University of Twente

EEMCS / Electrical Engineering
Control Engineering

Real-Time Network for Distributed Control

Yuchen Zhang

M.Sc. Thesis

Supervisors
prof.dr.ir. J. van Amerongen
dr.ir. J.F. Broenink
dipl.ing. B. Orlic
Ir. P.M. Visser

August 2005

Report nr. 031CE2005
Control Engineering
EE-Math-CS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Summary

Nowadays, complex control systems, e.g. for industrial automation, are evolving from centralized architectures to distributed architectures. To design a distributed control system, a critical issue is to lay out a hard real-time communication infrastructure. To this end, two kinds of solutions can be categorized from contemporary approaches: the hardware-based solution and the software-based solution. Compared with the hardware-based solution, the software-based solution is generally more cost-effective, adaptable and extendable. Therefore it is more widely applied, especially in laboratory.

FireWire is a high performance serial bus for connecting heterogeneous devices. Though firstly targeted for consumer-electronic applications, many of FireWire's features make it well fit in industrial and laboratorial context. In this MSc assignment, following the general principles of the software-based solution, the Real-Time FireWire Subsystem (RT-FireWire) in Linux/RTAI has been designed and implemented. RT-FireWire provides a customizable and extensible framework for hard real-time communication over FireWire. Via performance benchmarking, it has been shown that the transaction latency on RT-FireWire is limited to a certain range that is usable for distributed control applications, whether the system is under heavy load or not.

Ethernet Emulation over FireWire (Eth1394) has been implemented on RT-FireWire as a highlevel module in the application layer. Via Eth1394, RT-FireWire can be connected to another real-time software framework RTnet, which implements real-time networking on the IP layer. Therefore, FireWire has been introduced as a new medium alternative to Ethernet for real-time IP networking. The benchmarking on Eth1394 and Ethernet shows that the real-time performance of both is comparable.

The real-time networking support provided by RT-FireWire has been integrated to a toolchain for controller design and verification. The toolchain is developed at Control Engineering Group of University of Twente. By using this toolchain with the newly added networking support, a controller that has been designed and verified in simulation can now be easily deployed into multiple nodes. For demonstration, a simple but real-life distributed control system has been built by using this toolchain and FireWire. The measurement results on that system proofs that, FireWire, with RT-FireWire steering on it, can be used as a fieldbus for a distributed control application.

The development on RT-FireWire can be continued in several directions: a new interface can be developed to directly operate on RT-FireWire layer; new middleware application protocols (e.g. CANopen) can be investigated to see if they can be stacked on the basic real-time services provided by RT-FireWire; real-time vision control over FireWire is another interesting topic that has not been fully opened.

Preface

With this report, I finished my MSc study at University of Twente. I would like to give my thanks to all the people who helped me during these two years, especially during my thesis work in last 11 months.

I would like to thank Jan Broenink, Bojan Orlic and Peter Visser for their supervising on my work. Also, I would like to thank Marcel Groothuis for his help and suggestions to my work.

I have my special thanks to the Open Source community, especially to Jan Kiszka, the project leader of RTnet. Thanks for all his explanations about RTnet, and the wonderful discussions that I had with him via Email.

I would like to thank my parents also. Without their support, it would not be possible for me to study abroad.

Last but not least, I would like to thank all my friends in the Bible Study group. Thanks for their prayers.

Zhang Yuchen

Enschede August 29, 2005

Table of Contents

1 Introduction

1.1 Background

1.1.1 Real-Time Computer System

A real-time computer system is a computer system in which the correctness of the system behavior depends not only on the logical results of the computations, but also on the physical instant at which these results are produced.[Kopetz, 1997] A real-time computer system often co-exists with the other two subsystems, as shown in Figure 1-1.

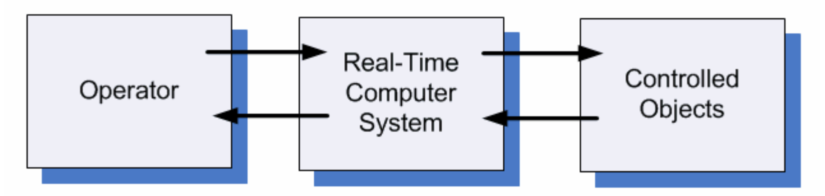


Figure 1-1 Real-Time Computer System and its Workaround

A real-time computer system must react to the stimuli from the controlled object (or the operator) within a time interval. The instant at which a result must be produced is called a deadline. If a catastrophe could result if a deadline is missed, the deadline is called hard. Otherwise it is soft. A real-time computer system that must meet at least one hard deadline is called a hard real-time computer system, or a safety-critical real-time computer system. If no hard deadline exists, then the system is called a soft real-time computer system.

The design of a hard real-time computer system is fundamentally different from the design of a soft real-time computer system. While a hard real-time computer system must sustain a guaranteed temporal behavior even under peak system load and any possible fault conditions, it is permissible for a soft real-time computer system to miss a deadline occasionally.

1.1.2 Centralized Architecture vs. Distributed Architecture

The architecture of real-time computer system can be centralized or distributed. A distributed real-time computer system consists of a set of nodes and a communication network that interconnects these nodes. Compared with the centralized architecture, the distributed architecture appears as a more preferable alternative for the implementation of hard real-time system. Several arguments are:

- In many engineering disciplines, large systems are built by integrating a set of well-specified and tested subsystems. It is important that properties that have been established on the subsystem level are maintained during the system integration. In the distributed architecture, such a constructive approach is much better supported, compared with centralized architecture.
- Almost all large systems evolve over an extended period of time, e.g. some years or decades. Therefore a scalable and extensible system is strongly desired. To deploy a scalable and extensible system, a distributed architecture is essential to provide the necessary framework since:

  I. Nodes can be added within the given capacity of the communication channel. This introduces additional processing power to the system.

II. If the processing power of a node has reached its limit, a node can be transformed into a gateway node to open a way to a new cluster. The interface between the original cluster and the gateway node can remain unchanged (Figure 1-2).

- Most of the critical real-time systems demand fault-containment or fault-tolerance, which means the system should continue functioning despite the occurrence of faults. To this end, only the distributed architecture gives the possibility to implement fault-containment or fault-tolerance via distributing the system functions to different nodes or replicating the function of a certain node to another.

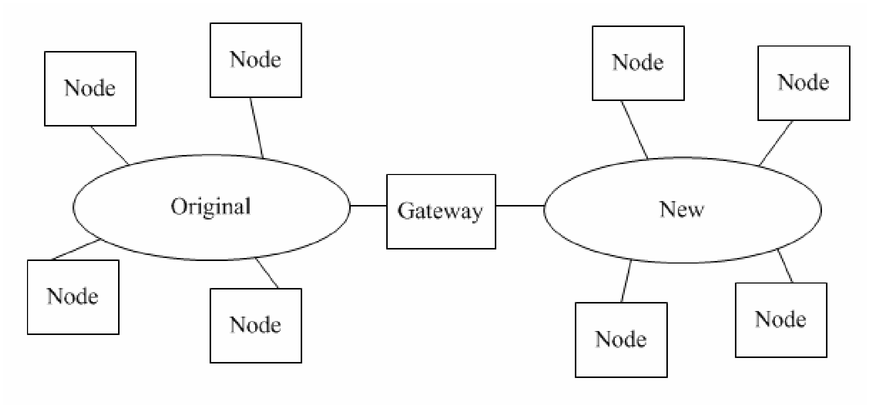


Figure 1-2 Transparent expansion of a node into a new cluster

1.1.3 Hard Real-Time Networking in the Distributed Architecture

To deploy a real-time computer system with a distributed architecture, one important issue is to lay out a hard real-time communication infrastructure, so-called fieldbus. To this end, two kinds of solutions can be categorized from contemporary design approaches.

- To use specifically adapted or designed hardware components to deploy a hard real-time network. These components may be real-time switches, network adapters with high innate intelligence or even fundamentally revised network controllers. By using these hardware components, a hard real-time system can be built. However, since this solution is fully implemented in hardware, a lot of effort and investments is needed. Moreover, the adapted or newly designed hardware can not be easily changed or extended.

- Instead of using hardware-based solution, the more flexible software-based solution can be chosen. In this solution, the standard, relatively cheap hardware components can be chosen, e.g. Ethernet, USB, and FireWire. Above these hardware components, a real-time, deterministic software stack (e.g. real-time operating system, real-time implementation of the network protocol stack, etc) should be built, which can steer the hardware to meet the real-time behavior requirements. The strength of this software-based solution is that, it does not need too much effort and investment for design and implementation, and one solution can be easily adapted for another problem or moved to another platform.

1.2 Research Context

At Control Engineering group of the University of Twente, one of the research directions is embedded control system. Along this direction, several topics are mainly focused on: design methodology for embedded control software; CSP-based concurrent programming; fieldbus-connected embedded control systems and hardware-in-the-loop simulation for embedded control system, etc.

Narrowed down to the research on distributed control systems, the main work is leaded by two PhD projects:

- CSP-channels for field-bus interconnected embedded control systems. It deals with hard real-time control using several co–operating processors in networked environments. The network itself is embodied by an industrial field bus, which are investigated with respect to real-time performance. During the work by previous students, CAN [Ferdinando, 2004], USB, Ethernet [Buit, 2004], FireWire [Zhang, 2004] and Profibus [Huang, 2005] has been investigated with respect to their suitability for use in real-time context.
- Boderc(Beyond the Ordinary: Design of Embedded Real-time Control): Multi-agents and CSP in Embedded Systems. In this project, a hardware-in-the-loop setup has been built by [Groothuis, 2004] to test distributed controllers with simulation model of various plants. In this setup, the communication channel between controllers is deployed on CAN.

1.3 Assignment

Following the second approach in 1.1.3, the objective of this MSc assignment is to adopt a standard, relatively cheap networking hardware component for deploying the hard real-time network in distributed control systems. Around the main goal, challenges exist on several aspects:

- The existing software on that hardware should be adjusted or even re-designed, so the hardware can be steered to behave in a deterministic way.
- The adjusted or re-designed software should be easily adaptable and extensible.
- Resource-constraint situation should be taken into account, like system with in-adequate memory.
- The adjusted or re-designed software should provide a friendly interface, which eases the development of applications (e.g. controllers) on it.

1.4 Initial Decisions

1.4.1 FireWire

FireWire, also known as IEEE 1394, is a high performance serial bus for connecting heterogeneous devices. Though firstly targeted for consumer-electronic applications, such as high-speed video transmission, many of FireWire’s features make it well fit industrial and laboratorial context. In this assignment, FireWire is chosen to be the implementation target of hard real-time networking. The direct significance after achieving this is adoption of FireWire as a new generation fieldbus, which comes with much higher performance than other existing alternatives (e.g CAN, Profibus).

1.4.2 Linux

Linux, an Open Source operating system kernel, is well known for its open structure, modular design and easy adaptability. In this assignment, Linux is chosen to be the Operating System kernel. Thereby, the FireWire Subsystem in Linux is taken as the starting point for investigation and implementation.

1.5 Report Outline

Chapter 2 firstly gives a detailed description about FireWire, including its characteristic on various aspects, e.g. bus topology, data transfer modes, etc. Secondly, the FireWire subsystem in Linux is described and the measurement results concerning its suitability for use in real-time is presented.

Chapter 3 presents the implementation of the Real-Time FireWire Subsystem (RT-FireWire), including the architecture, core components and protocol adaptation. Secondly, the measurement results concerning RT-FireWire's suitability for use in real-time is given, and compared with the results on the Linux FireWire Subsystem.

Chapter 4 presents the implementation of deploying real-time IP network over RT-FireWire. Secondly, the results of performance measurement on IP over FireWire is given, and compared with the performance of IP over Ethernet.

Chapter 5 presents the integration of RT-FireWire's networking support into a complete toolchain for design and verification of controllers. Based on the integration, a demonstration of using this toolchain for deploying a simple but real-life distributed control system is shown. The result of the demonstration is also presented.

In Chapter 6 the conclusions and recommendations for this project is given.

2 Introduction to FireWire and Its Subsystem in Linux

2.1 Introduction

This chapter first starts with a detailed description of FireWire, including the overview of bus topology, data transfer modes, the layered protocol structure, and a literature research concerning the protocol overhead and the transmission timing on FireWire. Then, the pointer goes to FireWire subsystem in Linux: the software architecture is introduced and the test bench to measure the suitability of using this subsystem in real-time is presented. Based on the measurement, the conclusion about whether FireWire subsystem in Linux is suitable for use in real-time is reached.

2.2 Overview of FireWire

2.2.1 Bus Topology

The IEEE 1394 specification defines the serial bus architecture known as FireWire. Originated by Apple Computer [Apple], FireWire is based on the internationally adopted ISO/IEC 13213 specification [IEEE, 1994]. This specification, formally named "Information technology - Microprocessor systems - Control and Status Registers (CSR) Architecture for microcomputer buses," defines a common set of core features that can be implemented by a variety of buses. IEEE 1394 defines serial bus specific extensions to the CSR Architecture.

The bus topology of FireWire is tree-like, i.e. non-cyclic network with branch and leaf nodes, for typical topology see Figure 2-1

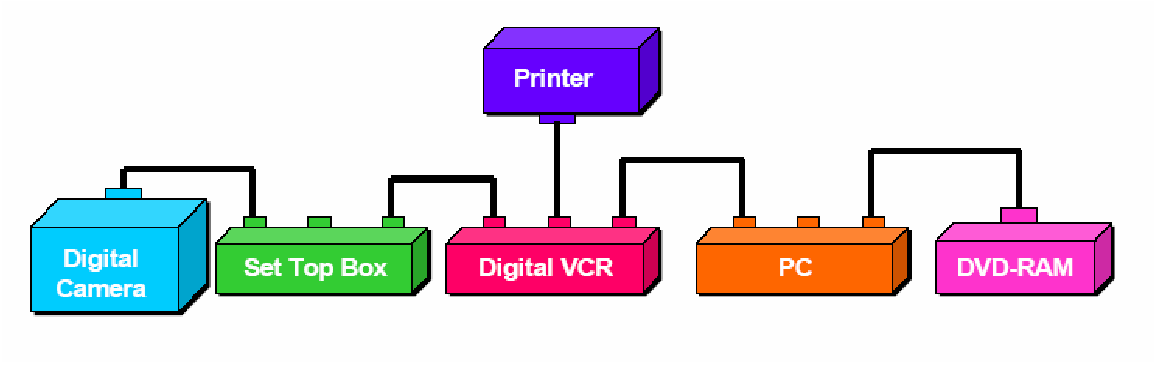


Figure 2-1 Example FireWire Network

Configuration of the bus occurs automatically whenever a new node is plugged in. It proceeds from leaf nodes (those with only one other node attached to them) up through the branch nodes. A bus that has three or more nodes attached will typically, but not always, have a branch node become the root node (e.g. Digital VCR in Figure 2-1).

Unlike most other serial buses designed to support peripheral nodes (e.g. Universal Serial Bus), FireWire is a peer-to-peer network with point-to-point signaling environment, so that any two nodes can exchange data without intervention from a third node. This important advantage allows FireWire to be used as fieldbus in distributed control, since direct data transfer between any two computing nodes is a definitely desired property in distributed control networks.

2.2.2 Data Transfer Modes

For data transfer on FireWire, two different types of packets are used: asynchronous packets for reliable, receipt-confirmed data, and isochronous packets for time-critical, unconfirmed data. A mix of isochronous and asynchronous transactions is performed across the serial bus by sharing the overall bus bandwidth. Notice that the bus bandwidth allocation is based on 125µs intervals, so called the FireWire transaction cycle, as shown in Figure 2-2.

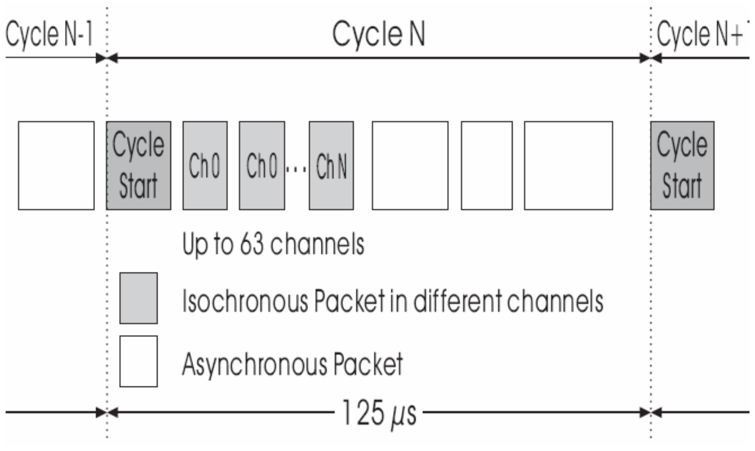


Figure 2-2 FireWire Transaction Cycle

The isochronous transfer mode is particularly suitable for the transmission of time-critical data in real time, e.g. for video or audio. It guarantees a firm bandwidth and sends packets in a fixed clock pulse (every 125µs). The packets are not addressed to individual nodes but are separately marked by a channel number. Because late data are unusable for time-critical applications, no acknowledgment of receipt is sent and incorrect packets are not resent. Asynchronous packets are sent peer-to-peer from one node to one or all other nodes. In the packet header the address of the destination node or nodes is indicated, as is the memory address, to which the data in the packet refer. With receipt of an asynchronous packet, an acknowledgment of the receiver node is sent as proof that the packet arrived. The speed of data transmission and associated maximum packet size of asynchronous and isochronous packets are listed in Table 2-1.

Cable Speed	Maximum Size (Byte) of Asynchronous Packet	Maximum Size (Byte) of Isochronous Packet
100Mb/s	512	1024
200Mb/s	1024	2048
400Mb/s	2048	4096

Table 2-1 Transmission Speed and Packet Size on FireWire

In asynchronous transfer mode, the FireWire bus appears as a large distributed memory space with each node hosting a 48-bit mapped address space (256 Terabytes). In addition, each bus is identified by 10-bit mapped id; hence a maximum of 1024 FireWire buses can be connected in single network. Every node on the bus is identified by 6-bit mapped id - hence a maximum of 64 nodes per bus. This gives a 64-bit mapped address, to support 16-Exabytes in total. The illustration is given in Figure 2-3. See [Anderson, 1999] for more a detailed description.

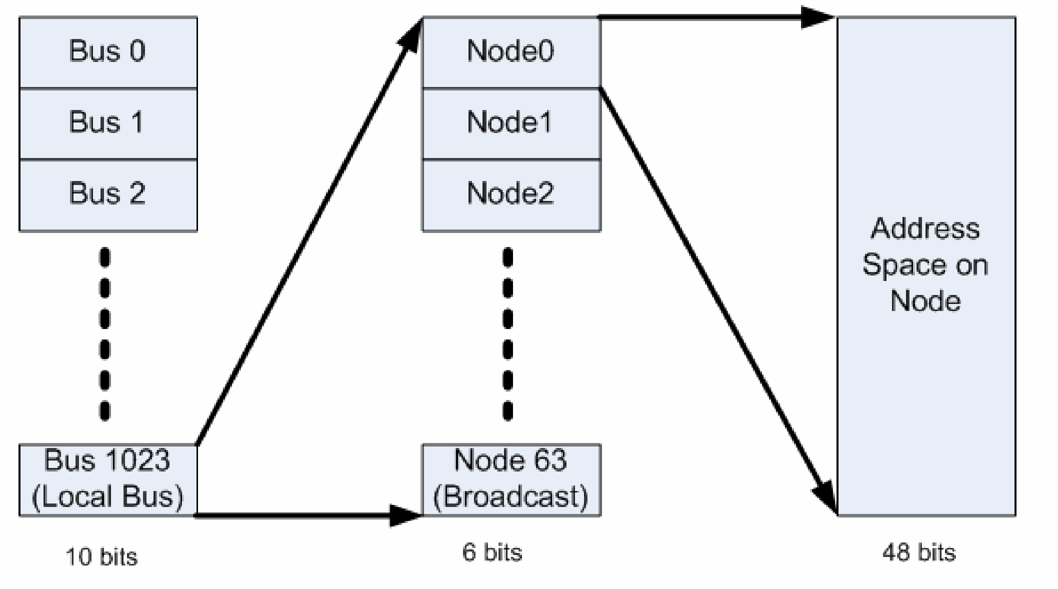


Figure 2-3 Address Space on FireWire

2.3 FireWire Protocol Layers

Four protocol layers are defined in FireWire, in order to separate its complexity in the several levels of abstraction and hence simplify the implementation of hardware and software. Each layer has associated set of services defined to fulfill its role, e.g. to support certain part of data transfer transactions and bus management, as shown in Figure 2-4.

2.3.1 Physical Layer

The Physical Layer is the hardware used to bridge between a local FireWire node and the whole network. This Layer has the following tasks:

- defines connectors and transmission medium
- performs bus initialization (configuration) after each Bus Reset
- manage the possession of the bus (bus arbitration)
- performs data synchronization
- performs coding and decoding of data messages
- determine signal level

On the Physical Layer, three different situations can result:

- The Physical Layer of a node receives a packet that is targeted to another node. In this case, the packet is passed further over all ports, except the one from which it was received.
- The Physical Layer of a node receives a packet that is targeted to this node itself. This packet is passed to the Link Layer. The Link Layer then passes it on to the Transaction Layer (in the case of an asynchronous transmission) or directly to the Application (in the

case of an isochronous transmission).

- The sending packet is issued from the Link Layer of local node. In this case the packet is passed on over all ports.

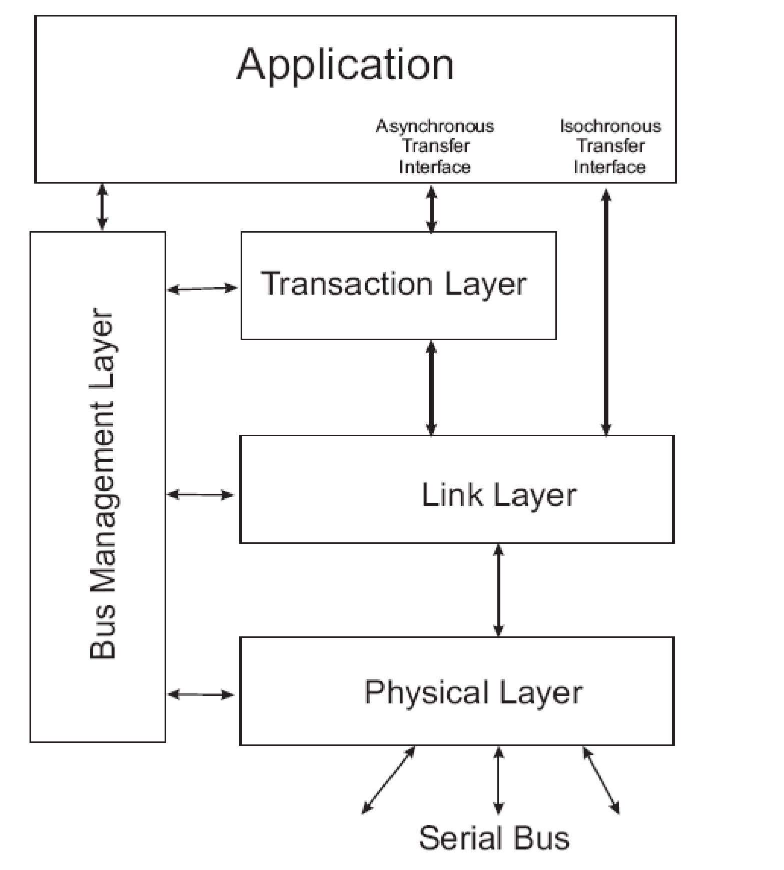


Figure 2-4 Structure of the 4-layer Model

2.3.2 Link Layer

The Link Layer is located between the Physical Layer and the Transaction Layer. It performs tasks related to sending and receiving asynchronous and isochronous packets.

For a received packet, the Link Layer is responsible for checking received CRCs to detect any transmission failure; for packet to be sent, it is responsible for calculating and appending the CRC to the packet. The Link Layer examines the header information of the incoming packet and determines the type of transaction that is in progress. For asynchronous transaction, the data packet is then passed up to the transaction layer. For isochronous transaction, the transaction layer is not used and therefore the Link Layer is directly responsible for communicating isochronous data to application.

2.3.3 Transaction Layer

The Transaction Layer is only responsible for the asynchronous operations Read, Write, and Lock. By means of these operations the access of the memory area (Figure 2-3) is possible.

If two nodes communicate with each other, then receipts of the transferred packets are confirmed on the level of their Transaction Layers. The transmission of incorrect packets is repeated or discarded. Depending upon the extent of the message, the Transaction Layer divides the transmission actions into individual sub-actions and handles these independently.

For these tasks as well as for the bus access management (bus arbitration) and the data synchronization, the Transaction Layer uses the following services of the Link Layer:

- Request Service (request to start a transfer)
- Indication Service (acknowledgment to the request)
- Response Service (response to the request)
- Confirmation Service (acknowledgment to the response)

2.3.4 Bus Management Layer

Each node has a Bus Management Layer which controls the bus functions in the different layers. Beyond that, the Management Layer makes a multitude of functions available concerned with the management of the power supply and the bus configuration. The actual functionality depends on the abilities of the nodes involved. However, the functions for automatic configuration must be present for all nodes.

The Bus Management is responsible for a set of tasks:

- assigning channel numbers and bandwidth allocation for isochronous transfers
- guaranteeing that, the nodes that get their power supply via the bus cable have sufficient power available
- adaptation of certain timing settings depending on the bus topology to increase the data flow-rate over the network
- supporting services, that allow other nodes to request information about topology and speed conditions

It is not necessary that all specified tasks are assigned to only one node. Rather these tasks are summarized in three global roles and during the configuration phase, efficiently divided among the attached nodes. Depending on the supported level of bus management functionality, three states based on presence/absence of the three corresponding roles are differentiated:

- "Non Managed"

  A non-managed bus possesses only one "Cycle-Master" and fulfills the minimum management requirements of an IEEE 1394-Bus. In each FireWire transaction cycle, the "Cycle Master" initiates the start of the bus cycle by sending cycle start message.

- "Limited Managed Bus with Isochronous Resources Manager"

  Such a bus contains an "Isochronous Resources Manager" (IRM) in addition to the "Cycle Master". The bandwidth allocation on the bus can get managed by the IRM.

- "Fully Managed Bus"

  The "Fully Managed Bus" represents a fully functional bus that, in addition to “Cycle Master” and IRM, contains the "Bus Manager". It is able to optimize the bus and possesses unrestricted "Power Management". The "Bus Manager" is able to collect information about the bus topology ("Topological Map") and the transmission rates between any two nodes ("Speed Map"). In this way the maximum data transmission rate can be determined for each cable distance and the bus can be optimized.

2.4 Protocol Overhead and Transmission Timing

This section, one step deeper is taken to analyze the protocol overhead introduced by FireWire’s packet structure and to determine the transmission timing on FireWire.

2.4.1 Asynchronous Transaction

Three different asynchronous transactions are used:

- Read
- Write
- Lock

With the Read operation, data will be read from the memory area of a node. With the Write operation, data can be written into the memory area of a node. The Lock operation is a mechanism, which allows/disallows a "protected" operation[Anderson, 1999].

An asynchronous packet consists of header and data, see Figure 2-5 for write request packet format and Figure 2-6 for the response. See Table 2-2 for description of each component.

As can be seen from above, the protocol overhead in FireWire asynchronous write request is 24bytes, i.e. 24 extra bytes needs to be transferred along with the application data. Besides, the asynchronous write response is 16byte. Both request and response are followed by an acknowledgement, which is short packet of 4 bytes. Therefore, a simple formula for the protocol efficiency is:

$$E_{asyn} = \frac{DataSize(byte)}{DataSize(byte) + 24 + 16 + 8} \times 100\%$$

Figure 2-7 present an example of asynchronous write transaction between two nodes. If node A wants to write data into a certain memory area of node B, it sends a write request to node B. Node B acknowledges the receipt of this request. The acknowledgement indicates only the receipt of the request, not yet the execution.

After node B has written the data into that memory area, it sends a response to node A. In this response, node A gets the message that the data has been submitted into the memory area of node B. This is the acknowledgement of execution. Node A acknowledges the receipt of this response, whereby the asynchronous transaction is finished.

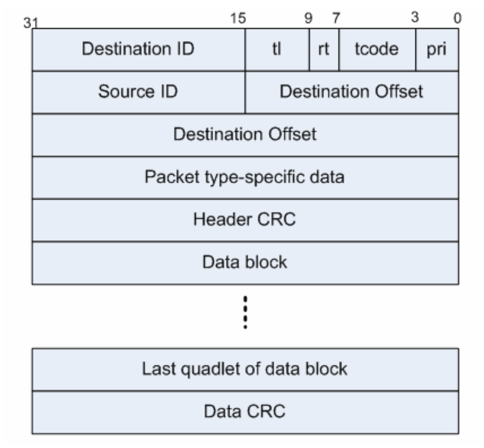


Figure 2-5 Asynchronous Write Request Packet

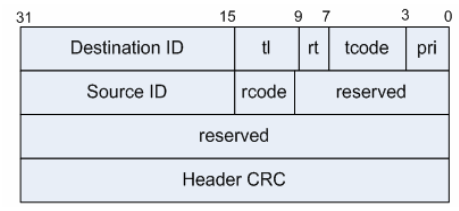


Figure 2-6 Asynchronous Write Response Packet

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. Only if the response packet contains a correct transaction label, it is possible to find the corresponding request packet.
RT	Retry Code that defines whether this is a retry transaction.
TCODE	Transaction Code defines the type of transaction (Read request, Read response, Acknowledgement, 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 header part
Optional Data	Quadlet aligned data specific to the type of the packet
Optional Data CRC	CRC for the Optional Data
Rcode	Response Code, specifying the result of this transaction.

