AN INVESTIGATION INTO THE APPLICATION OF THE IEEE 1394 HIGH PERFORMANCE SERIAL BUS TO SOUND INSTALLATION CONTROL

A thesis submitted in fulfilment of the
requirements for the degree of

Master of Science

of

Rhodes University

By

BRADLEY HUGH KLINKRADT

February 2003

Abstract

This thesis investigates the feasibility of using existing IP-based control and monitoring protocols within professional audio installations utilising IEEE 1394 technology. Current control and monitoring technologies are examined, and the characteristics common to all are extracted and compiled into an object model. This model forms the foundation for a set of evaluation criteria against which current and future control and monitoring protocols may be measured. Protocols considered include AV/C, MIDI, QSC-24, and those utilised within the UPnP architecture. As QSC-24 and the UPnP architecture are IP-based, the facilities required to transport IP datagrams over the IEEE 1394 bus are investigated and implemented. Example QSC-24 and UPnP architecture implementations are described, which permit the control and monitoring of audio devices over the IEEE 1394 network using these IP-based technologies. The way forward for the control and monitoring of professional audio devices within installations is considered, and recommendations are provided.

Acknowledgements

A very special thank you to Richard Foss for supervising this project, and for guiding its direction. I also gratefully acknowledge the support from the folk at Digital Harmony Technologies for funding the project and for providing technological support. The support and hardware provided by the Centre of Excellence within the Computer Science Centre at Rhodes University is also greatly appreciated.

Lastly, a very appreciative thank you to my much-loved wife, Mary. Your constant encouragement and support has finally paid off.

Table of Contents

List of Figures

List of Tables

List of Code Listings

Chapter 1

Introduction

Live entertainment venues, modern churches, shopping venues, and the like, are all equipped with some form of sound system and are collectively known as sound installations. Traditionally, these installations have been analog based, resulting in considerable wiring costs and reconfiguration complexity. However, these aggravations have been lessened through the introduction of digital networks, and software controllable devices. Although this has tidied up the cable clutter, digital networks are not without their own shortcomings.

In fact, the current state of the sound installations industry is one riddled with devices that are not able to interoperate. This has come about due to the majority of manufacturers each adopting their own proprietary approaches to sound installations and the control and monitoring of devices within them. This situation has been aggravated by the lack of a suitable audio and control data distribution network. Although recent developments are producing networks that are sufficient to support the audio installation industry requirements, audio manufacturers are reluctant to part with proprietary technology due to large time and financial investments.

Examples of such networks include IEEE 1394 bus and CobraNet, which provide facilities for the distribution of audio and control data over the same cable. IEEE 1394 networks can deliver audio data in real-time with guaranteed bandwidth through the use of standardised protocols. However, the control facilities present within these networks are currently less than adequate.

This thesis sets out to investigate the hypothesis that currently available IP-based protocols could provide a much needed solution for control and monitoring over IEEE 1394 networks. The approach consists of two parts. The first is to provide IEEE 1394 devices with the ability to transmit IP datagrams, thus permitting the transfer of IP-based protocols. The second is to provide appropriate criteria for the evaluation of

selected control and monitoring protocols. The selected protocols are then considered, using these criteria, to determine their suitability.

Chapter 2 provides an analysis of a selection of currently available technologies and the control and monitoring facilities they provide. The common characteristics across all of the technologies were extracted and formalised in the form of an object model. The purpose of this model is to aid in the formation of evaluation criteria for current control and monitoring protocols, and in the development of new protocols for the IEEE 1394 bus.

Chapter 3 provides an overview of the IEEE 1394 architecture, highlights the facilities that are applicable to sound installations, discusses the encapsulation and distribution of audio and Musical Instrument Digital Interface (MIDI) data over the IEEE 1394 bus, and introduces the Audio/Video Control (AV/C) protocol specification, which is currently utilised for control and monitoring.

As chapter 3 establishes the fundamentals of IEEE 1394 networks, chapter 4 can explore the applicability of IEEE 1394 networks to the control of audio devices within sound installations. The currently available IEEE 1394 control and monitoring protocols of AV/C and MIDI are considered using the evaluation criteria identified. This evaluation provides further motivation for the transport of existing IP-based control and monitoring protocols over the IEEE 1394 bus and the proposal of an IPv4 over IEEE 1394 implementation. Although any IP-based control and monitoring protocol could be selected for transmission, the QSC-24 and UPnP architecture protocols are evaluated.

Chapter 5 details the implementation of an IP stack above an embedded Digital Harmony DHIVA development board. This allows the transmission of IP datagrams over the IEEE 1394 bus. A QSC-24 message processor was developed above this stack to receive QSC-24 messages transmitted via IP, and transmit appropriate information out of a serial port present on the DHIVA.

This thesis concludes with a look at the UPnP architecture as an alternative to the control and monitoring of devices present within sound installations. Chapter 6

details this architecture for its applicability to control and monitoring over the IEEE 1394 bus. An example implementation is developed, which consists of a simulated audio device and a web browser as a control point. This chapter includes an evaluation of the UPnP architecture as a potential solution to control and monitoring over IEEE 1394 networks.

Chapter Two

An Overview of Sound Installation Technology

There are an abundance of audio systems currently utilised within the sound installation industry, each having distinct advantages and disadvantages. Through a closer examination of the protocols utilised within these systems, similarities between these protocols can be extracted, thereby allowing a generic model of sound installations to be constructed.

The three audio systems selected for this task are the:

- MediaMatrix System manufactured by Peavey Electronics Corporation,
- IQ Audio System manufactured by Crown Audio Inc., and
- QSControl System manufactured by QSC Audio Products Inc.

Once the generic requirements for an audio system have been identified through the examination of the above listed technologies, a set of criteria is provided against which other audio systems can be measured. In particular, current technologies utilised within the IEEE 1394 environment can be considered against these criteria as a measure of their functionality and suitability for professional audio installations.

2.1 MediaMatrix Audio System

The Peavey Electronic Corporation has taken a unique approach to sound systems with the development of their MediaMatrix Audio System. This system consists of a high-end PC with specialised audio hardware and software known as a MediaMatrix Frame, and one or more break-out-boxes (BoB's). The MediaMatrix Frame is essentially a range of simulated audio devices, such as mixers, equalisers, crossovers, level settings, meter readings, and audio I/O, packaged into one unit and thereby simplifying system construction and wiring [Peavey Electronics Corporation, 2001d].

The MediaMatrix Frame software presents the user with a CAD-type interface in which an audio system can be constructed using the wide range of devices provided, and allows the desired inputs and outputs on these devices to be connected. The system can then be compiled, which is the process of translating the designed audio system into a live system capable of processing audio [Peavey Electronics Corporation, 2001d]. Figure 2.1 illustrates one possible layout of the MediaMatrix Audio system components.

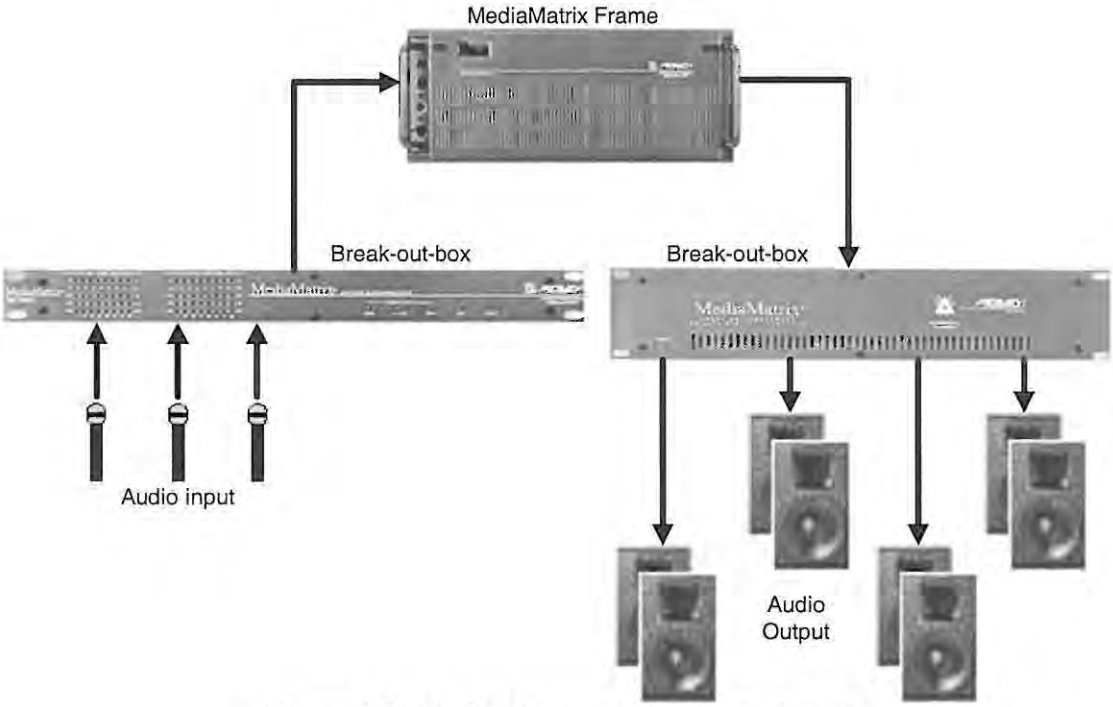

Figure 2.1: MediaMatrix Audio System layout

The MediaMatrix Frame operates in the digital domain, and all processing that occurs on the audio signal is digital. For instance, the audio devices that are provided within the MediaMatrix Frame are implemented by the digital signal processors (DSP) located on the specialised audio hardware [Peavey Electronics Corporation, 2001d].

The MediaMatrix system also provides facilities to allow traditional audio equipment to be included in the system. This is achieved through the use of the BoB's, of which there are two types currently available:

- Analog BoB
- CAB (Cobranet Audio Bridge)

The CAB BoB's are further divided into BoB's for analog input (CAB16i), analog output (CAB16o), and digital I/O (CAB16d) [Peavey Electronics Corporation, 2001c].

The main function of the analog BoB's is to provide an interface between the MediaMatrix Frame operating in the digital domain and the analog audio world. The facilities provided by the MM-8800 series of analog BoB's include [Peavey Electronics Corporation, 2001a]:

- Eight analog inputs
- Eight analog outputs
- Eight control inputs
- Eight control outputs
- High-speed data link to the MediaMatrix Frame

The typical operation of a BoB includes the analog to digital (A/D) conversion of analog audio signals destined for the MediaMatrix Frame, and the digital to analog (D/A) conversion of signals received from the MediaMatrix Frame. The development of the CAB BoB's was prompted by the need to allow the transmission of digital audio, and by the 6 foot restriction on the digital interface cable imposed by the analog BoB's [Peavey Electronics Corporation, 2001g]. The CAB BoB's have slightly more facilities than the analog BoB's, such as the ability to utilise off-the-shelf CAT5 cabling, Ethernet hubs, increased distances between MediaMatrix Frame and BoB, and the provision of a "Buddy Link" system which allows for the use of an external clock for synchronisation. However the main facilities and responsibilities of the BoB's are the same for both types [Peavey Electronics Corporation, 2001c].

There are also control input and output ports on the BoB's. The control output ports deliver logic output from the system, the voltage level of which is in TTL format. The two possible states are defined as follows [Peavey Electronics Corporation, 2001a]:

- 0 Volts equals *off*
- +5 Volts equals *on*

These logical outputs can be used to drive light emitting diodes (LED's), lights, relays, switches, and other indicators. For example a TTL output can be used to illuminate an LED on a remote hardware panel to indicate a system failure or warning.

The control input ports allow external control devices to be connected to the BoB, examples of such control devices include switches, opto-isolaters, and potentiometers (pots). These devices can then be configured through software to control certain aspects of the MediaMatrix system. For example a potentiometer connected to a control input can be used to control levels or specify routing selections [Peavey Electronics Corporation, 2001a].

The high-speed media link, which transfers control and audio data between the frame and BoB's, does differ between BoB types. Presently there are three such links defined, and are named according to the MediaMatrix Frame Digital Processing Unit (DPU) card [Peavey Electronics Corporation, 2001e]:

- "Legacy" DPU Card
- MM-DSP-RJ DPU Card
- MM-DSP-cn DPU Card

The legacy DPU card is the original MediaMatrix audio processing card that utilises a proprietary 9-pin DB-connector for the high-speed media link. An advantage of this link is that both the transmit and receive conductors are housed within one cable, with the downfall being that the cable is specialised and only available from Peavey Electronic Corporation [Peavey Electronics Corporation, 2001e]. The MM-DSP-RJ DPU card allows traditional CAT5 cabling to be used for the high-speed media link, but requires separate cabling for the receive and transmit lines. The interface is also a proprietary interface and should not be connected to equipment other than those specified to prevent serious damage [Peavey Electronics Corporation, 2001g]. The newest DPU card is the MM-DSP-cn card, which has a Cobranet interface. This interface allows CAT5 cabling and traditional Ethernet equipment to be used, as well as overcoming the distance limitations associated with the previous cards. The audio

input and output used to be present on one BoB, but with the addition of the Cobranet interface they have been separated into two separate BoB's.

2.1.1 Digital Synchronisation

All digital audio equipment samples incoming audio at a specific rate, this is known as the sampling rate. Most audio equipment can generate this rate from an internal clock and this is sufficient if there are no digital interconnections with other audio equipment. If however, audio equipment is digitally interconnected, some means must be provided to ensure that all sampling rates are precisely synchronised [Peavey Electronics Corporation, 2001c]. There are three main methods employed to ensure that the sample rates are synchronised, and these are [Rumsey et al, 1995]:

a) Frame alignment – Signals that have been delayed are likely to be out of phase with the reference signal, and need to be rephased before presentation. Frame alignment is, however, only applicable to signals that are 25%, or more, of a frame period out of phase.
b) Buffering – This mechanism is employed by devices that are synchronised to the same nominal sample rate, but are not locked to the same reference signal. Due to the lack of a common reference, the signals are likely to drift in relation to one another. If, however, these signals are written sequentially into a buffer, and read a short time later, it permits a small level of variation between the input and output signals. The possibility exists that these buffers could underflow or overflow, thereby introducing audible artifacts into the audio stream.
c) Sample rate conversion – If samples rates differ by a large factor, such as between 44.1 kHz and 48 kHz, it is possible to reframe the audio from one rate to the other.

The MediaMatrix Frame provides both internal and external synchronisation modes. When in internal mode the sample clock is generated by one of three quartz crystals on the first DPU card in the frame. The three clocks allow for the generation of sampling rates at 32Khz, 44.1Khz, and 48Khz, however only one sampling rate is produced at any one point, and is software selectable [Peavey Electronics Corporation, 2001c].

External synchronisation allows an external source to provide the sample clock to the MediaMatrix Frame. This external clock is applied to the first DPU card in the frame, from where it is distributed to the rest of the MediaMatrix system.

BoB's are configured with factory defaults that dictate the sampling rates at which they will operate, and this rate is not modifiable through software. For example the MM-8830 BoB's are set to a rate of 32Khz, and this rate cannot be altered [Peavey Electronics Corporation, 1996]. If the MediaMatrix Frame and the BoB are operating at different sampling rates, the BoB will not pass audio and will mute [Peavey Electronics Corporation, 2001c].

2.1.2 Software

The software provided on the MediaMatrix Frame can be broken down into four broad categories or modes [Peavey Electronics Corporation, 2001c]:

- Edit
- Wiring
- Control
- Paint

The Edit mode allows the sound system to be constructed by placing the selected objects, such as audio I/O, equalizers, and routers, on the screen. Once all the desired components have been arranged in an appropriate manner the devices can be wired together, which is done in the wiring mode. Dragging a "wire" from any input port to any output port sets up the wiring connections. The paint mode allows the colours of any component, control or wire to be changed to suit individual preference. At this point the sound system has been constructed, and is ready to be compiled. Figure 2.2 provides an illustration of such a system.

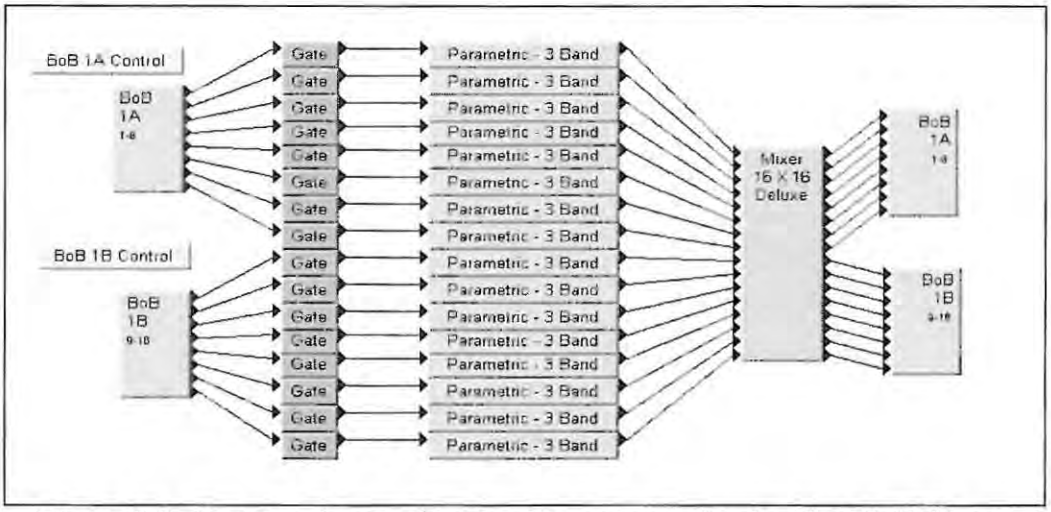

Figure 2.2: An example sound system layout created using the MediaMatrix frame software

The control mode allows the values of adjustable objects, such as knobs, faders and buttons, to be set. All the control within the MediaMatrix system is manipulated through software, of which there are two types:

- Control over simulated devices.
- Control using the control ports provided on the BoB's.

Once the initial system design, construction, and compilation have been completed, the user is presented with a similar interface for both types of control, making the control type transparent to the end user. The advantage this offers is uniformity and ease of use when making use of the system. Figure 2.3 is an example of this, and provides the graphical user interface for a 16-input by 16-output mixer, as seen in the MediaMatrix software.

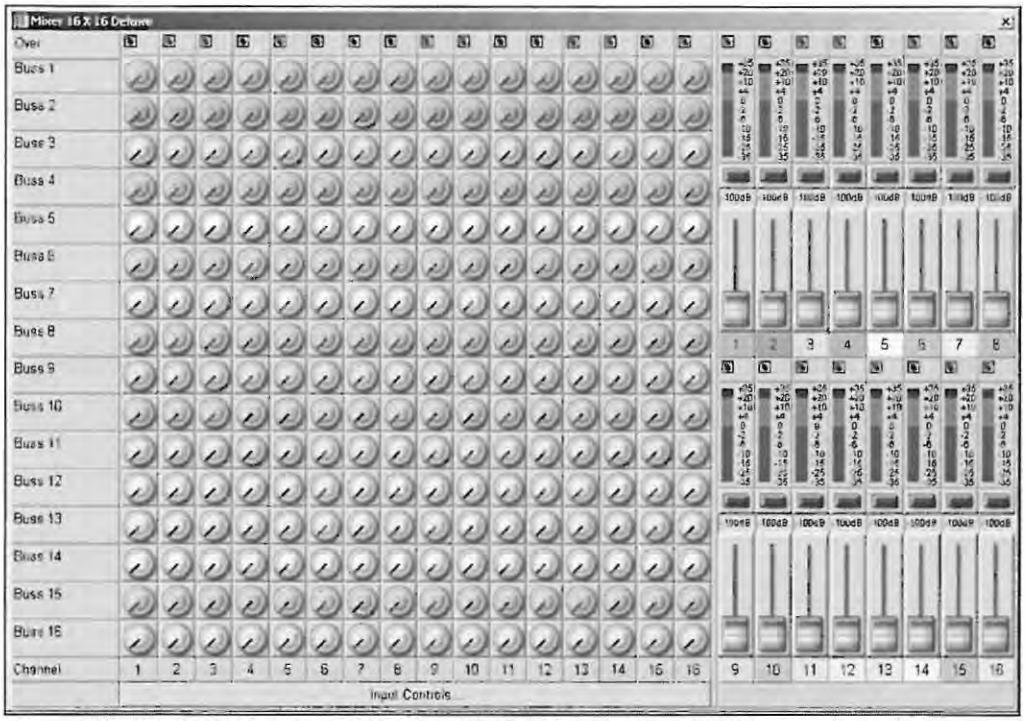

Figure 2.3: An example of a MediaMatrix graphical user interface

2.2 IQ Audio System

Crown Audio Inc. developed the IQ Audio System in 1988 for the remote control and monitoring of amplifiers in large sound systems. The system architecture consists of a system controller controlling a number of system devices. This system controller is usually a PC with custom software supplied by Crown Audio Inc., although it can be any device capable of RS232/422 communication [Crown Audio, Inc., 1996b].

The System Controller communicates, via a RS232/422 link, with the system devices through a system interface device, whose main function is to bridge communication between the Controller and the system devices. The communications network between the systems interface and the system devices is known as the IQ Bus and is a closed loop, as illustrated in Figure 2.4 [Crown Audio, Inc., 2001e]. Although not depicted in figure 2.4, a system controller could be attached to more than one system interface, and consequently have more than one bus loop.

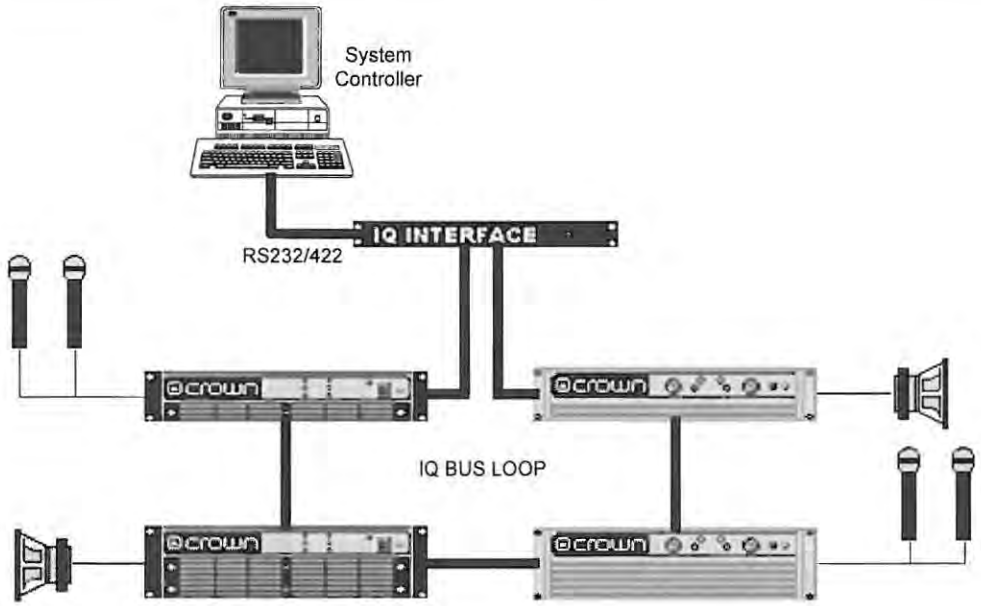

Figure 2.4: IQ Audio System Layout

2.2.1 Software

The software provided for the IQ Audio System consists of an IQ server and clients. A server is defined as being the PC that is directly connected to an IQ interface device, and runs the IQ Server software program [Crown Audio, Inc., 2001c]. Its primary function is as a gateway device between the clients and an IQ system. A client is described as a PC that is running the IQ for Windows program, and may be connected to the server through conventional TCP/IP networking [Crown Audio, Inc., 2001b]. It is this connection that allows the remote clients to control the system devices connected to the server. Both of these applications are 32-bit applications written specifically for the Microsoft Windows platform, and utilise the provided networking facilities [Crown Audio, Inc., 2001a]. As the IQ Audio System design philosophy utilises the existing TCP/IP technology it is of immediate interest in this study as it can easily exploit the IP over 1394 drivers within the Microsoft Windows Millennium and XP operating systems, without any need to alter or rebuild the IQ software.

The IQ server provides a number of support functions to facilitate its role as a gateway, and as a manager of system devices. These functions are [Crown Audio, Inc., 2001d]:

- The addition of new system devices to a specified loop.

- The removal of system devices from a loop.
- The ability to perform a roll call, which is the querying of the connected bus loops to determine which system devices are connected.
- An upload capability, which updates the system database to reflect the state of the connected system devices.
- The addition of new clients.
- Facilities to restrict the access criteria of connected clients.
- A log of client transactions.
- A facility to send network messages to connected clients.
- Sending notifications to clients when the server comes online and before it goes offline.

