University of Twente

Faculty EE-Math-CS
Department of Electrical Engineering

Real Time Control on Firewire

Zhang Yuchen

Individual Project

Supervisors	prof.dr.ir. J. van Amerongen
	dr.ir. J.F. Broenink
	dipl. ing. B. Orlic

June 2004

Report 012CE2004
Control Engineering
Department of Electrical Engineering
University of Twente
P.O.Box 217
7500 AE Enschede
The Netherlands

Summary

The goal of this project is to get insight into the use of Firewire as a field bus for real-time control. A characterization of Firewire's asynchronous transmission has been made by testing the point-to-point roundtrip in a 3-node Firewire network.

The results show Firewire's asynchronous transmission between 2 PC/104 stacks, using FCP (Functional Control Protocol) as the way of implementing, can transfer data in an average latency between 100μs and 140μs, depending on the data payload. The maximum variation in this latency is 20μs.

During this project, it is also found that while the payload is increased, the roundtrip time does not show a significant increase: it rises only around 40μs from 1byte payload to 449 bytes payload. So the bus is used most efficiently when the packet is fully loaded.

To get a complete insight of Firewire, it is recommended to implement Firewire's isochronous transmission and examine its applicability for real-time control also. And to use Firewire as a field bus in a real distributed control system with a plant is also advised to get a better understanding of Firewire.

Table of Contents

Chapter 1 Introduction

1.1 Context

This project is placed within the embedded control system research by the Control Engineering group in University of Twente, where *real-time control on field-bus interconnected heterogeneous systems* is one of the three sub Phd projects.

This sub Phd project deals with hard real-time control using different kinds of field bus in a network environment of several cooperating processors. The CAN bus, USB bus and RT-Ethernet have been investigated with respect to their real-time performance. And the CSP based parallel computing has been implemented in a CAN interconnected multiprocessor system, using CT-library (Communicating Threads Library), a basic framework for implementing concurrent embedded–system software on heterogeneous distributed hardware.

This assignment is to study and research Firewire, a high-speed serial bus, for its applicability in real-time control. More detailed information about this assignment is given in next section.

1.2 Assignment

The focus of this assignment is on real-time characterization on Firewire. Based on this focus, 2 stacks of technical questions are addressed. The first one is what is Firewire and how is the communication protocol realized? The answer should be addressed during literature study.

The second stack of questions is how to examine Firewire's real-time characteristic in a practical experiment. For this, a hardware setup should be built, using a real distributed system structure. Based on the hardware setup, a test bench should also be made.

Based on the testing results and analyzing, the conclusions and recommendations should be made for Firewire's applicability in real-time distributed control system and how to apply it in industry.

1.3 Structure of this report

Chapter2 gives the background knowledge of Firewire. Chapter3 gives the design of the hardware setup. The design of test bench based on the hardware setup is following in chapter4, the. The results and discussion is given in chapter5. Chapter6 is for conclusions and recommendations.

Chapter 2 Introduction to Firewire

This chapter gives an overview of Firewire. Most of the knowledge in 2.1 and 2.2 is adopted from chapter2-3, chapter13-21 of *Firewire System Architecture* [Anderson D., 1999]. Most of the knowledge in 2.3 is adopted from *1394 Open Host Controller Interface Specification, Release 1.1* [1394OHCI Specification, 2000].

2.1 History, basic specifications and applications

2.1.1 History

Development of Firewire began in the mid 1980s by Apple Computer. In fact, the term Firewire is a registered trademark of Apple Computer Corporation. As other manufacturers gained interest in Firewire, a working committee was formed to create a formal standard on this architecture. The resulting specification was submitted to IEEE and IEEE1394-1995 was adopted.

Early implementation based on different interpretations of the 1995 release resulted in some interoperability problems between different vendor parts. A supplement to the 1394-1995 specification is referred to as the 1394a, and is designed to eliminate these problems, also to add new functionality.

Yet another version of 1394 being developed is called the IEEE 1394b specification.

Note that in this report, ”Firewire”, “1394a” and “1394” are referred as the same.

2.1.2 Basic Specifications

Firewire is a kind of serial bus. It defines two bus categories: backplane and cable. The backplane bus provides an alternative serial communication path for parallel bus devices plugged into the backplane. The bus discussed in this chapter is the cable bus: a "non-cyclic network with finite branches" consisting of bus bridges and nodes (cable devices). “Non-cyclic” means devices can not be plugged together to create loops.

The node address in Firewire has 16 bits length, which provides up to 64K nodes in one system. In the 16-bit address, a 6-bit Node_IDs allow up to 63 nodes to be connected to a single bus bridge, and another 10-bit Bus_IDs allows up to 1,023 bridges in a system.

Each node normally has 3 connectors. Up to 16 nodes can be daisy-chained up to 4.5 m through the connectors for a total standard cable length of 72 m (longer using higher-quality "fatter" cables). Extra devices can be connected in a leaf-node configuration, as shown in Figure1. Physical addresses are assigned on bridge power up (bus reset), and whenever a node is added to or removed from the system. No device ID switches are required, and the nodes are hot pluggable, meaning that FireWire is a true plug-and-play bus.

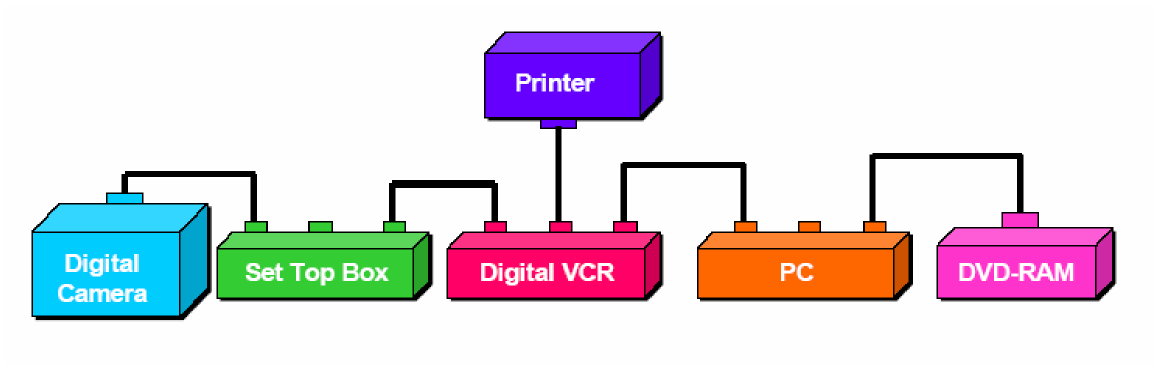

Figure 1 an example of Firewire network

The FireWire cable standard defines three signaling rates: 98.304, 196.608 and 393.216 Mbits/s. These are normally rounded up to 100, 200, and 400 Mbits/s, and often referred to as S100, S200 and S400. The signaling rate for the entire bus is usually governed by the slowest active node.

The Firewire protocol covers layers 1, 2 and 3 (physical, link and transaction) of the ISO's seven-layer OSI model. The standard 6-conductor cable has two separately shielded twisted-pair transmission lines for signaling (crossed for transmit-receive), two power conductors (8 to 40 V, 1.5 A max.), and an overall shield. Transformer or low-cost capacitive coupling provides galvanic isolation.

FireWire provides a flexible bus management system that connects between a wide range of devices, which does not need to include a PC or other bus controller. FireWire's isochronous data transport provides the guaranteed bandwidth and latency required for high-speed data transfer over multiple channels.

A simple comparison between Firewire and other field buses is listed in Table1:

Table 1 Comparison between Firewire and other Fieldbuses

Name	**Multimaster**	**Deterministic**	**Max Speed(Mbits/s)**
CAN	Yes	Yes	1
USB2.0	No	Yes	480
Ethernet	Yes	No	100
Profibus	Yes	No	12
Firewire 1394a	Yes	Yes	400
Firewire 1394b	Yes	Yes	3200

2.1.3 Applications

The scalable performance and support for both asynchronous and isochronous transfers makes Firewire an alternative for connecting a wide variety of peripherals including:

- Mass storage
- Video teleconferencing
- Video production
- Small networks

- High speed printers
- Entertainment equipment
- Set top box

Besides the application in Consumer Electronics, it also has been started to apply Firewire in industry. A number of European companies and organisations have come together in 1394automation [1394automation 2003], a group which believes in the power of FireWire and is defining a solution that will enable different controllers to communicate on the same FireWire bus. The Dutch company, Nyquist [Nyquist 2003] was the first European motion control company to introduce FireWire.

2.2 Address Space and Communication Layers

2.2.1 Address Space

A Firewire bus appears as a large memory mapped space with each node occupying a certain address range. The memory space is based on the IEEE 1212 Control and Status Register (CSR) Architecture with some extensions specific to the 1394 standard. Each node supports up to 48 bits of address space (256 Terabytes). In addition, each bus can support up to 64 nodes and totally, the system supports up to 1024 busses. This gives a grand total of 64 address bits, or support for a whopping total of 16 ExaBytes of memory space.

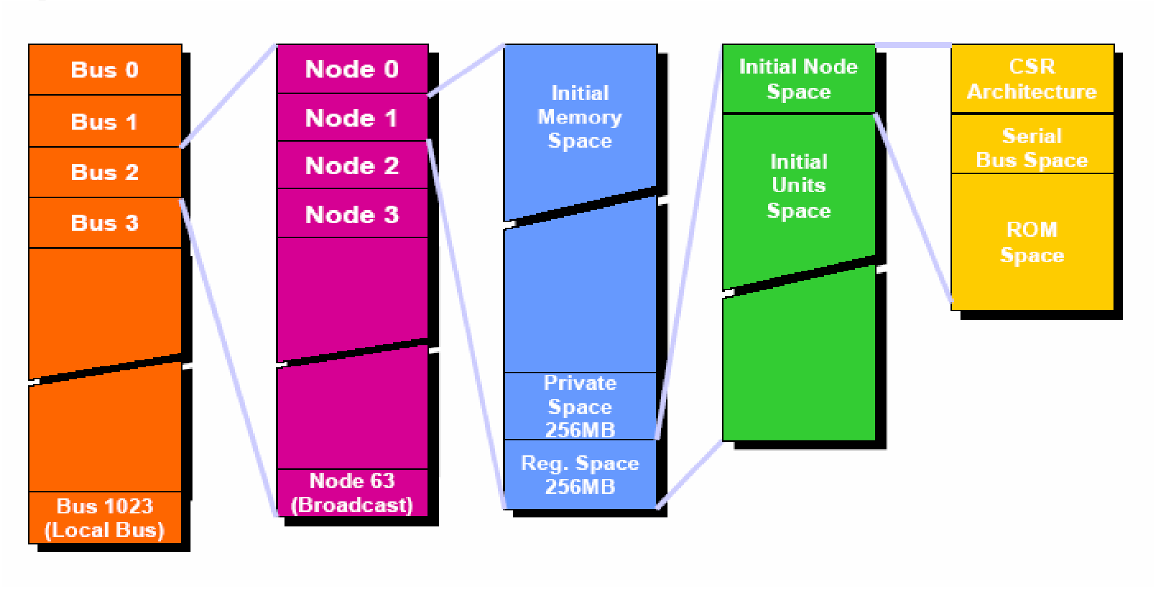

Figure 2. Address Space of Firewire

Figure2 represents the total 64 bits address space definition for the 1394 implementation. It can be seen that each node has 256 terabytes of address space. This space is divided into blocks defined for specific purposes, with address range labeled as:

- Initial memory space
- Private space
- Initial register space

- CSR architecture register space
- Serial bus space
- ROM (first 1 KB)
- Initial units space

2.2.2 Communication Layers

The 1394 specification defines four protocol layers, as represented in Figure3, although not all of them are used during all transfers.

1) Physical Layer

The physical layer of the Firewire protocol includes the electrical signaling, the mechanical connectors and cabling, the arbitration mechanisms and the serial coding and decoding of the data being transferred or received. The cable media is defined as a three pair shielded cable. Two of the pairs are used to transfer data, while the third pair provides power on the bus. The connectors are small six pin devices, although the 1394a also defines a 4 pin connector for self powered leaf nodes. The power signals are not provided on the 4 pin connector.

The two twisted pair used for signaling, called out as TPA and TPB, are both bidirectional and are tri-state capable. TPA is used to transmit the strobe signal and receive data, while TPB is used to receive the strobe signal and transmit data. The signaling mechanism uses data strobe encoding, a rather clever technique that allows easy extraction of a clock signal with much better jitter tolerance than a standard clock/data mechanism. With Data Strobe encoding, either the data or the strobe signal change in a bit cell, but not both of them. Data Strobe encoding is shown in Figure 4.

