented in the block following the CDB. Following this is the status of whether the command is successful or not, indicating if an error has occurred.

To view a trace in more detail, a user can click on the triangles pointing downwards in the two Exchange boxes. This allows the user to view the Sequences that makes up the Exchanges. A Fibre Channel Sequence is defined to be a series of one or more related frames transmitted in one direction from one port to another. It is unidirectional. Figure 7-4: INQUIRY Sequence shows the Sequences that make up the Exchanges in the SCSI Inquiry command:



**Figure 7-4: INQUIRY Sequence**

Hiding or collapsing specific data is a feature that the FCPTracer offers. Figure 7-5: INQUIRY Sequence Collapsed shows the Sequences by themselves with the Exchange data collapsed:

1	Sequence	S_ID	D_ID	OX_ID	RX_ID	SEQ_ID	EC	F/M/L	SI	F C P Request	FCP_CMND	SCSI LUN	SCSI CDB		
	0	0x010100	0x0102E4	0x3408	0xFFFF	0x7C	0	F	T	Init -> Trgt		0000 0000 0000 0000			
	INQUIRY	Metrics	# Frames	TimeDelta											
			1	2.105 $\mu$ s											
3	Sequence	S_ID	D_ID	OX_ID	RX_ID	SEQ_ID	EC	F/M/L	SI	F C P Request	FCP_CMND	SCSI LUN	SCSI CDB		
	1	0x010100	0x0102E4	0x3408	0xFFFF	0x7C	0	F	T	Init -> Trgt		0000 0000 0000 0000			
	INQUIRY	Metrics	# Frames	TimeDelta											
			1	371.000 $\mu$ s											
4	Sequence	S_ID	D_ID	OX_ID	RX_ID	SEQ_ID	EC	F/M/L	SI	F C P Response	FCP_DATA	144 bytes	Data		
	2	0x0102E4	0x010100	0x3408	0xFFFF	0x00	R	M	H	Trgt -> Init	Data-In		36 dwords		
	Metrics	# Frames	TimeDelta												
		1	865.000 ns												
2	Sequence	S_ID	D_ID	OX_ID	RX_ID	SEQ_ID	EC	F/M/L	SI	F C P Response	FCP_DATA	144 bytes	Data		
	3	0x0102E4	0x010100	0x3408	0xFFFF	0x00	R	M	H	Trgt -> Init	Data-In		36 dwords		
	Metrics	# Frames	TimeDelta												
		1	33.112 $\mu$ s												
4	Sequence	S_ID	D_ID	OX_ID	RX_ID	SEQ_ID	EC	F/M/L	SI	F C P Response	FCP_RSP	SCSI STATUS	Metrics	# Frames	
	4	0x0102E4	0x010100	0x3408	0xFFFF	0xFF	R	L	H	Trgt -> Init		GOOD		1	
	TimeDelta														
		622.500 ns													
2	Sequence	S_ID	D_ID	OX_ID	RX_ID	SEQ_ID	EC	F/M/L	SI	F C P Response	FCP_RSP	SCSI STATUS	Metrics	# Frames	
	5	0x0102E4	0x010100	0x3408	0xFFFF	0xFF	R	L	H	Trgt -> Init		GOOD		1	

Figure 7-5: INQUIRY Sequence Collapsed

In the preceding figure of the Inquiry command, it is apparent that the first 2 sequences of the trace are responsible for sending out the Inquiry command to the device. The FCP Request block confirms this by displaying an initiator to target field for the first 2 sequences and a target to initiator field in the last four sequences, indicating that it is a response from the device.

### 7.3 The Frame Level

To view a trace in even greater detail, it is possible to set viewing at the frame level. Similarly to collapsing data fields, the user is able to expand them by clicking on the triangles in the Sequences blocks to view the frames that make up the Sequence. All frames must be part of a Sequence. Frames within the same Sequence have the same SEQ\_ID field in the header. The SEQ\_CNT field identifies individual frames within a Sequence. For each frame transmitted in a Sequence, SEQ\_CNT is incremented by 1. This provides a means for the recipient to arrange the frames in the order in which they were transmitted and to verify that all expected frames have been received. Multiple Sequences to multiple ports may be active at a time. Figure 7-6 (continued): INQUIRY Frame presents a detailed view of the trace at the frame level:

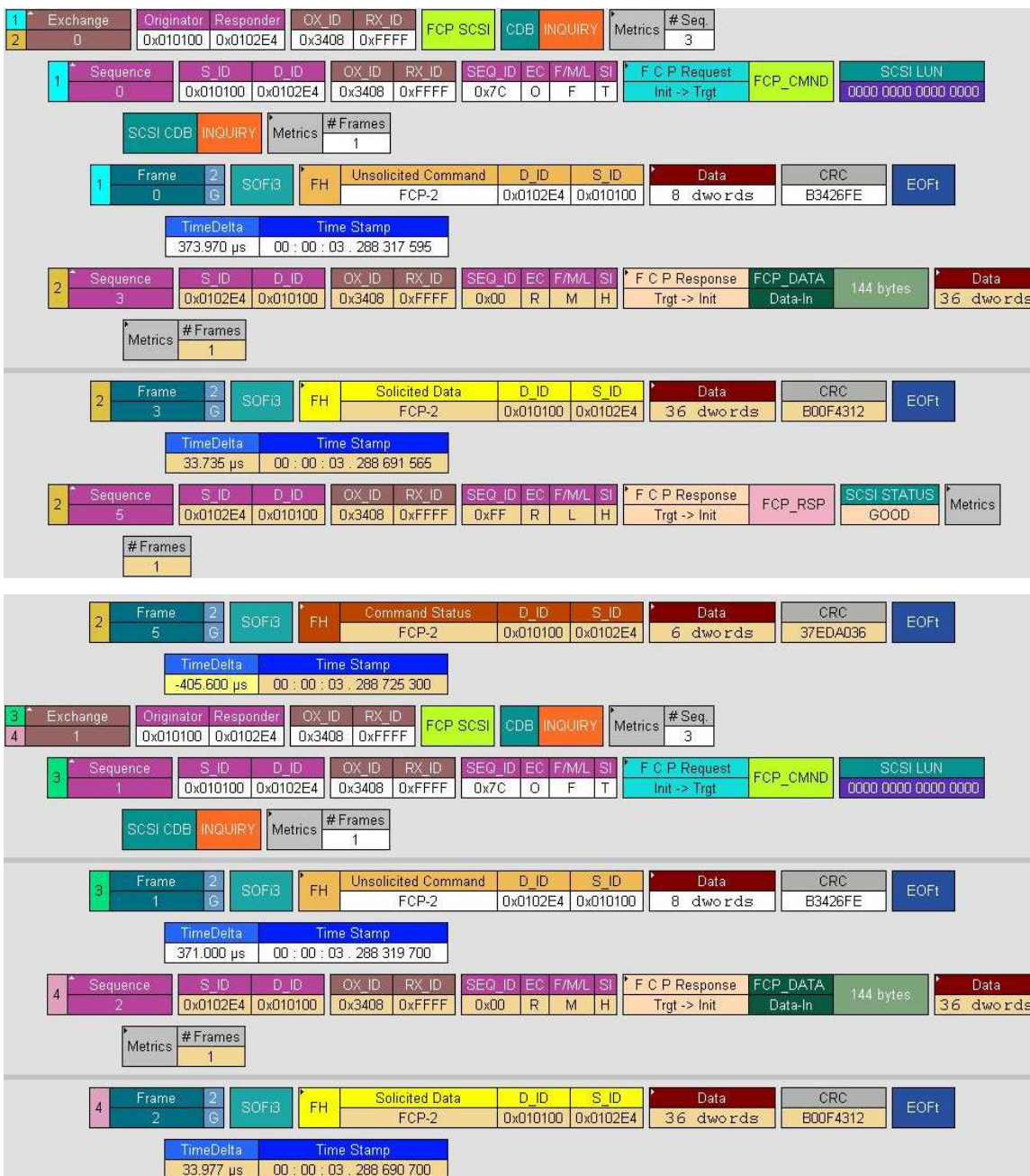
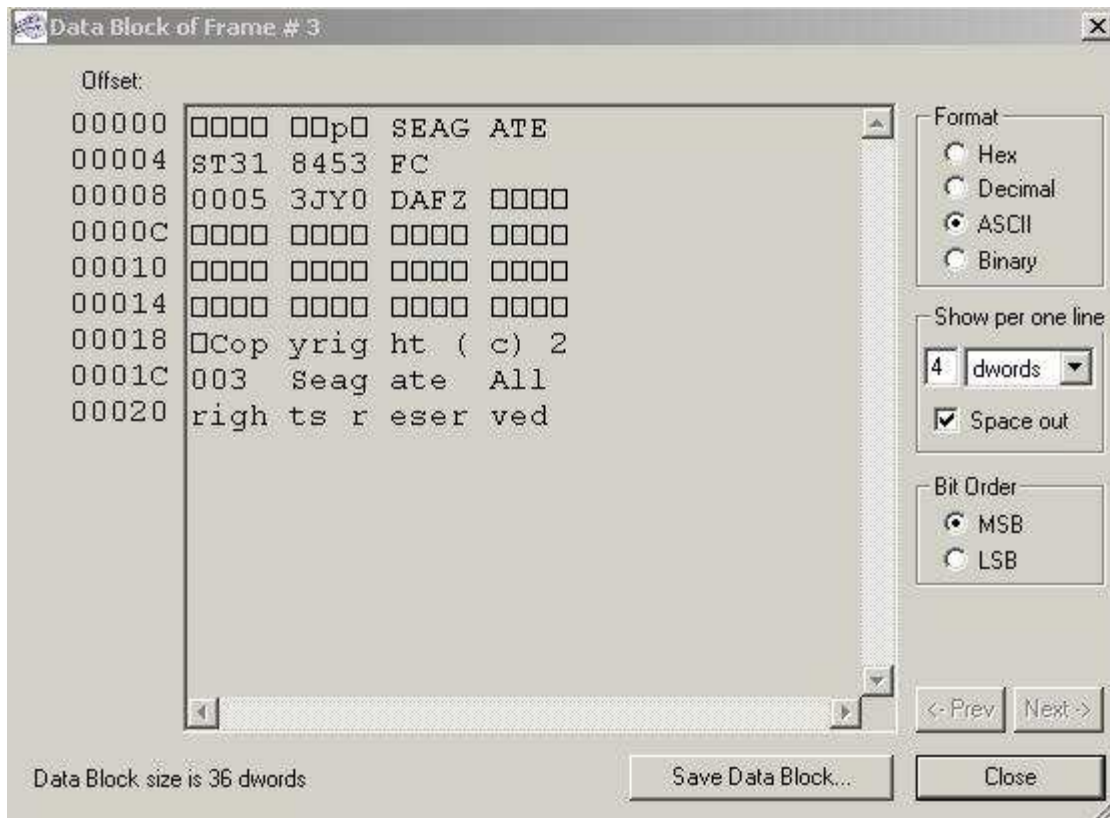