The IQ Server software is continuously being upgraded and new functions added, with the above list of functions pertaining to the IQ Server Version 5 software. An illustration of how the IQ Server software interacts with the clients is provided in figure 2.5 [Crown Audio, Inc., 2001d]. This figure also illustrates the use of traditional Ethernet technology within the system, through the use of an Ethernet Hub, and Ethernet cabling that is represented by the broken line.

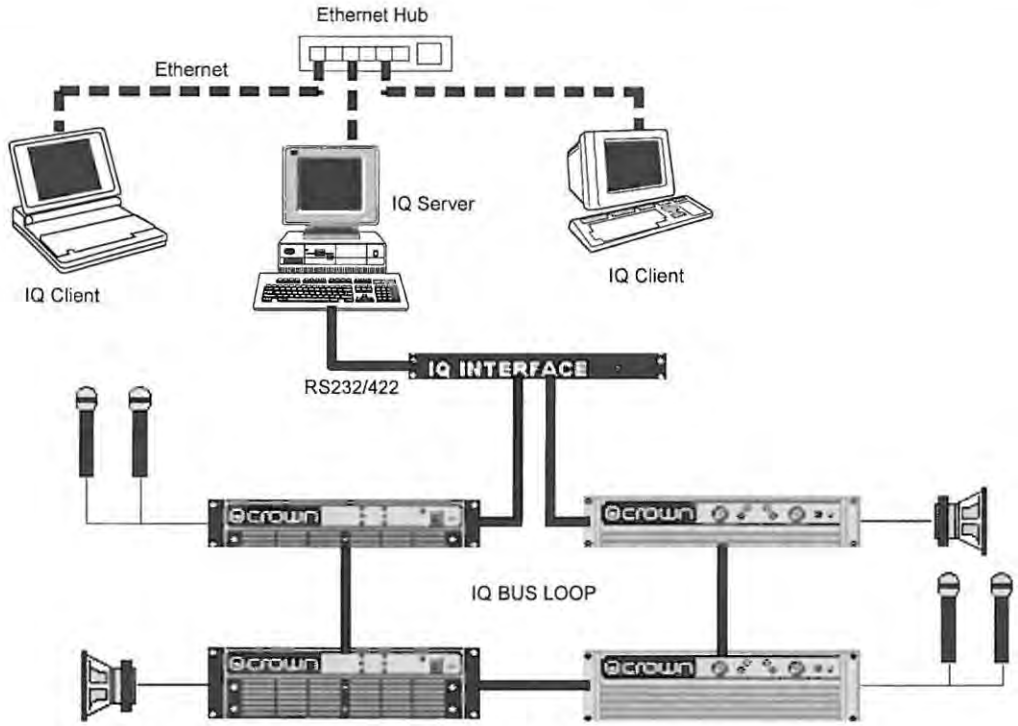

Figure 2.5: IQ Audio System Layout including IQ Servers and Clients

The IQ for Windows clients allow the setup, control, and monitoring of components present in the IQ system [Crown Audio, Inc., 2001a]. There are four main parts to the IQ for Windows software [Crown Audio, Inc., 2001b]:

- The main window
- Dataframes
- The workplace window
- Control panels

The main window is similar to the interface found in other windows programs, and consists of a menu, tool-, and status bar. It is from this window that new dataframes can be created and managed. A dataframe can be described as a snapshot of the system at a given point, and provides graphical representations of the audio devices present within the system [Crown Audio Inc., 2001b]. These dataframes can be saved to disk, and retrieved when needed. The graphical representations of these dataframes are presented in the workplace window, which allows the user to view the control panel for a device by simply double clicking on the icon that is its representation. This control panel contains graphical control and monitoring elements such as sliders, buttons, and level meters. Figure 2.6 illustrates a control panel that is associated with

a Crown SMX-6 mixer. When these controls are manipulated, appropriate messages are sent to the IQ Server, and then distributed across the IQ Bus to the relevant device.

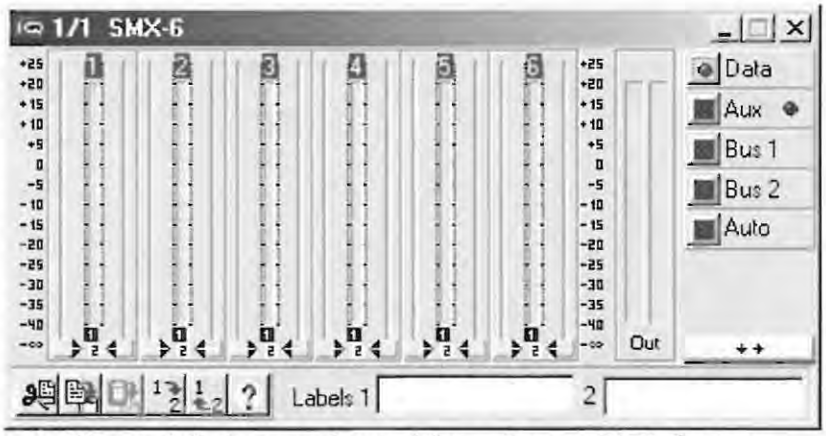

Figure 2.6: A control panel for a Crown SMX-6 mixer

The Client Software provides a tremendous amount of flexibility and customisation in the user interface that is used to control the device, to the extent that the user can assign custom images and control tabs to each device [Crown Audio, Inc., 2001a]. However, of importance to this study is the manner in which the client devices fit into the IQ Audio System, and exert control over the relevant system devices. Referring back to figure 2.5, it can be seen that all of the clients are connected via Ethernet to the IQ Server, which in turn interfaces with the actual devices. This illustrates the gateway functionality of the IQ Server in relaying messages to the relevant system device in the appropriate format, via the System Interface.

2.2.2 IQ Loop

The IQ Loop forms the core of the control network, as it connects the system devices to each other and to the System Interface. The loop is an opto-isolated, digital, half-duplex, asynchronous, 20ma current loop running at 38,400 baud [Crown Audio, Inc., 1996b]. Data is transmitted in a non-return-to-zero (NRZ) format utilising 10-bits, comprising 8-data bits, 1-start bit and 1-stop bit. Daisy-chaining all system devices from the output of the system interface back to the interfaces input connects the current loop, which was chosen for its high immunity to noise [Crown Audio, Inc., 1996b].

All devices present on the loop provide loop-thru connections for daisy-chaining, and are fitted with drop-out relays providing loop continuity should a device on the loop suddenly loose power. There are length restrictions on the length of the bus, and the published specification recommends anywhere between 300 to 3000 feet with 1000 feet as the "typical" length, although there are ways to increase the length of these loops, such as the introduction of fibre optic cabling [Crown Audio, Inc., 2001e]. The type of cabling normally associated with IQ loops is twisted pair with a very low capacitance. The connectors used include RJ45 connectors, 4 or 5-pin DIN plugs, and Euro-Style terminal blocks [Crown Audio, Inc., 2001e].

2.2.3 IQ Communication Protocols

The protocol associated with the initial IQ System was called the *FE-FF* protocol, but this protocol has been updated to the more robust UCODE protocol [Crown Audio, Inc., 1996a]. The UCODE protocol has been designed to be portable to any computer/control platform, and makes minimal assumptions about the lower level transport mechanisms associated with it. Instead the protocol defines a general-purpose communication language and its usage within the context of controlling and monitoring IQ system devices [Crown Audio, Inc., 1996a]. This suggests that the protocol can be utilised above various transport mechanisms, thereby allowing for the control and monitoring of devices on disparate networks.

The UCODE protocol describes a device in terms of objects, which are defined as controllable parameters within a device. Associated with these objects are classes and data types. A system device that has been designed with these UCODE objects can be described by an Object Information File (OIF), which is essentially a map of the devices' objects. The type of information provided in the file includes [Crown Audio, Inc., 1996b]:

- Device Objects
- Data Types
- Data Ranges
- Data Step Sizes

The format for the OIF is a text file following a standardised format as illustrated in listing 2.1. These OIF's can be utilised by a generic controller to "learn" the capabilities of the connected system devices. This is done by means of a system roll call, in which all connected devices respond with a Device Type identifier, UCODE version string, and Device Address. With this information the controller can search its database for an OIF matching a particular device and can display a user interface to enable the control and monitoring of that device [Crown Audio, Inc., 1996b]. Listing 2.1 provides an extract of an OIF for a Crown SMX-6 mixer, and illustrates the representation of devices within the IQ Audio System.

```
[Component]
ID=$0A
Description=Crown SMX-6
DLLName=MULTDLL.DLL
DefaultBMP=SMX-6.BMP

[Object1]
ID=1
Panel=User Label 1
Class=StringControl
Type=String
CutCopyLockout=1

[Object2]
ID=2
Panel=User Label 2
Class=StringControl
Type=String
CutCopyLockout=1

[Object3]
ID=3
Panel=Data LED
Class=BinaryControl
Type=Logical
DeviceMin=0
DeviceMax=1
Default=0
```

Listing 2.1: Example Object Information File for a Crown SMX-6 device

2.2.4 TCP/IQ

In November of 2002 Crown Audio Inc. released a TCP/IP based version of the IQ Audio System, called TCP/IQ. This implementation provides the same facilities as encountered within the original system, with the exception that the IQ loop has been replaced with an Ethernet or CobraNet network. IQ devices are now able to attach

directly to this network without the need for an IQ Interface device. Devices present on the TCP/IQ and the original IQ loop implementation can communicate through the use of a TCP/IQ gateway, which presents the loop devices to the IP network as TCP/IQ components.

As a result of the introduction of IP networks, the addressing scheme associated with IQ devices has been modified. Previously, a device was identified using its type, the number assigned to the loop on which it resided, and its hardware-settable address. The TCP/IQ approach requires that devices be identified by a software-settable IQ Network address which uniquely identifies a device.

2.3 QSControl System

The company QSC Audio Products, Inc. has developed the QSControl System, which is a powerful and flexible system that allows the control and monitoring of QSC amplifiers and other audio equipment via Ethernet [QSC Audio Products, Inc., 2000a]. The QSControl system consists of a System Controller, which is a Pentium based PC running QSControl software, connected to a CM16a Amplifier Network Monitor using standard Ethernet, as shown in figure 2.7 [QSC Audio Products, Inc., 2001b].

A CM16a Amplifier Network Monitor provides audio input, audio output, audio monitoring, amplifier management, and status control of up to 16 amplified channels for the attached QSC amplifiers. The CM16a and the attached amplifiers communicate via a DataPort connection, which is a standard Video Graphics Adapter (VGA) 15-pin port and cable [QSC Audio Products, Inc., 2000b]. This connection is utilised for the transmission of two amplifier input audio signals as well as control and monitoring information between the CM16a and an attached amplifier. An example of the layout of a typical QSControl network is illustrated in figure 2.7.

Source: http://www.qscaudio.com

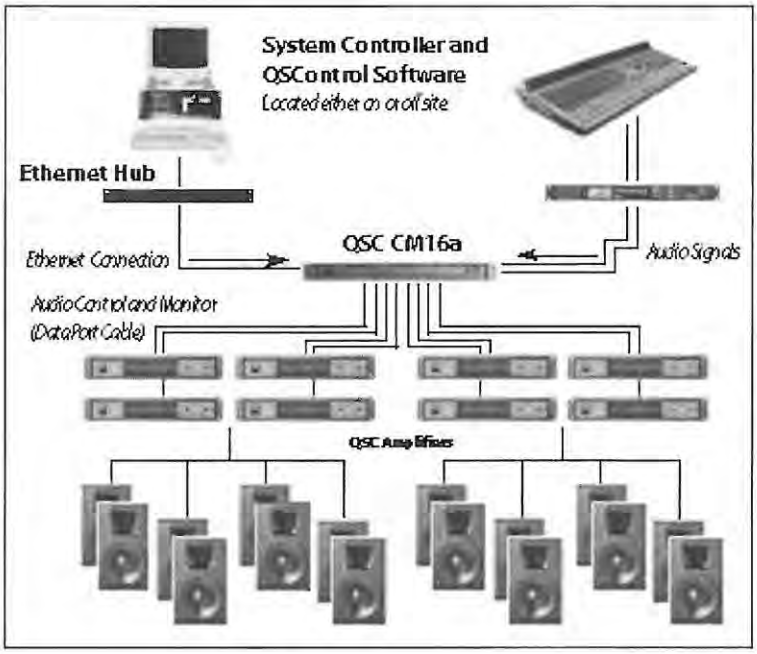

Figure 2.7: A simple QSControl network

The amplifier management facilities provided by the CM16a, include [QSC Audio Products, Inc., 2000b]:

- AC standby/operate mode control,
- AC power state indication,
- Temperature metering,
- Amplifier gain settings,
- Over-temperature detection,
- Stereo/parallel/bridge-mono indication,
- Amplifier model detection, and
- Protect status detection.

2.3.1 Software

The CM16a units are controlled using the System Controller, for which there are two options available [QSC Audio Products, Inc., 2000a]:

- System Manager Application, and
- Custom system

The System Manager Application was developed by QSC Audio Products Inc. as a generic application for testing, diagnostics, and the setup of an audio system connected to a QSControl network. It provides an intuitive user interface and has been designed to monitor and exercise every function within a CM16a device and its connected amplifiers [QSC Audio Products, Inc., 1997a].

The custom system consists of a number of Microsoft Visual Basic type plug-ins, such as ActiveX and COM controls, that allows control over the CM16a's and the attached amplifiers. The benefits of creating custom applications include the creation of a number of presets, the execution of pre-programmed functions, control over non-QSC equipment within the same application, automation, the provision of emergency functions, and security restrictions. An example of the benefits provided by this approach is the ability to provide a graphical representation of the facility in which the sound system is installed. This simplifies the use of the system, and provides an intuitive user interface. Figure 2.8 provides an illustration of a custom church application that was developed by QSC Audio Products Inc., as an example application, and demonstrates the versatility of the QSControl system. This example consists of a number of predefined zones, and when a particular zone is selected, the control panel for that zone is displayed, as shown in figure 2.9. This control panel allows the manipulation of the volume levels of the connected amplifier, as well as the selection of preset levels.

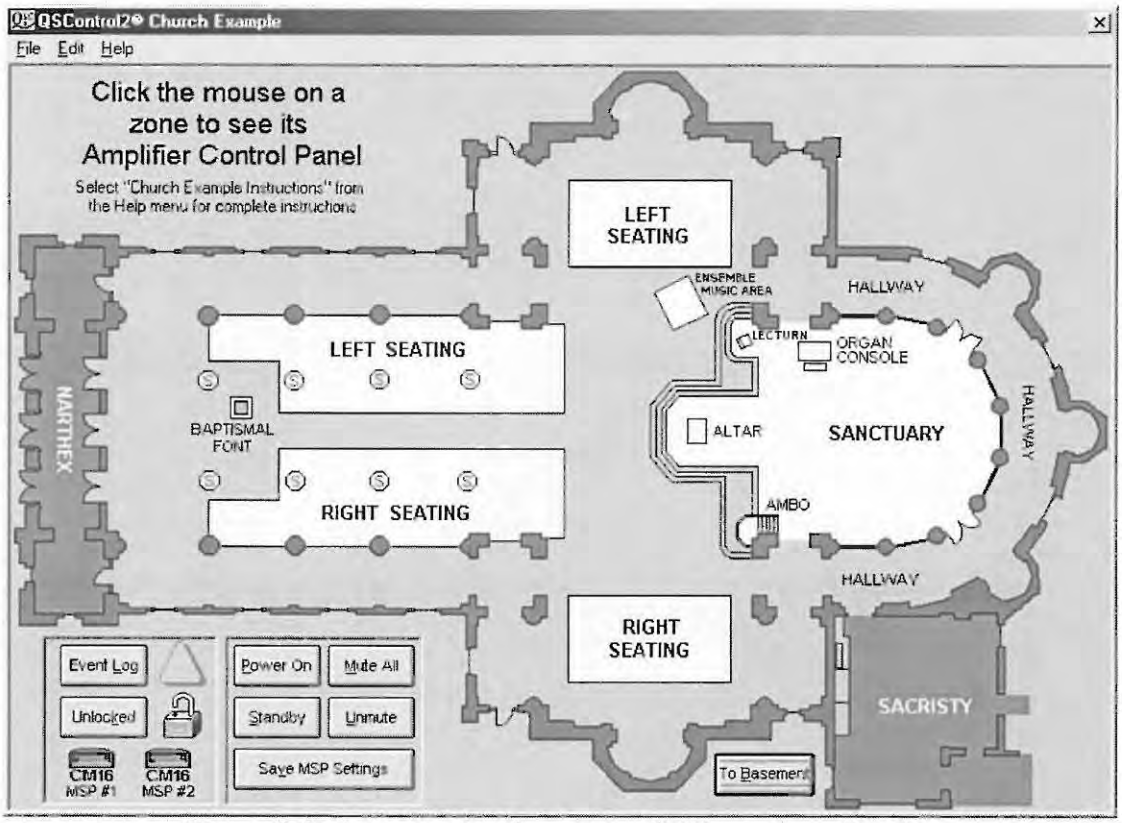

Figure 2.8: An example of a custom QSControl application

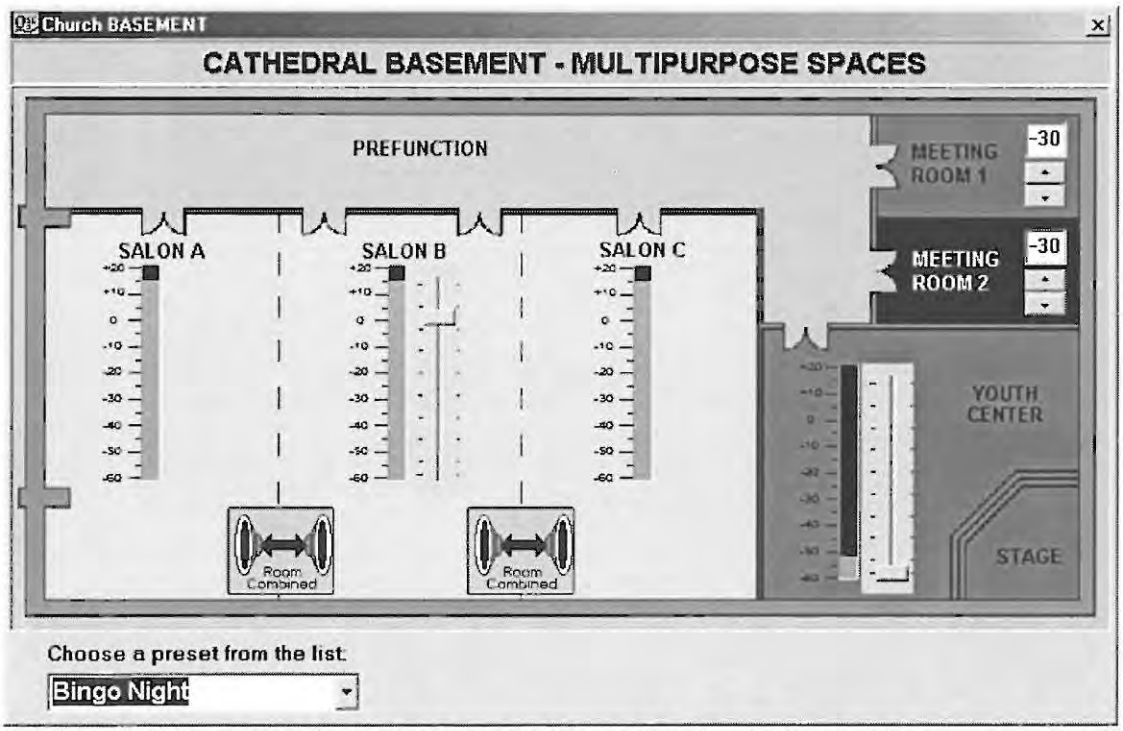

Figure 2.9: An example control panel for the QSControl System

The design philosophy of the QSControl software is similar to the IQ Audio System, in that extensive use is made of the networking facilities already provided within the Microsoft Windows Operating System. For example, the existing TCP/IP and

UDP/IP networking facilities are utilised for the transmission of control and monitoring information over Ethernet networks. Through the introduction of TCP/IP networking drivers for the IEEE 1394 bus to the Microsoft Windows Operating System, the QSControl system is able to utilise IEEE 1394 networks for the transmission of control and monitoring information. The ability to utilise existing resources was one of the motivating factors for the choice of the QSControl system as the system to adapt to an IEEE 1394 environment. In addition, the ability to develop custom applications for the QSControl system, allows for the integration of a number of other audio systems and devices, such as the inclusion of MIDI devices, RS-232 devices, Ethernet devices, etc.

2.3.2 Audio Distribution

The transmission of audio data over the QSControl system, as illustrated in figure 2,7, is achieved through the use a dedicated audio network, such as a Peak Audio Products, Inc., RAVE network, and not via the Ethernet link. This dedicated network allows for the transmission of multiple channels of audio signals over standard Ethernet hardware and cabling, through the utilisation of Peak Audio's, Inc. Cobranet technology, which will be discussed in section 2.4.2.

2.4 Common Characteristics

The previous sections introduced three sound systems that are currently available and have proven successful in industry. The motivation behind this section is to analyse these systems, and highlight the facilities that are common to all. In essence, the building blocks of sound systems are being extracted in order to build a UML model that would provide a basis for future sound system design.

From the three systems discussed, there are three core facilities that can be identified. These are:

- Audio Facilities
- Routing Facilities
- Control and Monitoring Facilities

2.4.1 Audio Facilities

The audio devices typically provide the audio facilities that are needed within a sound system. These facilities can be subdivided according to the type of audio device [Laubscher et al, 2000]:

- Generators
- Transformers
- Renderers
- Recorders

Generators are the sound sources in a sound system, examples of which are synthesisers, samplers, musical instruments, and live performers. The transformers process the generated audio, and include devices such as effects units, and mixers. The rendering devices are responsible for the "display" of the audio, of which speakers are a typical example. The recording devices, such as tape recorders and multi-track hard drives, provide a permanent means of audio storage and retrieval.

2.4.2 Routing Facilities

Routing within a sound system is the ability to send and receive audio signals by wiring the outputs of audio devices to the inputs of other audio devices. These signals can be in an analog or a digital format, and traditionally a cable run has been used for each channel of audio to achieve this. However, if a change in configuration is needed, it involves the rewiring of devices. The introduction of an analog patchbay solved this problem, by allowing inputs and outputs to be easily reconfigured. The patchbay functions by terminating all the inputs and outputs of the networked audio devices at a central location, and allows "patches" to be made between any desired analog input and output. Figure 2.10 illustrates a typical analog patchbay, where the patch cable, can be used for the bridging of input to outputs, with the inputs residing on the upper level, and the outputs on the lower. Many of these analog patchbays have internal patches between the input and output of the same channel number, for example the input of channel 3 would be internally routed to the output of channel 3. If, however, a patch cable is inserted, this internal routing is broken, with the resulting patch taking precedence.

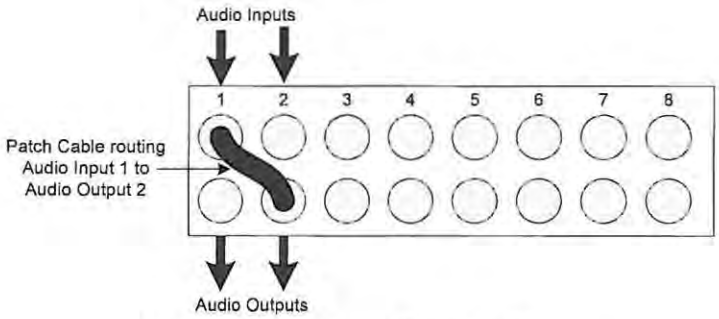

Figure 2.10: An analog Patchbay

With the introduction of the PC into the sound system environment, the popularity of software routing is increasing, whereby a controller PC can be attached to a patchbay, and the necessary routings set up through appropriate software. An example of such a patchbay is the Midiman DigiPatch 12x6 Audio Patchbay. This device allows the patching of digital audio signals in either S/PDIF (Sony/Philips Digital Interface Format) or ADAT (Alesis Digital Audio Tape) format, through the device front panel or a controller PC running the DigiPatch Panel software [Harmony Central, 1999]. The advantage of this approach is the elimination of plugging-and-unplugging of cables, as the routings are done within the patchbay device. This reduces the number of problem points in studio environments as the number of cables is reduced. The above example also introduced the routing of digital audio signals, which are inherently more difficult to route. This is due to a number of different formats existing that are often incompatible with each other.