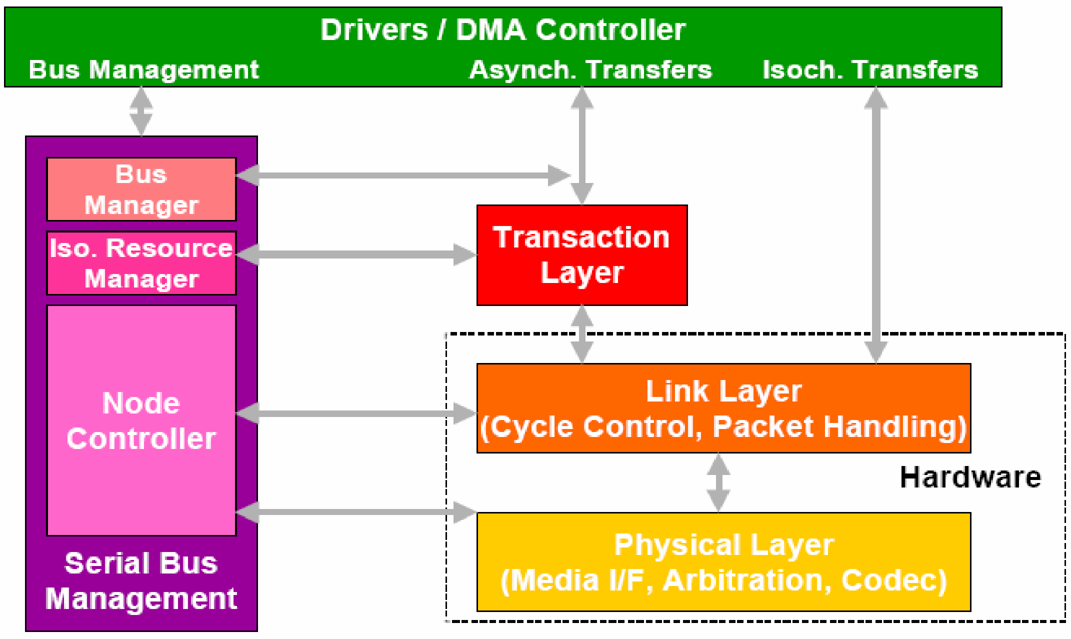

Figure 3 Firewire Protocol Layers

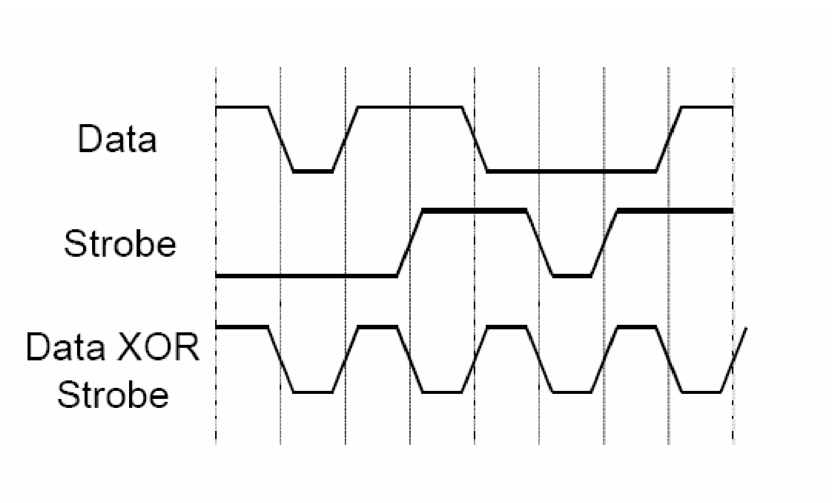

Figure 4 Data Strobe Encoding

Configuration

The physical layer also plays a major role in the bus configuration and normal arbitration phases of the protocol. Configuration consists of taking a relatively flat physical topology and turning it into logical tree structure with a root node at its focal point. A bus is reset and reconfigured whenever a device is added or removed – a reset can also be initiated via software. Configuration consists of bus reset and initialization, tree identification and self identification.

Reset

Reset is signaled by a node driving both TPA and TPB to logic 1. Because of the "dominant 1s"electrical definition of the drivers, a logic "1" will always be detected by a port, even if its bidirectional driver is in the transmit state. When a node detects a reset condition on its drivers, it will propagate this signal to all of the other ports that this node supports. The node then enters the idle state for a given period of time to allow the reset indication to propagate to all other nodes on the bus. Reset clears any topology information within the node.

Tree Identification

The tree identification process is how the bus topology is defined. Let's take the example of our example network shown in Figure 1. After reset, but before tree identification, the bus has a flat logical topology that maps directly to the physical topology. After tree identification is complete, a single node has gained the status of Root Node. The tree identification proceeds as follows:

After reset, all leaf nodes (nodes with only one other device connected to them) present a Parent_Notify signaling state on its Data and Strobe pairs. Note that this is a signaling state, not a transmitted packet – the whole tree identification process occurs in a matter of microseconds. In the example, the Digital Camera will signal the Set-Top Box, the Printer will signal the Digital VCR and the DVD-RAM will signal the PC. When a branch node receives the a Parent_Notify signal on one of its ports, it marks that port as containing a child, and outputs a Child_Notify signaling state on that port's Data and Strobe pairs. Upon detecting this state, the leaf node marks its port as a parent port and removes the signaling, thereby confirming that the leaf node has accepted the child designation. At this point our bus appears as shown in Figure 5. The ports marked with a P indicates that a device that is closer to the Root Node is attached to that port, while a port marked with a C indicates that a node further away from the Root Node is attached. The port numbers are arbitrarily assigned during design of the device and play an important part in the self identification process.

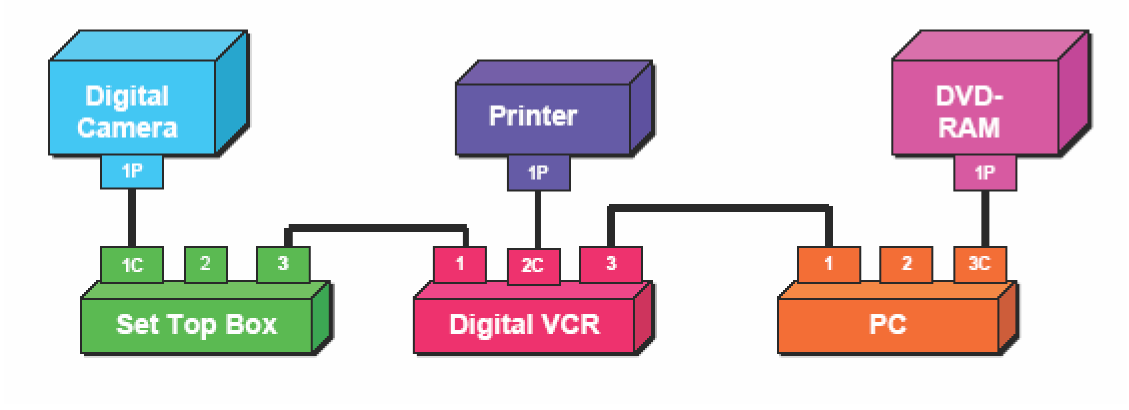

Figure 5 Bus after Leaf Node Identification

After the leaf nodes have identified themselves, the Digital VCR still has 2 ports that have not received a Parent_Notify, while the Set Top Box and the PC branch node both have only one port with an attached device that has not received a Parent_Notify. This being the case, both the Set Top Box and the PC start to signal a Parent_Notify on the one port that has not yet received one. In this case the VCR receives the Parent notify on both of its remaining ports, which it acknowledges with a Child_Notify condition. Since the VCR has marked all of its ports as children, the VCR becomes the Root Node. The final configuration is shown in Figure 6.

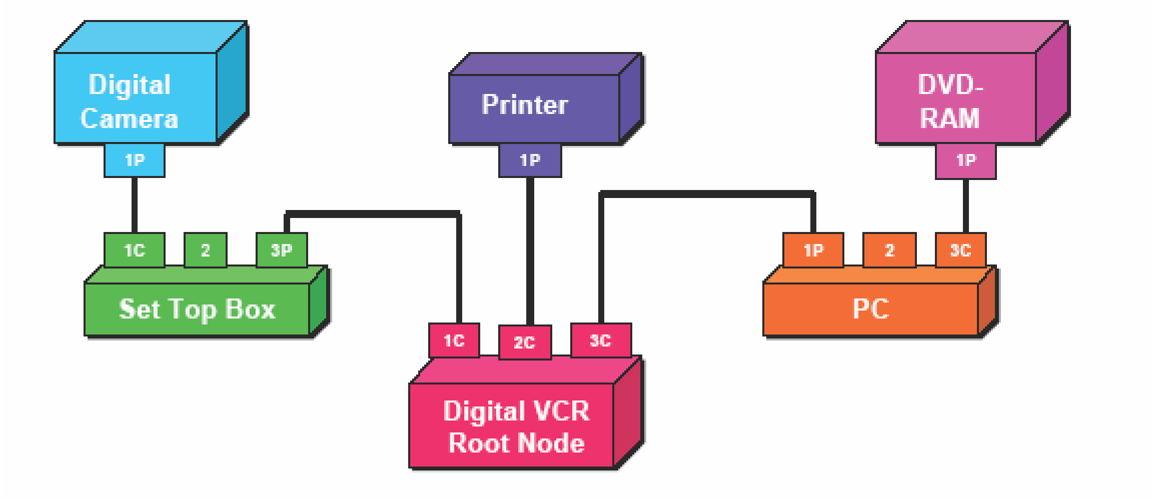

Figure 6 Bus after Tree Identification is completed

Note that it is possible for two nodes to be in contention for Root Node status at the end of the process. In this case, a random back-off timer is used to eventually settle on a Root Node. It is also possible for a node to force itself to become Root Node by delaying its participation in the Tree Identification process for a while.

Self Identification

Once the Tree topology is defined, the Self Identification phase begins. Self Identification consists of assigning Physical IDs to each node on the bus, having neighboring nodes exchange transmission speed capabilities and making all of the nodes on the bus aware of the topology that exists. The self identification phase begins with the Root Node sending an Arbitration Grant signal to its lowest numbered port. In the example the Digital VCR is the Root Node and it signals the Set Top Box. Since the Set Top Box is a branch node, it will propagate the Arbitration Grant signal to its lowest numbered port with a Child node attached. In this case this is the Digital Camera. Since the Digital Camera is a leaf node, it cannot propagate the Arbitration Grant signal downstream any further, so it assigns itself Physical ID 0 and transmits a Self-ID packet upstream. The branch node (Set Top Box) repeats the Self ID packet to all of its ports with attached devices. Eventually the Self ID packet makes it way back up to the Root Node, which proceeds to transmit the Self ID packet down to all devices on its higher numbered ports. In this manner, all attached devices receive the Self ID packet that was transmitted by the Digital Camera. Upon receiving this packet all of the other devices increment their Self ID counter. The Digital Camera also signals a Self ID Done indication upstream to the Set Top Box. This indicates to the Set Top Box that all nodes attached downstream on this port have gone through the Self ID process. Note that the Set Top Box does NOT propagate this signal upstream toward the Root Node because it has not completed the Self ID process.

The Root Node will then continue to signal an Arbitration Grant signal to its lowest numbered port which in this case is still the Set Top Box. Since the Set Top Box has no other attached devices, it assigns itself Physical ID 1 and transmits a Self ID packet back upstream. This process continues until all ports on the Root Node have indicated a Self ID Done condition. The Root Node the assigns itself the next Physical ID – the Root Node will always be the highest numbered device on the bus. If we follow our example through we come up with the following Physical Ids – Digital Camera = 0, Set Top Box = 1, Printer = 2, DVD-RAM = 3, PC = 4 and the Digital VCR, which is the Root Node is assigned Physical ID 5.

Note that during the Self ID process, parent and children nodes are also exchanging their maximum speed capabilities. Nodes can only transmit as fast as the slowest device in between the transmitting node and the receiving node. For example, if the Digital Camera and the Digital VCR are both capable of transmitting at 400 Mbps, but the Set Top Box is only capable of transmitting at 100Mbps, there is no way for the high speed devices to use the maximum rate to communicate amongst themselves. The only way around this problem is for the end user to reconfigure the cabling so the low speed Set Top Box is not physically between the two high speed devices.

Also during the Self ID process, all nodes wishing to become the Isochronous Resource manager will indicate this fact in their self ID packet. The highest numbered node that wishes to become resource manager will achieve the honor.

Normal Arbitration

Once the configuration process is complete, normal bus operations can begin. In order to fully understand arbitration, knowledge of the cycle structure of 1394 is necessary.

A 1394 cycle, as represented in Figure7, is a time slice with a nominal 125µs period. The 8 KHz cycle clock is kept by the cycle master, which is also the root node. To begin a cycle, the cycle master broadcasts a cycle start packet, which all other devices on the bus use to synchronize their time bases.

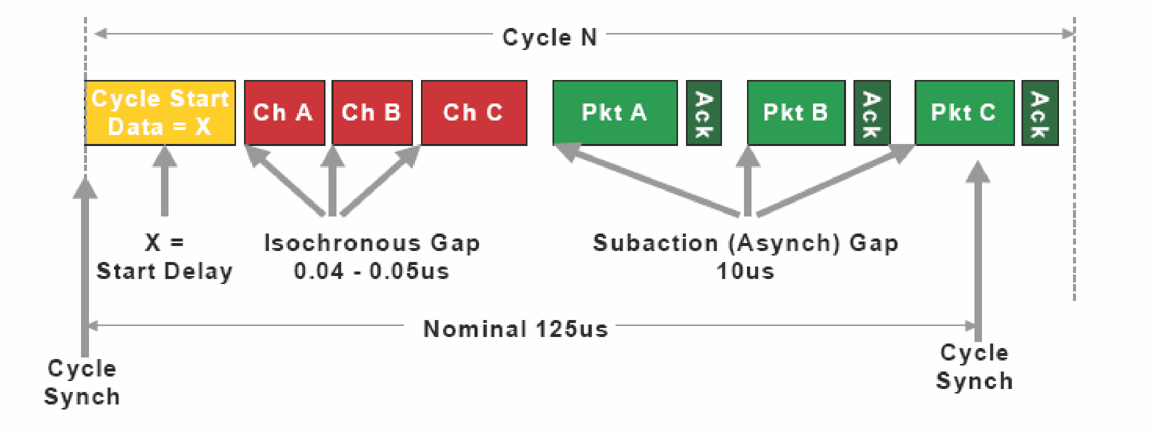

Figure 7 Typical Firewire Cycle

Immediately following the Cycle Start packet, devices that wish to broadcast their isochronous data may arbitrate for the bus. Arbitration consists of signaling their parent node that they wish to gain access to the bus. The parent nodes in turn signal their parents and so on until the request reaches the Root Node. In Figure1, suppose the Digital Camera and the PC wish to stream data over the bus. They both signal their parents that they wish to gain access to the bus. Since the PC's parent is the Root Node, its request is received first and it is granted the bus. From this scenario, it is evident that the closest device to the Root Node wins the arbitration. Since one Isochronous channel can only be used once per cycle, when the next Isochronous Gap occurs, the PC will no longer participate in the arbitration, unless PC also reserved the next channel. This allows the Digital Camera to win the arbitration this time.

When the last isochronous channel has transmitted its data, the bus goes idle waiting for another isochronous channel to begin arbitration. Since there are no more isochronous devices left waiting to transmit, the idle time extends longer than the isochronous gap until it reaches the duration defined as the sub action (or asynchronous) gap. It is at this time that asynchronous devices may begin to arbitrate for the bus. Arbitration proceeds in the same manner, with the closest device to the Root Node winning arbitration.