Figure 7-6: Inquiry Frame







**Figure 7-8: INQUIRY Data Block**

All of the data blocks can be double clicked on to show even greater detail.

## 8 Conclusions

In January 2004, our student group began the process of developing an application to assist General Dynamics with its Fibre Channel encryption project. We believe that we have succeeded in meeting expectations, both GD's and our own.

After completion, the SCSI Command Generator meets the informal requirements agreed upon in January. It supports each of the fourteen commands specifically chosen by GD developers, and this number is extended by allowing users to manually define other commands to send. The SCG includes scripting abilities resulting in longer streams of fully customized commands. It is also completely controllable from a simple, intuitive user interface. Perhaps most importantly, the source code is well documented and easy to modify or extend, ensuring its usefulness to GD developers.

Ultimately, the SCG should make testing and debugging Fibre Channel encryption methods a simpler task for General Dynamics. While there are many common applications they could use to create traffic over a SAN, the SCG gives the user much stricter control over the data being sent. As a result, small discrepancies in the traffic as recorded by the FCTracer can reveal the sources of potential problems quickly.

From the perspective of graduating WPI Computer Science majors, we have gained a great deal during the course of the project. For the first time in our college careers, we have created an application from the ground up that will be used in an industry setting. We also have achieved a higher level of understanding of technologies like SAN, Fibre Channel, and SCSI that we likely never would have been faced with otherwise.

## 9 Future Work

This project could have implications for work in the future. For now, the SCG will be used to generate traffic over an encrypted Fibre Channel connection, but the tool could also be applied to any type of SCSI connection as long as the *sg* driver is present. Using the SCG, a team could analyze traffic generated by various SCSI commands, device configurations, and bus settings to maximize the efficiency of a SAN. If the tool works well for testing General Dynamics' Fibre Channel devices, tools for similar purposes could be developed for connections other than SCSI.

Furthermore, the SCSI Command Generator itself could undergo improvements to make it more effective. More commands could be added to the current list, allowing greater versatility to the tool without requiring the repeated tedious creation of "custom" commands. An enhanced user interface could also improve the SCG. A GUI would simplify tasks like scripting and provide a visual representation of the progress of commands being executed.