Table 2-2 Components in an Asynchronous Packet

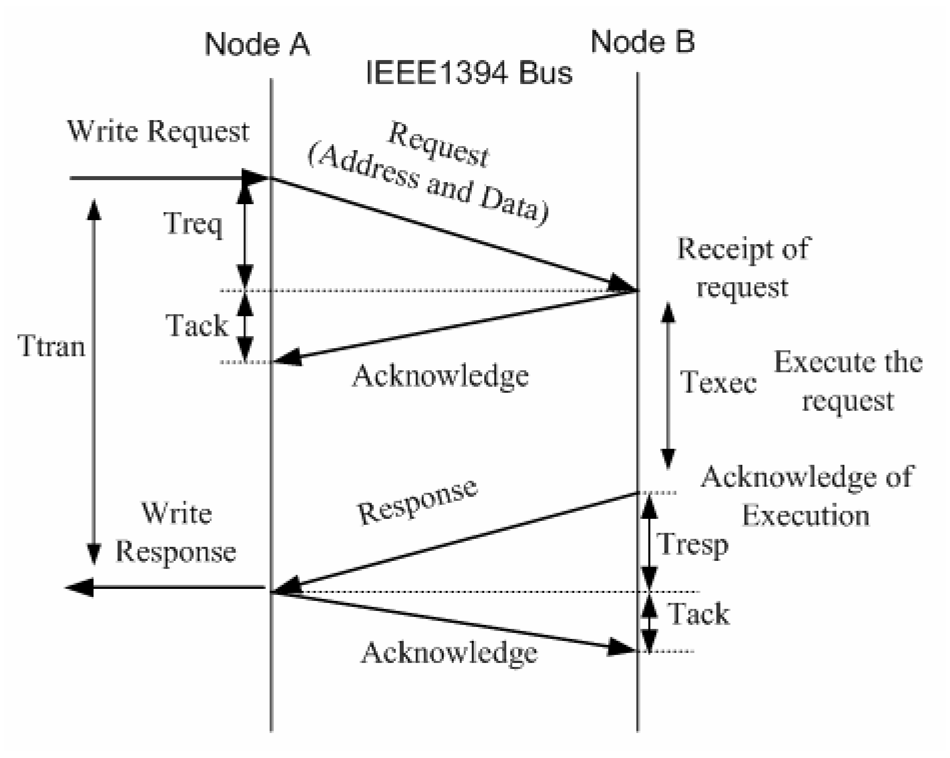


Figure 2-7 Asynchronous Transaction between Two Nodes

The timing of asynchronous transmission is shown in Figure 2-7.

$$T_{req} = \frac{(DataSize + 24) \times 8bits/byte}{400Mb/s}$$

$$T_{resp} = \frac{16bytes \times 8bits/byte}{400Mb/s} = 0.32\mu s$$

$$T_{ack} = \frac{4bytes \times 8bits/byte}{400Mb/s} = 0.08\mu s$$

$$T_{ack} = \frac{4bytes \times 8bits/byte}{400Mb/s} = 0.08\mu s$$

So the latency during one write transaction is the sum of the time for transferring the request, executing the request and transferring the response. Due to the relatively small value of the time of transferring the response, it can be omitted. Assuming we write data of 4, 56, 2048 bytes payload, the latency will be:

$$T_4 = 0.88\mu s + T_{exec}$$
$$T_{56} = 1.92\mu s + T_{exec}$$
$$T_{2048} = 41.76\mu us + T_{exec}$$

2.4.2 Isochronous Transaction

Compared with asynchronous transaction, the packet of isochronous transaction is relatively simpler, which is illustrated in Figure 2-8 and explained in Table 2-3.

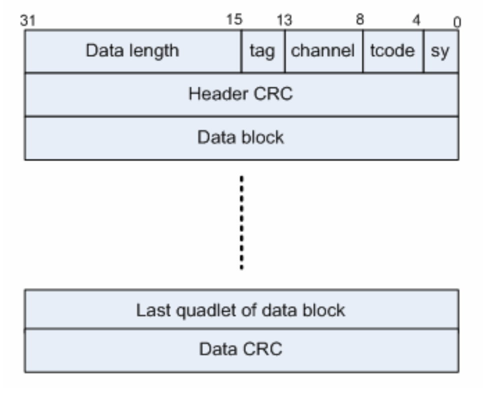


Figure 2-8 Isochronous Packet

Figure 2-9 gives an example of isochronous transaction between two nodes. Here node A is sending data on isochronous channel N to node B. No acknowledgment or response is generated from Node B. But the maximum sending rate is limited to 125µs, due to the cycled bandwidth allocation on FireWire.

Name	**Description**
Data Length	Data length, can be any value between zero and FFFFh
Tag	Isochronous Data format tag
Channel	Isochronous Channel Number
Tcode	The transaction code for isochronous data block is Ah
Sy	Synchronous Code, application specific

Table 2-3 Isochronous Packet Components

As can be seen from above, the protocol overhead in FireWire isochronous packet is 12bytes, i.e. 12 extra bytes needs to be transferred along with the data load. To calculate the protocol efficiency on isochronous transaction, a formula can be deducted:

$$E_{iso} = \frac{DataSize(bytes)}{DataSize(bytes)+12} \times 100\%$$

The latency for one way isochronous transmission is (assuming the bus speed is 400Mb/s):

$$T_{iso} = \frac{(DataSize+12)\times 8bits/byte}{400Mb/s}$$

So for data payload of 4, 56, and 2048 bytes, the latency will be:

$$T_4 = 0.32\mu s$$
$$T_{56} = 1.36\mu s$$
$$T_{2048} = 41.2\mu s$$

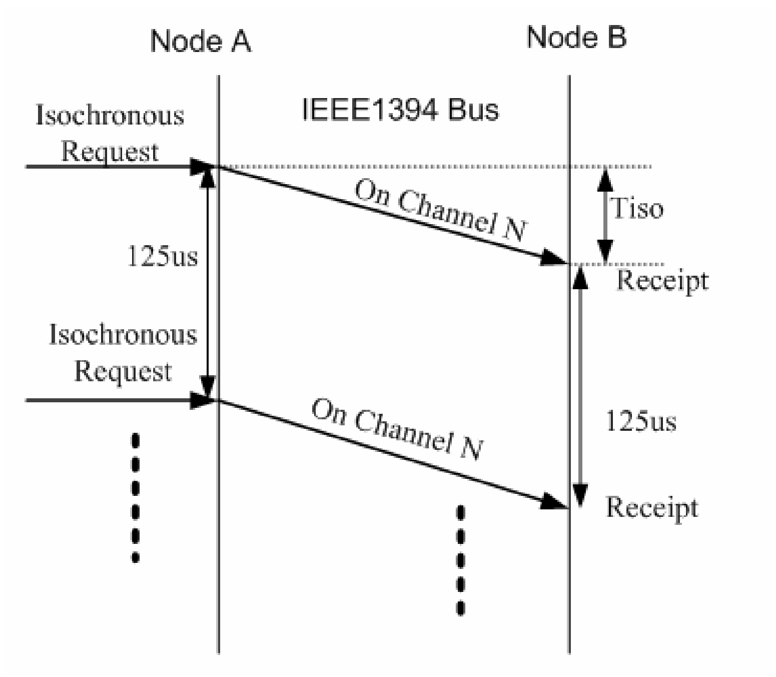


Figure 2-9 Example Isochronous Transaction

2.5 Linux FireWire Subsystem

2.5.1 Introduction

In this section, the overview of FireWire subsystem in Linux is presented, and the limitation to use it in real-time context is revealed through basic testing experiments.

2.5.2 System Overview

The overview of FireWire subsystem in Linux is given in Figure 2-10. It consists of FireWire subsystem kernel, adapter drivers and highlevel modules. Note that, the whole subsystem works in deep cooperation with the Linux kernel core, but it is beyond the scope of this report to explain relative dependencies and implementation details. Please refer to [Linux1394] for more detailed information.

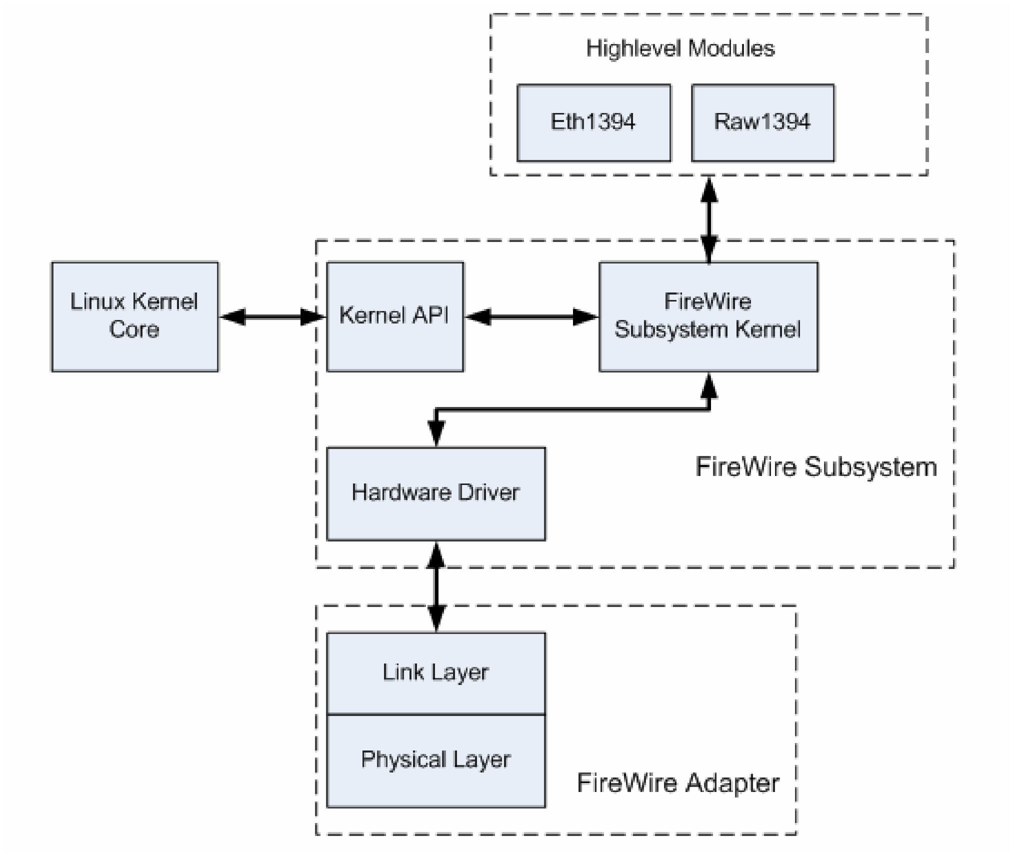


Figure 2-10 Linux FireWire Subsystem Overview

FireWire Subsystem Kernel

More internals of subsystem kernel is revealed in Figure, with explanation following.

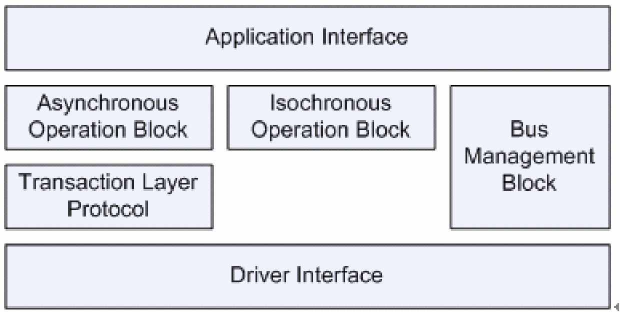

Application Interface
Asynchronous Operation Block
Isochronous Operation Block
Bus Management Block
Transaction Layer Protocol
Driver Interface

Figure 2-11 FireWire Subsystem Kernel

- The Driver Interface block takes care of the management of FireWire adapters (there can be more than one adapter registered to the kernel). Meanwhile it also abstracts out the specifics of various adapter hardware drivers, providing other modules with a common set of services.
- The Transaction Layer Protocol block implements the transaction layer protocol of FireWire.
- The Asynchronous Operation block is responsible for both taking packet send request from applications and dispatching received packets to applications.
- The Isochronous Operation block is responsible for both taking request from applications

to (de)allocate isochronous channel and send packet, as well as for dispatching received packets to applications.

- The Bus Management module is responsible for monitoring the bus status and performing bus management functions as described in 2.3.4.
- The Application Interface module has several functionalities: taking care of the application management, like registering of new application, implementing communication/synchronization between application and kernel and so on. It provides applications with common API that abstracts away from lower level transactions.

FireWire Adapter

The FireWire adapters available in the market are based on one of the following chips:

- aic5800 Adaptec AIC-5800 PCI-IEEE1394
- pcilynx Texas Instruments PCILynx
- Open Host Controller Interface (OHCI1394)

In this project, only adapter of the third type is used, therefore only the corresponding ohci1394 driver is used. See [1394OHCI, 2000] for the specification of OHCI1394.

Highlevel Modules

Highlevel modules in FireWire subsystem are separate functional modules with standardized interfaces connecting to subsystem kernel. Through these interfaces, a certain highlevel module can register itself as being responsible for handling certain events on the bus, e.g. read/write/lock transactions to a certain area of local address space. In another word, the highlevel module can allocate for itself a certain piece of address space on the network.

Here, two highlevel modules are named: eth1394 and raw1394.

Eth1394

Eth1394 stands for Ethernet Emulation over FireWire, all called IPover1394. By using Eth1394, all the applications built on Ethernet network can be directly applied on FireWire, therefore making FireWire a medium alternative for those applications that has been completely developed on Ethernet. See [Johansson, 1999] for IPover1394 protocol specification.

Raw1394

Raw1394 stands for Raw Access over 1394, which is to provide Linux user-space program an interface to directly send and receive packet on FireWire.

2.5.3 Performance Benchmarking on Linux FireWire Subsystem

In [Zhang, 2004], series of experiments were carried out on FireWire, employing Linux user-space programs to measure the latency of transactions on FireWire. But in this project, the Linux kernel in use has been updated to 2.6, which could have new influence on real-time performance. Therefore, new experiments are carried out on Linux FireWire Subsystem system with a 2.6 kernel to study its suitability for use in real-time context.

Test Bench Setup

2 PC104 stacks are employed in this experiment. Detailed information of stack components follows:

- PC104:
  VIA Eden 600 MHz, 256 Mb Memory, 32 Mb flash disk.
- FireWire Adapter:
  PC/104+ board with VIA VT6370L Link & Physical layer chip, supporting 400 Mb/s transferring speed at maximum. (See [Zhang, 2004] for more related information)
- Software in use: Linux kernel 2.6.12.

Experiment Cases

The performance is evaluated in 4 cases: asynchronous transaction without system load, asynchronous transaction with heavy system load, isochronous transaction without system load and isochronous transaction with heavy system load. The experiments on both asynchronous transaction and isochronous transaction are illustrated in Figure 2-18 and Figure. For each case, two nodes are involved in the experiment: one is requesting node that is actively sending the data; another is target node that is passively receiving the data, processing it, and (in asynchronous transaction) send the response back. The data sending rate on client node is 1 KHz. For each case, 100,000 data samples are collected for analyzing. During the experiment, the data load is always 56bytes.

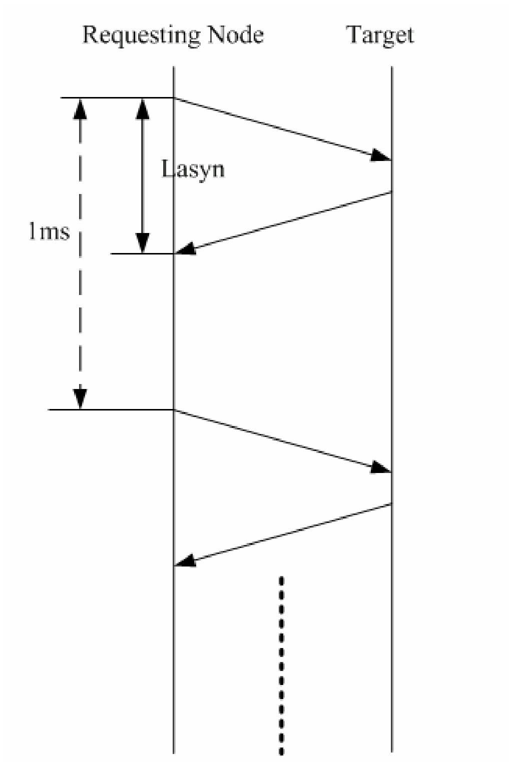


Figure 2-12 Asynchronous Transaction Latency

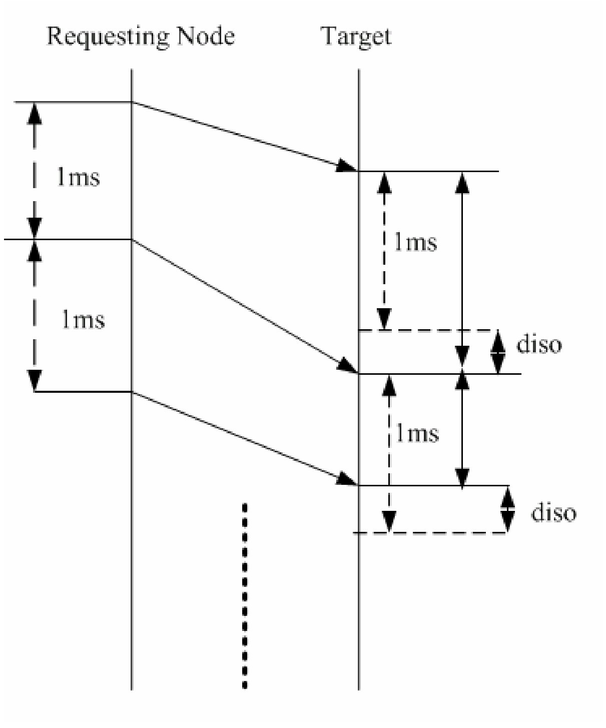


Figure 2-13 Drift of Data Receiving Rate on Isochronous Transaction

Imposing System Load

The put the experiment in an extremely loaded system, extra processing load needs to be imposed explicitly. Three ways of imposing system load are used together in this experiment.

- Creating a flood of interrupts from external world via network by using a third node to send a lot of random data to the nodes in experiment.

- Creating a flood of interrupts from hardware disk I/O by reading the whole hard disk.
- Creating a flood of system calls via Linux command line. This will make a lot of kernel-user space switch.

Measurement Results

The result is presented by using cumulative percentage curves. At any point on the cumulative percentage curve, the cumulative percentage value (y-value) is the percentage of measurements that had a latency less than or equal to the latency value (x-value). The latency at which the cumulative percentage curve reaches 100 percent represents the worst-case latency measured. For real-time transaction latency, the ideal cumulative percentage curve is one that is steep with a minimal decrease in slope as the curve approaches 100 percent.

Therefore, the cumulative percentage at a certain latency value can be interpreted as the probability of the transaction being able to meet real-time constraints when its deadline is assumed to be equal to that latency value. Since the network in concern will be used in distributed real-time control application, the latency can more or less determines the operating frequency of the system. For example, if the cumulative percentage at latency 100μs is 97%, that mean if the system on the network runs at 10 KHz, only 97% of the distributed data (sensor input, actuator output, etc) can be sent or received on time. The cumulative percentage over ascending latency values are shown in Figure 2-14 and Figure 2-15. The former is for the situation when system is not loaded, while the latter is for the situation when system is heavily loaded. Figure 2-16 and Figure 2-17 present the cumulative percentage over ascending drift values of data receiving rate on isochronous transaction, respectively for the situation of system not being loaded and the situation of system being heavily loaded.

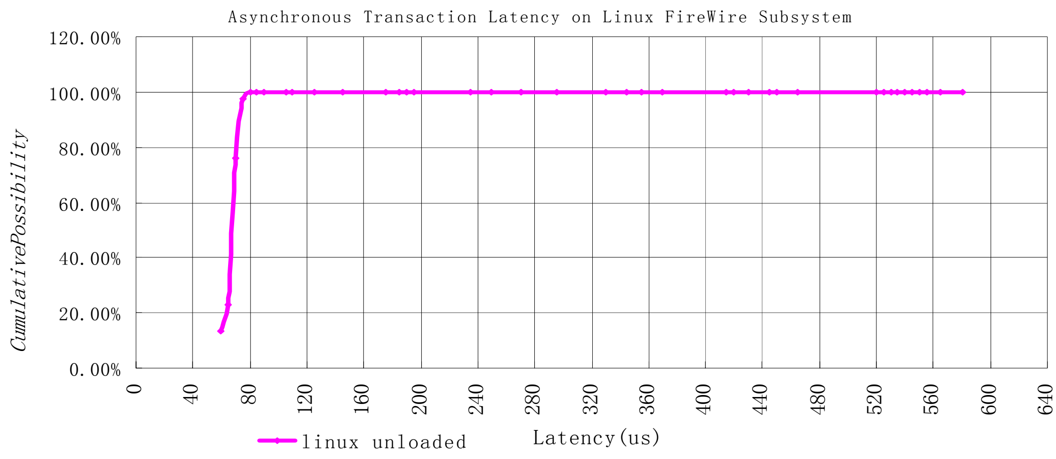


Figure 2-14 Asynchronous Transaction Latency on Linux FireWire Subsystem when system is not loaded

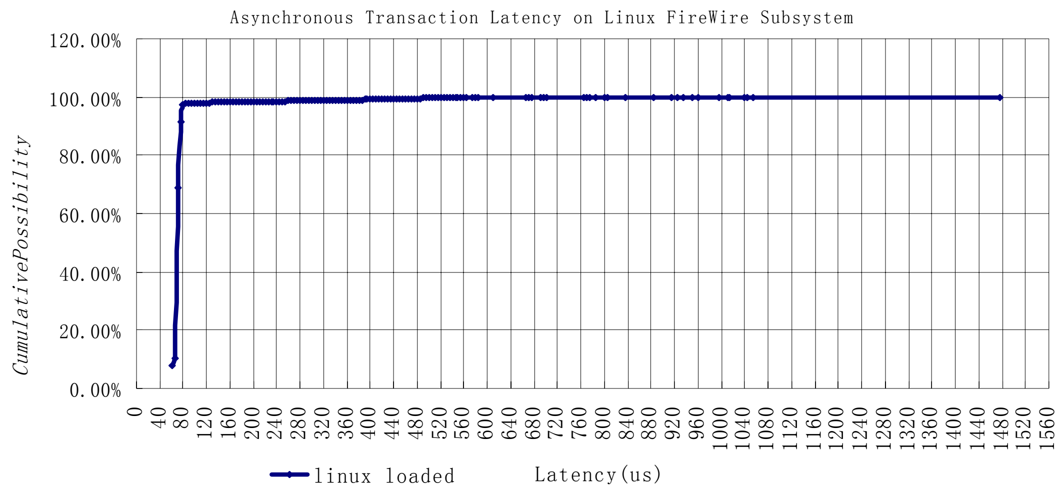


Figure 2-15 Asynchronous Transaction Latency on Linux FireWire Subsystem when system is loaded

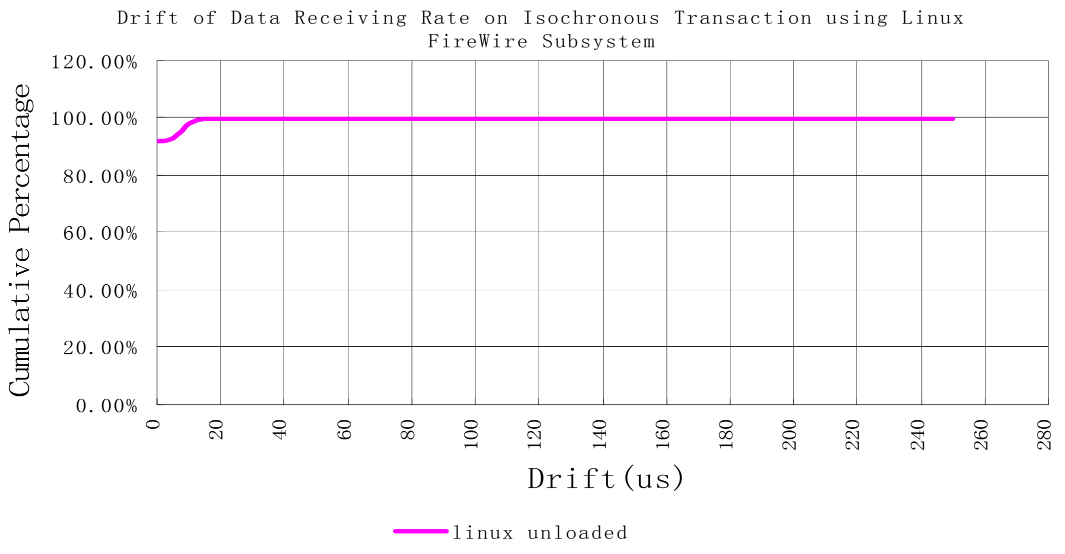


Figure 2-16 Drift of Data Receiving Rate on Isochronous Transaction using Linux FireWire Subsystem when system is not loaded

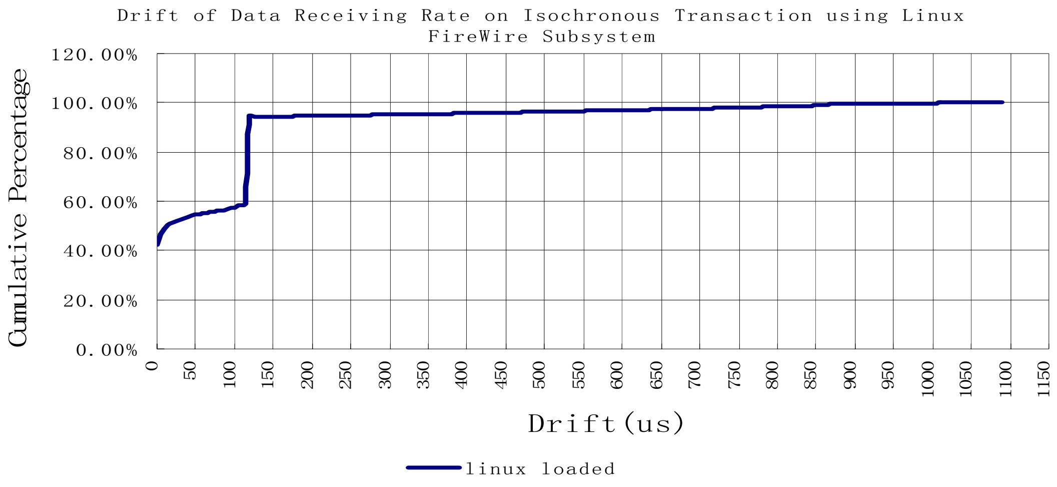


Figure 2-17 Drift of Data Receiving Rate on Isochronous Transaction using Linux FireWire Subsystem when system is loaded

Thereby for hard real-time application, low range of cumulative percentage values does not make any sense (deadline can not be missed that often), so only top of the curve, i.e. at least above 97%, is worth having a closer look, as shown in Figure 2-18 and Figure 2-19.

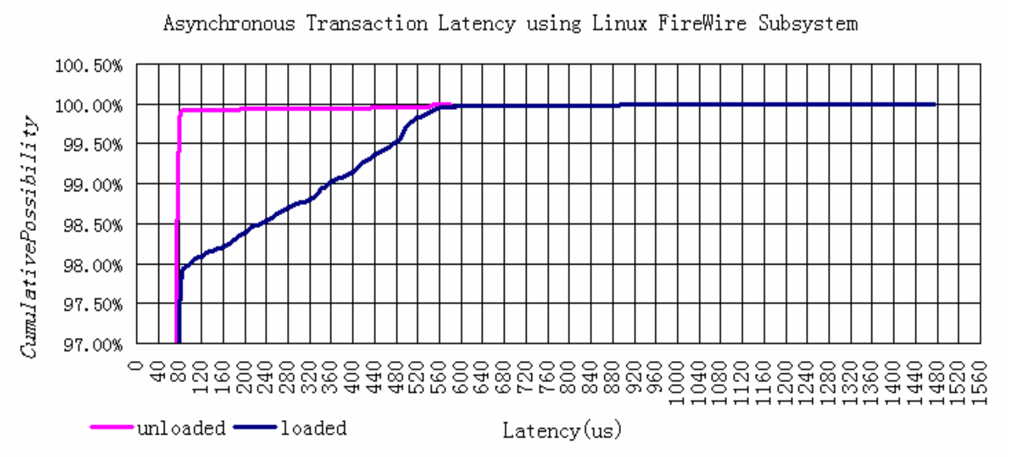


Figure 2-18 Asynchronous Transaction Latency on Linux FireWire Subsystem (top 3% of the cumulative curve)

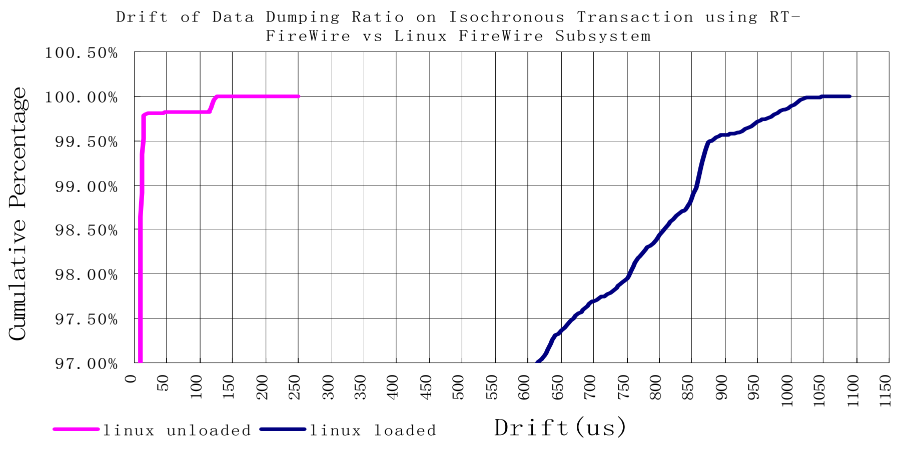


Figure 2-19 Drift of Data Receiving Rate on Isochronous Transaction using Linux FireWire Subsystem

Due to the wide spanning range, the chosen step on the latency value (x-value) is a bit big to make the curve fit in one figure. In Table 2-4 more precise values are presented on three thresholds, i.e. 97%, 99.999% and 100%

Cases	97% threshold	99.999% threshold	100% (Worst Case)
Asynchronous unloaded	70µs	565µs	580µs
Asynchronous loaded	80µs	1055µs	1475µs
Isochronous unloaded	10µs	175µs	250µs
Isochronous loaded	615µs	1085µs	1090µs

Table 2-4 Threshold Representatives of Real-Time Performance on Linux FireWire Subsystem

Discussion and Conclusion

When the system is not loaded, the experiment results on both asynchronous and isochronous transactions have already shown a relatively big difference in latency values or receiving rate drift in the critical range of cumulative percentage (e.g. between 97% and the worst case (100%) performance). With added load, performance is clearly worsened. Moreover when the system is heavily loaded, the curve is much less steep then in the case system is not heavily loaded. As already discussed, this indicates increased non-determinism and results in poorer real-time properties.