This brings up an interesting scenario – since asynchronous devices can send more than one packet per cycle, it might be possible for a device closest to the Root Node (or the Root Node itself) to hog the bus by always winning the arbitration. This scenario is dealt with using what is called the fairness interval and the Arbitration Rest gap. The concept is simple: once a node wins the asynchronous arbitration and delivers its packet, it clears its arbitration enable bit. When this bit is cleared, the physical layer no longer participates in the arbitration process, giving devices further away from the Root Node a fair shot at gaining access to the bus. When all devices wishing to gain access to the bus have had their fair shot, they all wind up having their arbitration enable bits cleared, meaning no one is trying to gain access to the bus. This causes the idle time on the bus to go longer than the 10μs sub action gap until it finally reaches 20μs, which is called the arbitration reset gap. Once the idle time reaches this point all devices may reset their arbitration enable bits and arbitration can begin all over again.

2) Link Layer

The Link Layer is the interface between the transaction layer and the physical layer. The Link Layer is responsible for checking received CRC, calculating and appending the CRC to transmitted packets. In addition, since isochronous transfers do not use the transaction layer, the Link Layer is directly responsible for sending and receiving isochronous data. The Link Layer also examines the packet header information and determines the type of transaction that is in progress. This information is then passed up to the transaction layer.

The Link Layer to Physical Layer interface consists of a minimum of 17 signals that must be either magnetically or capcitively isolated from the Physical Layer. These signals are defined in Table2.

Table 2 Link Layer to Physical Layer Interface

Signal Name	**Source**	**Description**
LReq	Link Layer	Link Request – used to initiate a request to send a packet as well as a request to read directly from a PHY register.
SClk	Physical Layer	49.152MHz clock used to synchronize data readout. (The frequency may change depending on data rates with 1394b)
Data[0:7]	Either	Data – higher transfer speeds use an increasing number of bits: 100Mbps – D[0:1] 200Mbps – D[0:3] 400Mbps – D[0:7] Note that the width of this data bus may expand to 16 bits with 1394b.
Ct[0:1]	Either	Control Interface – Defines what state the interface is in.
LPS	Link Layer	Link Power Status – Indicates that the Link Layer Controller is powered.
Link On	Physical Layer	Indicates that the Physical Layer has been powered on.
Direct	Neither	Indicates that no isolation barrier exists.
Backplane	Physical Layer	High if physical layer is a backplane implementation
Clk25	Neither	Indicates that SClk is only 24.576 MHz – valid in a backplane implementation only.

A typical Link Layer implementation has the Physical Layer interface, a CRC checking and generation mechanism, transmit and receive FIFOs, interrupt registers, a host interface and at least one DMA (Direct Memory Access) channel.

3) Transaction Layer

The transaction layer is used for asynchronous transactions. The 1394 protocol uses a request – response mechanism, with confirmations typically generated within each phase. There are several types of transactions allowed. They are listed as follows:

- Simple Quadlet (4 byte) Read
- Simple Quadlet Write
- Variable Length Read
- Variable Length Write
- Lock Transactions

Lock transactions allow for atomic swap and compare and swap operations to be performed. Asynchronous packets have a standard header format along with an optional data block. The packets are assembled and dis-assembled by the Link Layer controller. Figure 8 shows the format of a typical asynchronous packet, and the related explanation is given in Table3.

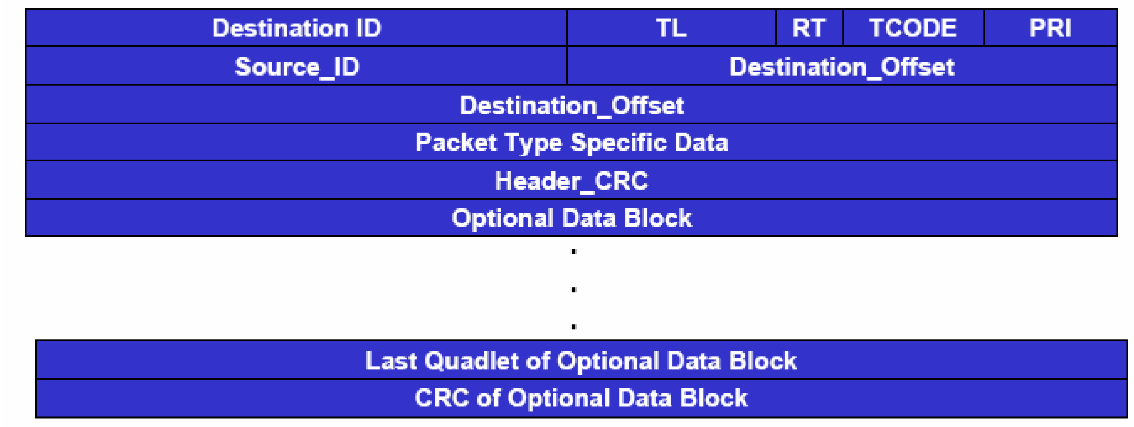

Figure 8 Asynchronous Packet Format

Table 3 Asynchronous Packet Format

Name	**Description**
Destination_ID	The concatenation of the Bus and Node address of the intended node. All ones indicate a broadcast transmission.
TL	Transaction Label specified by the requesting node. This value is also used in the response packet.
RT	Retry Code that defines whether this is a retry and what retry mechanism is being used.
TCODE	Transaction Code defines the type of transaction (Read

	request, read response, Acknowledge, etc..)
PRI	Priority – used only in backplane environments
Source_ID	Specifies Bus and Node that generated this packet.
Destination_offset	The address location within the destination node that is being accessed.
Packet Type Specific Data	Can indicate data length for block reads and writes, or contain actual data for a quadlet write request or quadlet read response.
Header_CRC	CRC Value for the Data
Optional Data	Quadlet aligned data specific to the type of the packet.
Optional Data CRC	CRC for the Optional Data

4) Bus Management

Bus management on a 1394 bus involves several different responsibilities that may be distributed among more than one node. Nodes on the bus must assume the role of Cycle Master, Isochronous Resource Manager and Bus Manager.

Cycle Master

The cycle master initiates the 125µs cycles. The Root Node must be the cycle master, if a node that is not cycle master capable becomes Root Node, the bus is reset and a node that is cycle master capable is forced to be the root. The cycle master broadcasts a cycle start packet every 125µs. Note that it is possible for a cycle start to be delayed while an asynchronous packet is being transmitted or acknowledged. The cycle master deals with this by including the amount of time that the cycle was delayed in the cycle start packet.

Isochronous Resource Manager

The isochronous resource manager must be isochronous transaction capable. In addition the isochronous resource manager must also implement several extra registers. These registers include the Bus Manager ID register, the bus bandwidth allocation register and the channel allocation register. Isochronous channel allocation is performed by a node that wishes to transmit isochronous packets. These nodes must allocate a channel from the channel allocation register by reading the bits in the 64 bit register. Each channel has one bit associated with it – channels are available if the bit is set to logic “1”. The requesting node sets the first channel bit available to logic “0” and uses this bit number as the channel ID.

In addition, the requesting node must examine the Bandwidth Available Register to determine how much bandwidth it can consume. The total amount of bandwidth available is 6144 allocation units. One allocation unit is 20.354ns. There are a total of 4915 allocation units available for isochronous transfers, i.e. 100µs. The rest 1229 units are for asynchronous transmission. Nodes wishing to use isochronous bandwidth must subtract the amount of bandwidth needed from the Bandwidth Available Register.

Bus Manager

A bus manager has several functions including publishing a topology map and a speed map, power management and optimizing bus traffic. The speed map is used by nodes to determine what speed it can use to communicate with other nodes. The topology map may be used by nodes with a sophisticated user interface that could instruct the end user on the optimum connection topology to enable the highest throughput between nodes. The bus manager is also responsible for determining whether the node that has become Root Node is cycle master capable. If it is not, the bus manager searches for a node that is cycle master capable and forces a bus reset that will select that node as Root Node. There may not always be a bus manager capable node on a bus, in this case at least some of the bus management functions are performed by the isochronous resource manager.

2.3 1394 Open Host Controller Interface

The 1394 Open Host Controller Interface (Open HCI) is an implementation of the link layer protocol of the 1394 Serial Bus, with additional features to support the transaction and bus management layers. The 1394 Open HCI also includes DMA engines for high-performance data transfer and a host bus interface.

2.3.1 Asynchronous functions

The 1394 Open HCI can transmit and receive all of the defined asynchronous packet. Packets to be transmitted are read out of host memory and received packets are written into host memory, both using DMA.

2.3.2 Isochronous functions

The 1394 OpenHCI is capable of performing the cycle master function as defined by 1394. This means it contains a cycle timer and counter, and can queue the transmission of special packet called a “cycle start” after every rising edge of the 8 kHz cycle clock. The 1394 OpenHCI can generate the cycle clock internally (required) or use an external reference (optional). When not the cycle master, the 1394 OpenHCI keeps its internal cycle timer synchronized with the cycle master node by correcting its own cycle timer with the reload value from the cycle start packet.

Conceptually, the 1394 OpenHCI supports one DMA controller each for isochronous transmit and isochronous receive. The isochronous transmit DMA controller can transmit from each context during each cycle. Each context can transmit data for a single isochronous channel. The isochronous receive DMA controller can receive data for each context during each cycle. Each context can be configured to receive data from a single isochronous channel. Additionally, one context can be configured to receive data from multiple isochronous channels.

2.3.3 Miscellaneous functions

Upon detecting a bus reset, the 1394 Open HCI automatically flushes all packets queued for asynchronous transmission. Isochronous packet reception continues without interruption, and a token appears in the received request packet stream to indicate the occurrence of the bus reset. When the PHY provides the new local node ID, the 1394 OpenHCI loads this value into its Node ID register. Asynchronous packet transmit will not resume until directed to by software. Because target node ID values may have changed during the bus reset, software will not generally be able to re-issue old asynchronous requests until software has determined the new target node IDs.

Isochronous transmit and receive functions are not halted by a bus reset; instead they restart as soon as the bus initialization process is complete.

A number of management functions are also implemented by the 1394 Open HCI:

- A global unique ID register of 64 bits which can only be written once. For full compliance with higher level standards, this register shall be written before the boot block is read. To make this implementation simpler, the 1394 Open HCI optionally has an interface to an external hardware global unique ID (GUID, also known as the IEEE EUI-64).
- Four registers that implement the compare-swap operation needed for isochronous resource management.

2.3.4 Hardware description

Figure9 provides a conceptual block diagram of the 1394 OpenHCI, and its connections in the host system. The 1394 Open HCI attaches to the host via the host bus. The host bus is assumed to be at least 32 bits wide with adequate performance to support the data rate of the particular implementation (100Mbit/sec or higher plus overhead for DMA structures) as well as bounded latency so that the FIFO's can have a reasonable size.

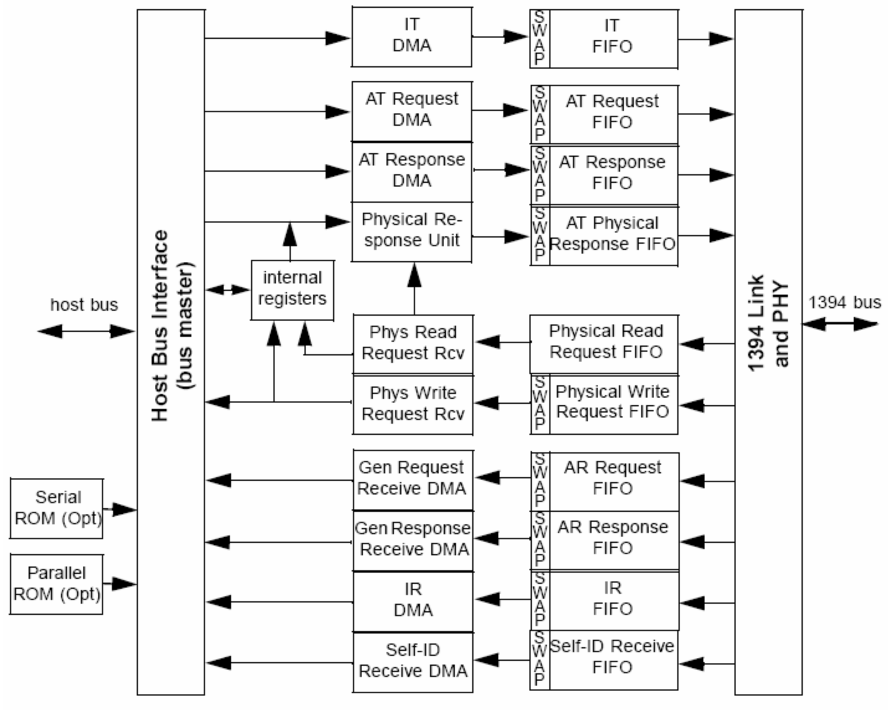

Figure 9 1394 Open HCI conceptual block diagram

Host Bus Interface

This block acts both as a master and a slave on the host bus. As a slave, it decodes and responds to register access within the 1394 Open HCI. As a master, it acts on behalf of the 1394 Open HCI DMA units to generate transactions on the host bus. These transactions are used to move streams of data between system memory and the devices, as well as to read and write the DMA command lists.

DMA

The 1394 OpenHCI supports seven types of DMA, as listed in Table4. Each type of DMA has reserved register space and can support at least one distinct logical data stream referred to as a DMA context.

Table 4 DMA controller types and contexts

DMA Controller Type	Number of Contexts
Asynchronous Transmit	1 Request, 1 Response
Asynchronous Receive	1 Request, 1 Response
Isochronous Transmit	4 minimum, 32 maximum
Isochronous Receive	4 minimum, 32 maximum
Self-ID Receive	1
Physical Receive & Physical Response	0 (not programmable like those above)

Each asynchronous and isochronous context is comprised of a buffer descriptor list called a DMA context program, stored in main memory. Buffers are specified within the DMA context program by DMA descriptors. Although there are some differences from DMA controller to DMA controller as to how the DMA descriptors are used, all DMA descriptors use the same basic format. The DMA controller sequences through its DMA context program(s) to find the necessary data buffers. The mechanism for sequencing through DMA contexts differs somewhat from one controller to the next.