## References

- [CAT04] Computer Access Technology Corporate Manual. 16 September 2004. CATC Corp. [http://www.catc.com/products/fibre\\_channel.html](http://www.catc.com/products/fibre_channel.html).
- [DEM00] Deming, David. An In-Depth Exploration of Small Computer System Interface. Saratoga, CA. ENDL Publications, 2000.
- [FAR01] Farley, Mark. Building storage networks, second edition. Berkeley, CA. Osborne/McGraw-Hill, 2001.
- [FIB00] Fibre Channel Association. Fibre Channel [electronic resource]: connection to the future. Eagle Rock, VA. LLH Technology, 2000.
- [FIB94] Fibre Channel Overview. 15 August 1994. Research Institute for Particle and Nuclear Physics. <http://hsi.web.cern.ch/hsi/fcs/>.
- [FIB95] Fibre Channel. 28 August 1995. Ohio State University. [http://www.cis.ohio-state.edu/~jain/cis788-95/fiber\\_channel/index.html](http://www.cis.ohio-state.edu/~jain/cis788-95/fiber_channel/index.html)
- [KUM99] Khattar, Ravi Kumar. Introduction to Storage Area Network, SAN. San Jose, CA. IBM Corporation, International Technical Support Organization, 1999.
- [LIN02] The Linux SCSI Generic (sg) HOWTO. 05 March 2002. TLDP Organization. <http://tldp.org/HOWTO/SCSI-Generic-HOWTO/>
- [LIN96] The Linux SCSI Programming HOWTO. 07 May 1996. Ibiblio Organization. [http://www.ibiblio.org/pub/Linux/docs/HOWTO/other-formats/html\\_single/SCSI-Programming-HOWTO.html#toc23](http://www.ibiblio.org/pub/Linux/docs/HOWTO/other-formats/html_single/SCSI-Programming-HOWTO.html#toc23)

- [REC03] Recover Data Fibre Channel Tutorial. 08 Nov. 2003. Recover Data Co.  
[http://www.recoverdata.com/fc\\_tutorial.htm#General%20overview](http://www.recoverdata.com/fc_tutorial.htm#General%20overview).
- [SEA03] Product Manual: Fibre Channel Interface. Scotts Valley, CA. Seagate Technology, 2003.
- [SCS00a] SCSI Direct Access Devices. 01 May 2000. University of Bologna.  
<http://www.danbbs.dk/~dino/SCSI/SCSI2-09.htm>.
- [SCS00b] SCSI-2 Specification. 15 January 2000. SCSI Mechanic Co.  
<http://www.scsimechanic.com/scsi/SCSI2.html>.
- [SCS03] SCSI Storage Interfaces. 18 December 2003. T10 Technical Committee.  
[http://www.t10.org/t10\\_main.htm](http://www.t10.org/t10_main.htm).
- [SCS04] The Linux SCSI Generic (sg) Driver. 09 January 2004. Torque Networks.  
<http://www.torque.net/sg/#SG%20device%20driver%20downloads>
- [SCS99] SCSI Command Summary. 22 September 1999. <http://www.bit-net.com/~rmiller/kits/scu/scu-scsi-summary.doc>.

## Appendix A

### The `sg_io_hdr` Structure

(Taken directly from [LIN02])

The main control structure for the version 3 SCSI generic driver has a struct tag name of "sg\_io\_hdr" and a typedef name of "sg\_io\_hdr\_t". The structure is shown in abridged form below. The "[i]" notation indicates an input value while "[o]" indicates a value that is output. The "[i->o]" indicates a value that is conveyed from input to output and apart from one special case, is not used by the driver. The "[i->o]" members are meant to aid an application matching the request sent to a write() to the corresponding response received by a read(). For pointers the "[\*i]" indicates a pointer that is used for reading from user memory into the driver, "[\*o]" is a pointer used for writing, and "[\*io]" indicates a pointer used for either reading or writing.

```
typedef struct sg_io_hdr
{
    int interface_id;           /* [i] 'S' (required) */
    int dxfer_direction;       /* [i] */
    unsigned char cmd_len;     /* [i] */
    unsigned char mx_sb_len;   /* [i] */
    unsigned short iovec_count; /* [i] */
    unsigned int dxfer_len;    /* [i] */
    void * dxferp;             /* [i], [*io] */
    unsigned char * cmdp;      /* [i], [*i] */
    unsigned char * sbp;       /* [i], [*o] */
    unsigned int timeout;      /* [i] unit: millisecs */
    unsigned int flags;        /* [i] */
    int pack_id;               /* [i->o] */
    void * usr_ptr;            /* [i->o] */
    unsigned char status;      /* [o] */
    unsigned char masked_status; /* [o] */
    unsigned char msg_status;  /* [o] */
    unsigned char sb_len_wr;   /* [o] */
    unsigned short host_status; /* [o] */
    unsigned short driver_status; /* [o] */
    int resid;                 /* [o] */
    unsigned int duration;     /* [o] */
    unsigned int info;         /* [o] */
} sg_io_hdr_t; /* 64 bytes long (on i386) */
```