For real-time application, it is the worst case (or almost worst case, like 99.999% threshold) that drives the choice for underlying system. And for normal real-time control application, e.g. motion control, the measured worst case performance can not satisfy the requirements.

Therefore, the conclusion can be reached: Linux FireWire Subsystem can not be used as underlying networking platform for real-time control application. Hence, there is a need to develop a special FireWire Subsystem for use in real-time control application.

3 Real-Time FireWire Subsystem

3.1 Introduction

In this chapter, the implementation of the RT-FireWire (Real-Time FireWire Subsystem) is presented, including the architecture, core components and protocol adaptation.

3.2 Fundamental of RT-FireWire

This section describes the fundamentals of RT-FireWire. In short, the newly designed FireWire subsystem is real-time because the whole software stack is moved to the real-time domain, i.e. RTAI [RTAI 2005]. To unveil more details, the story starts from the explanation about RTAI and its co-existence with Linux. After that, the settling of RT-FireWire in RTAI is described.

RTAI is based on Adeos, which is a resource virtualization layer available as a Linux kernel patch, a simple, yet efficient real-time system enabler, providing a mean to run a regular GNU/Linux environment and a RTOS (e.g. RTAI), side by side on the same hardware. Adeos enables multiple entities called domains to exist simultaneously on the same hardware. These domains do not necessarily see each other, but all of them see Adeos. All domains are likely to compete for processing external events (e.g. interrupts) or internal ones (e.g. traps, exceptions), according to the system-wide priority they have been given [FusionTeam, 2004]. See Figure 3-1 for the illustrated concept. Every domain can register to be notified about certain events. And events are handled in the pipeline way with higher priority domains getting to handle events before lower priority domains.

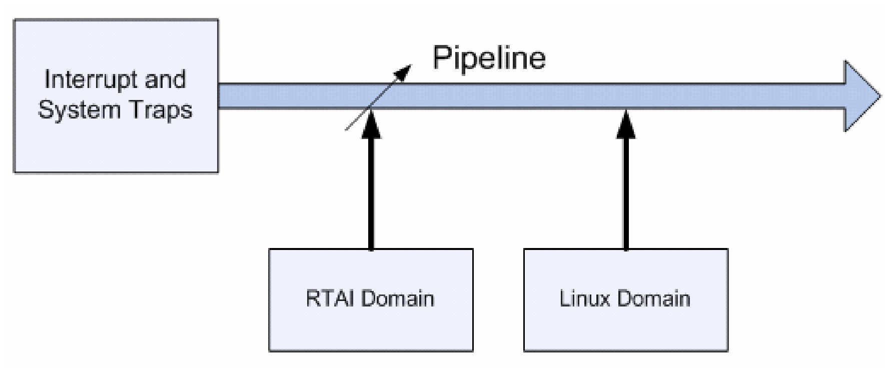


Figure 3-1 Conceptual Diagram of Domain Pipeline in Adeos

Because RTAI domain is ahead in the pipeline, it is the first to be notified of any incoming interrupts of interest, and because of its heading position, RTAI is totally in control of the interrupt propagation to other low-priority domains, mainly Linux. In other words, RTAI will not let any interrupts go to Linux, if it is busy dealing with some real-time task, e.g. handling a FireWire packet. That way, theoretically RTAI grasps the full control of CPU’s processing power, which is the most critical basis for any real-time subsystem built in it, e.g. RT-FireWire.

As important as the real-time interrupt handling is the task scheduler in the RTAI domain. The scheduler implements priority-based scheduling for tasks in the RTAI domain. The original Linux kernel is wrapped into a lowest priority task in this scheduler when RTAI is loaded, therefore all the real-time tasks will have a higher priority than Linux, so all of them can preempt Linux tasks. RT-FireWire employs more than one real-time task in RTAI for its internal processing.

3.3 Settling RT-FireWire in RTAI

This section describes the implementation of settling RT-FireWire in RTAI domain. First the system overview of RT-FireWire is given, based on which the design of task composition for RT-FireWire is presented. Based on the composition, the skeleton of RT-FireWire is built up. Second, the implementation of real-time memory management in RT-FireWire is presented. In the third part of this section, two other relatively minor features in RT-FireWire are introduced: real-time procedure call and packet capturing.

3.3.1 System Overview

Here we present the overview of RT-FireWire in Figure 3-2. Compared with Figure 2-10, the visible changes go to the driver for adapter, kernel implementation and interface to underlying OS, i.e. RTAI.

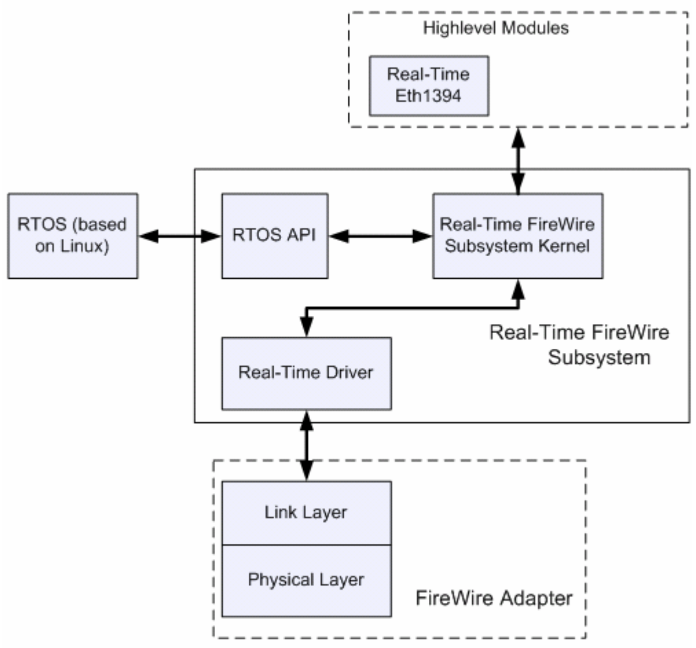


Figure 3-2 RT-FireWire Overview

Figure 3-3 shows the kernel diagram of RT-FireWire. Compared with figure2-11, two more function blocks are added: Real-Time Memory Management and RTcap. RTcap stands for Real-Time (Packet) Capturing, which is used to capture all incoming or outgoing packet. Captured packets are used later on for network behavior analysis.

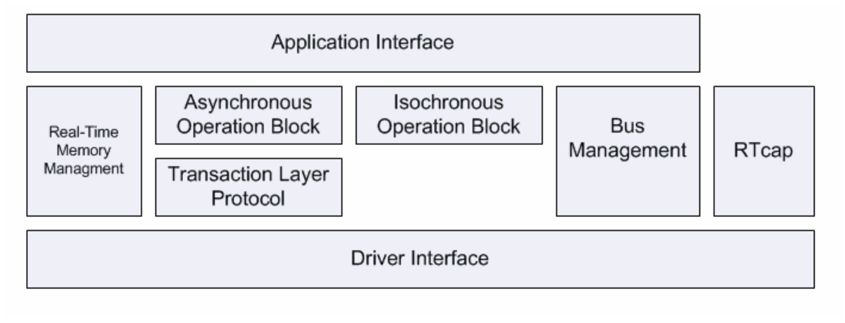

Application Interface
Real-Time Memory Managment
Asynchronous Operation Block
Transaction Layer Protocol
Isochronous Operation Block
Bus Management
RTcap
Driver Interface

Figure 3-3 RT-FireWire Kernel

3.3.2 Architecture and Task Composition

The architecture of RT-FireWire is strictly divided into several layers, each of which corresponds to one layer in the network protocol specification on FireWire. A top-view of the layered architecture is given in Figure 3-4.

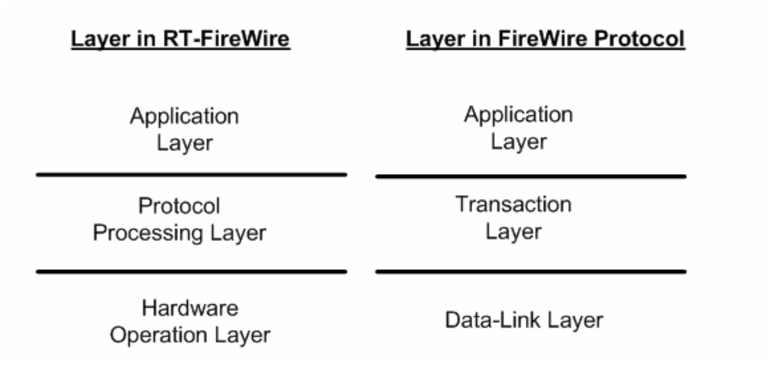


Figure 3-4 Layers in RT-FireWire, corresponding to the layers in FireWire protocol

RT-FireWire is composed of several tasks, each of which is a schedulable task object in the RTAI scheduler. All the tasks in RT-FireWire can be seen as servers that handle asynchronous events from outside. The top-view of task composition within RT-FireWire's layered architecture is shown in Figure 3-5. In next sections, task(s) on each layer will be described.

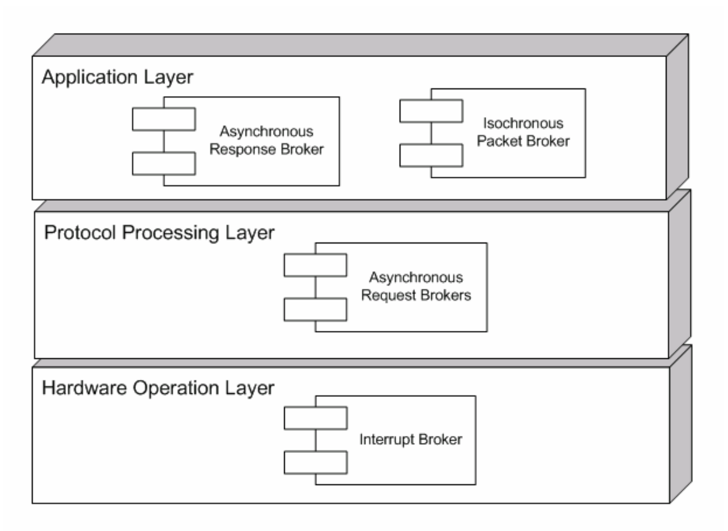

Application Layer
Asynchronous Response Broker
Isochronous Packet Broker
Protocol Processing Layer
Asynchronous Request Brokers
Hardware Operation Layer
Interrupt Broker

Figure 3-5 Task Composition in RT-FireWire

3.3.3 Hardware Operation Layer

Interrupt Handling

In the hardware operation layer, one task called “Interrupt Broker” is installed to handle the various bus events from external FireWire network. From Object-Oriented point of view, each event is represented by a class inherited from the super-class “ISR Event”, as illustrated in Figure 3-6.

Each event contains the pointer to the routine for handling the event (interrupt from hardware) in concern, and the argument to pass to that routine. So when a certain event is hooked to the broker, the routine addressed by the pointer is executed by the broker.

Short explanation about each event:

- Asynchronous event for request receiving occurs upon arrival of asynchronous request packet.
- Asynchronous event for response receiving occurs upon arrival of asynchronous response packet.
- Asynchronous event for request transmitting occurs after adapter has successfully transmitted a request packet and the acknowledgment has been received from targeting node.
- Asynchronous event for response transmitting occurs after adapter has successfully transmitted a response packet and the acknowledgment has been received from targeting node.
- Besides, there can be 64 events for each isochronous channel if adapter is tuned to listen to that channel.

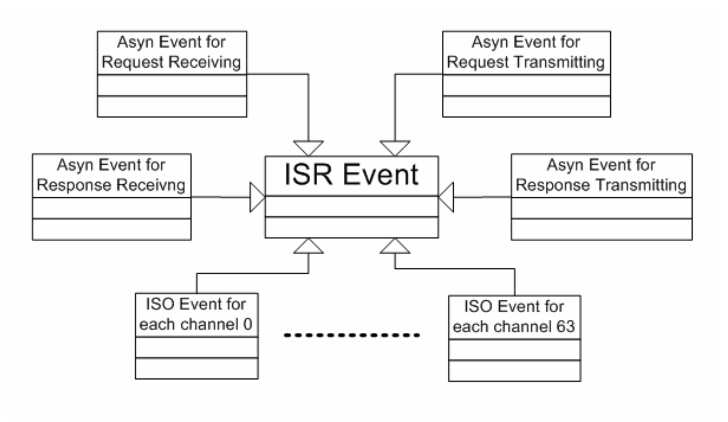

Asyn Event for Request Receiving
Asyn Event for Request Transmitting
Asyn Event for Response Receivng
ISR Event
Asyn Event for Response Transmitting
ISO Event for each channel 0
ISO Event for each channel 63

Figure 3-6 Events in Hardware Operation Layer

Time Stamping in Driver

In the hardware operation layer of RT-FireWire, receiving time of all incoming packets is stamped in the management header (which is not sent or received via the network) of the packet object in the driver’s receiving routine before they are passed on. For outgoing packets, the driver stamps the sending time (right before stuffing the packet into hardware) into the data part of packets upon the request of highlevel protocols. This is implemented via allowing highlevel protocols to assign a pointer to the data part. Stamping for both routine is shown in Figure 3-7.

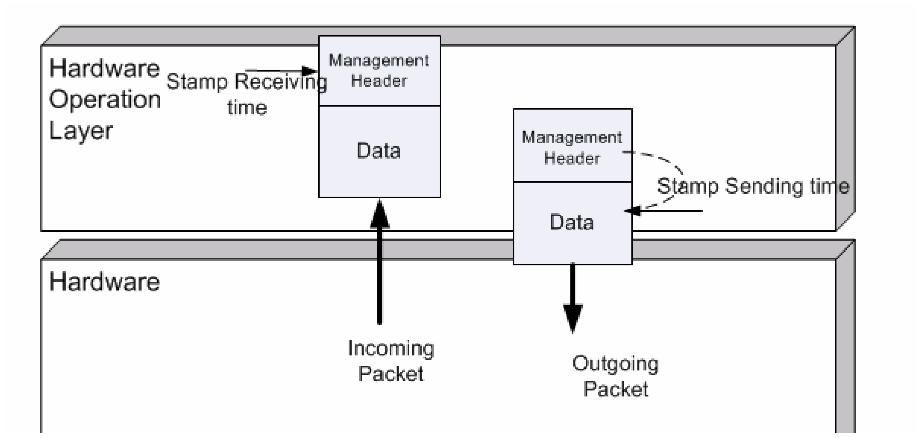


Figure 3-7 Time Stamping for Incoming and Outgoing Packets

3.3.4 Protocol Processing Layer

Prioritized Request

One limitation while using original FireWire transaction protocol in real-time context is the lack of priority in packets. Because the asynchronous transaction on FireWire consists of request sub-transaction and response sub-transaction, it will make the protocol fit more in real-time context if the request packet comes with a priority that determines how fast the request should be handled on the responding node. Moreover, it would fit more in real-time if the packet that arrived later but with a higher priority can preempt the ongoing processing of previous packet which has a lower priority. This preemptability of transactions, although only limited to the software stack (for now, it is not possible to have preemptive transaction on the Link-Layer of FireWire network), can improve the suitability of using whole FireWire subsystem in real-time context.

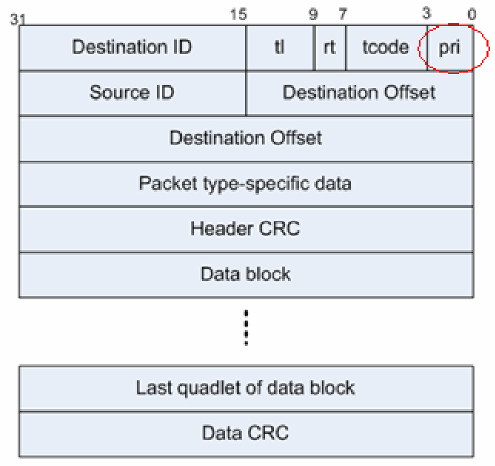


Figure 3-8 Prioritized Request

As shown in Figure 3-8, the last 4 bits in the first quadlet of asynchronous packet are used to represent the priority. These 4 bits are reserved for backplane environment in 1394 specification [Anderson, 1999], but since RT-FireWire only aims to be used in cable environment, it is free to use these 4 bits for other purpose here, i.e. carrying the priority of transaction issued by the application on requesting node. Therefore, we have 16 priorities, with 0 being defined as the highest. The highest priority is reserved for bus internal server, while the lowest one is reserved for non real-time applications. The rest 14 priorities are for real-time applications.

Prioritized Waiting Queue on Requesting Node

Before sending, the outgoing requests are queued according to the ascending order of their priorities. That way, the real-time requests, even if they are issued later, can still jump over the requests, which are queued before them but with a lower priority. In short, by using this mechanism, the real-time transaction is allowed to preempt the non real-time transaction on the requesting node. This preemption on requesting node is illustrated in Figure 3-9. The number in bracket is the priority.

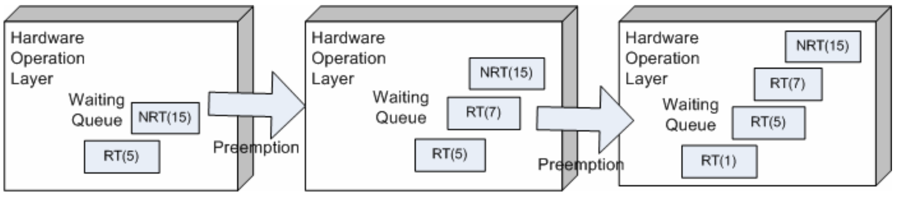


Figure 3-9 Transaction Preemption on Requesting Node

Brokers for Prioritized Requests on Responding Node

On the responding node, based on the packet priorities, three transaction servers (Request Broker for Bus Internal Service, Request Broker for Real-Time Application and Request Broker for Non Real-Time Application) are employed to handle the requests accordingly, as illustrated in Figure 3-10.

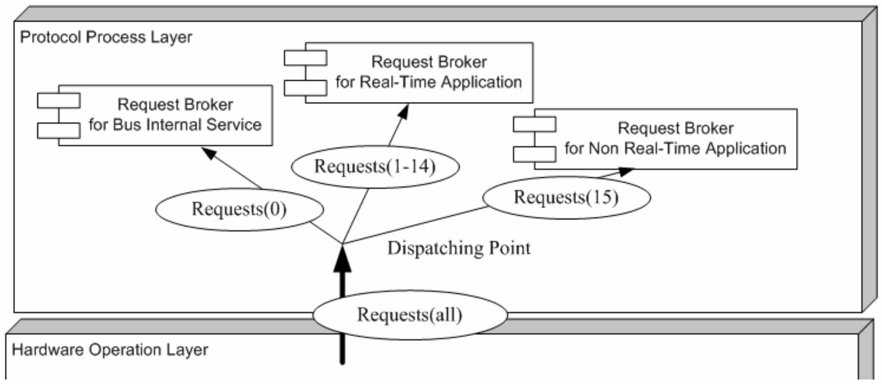

Protocol Process Layer
Request Broker for Real-Time Application
Request Broker for Bus Internal Service
Request Broker for Non Real-Time Application
Requests(1-14)
Requests(0)
Requests(15)
Dispatching Point
Requests(all)
Hardware Operation Layer

Figure 3-10 Request Brokers in Protocol Processing Layer

Broker for bus internal service has the highest priority among the three. The broker for non real-time application goes to the Linux domain, since it deserves the lowest priority.

3.3.5 Application Layer

In the application layer of RT-FireWire, two tasks are installed for dispatching asynchronous response packets or isochronous packet to applications: asynchronous response broker and isochronous packet broker.

Both tasks use “callback” to communicate with application, i.e. execute the callback routine provided by application. For asynchronous transaction, the pointer to the “callback” stays with the request packet; for isochronous transaction, the pointer to the “callback” stays in the settings for that certain channel. The “callback” allows the application to customize the way of synchronization between it and RT-FireWire. In case an immediate synchronization is needed, a semaphore can be used, as illustrated in Figure 3-11.

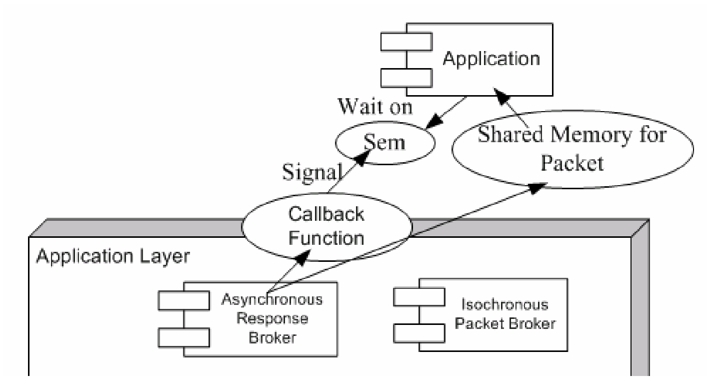


Figure 3-11 Brokers in Application Layer

3.4 Real-Time Memory Management

Another critical issue in general real-time system is resource allocation. The resource can be memory, hardware I/O, external storage, etc. But in most of the scenarios, memory is the main concern, therefore having a real-time memory allocation is as important as the architecture design. This section addresses the design and implementation of real-time memory management in RT-FireWire.

3.4.1 Common Packet Buffer Structure

To grant the system full extensibility, the static memory allocation in RT-FireWire uses the most generic memory object, so called real-time packet buffer (rtpkb). Rtpkb consists of a buffer management structure and a fixed-sized data buffer. It is used to store network packets on their way from the API routines through the stack to the hardware interface or vice versa. Rtpkb is allocated as one chunk of memory that contains both the management structure (rtpkb header) and the buffer memory itself, as shown in Figure 3-12.

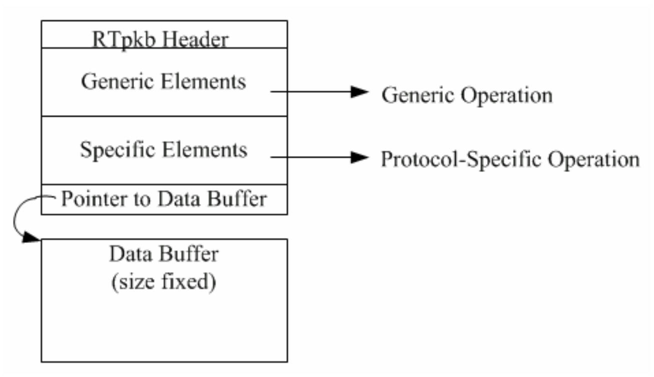


Figure 3-12 Real-Time Packet Buffer

All the generic operations from memory management module are carried out only with the generic elements of rtpkb header, while the protocol-specific operations, e.g. FireWire transaction protocol, are carried out only with the protocol-specific elements. Therefore all protocol-specific stuff is transparent to the memory management module, which is necessary to allow RT-FireWire to freely exchange packet buffer with the applications on it and vice versa.

3.4.2 Packet Buffer Queue

Based on the rtpkb, another component is designed for memory management module, i.e. Packet Buffer Queue. A queue can contain an unlimited number of rtpkbs in an ordered way. An rtpkb can either be added to the head or the tail of a queue. When a rtpkb is removed from a queue, it is always taken from the head.

3.4.3 Packet Buffer Pool

During the initialization of whole system or a certain application, an estimated number of packet buffers must be pre-allocated and kept ready in so-called buffer pools. Most packet producers (e.g. interrupt broker in hardware operation layer, etc) have their own pools in order to be independent of the load situation of other parts of the system. Pools can be extended or shrinked during runtime. Before shutting down the whole system, every pool has to be released.

Pools are organized as normal rtpkb queues. When a rtpkb is allocated, it is actually dequeued from the pool's queue. When freeing an rtpkb, the rtpkb is enqueued to its owning pool. rtpkbs can be exchanged between pools. In this case, the passed rtpkb switches over from its owning pool to a given pool, but only if that pool can pass an empty rtpkb (as for compensation) from its own queue back. This is necessary to keep the memory allocation in each pool clearly independent. This way, the chance for non real-time processing to starve real-time processing for memory is clearly prevented, because each application or processing, either real-time or not, can only hold memory on its own expense, i.e. from its own pool. The buffer exchanging between pools is illustrated in Figure 3-13.

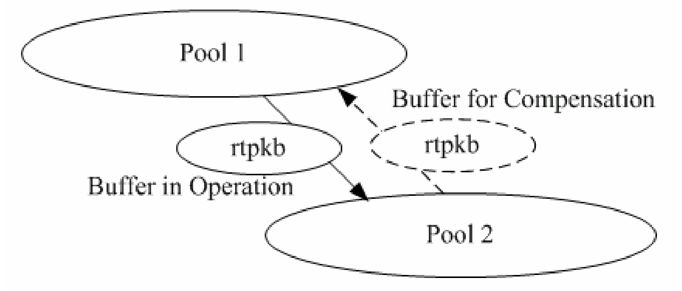


Figure 3-13 Buffer Exchanging between Pools

The deployment of memory pools in RT-FireWire reflects its internal layered structure. See Figure 3-14.

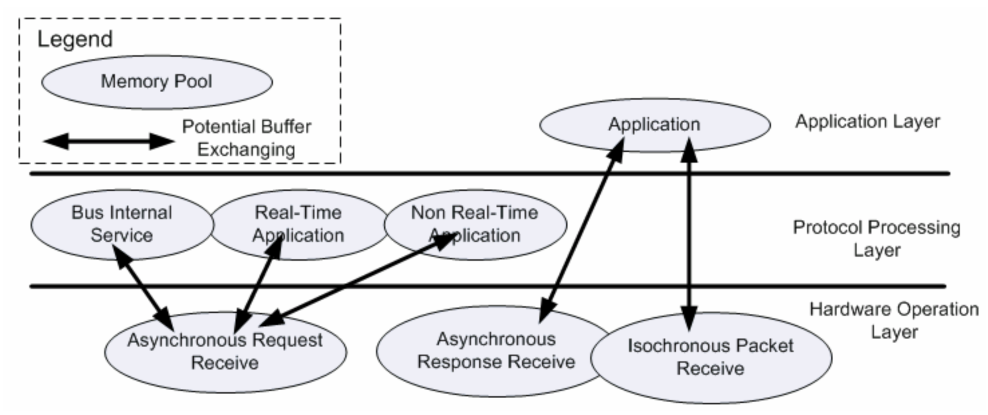

Legend
Memory Pool
Potential Buffer Exchanging
Application
Application Layer
Bus Internal Service
Real-Time Application
Non Real-Time Application
Protocol Processing Layer
Hardware Operation Layer
Asynchronous Request Receive
Asynchronous Response Receive
Isochronous Packet Receive

Figure 3-14 Layered Deployment of Memory Pools in RT-FireWire

3.5 Other Design Issues in RT-FireWire

3.5.1 Real-Time Procedure Call

In RT-FireWire, there is a need to trigger the real-time transaction from non real-time context, i.e. Linux domain. To this end, the Real-Time Procedure Call (RTPC) is designed and implemented. RTPC is an approach to let non real-time task, e.g. task in Linux, run a certain piece of code in real-time context. The rationale behind is illustrated in Figure 3-15.

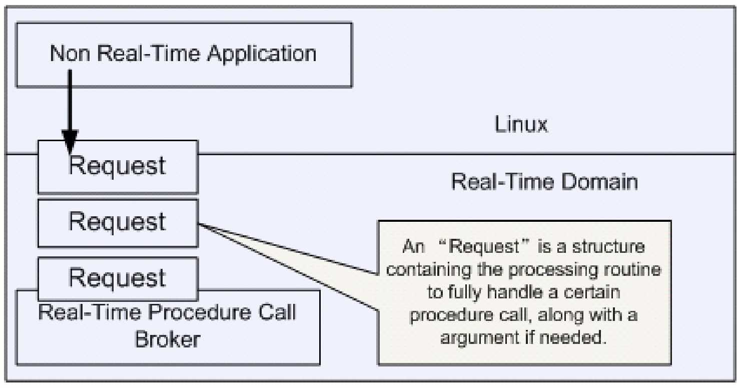

Non Real-Time Application
Linux
Request
Real-Time Domain
Request
Request
Real-Time Procedure Call Broker
An "Request" is a structure containing the processing routine to fully handle a certain procedure call, along with a argument if needed.

Figure 3-15 Conceptual Diagram of Real-Time Procedure Call

During system initialization, the "Real-Time Procedure Call Broker" is created in the real-time domain as a real-time task. The request to that broker is sent by tasks in the non real-time domain, possibly user-space task in Linux. The request contains the pointer to the routine that should be run in real-time, the execution arguments and the buffer for storing execution results. The broker handles requests in FIFS (First In First Served) fashion. After finishing a request, it wakes up the corresponding non-real-time task to take back the results.

The current usage of Real-Time Procedure Call in RT-FireWire is for processing request generated from user-interface console. For example, user can request a latency calibration between local node and one remote node. The calibration task should then be switched to real-time context in order to accordingly measure an accurate latency.

3.5.2 Real-Time Packet Capturing

Another feature in RT-FireWire is Packet Capturing service. The whole service consists of two parts: packet capturing module in the kernel side and analysis tool in the user side.

The kernel-side module captures both incoming and outgoing packets and put them to a so called “Captured Packet Queue”. The captured packets are passed to analysis tool, which could stay in user space. See Figure 3-16 for the illustration.