Figure 2.11 illustrates a PC-controllable patchbay, whose audio inputs and outputs can be either in digital or analog format. PC software will transmit connection information, according to user selections, to the audio patchbay, which will set up the necessary routings within the actual device.

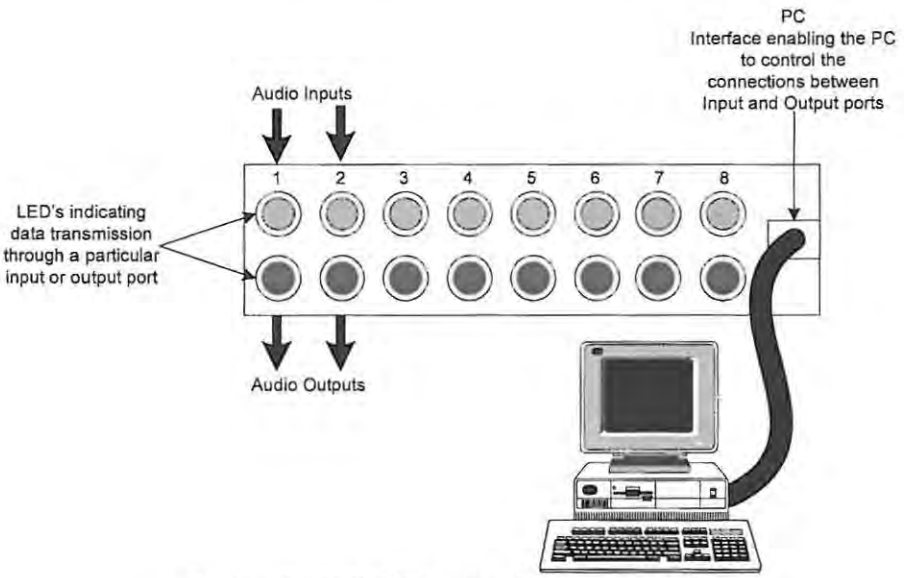

Figure 2.11: A PC-controllable patchbay

Whether the audio signal is in analog or digital format has a significant impact on how the signal is routed. Generally analog audio signals are more expensive to route, as they require numerous cable runs, whereas digital audio signals are easily multiplexed, allowing more than one or two audio channels per cable. Major advantages of digital over analog audio signalling include high immunity to noise and the error detection and correction facilities, ensuring that the data can be verified before it is processed or rendered.

It is also possible to translate between various digital audio formats or perform digital signal processing (DSP) on the audio signal before it is routed. An example of this is the MediaMatrix system, illustrated in figure 2.12, where analog audio input is sourced at a particular port. This audio is passed through the MediaMatrix system, which performs user defined processing on the signal before it is routed back to a BoB, and output to an attached device. In such a system, there is no default relationship between inputs and outputs, and all routings must be manually set up, in this case using the provided MediaMatrix software.

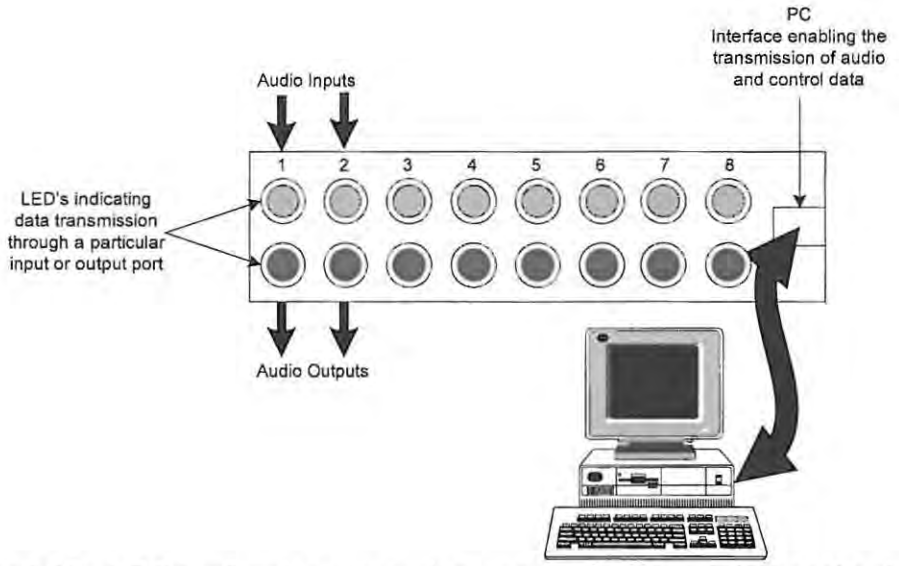

Figure 2.12: A PC performing processing on an audio signal, before the signal is routed through the patchbay

The final type of routing system, which is most applicable to the IEEE 1394 environment, and is found in all three of the examples previously discussed, is CobraNet. CobraNet is a routing technology that has been developed by the Company Peak Audio Inc. [Peak Audio Inc., 2001a].

2.4.2.1 Routing Technology - CobraNet™

CobraNet was developed to allow numerous audio channels to be transmitted utilising existing Ethernet technology, and is comprised of a CobraNet Interface, and a network protocol [Peak Audio Inc., 2001c]. These additions to Ethernet provide isochronous data transport, sample clock distribution, and a vehicle for the transport of control and monitoring data such as the Simple Network Management Protocol (SNMP).

The CobraNet interface consists of the actual CobraNet hardware and software supplied by Peak Audio Inc. Traditional physical and link Ethernet layers are still present on the interface hardware, but a digital signal processor (DSP) and a few support components have been added. This DSP forms the heart of the CobraNet interface, as it implements the network protocol stack, performs isochronous to asynchronous conversion, and plays a part in sample clock generation [Peak Audio Inc., 2001c].

Three packet types make up the CobraNet networking protocol, namely the beat, isochronous data, and reservation packets. The beat packet is directed at a multicast address and carries the network operating parameters, clock, and transmission permissions. Only one device on the network will transmit beat packets, but all devices on the network are required to listen for these packets. The beat packet is used to indicate the start of an isochronous cycle, and all devices are required to lock their sample clocks according to this packet. It is therefore essential that this packet be delivered timeously, as delays could prevent devices locking to the master clock [Peak Audio Inc., 2001c]. The size of this packet is generally small, but can be large if the network is highly active. Figure 2.13 provides an illustration of services provided by CobraNet, and the manner in which these services reside within the Ethernet network.

Although it is theoretically feasible that both normal Ethernet traffic and CobraNet traffic be transported over the same repeater network simultaneously, in practice it is found that the network quickly saturates. As a result when CobraNet technology is being utilised on a repeater network, other Ethernet traffic, such as data transmissions, are not recommended [Peak Audio Inc., 2001a]. However, switched networks are more flexible and allow the transmission of normal Ethernet traffic and CobraNet data, although the network design is an important determinant of the carrying capacity of the network. The reason for the switched network being able to carry the CobraNet data and other Ethernet traffic is due to the intelligence built into most modern switches [Peak Audio Inc., 2001a]. For example, the majority of CobraNet traffic is isochronous data, which is either multicast or unicast traffic. Modern switches have facilities to filter certain multicast traffic and direct unicast traffic onto relevant segments, thereby reducing the load on the network.

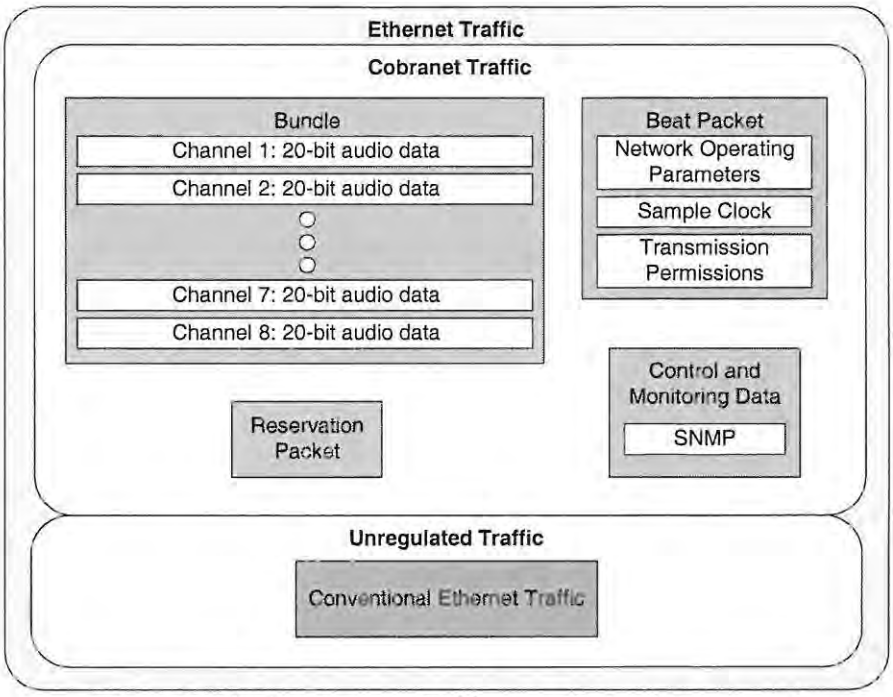

Figure 2.13: Services available on a CobraNet network

The isochronous data packet is the carrier for the audio data, and is subdivided into bundles. A bundle is the smallest network audio routing envelope, with a capacity for transmitting between 0 and 8 audio channels. There are two types of bundles defined, namely multicast and unicast bundles. Multicast bundles have one transmitter and many receivers, while unicast bundles are directed messages from one transmitter to one receiver. Due to the possibility of isochronous data packets being received out of order, they are buffered to allow the original sequence to be reconstructed [Peak Audio Inc., 2001c].

Nodes wishing to receive particular bundles issue reservation packets indicating to the transmitter whether or not unicast or multicast bundles are appropriate. The process of the transmitting node identifying receiving nodes is termed "reverse reservation" [Peak Audio Inc., 2001c].

Audio routing, depicted in figure 2.14, is achieved through the use of 64 internal audio routing channels, of which channel 0 has been designated as a special channel to indicate no routing. Channels 1 through 32 are used for audio destined for the network or from local audio input ports, and channels 33 to 64 are reserved for audio received from the network or destined for local audio output ports. A collection of

variables, known as the audioMap, maintains the mapping of internal audio channel numbers to local input and output connections, although this is generally preset according to the products specifications. Bundles are the transport mechanism for network channels, which have no direct correlation to audio channels. Instead bundle transmitters and receivers are responsible for the mapping of audio channels to and from network channels respectively.

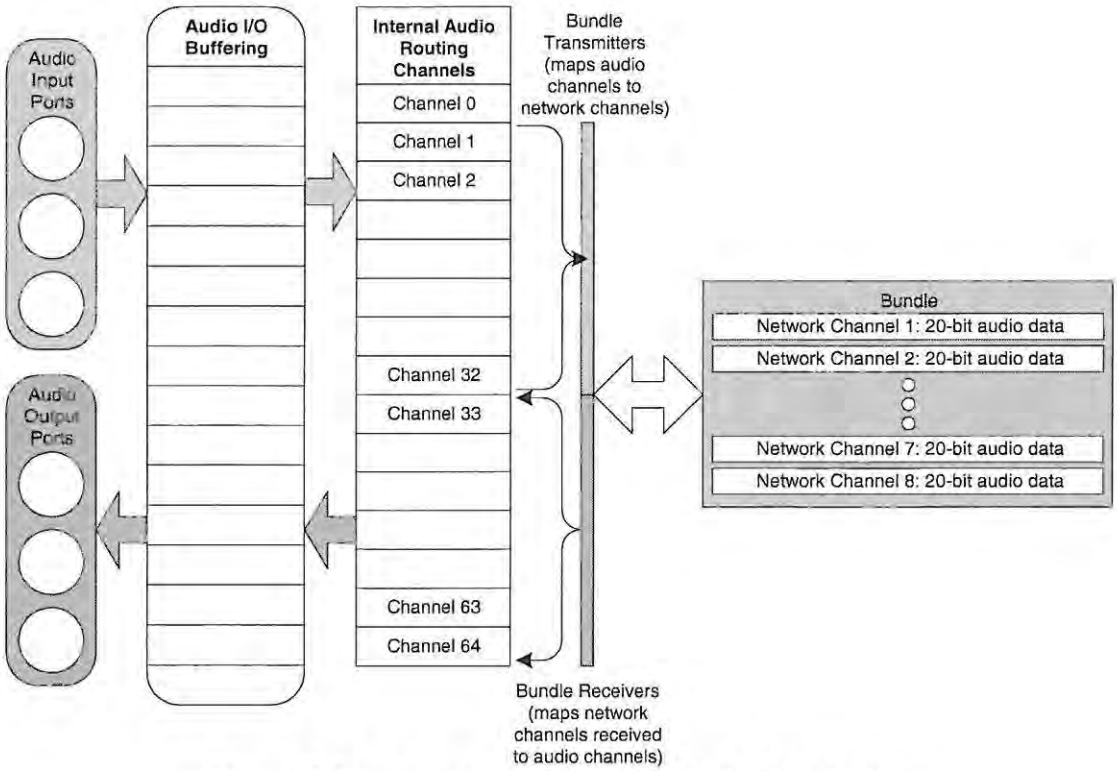

Figure 2.14: Audio Routing within a CobraNet Device

There are a maximum of four permissible bundle transmissions for each CobraNet device, the identification number for each bundle may be defined by altering the appropriate *txChannel* variable associated with that particular bundle. The 8 audio channels that will be encapsulated within this bundle are defined within 8 *txSubMap* variables, with each variable providing a value for a particular input. For example, had a device been configured to utilise only two bundles and to transmit all 8 if its audio inputs in both bundles, the variable values would appear as shown in table 2.1.

Bundle 1

Name	Value
txSubMap.1.1	1
txSubMap.1.2	2
txSubMap.1.3	3
txSubMap.1.4	4
txSubMap.1.5	5
txSubMap.1.6	6
txSubMap.1.7	7
txSubMap.1.8	8

Name	Value
txSubMap.2.1	1
txSubMap.2.2	2
txSubMap.2.3	3
txSubMap.2.4	4
txSubMap.2.5	5
txSubMap.2.6	6
txSubMap.2.7	7
txSubMap.2.8	8

Bundle 2

Table 2.1: Example txSubMap values

The *Name* field in table 2.1 provides not only the name of the SNMP variable, but also specifies which of the two bundles is currently being modified, indicated by the first digit. The second digit indicates the position within this bundle that is being set, with the *Value* field containing the internal audio routing channel that will be transmitted. Figure 2.14 provides the logical audio routings that are present within a CobraNet enabled device, including the internal audio routing channels. Note from figure 2.14 that the internal routing channel's 1 through 32 are reserved for audio that is to be transmitted over the CobraNet network, and channel 33 through 64 are reserved for audio received from the network. Channel 0 is reserved to indicate that no routing is present.

A similar approach is followed for the reception of bundles as the *rxSubMap* variable has the same format as described for transmissions. However, the channel numbers specified within the Value field will contain a value in the range from 32 through 64 which are reserved for the reception of audio data.

2.4.3 Control Facilities

The effective management of audio devices is a crucial component within any sound installation. This management is achieved through the use of control facilities which provide for the monitoring of critical parameters, such as temperature and clip metering, and for controlling devices attached to the network. Due to the importance of these facilities, many audio companies have developed proprietary control and monitoring schemes suited to their own needs, which has led to a lack of interoperability between devices from various manufacturers. There have been

attempts to produce a standardised protocol, an example of which is AES-24 [Audio Engineering Society, 1997], but audio manufacturers have been understandably reluctant to part with their proprietary protocols due to large time and financial investments.

The particular control and monitoring facilities needed are determined by the complexity, level of control, and the types of audio devices desired on the network. A good example of an extensible control technology that can be adapted to most networks is the QSC-24 protocol [QSC Audio Products, Inc., 1997b], which is a derivative of AES-24. QSC-24 has been developed by QSC Audio Inc. and is used in the QSControl example previously discussed.

2.4.3.1 Control and Monitoring Technology – QSC-24

QSC-24 has its origins in AES-24, which was an attempt to create a standardised protocol for the control and monitoring of audio devices from disparate manufacturers, over any digital data network [Audio Engineering Society, 1997]. QSC-24 is a subset of this original specification, but has maintained the flavour of extensibility offered by the original AES-24 specification, and adheres to the core protocol features [QSC Audio Products, Inc., 1997b].

The QSC-24 protocol is based on the object model proposed within the AES-24 specification, which defines a device as a collection of individually addressable objects, with each object adding some capability to the device. Therefore devices are described according to the objects they are comprised of, as illustrated in figure 2.15. These objects consist of methods that control device behaviour, and are remotely accessible by other devices and control points. It is the invocation of these methods that provides the functionality of devices attached to a QSC-24 network. These methods are typically divided into Get and Set operations that act on specific variables in the object. These variables represent the state of that object and ultimately the state of the device.

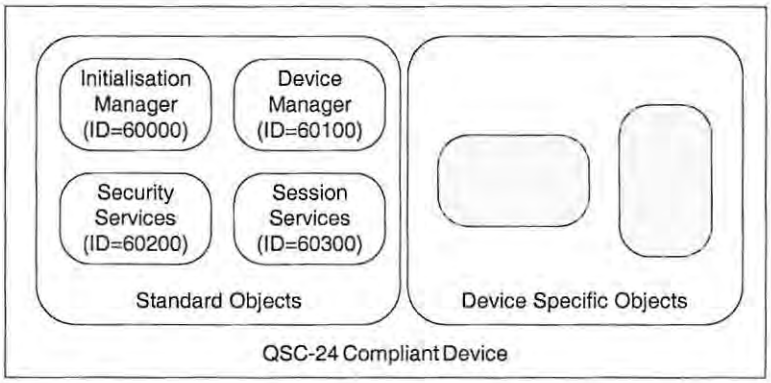

Figure 2.15: The device model associated with QSC-24 compliant devices

Various levels of addressing have been provided that allow the addressing of devices, objects, and methods. It is the combination of these addressing levels that allow individual control over the objects contained within a device.

A simplification of the original AES-24 specification introduced by QSC-24 is that the transport network has been defined to be TCP/IP [QSC Audio Products, Inc., 1997b]. This simplifies the addressing scheme, as nodes can be targeted using their assigned IP address, which eliminates the need for an address resolution service, as all objects within a device are addressed in an orderly manner according to a predefined object tree. A graphical representation of an extract of this object tree is provided in figure 2.16. The methods within objects are assigned addresses within the QSC-24 specification, thereby eliminating the possibility of addressing an incorrect method. The object tree and protocol are, however, extremely extensible, and as new objects are conceived they can be added to the tree without having to reassign addresses.

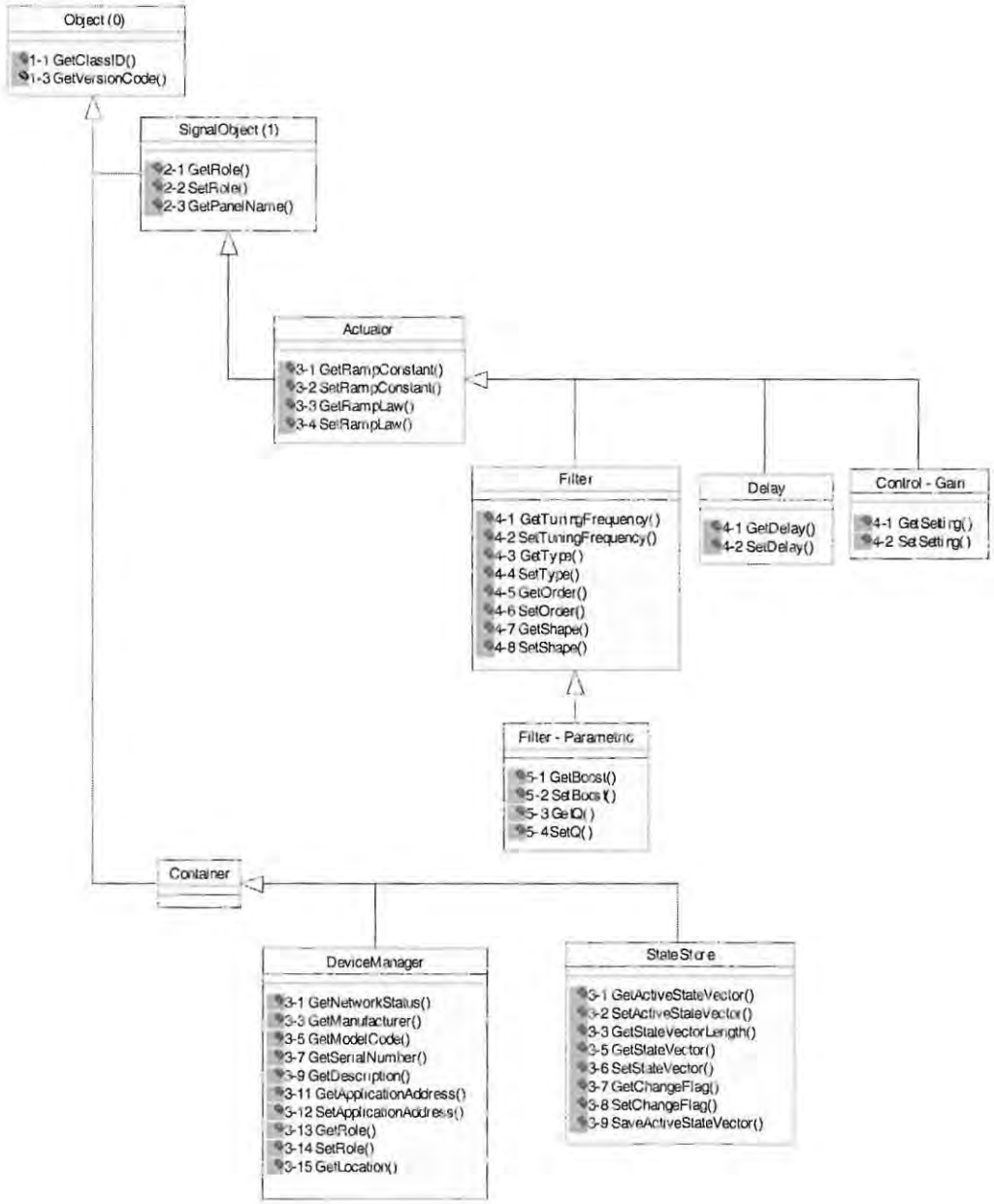

Figure 2.16: An extract of the QSC-24 object hierarchy

Messaging within QSC-24 is by default acknowledged, thereby providing confirmation of delivery and the return of a status code; however certain transactions are not acknowledged, such as high volume status updates. All QSC-24 messages, as shown in figure 2.17, are encapsulated in a Protocol Data Unit (PDU), which contains:

- Protocol Version Code – The version of QSC-24 being used.

- Transaction Type Index – The type of message, for example a command or acknowledged message.
- Reply-To-Object Address – The address to which replies should be directed.
- Destination Object Identifier – The address of the object that must receive the message.
- Destination Method Identifier – The method that is to be invoked within the addressed object.
- Method Parameters – Zero or more parameters that are to be passed to the called method.

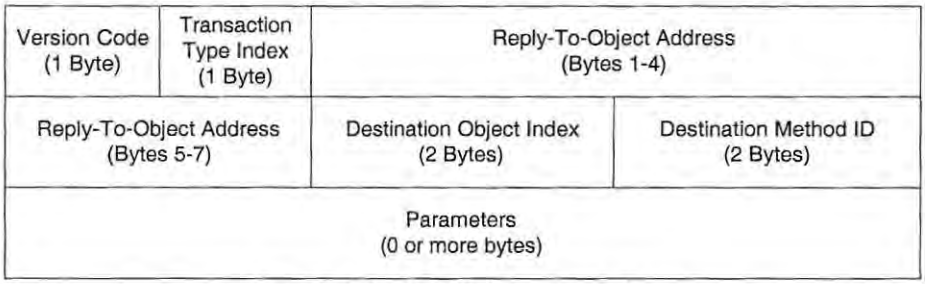

Figure 2.17: QSC-24 message format

A major advantage that QSC-24 has over other control and monitoring protocols is its ability to adapt to different platforms and industries. For example, the same communication protocol could be used in the lighting industry, with the addition of an appropriate object set. This highlights the underlying design principle inherited from AES-24, and that is to provide a protocol that was adaptable to various manufacturers needs without having to divulge trade implementation secrets. This is achieved by only exposing the necessary object interfaces to control points.

2.5 Object Oriented Model of a Sound System

The common characteristics described in the previous section can be further expanded through the use of an Object Oriented model, using the UML notation, to describe the basic level of control needed within a generic sound system. This is possible, as most sound systems have a common level of functionality, albeit at a very low level and at varied levels of performance. It is this functionality that is encapsulated in the UML model presented in this section.