## Appendix B

### **SCG Documentation**

```
*****
***USER'S GUIDE***** Authors: Alex Kinney and Minh Huynh
***SCSI COMMAND GENERATOR *****
***3/2/04*****
*****
```

#### --COMPILING

At the moment, the files making up this application are on the HBA1 Linux PC, in the directory /home/wpi/scsi/src/.

The files making up this application are the following:

```
go.c
commands.c
commands.h
lib.c
Makefile
```

To compile, go.c must be compiled along with lib.c and commands.c (gcc go.c commands.c lib.c -o Go). Typing 'make' will run the Makefile, which accomplishes this.

#### RUNNING THE PROGRAM

After starting the program, it will scan for generic SCSI device files in the /dev/ directory. With the 2 hard disks attached via fibre channel, it will find each of these devices as /dev/sg0 and /dev/sg1.

The user will then see a command line interface that accepts user input.

#### --SENDING A COMMAND

To view the accepted commands, type 'help' at the command line, followed by ENTER. To execute one of the commands, enter the command name. You then may be prompted for additional information about the command, if applicable.

In general, numbers should be entered in decimal format, unless otherwise specified. Hexadecimal numbers should not include uppercase letters (eg. f01a, not F01A).

If the command requests data from the target device, the data will be displayed. If an error occurs, the sense information will be displayed.

A special case is the 'custom' command, which allows a user-defined command to be sent. Here, the user must specify the command bytes, the direction of data transfer, and the data (if applicable).

#### --RECORDING A SCRIPT

To record a sequence of commands to be sent in rapid succession, enter "rec <file>", where <file> is the name of the script file you wish to record to. Once recording, sending commands as described above will not actually communicate with the device, but rather it will record each event in order to the script file. To stop recording, enter 'stoprec'.

To begin a loop, enter "startloop #" where # is the number of iterations. To end the loop, enter "endloop". Every command entered between will be considered part of the loop. Nesting is also supported, allowing one loop to occur within the scope of another loop. Certain fields in some commands (like read / write) may also be incremented / decremented per iteration.

To run the script, enter "script <file>". The commands will be sent in rapid succession, and output data will not be displayed on screen, but may be output to a log file if so chosen. The data output to the log file will consist of the command sent, any errors, and the data transferred (if applicable).

#### --SCRIPT FORMAT

The script file is in ASCII format. Leading white spaces (tabs, space, new lines) are ignored, and can be used freely for formatting. Each line represents a new action. The types of actions are:

repeat:#

Begins a loop of # iterations

loop

Jumps to the top of the associated loop

cmd:<cmd>

Sets the command to <cmd>, in hex

Within <cmd>, a + or - sign will signify a value that will be decremented/incremented by a set amount each iteration. Following the + or - should be an integer specifying the amount to increment/decrement. The portion of the <cmd> to change should be contained in brackets []. The program will prompt for this "iterative change" for certain commands.

Example: 0000+4[00]00 will behave the same as 00000000 => 00000400 => 00000800 => etc.

data:#:<data>

Sets the data to whatever is specified by <data>. # must specify the exact length of <data> (to allow '\n' and special characters to appear here without disrupting the file)

xlen:#

Sets the data transfer length, if applicable

xdir:[t/f]

Sets the transfer direction. 't' for TO\_DEV, 'f' for FROM\_DEV, any other for NONE.

send

Sends the command as prepared.

end

Signifies that the script is done

Using this format, it is possible to create a scripted series of commands without using the utility's 'rec' function. Unrecognized actions are simply bypassed. The script format would be expandable for the future, as different parameters could be specified in the file and processed with little extra programming.