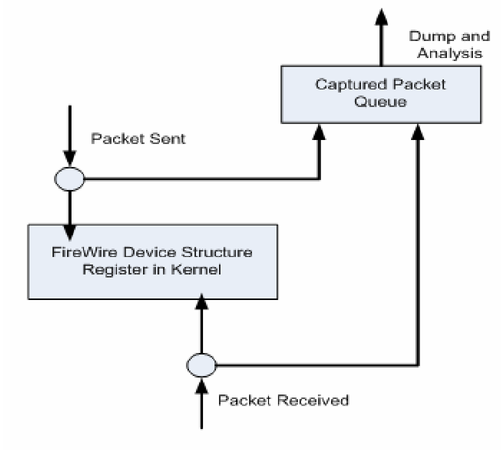


Figure 3-16 Working of Packet Capturing

Note that the procedure of capturing packet includes no copying, instead, the efficient “pointer assigning” is used. The head of “Captured Packet Queue” is just a pointer to “Real-Time Packet buffer”, and in each “Real-Time Packet Buffer” object there is also a pointer to another buffer object. That way, it is possible to just link all captured buffer object to the “Captured Packet Queue”. Due to the zero-copy linking, a new concern pops out, which is about memory leakage. Each captured packet in the queue is also waiting for being processed by the “traffic analyzing tool”, so their memory can not be freed immediately after the operation on that packet is finished. But if the memory is not freed in time, it will cause kind of memory leakage to the memory pool where these packets come from, i.e. the memory pool that belongs to the specific application. To prevent this, a memory pool is also pre-allocated for the packet capturing module. In case a packet is captured, a compensating packet buffer is allocated from the pool of packet capturing module and linked to the captured packet. When the application attempts to free that packet, the “packet-free” function (from memory management module) will be called and it will check if the packet has another compensating packet linked. If yes, the compensating packet will be freed instead. That way, the packet capturing stays transparent to applications. See Figure 3-17 for illustration of the whole procedure.

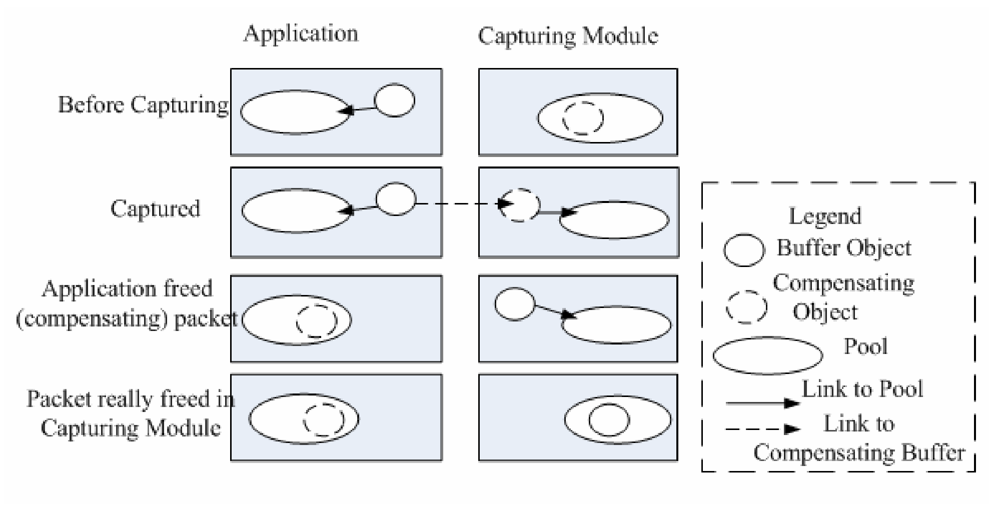

Application
Capturing Module
Before Capturing
Captured
Application freed (compensating) packet
Packet really freed in Capturing Module
Legend
Buffer Object
Compensating Object
Pool
Link to Pool
Link to Compensating Buffer

Figure 3-17 Packet Capturing Procedure

3.6 Performance Benchmarking on RT-FireWire

Like what has been done on Linux FireWire Subsystem, a performance benchmarking is also carried out on RT-FireWire to see its suitability for use in real-time.

Test Bench Setup

To make the results directly comparable, the hardware employed in this experiment is exactly the same as in the experiment on Linux FireWire Subsystem.
2 PC104 stacks are employed in this experiment. Detailed information follows:

- PC104:
  VIA Eden 600 MHz, 256 Mb Memory, 32 Mb flash disk.
- FireWire Adapter:
  PC/104+ board with VIA VT6370L Link & Physical layer chip, supporting 400 Mb/s transferring speed at maximum. (See [Zhang, 2004] for more related information)

The software (Operating System) is a bit different, since now it has been a real-time Operating System.

- Software in use: Linux kernel 2.6.12 plus RTAI/fusion 0.9.

Experiment Cases

The performance in 4 cases are evaluated: asynchronous transaction without system load, asynchronous transaction with heavy system load, isochronous transaction without system load and isochronous transaction with heavy system load. The experiments on both asynchronous transaction and isochronous transaction are illustrated in Figure 3-18 and Figure 3-19. For each case, two nodes are involved in the experiment: one is so-called requesting node that is actively sending the request; another is so-called target node that is receiving the requests, processing them, and (in asynchronous transaction) send responses back. The data sending rate on client node is 1 KHz. And the amount of collected samples for each case is 100,000. For each experiment, the data load is set to 56bytes.

Imposing System Load

To put the experiment in an extremely loaded system, extra processing load needs to be imposed explicitly. Three ways of imposing system load are used together in this experiment.

- Creating a flood of interrupts from external world via network by using a third node to send a lot of random data to the nodes in experiment.
- Creating a flood of interrupts from hardware disk I/O by reading the whole hard disk.
- Creating a flood of system calls via Linux command line. This will make a lot of kernel-user space switch.

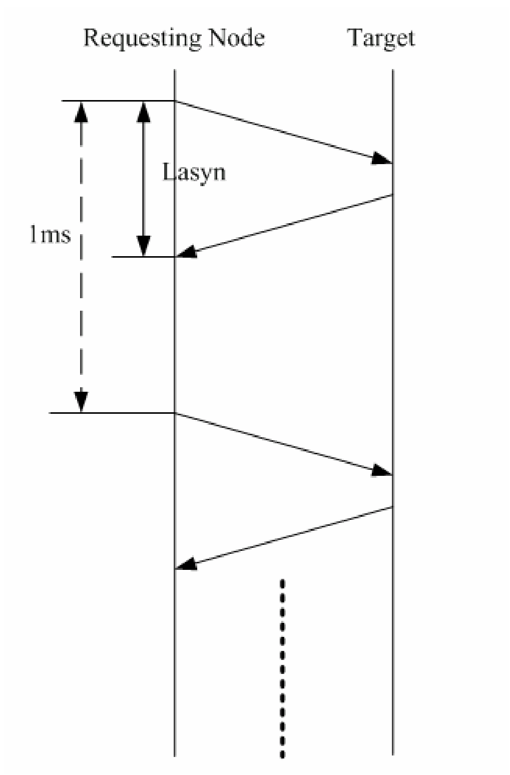


Figure 3-18 Asynchronous Transaction Latency

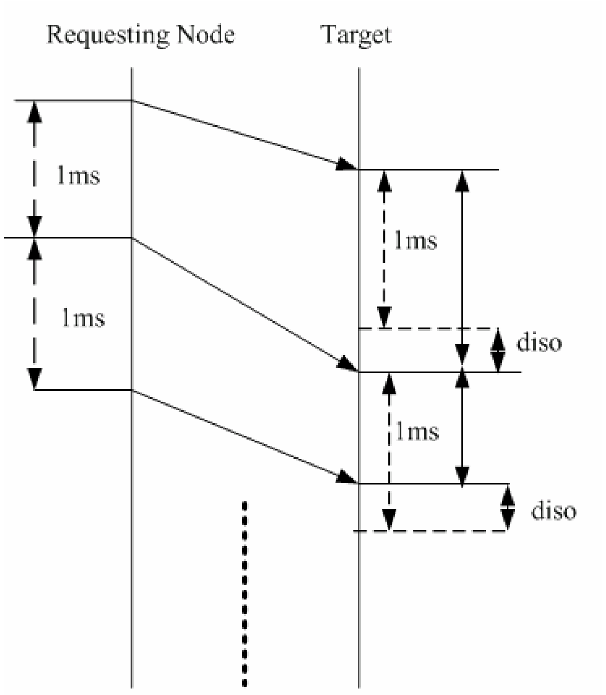


Figure 3-19 Drift of Data Receiving Rate on Isochronous Transaction

3.6.1 Measurement Results

The methodology to present the results of measurements on Linux FireWire Subsystem is reused here. The result is presented by using cumulative percentage curves. At any point on the cumulative percentage curve, the cumulative percentage value (y-value) is the percentage of measurements that had a latency less than or equal to the latency value (x-value). The latency at which the cumulative percentage curve reaches 100 percent represents the worst-case latency measured. For real-time transaction latency, the ideal cumulative percentage curve is one that is steep with a minimal decrease in slope as the curve approaches 100 percent.

Therefore, the cumulative percentage at a certain latency value can be translated to be the probability of the transaction being able to meet real-time constraints when deadline is assumed to be equal to that latency value, as shown in Figure 3-20 and Figure 3-21.

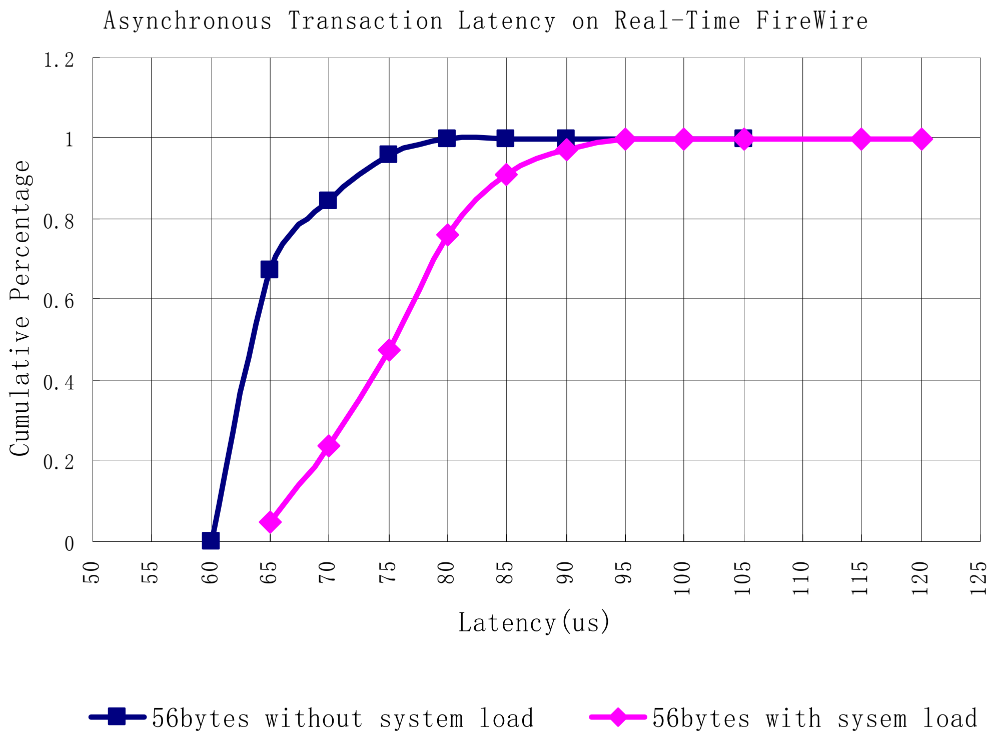


Figure 3-20 Asynchronous Transaction Latency using RT-FireWire

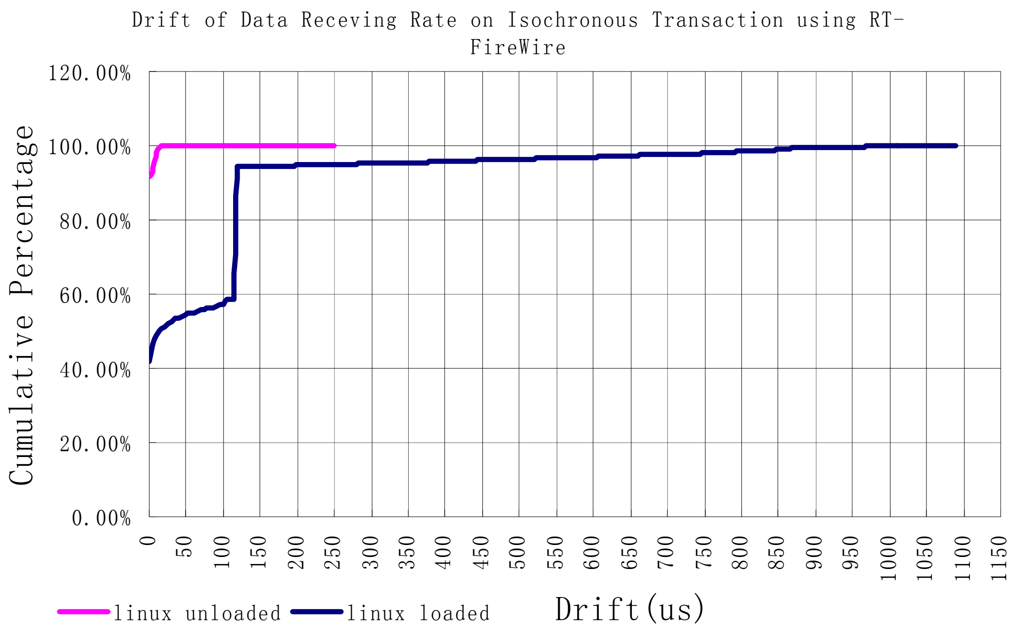


Figure 3-21 Drift of Data Receiving Rate on Isochronous Transaction using RT-FireWire

In Table 3-1, more precise values are presented on three thresholds, i.e. 97%, 99.999% and 100%.

Cases	97% threshold	99.999% threshold	100% (Worst Case)
Asynchronous unloaded	75µs	90µs	105µs
Asynchronous loaded	90µs	115µs	120µs
Isochronous unloaded	10µs	45µs	50µs
Isochronous loaded	45µs	90µs	95µs

Table 3-1 Threshold Representatives of Real-Time Performance on Linux FireWire Subsystem

The plot on Linux FireWire Subsystem (chapter2) is put together with the one on RT-FireWire in Figure 3-22 and Figure 3-23, which gives more insight about how the performance is improved by RT-FireWire. (Only top of the curves are presented here)

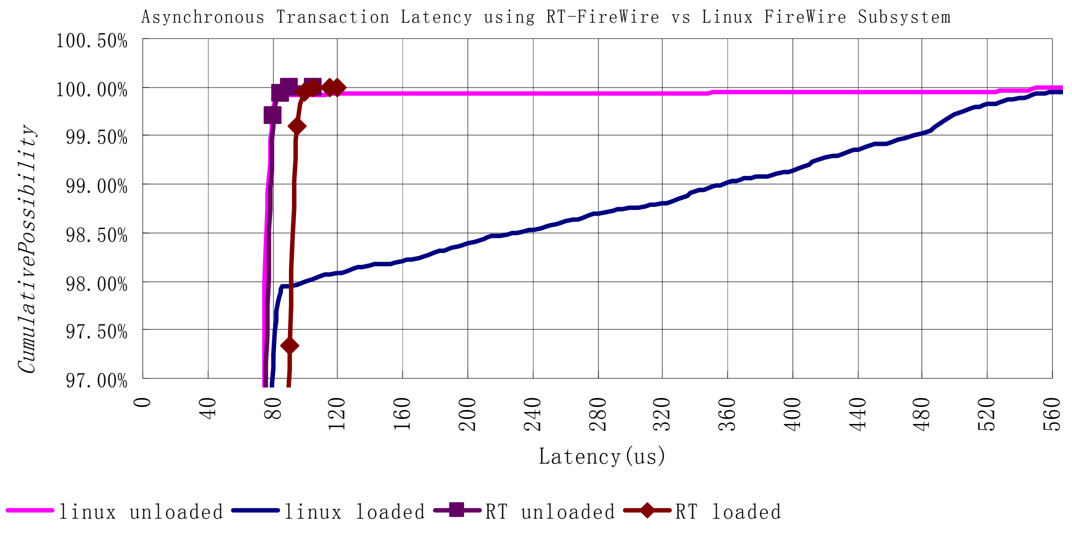


Figure 3-22 Comparison between RT-FireWire and Linux FireWire Subsystem (Asynchronous Transaction)

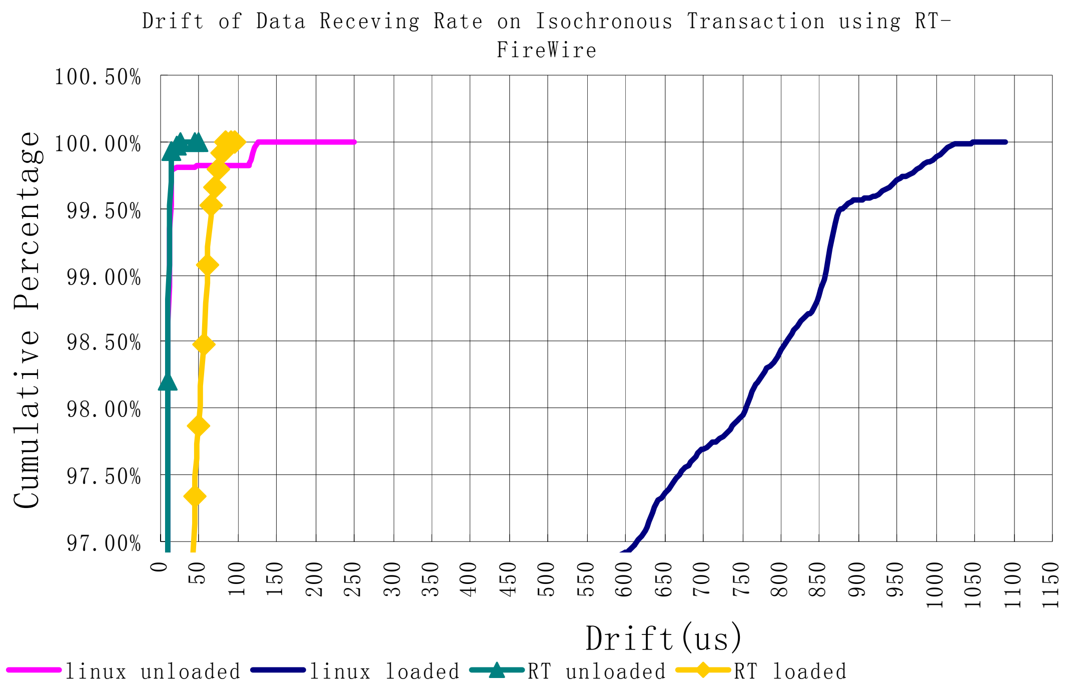


Figure 3-23 Comparison between RT-FireWire and Linux FireWire Subsystem (Isochronous Transaction

The data load is another issue that may influence the real-time behavior of RT-FireWire. Figure 3-24 and Figure 3-25 presents the latency or drift over different data load.

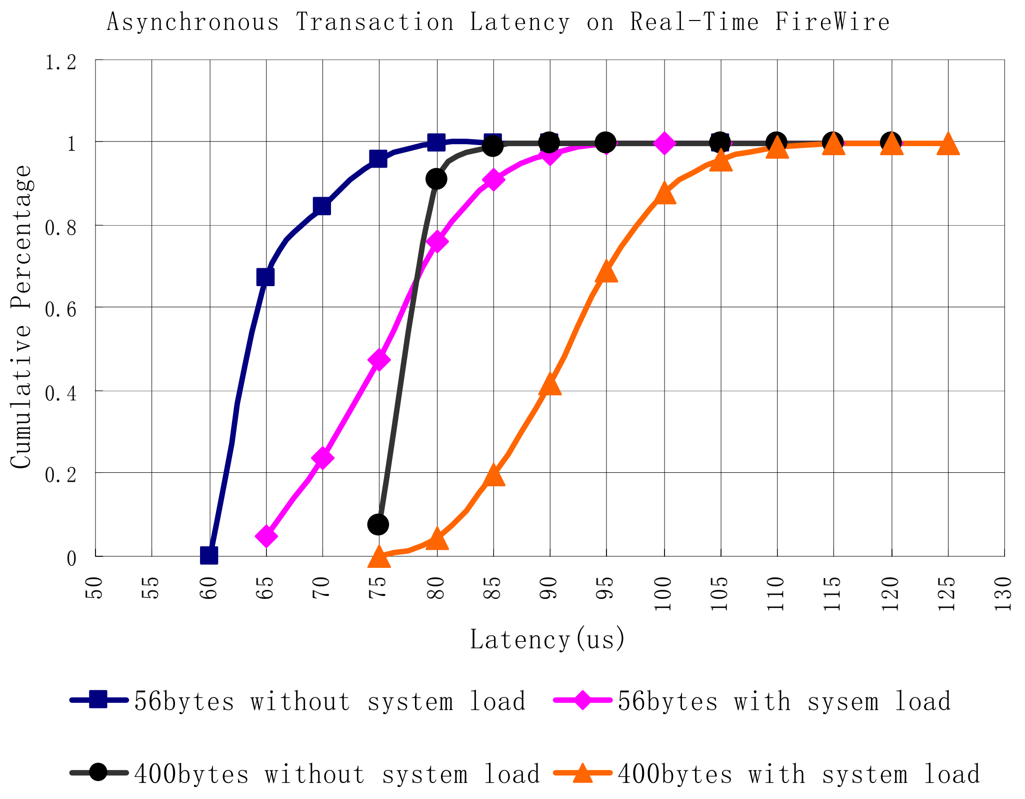


Figure 3-24 Asynchronous Transaction Latency on RT-FireWire with different data load

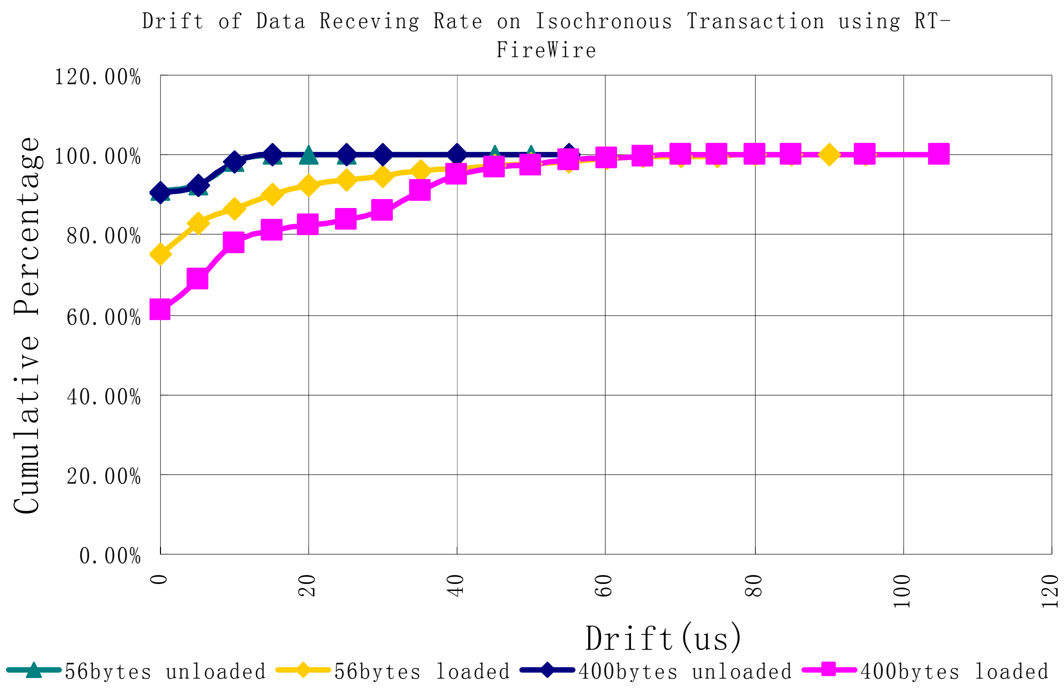


Figure 3-25 Drift of Data Receiving Rate on Isochronous Transaction using RT-FireWire with different data load

3.6.2 Discussion and Conclusion

Compared with Linux FireWire Subsystem, both asynchronous latency and isochronous drift on RT-FireWire give quite steep curves, as shown in the figures above. That means RT-FireWire gives much more deterministic behavior, which is especially crucial when time critical communication is needed.

But there still exists an un-ignorable gap between the average performance on RT-FireWire and the worst case performance. In case of asynchronous transaction it is the gap between worst case latency and 97% threshold latency value; in case of isochronous transaction it is the worst case data receiving rate drift and 97% threshold drift value. These gaps can not be filled by RT-FireWire due to the limitation of current solution, i.e. RT-FireWire is only software-based solution trying to achieve hard real-time communication. From RT-FireWire point of view, the hardware underlying, even the internal implementation of the Operating System underlying can be kind of “black box”. No attempts are made in this project to open these “black box”.

Based on the work presented in this chapter, RT-FireWire has been converted to a Open Source project, register in www.berlios.de. It can be directly visited via rtfirewire.berlios.de.

4 Real-Time IP Network over RT-FireWire

4.1 Introduction

This chapter presents the implementation of stacking real-time IP network in the application layer of RT-FireWire. First, the emulation of Ethernet over FireWire is explained. Second, another Open Source project, RTnet is introduced. RTnet implements a real-time networking framework on Ethernet. Based on the Ethernet emulation, RTnet can be stacked on RT-FireWire. In last part of this chapter, the results of performance measurement on IP over FireWire is given, and compared with the performance of IP over Ethernet.

4.2 Ethernet Emulation over RT-FireWire

4.2.1 "IPover1394" Spec

Before this project, a spec called "IP over 1394" [Johansson, 1999] has been released over Internet, which standardizes the protocol of transferring IP packets on FireWire's primary transactions, i.e. asynchronous and isochronous, so to make FireWire appear almost the same as Ethernet from the application point of view. In Linux FireWire Subsystem, a highlevel module, Eth1394 (Ethernet Emulation over FireWire), has been developed according to this specification. In this section, the basic rationale behind "IPover1394" is explained at first. After that, the modifications to "IPover1394" are presented and explained which is implemented in the re-implementation of Eth1394 in RT-FireWire.

4.2.2 Minimum Requirements to be IP-capable

Not all serial bus devices/nodes are capable of reception and transmission of IP packets. Several minimum requirements should be fulfilled for a node to be IP-capable:

- Nodes have unique hardware address (unique in the network scope). This is important because transaction on the IP level is peer-to-peer. A peer-to-peer relation needs to be established between the IP address and the hardware address of a certain medium.
- Nodes support multicasting. This is especially important because the multicasting ARP protocol needs to be carried out for mapping between IP address and the underlying hardware address.

Besides the above two essential requirements, another desired one is that the underlying medium should support transmitting packets of relatively large data size. This is because IP header introduces some overhead (20 bytes of data). Therefore the medium should at least be able to transmit packets that contain more data than the IP header. Moreover, it is better to have large size packet under IP, since that relatively decreases the protocol overhead.

4.2.3 Addressing Mechanism

This section explains the mechanisms to establish the peer-to-peer relation between IP address and FireWire node address. Two different mechanisms are introduced. The first one is from original "IPover1394" specialization; the second is customized in this project. Through the comparison, it can be seen that the newly customized addressing mechanism fits better in real-time application context.

The original IPover1394 spec employs the 64-bit GUID (Global Unique ID) of each FireWire adapter chip as the hardware address. The GUID is the ID from manufacturer of each single FireWire adapter, similar to the MAC address of Ethernet adapter. The GUID can be read from the internal register of a certain adapter by using normal asynchronous transaction access on FireWire. That way, the GUID of any FireWire node (adapter) can be known to the whole network, and the peer-to-peer relation between GUID and Node ID can be established.

The strength of using GUID as the hardware address in IP over FireWire is that, the "hardware address" can be guaranteed to be unique even in the world scale, just like the MAC address of Ethernet. But in Ethernet, MAC address is directly used as link layer address for transaction, but in FireWire GUID is not used in transaction. Therefore, any packet that is stamped using GUID must go through an address resolution procedure before it can be put on the fly. Including the address resolution in IP protocol itself, i.e. the resolution between IP address and hardware address (in this case, it is GUID), the whole address resolution procedure includes two sub-resolution, which is not considered being efficient. The conceptual diagram of the address resolution process based on 64-bit GUID is given in Figure 4-1.

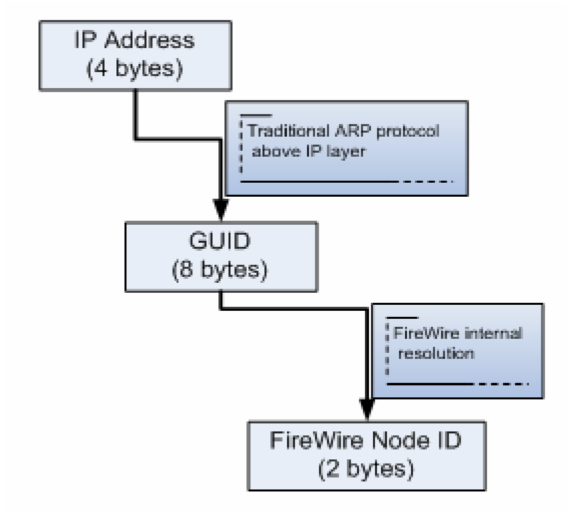

IP Address
(4 bytes)
Traditional ARP protocol above IP layer
GUID
(8 bytes)
FireWire internal resolution
FireWire Node ID
(2 bytes)

Figure 4-1 Addressing Mechanism in "IPover1394" Spec

As stated above, the addressing mechanism in "IPover1394" spec is not considered to be optimal and efficient, especially for real-time context, therefore some modification is needed.

The new addressing mechanism for IP over FireWire is demonstrated in Figure 4-2. As shown, the FireWire node ID is directly employed as hardware address of each IP-capable FireWire node. That way, the resolution procedure from IP address to hardware address (FireWire node ID) only includes the resolution in IP protocol itself. In this project, 1394 address space is allocated statically to IP module, i.e. on each node, the 48 bits address offset for Eth1394 module are exactly the same. Therefore, the 16 bits Node ID is enough to represent the hardware address.