All of the scenarios presented are taken from the perspective of the controller, and an audio device attached to the network. The controller is any device that allows the user to interact with the audio devices present within the sound system through the modification of variables within those devices. It is the responsibility of the controller to present a level of abstraction to the user whereby the user need not be aware of device capabilities or location. An example of a typical controller is a PC. It should be noted that although the controller is the point of interaction with the sound system, the capabilities of the controller are largely determined by the facilities provided within the selected control protocol. The same is true of the audio devices present, as they are dependant on the control protocol through which they are to communicate.

2.5.1 Use-case Diagram

The use-case diagram upon which the object model and scenarios are based is presented in figure 2.18. This diagram provides a high-level overview of the various facilities required within a sound system, such as the synchronisation of audio and the establishment of connections.

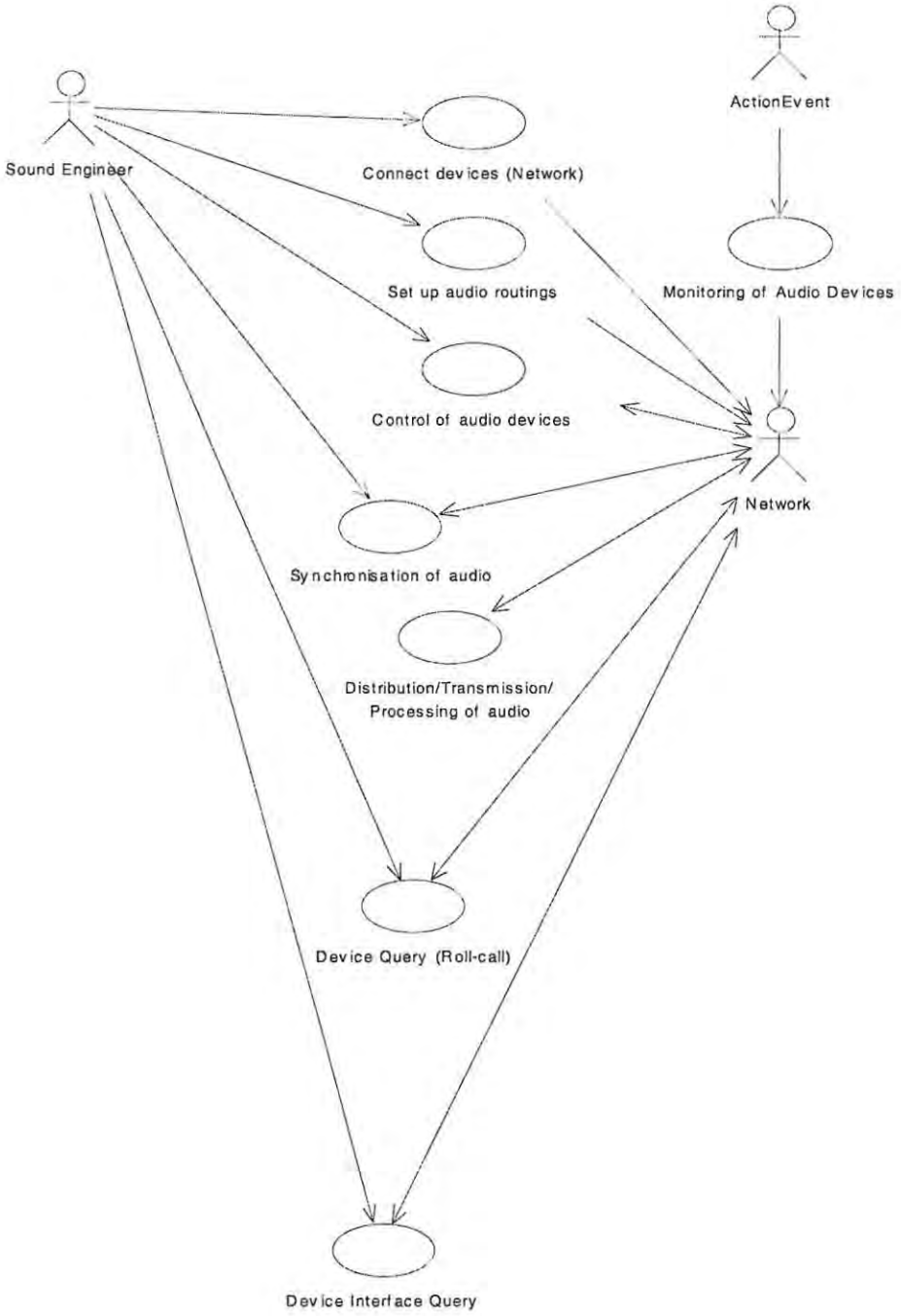

Figure 2.18: Use Case view

2.5.2 Scenarios and Sequence Diagrams

From the use-case diagram presented the following scenarios are extracted:

- Device Roll-call
- Device Capabilities Query

- Audio Routings
- Device Control
- Device Monitoring
- Device Interface

Each of these scenarios can be represented using textual descriptions, and sequence diagrams constructed from these descriptions. The textual descriptions extracted are included in Appendix A, while the completed sequence diagrams are included within the text for each of the above scenarios.

2.5.2.1 Device Roll-call

In order for audio devices and controllers to communicate, a mechanism is needed whereby all of the devices on the network can be discovered. This mechanism can be referred to as a device roll-call, and consists of a roll-call message being issued to all devices attached to the network. Devices that receive this message respond to the requesting node with information that uniquely identifies the responding node on the network.

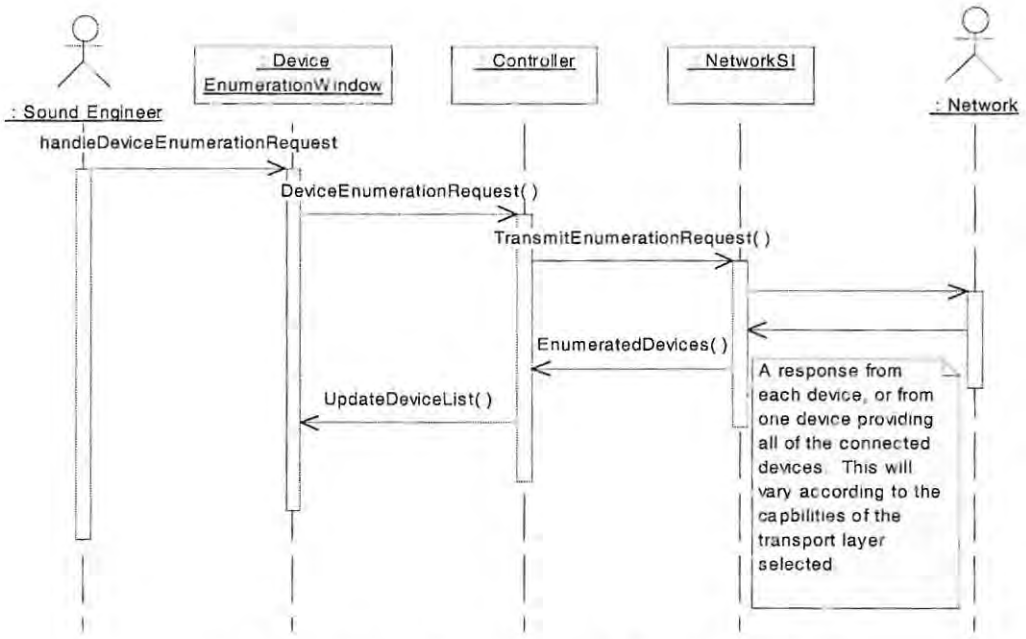

Figure 2.19: Audio device Roll-Call (controller)

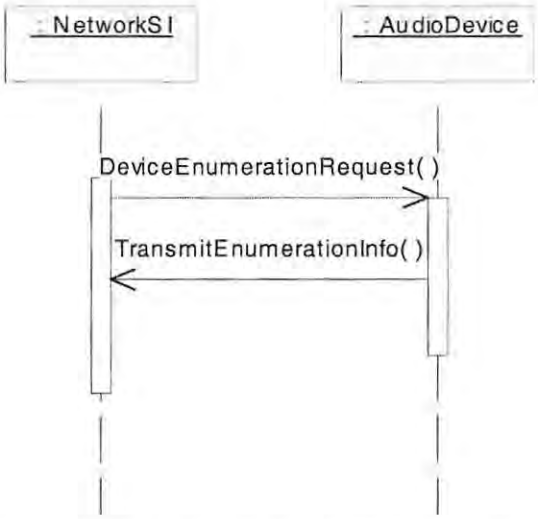

Figure 2.20: Audio device Roll-Call (audio device)

2.5.2.2 Device Capabilities Query

Once the controller has discovered all of the devices attached to the network, these devices can be queried to determine their capabilities. The controller may wish to perform a roll-call and query sequence upon network initialization, at the users request, or after devices are added to or removed from the network. It is this sequence that provides the necessary information that the controller requires in order to accomplish its responsibilities, and present an abstracted view of the network to the user.

There are a variety of formats currently used to describe the capabilities of devices attached to networks, examples of which are the configuration ROM in IEEE 1394 networks [Anderson, 1999], or XML descriptors utilized in Universal Plug-and-Play (UPnP) enabled networks [Microsoft Corporation, 2000]. It is essential that all connected devices are able to communicate these responsibilities in a coherent and standardised manner. Failure to achieve this will result in devices that are not able to communicate with other devices attached to the network.

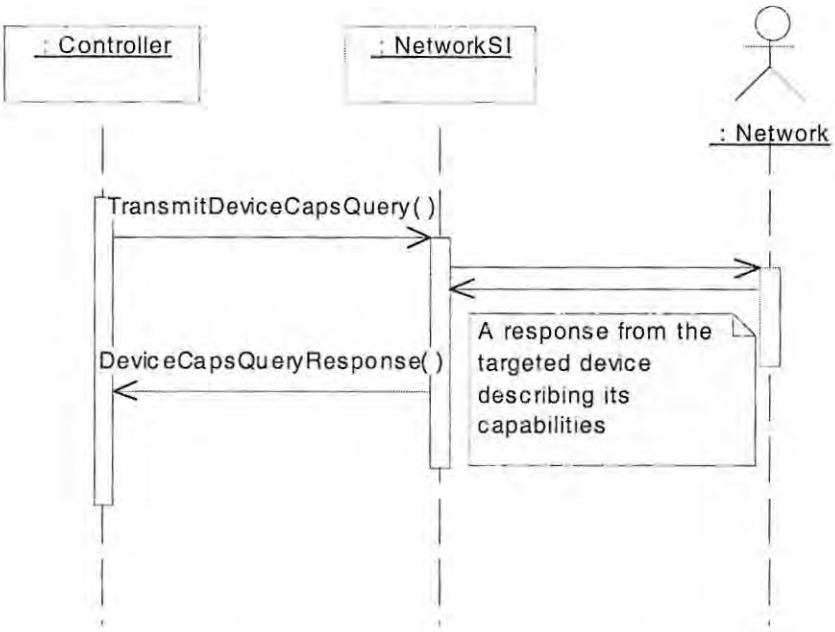

Figure 2.21: Audio Device capabilities query (controller)

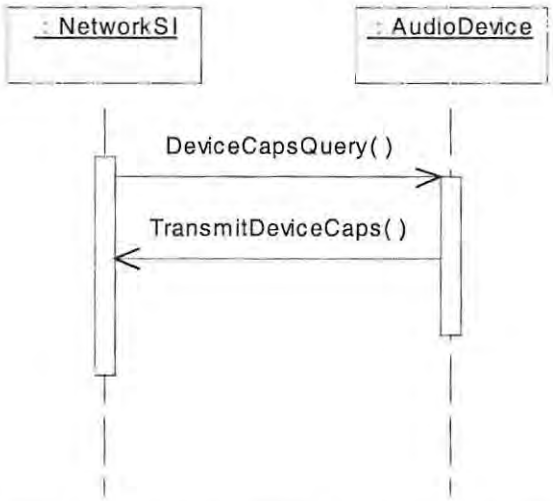

Figure 2.22: Audio Device capabilities query (audio device)

2.5.2.3 Audio Routings

Although the modification of parameters to facilitate the flow of audio could be classified as connection management rather than control, it is still the responsibility of the controller to ensure that these connections are established. This process involves the selection of an audio sink and source port, and the establishment of a connection between these two ports. The manner in which these connections are established is again network/device dependent. The Connection and Compatibility Management (CCM) facilities of AV/C are utilised within consumer IEEE 1394 devices for the establishment of these connections [1394 Trade Association, 2000]. Once a

connection has been established, all audio that is received on the source port, is automatically routed to the sink port.

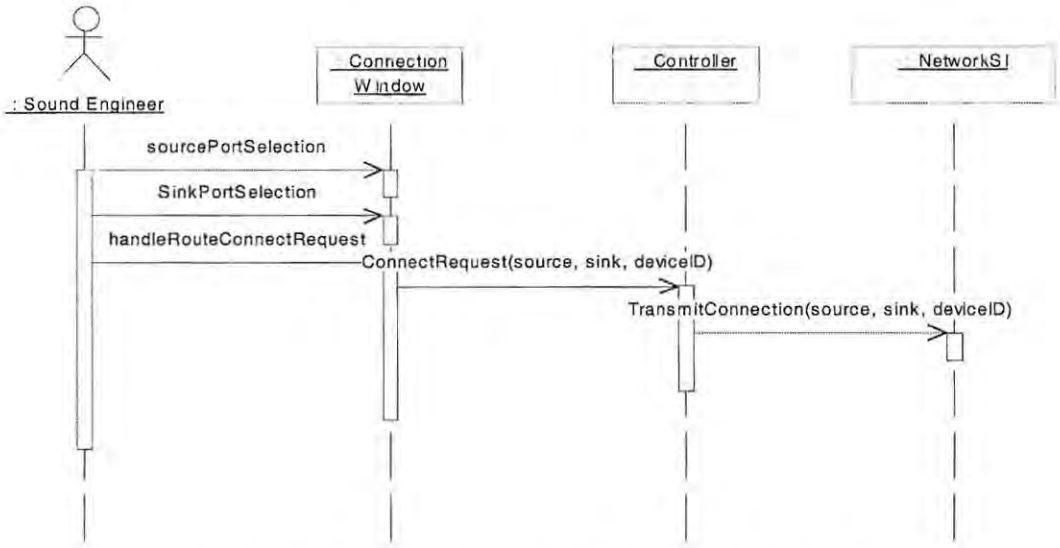

Figure 2.23: Establishment of audio routings (controller)

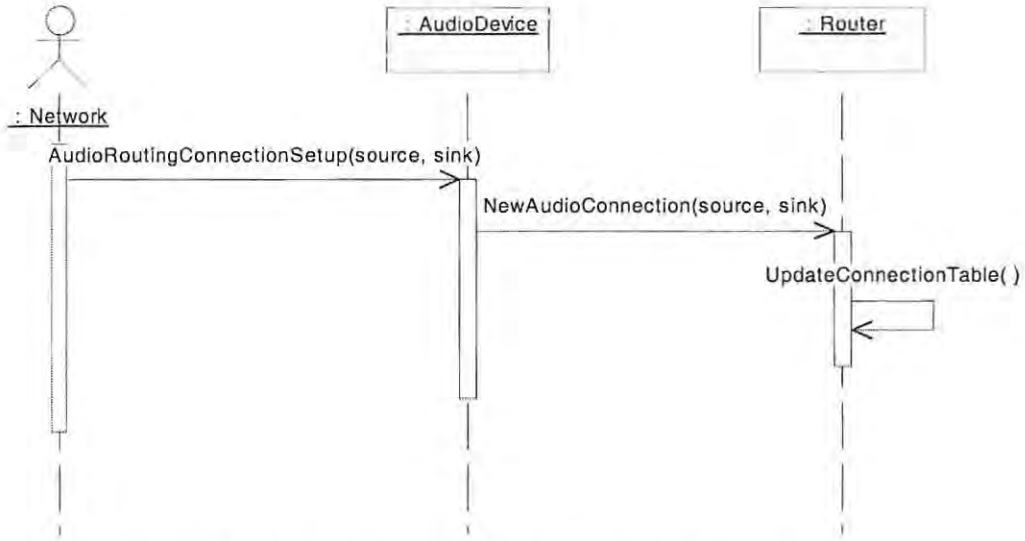

Figure 2.24: Establishment of audio routings (audio device)

2.5.2.4 Device Control

In order for a sound system to be dynamic, the attached audio devices must provide a number of variable parameters that control the behavior and state of the device. These parameters and the permissible level of control over them are encapsulated within the device capabilities transmitted to the controller. The controller alters these parameters

in order to manipulate and exert control over the device, which is achieved through the transmission of an appropriate parameter modification message.

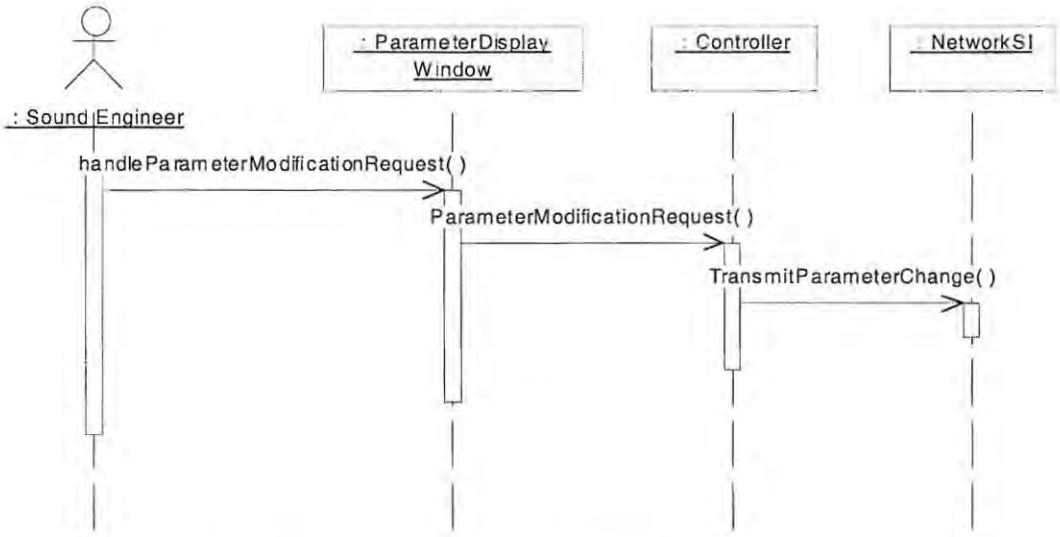

Figure 2.25: Audio device control (controller)

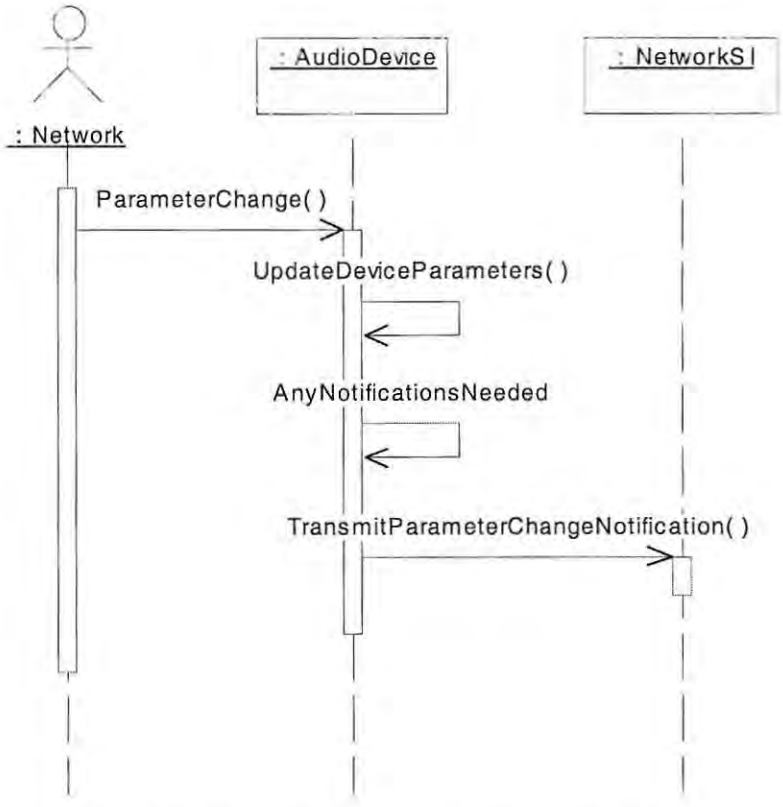

Figure 2.26: Audio device control (audio device)

2.5.2.5 Device Monitoring

This facility allows devices to report status information to other audio devices or the controller, usually upon a request for such notification. These notifications could include device failure, clipping, and other critical parameters. Typical uses of monitoring information reported by devices include the analysis of device failure, the disconnection of connections upon such a failure, and the display of data through a meaningful interface, such as a meter level.

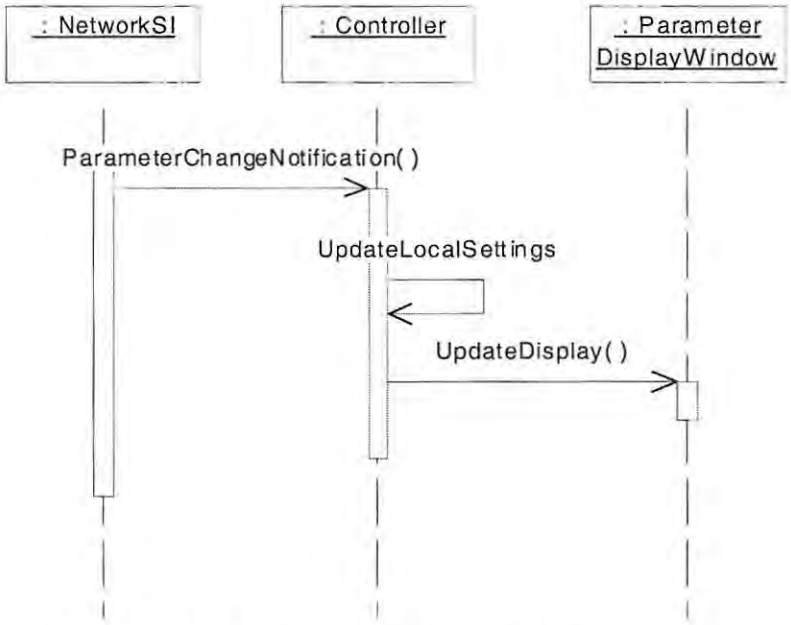

Figure 2.27: Audio device monitoring (controller)

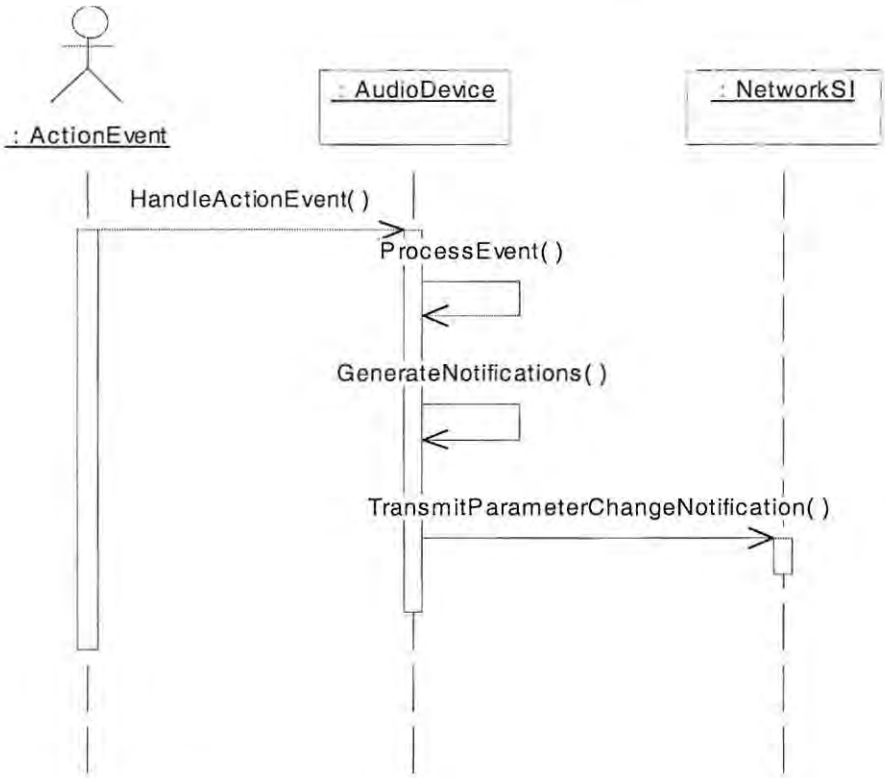

Figure 2.28: Audio device monitoring (audio device)

2.5.2.6 Device Interface

The above categories cater for the predominant functionality needed within a control protocol that is suited to a sound system. However the ability to represent a device within an appropriate or custom interface is often neglected. There are a variety of forms in which an interface can be represented, such as an HTML document (the approach taken within the UPnP architecture), a bitmap, or a device description language (DDL) such as utilised in the IQ Audio System, described in section 2.2.