The Self-ID receive controller does not utilize a DMA context program and consists instead of a pair of registers; one to be configured by software, and one to be maintained by hardware.

The 1394 Open HCI also has a physical request DMA controller that processes incoming requests that read directly from host memory. This controller does not have a DMA context; it is instead controlled by dedicated registers.

Chapter 3 Experiment Setup

3.1 Functional requirements of the setup

To examine the real-time characteristic of Firewire, two kinds of time overhead should be measured: time overhead on the interface between application and protocol stack, and time overhead on protocol stack and bus, as demonstrated in Figure10.

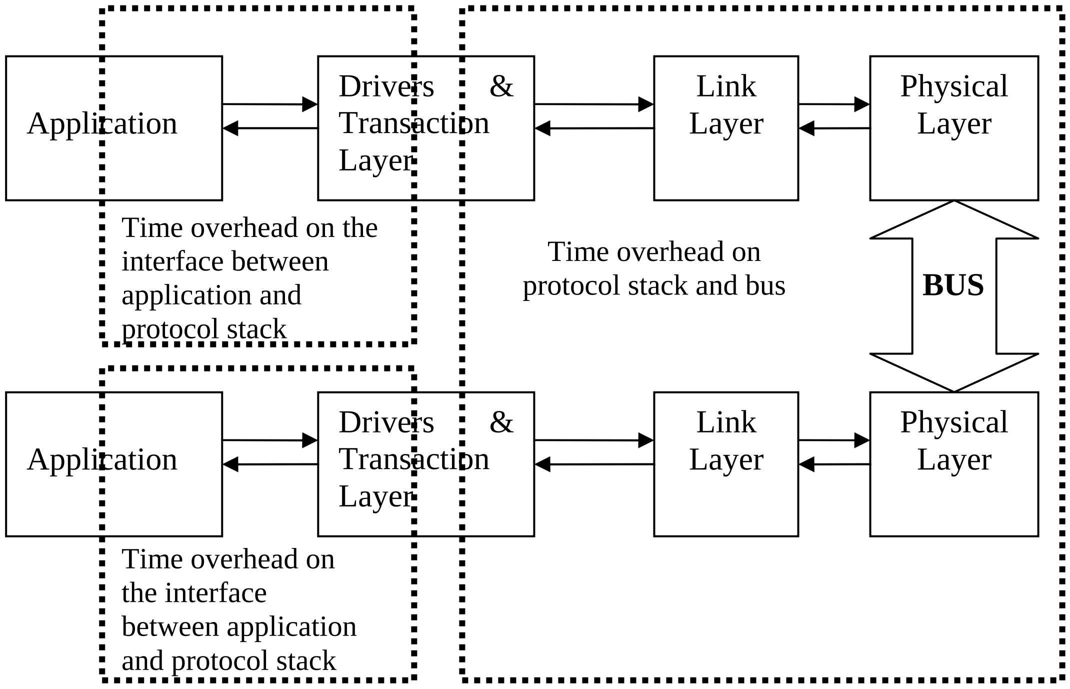
Application
Drivers & Transaction Layer
Link Layer
Physical Layer
Time overhead on the interface between application and protocol stack
Time overhead on protocol stack and bus
BUS
Application
Drivers & Transaction Layer
Link Layer
Physical Layer
Time overhead on the interface between application and protocol stack

Figure 10 Time overhead during asynchronous transmission

Time overhead on the interface between application and protocol stack is the time for transferring the data between application and Firewire protocol stack. This includes reading and writing data from and to protocol stack. Time overhead on protocol stack and bus is the time for transferring the data through the protocol stack and bus. This includes not only the time overhead on the Firewire board and bus, but also the time overhead on the software drivers for Firewire.

Besides the average values of the two kinds of time overhead, the variation from the average values of them, or so called jitter, should also be determined.

3.2 Structure of the setup

To measure these values, a 3-node Firewire network was built. The Firewire network structure is shown in Figure11.

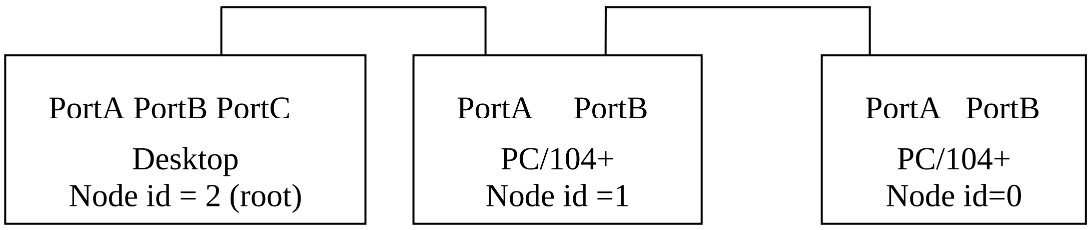

Figure 11 Firewire network diagram

As demonstrated by Figure13, the topology of the network is fixed by forcing the node id of desktop to be 2, which is root. The reason of doing this is because the change of topology may affect the experiment results.

3.3 Hardware used

The hardware used in this setup is listed below:

- Desktop CPU: AMD Athlon 700MHz
- Desktop Firewire Card: PCI card with NEC PD72874 as Link layer and Physical layer chip
- PC/104 CPU: VIA Eden 600MHz
- PC/104 Firewire board: PC/104+ board with VIA VT6307L as Link layer and Physical layer chip (from Ampro Computer, Inc)

For Firewire board on PC/104 stack, an investigation was made for the available products in market. Specifications of all these boards, and the decision table for comparison, are put in appendix A.3.

The same kind of PC/104 has also been used in other projects [Buit E., 2004] at the CE lab.

3.4 Software used

3.4.1 Operating System

The operating systems used for both Desktop and PC/104 boards are Linux 2.4.26, which is the latest 2.4 version of the Linux Kernel.

3.4.2 RTAI

RTAI means Real Time Application Interface [RTAI, 2004]. It is based on the Linux kernel, providing the ability to make Linux fully pre-emptable.

RTAI offers the same services of the Linux kernel core, adding the features of an industrial real-time operating system. It consists basically of an interrupt dispatcher: RTAI mainly traps the peripherals interrupts and if necessary re-routes them to Linux. RTAI uses the concept of HAL (hardware abstraction layer) to get information from

Linux and to trap some fundamental functions RTAI considers Linux as a background task running when no real-time activity occurs.

There are two ways to implement the application program with RTAI, i.e. implementing in Kernel space as a module or implementing it in User space with LXRT (LinuX Real Time). To make the application in Kernel space, it means the application program should be programmed without any library support. The only external functions, which can be used during programming for Kernel space use, come from the Kernel itself. These functions are called system call, which can only do very basic work.

Another option is to implement the application in User space with LXRT (LinuX Real-Time). By using LXRT, the real-time threads and non-real-time threads can be combined in one program. During the execution of the program, the context will be switched between Kernel space and User space. But like programming in Kernel space, the same trick is also here: if the real-time part of the program is not implemented with only support from system calls, it will still be executed in User space, even if it is required to be real-time.

3.4.3 Drivers for Firewire

The Firewire subsystem in Linux kernel space, which is used in this setup, is presented in Figure12. [Linux1394, 2004]

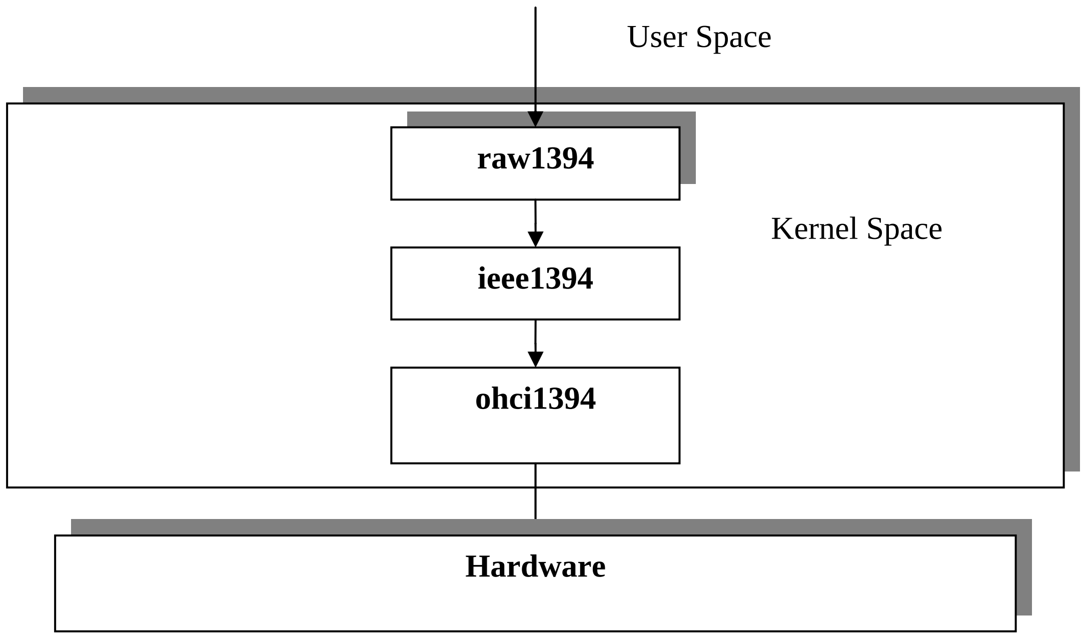

Figure 12 Driver Hierarchy of Firewire

The core of the entire subsystem is the module ieee1394. It manages all high-level and low-level drivers, handles transactions, and provides a mechanism for event triggering. Below the ieee1394 module are the low-level (hardware) driver modules, which is ohci1394. Above the ieee1394 module are the high-level driver modules, which is raw1394. This driver provides an interface for user space applications to access the raw 1394 bus.

To access the raw1394 bus from user space, all the application should be linked with libraw1394 [libraw1394, 2004], which handles the communication with the raw1394 high-level driver.

Because raw1394 is a high level driver, there is no interrupt support for it. So the border between the time overhead on the interface between application and raw1394 and the time overhead on the whole protocol stack and bus can not be determined. Then only the sum of the two time overheads can be measured, if the measurement is done on the raw1394 level.

If integrate the driver hierarchy of the Firewire subsystem into Figure10, the setup architecture, including both software and hardware, can be seen more clearly, which is presented in Figure13. The concept is time overhead summing is presented by the dashed line.

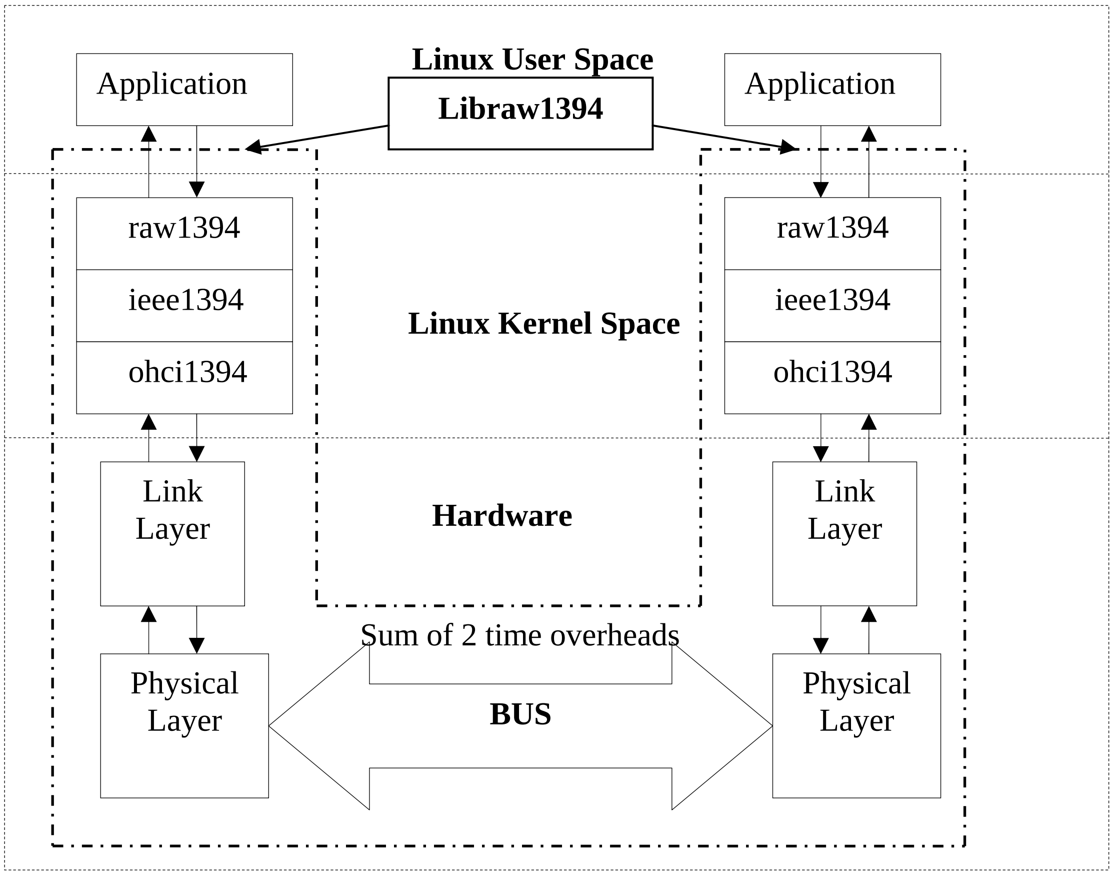

Figure 13 Complete structure of experiment setup

3.5 Implementation of asynchronous transmission

There are two ways of transferring data asynchronously on Firewire, using FCP or using ARM.

ARM stands for Address Range Mapping. It maps a certain range of the host memory to a certain range of the address space in Firewire network. Both the range of host memory and the range of address space in Firewire network are defined by user. So when the certain range of address space in Firewire network is targeted in an asynchronous packet, actually the corresponding range of the host memory will be accessed. The access to host memory is implemented through DMA (Direct Memory Access). Since the transmission can directly access the host memory, there is no packet payload limitation besides the maximum data payload for asynchronous packet, which is 2048 bytes when the speed is 400 Mbits/s. There is some example code of how to use ARM on Manfred Weih's site [Weih M., 2004].