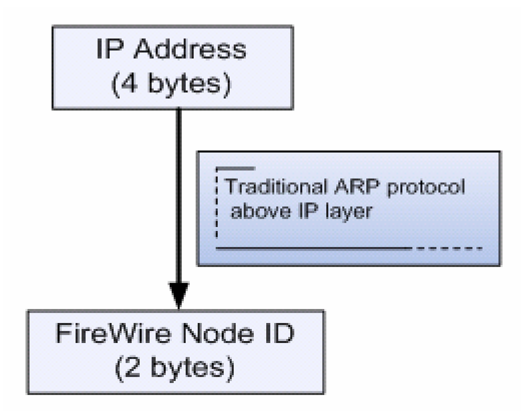


Figure 4-2 Modified Addressing Mechanism in RT-FireWire

To give exactly the same look as normal Ethernet devices, the “MAC” address of Eth1394 is extended to 6-bytes by filling 0 after the 2 bytes FireWire node id, as shown in Figure 4-3. This way all the highlevel stuff that is already working on Ethernet can be directly moved to FireWire, due to the same interface between Ethernet and Eth1394. Figure 4-4 gives the console view of Eth1394 interface on Linux/RTAI.

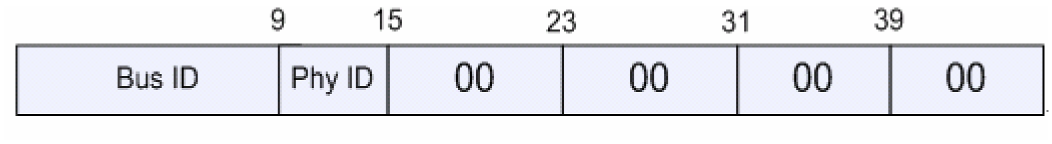
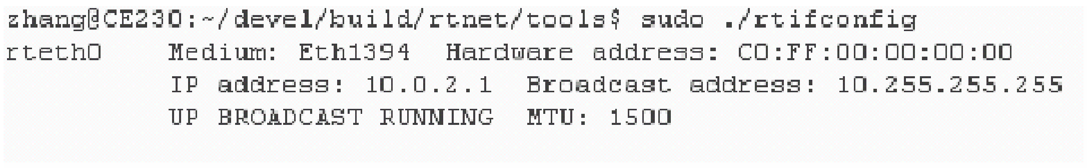


Figure 4-3 MAC address of Eth1394



```
zhang@CE230:~/devel/build/rtnet/tools$ sudo ./rtifconfig
rteth0    Medium: Eth1394  Hardware address: C0:FF:00:00:00:00
          IP address: 10.0.2.1  Broadcast address: 10.255.255.255
          UP BROADCAST RUNNING  MTU: 1500
```

Figure 4-4 Console view of Eth1394 interface

4.2.4 Address Resolution Protocol

The address resolution protocol (ARP) is a protocol used by the IP, specifically IPv4, to map IP address to the hardware addresses used by a data link protocol. The protocol operates below the network layer as a part of the interface between the OSI network and OSI link layer.

On Ethernet, the address resolution protocol is only used to map IP address to hardware address. E.g. the ARP packet only carries the IP address of certain Ethernet nodes. On FireWire, the transmission speed and packet size need to be specified before the packet is delivered to driver, which depends on the collected information about target node: the maximum speed and maximum packet size it can accept. Therefore, on Eth1394, it is required that ARP packet also carries the information about maximum speed and maximum packet size of the sending node. The 1394-specific ARP is called 1394ARP. Since the addressing mechanism has been adjusted, the original 1394ARP in IPover1934 can not be directly applied either. In this section, the complete ARP used for new addressing mechanism is presented, without referring back to the original one.

The ARP format on Eth1394 and Ethernet are given in Figure 4-5 and Figure 4-6 respectively.

Explanation about some fields:

- Hardware type indicates the underlying medium. E.g. 1 for Ethernet, 24 for FireWire.
- Protocol type indicates the protocol. In case of ARP, it is 0x0080.
- Hw_addr_len is the length of hardware address in bytes. In case of Eth1394, it is 6.
- Lg_addr_len is the length of logical address in bytes. In case of IP, it is 4.
- Operation code indicates the operation type of current packet, 1 for request 2 for response.
- Max_rec indicates the maximum packet size that can be accepted by the sender node.
- Sspd indicates the maximum speed that can be accepted by the sender node.

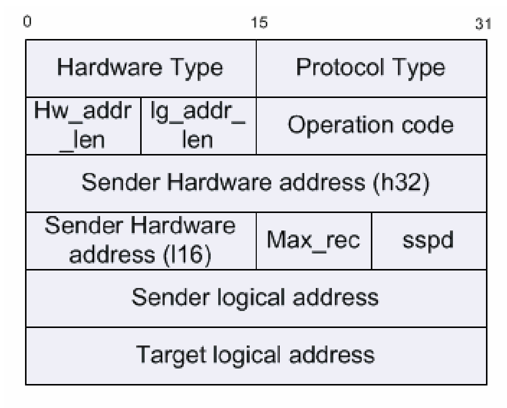


Figure 4-5 ARP packet format on Eth1394

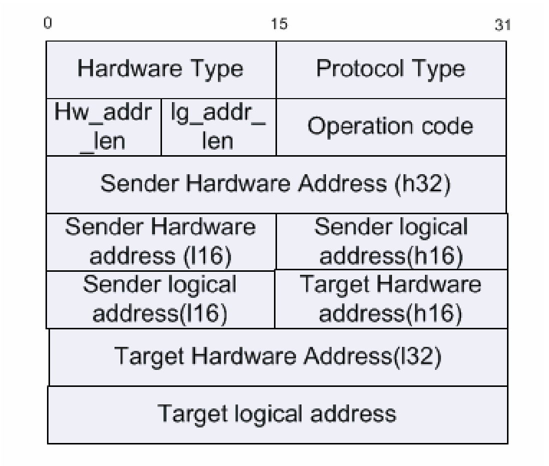

0
15
31
Hardware Type
Protocol Type
Hw_addr _len
lg_addr_ len
Operation code
Sender Hardware Address (h32)
Sender Hardware address (l16)
Sender logical address(h16)
Sender logical address(l16)
Target Hardware address(h16)
Target Hardware Address(l32)
Target logical address

Figure 4-6 ARP packet format on Ethernet

Because Eth1394 is required to give the exactly same look as normal Ethernet, the 1394ARP packet is converted to a standard ARP packet before it can be delivered to the IP layer, meanwhile, the FireWire layer records down the information that is needed by itself, i.e. max_rec and sspd, see Figure 4-7.

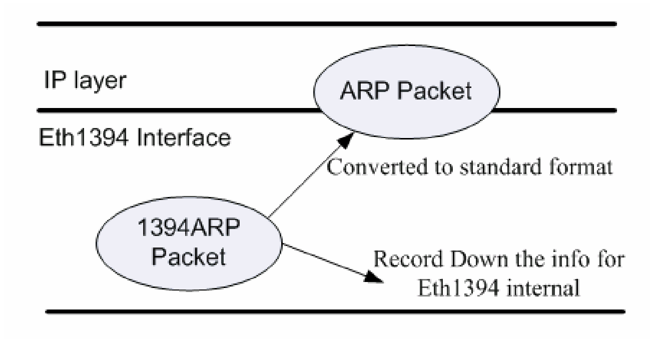

IP layer
ARP Packet
Eth1394 Interface
Converted to standard format
1394ARP Packet
Record Down the info for Eth1394 internal

Figure 4-7 Handling ARP packet in Eth1394

4.2.5 Implementation of Eth1394

This sub-section gives a summary of the implementation of Eth1394. It covers the unicast and broadcast transaction of Eth1394.

Unicast Transaction

The implementation of unicast transaction is based on FireWire's asynchronous transaction. During the initialization of the interface, a certain piece of address area on the FireWire is allocated. Then a handler (Eth1394_write) is installed to handle all write transactions into that address area (For transactions between Eth1394 modules, only write transaction is used.).

Broadcast Transaction

The implementation of broadcast transaction is based on FireWire's isochronous transaction. During the initialization of the interface, a certain isochronous channel is allocated. Then a handler (Eth1394_iso) is installed to handle all packets transmitted through that channel.

4.3 Stacking RTnet over RT-FireWire

4.3.1 Introduction about RTnet

RTnet provides a customizable and extensible framework for hard real-time communication over Ethernet. Conceptually similar to RT-FireWire, RTnet also employs static memory management, real-time interrupt handling, non real-time/real-time transaction differentiation to implement a basic real-time stack. The stack overview of RTnet is shown in Figure 4-8. In next section, the application programming interface based on RTnet is presented. For other features of RTnet, one can refer to [RTnet, 2005] and [Kiszka, Zhang et al, 2005]. The latter is attached to this report as appendix.

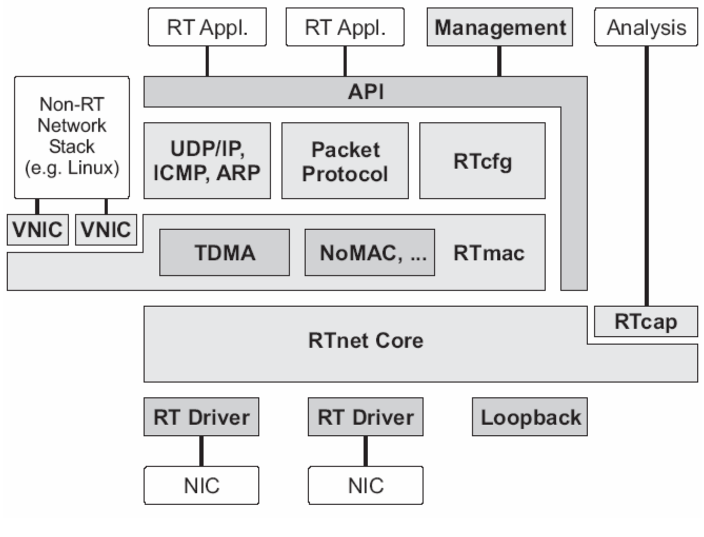


Figure 4-8 RTnet Stack Overview[Kiszka, Zhang et al 2005]

4.3.2 Application Programming Interface based on RTnet

One important reason to port Eth1394 to RTnet is that, the application programming interface on RTnet can thereby directly be used on FireWire.

RTnet provides its real-time services via real-time variants of POSIX-conforming socket and

I/O interfaces. This socket interface offers UDP and packet sockets for exchanging user data deterministically. Just as RTAI, RTnet permits both the classic kernel mode and more convenient user mode usage (Linux processes) of the interfaces. In this project, applications on RT-FireWire are deployed mainly by using the real-time socket interface via Eth1394 module.

4.3.3 Media Access Control

As important as a real-time-capable stack implementation is a deterministic communication media. But compared to specifically designed field-bus, e.g. CAN, FireWire's native media access control does not support prioritized transactions. In other words, packets from real-time and non real-time applications can not be differentiated on the Link-Layer level. Therefore, there exists the possibility that transactions from node running non real-time operations can block the transactions from node running real-time operations. To solve this problem, there is a need to add extra media access control layer above that native one.

RTnet addresses this demand with its RTmac (Real-Time media access control) layer. RTmac is designed to be a socket, where all customizable media access disciplines can be plugged in. Here the already-developed TDMA discipline is introduced, based on which the problem stated above can be solved.

RTnet's TDMA is a master-slave protocol. It synchronizes the clocks of all nodes in network. By assigning time slot to different node, which is actually determined by the offset relative to the synchronization messages the master nodes issues periodically, the transmission time of packets from each node can be explicitly separated. See Figure 4-9 for the illustration.

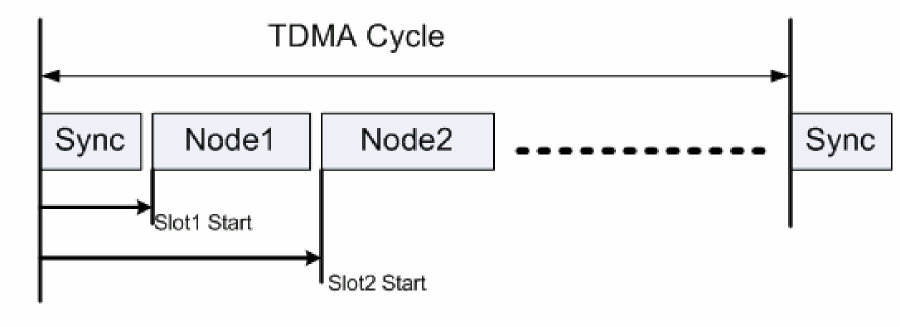


Figure 4-9 TDMA cycle in RTnet

By applying TDMA principle to RT-FireWire nodes through Eth1394, the transaction issued by non real-time nodes can be bounded to certain slots in the TDMA cycle, which means they can not influence the transaction issued by real-time nodes.

4.4 Test Bench

4.4.1 Bench Settling and Measurement Results

Based on the RTnet interface, a test bench is built up between two FireWire nodes, to measure the roundtrip between them. The whole procedure is shown in Figure 4-10, where the "server" node receives the data that is sent by "client" and sends it back. The roundtrip latency is thereby measured on the "client" side, i.e. the time between sending the data and receiving it.

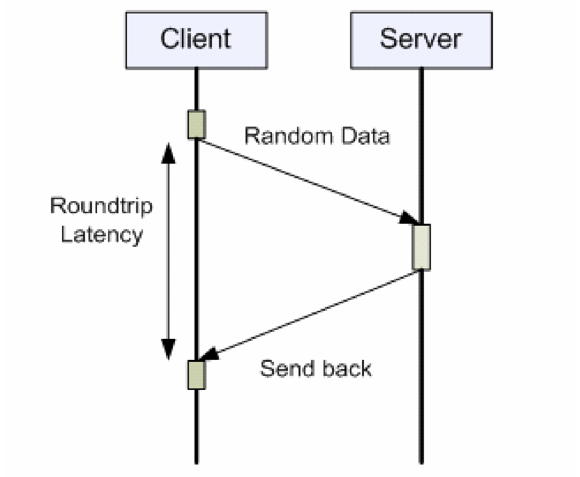


Figure 4-10 Test bench on Eth1394

Since both Eth1394 and Ethernet appear as the same medium under RTnet interface, the same experiment is repeated on Ethernet also.

The same hardware mentioned in Chapter2 is reused in this test bench. All FireWire transactions run at 400Mb/s, while Ethernet transactions run at 100Mb/s.

First situation in experiment is when both sides are not loaded. The roundtrip latency (97% threshold and 100% worst case) over ascending data loads on both Eth1394 and Ethernet are plotted together in Figure 4-11. Second situation in experiment is when both sides are fully loaded. The roundtrip latency (97% threshold and 100% worst case) over ascending data loads for both are plotted together in Figure 4-12.

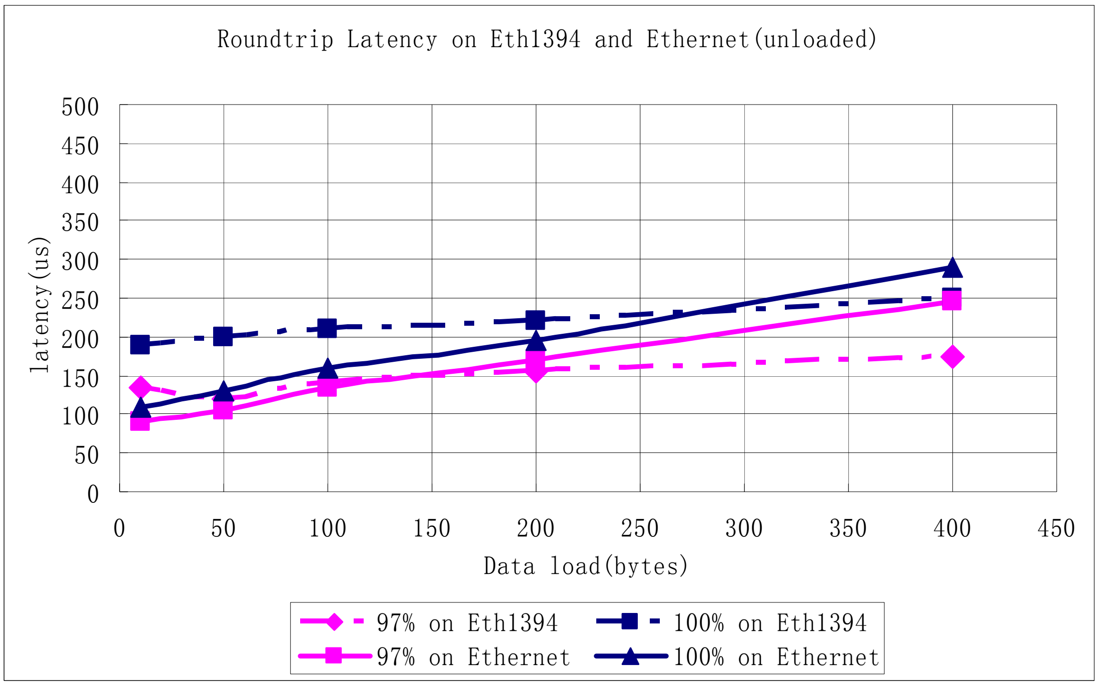


Figure 4-11 Roundtrip Latency on Eth1394 and Ethernet (when system is not loaded)

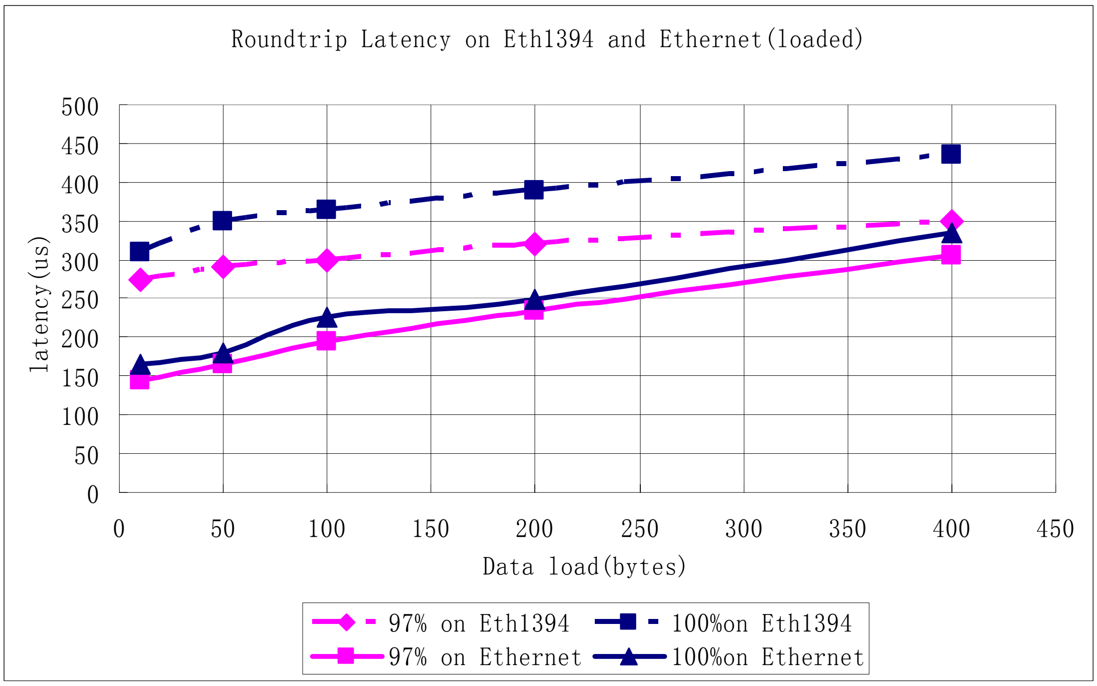


Figure 4-12 Roundtrip Latency on Eth1394 and Ethernet (when system is loaded)

4.4.2 Discussion

- The latency variation (jitter), i.e. the difference between 97% threshold latency and worst case latency on Eth1394 is larger than on Ethernet. This is due to the complex software stack under Eth1394. The whole RT-FireWire, which is under Eth1394, includes more task handover, context switches, etc, due to the layered structure. Instead, in Ethernet, only a driver layer is under the device interface.
- Also due to the complex layered software under Eth1394, the latency is more worsened by system load noise, compared with the influence on Ethernet.
- The latency on Eth1394 gives a less leaning slope, compared with Ethernet's slope. This is due to the high data transfer rate on FireWire. I.e. the FireWire in experiment can transfer data at 400Mb/s, while the Ethernet in experiment can only transfer at 100Mb/s.

5 Integration to Design Toolchain and Demonstration

5.1 Introduction

This chapter presents the integration of the real-time networking support provided by RT-FireWire into a complete working around, which is used for designing and verification of controllers. The toolchain covers the whole procedure from the design and simulation of a certain control system to the distributed deployment of that system to multiple computing boxes. To demonstrate the utility, a practical case is used, which is also introduced in this chapter. Also the performance measured from that practical case is presented, based on which the comparison between distributed control and centralized control is done.

5.2 Integration to the Design Toolchain

5.2.1 MSC Toolchain

MSC (Mechatronic Stack Connection) toolchain is a set of tools developed by [Buit, 2005] to facilitate the procedure of realizing controllers in software code, and deploying the controllers to computing boxes. An abstracted working sequence of this toolchain can be seen in Figure 5-1.

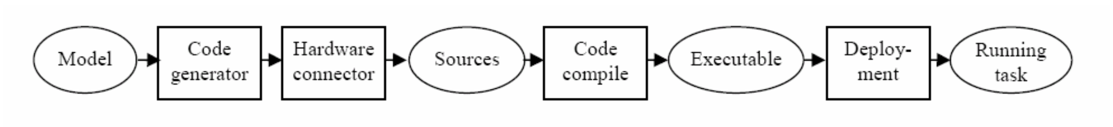


Figure 5-1 Working Sequence of MSC toolchain [Buit, 2005]

The strength of MSC toolchain is it takes care of the hardware I/O (Physical Input and Output) connection automatically. But in the project of [Buit, 2005], deployment of controller is only limited to centralized realization. Because of that, a solution needs to be found to enable the toolchain to deploy controllers into multiple boxes, i.e. distributed control.

5.2.2 Adding Networking Support to MSC Toolchain

To add networking support to MSC toolchain, only one phase in Figure 5-1 needs to be modified. That is the Code Generation. As shown in Figure 5-2, the configuration decision has to be made after the control model is ready from simulation tool, but before the code is generated. The designer has full freedom to choose the configuration. In case one has a 2-box network, 3 configurations are available.

In Figure 5-3, the 2-way configuration is shown where the controller and I/O are totally separated on two boxes. By using 2-way configuration, one introduces round trip network latency to the controller realization.

Figure 5-4 gives the 1-way configuration, where the controller stays with either Encoder input or PWM output. By using 1-way configuration, one introduces only single trip network latency to the controller realization.

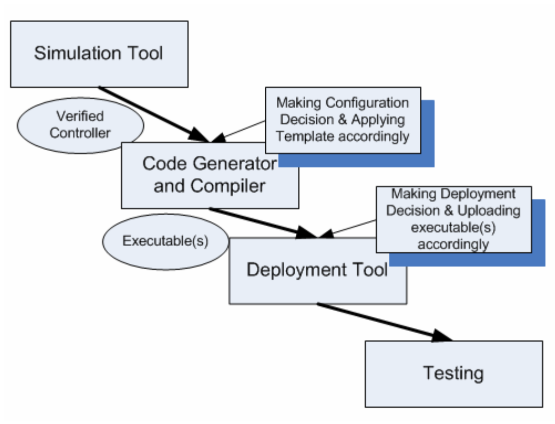

Simulation Tool
Verified Controller
Making Configuration Decision & Applying Template accordingly
Code Generator and Compiler
Making Deployment Decision & Uploading executable(s) accordingly
Executable(s)
Deployment Tool
Testing

Figure 5-2 Working Sequence of Modified Toolchain

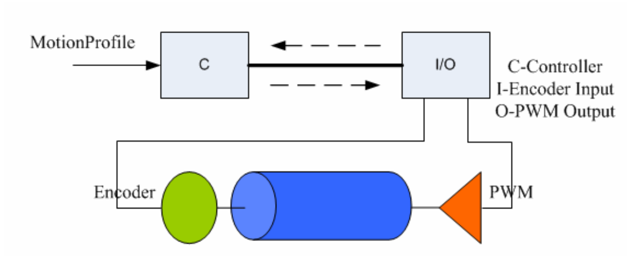


Figure 5-3 2-way Configuration

To realize a distributed controller, a network interface needs to be implemented. In this project, this is implemented by using socket programming. The reasons are:

- Interface to IP network has been implemented on RT-FireWire
- Socket interface has been fully adopted in RTAI/fusion, so the interface can be fully compatible with real-time application.
- By using socket interface, Ethernet can replace FireWire without any change in the application program, which gives a lot of space for introducing other real-time features, e.g. fault tolerance upon communication channel failure.

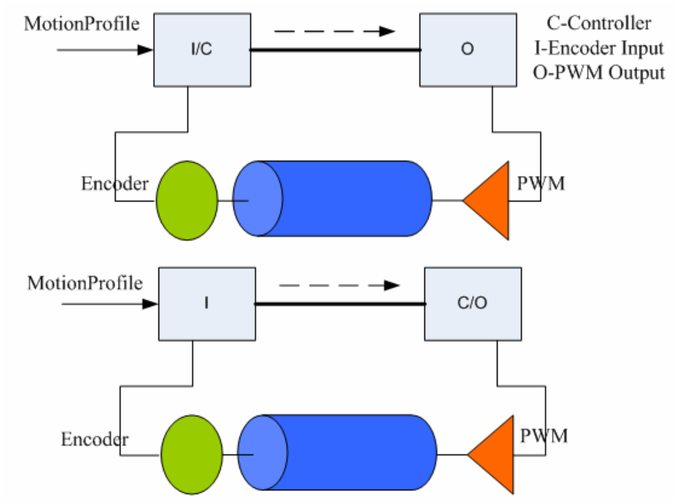


Figure 5-4 1-way Configuration

5.3 Demonstration

This section describes the demonstration setup that is used in this project to test the utility of RT-FireWire in a practical case. A real motor plant and a proved PID controller are used, which are described in next two sections respectively. The experiments are carried out using different sampling frequencies, i.e. 1 KHz and 5 KHz. In both cases, centralized control is also tested, as a comparison with distributed control. For distributed control, only 2-way configuration is tried out.

5.3.1 Plant and Controller

A demo setup built in Control Engineering Group is used, which is called LINIX, as shown in Figure 5-5.

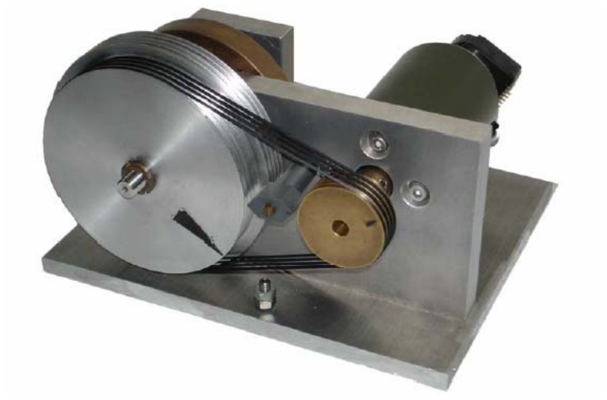


Figure 5-5 LINIX plant

Reason to choose LINIX:

- It is fully ready to be used.
- It is relatively simple, and straightforward.
- Many previous students also used the plant in their projects, which proves it is quite useful for demonstration.

The 20Sim model of LINIX is shown in Figure 5-6.

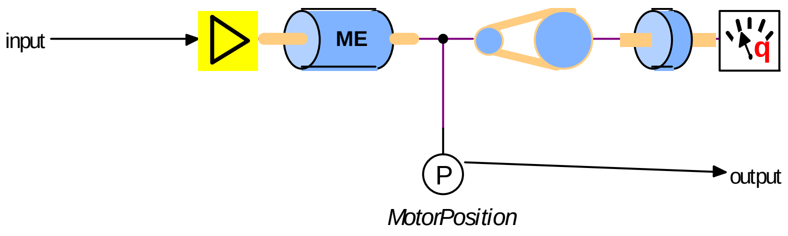


Figure 5-6 20Sim model of LINIX Plant

The models of the system and of the controller are given in Figure 5-7 and Figure respectively.

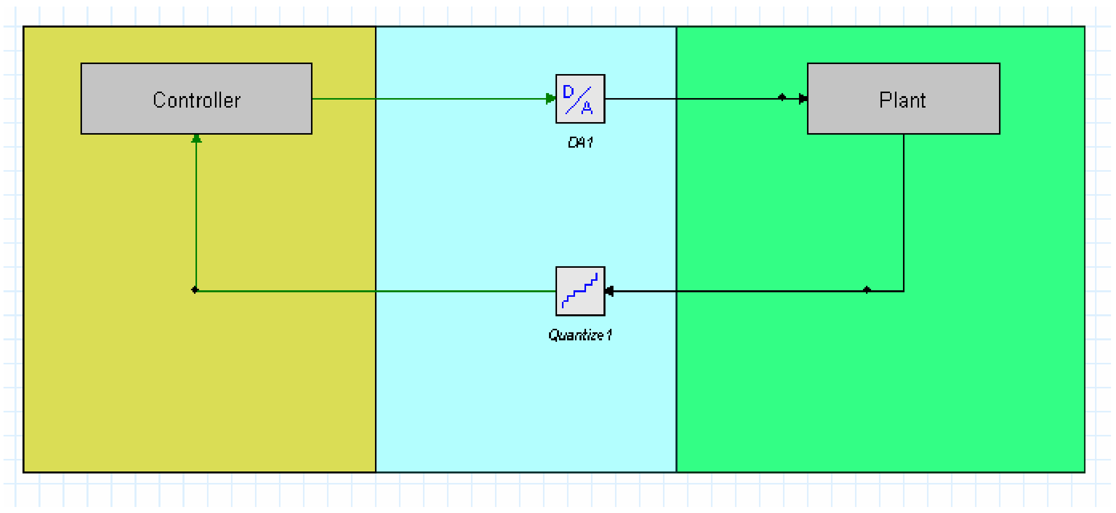


Figure 5-7 Demo System Model

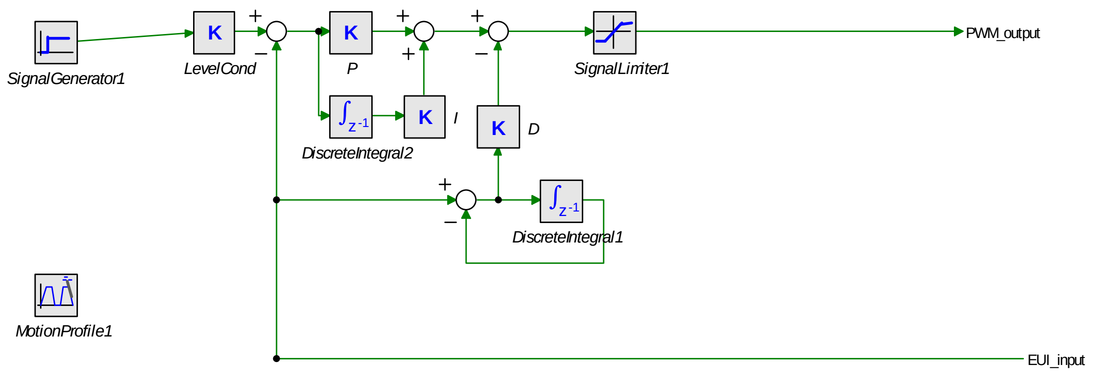


Figure 5-8 20Sim model of Controller

Figure 5-9 shows the simulation results while a motion profile is applied. Both PWM output and Encoder input signals are measured. The simulation frequency is 1 KHz.