The advantage gained by allowing devices to completely describe or provide their own interface's is that interoperability between controller and audio devices is increased, and the complexity of controllers is reduced.

2.5.3 Object Model

Through the construction of the above sequence diagrams the object diagram presented in figure 2.29 is established.

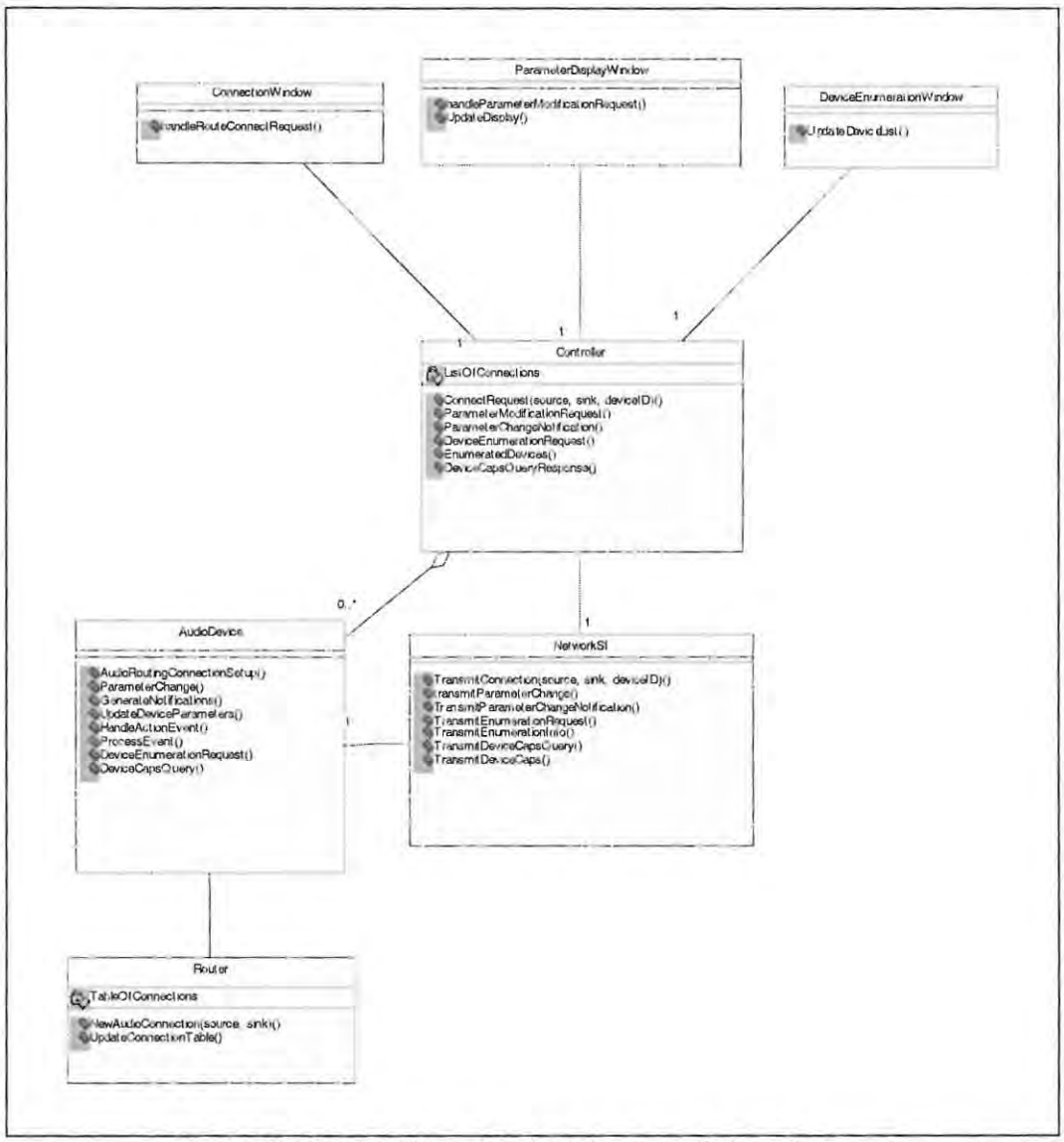

Figure 2.29: An Object Model of a typical Sound System

2.6 Summary

The examination of currently available sound system technologies enabled a list of common characteristics to be extracted. This list formed the scenarios that were needed for the development of an applicable UML model for a typical sound system, and provides insight into the control facilities required within an IEEE 1394 environment

The next chapter provides a brief overview of the IEEE 1394 architecture, and discusses the various transmission mechanisms defined to allow the transport of audio control and monitoring data. The transmission of audio and music data is also considered, and the relevant protocols described.

Chapter Three
An Overview of the IEEE 1394 Architecture

Apple Computer was the originator of the IEEE 1394 serial bus architecture, which they dubbed "FireWire". Although uncertainty surrounds the original purpose of the IEEE 1394 bus, it has nevertheless found a wide range of applications, providing a cheap, high-performance digital interconnect. It provides true plug-and-play capabilities, variable speeds, multiple media types, asynchronous and isochronous data transfer, and power distribution.

This chapter provides a general overview of the IEEE 1394 serial bus specifications, including the current enhancements to the bus, and describes the protocols defined for the transmission of audio and consumer control data. Included, is a discussion of the Audio/Video Control (AV/C) protocol, and the Audio and Music data transmission (A/M) protocol.

3.1 ISO/IEC 13213

The ISO/IEC 13213 specification, formally named "Information technology – Microprocessor systems – Control and Status Registers (CSR) Architecture for microcomputer busses" [1], forms the foundation of the 1394 bus architecture. The primary goals associated with the CSR specification are to:

- Reduce the amount of customised software needed to support a given bus standard.
- Simplify and improve interoperability of bus nodes based on different platforms.
- Support bridging between different bus types.
- Improve software transparency between different bus types [Anderson, 1999].

[1] The ISO/IEC 13213 (ANSI/IEEE 1212) specification will be referred to as the CSR specification in this dissertation.

The IEEE 1394 (FireWire), IEEE 896 (Futurebus+), and the IEEE 1596 (Scalable Coherent Interface) working groups, have aided in the definition and adoption of the CSR specification, which defines the following features:

- Node architecture,
- Address Space,
- Common transaction types,
- Control and Status Registers (CSR),
- Configuration ROM format and content,
- Message broadcast mechanisms to all nodes and units, and
- Interrupt broadcast to all nodes.

The CSR specification also permits bus-dependant extensions and features to be defined, as is the case with the IEEE 1394 serial bus specification. The following section discusses the features of the CSR specification that are relevant to IEEE 1394, as defined by the IEEE Standard for a High Performance Serial Bus (IEEE Std 1394-1995) [IEEE Computer Society, 1996].

3.1.1 Node Architecture

IEEE 1394 devices are comprised of modules, nodes and units. A module represents a physical device attached to the bus that contains one or more nodes. A node is a logical entity that is visible to the initialisation software and contains CSRs and ROM entries, which are mapped into the initial node's address space. Units are the subcomponents of nodes, and provide processing, memory, or I/O functionality. Units within a node are typically independent of one another, and their defined registers are mapped onto the address space of the node in which they are contained.

3.1.2 Address Space

The 1394 bus follows the 64-bit addressing scheme, defined within the CSR specification. This scheme defines a 16-bit node address (*node_ID*), which allocates address space for up to 65536 nodes. The IEEE 1394 specification further divides this 16-bit address space into 10-bits for a Bus Identifier (*bus_ID*), and the remaining 6-bits identify a particular node on the selected bus (*physical_ID*). This extension

provides address space for up to 1023 buses, and 63 independent nodes within each bus. The remaining 48 least significant address bits define 256 terabytes (TB) of address space for each node. Within this address space, space is allocated for:

- Private space (256MB)
- Initial memory space (256TB – 512MB)
- Register space (256MB)
    - CSR architecture (512B)
    - Serial bus (512B)
    - ROM space (1024B)
    - Initial units space (256MB – 2kB)

Figure 3.1 displays the node addressing scheme utilised by the IEEE 1394 architecture.

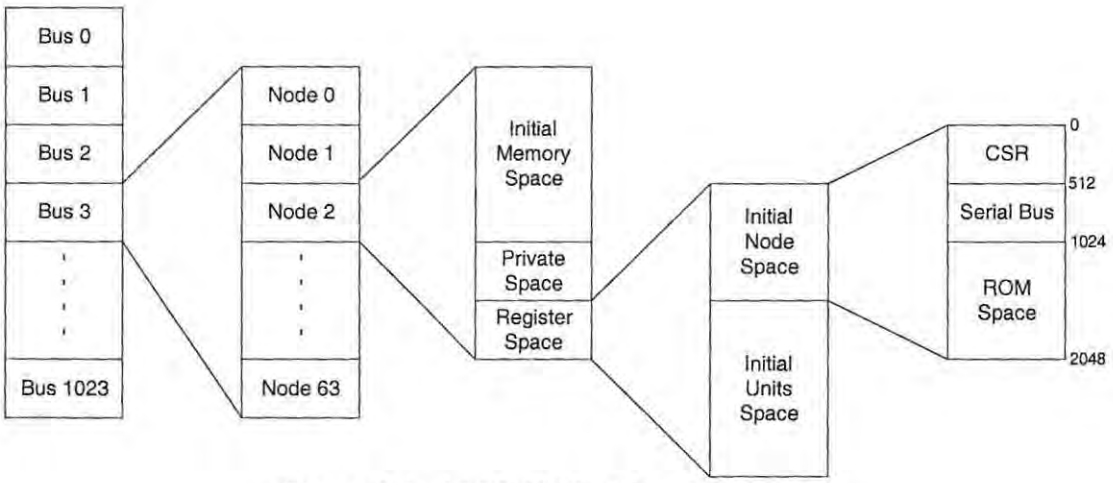

Figure 3.1: IEEE 1394 node address space

3.1.3 Common Transaction Types

The two basic data transfer services available to nodes utilising the 1394 bus are asynchronous and isochronous transfers. Asynchronous transfers provide a protocol that allows confirmation of delivery of explicitly addressed, variable-length packets that are sent periodically.

All asynchronous transfers are initiated by a requester node, and are targeted at a responder node. These transfers can be broken down into two subactions:

- Request subaction – The transfer of the address, command, and data from the requester node to the responder.
- Response subaction – The transfer of completion status, or data from the responder to the requesting node.

These subactions form the basis for the three basic transaction types that are defined for asynchronous transfers:

- Reads – Data is returned within the response subaction from the address specified in the request.
- Writes – Data, provided in the request subaction, is written into the specified address. The completion status would be returned to the requester in the response subaction.
- Locks – This mechanism allows the requester to perform an atomic read-modify-write operation into the specified address. The modified data would be returned to the requester in the response subaction.

Isochronous transfers provide a means to transfer variable-length packets at regular intervals. This mechanism allows one requester (*isochronous talker*) to send to one or more responders (*isochronous listeners*), and as such could be termed a broadcast mechanism. An isochronous transfer occurs when a requester node broadcasts data on a particular isochronous channel. Responder nodes can listen on this channel. The IEEE 1394 architecture supports up to 64 isochronous channels on a single bus segment.

3.1.4 Control and Status Registers (CSRs)

The CSR specification defines a number of core control and status registers, as detailed in table 3.1, that 1394 nodes must implement. All of these core registers are accessible at standardised location offsets within the initial register address space. The starting address for the initial address space is found at offset FFFF F000 0000h.[2] In addition to these registers a number of 1394 bus-specific registers have been defined, some of which are provided in table 3.1.

[2] The lowercase h, shows that the preceding string is to be interpreted as a hexadecimal value (Base 16).

Offset	Name	Description
000h	STATE_CLEAR	State and Control Information
004h	STATE_SET	Used in setting the STATE_CLEAR register
008h	NODE_IDS	16-bit Node ID
00Ch	RESET_START	Resets the state of the node
018h - 01Ch	SPLIT_TIMEOUT_HI SPLIT_TIMEOUT_LO	The timeout value for detecting split-transaction errors (Implemented by transaction capable nodes)
200h	CYCLE_TIME	Implemented by isochronous capable nodes for synchronisation
204h	BUS_TIME	Implemented by cycle-master capable nodes. Provides a measure of the current time in seconds
208h	POWER_FAIL_IMMINENT	Provides notification that power is about to fail
20Ch	POWER_SOURCE	Used with the POWER_FAIL_IMMINENT register to validate power failure notifications
210h	BUSY_TIMEOUT	Time limit for retry attempts
21Ch	BUS_MANAGER_ID	Provides the physical ID of the bus manager
220h	BANDWIDTH_AVAILABLE	Used in the management of isochronous bandwidth allocation
224h - 228h	CHANNELS_AVAILABLE	Used in the management of of isochronous channel allocation
22Ch	MAINT_CONTROL	Used to diagnose and verify error detection logic
230h	MAINT_UTILITY	Read/Write register used in debugging

Core CSR Locations
Serial-Bus-Dependent CSR Locations

Table 3.1: CSR register locations and definitions

Full descriptions of these CSR registers can be found in section 8.3.2, of the IEEE Standard for a High Performance Serial Bus [IEEE Computer Society, 1996].

The address space above address FFFF F000 0800h, is reserved for node-dependant resources which are within the initial units space. However the address-offset 0800h to FFFCh is reserved for bus-dependant use, as can be seen in table 3.2. Two address ranges within this space are reserved for serial bus management functions. Some of the registers within the reserved address ranges are populated with values as defined by the IEC 61883-1 specification [IEC 61883-1, 1998].

Offset	Name	Description
800h - FFCh		Reserved for Serial Bus
1000h - 13FCh	TOPOLOGY_MAP	Used to deduce the topology of the bus. Present at the bus manager only.
1400h - 1FFCh		Reserved for Serial Bus
2000h - 2FFCh	SPEED_MAP	Used in determining the maximum speed at which nodes can send/receive data. Present at the bus manager only.
3000h - FFFCh		Reserved for Serial Bus

Table 3.2 Serial-Bus-Dependent registers in the initial units space

3.1.5 Configuration ROM

The CSR specification states that all transaction-capable nodes on the serial bus shall implement a configuration ROM. The purpose of this is to:

- Provide information to facilitate the association of I/O software drivers and diagnostic software with a particular module, node or unit.
- Define additional parameters needed by the software to access a device correctly.
- Define the capabilities of a node.

Two ROM formats are supported:

- Minimal format
- General format

The minimal ROM format, illustrated in figure 3.2, provides a 24-bit company identifier, while the general ROM format provides information in a bus information block (*Bus_Info_Block*), and a root directory (*Root_Directory*), as seen in figure 3.3.

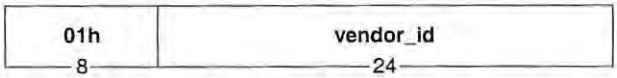

Figure 3.2: Minimal ROM format

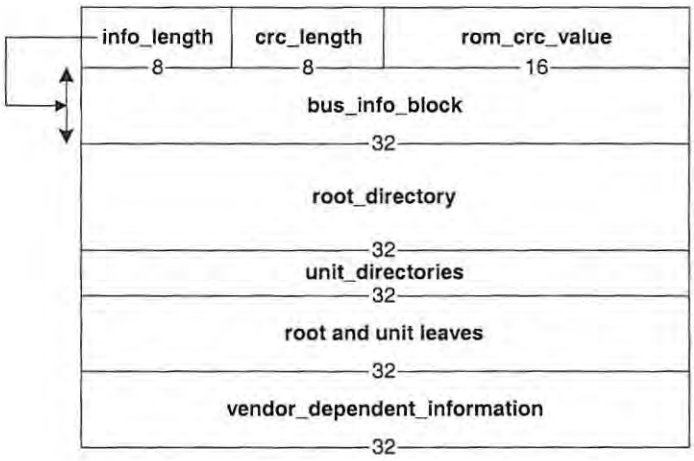

Figure 3.3: General ROM format

The *Bus_Info_Block* provides information pertaining to the bus related capabilities of this node. The CSR specification defines only a *bus_name* field within the first quadlet of this block, with the serial bus standard defining the remaining fields within the block. The *Root_Directory* specifies the content and organisation of the rest of the ROM, and may include pointers to other directories or data structures, called leaves. Figure 3.4 illustrates a simple tree that is possible using the general ROM format. Specific details on the information and fields contained within the general ROM format can be found in the IEEE Standard for a High Performance Serial Bus [IEEE Computer Society, 1996].

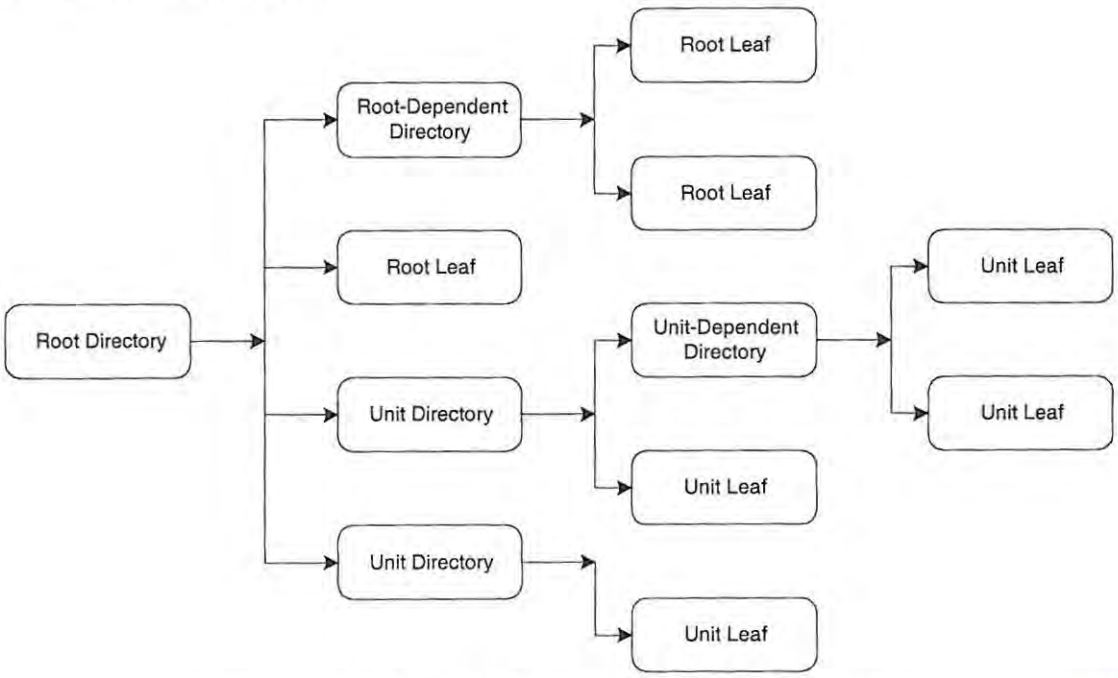

Figure 3.4: Example Configuration ROM Hierarchy

3.1.6 Message Broadcasting

The CSR specification provides a mechanism whereby a message that is intended for multiple nodes can be broadcast over the bus. This is accomplished by addressing node 63 on the bus, which has been reserved as a broadcast address. This mechanism allows asynchronous transactions to be sent to multiple nodes. The nodes listening for broadcast transactions must not respond, as there are possibly multiple receivers and contention on the bus must be avoided. A broadcast mechanism for isochronous transactions is not needed, as all such transmissions are inherently broadcast.

3.1.7 Interrupt Broadcasting

INTERRUPT_TARGET and INTERRUPT_MASK are two registers defined by the CSR specification to allow the broadcasting of interrupts to all units within a node, this being referred to as a nodecast. The INTERRUPT_TARGET register allows up to 32 different interrupt events to be defined, while the INTERRUPT_MASK register is used as a mask to the INTERRUPT_TARGET register.

3.2 Protocol Architecture

Serial bus protocols are typically described as a set of three stacked layers; known as the transaction, link, and physical layer. The IEEE 1394 specification has added a serial bus management layer to provide basic control functions and CSR's needed to control nodes and manage bus resources. Figure 3.5 provides a graphical representation of these layers, and the interactions that occur.

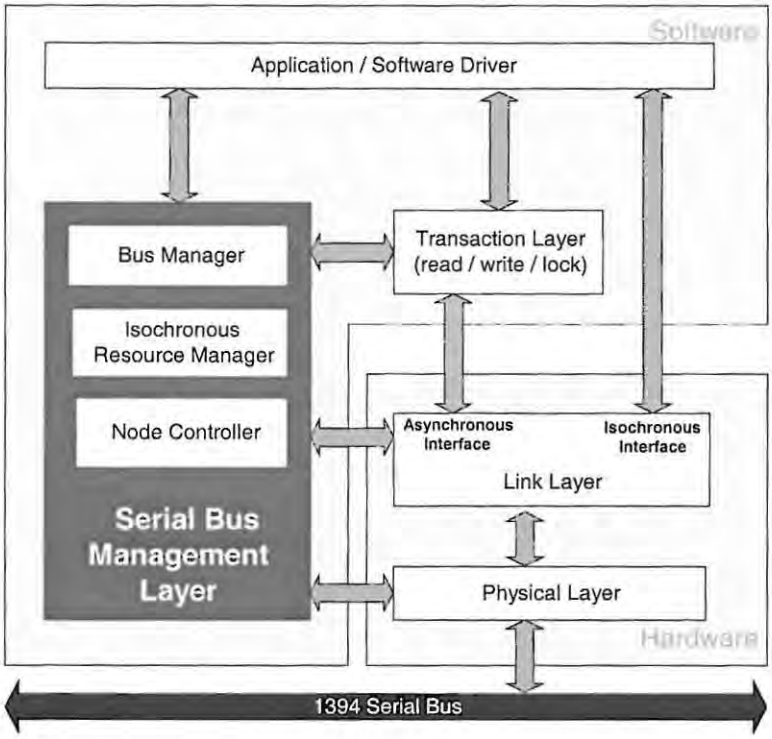

Figure 3.5: IEEE 1394 Serial Bus Protocol Layers

3.2.1 Serial Bus Management Layer

Three management entities are defined to support a completely managed bus:

- Cycle Master
- Isochronous Resource Manager
- Bus Manager

The role of the cycle master is to provide control over the interval at which isochronous transactions are performed. The cycle master is also the root node, and is responsible for the generation and broadcasting of cycle start packets at 125μs intervals, thereby indicating the next series of isochronous transactions [Anderson, 1999].

The facilities provided by the isochronous resource manager are:

- Isochronous bandwidth allocation
- Channel number allocation
- Selection of cycle master

Registers within the isochronous resource manager are situated at standardised addresses known to all of the isochronous capable nodes on the bus. These nodes must first acquire a channel number and bus bandwidth from the isochronous resource manager before beginning isochronous communication. This is achieved by performing write operations into the CHANNELS_AVAILABLE and BANDWIDTH_AVAILABLE registers, respectively.

The responsibilities of the bus manager are to provide the following services:

- Bus power management
- Speed map maintenance
- Topological map maintenance
- Bus optimisation based on information contained within the topological map

The bus manager node can reside anywhere on the bus. During cable configuration, self identification (*self-ID*) packets are captured by the bus manager and the data contained within them is used to construct the topological and speed maps. Subsequent to a bus reset, self-ID packets are transmitted by every node on the bus and consist of data characterising certain serial bus parameters and requirements associated with each node. The process of determining the bus manager and details of its responsibilities can be found in Anderson [Anderson, 1999].

3.2.2 Transaction Layer

The transaction layer provides an interface between the application and the link layer for asynchronous transactions, and plays no part in isochronous communication. IEEE 1394 applications typically issue data transfer requests to the transaction layer. This software layer translates the transfer request into the necessary transaction requests needed to complete the transfer.

Four service primitives are defined for asynchronous read, write and lock transactions:

- Request Service – Used by the requester to start the transaction (initiates the request subaction).

- Indication Service – Notification to the responder of an incoming request (completes the request subaction).
- Response Service – The responder's reply containing status information or data (starts the response subaction).
- Confirmation Service – The requester is notified that the response has been received (completes the response subaction).