If only a small data payload needs to be transferred, then FCP (Function Control Protocol) turns to be a good choice. The FCP uses the FCP COMMAND register and FCP RESPONSE register in the CSR (Control Status Registers) architecture. See chapter 21 of *Firewire System Architecture 2nd Edition* [Anderson D., 1999]. Unlike ARM, the destination offset address of a FCP message is not user defined. It is fixed to the register address of FCP COMMAND or FCP RESPONSE. And for the space in register is limited, the maximum data payload for FCP packet is 512 bytes, which is much lower than ARM. In a real time control context, the data payload is not required to be high, so the relatively low payload of FCP will not cause any problem. Due to the time limitation, only FCP was implemented in this project.

Chapter 4 Design of the Test Bench

4.1 Roundtrip of Point-to-Point Connection

To examine the real-time characteristic of Firewire's asynchronous transmission, a roundtrip test bench was used.

The roundtrip test bench consists of 2 nodes. One is called server and another is called client. The test bench has a similar principle with Eric Buit's benchmark for RT-Ethernet [Buit E., 2004]. The sequence diagram is presented in Figure14.

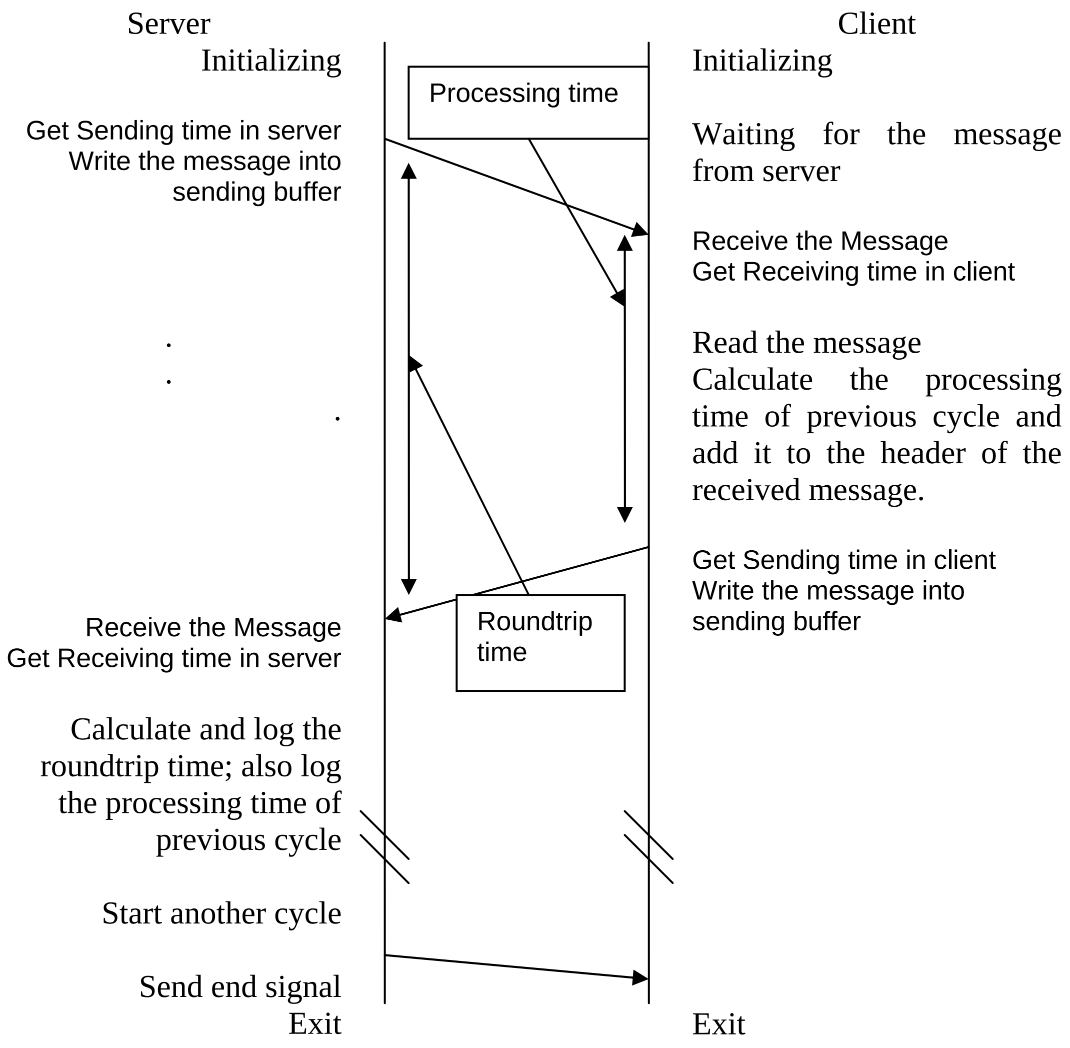
Server
Client
Initializing
Initializing
Processing time
Get Sending time in server
Write the message into sending buffer
Waiting for the message from server
Receive the Message
Get Receiving time in client
Read the message
Calculate the processing time of previous cycle and add it to the header of the received message.
Get Sending time in client
Write the message into sending buffer
Receive the Message
Get Receiving time in server
Roundtrip time
Calculate and log the roundtrip time; also log the processing time of previous cycle
Start another cycle
Send end signal
Exit
Exit

Figure 14 Sequence Diagram of Asynchronous Transmission Test Bench

After initializing, the server node first read the current time as "Sending time in server" of cycle1. Afterwards, it sends a message to client node. When the sending process is finished, it starts waiting for the arrival of return message.

In the client node, when it receives the message from server node, it reads the time as "Receiving time in client" of cycle1. Afterwards, it reads the message, calculates the processing time of previous cycle and adds the result to the header of the received message. When this is completed, the client node reads the time as “Sending time in client”, sends the message back to the server node, and starts waiting for next message.

When the server node receives the return message from client node, it reads the time as “Receiving time in server”, calculates the roundtrip time and logs it. Also it reads the processing time of client node in previous cycle and logs it. After that, the server can either start a new cycle or send an exit message to client.

The user can press Ctrl+C in server node to make it send an exit message to client. The server node will also exit after that.

From Figure14, it can be seen that the roundtrip time consists of 2 parts, as demonstrated in Figure15:

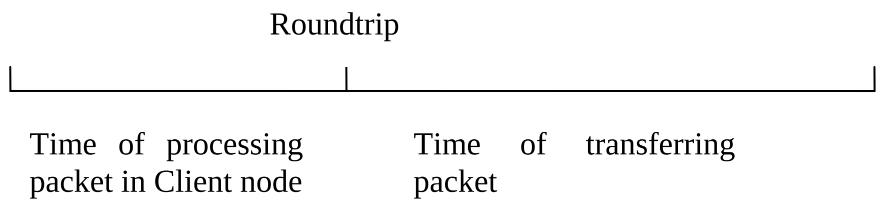
Roundtrip
Time of processing packet in Client node
Time of transferring packet

Figure 15 Roundtrip time

The time of transferring packet is the sum of the time overhead on interface between application and Firewire protocol and the time overhead on transferring the packet through protocol layers and bus, as presented in last chapter. Instead of being measured directly, it is the result of subtracting the time of processing packet in Client node from the total Roundtrip time.

For the sake of easy notation, the time of processing packet in Client node is called Processing Time and the time of transferring packet is called Transferring Time in the rest of this report.

4.2 Time measurement function

The time at different point is read by using gettimeofday( ), an API function in Linux. This function returns the current time expressed as seconds and microseconds since 00:00 Coordinated Universal Time (CUT), January 1, 1970.

4.3 The way to do measurement real time

Currently, libraw1394, the library which is used to access Firewire bus in this experiment, is not programmed and complied for Kernel space use, so the measurement application can not be made in Kernel space. Another option is to do the measurement in User Space, which requires LXRT. But even if LXRT is used, all the functions in libraw1394 will still be run in User space.

Then the only solution is to make the measurement application run in User space, but with the highest priority within an appropriate scheduling algorithm. The algorithm must guarantee the control of the processor to the highest priority process until that process willingly releases the processor or is blocked on a contended resource. SCHED_FIFO is such an algorithm. In SCHED_FIFO, when there is more than one runnable highest priority process, the highest priority process waiting the longest is granted control of the processor.

Following is the program template to set the scheduling algorithm and priority to certain process. [Soetens P., 2004]

```
int main(void)
  {
    struct sched_param mysched;

    mysched.sched_priority = sched_get_priority_max(SCHED_FIFO) - 1;
    if( sched_setscheduler( 0, SCHED_FIFO, &mysched ) == -1 ) {
      puts("ERROR IN SETTING THE SCHEDULER");
      perror("errno");
      exit(1);
    }
/* my application */

    return 0;
  }
```

The *sched_setscheduler* system call is used to set the scheduling policy and related parameters for the process specified by *pid*. If *pid* is set to zero, then the scheduling policy and parameters of the calling process will be affected.

Chapter 5 Measurement Results and Discussion

5.1 Measurement results

Figure16, 17, 18 show the roundtrip time, processing time and transferring time in the roundtrip between Desktop and PC/104(1) with payload=1byte.

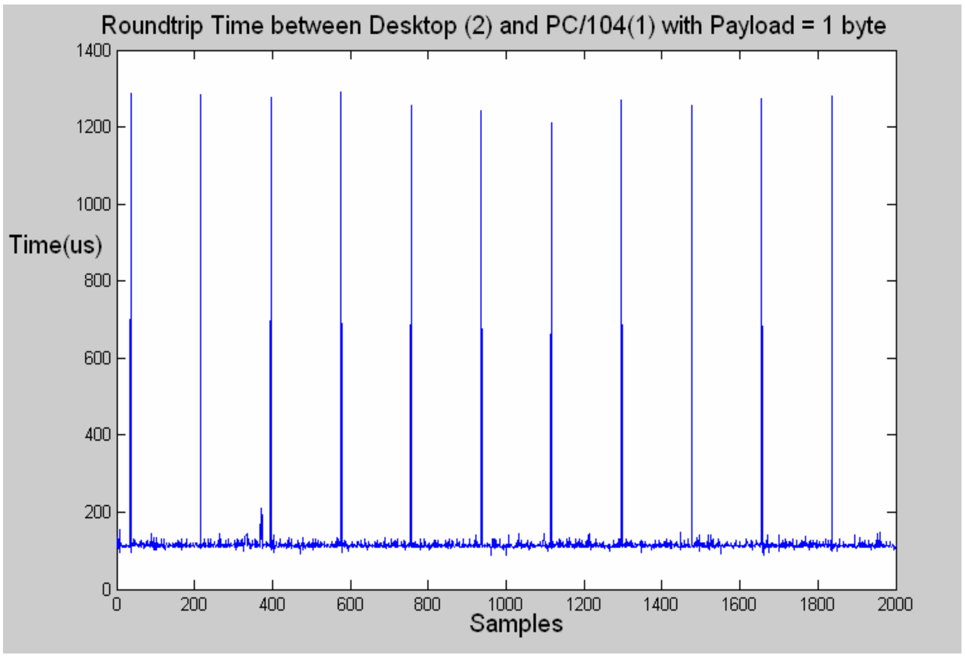

Figure 16 Roundtrip Time between Desktop (2) and PC/104(1) with Payload = 1byte

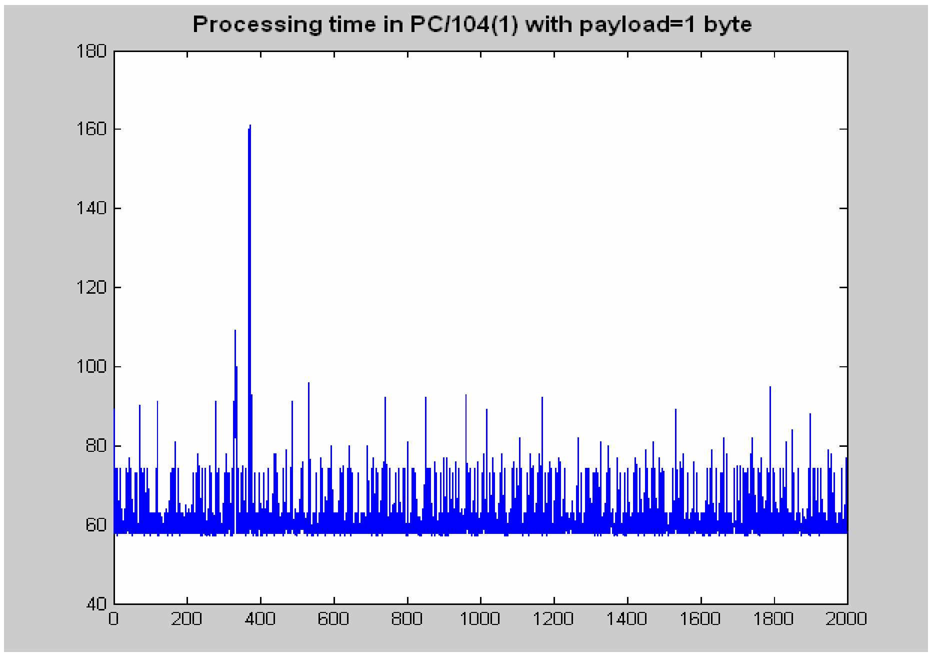

Figure 17 Processing Time in PC/104(1) with Payload=1byte

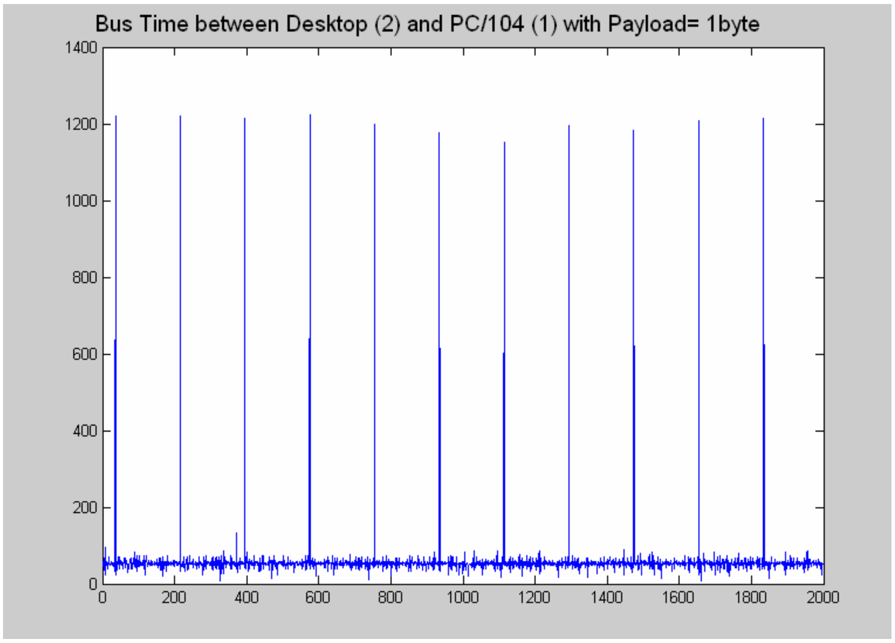

Figure 18 Transferring Time between Desktop and PC/104(1) with Payload=1byte

It is notable that there is a periodical peak in the Roundtrip time and Transferring time, but no peak in the Processing Time. So the jitter is in the Transferring time. After a careful observation, it was found that the AT (Asynchronous Transmission) DMA resets periodically during the test, and the number of resets and the number of peaks are the same. Therefore it is very possible that AT DMA's periodical reset is the cause of the periodical peak in Bus time.