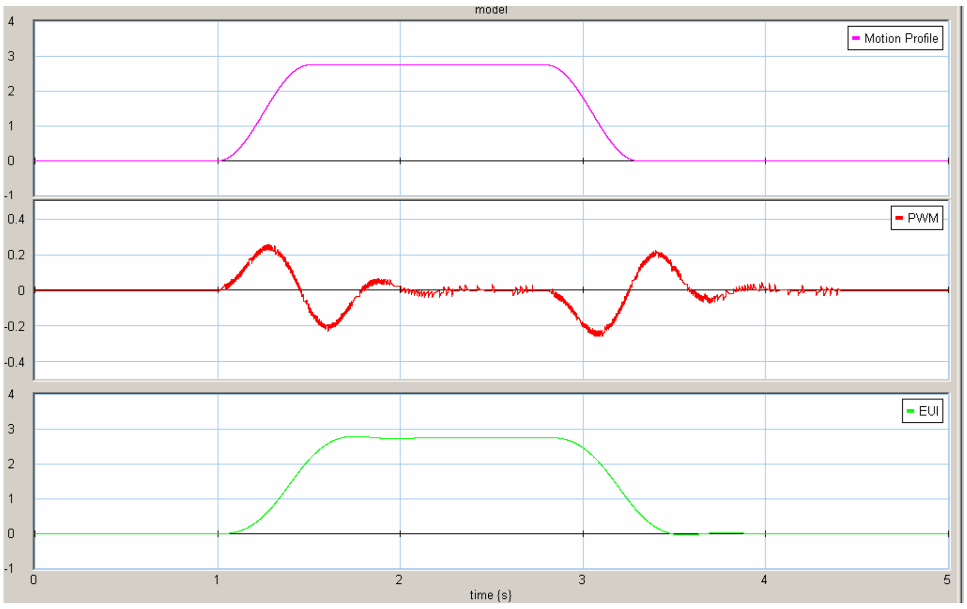


Figure 5-9 Simulation Results from 20Sim (1 KHz)

5.3.2 Performance Comparison

First, the controller is running at 1 KHz. In both centralized control and distributed control, the PWM output and Encoder Input are recorded, which is given in Figure 5-10.

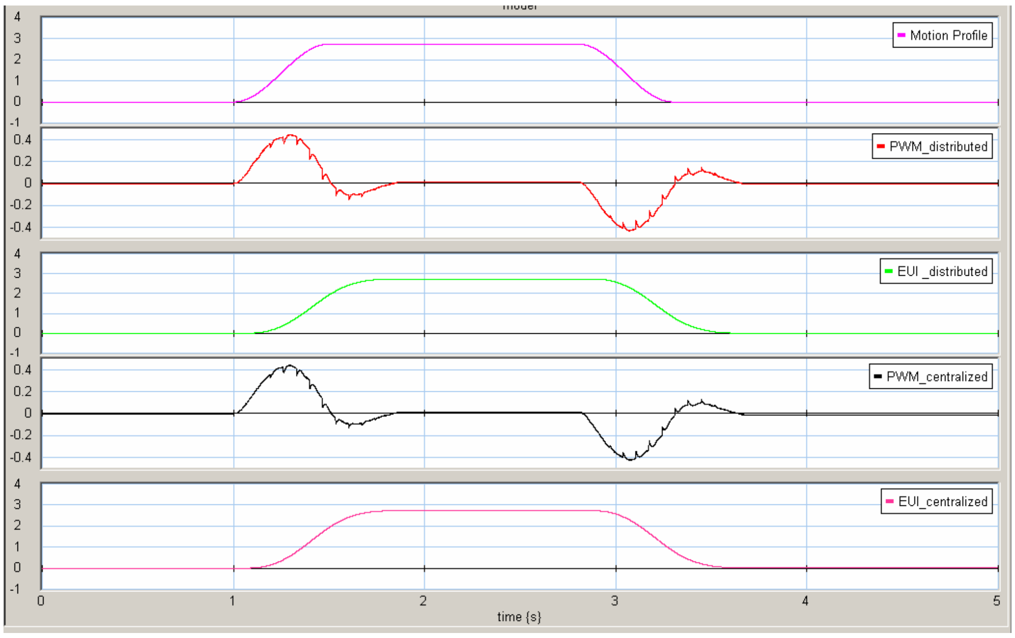


Figure 5-10 Comparison between Centralized Control and Distributed Control (1Khz)

Second, the controller is running at 5 KHz. The PWM output and Encoder input signal are recorded and shown in Figure 5-11. As can be seen, because of the relative high sampling frequency, the distributed controller is losing data due to the network latency. That can be seen from the difference of PWM output of both configurations. To make the different more obvious, Figure 5-12 plots them together.

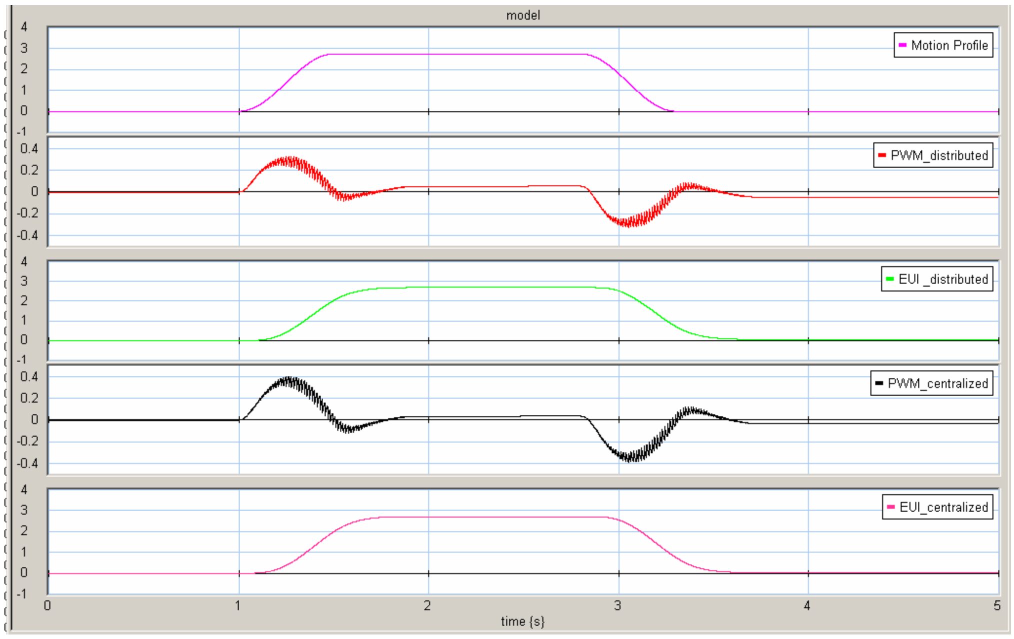


Figure 5-11 Comparison between Centralized Control and Distributed Control (5 KHz)

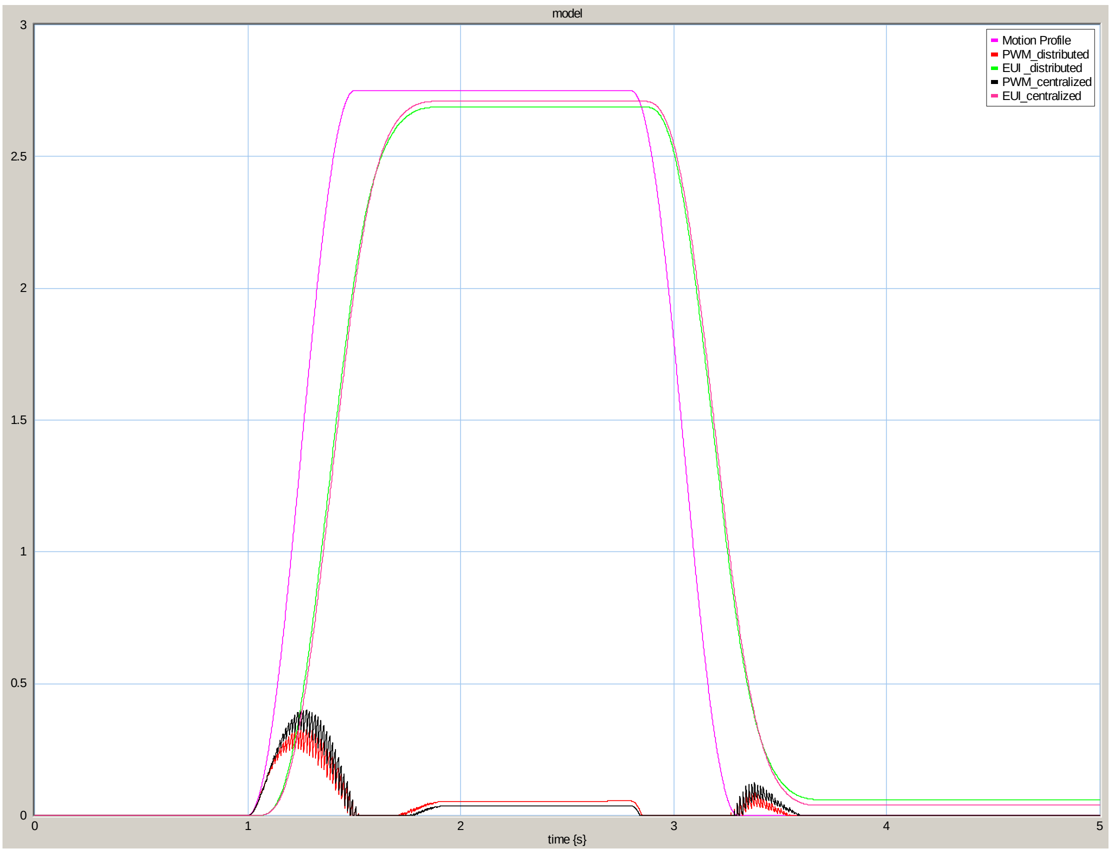


Figure 5-12 Comparison between Centralized Control and Distributed Control (5 KHz)

5.4 Discussions

Although some data is lost in the distributed configuration due to the latency on FireWire, the motion of the LINIX motor for both configurations is almost the same. This is due to the fact that LINIX motor (its dynamics) is a relatively slow. According to the simulation results, as shown in Figure 5-13 and Figure 5-14 , the LINIX motor gives the same step response when running at 1 KHz sampling frequency and when running at 5 KHz sampling frequency.

So to control a relatively slow plant like LINIX, 1 KHz sampling frequency is already enough. Increasing the sampling frequency to 5 KHz does not have any new influence, since the slow dynamics of LINIX plant is a dominating factor for the system reaction. Because of this, it can be concluded that FireWire is fully usable to control slow plant like LINIX, since it does not miss any deadline (in control intervals) when the sampling frequency is around 1 KHz.

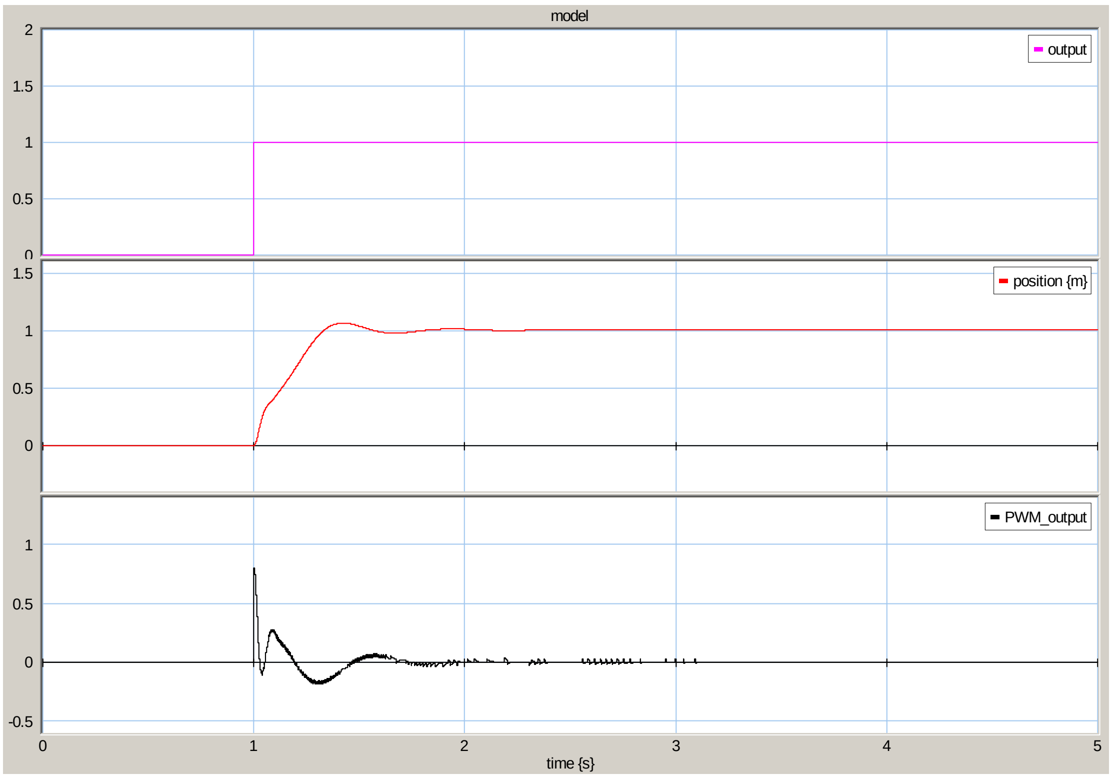


Figure 5-13 Step Response of LINIX in simulation (1 KHz)

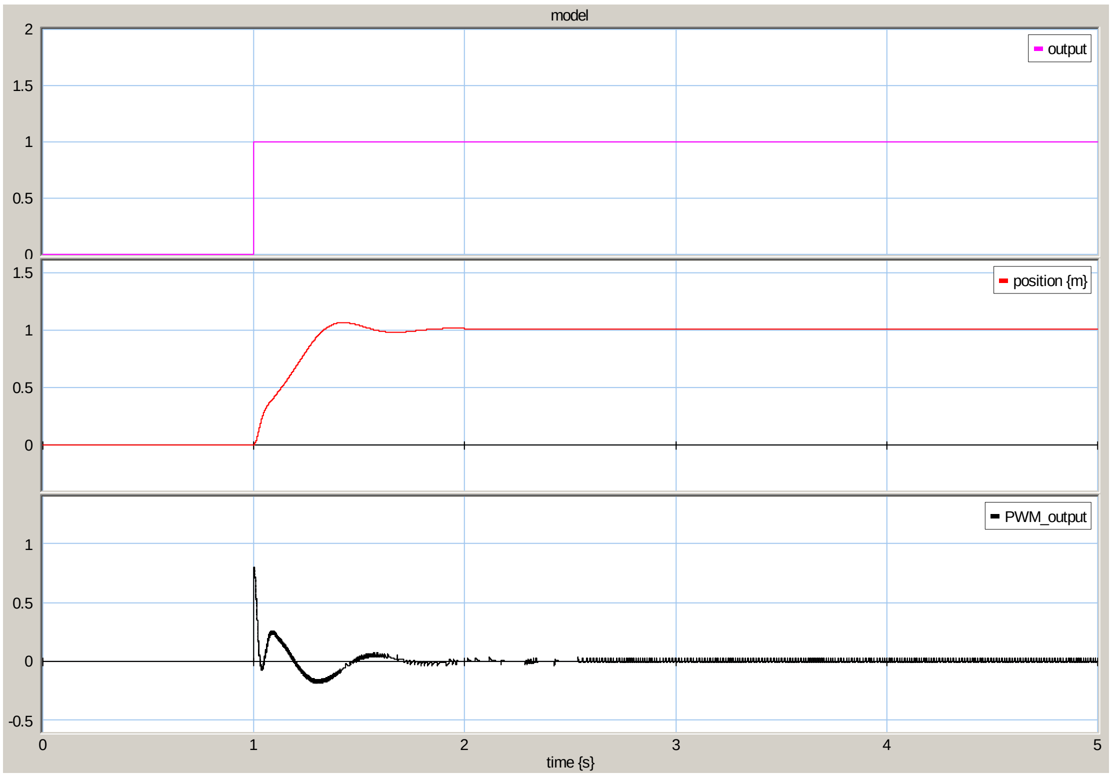


Figure 5-14 Step Response of LINIX in simulation (5 KHz)

6 Conclusions and Recommendations

6.1 Conclusions

Adoption of FireWire into Distributed Control

As to adopt FireWire as a new generation fieldbus for distributed control application, a software-based solution has been followed. The real-time software subsystem on FireWire (RT-FireWire), which currently works in Linux/RTAI (the real-time Operating System) has been fully designed and implemented in this project. The results from the performance benchmarking on RT-FireWire shows that, by applying RT-FireWire on FireWire hardware, the transaction latency on FireWire can be limited to a certain range that is fully usable for distributed control application, whether the system is under heavy load or not.

Real-Time IP over FireWire

Ethernet Emulation over FireWire (Eth1394) has been fully implemented on RT-FireWire as one highlevel module in the application layer. Via Eth1394, RT-FireWire can be connected to another real-time software framework, RTnet, which implements real-time networking on the IP layer. Therefore, besides Ethernet FireWire has been introduced as a new medium alternative for real-time IP networking. The performance benchmarking on Eth1394 and Ethernet shows that the performance from both is comparable.

Integration to Design Toolchain

Via Real-Time IP over FireWire, the real-time networking support provided by FireWire has been integrated to the design toolchain which covers the whole procedure from the design and simulation of a certain control system to the distributed deployment of that system to multiple computing boxes. As a result of this integration, the controller designed in current toolchain can be directly deployed to multiple nodes, as a simple but straightforward realization of distributed controller.

6.2 Recommendations

Short-term

Raw Interface on RT-FireWire layer

This is to develop a raw interface on RT-FireWire. So via this interface, operation can be directly applied on FireWire layer, e.g. issuing transaction, allocating bus address space or isochronous channels. The current “raw1394” module in Linux already implements the similar functions, but of course in a non real-time manner. Like the developing path of the whole RT-FireWire stack, “raw1394” in Linux can be the starting point. The developed interface, whether based on “raw1394” or not, should be conforming to the Real-Time Driver Model (RTDM) in RTAI/fusion. RTDM was originally developed by RTnet team, but now it has been fully integrated to RTAI/fusion as a new skin on the fusion nucleus. It extends the RTAI interface in a regular and well-defined way for providing device access. More information can be found in the RTAI mailing list. At the time of writing, all the IP-based protocols in RTnet have been ported to RTDM, so called protocol devices. Moreover, at the time of writing, another RTDM-conforming driver on InterBus adapter has been developed and announced in the RTAI community.

Media Access Control in RT-FireWire

During this project, media access control on FireWire is based on the implementation in RTnet. That is because on current stack, only the socket interface (via RTnet) has been fully ready. But in case the raw interface is ready, a media access control layer is desired to be built in RT-FireWire internal. The whole concept (even part of the implementation) of media access control module in RTnet can be moved to RT-FireWire. When this module is ready, the TDMA protocol can be applied directly on FireWire first. Besides, some other new protocols are also desired, probably with the implantation of a more complex algorithm.

Long-term

Stacking one or more middleware frameworks onto RT-FireWire

Nowadays, quite a few middleware frameworks for real-time control application have been developed or are under development. One example is CANopen, which has been developed by CAN in Automation organization as an application protocol and device model for the automation domain. If CANopen, or other middleware frameworks, can be stacked on RT-FireWire, it would enable automation applications to run straightly over RT-FireWire. Investigation on one or more specific middleware frameworks and a clear specification about the implementation should be done before starting the real work.

Porting New Hardware Drivers to RT-FireWire

During this project, only driver for OHCI-compliant adapter has been ported to RT-FireWire. It is desired that the driver for other non OHCI-compliant adapters can also be used under RT-FireWire. One step further, it would be very nice if the 1394b adapter (supporting 3.2 Gb/s) can be used under RT-FireWire.

Real-Time Vision Control over RT-FireWire

Real-Time vision control is control system using video signal input, e.g. via camera. During this project, some inquiries were received from the community about whether RT-FireWire supports real-time video transmission, e.g. from a FireWire camera. Due to the limitation of time and hardware, this topic was not opened. To develop real-time vision control over RT-FireWire, the current implementation of relative video data protocols (dv1394, video1394 module) in Linux can be studied and ported to RT-FireWire. Also the implementation should conform to RTDM.

Appendix1 Modification to the MSC toolchain

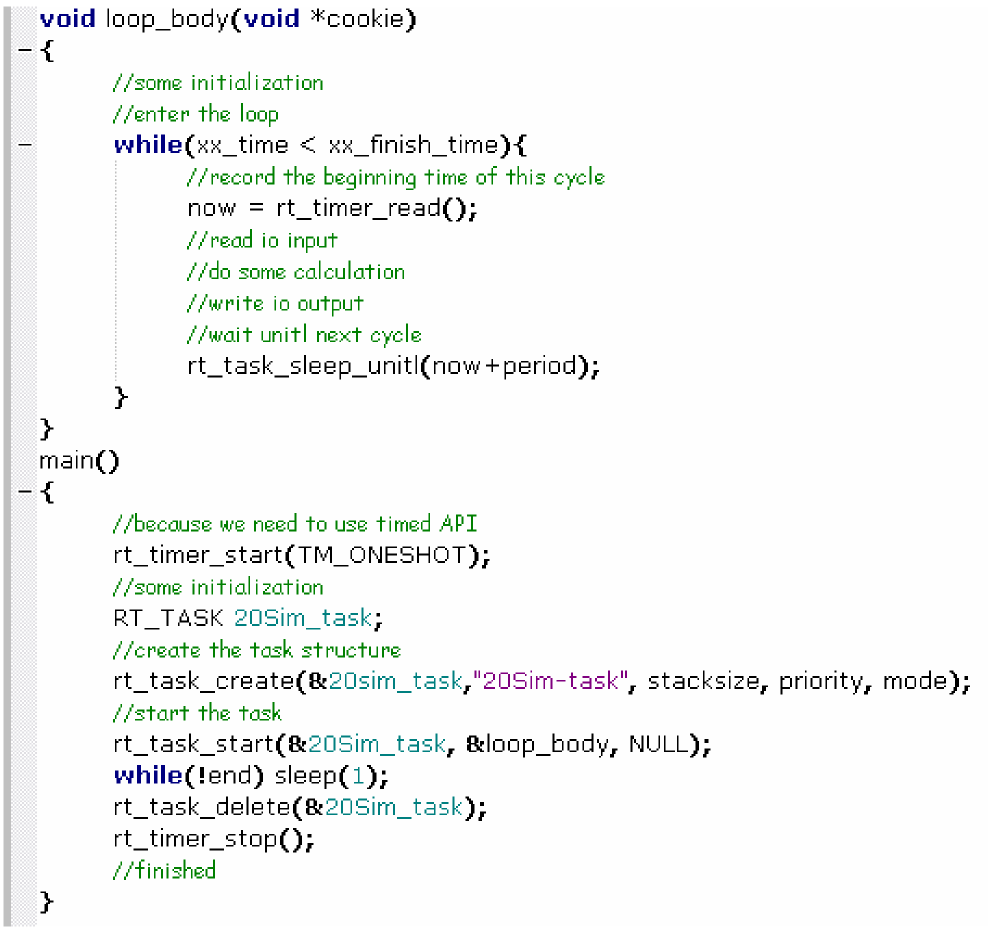
This appendix presents the modification made to the MSC toolchain [Buit, 2005]. First section describes the changes made to MSC toolchain for porting it to RTAI/fusion; second section describes the implementation of adding distributed controller deployment.

Porting to Fusion

In [Buit, 2005], LXRT on classical RTAI is used. But in this project, RTAI/fusion is chosen due to its better structured design and more extensibility. Compared with classical RTAI, Fusion uses a totally different mechanism for deploying real-time tasks in user space. Therefore, the relative function calls in MSC toolchain have been changed according to the Fusion standard. For a practical guide about how to use fusion API calls in user space, please refer to [RTAI, 2005].

Changes of the Code Generation Template

The code generation template is used when a certain simulation block in 20Sim is converted to C code. In MSC toolchain, the template is also used to deploy any generated 20Sim code to RTAI/Linux user space as a real-time task. Listing A0-1 gives an overview of initialization and execution of 20Sim task in the template.



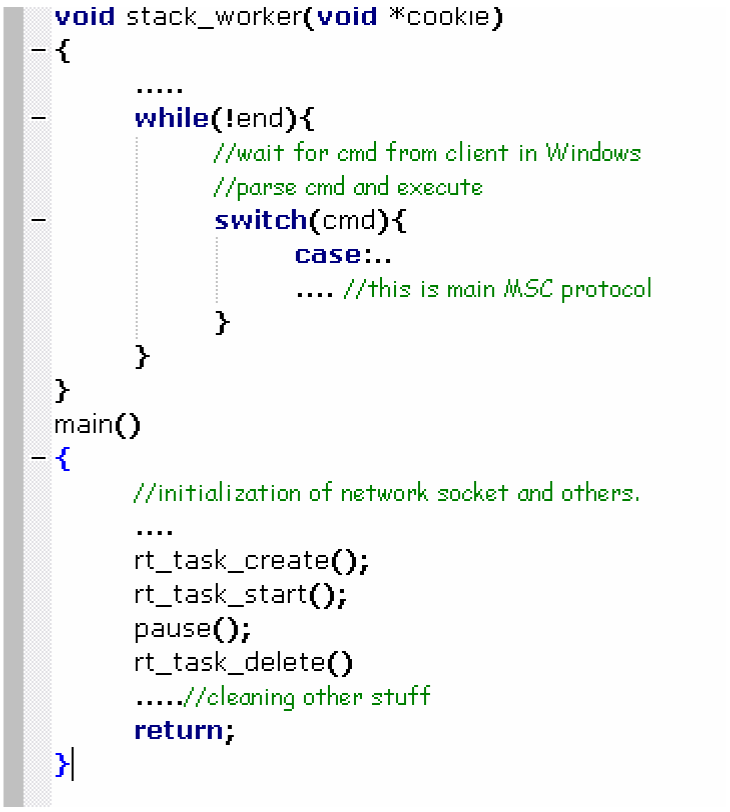
```
void loop_body(void *cookie)
{
        //some initialization
        //enter the loop
        while(xx_time < xx_finish_time){
                //record the beginning time of this cycle
                now = rt_timer_read();
                //read io input
                //do some calculation
                //write io output
                //wait unitl next cycle
                rt_task_sleep_unitl(now+period);
        }
}
main()
{
        //because we need to use timed API
        rt_timer_start(TM_ONESHOT);
        //some initialization
        RT_TASK 20Sim_task;
        //create the task structure
        rt_task_create(&20sim_task,"20Sim-task", stacksize, priority, mode);
        //start the task
        rt_task_start(&20Sim_task, &loop_body, NULL);
        while(!end) sleep(1);
        rt_task_delete(&20Sim_task);
        rt_timer_stop();
        //finished
}
```

Listing A0-1 Real-Time 20Sim Task in Fusion

- RT_TASK is the structure for the real-time task, which is the same in both kernel space and user space.
- Because a lot of API services involving timeouts, delays are used in the template, the *rt_timer_start ()* must be called first. The same as in classical RTAI, two modes can be chosen to start the timer: oneshot or periodic. In one shot mode, the underlying timer will be reprogrammed after each clock tick so that higher accuracy of timing can be gained, while in periodic mode, the timer will only be reprogrammed after each period, which can be specified as argument to *rt_timer_start()*. Therefore, the timer cost less time for reprogramming, but at the expense of lower accuracy.
- Here, *rt_task_sleep_until()* is used to wait for starting of next cycle. Another function, *rt_task_wait_period()* can be used for the same purpose, but the period of task must be set first by using *rt_task_set_period().*

Change to Stack Daemon

Because the stack daemon uses a lot of fusion API calls, most of which are not allowed in the Linux domain. Therefore, the whole stack daemon is moved to the real-time domain, as a low priority real-time task. See Listing A0-2.



```
void stack_worker(void *cookie)
{
        .....
        while(!end){
                //wait for cmd from client in Windows
                //parse cmd and execute
                switch(cmd){
                        case:..
                        .... //this is main MSC protocol
                }
        }
}
main()
{
        //initialization of network socket and others.
        ....
        rt_task_create();
        rt_task_start();
        pause();
        rt_task_delete()
        .....//cleaning other stuff
        return;
}
```

Listing A0-2 Stack Daemon Task in Fusion

Connection Objects

In MSC toolchain, a lot of connection objects are used between the 20Sim task to the stack daemon, see Figure A1-1, which shows all the shared memory and semaphores used in MSC toolchain on fusion. Mutex is new primitive in fusion, used to synchronize concurrent access to shared resource, e.g. shared memory here. Its antecedent is Resource Semaphore in classical RTAI. All the connection objects are created by 20Sim task using rt_xxx_create() functions. Later on, the stack daemon should call the rt_xxx_bind() functions to find all these objects from the global registry.