The IEEE 1394 specification has added verification of packet delivery to the transaction model. This verification process is the responsibility of the transaction layer, and is achieved by transferring a one-byte acknowledgement packet to the sender upon receipt of a non-broadcast asynchronous packet. In the event that the acknowledgement is not received, retries may be attempted. Figure 3.6 illustrates the interactions between the transaction layers on the requester and responder nodes.

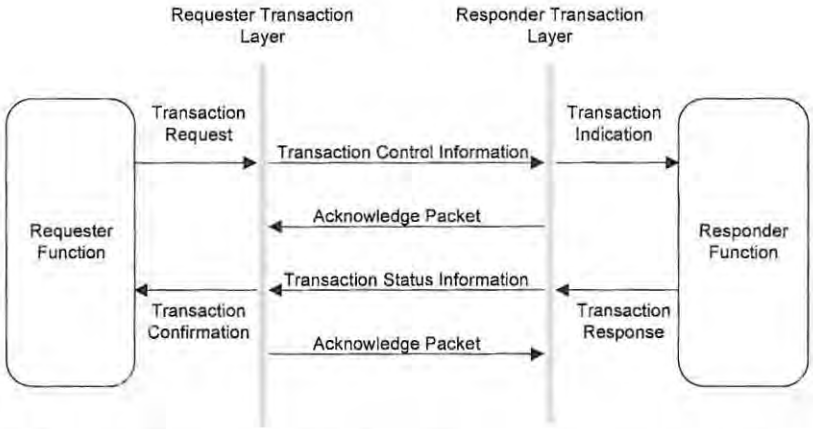

Figure 3.6: Transaction layer service communications

3.2.3 Link Layer

The link layer provides a half-duplex packet delivery service. The delivery of a single data packet is termed a *subaction*, and two types are defined:

- Asynchronous Subaction – A variable amount of data and several bytes of transaction layer information are transferred to a specified address and an acknowledgement is returned.
- Isochronous Subaction (channel) – A variable amount of data is transferred at regular intervals on a specified channel, but no acknowledgement is provided.

The link layer provides the interface between the transaction layer and the physical layer for asynchronous transactions, and between the physical layer and the application / Software Driver for isochronous transactions. The services used by the link layer are similar to those described in the transaction layer, and follow a similar request – response model. For example, asynchronous transaction requests received by the requester's link layer from the transaction layer will be translated into IEEE 1394 packets to be transmitted over the bus. The packet received by the responder is translated by its link layer and passed to the transaction layer for processing. Isochronous transactions follow the same model with the exception that the link layer communicates with the Software Driver and not the transaction layer.

3.2.4 Physical Layer

Different physical layers exist for the backplane and the cable environments, of which only the cable environment is relevant to this study. In the cable environment the physical (*PHY*) layer provides the actual interface to the 1394 bus, and has the following primary functions:

- Cable configuration
- Transmission and Reception of data bits
- Arbitration
- Provision for the electrical and mechanical interface

3.2.4.1 Cable Configuration

There are three phases to cable configuration:

- Bus initialisation
- Tree identification
- Self-identification

Bus initialisation occurs when a node joins or leaves the bus, as a bus reset signal is generated forcing all nodes to clear their topology information, and enter into an idle state. At this point nodes are only aware of their state as either a branch (more than one directly connected neighbour), a leaf (only one connected neighbour), or isolated

(no connected neighbours). After a sufficient period of time to allow all nodes to be reset, the next phase of tree identification begins.

The tree identification process translates the general network topology into a tree. This involves establishing a root node, and providing all physical connections an associated direction pointing towards the root node. This process entails that each port associated with a node be identified as either a "parent" or "child" port. A node with a port designated as parent indicates that the node connected to that port is closer to the root node, while a node connected to a child port is further away from the root node. The node that has all its connected ports labelled as child ports becomes the root node, regardless of where in the network it is connected. A node wishing to become the root node can bias itself, by delaying its participation in the tree identification process. Once the forced delay expires and the node begins participation in the tree identification process, the signals on all of its ports indicate that it is the parent node. This is due to all other nodes having completed signalling that their ports are parent ports, therefore all of the nodes connected ports are designated as child ports and it assumes the role of root. A requirement is that the root node must also be the cycle master, as it has the highest priority.

At this point the self-identification phase can begin. This phase allows a node to select a *physical_ID* and to identify itself to any management entities on the bus using self-ID packets. Contained within the self-ID packets are:

- Physical identifier of the node sending the packet
- An indicator if its link and transaction layers are active
- Current value of its gap count
- The node's speed capabilities
- Whether it is a contender for isochronous resource manager or bus manager
- Power requirements
- Port status information

Nodes that are isochronous resource manager (IRM) capable must monitor all self-ID packets, to determine if other nodes on the bus are also contending for the role of IRM. The node with the highest *physical_ID* assumes the role of IRM. Once the

IRM has been established, any bus manager capable node can attempt a locked compare and write transaction into the IRM's BUS_MANAGER_ID register. The node that succeeds assumes the role of bus manager.

The bus manager inspects the root node to ensure that it is cycle master capable. If no bus manager is present, this responsibility falls to the IRM. If the root node fails this inspection, the other available nodes bus information blocks are scrutinised for a set *cmc* (cycle master capable) bit. Should such a node be found, a bus reset occurs and the identified node will become root due to a forced root delay, as described in the discussion of the tree identification process.

3.2.4.2 Transmission and Reception of data bits

Data delivery on the 1394 bus is bi-directional, and occurs via twisted pair, using a combination of non-return to zero (NRZ) and data-strobe encoding. Since the speed of the bus is variable, speed information is also sent during packet transmission. A variety of speed codes have been defined for bit rates applicable to the IEEE 1394 cable environment. These codes and associated bit rates are presented in table 3.3.

Speed code	Bit rate (Megabits per second)
S100	98.304
S200	196.608
S400	393.216

Table 3.3: Cable environment speed codes and associated bit rates

For example, a speed code of S200 indicates a transmission speed of 196.608 Mbps or approximately 200 Mbps.

3.2.4.3 Arbitration

Nodes wishing to transmit on the bus, must first arbitrate for ownership before transmission. Arbitration is based on guaranteed bus bandwidth for isochronous channels, and a fairness interval for asynchronous transmissions. Isochronous transactions begin immediately after a cycle start packet which is broadcast every 125 μs. The fairness interval associated with asynchronous transmission permits only one asynchronous transaction per interval, thereby ensuring a fair rotation arbitration scheme. The end of this fairness interval is indicated by 20 μs of idle time on the bus.

The arbitration process begins when an idle state is recognised on the bus, indicating the end of a previous transmission. The period of this idle time, termed *gap timing*, varies for isochronous and asynchronous transactions. The *isochronous gap* detection is in the range 0.04μs and 0.05μs [Anderson 1999], while the *asynchronous gap* can be tuned to begin as early as possible. Note that acknowledgments do not require arbitration and are treated as an atomic operation across the bus. The arbitration process can begin once the acknowledgment has been returned.

3.2.5 Asynchronous Packet Format

Figure 3.7 is an illustration of the asynchronous packet format associated with a request packet. The presence of the data block is optional, depending on the amount of data to be transferred. Asynchronous packets are utilised by the AV/C protocol, described in section 3.4.1, to exert control over IEEE 1394 devices, including audio and video devices.

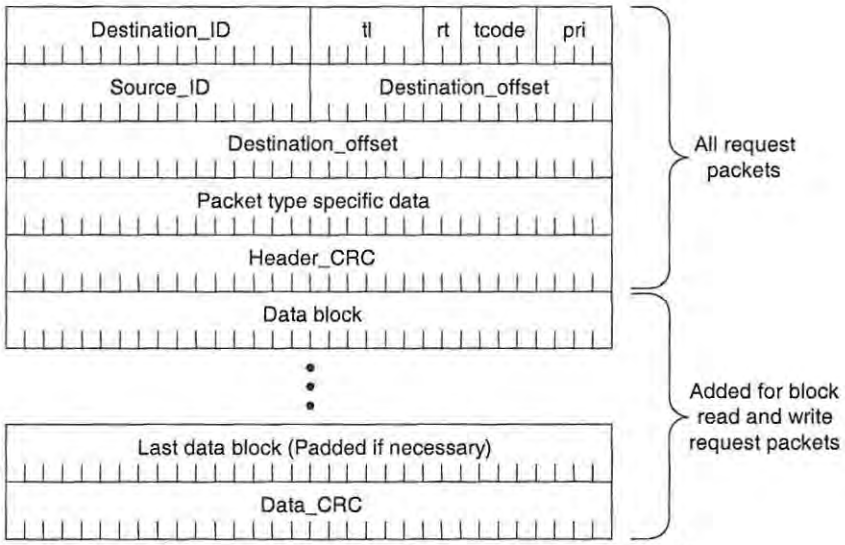

Figure 3.7: Asynchronous packet format

- The *DestinationID* field is a 16-bit field, 10-bits for the bus identifier and 6 for the physical_ID.
- The *Destination_offset* field provides an address location within the targeted node. The combination of the *Destination_ID* and *Destination_offset* fields, equate to the 64-bit addressing scheme defined within the CSR specification.

- The transaction label (*tl*) field identifies the transaction, and is specified by the requester. The *tl* field is used to match responses to requests.
- The retry code (*rt*) specifies whether this packet is a retry attempt and the retry protocol that the target node should follow.
- The transaction code (*tcode*) specifies the type of transaction and the packet format. A list of common transaction codes and their associated transactions are provided in table 3.4.

Transaction Name	tcode
Write request for data quadlet	0h
Write request for data block	1h
Write response	2h
Read request for data quadlet	4h
Read request for data block	5h
Read response for data quadlet	6h
Read response for data block	7h
Cycle start	8h
Lock request	9h
Asynchronous Streaming Packet	Ah
Lock response	Bh

Table 3.4: Transaction codes

- The priority (*pri*) field has no meaning in the cable environment.
- The *Source_ID* field provides the *node_ID* of the sending node.
- The value of the *packet type specific data* field depends on the type of packet being transmitted.

Of particular importance to this study is the asynchronous streaming packet transaction code, defined within the 1394a specification. This transmission type resembles an isochronous packet, as can be seen in figures 3.8 and 3.9, which is transmitted subject to asynchronous arbitration. This allows applications to perform data streaming when guaranteed latency is of little concern. Asynchronous streaming packets do not require the allocation of bandwidth, although they do require the allocation of a channel number. The channel allocation happens in the same way as it does for isochronous transmissions. The packet size constraints applicable to other asynchronous transactions are also applicable to asynchronous stream packets. These size limits are provided in table 3.5. As asynchronous stream packets are broadcast packets, an acknowledgement or response is not required. In addition, a global

asynchronous stream packet (GASP) has been defined and is illustrated in figure 3.9. It is this packet format that is utilised by the IP-stack implementation described in chapter 5. The *Source_ID* field specifies the address of the sender of the stream packet, and the *Specifier_ID_hi* and *Specifier_ID_lo* fields contain a 24-bit organizationally unique identifier (OUI) which identifies the meaning and usage of the data payload. The meaning and usage of the *Version* field is defined by the owner of the OUI.

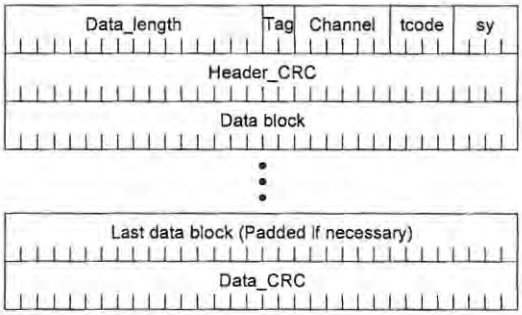

Figure 3.8: Asynchronous stream packet format

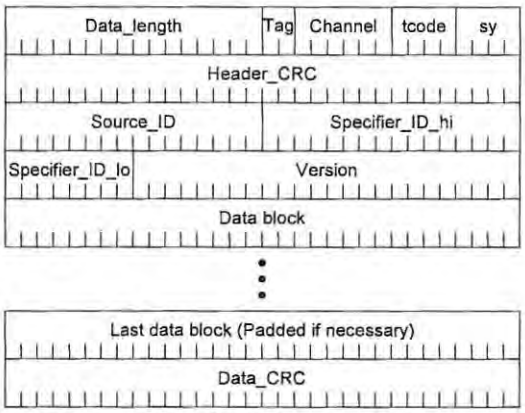

Figure 3.9: Global asynchronous stream packet (GASP) format

Cable Speed	Maximum Data Payload (Bytes)
100 Mb/s	512
200 Mb/s	1024
400 Mb/s	2048
800 Mb/s	4096
1.6 Gb/s	8192
3.2 Gb/s	16384

Table 3.5: Maximum data payloads for asynchronous transactions

3.2.6 Isochronous Packet Format

Isochronous transactions are broadcast or multicast in nature, and as such do not specify a destination and do not receive a response or acknowledgement. Instead they are transmitted on a channel number, obtained from the isochronous resource manager. Isochronous data is transmitted in a *streaming data packet*, the format of which is provided in figure 3.10. Before an isochronous stream packet can be sent, the requesting application must obtain the necessary bandwidth from the isochronous

resource manager. Isochronous packets provide the basic transport mechanism for audio and music data, which is detailed in section 3.4.

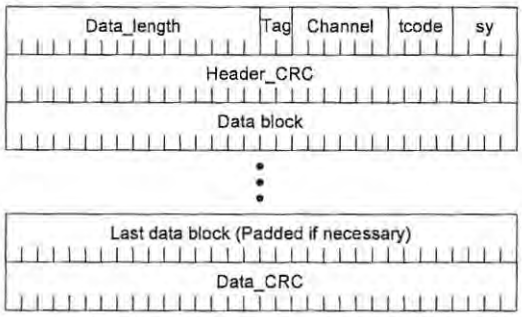

Figure 3.10: Isochronous stream packet format

- The *Data_length* field specifies the length of the data being transmitted. The data length must be a multiple of four bytes, which may be padded if necessary.
- The *Tag* field indicates the isochronous data format of the packet.
- The *Channel* field provides the isochronous channel number for the packet.
- The *tcode* for an isochronous transaction is defined to be Ah.
- The synchronisation (*sy*) code is application specific.

3.3 Enhancements to the IEEE 1394 – 1995 Specification

Early implementations of IEEE 1394 products led to varied interpretations of the IEEE 1394 - 1995 specification and consequently interoperability problems. The 1394a specification was developed to clarify the original 1394 specification. 1394a also served to introduce new features and enhancements, to improve performance and efficiency. The requirements for higher speeds and improved reach of a 1394 interconnect, led to the development of 1394b. Further architecture improvements are provided in the 1394.1 specification, which specifies the way in which multiple IEEE 1394 busses are to be connected. A discussion of this specification is not included, as it falls beyond the scope of this project.

3.3.1 1394a

The 1394a specification addresses the following issues:

- Cables and connectors for a 4-pin variant,
- Standardisation of the PHY/LINK interface,

- Performance enhancements to the PHY layer, which are backwards compatible with the original 1394 specification,
- A redefinition of the isochronous data packet, to allow its use within the asynchronous and isochronous cycles,
- Clarification on the power and electrical isolation requirements, and
- Miscellaneous clarifications to the original specification.

This specification led to accelerated arbitration which greatly improved the efficiency of the bus, particularly those with less than 63 nodes. The disruptions caused by bus resets are greatly reduced, and the power requirements refined.

3.3.2 1394b

While 1394a introduced general improvements to the bus, the extensions described within 1394b are particularly important to sound installations. 1394b introduced gigabit speed extensions, as well as providing single hop distances of up to 100 metres. 1394b is still however backward compatible with the IEEE 1394-1995 and 1394a specifications.

These improvements were gained due to the following:

- New connection model,
- Additional transmission types,
- Arbitration improvements, and
- Miscellaneous improvements.

The connection model proposed utilises a new physical layer that communicates using a continuously transmitted full duplex signal using 8B10B encoding [IEEE Computer Society, 2000]. A new arbitration scheme called "*Bus Owner/Supervisor/Selector*" (BOSS) is introduced, which dramatically reduces arbitration overhead, and allows arbitration to be pipelined [IEEE Computer Society, 2002].

The additional media types defined include copper, optical fibre and category 5 unshielded twisted pair (UTP5). IEEE 1394-1995 suggests a maximum cable length

of 4.5 metres, which is restricting to many applications. The new media types defined provide the following cable length improvements:

- Plastic optical fibre can support cable lengths of up to 50 metres,
- Glass optical fibre cable runs can be up to 100 metres long, and
- CAT-5 lengths of up to 100 metres are supported operating at S100 (98.304 Mbps).

3.4 Audio and Music Distribution over IEEE 1394 networks

The "Audio and music data transmission protocol" (A/M protocol) [IEC 61883-6, 1998], was defined to allow the transmission of audio and music data, including the transport of IEC 60958 digital format, raw audio samples, and MIDI data, over IEEE 1394 networks. The transport mechanism defined builds on the isochronous transmission method associated with IEEE 1394, by defining the necessary field values and packetising audio and MIDI data clusters within a common isochronous packet (CIP).

3.4.1 Isochronous and Common Isochronous Packet Headers

Figure 3.10 presented the format of an isochronous packet, whose fields must contain the values provided in table 3.6 to comply with the A/M protocol.

Field	Value	Description
tag	01_b	Indicates that a CIP header in included with this packet
tcode	Ah	Indicates that this is an isochronous data packet
sy	0h	This field is reserved for copy protection, and shall be set to 0_{16}

Table 3.6: Isochronous packet header field values

The common isochronous packet (CIP) format [IEC 61883-1, 1998], was developed to provide for the real time transmission of data, and is contained within the data field of an isochronous packet as is illustrated in figure 3.11. Two CIP header formats exist, one that contains an SYT field, and one that does not. The SYT field is used in the real time transmission of data, as it contains a time stamp consisting of the lower 16-bits of the CYCLE_TIME register. Because it is essential that the transmission of music and audio data occur in real time, the format containing the SYT field, as illustrated in figure 3.11, is employed.

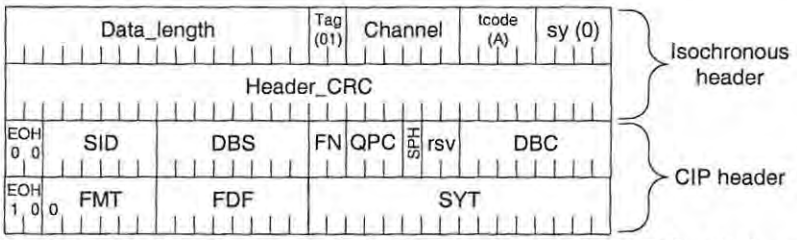

Figure 3.11: Isochronous and CIP header (with SYT field) format

The descriptions of the fields contained within the CIP header are as follows:

- The *SID* field contains the node identification number of the transmitting node.
- The data block size (*DBS*) field specifies the size of the data block in quadlets.
- The fraction number (*FN*) contains the number of data blocks into which a source packet is divided. Acceptable values are:
    - not divided (00_b),
    - 2 data blocks (01_b),
    - 4 data blocks (10_b), or
    - 8 data blocks (11_b).

  The *FN* field shall contain the value 0 for the A/M protocol.
- *QPC* specifies the quadlet padding count, which is the number of dummy quadlets padded to every source packet to enable division into equally sized blocks. The *QPC* field will contain the value 0, as defined within the A/M protocol
- *rsv* indicates that the field is reserved.
- The *DBC* (Data Block Count) field contains the number continuity counter for the detection of a loss of data blocks.
- *FMT* contains the format identification number, which specifies the format of the encapsulated data. The code 10h has been assigned to indicate that the data is in the audio and music format.
- *FDF* is the format dependent field, and is defined for each *FMT*. For the transmission of audio and music data, the *FDF* field is subdivided into two fields. These two fields are the *EVT* (2-bit) and *NSF* (3-bit) fields that specify the event type (*EVT*) and nominal sampling frequency (*NSF*) respectively. Tables 3.7 and 3.8 provide the values that have been defined for these fields.

Value	Description
0	AM824 data
1	24-bit * 4 audio pack
2	32-bit floating point data
3	Reserved for 32-bit or 64-bit data

Table 3.7: Event type value definitions

Value	Description
0	32 kHz
1	44.1 kHz
2	48 kHz
3	Reserved
4	96 kHz
5-7	Reserved

Table 3.8: Nominal sampling frequency value definitions

- *SYT* represents the time stamp field, and contains the lower 16-bits of the CYCLE_TIME register.

3.4.2 Events

If event sequences such as sequences of audio samples are synchronized, it is possible to cluster the sequences. Those events that occurred at the same time can be grouped into a single data block for transmission. More than one data block may be incorporated into a single packet, as illustrated in figure 3.12. The number of sequences clustered is referred to as the CLUSTER_DIMENSION. For example, if audio is sample at 48 KHz there will be 6 clusters or data blocks within each isochronous packet, as isochronous packets are transmitted at 8 KHz [Fujimori and Foss, 2002].

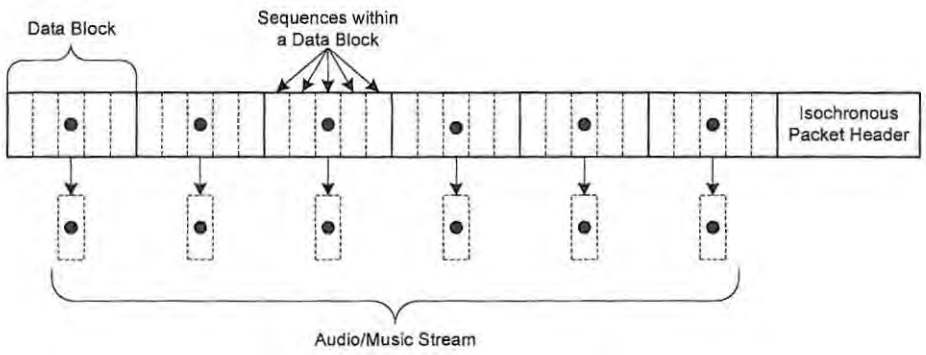

Figure 3.12: Audio/Music stream encapsulation

All events are required to be packed into 32-bit quadlets. However, non-32-bit events also occur for which an efficient packing mechanism is needed. This packing mechanism is referred to as a *pack* event. Pack, and other data, can be combined into units that form a cluster. For example, audio transmitted in 24-bit sequences, needs to be broken into 32-bit events for transmission. A simple solution is to allow four 24-bit events to be packed into 3 quadlets, as illustrated in figure 3.13. However, as audio is commonly transmitted in the 32-bit AM824 format, described in section 3.4.3, theoretically the packet event will consist of one AM824 data block.

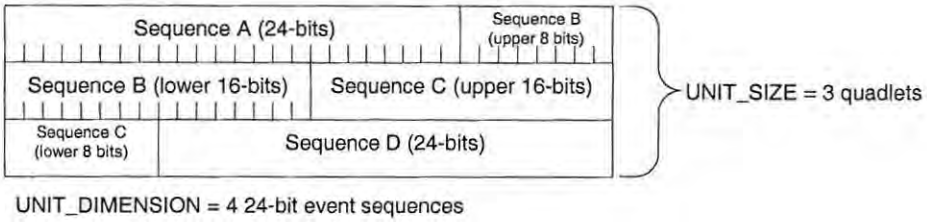

Figure 3.13: Pack event structure containing 24-bit event sequences

3.4.3 AM824 Data

AM824 data consists of audio or music data framed with an 8-bit label and 24-bit data, with the *label* field identifying the format of the data contained within the *24-bit data* field. Table 3.9 provides a listing of the defined data formats and their associated values. The *label* field is required to be checked, by receivers, for each AM824 data received within a sequence.

Value range	Description
00h – 3Fh	IEC 60985 conformant data
40h – 43h	Raw audio
44h – 7Fh	Reserved
80h – 83h	MIDI conformant data
84h – FFh	Reserved

Table 3.9: Defined label codes for AM824 data

Each of these data formats has a defined bit pattern describing the data that follows. For example, raw audio data conforms to the structure presented in figure 3.14.

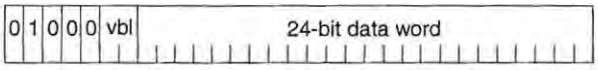

Figure 3.14: Raw audio AM824 format

The *VBL* field denotes the variable bit length of the data contained within the data field, for which valid entries are provided in table 3.10. If the length of the audio data is less than the allocated 24-bits, the remaining bits within the data field shall be padded with zero-bit values, with the padding occupying the least significant bit positions.