Example: the following script first sends an Inquiry command, and retrieves 96 bytes of inquiry data. Then it enters a 10-iteration loop. Each iteration, it writes 2 512 byte blocks to the specified, and reads the data back. The first time, it writes to block 0004; the second to 0006, third to 0008, etc. After 10 repetitions the loop ends, and then the script ends as well.

```

cmd:080000000000
data:0:
xlen:96
xdir:f
send
repeat:10
  cmd:0a00+2[0004]0200
  data:5:hello
  xlen:1024
  xdir:t
  send
  cmd:0800+2[0004]0200
  THIS LINE IS TOTALLY IGNORED::1::!!!
  data:0:
  xlen:1024
  xdir:f
  send
loop
end

```

```

*****
***(SHORT) PROGRAMMING GUIDE***
*****
*****

```

#### --PROGRAM STRUCTURE

The main flow of the program happens in `go.c`, where user input is interpreted, and functions in `commands.c` are called accordingly. To add support for another command, the first step would be to insert a check in `go.c` to compare user input to the new desired string, and to call a new function to handle the new case.

#### --ADDING AND MODIFYING COMMANDS

The meat of the program is located in `commands.c`. This file includes a function for every command.

```
do_custom(int fd)
```

```

do_inquiry(int fd)
do_modesense(int fd)
do_read(int fd)
do_readcapacity(int fd)
do_readdefectdata(int fd)
do_receivediagnostic(int fd)
do_reportluns(int fd)
do_requestsense(int fd)
do_senddiagnostic(int fd)
do_testunitready(int fd)
do_verify(int fd)
do_write(int fd)

```

To add a new command, one would need to add a new function to handle it.

Within each command function, the application prompts the user for input regarding various configurable parameters for the command. Using this data, the function should set up the `sg_io_hdr_t` structure 'hdr', including the command (cmdp) and the data (dxferp). After setting this up, it should call `do_scsi()`, which will execute the command specified by hdr. If desired, the function could also process any received data and display it to the user.

When trying to add a command, the best method would probably be to copy an existing command (like `do_inquiry`) and modify it.

Existing command functions can also be modified to allow further customization of more SCSI parameters (such as timeout). This can be accomplished by inserting more prompts for user input, and setting appropriate header fields to non-default values.

--HDR, DATA, CMD, and SB

For ease of use, these are global buffers that store the SG header, the data buffer, and the command. CMD will store the 6, 10, 12, or 16 bytes specifying the command. DATA will store any data to be transferred, and will contain any data received after the command is executed. SB will contain sense information, which normally describes the nature of an error that has occurred. HDR is the structure that is written to the SG driver itself, and contains pointers to DATA (dxferp), CMD (cmdp), and SB (sbp). More information about using this structure can be found at [http://www.torque.net/sg/p/sg\\_v3\\_ho/](http://www.torque.net/sg/p/sg_v3_ho/).

#### --STRUCT ITERATIVE\_CHANGE

This struct defines the iterative change that is desired when recording a command. `start-byte` and `span` specify exactly what bytes to change (eg. 0 and 4 respectively would mean the first 4 bytes of the command). `Change` is the amount to change this number by each iteration. `do_scsi()` will check this structure and record a command in a script file accordingly.

#### --NUM\_LOOPS and SCRIPTING

`num_loops` describes the current depth of the looping during a script record session. If 0, there is no loop; if 5, we are in a loop, which is nested within 4 others. Higher values of `num_loops` result in more whitespace added at the beginning of each script line. If a scripting session is ended when `num_loops` is not 0, `end_loop()` will be called until `num_loops` is 0.

#### --LOGGING

logging is equal to 1 if when running a script, the user has chosen to output the results to a log file. When `logging = 1`, `do_scsi()` will output the command bytes, any error information, and the data (if any) to the file referred to by `logfd`.

#### --OTHER FUNCTIONS

There are also other functions that aide in functionality. The functions in `lib.c` generally simplify tasks like getting user input and converting between data types (eg. Integers to 4 unsigned bytes, hexadecimal string to integer, etc). More detailed descriptions of all functions can be found commented in the code.

Additional functions in `commands.c` help to perform user-entered tasks that are not actually commands. These include things like starting loops, recording, logging results to a file, etc.

#### --CONTACT INFORMATION

For any questions that might arise in the future, the creators of this tool can be reached at the following email addresses:

Alex Kinney ([almiki@wpi.edu](mailto:almiki@wpi.edu))  
Minh Huynh ([hminh18@wpi.edu](mailto:hminh18@wpi.edu))