Due to the time limitation, the solution for the DMA reset problem was not found in this project, but it is assumed that the peak problem can be solved if something can be changed in the software of the test bench. Hence, the experiment went on, temporarily skipping the peaks.

Figure19 shows the Roundtrip time has a small variation without the peaks. This gives an impression that the asynchronous transmission of Firewire is deterministic.

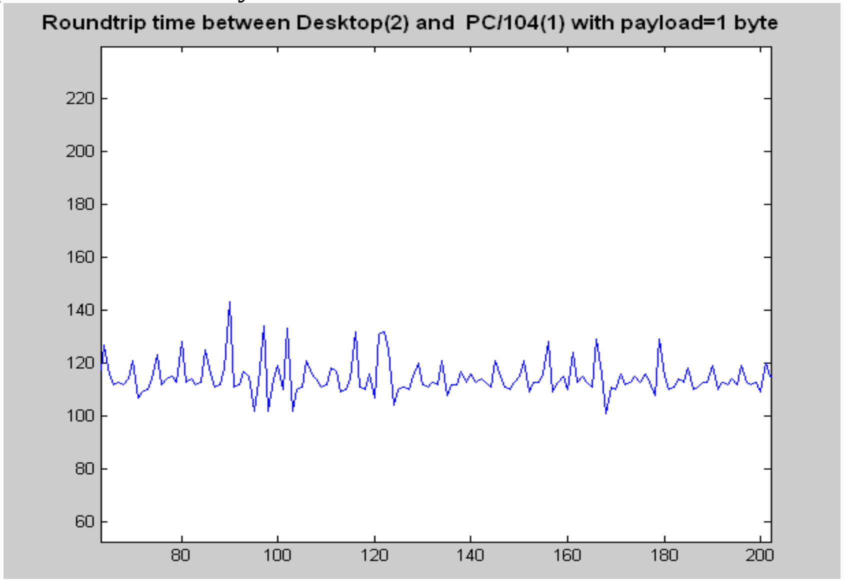

Figure 19 Partly Zoom in of Figure18

Figure 20 and 21 show the Roundtrip time and Processing time when the payload is increased to 129 bytes.

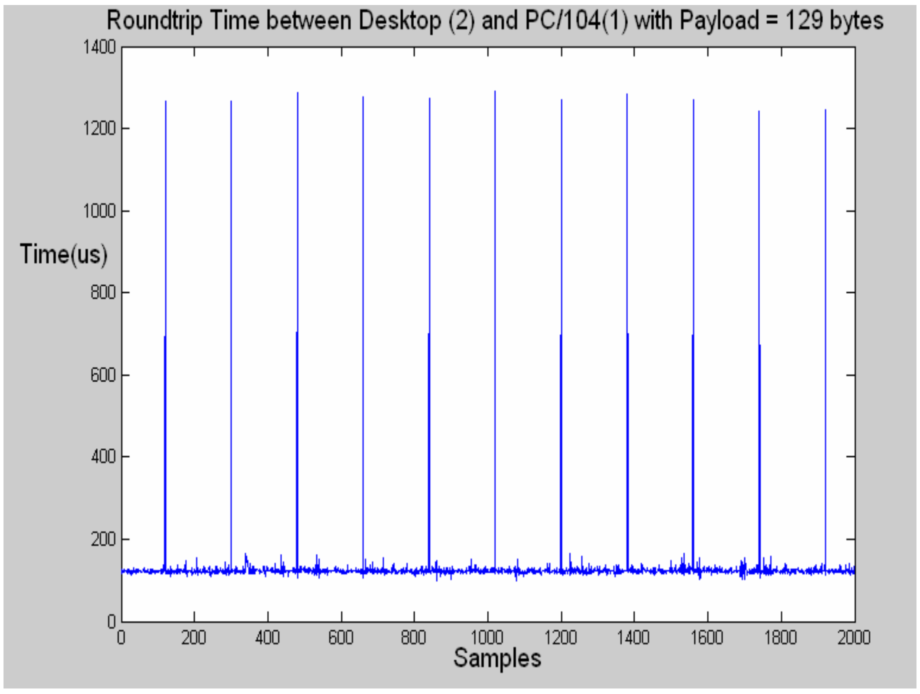

Figure 20 Roundtrip Time between Desktop and PC/104(1) with Payload=129bytes

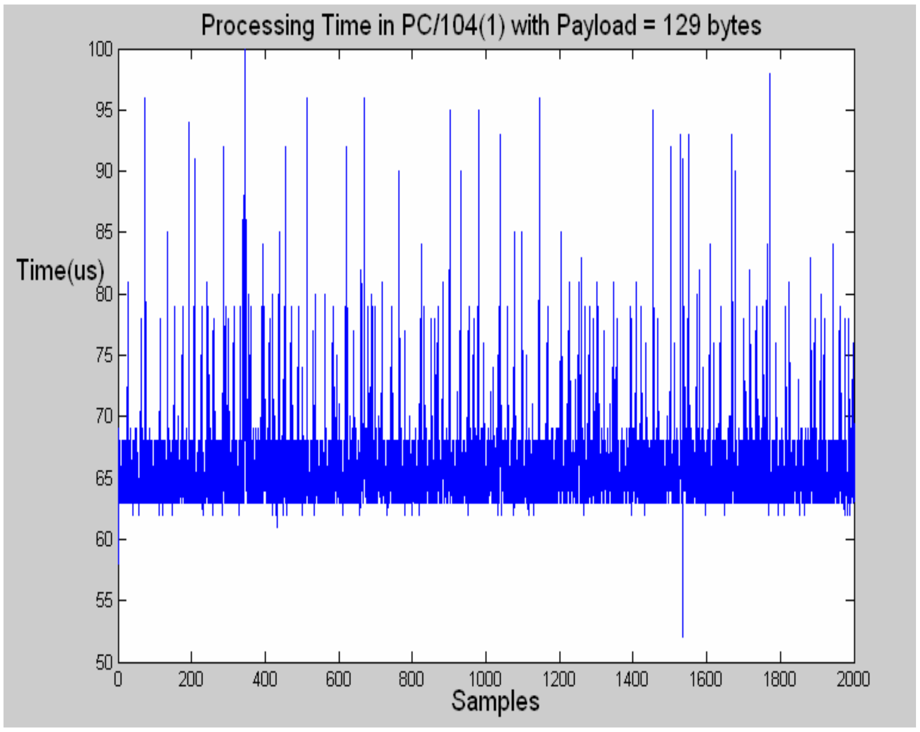

Figure 21 Processing Time in PC/104(1) with Payload=129 bytes

It can be seen that besides the increase on the average value (around 5μs), the whole distribution and variation of the Roundtrip time and Processing time are still similar as when the payload equals to 1byte. So to make the comparing easier, instead of presenting separate plots with every different payload here, the average values of Roundtrip time, Processing time and Transferring time at various payloads are plotted together in Figure22. The payload is increased from 1byte to 449 bytes, 64 bytes as step size. To calculate the average value, the first 10 cycles at various payloads were chosen. For the numerical values, please refer to the appendix A.2.

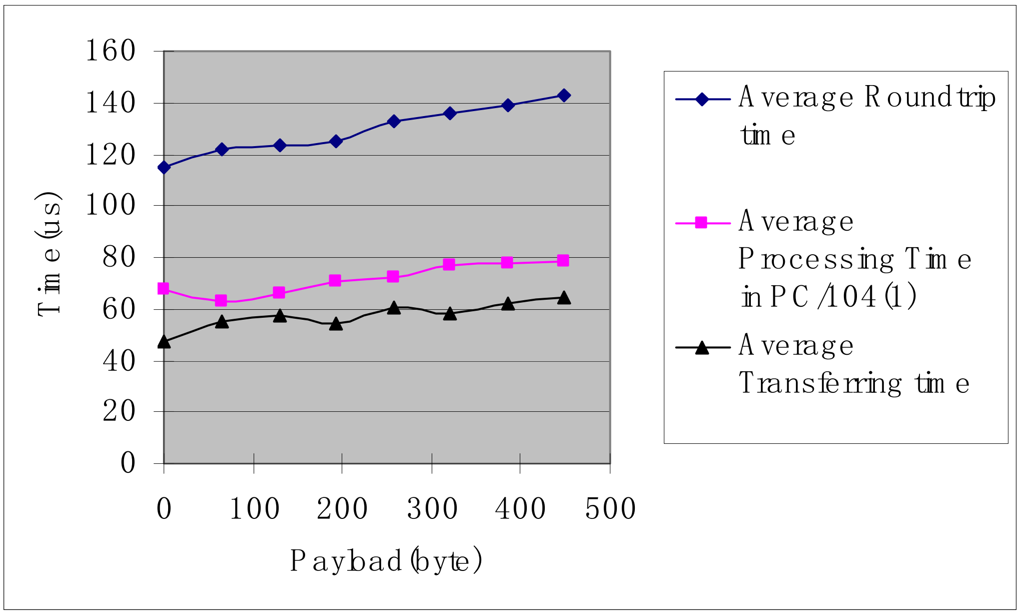

Figure 22 Measurement Results of Roundtrip between Desktop (2) and PC/104(1)

The results of roundtrip between Desktop (2) and PC/104(0) are given in Figure23.

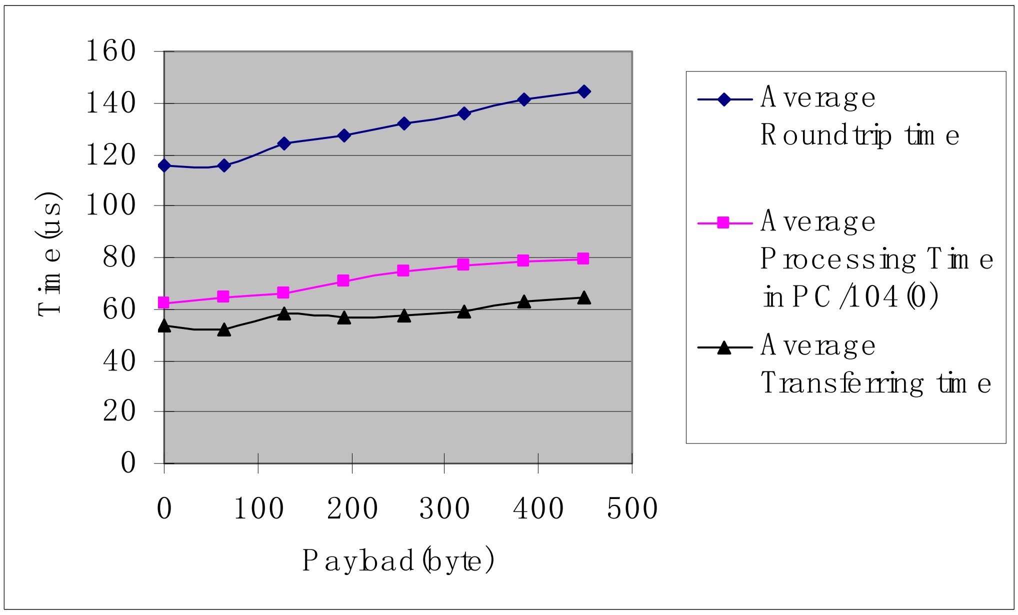

Figure 23 Measurement Results of Roundtrip between Desktop (2) and PC/104(0)

The results of roundtrip between PC/104(1) and PC/104(0) are given in Figure24.