Value	Description
00_b	24-bits
01_b	20-bits
10_b	16-bits
11_b	Reserved

Table 3.10: Variable bit length values

3.5 Unit Architecture Specifications

A wide variety of functional devices exist that can host 1394 nodes. Examples of these include hard drives, video cameras, and sound equipment. These 1394 nodes are implemented based on *unit architectures*, illustrated in figure 3.15, which define the details of the relevant device, such as:

- Protocols
- ROM entries
- Relevant CSRs

An example of such a unit architecture is the audio-video control (AV/C) unit architecture, which describes the use of audio/visual functions on the 1394 bus.

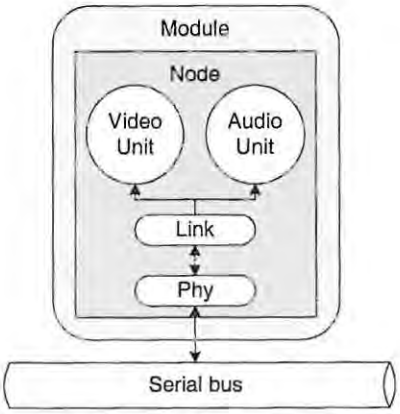

Figure 3.15: Illustration of module, node, and unit architecture

3.5.1 Audio/Video Control

The Audio/Video Control (AV/C) specification defines a command set that allows for the control of consumer and professional IEEE 1394 devices. This specification consists of a variety of defined commands and responses, which are transported by the Function Control Protocol (FCP).

3.5.1.1 Function Control Protocol

The Function Control Protocol (FCP) provides an appropriate transport mechanism for AV/C, with the provision of both request and response facilities over the IEEE 1394 bus. All of these request and response packets are, however, restricted to 512 bytes in length.

FCP messages are encapsulated and transmitted over the bus in the form of asynchronous block write transactions, as illustrated in figure 3.16, or quadlet write transactions. These asynchronous write transactions are addressed to registers that are located at predetermined offsets within the target node's address space. Typically these registers are referred to as the FCP_COMMAND and FCP_RESPONSE registers, and are located at offsets FFFF F000 0B00h and FFFF F000 0D00h, respectively.

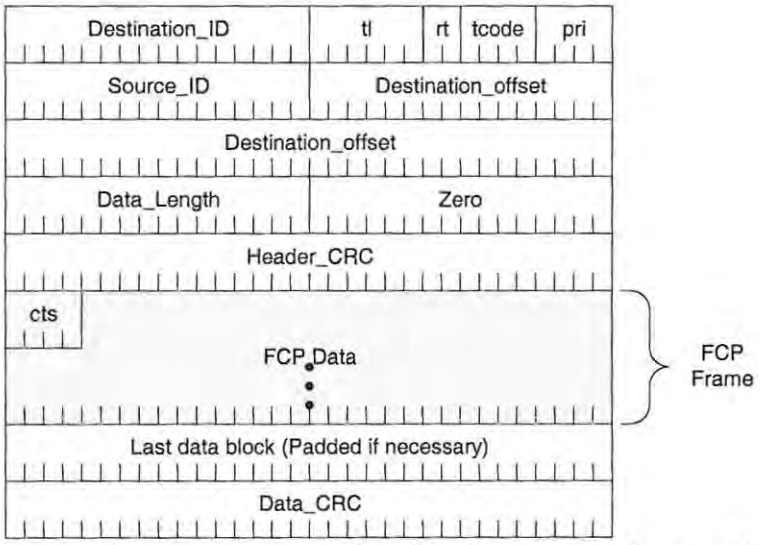

Figure 3.16: An FCP frame encapsulated within an asynchronous block write transaction

3.5.1.2 AV/C Commands and Responses

AV/C commands and responses are encapsulated within FCP frames for transmission over the IEEE 1394 bus, and target the FCP_COMMAND and FCP_RESPONSE registers, which are described in section 3.5.1.3. All AV/C messages are encapsulated within an appropriate frame for transmission, of which the command and response frames differ slightly, as shown in figures 3.17 and 3.18.

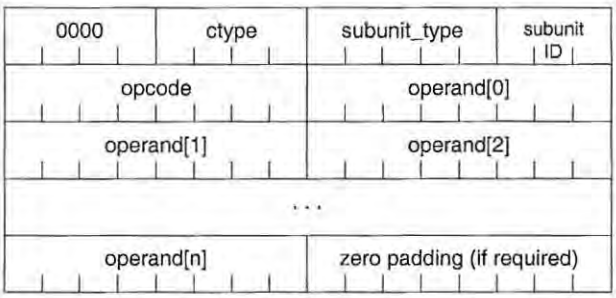

Figure 3.17: AV/C command frame

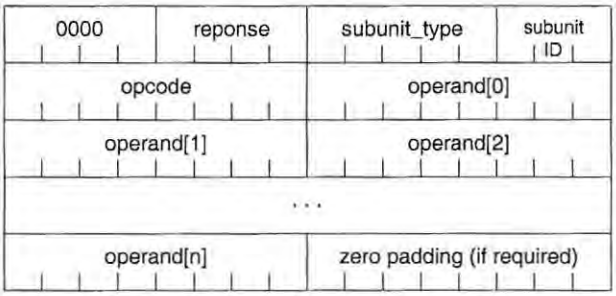

Figure 3.18: AV/C response frame

The *ctype* field in the AV/C command frame specifies a command type, of which there are five defined commands, and the *response* field in the response frame specifies the defined response types. These defined commands, responses and associated values are presented in table 3.11.

Value	Command *(ctype) and* Response *(response)* types
0	CONTROL
1	STATUS
2	SPECIFIC INQUIRY
3	NOTIFY
4	GENERAL INQUIRY
5-7	Reserved for future specification
8	NOT IMPLEMENTED
9	ACCEPTED
10	REJECTED
11	IN TRANSITION
12	IMPLEMENTED/STABLE
13	CHANGED
14	Reserved for future specification
15	INTERIM

Table 3.11: Control and response types and associated values

The remaining fields within the AV/C command and response frames are identical in meaning, although the values that they contain vary. For example, the *subunit_type* and *subunit_ID* provide a means of addressing the recipient of a command in an AV/C command frame, whereas they address the source of a response in the response frame. The *subunit_type* field contains a value that specifies the type of subunit involved in this transaction. The possible subunit types are provided in table 3.12. The *subunit_id* field's purpose is to specify the instance number of the subunit addressed, although a number of other values are also possible. A selection of these values are provided in table 3.13

subunit_type	Meaning
0	Video monitor
3	Disc recorder/player
4	Tape recorder/player
5	Tuner
7	Video camera
1Ch	Vendor unique
1Dh	All subunit types
1Eh	*subunit_type* extended to next byte
1Fh	Unit
All other values are reserved for future specification	

Table 3.12: subunit_type values and meaning

subunit_ID	Meaning
0-4	Instance number
5	*Subunit_ID* extended to next byte
6	Reserved for all instances
7	Ignore

Table 3.13: subunit_ID values and meaning

The *opcode* field specifies the operation that is to be performed, or status to be returned within the context of the command specified. The permissible *opcode* values, listed in table 3.14, are divided into ranges for commands addressed to subunits, units, or both.

Value	Addressing Mode
0h – Fh	Unit and subunit commands
10h – 3Fh	Unit commands
40h – 7Fh	Subunit commands
80h – 9Fh	Reserved for future specification
A0h – BFh	Unit and subunit commands
C0h – DFh	Subunit commands
Eeh – FFh	Reserved for future specification

Table 3.14: opcode ranges and the specified modes

An example of the use of the *opcode* field is the function block control command, where the value of the *opcode* field is defined to be B8h [AV/C Monitor Subunit Model and Command Set, 1999]. The *operand* fields contain values that are relevant to the specified subunit, and opcode.

3.5.1.3 AV/C Communications

The communication model associated with AV/C is one of command and response, with the node issuing the command being referred to as the controller, and the node

receiving the command as the target. The controller directs its messages, which consist of an AV/C command frame encapsulated within an FCP frame, to the FCP_COMMAND register on the targeted node. The target is required to respond within 100ms, with a response directed at the FCP_RESPONSE register on the controller. This communication model is represented graphically in figure 3.19.

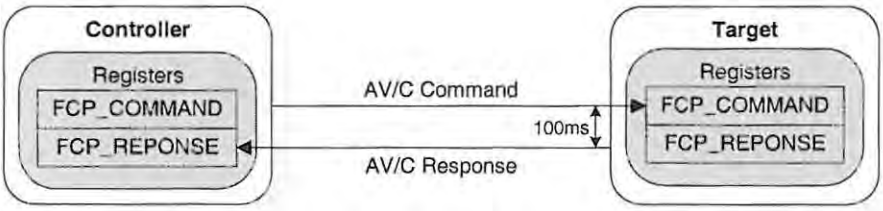

Figure 3.19: AV/C communications model

The possibility exists that the target will not be capable of completing the command issued by the controller in the specified 100ms, in which case the target will respond with an INTERIM, or intermediate, message. A final response will then be issued by the target to the controller, upon completion of the command. The communication model that involves this intermediate step is demonstrated graphically in figure 3.20.

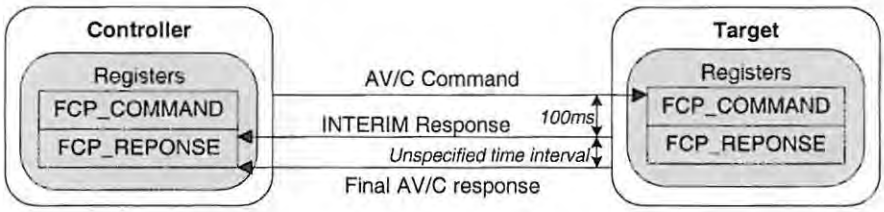

Figure 3.20: AV/C intermediate communications model

Permissible responses by targets include, NOT IMPLEMENTED, ACCEPTED. REJECTED, and INTERIM, the values of which are provided in table 3.11. A NOT IMPLEMENTED response indicates that the target does not support the control command requested by the opcode and operand[n], or the command is addressed to a subunit that is not implemented. ACCEPTED indicates that the specified control command is supported by the target, and the target is in a state favorable to the execution of the command. It does not, however, indicate the successful completion of the specified command. A REJECTED response indicates to the controller that the command is supported by the target, but cannot be executed due to the state of the target. The INTERIM command is used to indicate that the target cannot generate a final response to the command within the specified 100ms time limit, as illustrated in

figure 3.20. In this case the target will, in an unspecified time limit, generate an ACCEPTED or REJECTED response and transmit this response to the controller.

As illustrated in table 3.11, there are a number of defined command types (*ctype*). The communication model provided above was explained based on the CONTROL command type. However, the same model is followed for all of the other command types. For example, controllers issue the STATUS command in order to determine the current status of targets. Targets that receive a STATUS command will respond with one of the following responses: NOT IMPLEMENTED, REJECTED, IN TRANSITION, or STABLE. The NOT IMPLEMENTED and REJECTED responses are equivalent for both STATUS and CONTROL commands, and have already been discussed. The IN TRANSITION response indicates that the target is in a state of transition, due to an already acknowledged command or manual intervention. The STABLE command is returned, along with the information requested, when the target is in a state that permits its response.

Other command types include SPECIFIC INQUIRY, NOTIFY, and GENERAL INQUIRY commands. The inquiry commands provide controllers with a means to determine the commands that are supported by targets, while the NOTIFY command enables a controller to receive notifications about state changes within targets through the use of a CHANGED message. This mechanism is discussed further in chapter 4.

3.5.1.4 AV/C Descriptors

The AV/C specification defines a descriptor mechanism that allows controllers to discover and access static and dynamic data structures that describe AV/C units, subunits, and other related components, such as media contents and plugs. An example of such a descriptor is the *subunit identifier descriptor*, which is a data block that contains attribute information describing a particular type of subunit. However, the format and contents of the subunit identifier descriptor vary according to the subunit that it describes.

These descriptors are composed of objects lists and object entries. An object entry is a data structure, presented in table 3.15[3], which is used to describe some entity. Object lists contain these object structures and provide information that is common throughout the list. This object list information is contained within an object list descriptor, provided in table 3.16. For example, a CD could be represented by an object list, with the tracks represented by object entries within that list.

Address offset	***Contents***
00h – 01h	Descriptor_length – Number of bytes that follow
02h	entry_type – Specific type values are defined within subunit specifications
03h	Attributes – Bit flags providing more detailed object information
04h – 05h	child_list_ID – The ID of a child list associated with this entry, if any at all
06h – 09h	object_ID – Uniquely identifies this object, within a specified scope
0Ah – 0Bh	size_of_entry_specific_information – Specifies the number of bytes used by the entry_specific_information field
0Ch – 0Eh	entry_specific_information – Specific to the type of object being referenced

Table 3.15: General object entry descriptor

Address offset	***Contents***
00h – 01h	Descriptor_length – Number of bytes that follow
02h	list_type - Specific type values are defined within the subunit specifications
03h	attributes – Bit flags providing more detailed object list information
04h – 05h	size_of_list_specific_information – Specifies the number of bytes used by the list_specific_information field
06h – 08h	list_specific_information – Specific to the type of list being referenced
09h – 0Ah	number_of_entries(n) – The number of object entries within this descriptor
0Bh – 0Dh	object_entry[0] – The objects described by table 3.10
...	...
...	object_entry[n-1]

Table 3.16: General object list descriptor

In order to utilize these lists effectively, a mechanism is required that provides controllers with the ability to individually address or reference objects within these

[3] The address offsets presented in tables 3.10 and 3.11 are provided for clarity, and are not absolute addressing positions.

lists. Objects are thus identifiable through a list position, or an assigned object identification number (*object_ID*).

A number of commands have been defined that allow controllers to utilize these descriptors. Table 3.17 provides a list of these defined commands, their values, and a brief description of each. These descriptor commands are utilized in the same manner as the AV/C commands discussed in section 3.5.1.3.

Descriptor Command	Value	Description
OPEN_DESCRIPTOR	08h	Enables access to the descriptor
READ_DESCRIPTOR	09h	Reads data from the descriptor
WRITE_DESCRIPTOR	0Ah	Writes data to the descriptor
SEARCH_DESCRIPTOR	0Bh	Searches for a specified data pattern
OBJECT_NUMBER_SELECT	0Dh	Select one or more objects

Table 3.17: Descriptor commands, values and descriptions (common to units and subunits)

3.5.1.5 Unit Commands

All AV/C devices are implemented as units for which a number of unit-specific commands have been defined. An example of which is AV/C commands that are only applicable to an AV device. These commands have a *subunit_type* of 1Fh and a *subunit_ID* of 7, which specifies that the command be addressed to a unit, and all subunits are to ignore the command. Table 3.18 provides a list of valid unit commands, values, and a short description of each.

Opcode	Value	Description
CHANNEL_USAGE	12h	Report on isochronous channel usage
CONNECT	24h	Establish connections between plugs and subunits
CONNECT_AV	20h	Establish AV connections between plugs and subunits
CONNECTIONS	22h	Report connection status
DIGITAL_INPUT	11h	Used to establish and break broadcast connections
DIGITAL_OUTPUT	10h	
DISCONNECT	25h	Break connections between plugs and subunits
DISCONNECT_AV	21h	Break AV connections between plugs and subunits
INPUT_PLUG_SIGNAL_FORMAT	19h	Configure or report plug signal formats
OUTPUT_PLUG_SIGNAL_FORMAT	18h	
SUBUNIT_INFO	31h	Report subunit information
UNIT_INFO	30h	Report unit information

Table 3.18: Unit commands, values and descriptions

The majority of these commands provide facilities to perform operations on connections between units and subunits. The plugs involved in these connections are serial bus input and output plugs, subunit source and destination plugs, or external input and output plugs. The serial bus input and output plugs are an abstraction of the serial bus interface located on an AV unit. It is this interface that permits the unit access to the IEEE 1394 bus. The subunit source and destination plugs provide subunits with a means of routing streams to and from the subunit respectively, while the external input and output plugs allow the unit to interface with AV units not on the serial bus.

An example of the use of these plugs is demonstrated by the CONNECT command, between which the following connections are permissible:

- A connection between a source and destination subunit plug, serves to carry a stream within an AV unit.
- A source subunit plug may be connected to a serial bus, or external output plug to allow a stream to flow from a subunit.
- A serial bus, or external input plug may be connected to a destination subunit plug, to allow a stream to enter a subunit.

3.5.1.6 Common Unit and Subunit commands

The remaining commands to be discussed are those that are applicable to both units and subunits regardless of type. These commands are provided in table 3.19, with the exception of the descriptor commands that are common to both units and subunits but have been discussed in section 3.4.1.4 and are presented in table 3.17.

Opcode	Value	Description
VENDOR_DEPENDENT	00h	Vendor dependent commands
RESERVE	01h	Acquire or release exclusive control of target
PLUG_INFO	02h	Information about serial bus and external plugs
POWER	B2h	Control power state

Table 3.19: Common unit and subunit commands

All of the commands discussed are supported by units and only a selection of these are applicable to subunits. This selection is represented in table 3.20.

Opcode	Value
VENDOR-DEPENDENT	00h
RESERVE	01h
PLUG_INFO	01h
OPEN_DESCRIPTOR	08h
READ_DESCRIPTOR	09h
WRITE_DESCRIPTOR	0Ah
SEARCH_DESCRIPTOR	0Bh
OBJECT_NUMBER_SELECT	0Dh
POWER	B2h

Table 3.20: Commands applicable to Subunits

An example of a subunit to which these commands apply and which is relevant to sound installations is the audio subunit.

3.5.1.6.1 AV/C Audio Subunit

The audio subunit specifies the data structures and command sets that are supported by consumer electronic audio devices, and is concerned with syntax and semantics of commands sent to audio subunits implemented on these devices. It also describes data structures that indicate which audio formats can be processed by audio subunits [1394 Trade Association, 2000].

The AV/C model presented in the previous section allowed the connection of various plugs, and can be referred to as a *patch* model, as any input can be patched to any output. The drawback of this method is that not all inputs and outputs can be connected in a meaningful manner due to either incompatible audio streams, or nonexistent audio paths, and although an arbitrary number of subunits can be defined, the number of plugs within each subunit would prove restrictive for a complex audio device [AV/C Audio Subunit Specification, 1999]. A possible solution to this problem is the use of function blocks, which reside within an audio subunit, and provide the basic building blocks for complex audio systems.

Connections between audio function blocks, illustrated in figure 3.21, are established through the use of function block, and subunit plugs. The subunits plugs provide an interface to allow streams to enter and exit the subunit, while the function block plugs serve to establish connections between other functions block plugs, or the subunit plugs. The subunit plugs have the added responsibility of ensuring that streams are unpacked, unframed, decoded, and the relevant copy protection catered for, as is required by the function blocks. An audio CODEC function block, if present, provides audio conversion facilities that may be utilised by subunit plugs for this purpose.

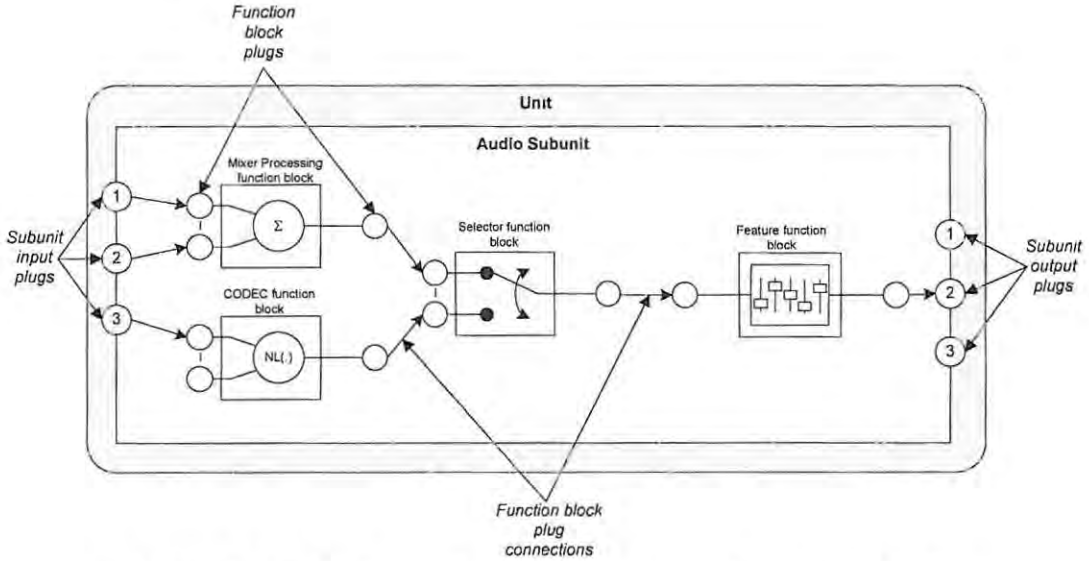

Figure 3.21: Audio function subunit, with audio function blocks

Function block plugs carry data, termed an audio channel cluster, into the function blocks. The audio channel clusters consist of a number of logical channels that share a close synchronous relationship, and are not necessarily in digital form. An example

of such an audio stream is a simple stereo signal, where both left and right channels share such a relationship.

Four function block types have been defined, which are considered adequate to represent most consumer audio functions available, these are the:

- Selector function block,
- Feature function block,
- Processing function block, and
- CODEC function block.

The selector function block selects one input signal from a range of *n* input signals, and routes this unaltered signal to the one function block output plug. The selector function block icon is presented below in figure 3.22.

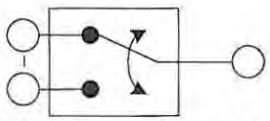

Figure 3.22: Selector function block icon

The feature function block, represented by the icon in figure 3.23, provides for the manipulation of multiple incoming channels. This function block optionally exerts the following control over each logical channel and over all channels as a master controller [1394 Trade Association, 2000]:

- Mute,
- Volume,
- Balance,
- Tone control,
- Graphic equalisation,
- Automatic gain control,
- Delay,
- Bass boost, and
- Loudness.

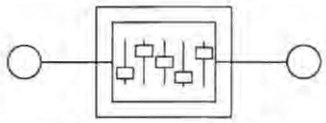

Figure 3.23: Feature function block

Typical attributes that are associated with these controls are:

- CURRENT setting,
- MINIMUM setting,
- MAXIMUM setting,
- RESOLUTION,
- DURATION,
- MOVE,
- DELTA, and
- DEFAULT.

The usefulness of these attributes is demonstrated with a simple example. Consider a volume level control within a feature function block; by issuing the relevant function block command, the values of these attributes can be retrieved or modified. This allows a controller to set the desired volume level for any particular channel.

The processing function block permits multiple logical input channels, which exist in a number of different audio channel clusters, to be processed and grouped into multiple channels existing in one audio channel cluster. This cluster is routed to the single function block output plug. The mixer function block, represented by the icon in figure 3.24, is such an example, as all input logical channels can be mixed into any logical output channel that resides within a single audio channel cluster. Other processing function blocks include the generic, up/down-mix, Dolby Pro-logic, 3D-stereo extender, reverberation, chorus, and dynamic range compressor processing function blocks.

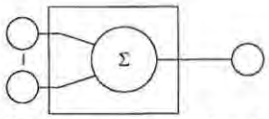

Figure 3.24: Mixer function block icon

The remaining function block is the CODEC function block, illustrated in figure 3.25, which is responsible for the transformation of non-linear encoded audio bit-streams into linear audio bit-streams. CODEC function blocks may be connected to subunit plugs to provide the necessary audio conversion facilities.

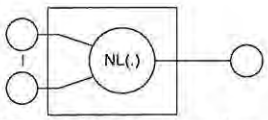

Figure 3.25: CODEC function block icon

Features common to all of these function blocks include:

- A single function block output plug, and
- Described by function block dependent information

3.6 Summary

This chapter provided a general overview of the IEEE 1394 architecture, and provided insight into the defined protocols applicable to sound installations. The mechanisms defined within the A/M protocol for the transmission of audio and music data, including raw audio and MIDI, were examined. Finally, a discussion of AV/C, and the audio subunit, which permits the control of IEEE 1394 audio devices, was included.

The next chapter examines the applicability of the IEEE 1394 bus to professional audio installations and describes an approach that allows the IEEE 1394 bus to function as the transport mechanism for existing Internet Protocol (IP) based control protocols.

Chapter Four

The Application of IEEE 1394 to Sound Installation Control

As the development of professional audio devices often receives less attention than consumer devices, which can be attributed to market demand, there is a tendency to adapt protocols developed for home entertainment products to suit the professional arena. While this approach allows professional products to be rapidly designed and produced, the features often sacrificed are interoperability and extensibility.

This chapter lays out the topology of professional sound installations, and examines the applicability of IEEE 1394 to these environments. The discussion includes an examination of currently available protocols, as well as the feasibility of an IP-based approach. The limiting factors and advantages gained through the use of these approaches are highlighted within the discussion.