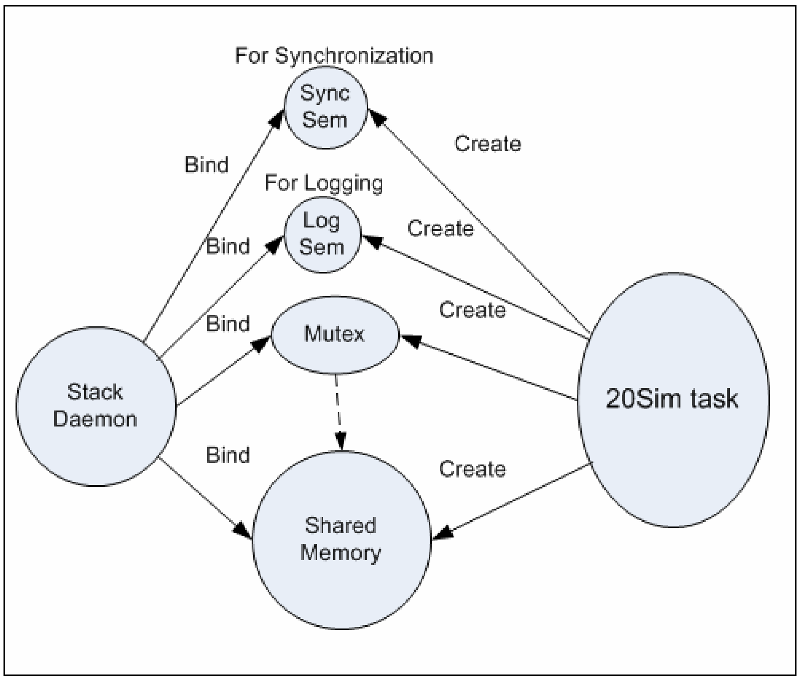
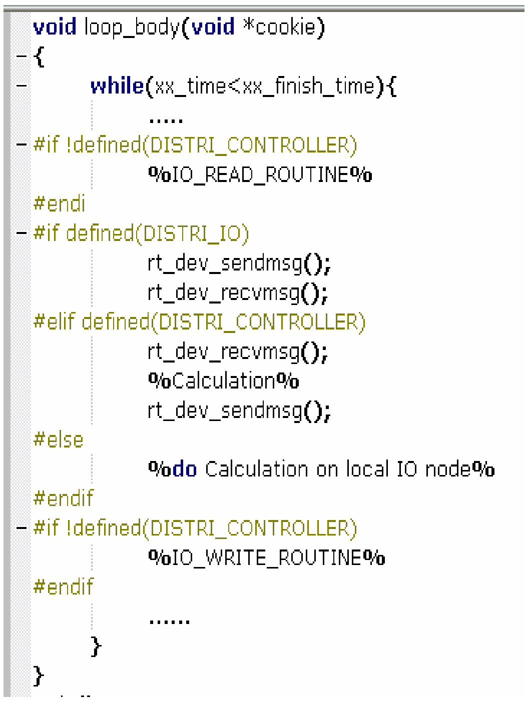


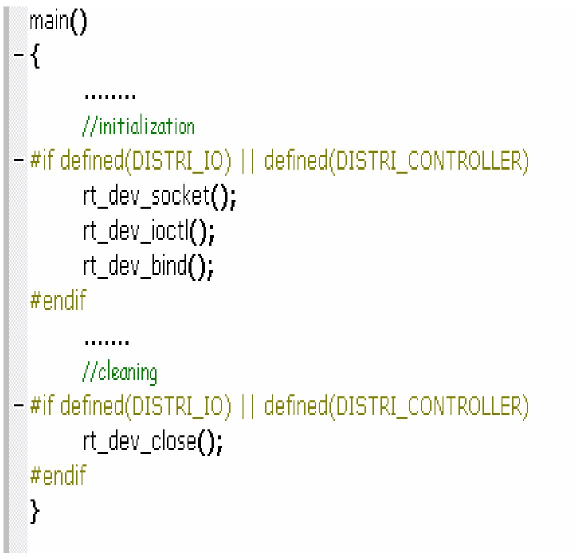
Figure A0-1 Connection Objects in MSC Toolchain on Fusion

Adding the Distributed Controller Deployment

Main change for this purpose goes to the code generation template, see Listing A0-3.



```
void loop_body(void *cookie)
{
        while(xx_time<xx_finish_time){
                .....
#if !defined(DISTRI_CONTROLLER)
                %IO_READ_ROUTINE%
#endi
#if defined(DISTRI_IO)
                rt_dev_sendmsg();
                rt_dev_recvmsg();
#elif defined(DISTRI_CONTROLLER)
                rt_dev_recvmsg();
                %Calculation%
                rt_dev_sendmsg();
#else
                %do Calculation on local IO node%
#endif
#if !defined(DISTRI_CONTROLLER)
                %IO_WRITE_ROUTINE%
#endif
                ......
        }
}
```



```
main()
{
        ........
        //initialization
#if defined(DISTRI_IO) || defined(DISTRI_CONTROLLER)
        rt_dev_socket();
        rt_dev_ioctl();
        rt_dev_bind();
#endif
        .......
        //cleaning
#if defined(DISTRI_IO) || defined(DISTRI_CONTROLLER)
        rt_dev_close();
#endif
}
```

Listing A0-3 Distributed Controller Support in Template

The implementation of network interface is based on real-time variant of socket programming, which is in the API of fusion.

Since only the “2-way” configuration is used in this project, two “define”s, i.e. DISTRI_IO for IO node and DISTRI_CONTROLLER for controller node, are enough to tell the compiler for which node the code is being complied for. Accordingly, there is one CCE config file for each node.

Appendix 2 Non Real-Time Factors in Linux FireWire Subsystem

This appendix unveils part of the non real-time factors in Linux FireWire Subsystem

Layered Architecture and Task Handover Latency

The Linux FireWire Subsystem is organized hierarchically into different layers, with each layer containing one or more components to implement the corresponding network protocol or other specific functionalities on that layer. Roughly, it consists of 3 layers, which is presented in Figure A2-0-1.

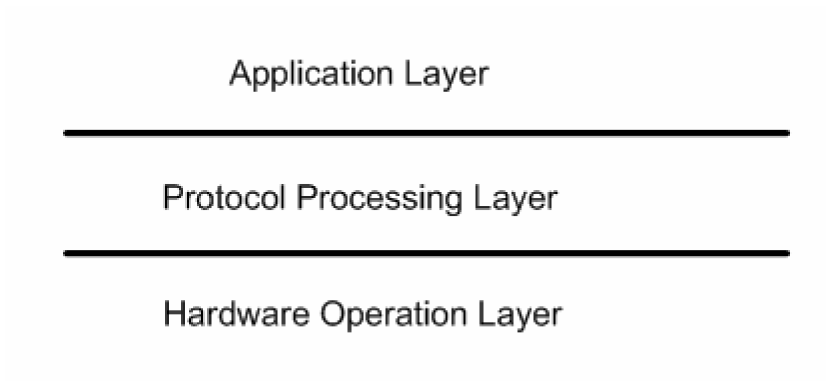


Figure A2-0-1 Layers in Linux FireWire Subsystem

For each incoming packet, the processing task starts from Hardware Operation Layer (i.e. fetching the packet from DMA –mapped memory), and ends in the Application Layer (i.e. the application does some specific job on/according to the packet). Here, no attempts are made to explore the details in each layer, but attentions are paid to the Task Handover mechanisms between the layers. Task Handover is the way that processing routine in one layer wakes up the processing routine in another layer, and later ends itself, so that the packet processing task continues in another layer.

In Linux, there are quite a few variant implementations for the Task Handover:

- Software IRQ
- Tasklet
- Obsolete Bottom Half struct (only in 2.4 kernel and before)
- Kernel Timer
- Semaphore
- Waiting Queue
- …..

Since these are really deep Linux kernel internals, it is beyond the scope of this appendix to give explanations of them. Please refer to chapter 6 of [Rubini, 2001] for more knowledge about their definitions and usages. Here, the focus is only on the deployment of the Task Handover in FireWire Subsystem and the latency introduced by it.

First, the deployment of task handover in Linux FireWire Subsystem is presented in Figure A2-0-2. In Hardware Operation Layer, the routine is broken into 2 parts: Top Half and Bottom Half. The Top Half runs to acknowledge the interrupt, identify the hardware device raising the Interrupt, check for data or status on the I/O device and return as quickly as possible to avoid missing any new interrupts. Then, the Bottom Half is scheduled at some later time by using Tasklet, to complete the service of the Interrupt or to do the actual work required to service the hardware device.

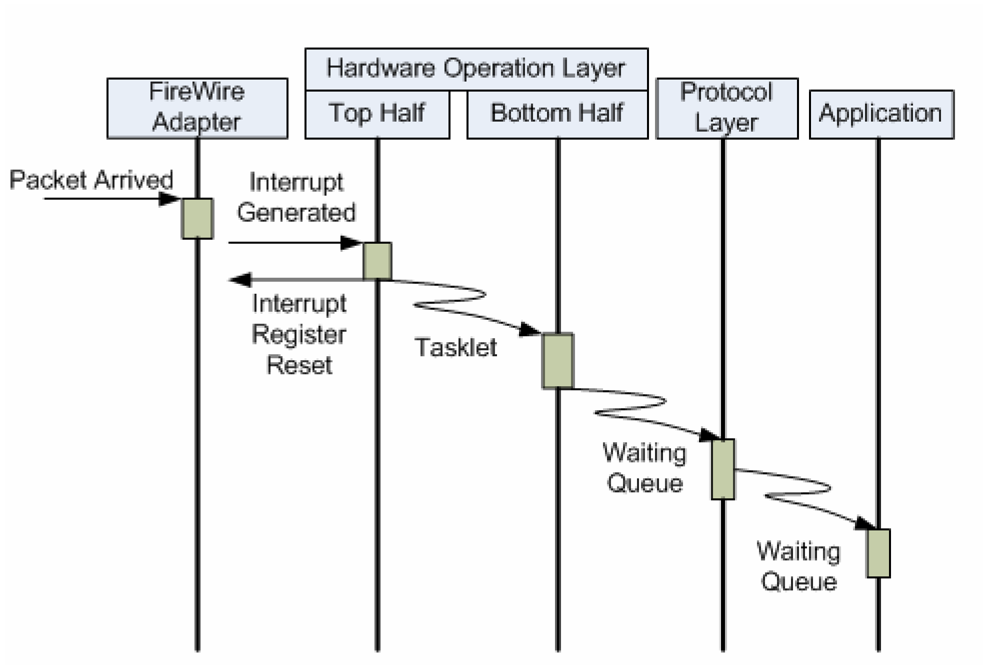


Figure A2-0-2 Deployment of Task Handover in Linux FireWire Subsystem

In next section, the measurement results about the Task Handover latency in Linux FireWire Subsystem are given.

Measuring of Task Handover Latency in Linux FireWire Subsystem

A specific series of experiment is carried out to measure the task handover latencies between layers. Because the Task Handover between Hardware Operation Layer and Protocol Layer uses the same mechanism as the one between Protocol Layer and Application Layer, so only the latter is measured. As can be seen from Figure A2-0-3 and Figure A2-0-4, when the system is not loaded, the task handover latency in both cases have already shown a relatively big difference in latency values in the critical range of cumulative percentage (e.g. between 97% and the worst case (100%) performance). With adding load, performance is clearly worsened. What’s more when the system is heavily loaded, the curve is much less steep then in the case system is not heavily loaded. As already discussed, this indicates increased non-determinism and results in poorer real-time properties.

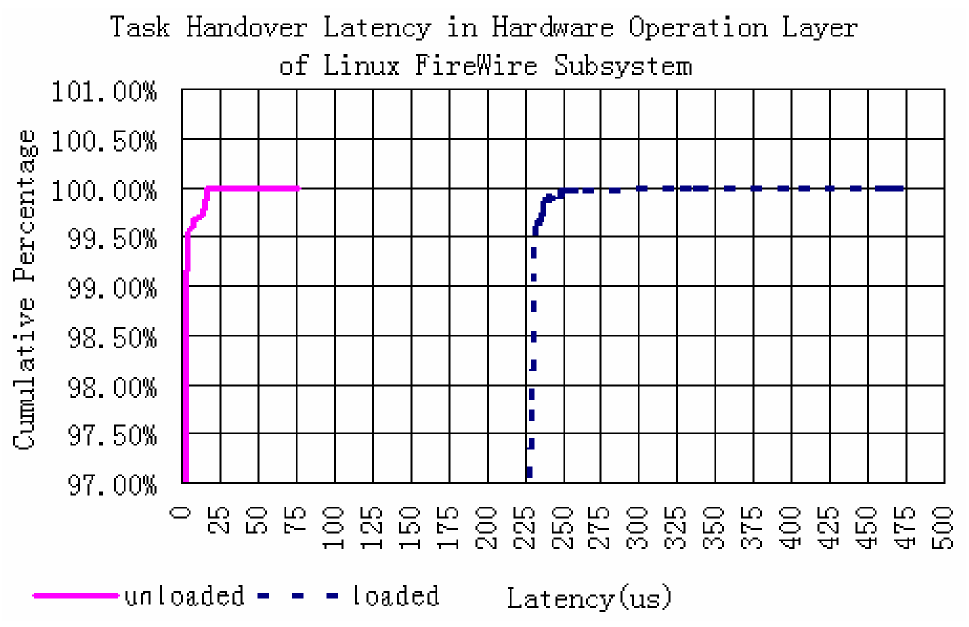


Figure A2-0-3 Task Handover Latency in Hardware Operation Layer of Linux FireWire Subsystem

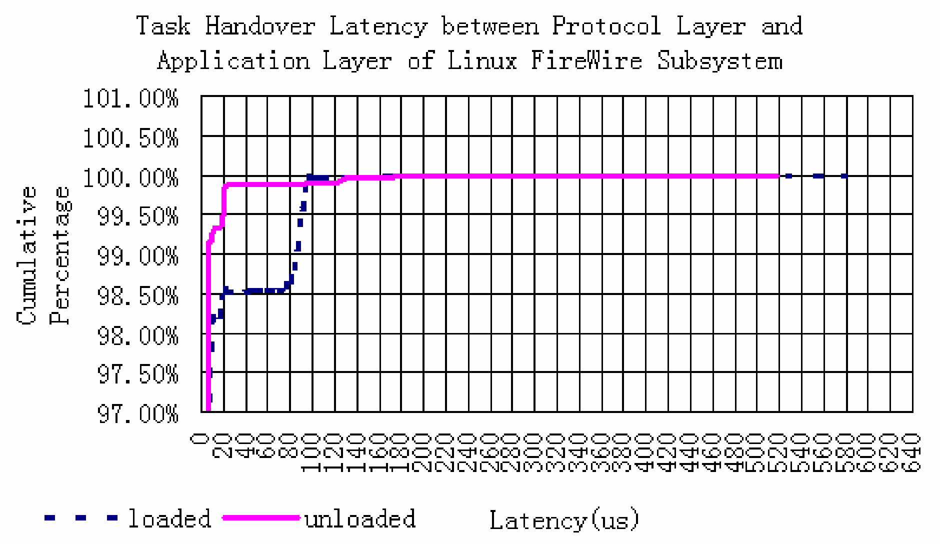


Figure A2-0-4 Task Handover Latency between Protocol Layer and Application Layer of Linux FireWire Subsystem

Conclusion

In short, these experiment results give a clear proof that the Linux FireWire Subsystem can not be used in real-time because of its internal software architecture and Task Handover mechanisms.

Appendix 3 Practical Information about RT-FireWire

Project Location

The Open Source project RT-FireWire is located at Berlios (www.berlios.de). The exact address of RT-FireWire is http://developer.berlios.de/projects/rtfirewire/ . At the time of writing, homepage for RT-FireWire is maintained in author's own web space, but it can be accessed via (rtfirewire.berlios.de).

Requirements

- Linux kernel 2.6.x
- RTAI/fusion 0.9 or newer (at the time of writing, it is still the cutting-edge CVS version)
- X86 platform
- FireWire card(s). (at the time of writing, only the driver for OHCI compliant FireWire card has been ported to RT-FireWire)
- Download latest RT-FireWire package from the project homepage.

Installation

1. Install and test suitable version of RTAI/fusion
2. *cd* to preferable directory (e.g. /usr/src)
3. *tar xvjf <PATH-TO-RT-FireWire-ARCHIVE>/rt-firewire.tar.bz2*
4. *cd rt-firewire*
5. Run *./configure --with-rtai=<PATH-TO-RTAI> <options> [--prefix=<PREFIX>]*
   <PATH-TO-RTAI> is installation directory of RTAI/fusion.
6. *<PREFIX>* is the installation path prefix (see below). Default *<PREFIX>* is */usr/local/rt-firewire*. The complete list of parameters is shown when calling *./configure --help*. RT-FireWire can also be build out-of-tree, just run configure from a newly created directory.
7. *make*
8. *make install*
   This will create the directories *<PREFIX>/sbin* with all configuration tools, *<PREFIX>/modules* containing all core modules, *<PREFIX>/include* with the required API header files.
9. In case the char device */dev/rt-firewire* is not created automatically, you can create it manually, by *mknod /dev/rt-firewire c 10 241*.

Initialization

1. Shutdown your FireWire card and unload the Linux driver
2. Load the RTAI/fusion modules: *rtai_hal.ko rtai_nucleus.ko rtai_rtdm.ko*
3. Load the RT-FireWire modules: *rtpkbuff.ko* (real-time buffer module),
   rt_serv.ko (real-time server module)
   rtpc.ko (real-time procedure call module)
   rt_ieee1394.ko (RT-FireWire kernel module)
   rt_ohci1394.ko (OHCI driver)
   bis1394.ko (bus internal service module)

First play-around

After *insmod* all the modules, you can run *hostconfig* now to see all your FireWire adapters, For example:

```
===========
hostconfig -a
===========
```

This gives a view of all local hosts.

If you have 2 machines, you can run *rtping* to test the latency between request and response. For example:

```
======================
rtping -h fwhost0 -d 0 -s 50.
======================
```

This does a test between local host "fwhost0" and remote node that has a node id 0. If the id of local host is used, then it is a loop back test. The value after “-s” specifies the size of data load for this test.

Appendix 4 Publication to 10th IEEE International Conference on Emerging Technologies and Factory Automation

In this appendix, the paper <<RTnet - A Flexible Hard Real-Time Networking Framework>> is attached. Jan Kiszka, the RTnet leader in University of Hannover, Germany, is the main author of this paper. FireWire and its integration to RTnet are introduced in this paper. At the time of writing, the conference has not been held. It will be held at 19-22 September 2005, in Facolta' di Ingegneria, Catania, Italy. This paper will be presented by Jan Kiszka during the conference, and will be published in the proceedings.

This paper is one product of the cooperation between RTnet development team in University of Hannover and the Control Engineering Group in University of Twente. The initial contact was established via the RTAI and RTnet community.

RTnet – A Flexible Hard Real-Time Networking Framework

Jan Kiszka, Bernardo Wagner
Institute for Systems Engineering
Real-Time Systems Group
University of Hannover, Germany
{kiszka, wagner}@rts.uni-hannover.de

Yuchen Zhang, Jan Broenink
Control Engineering Group
Department of Electrical Engineering
University of Twente, the Netherlands
y.zhang-4@student.utwente.nl,
j.f.broenink@utwente.nl

Abstract

In this paper, the Open Source project RTnet is presented. RTnet provides a customisable and extensible framework for hard real-time communication over Ethernet and other transport media. The paper describes architecture, core components, and protocols of RTnet. FireWire is introduced as a powerful alternative to Ethernet, and its integration into RTnet is presented. Moreover, an overview of available and future application protocols for this networking framework is given.

1 Introduction

Real-time Ethernet has grown to one of the core topics in current industrial automation research and application. A significant number of vendor-driven solutions have shown up on the market during the last years, claiming to replace traditional fieldbuses. The overview of available solutions on [18] currently lists 16 soft and hard real-time Ethernet variants. Most of them either require special hardware extensions to nodes or infrastructure components, or they provide only soft real-time guarantees. Academia approaches are typically designed to demonstrate specific concepts and lack common OS or hardware support. A broad overview of soft and hard real-time protocol research is given in [7]. Some recent approaches are for example FTT-Ethernet [16], RT-EP [12], or the combination of switches and traffic shapers [11].

All these approaches come with various transport and application protocols as well as programming interfaces, which are generally not compatible with each other. Additionally, there are other transport media beyond Ethernet 100Base-T approaching the real-time domain: Gigabit Ethernet, wireless media as IEEE 802.11 or Bluetooth, and also promising trends like using FireWire for time-critical control and measuring tasks. While this diversity of solutions can stimulate competition, it also interferes with the portability and extensibility of applications both in research and industrial scenarios. Furthermore, the question arises which solutions can guarantee long-term availability, especially when taking their specific hardware dependencies into account.

With the goal to provide a widely hardware-independent and flexible real-time communication platform, the RTnet project has been re-founded in 2001 at the University of Hannover, based on ideas and source code of a previous effort to provide deterministic networking [10]. RTnet is a purely software-based framework for exchanging arbitrary data under hard real-time constraints. The available implementation is founded on Linux with the hard real-time extension RTAI [17].

The design of the RTnet stack as depicted in Figure 1 was inspired by the modulised structure of the Linux network subsystem. It aims at scalability and extensibility in order to comply with the different requirements of application as well as research scenarios. RTnet's software approach addresses both the independence of specific hardware for supporting hard real-time communication and the possibility to use such hardware nevertheless when it is available. Furthermore, it enables the integration of various other communication media beyond Ethernet.

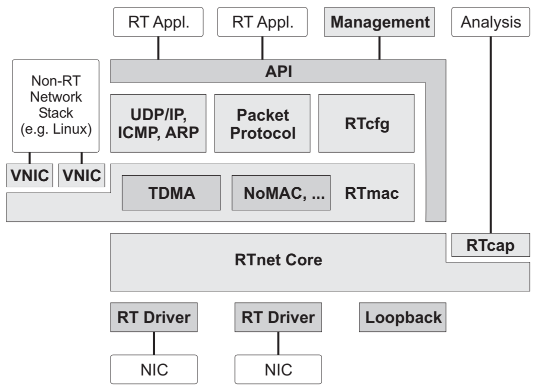


Figure 1. RTnet Stack

This paper presents the architecture of RTnet and the realisation of its central components. Section 2 describes the RTnet base services consisting of the stack core, the driver interface, available transport protocols like the real-time UDP/IP implementation, the programming interfaces provided to management tools and real-time applications, and the packet capturing extension RTcap. The deterministic media access control framework RTmac, including its tunnelling network devices for time-uncritical traffic (VNIC), is introduced in Section 3. That section will furthermore present RTnet's default access control discipline for Ethernet, TDMA, in details. Finally, Section 4 closes the stack overview by addressing the real-time configuration service RTcfg. So far, the implementation of RTnet has been focused on Ethernet. Section 5 presents the concepts and recent advances to add real-time IEEE 1394 (FireWire) support to the framework. The section also points out the advantages of that media type and the possible applications in the automation domain. Furthermore, available and future application protocols and full-featured middlewares working over RTnet are described in Section 6.

2 Base Services

RTnet contains a set of central services which are required for most scenario. In the following, these service will be introduced.

2.1 Packet Management

One of the crucial parts of RTnet deal with the management of packets which contain the incoming and outgoing data. Packets that ought to be transmitted are passed through the stack in the context of the sending task, i.e. a real-time application or an internal RTnet service. In contrast, incoming packets are first passed from the network controller driver to a so called stack manager. This real-time task demultiplexes the packet according to their protocol types by passing them to the respective handlers. The priority of the stack manager has to be above all applications using RTnet services in order to avoid priority inversions. This concept is similar to bottom-half interrupt handling as it can be found in most operating systems.

The stack and the drivers use a unified data structure called `rtskb` (derived from the Linux `sk_buff` structure) to handle packet buffers. While classic network stacks allocate such buffers and management structures dynamically, RTnet has to use a different scheme due to the real-time requirements. First, all `rtskb`s are preallocated during set-up. As currently RTnet does not support buffer sharing between multiple users, the management structure and the payload buffer are forming a single memory fragment. And second, every `rtskb` has a fixed size and can always carry the largest physical packet. This limitation is necessary to avoid shortages due to memory fragmentation and to allow exchanging of arbitrary `rtskb`s between users.

Packet producers and consumers within RTnet have to create pools of `rtskb`s in order to take part in the communication. During runtime, new `rtskb`s are allocated from these pools. A reference in the `rtskb` to its original pool allows to return it to its owner upon release. When a packet producer hands over a `rtskb` to the destined consumer, the ownership changes only if the consumer is able to provide a free compensation `rtskb` from its own pool. Otherwise the packet is dropped, and the related buffer can immediately be reused.

Typical producers and consumers are the adapter drivers on one side and the sockets on the other. But also VNICs or management protocols like RTcfg and ICMP provide their own pools. Pools are created or resized in non-real-time context using the indeterministic memory allocation service of the underlying operating system. In order to allow socket creation and pool extension also in real-time context, the required `rtskb`s are transferred in that case from a special global pool of preallocated buffers that has been created during the stack initialisation.

2.2 UDP/IP Implementation

Compared to a standard UDP/IP stack, several modifications were required to create the deterministic variant contained in RTnet. First, the dynamic Address Resolution Protocol (ARP) was converted into a static mechanism which is executed during the set-up. If a destination address is later unknown, no resolution requests are issued but a transmission error is returned to the caller. Otherwise, the worst case transmission latency of a packet would include the delay of a potential address resolution.

Second, the routing process was simplified. The output routing tables were optimised for the limited amount of entries used with RTnet. To accelerate the packet set-up, the tables also include the ARP results, i.e. the destination hardware addresses.

The defragmentation of IP packets needs special attention. In classic network stacks, this task is performed by the IP layer before any higher layers like UDP are involved. Thus, as the actual receiver is yet unknown, a global `rtskb` pool is required for buffering all fragments before the last one has arrived. The addition of new fragments to an existing chain demands a lookup in the global list of all currently pending IP packets chains. Furthermore, incomplete chains have to be cleaned up after a timeout to avoid buffer shortages and to keep the global IP fragment list small.

The UDP/IP stack of RTnet contains several mechanisms to confine the effects of the defragmentation as far as possible to the receiving socket. For this purpose, the first fragment is used to immediately resolve the destination socket using an extended interface to layer 4. This information is then stored together with the fragment in a collector data structure. Further fragments are identified as usual by their IP addresses and IDs. To allow an efficient implementation of the collector, incoming fragments have to arrive in a strictly ascending order, otherwise the whole chain is dropped. Incomplete chains are cleaned up when the related socket is closed. The total number of collectors is limited in order to be able to specify an upper bound for the lookup latency.

2.3 Driver Layer

Network interface cards (NIC) are attached to the stack core using a Linux-like driver interface. This allows straightforward porting of standard Linux drivers to RTnet, which has already been performed for about ten widely-used NICs. The NIC initialisation, configuration, and shutdown is still performed in non-real-time context under RTnet; porting standard drivers only requires to use the appropriate synchronisation mechanisms of the underlying RTOS here. However, special care has to be paid on the time-critical reception and transmission paths. They have to be audited in order to detect and avoid potential long delays while accessing the hardware.

A few extensions compared to the standard driver model are required to provide accurate timestamp services. RTnet does not depend on built-in timestamp clocks of the NIC, which are still not commonly available. Instead, the driver has to provide the packet reception and transmission time as precise as feasible. This means that the reception timestamp has to be taken for every packet right at beginning of the interrupt handler called upon the arrival. Furthermore, the driver has to provide the functionality to store the current time in an outgoing packet and trigger its transmission atomically. These measures widely reduce packet timestamp jittery to the single interrupt jitter which characterises platform and RTOS.

The driver layer furthermore provides two per-device hooks for redirecting transmission requests and MTU (maximum transmission unit) queries. Both hooks are transparent to the drivers. The transmission hook is used by the media access control layer RTmac and the capturing extension RTcap for managing, respectively, analysing outgoing packets. While standard network stacks typically provide only static device MTUs, RTnet offers logical channels of variable size up to the physical MTU to higher layers. The RTmac discipline TDMA utilises these channels to enforce specific slot sizes (see Section 3.2).

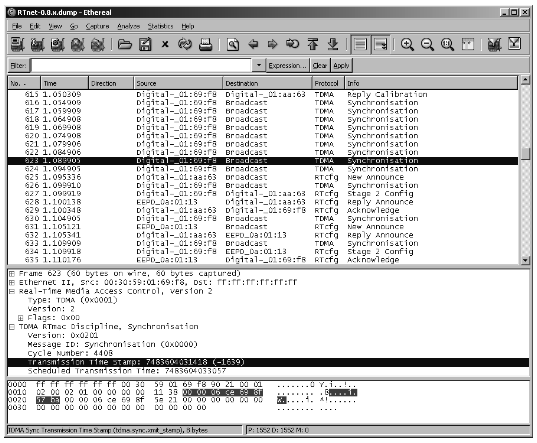
2.4 Application Programming Interface

Application programs can attach to the RTnet real-time services via a widely POSIX-conforming socket and I/O interface. The socket interface offers UDP and packet sockets for exchanging user data deterministically. The I/O interfaces provides access to additional features that services like TDMA (see Section 3.2) exports to users, for example clock synchronisation. Just as RTAI, RTnet permits both the classic kernel mode and more convenient user mode usage (Linux processes) of the API.

The related socket and I/O API functions are part of a separate interface concept called Real-Time Driver Model (RTDM). This interface addresses the specific requirements when accessing hardware on a mixed real-time system like Linux/RTAI, for instance differentiation between real-time and non-real-time service invocation. Currently, an implementation of RTDM comes with RTnet, but plans exist to merge the functionality into the RTAI project. This would also enable to utilise RTDM for other real-time devices drivers beyond RTnet.