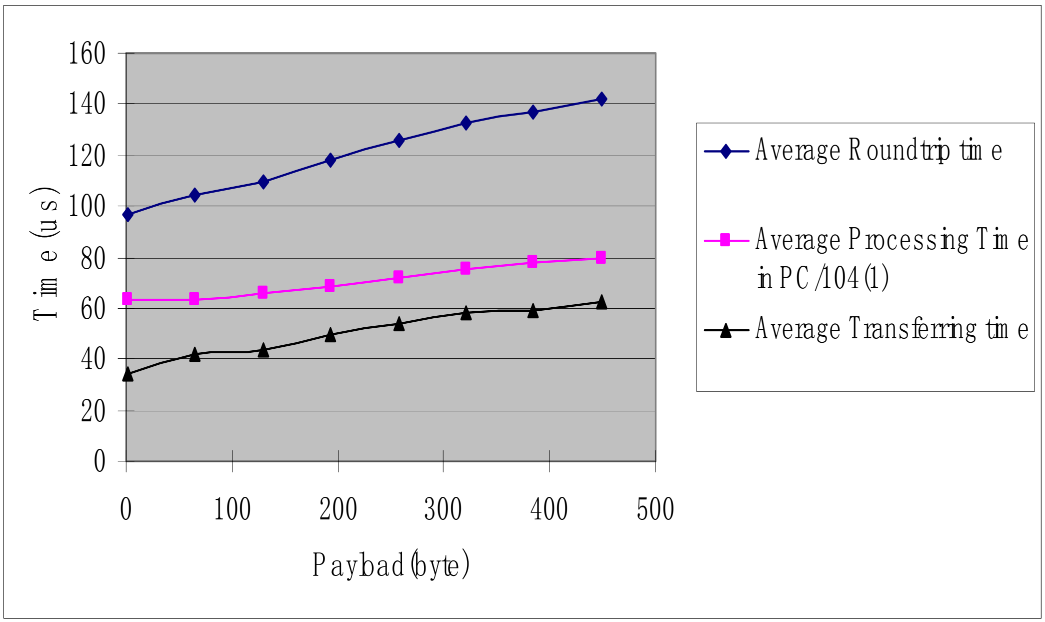

Figure 24 Measurement Results of Roundtrip between PC/104(0) and PC/104(1)

The Standard Deviation of the Roundtrip time between different two nodes is given in Figure25.

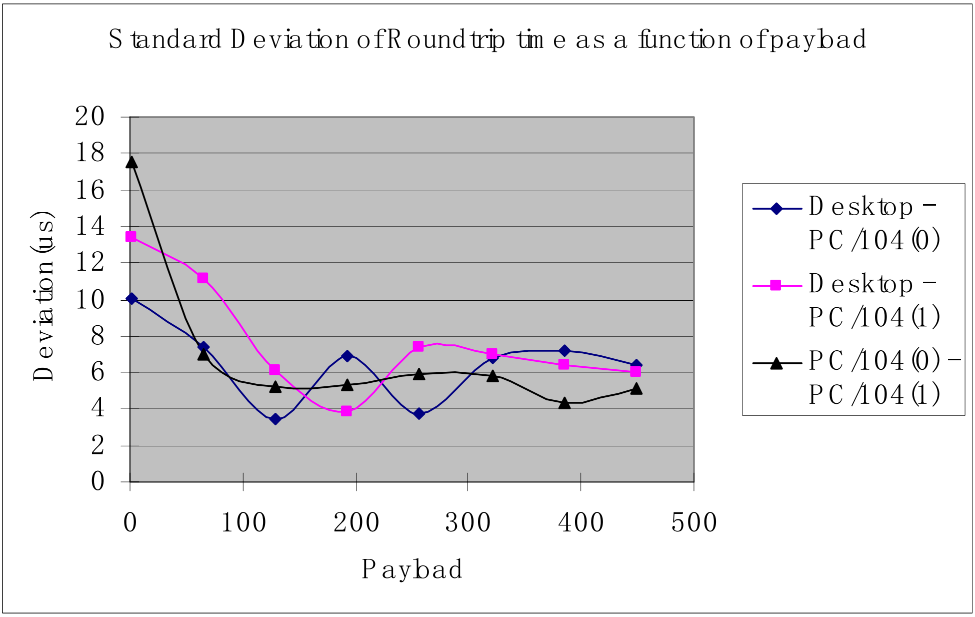

Figure 25 Standard Deviation of Roundtrip time as a function of payload

The Standard Deviation of the Transferring time between different two nodes is given in Figure26.

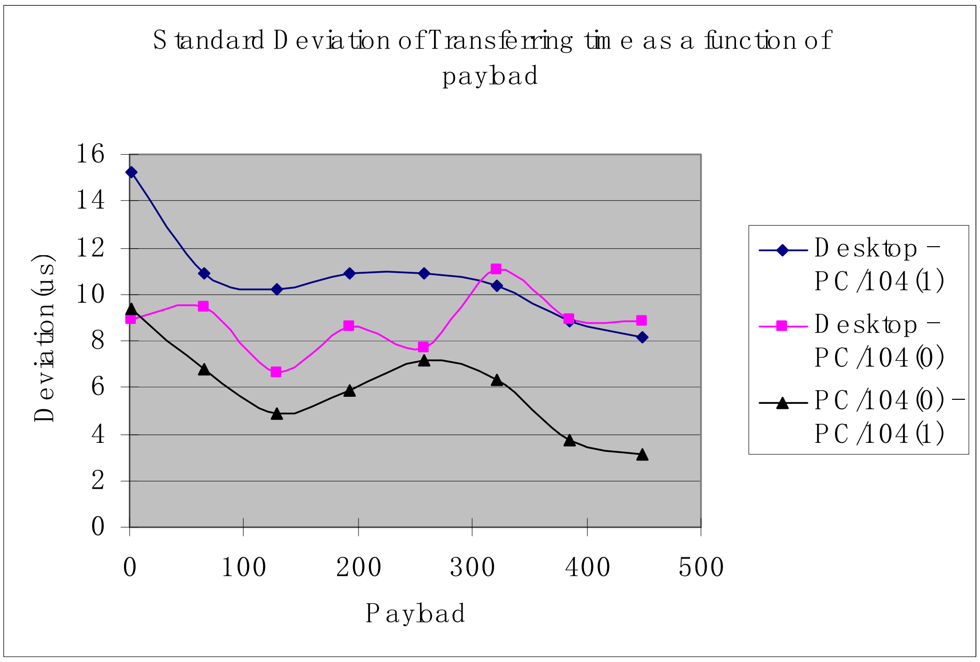

Figure 26 Standard Deviation of Transferring time as a function of payload

In real-time control, it is also important to specify the variation of transmission speed. Here, variation is determined as the difference between maximum value and minimum value.

The variation range of Transferring time is given in Figure27.

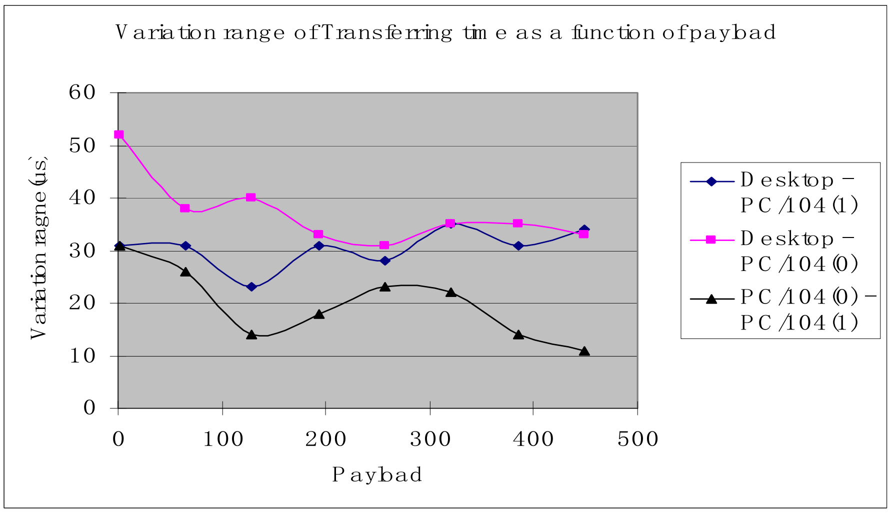

Figure 27 Variation range of Transferring time as a function of payload

The variation range of Roundtrip time is given in Figure28.

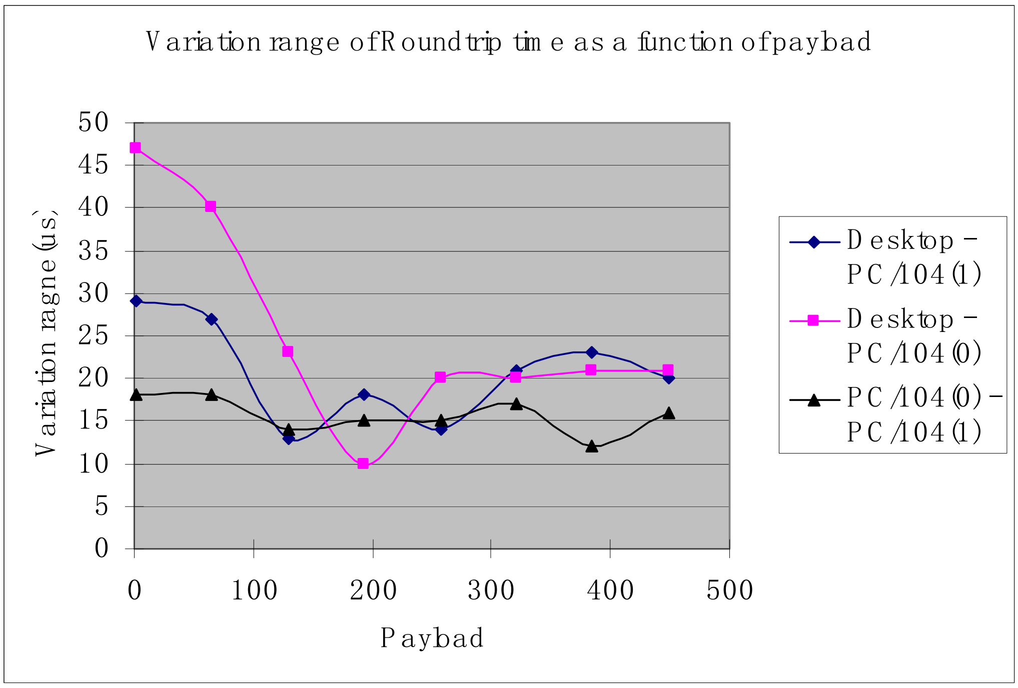

Figure 28 Variation range of Roundtrip time as a function of payload

5.2 Discussion

5.2.1 Average latency

The average latency of Roundtrip time between PC/104 stacks is from 100µs to 140µs, depending on the data payload. The average latency of Roundtrip time between Desktop and PC/104 stack is from 120µs to 150 µs, depending on the data payload.

5.2.2 Real-Time characteristic

Take the point-to-point connection between two PC/104 stacks as an example. When transferring packets of constant length, the Roundtrip time between these two stacks changes in a range of 10-20µs. The transmission with this variation range (jitter) is deterministic for a distributed control system with normal real-time requirement.

5.2.3 Protocol overhead and bus efficiency

Still take the point-to-point connection between two PC/104 stacks as an example. When the payload is increased, the Roundtrip time does not show a significant increase, the increasing range is only 40µs from 1byte payload to 449 bytes payload. So it can be seen that when transferring higher data payload, the protocol overhead in one packet is relatively lower, so the Roundtrip time is not increased linearly with the increasing data payload. This means the bus is used most efficiently when the packet is fully loaded.

5.2.4 Stable performance of PC/104 stack

As shown in Figure27, the variation range of Transferring time between PC/104 stacks is obviously lower than the other two. This implies the more stable performance of PC/104 stack, compared with Desktop. This is because the Firewire chip in Desktop is a cheap one, so not reliable to provide high, industrial performance. The chip in PC/104 stack is more expensive, so provides a relatively high performance.

Chapter 6 Conclusions and Recommendations

6.1 Conclusions

- The average latency of asynchronous transmission is from 100µs to 150µs, depending on the data payload.
- Asynchronous Transmission of Firewire is deterministic in a certain range. For the Roundtrip time between 2 PC/104 stacks, the range is 10-20µs. This means the asynchronous transmission of Firewire can be used for distributed control systems without a strict real-time requirement.
- The latency increase due to the increasing data payload is not very significant. For the Roundtrip time between 2 PC/104 stacks, the increase range is only 40µs from 1 byte to 449 bytes. This implies the asynchronous transmission of Firewire can still give a deterministic behavior when the data payload is variable.
- The absolute protocol overhead is constant. When the data payload is higher, the protocol overhead is relatively lower. Hence the bus is used more efficiently. It can be seen that during asynchronous transmission, the bus is most efficiently used when the packet is fully loaded.

6.2 Recommendations

- Find solution for peak problem.
- Implement ARM, another way of doing asynchronous transmission on Firewire.
- Implement isochronous transmission of Firewire.
- Implement Firewire into CT library
- Use Firewire as the field bus in a real distributed control system with a physical plant.

Appendix

A.1 Investigations of other alternative field buses for this project

1. DS-Link

DS-Link is part of the IEEE 1355-1995 standard, which is standardized for heterogeneous interconnected system. It is a point-to-point connection between devices, including chips, boards, etc. It implements a fast, reliable and low cost link for distributed system. Because DS-Link was originally designed for transputer network, it should be suitable for real-time distributed control system. [IEEE1355, 1998]

Besides the PCI-1355-01/02 board from 4Links, there is a lack of other available products of DS-Link in the market.

2. HS-Link

HS-Link is another part of IEEE 1355-1995 standard, which is also for heterogeneous interconnected system. Its difference from DS-Link is its physical layer implementation. It uses high-speed fiber or coaxial cable, while DS-Link uses cable or copper wire. In the protocol stack of the two links, they are almost similar.

In the project of Arches (1998), CERN designed the PCI HS-Link interface board and HS-Link switch. In the project of Arches 2 (1999), CERN designed a GE-HSL adapter, which is the interface between IEEE 802.3z Gigabit-Ethernet and IEEE 1355 HS-Links.

The FastHSL board was developed by ASIM (Équipe Achitecture des Systèmes Intégrés et Micro-Électronique) at the Laboratoire d'Informatique de Paris 6 in the Université Pierre et Marie Curie. It is PCI-Controller for IEEE 1355/HSL Networks. It includes a PCI controller implementing the Direct-DepositState-Less Receiver communication protocol (PCI-DDC chip), and an 8x8 router using Gigabit per second HS-Links (RCube chip). As the board contains both protocol and routing capability, it is possible to build clusters of PCs with one FastHSL board per PC.

But it seems no company is offering the products of HS-Link now.