2.5 Capturing Extension

A powerful extension of the RTnet core is the RTcap plug-in. It acts as a standard traffic capturing service for both incoming and outgoing packets over real-time NICs. Arriving packets are recorded together with a reliable high precision timestamp, solely depending on the interrupt jitter of the capturing system. RTcap adds only a small bounded overhead to the time-critical data paths when being installed on an active RTnet node. It furthermore cannot starve out any other packet user with respect to `rtskbs` because it maintains separate buffer pools for captured packets.



```
No.   Time      Direction  Source               Destination          Protocol  Info
  615 1.050309             Digital-_01:69:f8    Digital-_01:aa:63    TDMA      Reply Calibration
  616 1.054909             Digital-_01:69:f8    Broadcast            TDMA      Synchronisation
  617 1.059909             Digital-_01:69:f8    Broadcast            TDMA      Synchronisation
  618 1.064908             Digital-_01:69:f8    Broadcast            TDMA      Synchronisation
  619 1.069908             Digital-_01:69:f8    Broadcast            TDMA      Synchronisation
  620 1.074908             Digital-_01:69:f8    Broadcast            TDMA      Synchronisation
  621 1.079906             Digital-_01:69:f8    Broadcast            TDMA      Synchronisation
  622 1.084906             Digital-_01:69:f8    Broadcast            TDMA      Synchronisation
  623 1.089905             Digital-_01:69:f8    Broadcast            TDMA      Synchronisation
  624 1.094905             Digital-_01:69:f8    Broadcast            TDMA      Synchronisation
  625 1.095336             Digital-_01:aa:63    Broadcast            RTcfg     New Announce
  626 1.099910             Digital-_01:69:f8    Broadcast            TDMA      Synchronisation
  627 1.099919             Digital-_01:69:f8    Digital-_01:aa:63    RTcfg     Stage 2 Config
  628 1.100138             EEPD_0a:01:13        Digital-_01:aa:63    RTcfg     Reply Announce
  629 1.100348             Digital-_01:aa:63    Digital-_01:69:f8    RTcfg     Acknowledge
  630 1.104905             Digital-_01:69:f8    Broadcast            TDMA      Synchronisation
  631 1.105121             EEPD_0a:01:13        Broadcast            RTcfg     New Announce
  632 1.105341             Digital-_01:aa:63    EEPD_0a:01:13        RTcfg     Reply Announce
  633 1.109909             Digital-_01:69:f8    Broadcast            TDMA      Synchronisation
  634 1.109918             Digital-_01:69:f8    EEPD_0a:01:13        RTcfg     Stage 2 Config
  635 1.110176             EEPD_0a:01:13        Digital-_01:69:f8    RTcfg     Acknowledge

⊞ Frame 623 (60 bytes on wire, 60 bytes captured)
⊞ Ethernet II, Src: 00:30:59:01:69:f8, Dst: ff:ff:ff:ff:ff:ff
⊟ Real-Time Media Access Control, Version 2
    Type: TDMA (0x0001)
    Version: 2
  ⊞ Flags: 0x00
⊟ TDMA RTmac Discipline, Synchronisation
    Version: 0x0201
    Message ID: Synchronisation (0x0000)
    Cycle Number: 4408
    Transmission Time Stamp: 7483604031418 (-1639)
    Scheduled Transmission Time: 7483604033057

0000  ff ff ff ff ff ff 00 30  59 01 69 f8 90 21 00 01   .......0 Y.i..!..
0010  02 00 02 01 00 00 00 00  11 38 00 00 06 ce 69 8f   ........ .8....i.
0020  57 ba 00 00 06 ce 69 8f  5e 21 00 00 00 00 00 00   W.....i. ^!......
0030  00 00 00 00 00 00 00 00  00 00 00 00               ........ ....

TDMA Sync Transmission Time Stamp (tdma.sync.xmit_stamp), 8 bytes    P: 1552 D: 1552 M: 0
```

Figure 2. Using Ethereal with RTnet

Normal analysis network tools can be used with RTcap because a pseudo, read-only network device is created for every real-time NIC to forward the captured packets. Especially Ethereal [5], shown in Figure 2, is well-suited to dissect real-time communication as it fully understands the RTnet protocols. But the usage of RTcap in combination with traffic analyser is, of course, not limited to RTnet-managed networks or Ethernet. In principle, any transport media with RTnet-enabled drivers can be studied with RTcap's high timestamp accuracy.

3 Real-Time Media Access Control

As important as a real-time-capable stack implementation is a deterministic communication media. For instance, standard Ethernet, so far RTnet's primary media, does not provide adequate Quality of Service (QoS) features for hard real-time applications. Unpredictable collisions in hub-based Ethernet segments prevent short deterministic transmission times. Switches can overcome this issue but suffer from the risk of congestions which lead to packet delays or drops. QoS-enabled switches according to IEEE 802.1q are partly improving this situation, but they still require a centralised cabling which is often too costly for industrial applications.

Also other shared communication media may demand additional control over outgoing traffic in order to translate QoS parameters to a media-specific scheme or to extend existing QoS features where necessary. RTnet addresses the demand for deterministic and flexible media access control (MAC) mechanisms with its RTmac layer as described in the following. Moreover, as an example of a MAC discipline which is pluggable into the RTmac interface, a TDMA-based protocol is presented.

3.1 Pluggable MAC Layer

The RTmac is an optional extension to the RTnet stack. Although the stack is already functional without RTmac, it becomes mandatory if an underlying communication media lacks a deterministic access protocol. The RTmac layer was designed to provide these four elementary services to arbitrary software-based MAC implementations, here called disciplines:

- Interception of the crucial packet output path and redirection to discipline-specific handlers. For transmitting packets, this is performed right before the packet is passed to the NIC driver. Furthermore, a handler to override the device MTU on a per-packet basis can be installed.
- Exchanging discipline-defined control or data messages in a RTmac frame aside any user protocols.
- Discipline management on a per-device basis. To every real-time NIC, an individual MAC discipline can be assigned when it was registered with the RTmac layer.
- Packet tunnelling service for time-uncritical data as generated or received by the non-real-time network stack. This service creates a virtual network device for every RTmac-managed real-time NIC. Tunnelled packets are encapsulated by the RTmac protocol frame to distinguish between otherwise identical real-time and non-real-time protocols like UDP.

3.2 TDMA Discipline

Primarily for the use with standard Ethernet, RTnet provides a timeslot-based MAC discipline called TDMA (Time Division Multiple Access). TDMA in its current revision 2 is a master-slave protocol. It synchronises the clocks of RTnet nodes within a network segment. Furthermore, it defines the transmission time of any payload packet relative to sychronisation messages the master issues periodically.

A TDMA slave node can join a running network segment at any time provided it knows at least one parameter set of its slots. This set can either be configured statically or distributed via the RTcfg protocol (see Section 4). Given these parameters, the slave starts to join by sending a calibration request to the master. The master, in turn, replies with a message that contains the request arrival and reply departure times, both as precise as the system allows (see also Section 2.3). By taking its local departure and arrival times into account, the slave is able to calculate the packet round-trip delay. This procedure is repeated over a certain interval in order to estimate the medium time t_{travel} between starting to transmit a packet on the master and gaining its reception time on the slave.

$$t_{travel} = \frac{1}{2n} \sum_{i=1}^{n} T^{slave}_{recv,i} - T^{slave}_{xmit,i} - (T^{master}_{xmit,i} - T^{master}_{recv,i}) \qquad (1)$$

The master's synchronisation message contains the scheduled transmission time T_{sched} together with the timestamp taken right before packet release. This permits the slave to compensate potential scheduling jitters on the master node when calculating t_{offset}, the offset between local and global system clock. The slave can furthermore improve the precision of its own slot starting times T_{slot}.

$$\begin{aligned} t_{offset} &= T^{master}_{xmit} + t_{travel} - T^{slave}_{recv} & (2) \\ T_{slot} &= T_{sched} + t_{slot} - t_{offset} & (3) \end{aligned}$$

Time slots can be freely arranged within an elementary TDMA cycle as depicted in Figure 3. Besides node assignment and offset, also the slot size can be defined within physical limits of the transport media. TDMA allows that a node uses multiple time slots per cycle. Furthermore, it is possible to set custom periodicity and phasing of a slot to limit the network load or to share slots between different nodes. A management tool is available under Linux to create and maintain individual configurations based on scripts. Even a runtime reconfiguration within certain constraints is feasible.

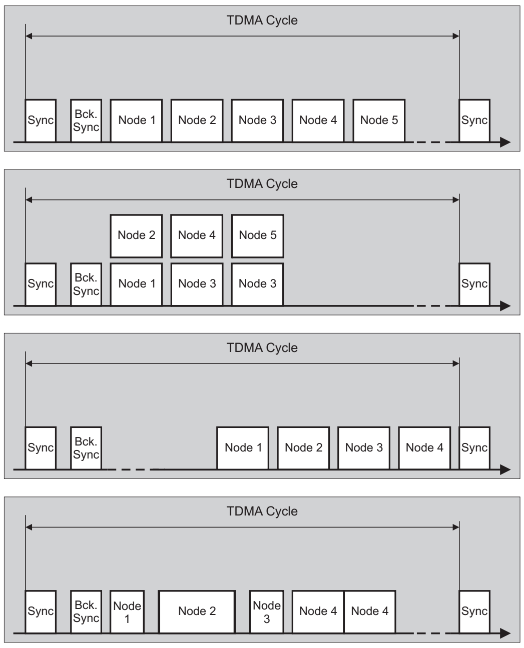


Figure 3. Flexible TDMA Slot Setup

In case multiple packets have been queued on a slot, the transmission order is defined by their priorities which can be set by real-time applications or RTnet services for each message. 31 real-time levels are available, the 32nd and lowest one is reserved for time-uncritical data, i.e. VNIC traffic. With multiple slots per node, the need for a scheduling scheme arises. For efficiency reasons, TDMA provides explicit scheduling only. Slots are numbered on each node with ID 0 predefined for default real-time and ID 1 for non-real-time traffic. In case only a single slot is available, ID 1 is mapped on slot 0. Any additional slots are reserved for explicit assignment to arbitrary real-time applications via the socket API.

As the master is a single point of failure, its services can be backed up by one or more secondary masters. An additional time slot has to be assigned to every such backup master, marked as "Bck. Slot" in Figure 3. In case the primary master fails to transmit a synchronisation message, the next backup master on the time axis will start issuing its own messages. The offset between primary and secondary master is automatically compensated with a now larger difference between scheduled and actual transmission time contained in every synchronisation frame. When the primary master has been fixed and starts taking over again, it first synchronises its own clock on the active backup master in order to avoid significant clock skews. Afterwards it issues its own synchronisation messages again, and the backup master switches to standby.

The TDMA discipline creates a RTDM I/O device for every controlled network device. These I/O devices can be used to retrieve the clock offset introduced above and to synchronise a real-time task on the TDMA cycle.

4 Real-Time Configuration Service

During the revision of the first TDMA protocol it became apparent that a clear separation between RTmac disciplines on the one side and generic configuration as well as monitoring services on the other is essential for RTnet's extensibility. For this reason, the Real-Time Configuration Service RTcfg has been designed in a discipline- and media-agnostic manner. It does not depend on a specific communication media given that broadcast transmissions are supported. The IPv4 protocol is supported but not mandatory. Other network protocols like IPv6 can be integrated, and physical addresses may be used even purely. The concrete tasks of RTcfg are:

- Distribution of essential discipline configuration data to newly joining nodes. This information is issued unsolicited, thus enabling nodes to join real-time networks on-the-fly as far as physical media and RTmac discipline allow.

- Monitoring of active nodes and exchange of their physical and logical addresses. This service can be used, for example, to set up and maintain the static ARP tables mentioned in Section 2.2. It is furthermore possible to build real-time network monitoring tools on top of RTcfg's interfaces.

- Synchronisation of the real-time network start-up procedure. Specific RTmac disciplines or certain application scenarios may require common rendezvous points in order switch network mode or start applications synchronously.

- Distribution of arbitrary configuration data, even in the absence of TCP/IP with its typically used file transfer protocols like TFTP/FTP etc.

RTcfg is based on a client-server protocol. A central configuration server stores parameter sets of every managed client in a network segment. This information is used by the server to continuously invite any known but yet inactive client to join. The client's start-up procedure as shown in Figure 4 consists of three stages. The first stage is completed after the client has received its single packet of initial parameters that is identifiable either through the physical or logical destination address. These parameters typically contain the minimum information required to set up a possible RTmac discipline, for example at least one TDMA slot configuration.

In the second stage after completing the discipline initialisation, the client announces its presence to any other network nodes which can then update their address information like static ARP tables. Already active clients reply on this announcement by sending the new node their own identification. The server replies in contrast by transmitting an optional second set of configuration data which can be scattered over multiple packets. After the server has received the final stage 2 acknowledge message from the last missing client node, the network is ready for a potential common operating mode switch in case such synchronisation is required.

As stage 3, an optional second rendezvous point is provided to both server and clients. It can be utilised to wait for all nodes to complete processing the configuration data they received during stage 2.

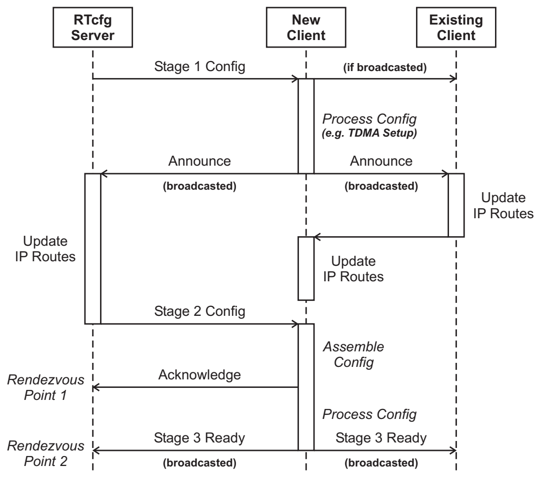


Figure 4. RTcfg Client Start-up in 3 Stages

After the setup completion the clients can be instructed to transmit low-frequent heartbeat frame to the server in order to track potential node failures. If the server detects lacking heartbeat frames, it declares the client dead by broadcasting a related message to the remaining nodes. As a result, all nodes will remove any address of the broken client from their local tables. This enables a restart procedure of the repaired or replaced node. A failing RTcfg server can also be restarted, even on a different system, without the need to go through the full start-up procedure of every running node once again.

5 Integration of FireWire

FireWire, also known as IEEE 1394 [8], is a high-performance serial bus for connecting heterogeneous devices. Though firstly targeted for consumer-electronic applications, such as high-speed video transmission, many of FireWire's features make it well fit industrial and laboratorial context. In the following subsections, an overview of FireWire is given and the current status of its integration to RTnet is described.

5.1 FireWire Overview

The bus topology of FireWire is tree-like, i.e. non-cyclic network with branch and leaf nodes. The physical medium supports data transmission up to 400 Mbps in 1394a specification. In 1394b specification, the speed even rises to 3.2 Gbps. Two types of data transaction are supported on FireWire: asynchronous and isochronous. As illustrated in Figure 5, a mix of isochronous and asynchronous transaction is performed by sharing the overall bus bandwidth, of which the allocation is based on 125 μs intervals, so called FireWire cycles.

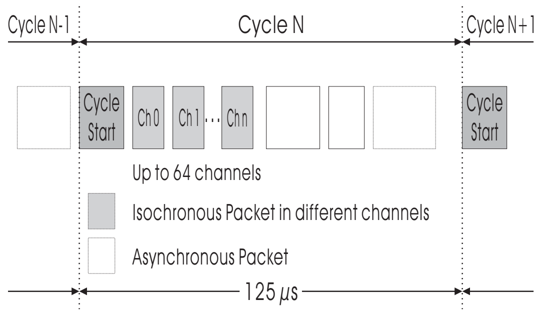


Figure 5. FireWire Cycle

Isochronous transaction targets one or more nodes by being associated with a multicasting channel number. There can be maximally 64 channels in total. Once bus bandwidth has been allocated for an isochronous transaction, the associated channel can receive a guaranteed time-slice during each 125 μs cycle. Up to 80% (100 μs) of each bus cycle can be allocated to isochronous channels. Because this transaction type does not re-transmit broken packets, but deliver data at constant, real-time rate, it is well suited for the time-triggered state message transmission in distributed control systems.

In the asynchronous transaction phase, the whole network on FireWire appears as a large 64-bits mapped bus address space, with each node occupying a 48-bits mapped space. The high-order 16 bits of address are used to identify nodes[1]. An asynchronous transaction is split into two sub-transactions: request to access a piece of address on another node and response. Coordination between request and response is ascertained by the transaction layer protocol. Since guaranteed data delivery is provided through acknowledgement, asynchronous transaction is targeted for non-error-tolerant applications, like command and control message transmission in distributed control system.

Bus management on FireWire includes different responsibilities that can be distributed among one or more nodes: Cycle Master, Isochronous Resource Manager and Bus Manager. The Cycle Master broadcasts a start message at the beginning of each cycle. The Isochronous Resource Manager takes care of the allocation of bus bandwidth and isochronous channels. The Bus Manager has several functionalities including publishing the bus speed map and the bus topology map. Since FireWire connects devices that may not support the same top speed of data transmission, the bus speed map is used by a certain node to determine at what speed it can communicate with another node. The topology map may be used by end-users to optimise the bus topology for a highest throughput.

5.2 FireWire Stack and Connection to RTnet

The FireWire stack, as shown in Figure 6, is adapted from the Linux variant[9]. Functions in the kernel are decoupled into several modules. Application on the stack acquires either a portion of bus address or one or more multicasting channels, by using the primitives from the Application Interface and Management layer.

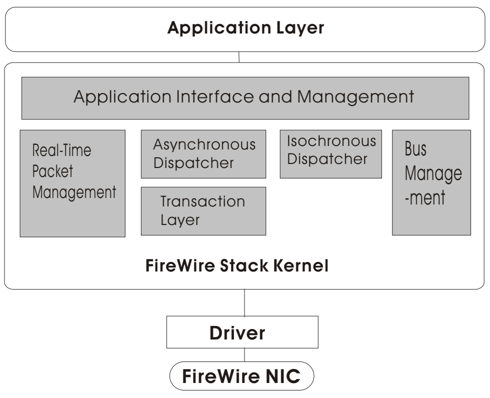


Figure 6. FireWire Stack

The RTnet mechanism for real-time packet management is applied to the FireWire stack as well. Both NIC driver and high-level applications are potential producers and consumers of packets. All packets are carried by a generic packet buffer structure `rtpkb`. Like in RTnet, pre-allocation of `rtpkbs` is done during set-up, with

[1]Here, we only talk about the peer-to-peer asynchronous transaction. In 1394a supplement, the multicasting packet in asynchronous transaction is also defined.

each `rtpkb` carrying a fixed size of payload that is large enough to meet various scenarios.

The path of delivering incoming packets to application layer is realised by a real-time task, the so-called tasklet server. Upon arrival of a new packet, a suitable processing routine, either for asynchronous or isochronous, is hooked to the server as a tasklet. The server works under the rule First In First Served (FIFS), which means the packets are processed in the order of arrival time. When no tasklet is being queued, the server stays in idleness until the next packet arrives. A RTOS semaphore is used for the synchronisation between server and tasklet queue. Like the stack manager in RTnet, the server runs at a higher priority than application tasks.

The connection between FireWire stack and RTnet core is implemented through Ethernet emulation. The emulation is a module on application layer, using a portion of bus address to employ a protocol converting FireWire packets to Ethernet packets and vice versa. By using Ethernet emulation, FireWire functions the same as other real-time Ethernet devices in RTnet.

6 Application Protocols and Frameworks

The advantage that RTnet provides its real-time communication services through a widely standardised API instead of, for example, a specialised, solely fieldbus-oriented interface becomes obvious when considering application protocol layers. This section introduces some of them and also presents an exemplary concept for mapping an existing fieldbus middleware, CANopen, on RTnet's services.

6.1 netRPC – Remote Real-Time Procedure Calling

One of the first user of RTnet was its primary real-time execution platform itself. RTAI (3.x series) [17] comes with a plug-in called netRPC that enables a distributed usage of its RTOS services. This remote procedure calling service (RPC) is built upon the UDP/IP protocol. It can either be attached to the Linux non-real-time network stack, typically for testing and demonstration purposes, or to the RTnet API. In the latter case distributed hard real-time is provided to the RTAI applications almost transparently. Some of the RTAI developers make use of this feature in their real-time multi-body dynamics analysis software MBDyn [13].

6.2 RTPS Protocol

The Real-Time Publish-Subscribe Protocol (RTPS) [14] has been developed in order to provide real-time communication services over unreliable IP networks like Ethernet. The protocol contains mechanisms to detect critical packet delays or losses and avoids indeterministic retransmissions, as for example TCP causes, by using UDP as transport protocol. In order to keep real-time communication operational on Ethernet, only a low network load is acceptable in RTPS segments. RTPS is available as a commercial product (NDDS) and is included in various industrial products, for instance in certain Schneider PLCs.

Moreover, an Open Source implementation of RTPS called ORTE [2] is available. ORTE runs on a large number of platforms over conventional UDP/IP stacks and, additionally, supports RTnet on top of RTAI. By utilising RTnet's hard real-time UDP/IP services, RTPS can now be used even under high non-real-time network load, as RTnet reliably separates this traffic from the time-critical data.

6.3 Real-Time Control Frameworks

Both for research and industrial scenarios, increasingly complex control tasks demand powerful frameworks to facilitate the development of distributed real-time systems. One of such frameworks has been developed at the Real-Time Systems Group in Hannover with the focus on robotic research [20]. This framework transparently supports distributed applications both deterministically over RTnet (UDP/IP) and without timing guarantees over standard TCP/IP. Its communication models include remote procedure calling as well as producer-consumer schemes.

A similar framework, OROCOS, also makes use of RTnet for closed-loop control [15]. Moreover, plans exist for OROCOS and the related OCEAN project to run RT-CORBA over RTnet. The latter project already evaluated an earlier version of RTnet and concluded that integrating it as pluggable protocol into the RT-CORBA implementation ACE/TAO is a promising approach [19].

6.4 CANopen

The CAN in Automation organisation has developed CANopen as an application protocol and device model for the automation domain [1]. Beyond its original use on top of the CAN fieldbus, CANopen has recently been adopted by two commercial real-time Ethernet solutions, ETHERNET Powerlink [3] and EtherCAT [4]. Both approaches are, as well as RTnet, quite different compared to the CAN bus with respect to node addressing, message priorities, or communication models. Therefore, ETHERNET Powerlink and Ethercat only reuse the device profiles specified by CANopen. In following, the feasibility and potential of adopting CANopen to RTnet is briefly analysed. Such an extension would enable classic automation applications like soft-PLCs to run more straightly over RTnet.

As CAN itself is agnostic to message source and destination addresses, CANopen maps the common three addressing modes broadcast, unicast, and multicast on CAN message identifiers. Broadcast messages are used for network management, synchronisation, time stamping, and alarming purposes. CANopen exchanges so called Service Data Objects (SDO) for time-uncritical direct communication between two nodes as unicast messages. Process Data Object carrying the real-time data are transmitted according to the multicast scheme with a single producer and an arbitrary number of consumers.

RTnet supports broadcast as well as unicast both via UDP and user-defined Ethernet protocols. As multicast support is not yet part of RTnet, such messages can be issued transitionally either via unicast in case only a single consumer exists or as broadcasts using additional software filters on the receiving nodes. Basically, an extension of the Communication Object ID (COB-ID) format is required, which was originally defined with solely CAN IDs in mind. While CAN prioritise messages implicitly according to their ID, an explicit value is now required which also encode the output channel on RTnet. An extended COB-ID would demand the following fields:

- ID type (UDP/IPv4, UDP/IPv6, Ethernet, CAN, etc.)
- Destination node address (IP, Ethernet MAC, etc.)
- Message ID (UDP destination port, Ethernet frame type, CAN ID, etc.)
- Priority and channel (RTmac queuing priority, TDMA slot, etc.)

The CAN-specific Remote Transmission Requests (RTR) are utilised by consumers for soliciting a PDO from the producer. This protocol can be emulated by sending an empty PDO with identical COB-ID to the producer.

Based on the proposed addressing scheme, typical CANopen stacks, for instance one of the various free implementations [6], may already be reused on top of RTnet. Certain CANopen services could be mapped directly on RTnet equivalents. RTcfg provides heartbeat mechanism which can replace CANopen's variant. TDMA comes with an API to synchronise nodes and distribute a common time base, services that be used in place of the CANopen protocol. Additional optimisation potential lies in larger transfer fragments when exchanging SDOs. CANopen's limitation to CAN-related 8 bytes can be easily overcome by defining new, COB-ID-specific SDO upload and download protocols that make use of different maximum packet sizes (e.g. almost 64 KB via UDP/IP).

7 Summary and Outlook

This paper introduced RTnet as an adaptable and extensible framework for deterministic communication over standard Ethernet, FireWire, or other suited media. Its open, standard-oriented, and modulised structure allows numerous application scenarios like distributed real-time systems, fieldbus coupling devices, intelligent I/O interfaces, low-cost real-time network analysers, etc. Application software may either interact directly with the RTnet API, or middlewares like RTPS or CANopen can be build over RTnet's services.

Future work will focus on further integration of FireWire, new media like Gigabit Ethernet, and interoperation with additional middlewares. To decouple organisational dependencies, the RT-FireWire stack has recently become a separate project. Based on the connection to RTnet via Ethernet emulation, the adoption of FireWire's transaction modes and clock synchronisation for RTnet services will now be addressed. Furthermore, the potential of layering CANopen over RTnet will be analysed and can lead to the implementation of an extended CANopen stack.

The current RTnet implementation has been build upon free software, it tightly interacts with many Open Source projects, and it is therefore available under Open Source licenses, too. For downloads and further information, visit

www.rts.uni-hannover.de/rtnet

References

[1] CiA. *CANopen, Application Layer and Communication Profile*. CAN in Automation, Feb. 2002.

[2] O. Dolejs, P. Smolik, and Z. Hanzalek. On the Ethernet use for real-time publish-subscribe based applications. In *5th IEEE International Workshop on Factory Communication Systems*, Vienna, Austria, Sep. 2004.

[3] ETHERNET Powerlink Standardization Group. www.ethernet-powerlink.org.

[4] EtherCAT Technology Group. www.ethercat.org.

[5] Ethereal. www.ethereal.com.

[6] CANopen free software resource center. canopen.sourceforge.net.

[7] F. T. Y. Hanssen and P. G. Jansen. Real-time communication protocols: an overview. Technical Report TR-CTIT-03-49, Centre for Telematics and Information Technology, Univ. of Twente, The Netherlands, Oct. 2003.

[8] IEEE. *IEEE standard for a high performance serial bus, Std 1394-1995 and amendments*, 2002.

[9] IEEE 1394 for Linux. www.linux1394.org.

[10] LinuxDevices.com. Lineo announces GPL real-time networking for Linux: RTnet. www.linuxdevices.com/news/NS4023517008.html, July 2000.

[11] J. Loeser and H. Haertig. Low-latency hard real-time communication over switched Ethernet. In *16th Euromicro Conference on Real-Time Systems*, Catania, Italy, 2004.

[12] J. Martínez, M. Harbour, and G. J.J. A multipoint communication protocol based on Ethernet for analyzable distributed real-time applications. In *2nd International Workshop on Real-Time LANs in the Internet Age*, 2003.

[13] Multibody dynamics analysis software on real time distributed systems. www.aero.polimi.it/~mbdyn/mbdyn-rt.

[14] Real-Time Innovations, Inc. *Real-Time Publish-Subscribe Wire Protocol Specification, Protocol Version 1.0, Draft Document Version 1.17*, 2002.

[15] Open Robot Control Software. www.orocos.org.

[16] P. Pedreiras, L. Almeida, and P. Gai. The FTT-Ethernet protocol: Merging flexibility, timeliness and efficiency. In *14th Euromicro Conference on Real-Time Systems*, 2002.

[17] Real Time Application Interface. www.rtai.org.

[18] J. Schwager. Real-time Ethernet in industry automation. www.realtime-ethernet.de.

[19] M. Wild et al. OCEAN deliverable D2.1: Design of the DCRF lower layers including hardware requirements. www.fidia.it/download/ricerca/ocean/deliverable2_1.pdf, 2003.

[20] O. Wulf, J. Kiszka, and B. Wagner. A compact software framework for distributed real-time computing. In *5th Real-Time Linux Workshop*, Valencia, Spain, Nov. 2003.

References

1394OHCI (2000). 1394 Open Host Controller Interface Specification.

Anderson, D. (1999). FireWire System Architecture, Addison-Wesley.

Apple "FireWire Homepage in Apple."

ControllabProducts (2005). "20SIM Homepage."

FusionTeam (2004). "Life with Adeos."

IEEE (1994). Information technology -- Microprocessor systems -- Control and Status Registers (CSR) Architecture for microcomputer buses.

IEEE (2002). "IEEE standard for a high performace seial bus, std 1394-1995 and amendments."

Kiszka, J., Wagner,B., Zhang, Y. and J.F. Broenink (2005). RTnet -- A Flexible Hard Real-Time Networking Framework. IEEE Emerging Technology of Factory Automation. Italy.

Kopetz, H. (1997). Design Principles for Distributed Embedded Applications, kluwer Academic Publishers.

Linux1394 "Linux1394 Homepage."

Mullender, S. (1993). Distributed Systems, Addison-Wesley.

Plummer, D. C. (1982). "RFC 826 - Ethernet Address Resolution Protocol."

Pranevich, J. (2003). "The Wonderful World of Linux 2.6."

RTAI (2005). "RTAI Homapage."

RTnet (2005). "RTnet Homepage."

Rubini, A. (2001). Linux Device Driver, O'REILLY.

Buit, E. (2005). PC104 Stack Mechatronic Control Platform, Control Laboratory, University of Twente, the Netherlands

Buit, E. (2004). Real-time network performance characterization, Control Laboratory, University of Twente, the Netherlands

Ferdinando, H. (2004). Testing CAN for Robotic Control, Control Laboratory, University of Twente, the Netherlands

Groothuis, M. A. (2004). Distributed HIL simulation for BodeRC, Control Laboratory, University of Twente, the Netherlands

Huang, Y. (2005). Time characteristics of PROFIBUS on Windows XP, Control Laboratory, University of Twente, the Netherlands

Zhang, Y. (2004). Real-Time Control on FireWire Control Engineering Group, University of Twente, the Netherlands