3. Spacewire

Spacewire is another standard of high speed data link, which is intended to meet the needs of future, high capability, remote sensing instruments and other space mission. Spacewire provides a unified high speed data-handling infrastructure for connecting together sensors, processing elements, mess memory units, downlink telemetry subsystems, etc. Spacewire has taken into consideration two existing standards, IEEE 1355-1995 and ANSI/TIA/EIA-644(LVDS). It is specifically for use onboard a spacecraft. [Spacewire, 2004]

Spacewire is mainly developed in 4Links. Currently, there are a lot of products from that company, including bridge between Spacewire and PCI/cPCI (Compact PCI), but most of them have not been shipped to market yet, and the price of the available are expensive. [4Links, 2004]

A.2 Numerical Values of Measurement Results

Table 5 Roundtrip time between Desktop (2) and PC/104(0)

Payload (byte)	1	65	129	193	257	321	385	449
Average Roundtrip Time (µs)	115.6	116	124	127.4	132	136.3	141.4	144.2
Max (µs)	134	130	130	139	141	148	157	156
Min (µs)	105	103	117	121	127	127	134	136
Standard Deviation	10.04	7.37	3.49	6.85	3.74	6.76	7.22	6.42

Table 6 Processing time in PC/104(0)

Payload (byte)	1	65	129	193	257	321	385	449
Average Processing Time (µs)	62.2	64.3	65.7	70.7	74.2	77.1	78.4	79.5
Max (µs)	81	78	78	82	83	97	88	92
Min (µs)	58	60	63	66	68	71	73	75
Standard Deviation	6.97	6.23	4.76	6.48	4.98	8.38	4.16	5.21

Table 7 Transferring time between Desktop (2) and PC/104(0)

Payload (byte)	1	65	129	193	257	321	385	449
Average Transferring time (µs)	53.4	51.7	58.3	56.7	57.8	59.2	63	64.7
Max (µs)	73	60	66	72	72	75	80	80
Min (µs)	42	29	43	41	44	40	49	46
Standard Deviation	8.93	9.45	6.63	8.63	7.70	11.01	8.92	8.87

Table 8 Roundtrip time between Desktop (2) and PC/104(1)

Payload (byte)	1	65	129	193	257	321	385	449
Average Roundtrip Time (µs)	114.9	122	123.4	124.8	132.6	135.7	139.2	143.1
Max (µs)	148	136	139	130	145	148	155	157
Min (µs)	101	96	116	120	125	128	134	136
Standard Deviation	13.37	11.17	6.06	3.80	7.40	7.00	6.43	6.01

Table 9 Processing Time in PC/104(1)

Payload (byte)	1	65	129	193	257	321	385	449
Average Processing Time (µs)	67.7	63.2	66.2	70.3	72.3	77.2	77.4	78.4
Max (µs)	84	76	81	90	84	98	90	91
Min (µs)	58	60	63	65	68	70	73	75
Standard Deviation	10.10	5.20	5.85	8.59	6.11	9.05	5.02	4.97

Table 10 Transferring time between Desktop (2) and PC/104(1)

Payload (byte)	1	65	129	193	257	321	385	449
Average Transferring time (µs)	47.2	54.8	57.2	54.5	60.3	58.5	61.8	64.7
Max (µs)	72	73	75	64	77	77	79	81
Min (µs)	20	35	35	31	46	42	44	48
Standard Deviation	15.22	10.92	10.21	10.93	10.86	10.36	8.83	8.15

Table 11 Roundtrip time between PC/104(0) and PC/104(1)

Payload (byte)	1	65	129	193	257	321	385	449
Average Roundtrip Time (µs)	97	104.5	109.5	117.8	125.7	132.9	137.1	142.3
Max (µs)	147	115	118	128	134	142	143	151
Min (µs)	91	97	104	113	119	125	131	135
Standard Deviation	17.53	7.04	5.20	5.31	5.87	5.86	4.33	5.08

Table 12 Processing time in PC/104(1)

Payload (byte)	1	65	129	193	257	321	385	449
Average Processing Time (µs)	62.9	62.9	66.2	68.6	71.6	75	78	79.6
Max (µs)	68	72	73	79	77	84	85	88
Min (µs)	56	56	59	62	66	68	72	71
Standard Deviation	5.70	5.89	5.47	6.06	4.77	5.73	5.31	6.02

Table 13 Transferring time between PC/104(0) and PC/104(1)

Payload (byte)	1	65	129	193	257	321	385	449
Average Transferring Time (μs)	16.5	21.6	23.3	29.2	34.1	37.9	39.1	42.7
Max (μs)	34	32	28	34	45	51	45	46
Min (μs)	3	6	14	16	22	29	31	35
Standard Deviation	9.34	6.80	4.90	5.90	7.16	6.30	3.72	3.16

A.3 Selection of Firewire boards

1) SedNet of Mindready

Mindready's SedNet™ 1394a and 1394b OHCI board family provides a complete range of industrial-grade interface communication adapters for the IEEE-1394/FireWire bus. The hardware specifications are listed in the table below:

Table 14 Hardware Specifications of Mindready's SedNet board

Form Factor	PCI, Compact PCI 3U and 6U, PMC single deck, PC104+
Connector Type	• Standard IEEE-1394a 6-pin connector for IEEE-1394a units • Standard IEEE-1394b bilingual connectors for IEEE-1394b units • Plastic Optical Fiber (POF) connector
Connector Allocation for IEEE-1394a Devices	Three standard 1394a ports
Connector Allocation for IEEE-1394b Devices	Bilingual and POF ports (see over for details)
Power Consumption	4.8 watts for the card, and up to 15 watts if power is fed to the bus
Host Power Feed	Support of +5 volt and +3.3 volt power supplies
Operating Temperature	0C to +70C (+32F to +158F)
Storage Temperature	-40C to +85C (-40F to +185F)
Host Interface Standard	OHCI v1.0

Link Layer Controller	Texas Instruments TSB12LV26
Physical Layer Controller	Various

The key features among these specifications are the form factor (PC/104+) and host interface standard (OHCI). Therefore, the firewire board of Mindready perfectly fits in my project.

2) FireSpeed2000 of ADVANCED MICRO PERIPHERALS Ltd

The FireSpeed2000 is a high performance Controller providing 3 serial ports conforming to the IEEE-1394 OHCI specification - popularly known as FireWire. The FireSpeed2000 provides an ideal interface for attaching high speed Audio, Video and Storage peripherals to an embedded system. Automatic detection and configuration of devices (without the need for on-board jumpers) ensures robust system operations and high reliability.

PRODUCT HIGHLIGHTS

- Meets IEEE-1394 Fire Wire Standard
- Three high speed Serial Interface
- Speeds of 100/200/400 MBits/sec
- Link up to 63 devices together
- Video/Audio/Mass Storage auto-detection
- Hot pluggable connections
- Multiple FireSpeed2000 cards per system
- High Performance PC/104+ Bus Master
- Drivers for Win95/98/NT/2000, Linux
- Compact 3.6 x 3.8in PC/104+ form factor

3) MiniModule1394 of Ampro Computers, Inc

The MiniModule™ 1394 board provides a high performance PCI-based platform for incorporation of up to two IEEE 1394 Firewire bus interfaces into a PC/104-Plus embedded system. The card provides both PC/104-Plus and PC/104 interfaces, although the PC/104 interface is utilized only to pass through bus signals and for mechanical stability. Control signals are all obtained from the PC/104-Plus (PCI) bus.

Features

- Via VT 6307L IEEE 1394 Host Controller
- Compliant with 1394 Open HCI Rev. 1.0 and 1.1
- Descriptor based isochronous and asynchronous DMA channels for receive/transmit packets
- Fully interoperable with IEEE standard 1395-1995 devices
- Provides two 1394a fully compliant ports at 100/200/400 Mbp
- Cable power presence monitoring

- Supports separate TPBIAS for each port
- Supports 32-bit 33MHz PC/104-Plus host interface compliant with PCI Rev. 2.2
- Configurable for any PC/104-Plus bus “slot”

4) CM17208HR board of RTD Finland

The CM17208HR board is designed to allow a direct interface to various mass storage media and other Firewire peripherals. The two ports are equipped with +12V 1A power supply. This supply allows powering of Firewire devices through the cable simplifying power supply design.

The CM17208HR with the TI TSB43AB21/TSB43AB22 host controller is fully compatible with various operating system drivers, such as Windows 98 Second Edition (SE), Windows (Me), Windows XP and Windows 2000.

Features

- One (CM17208HR) or two (CM17208) IEEE Std 1394a-2000 fully compliant cable ports at 100M bits/s, 200M bits/s, or 400M bits/s
- TI TSB43AB21/TSB43AB22 PCI to IEEE 1394 host controller
- Onboard +12V supply for the IEEE 1394 bus
- PCI burst and deep FIFOs; supports PCI_CLKRUN\ protocol
- Compliance to PCI Rev 2.2, 33MHz 32-bit PCI bus
- PCI power-management D0, D1, D2, and D3 power states
- Can be configured for any PC/104+ "slot"
- Available in IDAN and FENIX enclosures
- -40 to +85C operating temperature (1 channel version)
- 0 to +70C operating temperature (2 channel version)

5) Picasso1394 of ARVOO Netherlands

The picassoTM 1394 series consists of two models: PCI-1394-fi and 104-1394. Both models have digital IEEE-1394 ports to interface with FireWire cameras. The PCI model additionally offers a fiber port, which connects to an optic*link*TM 1394 unit. Windows 2000 (SP3 or higher), Windows XP and Linux support these boards without additional drivers, because the boards are compliant to the Open Host Controller Interface (OHCI).

Key features

- digital IEEE-1394 (FireWire) interface
- compliant with the Open Host Controller Interface (OHCI)
- two models available: PCI-1394-fi for standard PCI and 104-1394 for PC/104 *plus*
- three IEEE 1394 ports on the PC/104 *plus* model
- two IEEE 1394 ports and a fiber port for long haul FireWire transmission with optic*link*TM 1394 on the PCI model
- supports up to 62 IEEE-1394 devices in a tree connection topology up to 400 Mbit/s
- PCI-1394-fi is connected to an ARVOO optic*link*TM 1394 interface unit
- power for IEEE 1394 drawn directly from host power supply

6) EM104P-1394 of ARBOR Taiwan

Features

- Compatible IEEE-1394a-2000 Fire Wire Standard
- Open Host Controller Interface (OHCI) compliant
- Three high speed Serial Interface
- Speeds of 100/200/400MB its/sec
- Standard 6-way IEEE-1394 Connectors for self-powered or non powered devices

7) Firewire Card of Embedded Designs Plus

Specifications

Supply Voltages & Current
- 5 VDC
20mA (no firewire device connected)
150mA (firewire device connected)
- 12 VDC
0mA (no firewire device connected or self powered firewire device connected)
1.5A (max w/firewire device connected)
BIOS
- Compatible with most PC BIOS's
- Year 2000 compliant
PC/104+
- 16 Bit PC104 (ISA) Interface (pass-thru)
- 32 Bit PC104+ (PCI) Interface
Physical Dimensions
- 3.55"(L) x 3.75"(W)

Firewire Ports
- P1 and P2 Directed External On The Board
- P4 Redirected Internal On The Board
- *P3 Directed External On The Board - Special Order*
Operating System Compatible
- Windows 98SE, 2000, XP
- Linux & Others
PCI Interrupt Addressing
- 0,1,2 or 3 Jumper Selectable
Environmental Information
- Operating temperatures: 0~70&deg;C
- Storage temperatures: -20~80&deg;C
- Relative humidity: 10~90% non-condensing
IEEE 1394 Chipset
- Texas Instruments TSB12LV26
- Texas Instruments TSB41AB3

8) Decision Table

Table 15 Decision Table for Firewire board

Company / Location	*Order Info*	Number of Ports	Connector Type	Link Layer Controller	Host Interface Standard	Galvanic Isolation	Power Supply to bus	External Power Needed	Price	Linux driver offered	Rank
Mindready / Northern Ireland	SedNet 1394 a&b OHCI Board	3	1394a 6-pin	TI TSB12LV26	OHCI v1.0	Yes	No	Yes	$520	Upon request	**4**
Advanced Micro Peripherals Ltd / England	FireSpeed2000	3	1394a	VIA ***	OHCI Compliant			Yes			
Ampro Computers, Inc / USA	MiniModule 1394	2	1394a	VIAVT6307L	OHCI v1.0 and 1.1			Yes	€ 119.66	No	**1**
RTD / Finland	CM17208HR	2	1394a	TI TSB43A21	OHCI Compliant			Yes	€ 271.00	No	**3**
ARVOO / Netherlands	Picasso 1394	3	1394a-6pin		OHCI Compliant				€ 690	No	
ARBOR / Taiwan	EM 104-1394	3	1394a-6pin		OHCI Compliant		Yes	Yes			
Embedded Designs Plus	PC104 FireWire / IEEE1394 Card	2	1394a	TI TSB12LV26	OHCI Compliant			Yes	$249.95	No	**2**

References:

Don Anderson (1999), *Firewire System Architecture: IEEE1394A 2nd Edition*, Addison-Wesley

1394OHCI Specification (2000), *1394 Open Host Controller Interface Specification, Release 1.1*, http://developer.intel.com/technology/1394/download/ohci_11.htm

1394Automation (2003), *Homepage of "1394 Automation"*,
http://www.1394automation.org/

Nyquist (2003), *Homepage of "Nyquist"*,
http://www.nyquist.com/

Eric Buit (2004), *"Real-time network performance characterization"*,
Control Laboratory, University of Twente

RTAI (2004), *Homepage of RTAI*, http://www.aero.polimi.it/~rtai/

Linux1394 (2004), *Homepage of Linux1394*, http://www.linux1394.org/

Libraw1394 (2004), *Manual of Libraw1394*,
http://www.linux1394.org/doc/libraw1394/book1.html

Manfred Weihs (2004), *Homepage of Manfred Weihs*,
http://www.ict.tuwien.ac.at/ieee1394/opensource.html

Peter Soetens (2004), *Porting your C++ GNU/Linux application to RTAI/LXRT*
http://people.mech.kuleuven.ac.be/~psoetens/portingtolxrt.html

IEEE1355 (1998), *Homepage of IEEE1355*, http://grouper.ieee.org/groups/1355/

Spacewire (2004), *Homepage of Spacewire*, http://www.estec.esa.nl/tech/spacewire/

4Links (2004), *Homepage of 4Links*, http://www.4links.co.uk/