4.1 IEEE 1394 Sound Installation Configurations

There are a variety of topological layouts utilized within the sound installation industry, with the final layout often being chosen based on the limiting factors associated with the technology used. Examples of such limitations include cable length restrictions, multiplexing capabilities, and data throughput. Additionally, many existing sound installations, such as the QSControl system, employ separate cable runs for the transmission of audio and control data, as is depicted in figure 2.7.

With the introduction of IEEE 1394 to the sound installation environment, a number of these limitations are removed. For example, audio and control data can be transmitted using the same technology and via a single cable, as illustrated in figure 4.1. Peak Audio provided a similar solution with the development of their CobraNet technology, which also permits the simultaneous transmission of audio and control

data over the same cable. Section 2.4.2.1 provides a discussion of CobraNet technology.

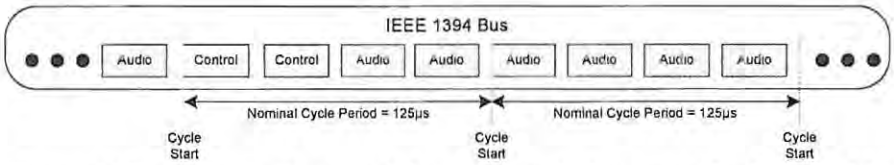

Figure 4.1: Audio and control data transmission over the IEEE 1394 bus

IEEE 1394 technology stretches the boundaries of the current limitations. For example, isochronous audio transmissions are guaranteed real-time delivery and the necessary bandwidth, while control data can be sent asynchronously, utilizing few resources. This reduces the number of cable runs required, and simplifies the conceptual visualization of the system for the user, as all connections and routings can be made through intuitive connection management software.

It is possible that established control or audio transmission mechanisms within legacy devices remain in place, and legacy IEEE 1394 adapters are fitted to these devices, thereby allowing these legacy devices access to the IEEE 1394 bus, as illustrated in figure 4.2. An example of such a device is the FB-88 (FireBob) [Digital Harmony Technologies, Inc., 2000], which permits both control and audio data to be routed to the device via the IEEE 1394 bus. This is achieved through the provision of suitable interfaces between the legacy device and IEEE 1394 adapter. Section 4.1.3 expands further on the use of a legacy device and adapter.

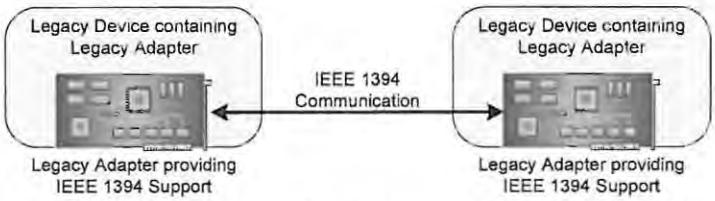

Figure 4.2: Legacy device and adapter enabling IEEE 1394 support

4.1.1 MediaMatrix Audio System

The MediaMatrix Audio system was introduced in section 2.1. It consists of a high-end PC with specialized hardware and software known as the MediaMatrix Frame and one or more BoB's (Break out Boxes). These BoB's exist in analog and CobraNet

form, which provide a variety of configurations. The CobraNet BoB's alleviate the distance restrictions encountered with its predecessors, as described in section 2.1, and therefore provide a greater amount of flexibility.

As the CobraNet facilities, described in section 2.4.2.1, are similar to those provided by the IEEE 1394 architecture, it is feasible that CobraNet devices be retrofitted with an IEEE 1394 interface. An example of such a device is the BoB associated with the MediaMatrix Audio System, as depicted in figure 4.3. The advantages of such an implementation include the distribution of additional audio sequences, guaranteed bandwidth for audio data, very low latency, and an appropriate mechanism for the distribution of control data. The control and monitoring mechanism currently utilized by CobraNet-enabled devices is SNMP. This control and monitoring mechanism could be maintained on an IEEE 1394 network through the introduction of appropriate IP transmission and reception facilities. The feasibility of introducing such a facility is discussed in section 4.3 and chapter 5.

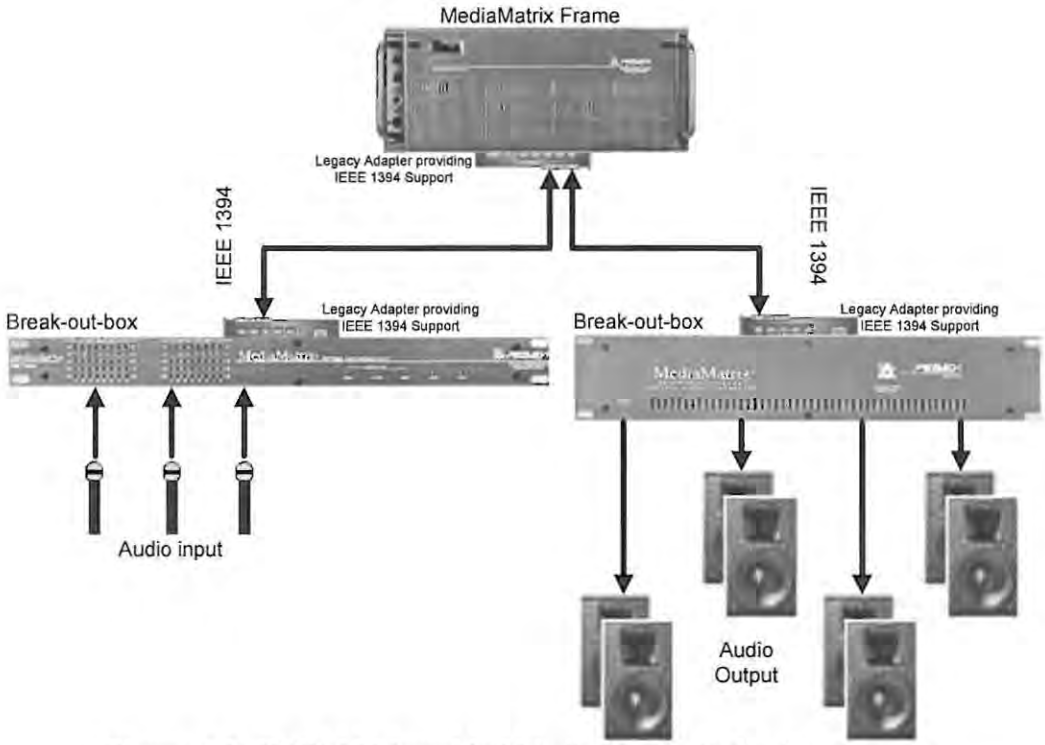

Figure 4.3: IEEE 1394 enabled MediaMatrix Audio System

Figure 4.3 provides a similar illustration to that presented in figure 2.1, with the exception that the above system utilises IEEE 1394 networks for the transport of

audio and audio control data. All of the IEEE 1394 devices illustrated in the above figure require the addition of an IEEE 1394 interface, such as a legacy adapter. For example, the MediaMatrix frame would require the addition of an IEEE 1394 enabled DPU card. However, this would not alter the functions that are performed by the MediaMatrix Frame. The above IEEE 1394 MediaMatrix solution could be extended by providing IEEE 1394 interfaces to audio renderers, represented by the speakers in figure 4.3, thereby eliminating the need for a BoB to cater for the conversion of audio from digital to analog form.

4.1.2 IQ Audio System

The IQ Audio System is introduced in section 2.2 where the audio devices reside on an IQ Bus loop. Each of the devices on this loop can be controlled via an IQ Interface device attached to a PC. Through the introduction of an IQ Server, remote clients are able to control devices that are located on a bus that is connected to the server. The cable restriction associated with the bus loop is 3000 feet.

Crown Audio Inc. recently launched an IP-based version of the IQ Audio System, known as TCP/IQ. The software for this version resembles the client encountered in the previous version, although audio devices are no longer required to be connected to a bus loop. Instead, IQ compatible audio devices contain an Ethernet port which is utilized for the transport of audio and control data using CobraNet technology. The legacy bus loop and the TCP/IQ network can interoperate through the use of a TCP/IQ Gateway.

Figure 4.4 represents a TCP/IQ network that is IEEE 1394 enabled. Each of the devices present on the network requires an appropriate IEEE 1394 interface, which is likely to be a legacy adapter for the IQ devices. The IEEE 1394 interface for the IQ Client could simply be an IEEE 1394 compliant expansion card. As the TCP/IQ network is IP based, no changes to the communications protocol or addressing mechanisms would be required as IP datagrams can be transmitted over the IEEE 1394 bus. This would allow the software currently utilised on the client to be reused without any alterations. Chapter 5 focuses on the provision of IP capability to the IEEE 1394 bus to allow the reuse of protocols and software.

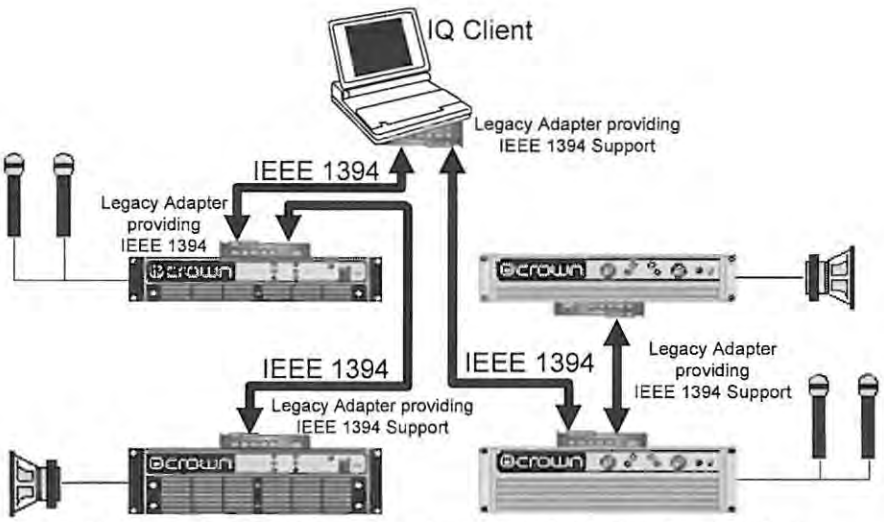

Figure 4.4: TCP/IQ network using IEEE 1394 technology

4.1.3 QSControl System

The commercially available QSControl system, as introduced in section 2.3, requires separate cables and transmission technologies for the distribution of audio and control data. If however, the devices within the QSControl system were IEEE 1394 compliant, then control and audio data could be transmitted simultaneously through the IEEE 1394 bus, as is illustrated in figure 4.5. This would reduce system complexity, wiring costs, and extend the capabilities of the QSControl system.

A legacy device, in this context, is an audio device that is not IEEE 1394 compliant, but has the capability of being used on the IEEE 1394 bus, whereas a legacy adapter is an IEEE 1394 node that has the capability of interacting with an existing audio device providing it with IEEE 1394 capabilities. For example, the CM16 devices utilised within the QSControl network can be considered as a legacy device. These devices allow for the control of their attached amplifiers and the distribution of audio data to those amplifiers. An example of a legacy adapter is a Digital Harmony Technologies DHIVA IEEE 1394 device, which is detailed in the following chapter.

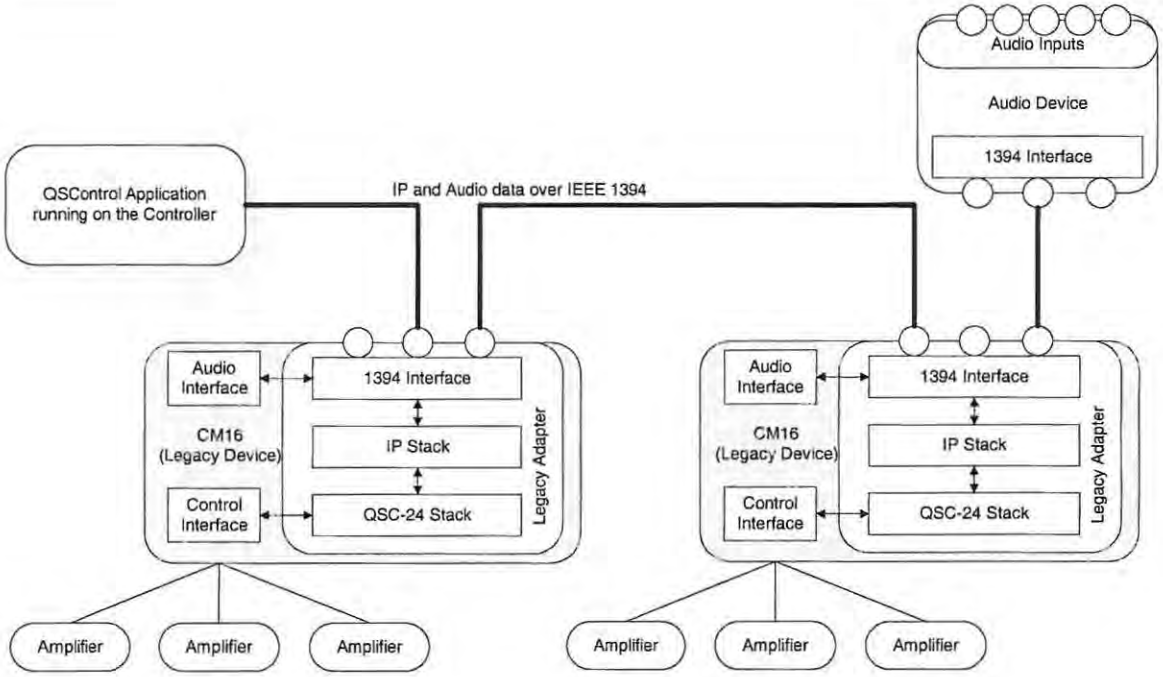

Figure 4.5: An illustration of control and audio data being transmitted over the IEEE 1394 bus, using the QSControl system

A suitable interface is required in order for the legacy device and adapter to communicate, which in this case is an RS-232 serial port and compatible audio ports for the transfer of control and audio data respectively. As this study is focused on the distribution of control and monitoring data over the IEEE 1394 bus, the transmission of audio data is largely ignored. However, the IEEE 1394 development board utilized within this study and described in chapter 5, provides a Fully Programmable Gate Array (FPGA) for the extraction of audio in a variety of supported formats, such as I^2S. It is this facility that provides the audio interface to legacy devices, as the audio stream extracted from the IEEE 1394 bus can be routed to a compatible port on the legacy device.

As each of the technologies discussed utilize IP-based networks and provide similar facilities, the next chapter provides implementation specifics on the addition of IP capability to the IEEE 1394 bus. This allows these protocols to be utilized over the IEEE 1394 bus. However, an examination of the control and monitoring protocols currently available for sound installations is required, in order to determine the necessity for the provision of non-IEEE 1394 based protocols. The criteria against which these protocols are to be considered is described in the following section.

4.2 Evaluation Criteria

Chapter 2 introduced a number of possible configurations and technologies encountered within sound installations. From these installations a number of common characteristics were extracted, and presented in section 2.5 in the form of an object model. The scenarios presented within this object model provide some of the evaluation criteria against which control and monitoring protocols will be considered for IEEE 1394 based sound installations. The evaluation criteria can be divided into *device* and *installation* specific criteria. The device criteria are specific to an individual device, whereas the installation specific criteria are required to cater for a system configuration across multiple devices. Table 4.1 provides a selection of the scenarios extracted and the terms associated with the device criteria that are to be used for evaluation.

Scenario Name	Evaluation Criteria
Device Roll-call	Discovery
Device Capabilities Query	Description
Device Control	Control
Device Monitoring	Monitoring
Device Interface	Presentation

Table 4.1: Sound installation evaluation criteria

The installation specific criteria that are to be considered include connection management, latency management and synchronization. Connection management is the procedure of establishing an audio path between input and output ports. These procedures are often catered for within the control and monitoring protocol, but are occasionally deferred to the transport mechanism utilized. As the transmission latency associated with IEEE 1394 network is fixed at a worst case value of 354.17μs, there is little need to consider the management of this latency on a single segment IEEE 1394 bus. However, the latency present within bridged environments requires further consideration as audio is required to traverse multiple busses, but is beyond the scope of this discussion.

The selection of an appropriate synchronization source forms an integral component of any sound installation, as all digital audio devices are typically synchronized to the same clock. Facilities are provided for the distribution of synchronization information over the IEEE 1394 bus in the form of a time-stamping mechanism for isochronous

data, with the time-stamp being provided by the transmitting device. However it is imperative that the transmitting device share a common clock with the Cycle Master present on the bus. This is possible as the Cycle Master transmits cycle start packets every 125μs. Further information regarding the role of the Cycle Master is provided in section 3.2.1.

As this chapter deals with the adaptation of legacy devices to utilize the IEEE 1394 bus, the synchronization sources utilized by these devices and the synchronization options provided by the legacy adapter must be considered. The adapter utilized within this study, which is described in detail in chapter 5, provides synchronization options for an internal clock, word clock obtained from an external source, clock derived from an ADAT stream, or determined from an isochronous stream. The synchronization option utilized depends on the legacy device to which the adapter is attached. Of importance to this discussion is whether facilities are provided within the control and monitoring protocols considered to allow the selection of an appropriate clock. In the discussion that follows the selection of a suitable clock source can be considered to be part of the control facilities provided by the protocol under examination, unless specified otherwise.

4.3 Currently Available IEEE 1394 Protocols for Sound Installations

There are a select few control protocols already in operation over the IEEE 1394 bus, with the AV/C, discussed in chapter 3, and MIDI protocols probably being the most prevalent. The AV/C protocol has been developed largely for the control of home entertainment devices, and tends not to scale well to professional audio devices. This can be attributed to the complex nature of professional devices, their extensive features, and the large number of variable parameters within them.

The next two sections consider the suitability of the AV/C and MIDI protocols against the criteria identified and detailed within section 4.2.

4.3.1 AV/C and Sound Installation Control

The evaluation criteria of discovery and description are fulfilled through the AV/C descriptor and information block mechanism. This mechanism provides static and dynamic information that describe AV/C units, subunits and other related components

such as media contents and plugs. These descriptors are all composed of object lists and object entries, which a controller can traverse to extract the relevant information concerning the device. Section 3.5.1.4 provides further details regarding these mechanisms.

Due to the standardisation of the FCP command and response registers, no prior knowledge of a device is required before AV/C communication can begin. Controllers can ascertain which devices present on the bus are AV/C capable by examining their Configuration ROM's, as all AV/C devices contain an entry indicating this capability. Professional audio devices are inherently complex; consequently the description of these devices is also complex. The ideal description mechanism is therefore one that is light-weight and easily processed. AV/C information blocks are easily parsed by a controller and are extensible [1394 Trade Association, 2000].

Section 3.5.1.2 introduced the commands associated with AV/C which are specified within the *ctype* field of the AV/C command frame. One of these commands was the *control* command, which provides a mechanism that can be used for control within professional sound installations, while the required monitoring facilities are provided by the notification mechanism (*notify* command). This notification mechanism allows devices or controllers to subscribe to device state changes within other nodes, and will in turn be notified should such an event occur. However, each node is notified only once, after which devices must re-register for future state changes. The AV/C *status* command can function as a monitoring mechanism by allowing devices to query, but not alter, another devices state.

The presentation of a device is catered for by the Panel Subunit specification [1394 Trade Association, 2001], which provides user interface services. This is achieved through the definition of a number of standardized control types, such as buttons, and text fields, and a command set to allow access to these controls. A controller that requires the control of a device, queries the panel subunit for the graphical data, and constructs the user interface from the data returned. The user manipulation of these on-screen controls generate commands which are sent to the panel subunit. The panel

subunit receives these commands, performs the requested action and, if necessary, updates the graphical interface.

AV/C connection management is utilised for the establishment of an audio path between input and output plugs. As the AV/C model consists of a number of different types of plugs, there are a number of connections that can be established for every path. For example, figure 4.6 illustrates source and destination units. The source plug associated with the source unit is where the audio stream will leave the subunit. This source plug must be connected to a serial bus output plug, in order for the audio data to be transmitted across the IEEE 1394 bus within an isochronous transaction. On the destination unit the serial bus input plug will receive the data from the bus, and forward it on to the connected destination plug. The connection management procedures associated with AV/C are responsible for the creation of these connections, and are defined within the "Connection and Compatibility Management Specification" (CCM) [1394 Trade Association, 2000].

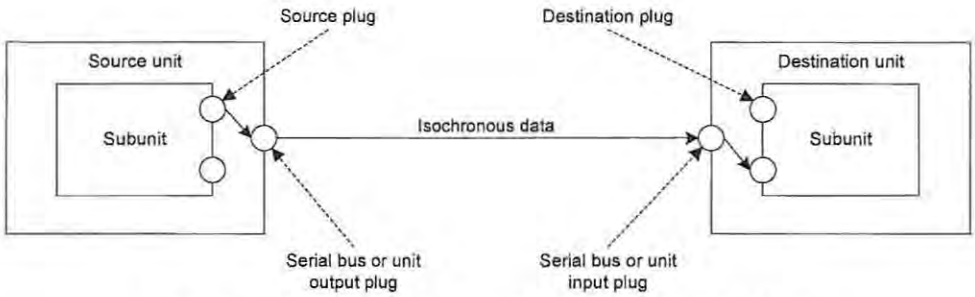

Figure 4.6: Plugs associated with a unit and subunit

The CCM specification caters for the connection of data streams, but more than one audio sequence can exist within a stream. It is therefore essential that these encapsulated sequences be individually extractable. This extraction facility is provided by the "Music Subunit" specification [1394 Trade Association, 2001], which describes data structures and a command set for professional audio equipment that have MIDI capability and are connected to the IEEE 1394 bus. This specification provides for MIDI, Audio, SMPTE time code and Sample Count data signal flows, as well as establishing Audio reference synchronisation methods for the synchronisation of Audio and MIDI signals between devices [1394 Trade Association, 2001].

Data streams are carried to the Music Subunit using isochronous transactions, which terminate at the destination plugs present on the subunit. The streams can be selectively routed to Music Input plugs, as represented in figure 4.7. A destination plug may be routed to one or more Music Input plugs. Similarly, there exist Music Output plugs which carry data generated by the Music Subunit to source plugs for transmission as isochronous transactions. One or more Music Output plugs may be connected to a single source plug. The types of plugs currently defined include:

- Audio,
- MIDI,
- SMPTE time code,
- Sample Count, and
- Audio SYNC.

Source: AV/C Music Subunit 1.0 [1394 Trade Association, 2001]

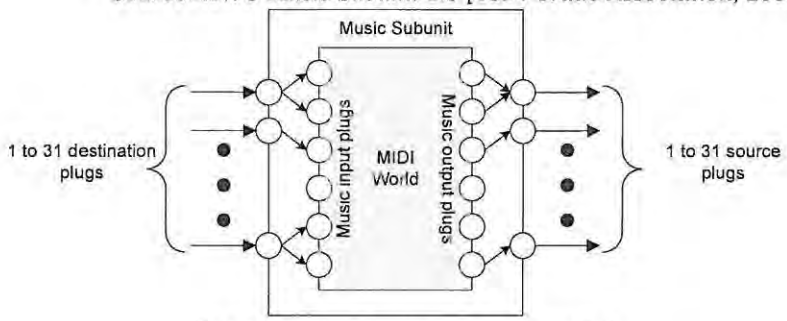

Figure 4.7: Music subunit model

The above Music Subunit model allows the extraction of individual sequences from within a stream, but does not provide any processing capabilities that might be required by professional audio devices. However, the Audio Subunit introduced in section 3.5.1.6.1, provides processing capabilities through the definition of function blocks. There is currently no defined mechanism to allow the Music and Audio Subunits to interoperate. It is possible that the source plugs of a Music Subunit be connected to the destination plugs of an Audio Subunit. This would allow the streams extracted within the Music Subunit to be processed by the function blocks provided within the Audio Subunit.

The Audio Subunit defines a number of facilities that professional audio devices require, but these require closer inspection for their suitability to professional audio

devices. For example, an investigation of using the Audio Subunit plugs to model the facilities provided within a Yamaha 03D mixer [Yamaha Corporation, 1997], revealed the following:

- Although a number of the required facilities are defined they are not of the same quality as professional audio devices. For example, the controllable parameters within the Chorus processing function block consist of the Chorus Level, Chorus Modulation Rate, and Chorus Modulation Depth. The Chorus component present on the O3D mixer provides for the control of modulation speed, depth of pitch modulation, depth of volume modulation, frequency of low shelving filter, gain of low shelving filter, delay time from the direct time to the modulated time, frequency of the parametric equaliser, gain of the parametric equaliser, modulation waveform, frequency of the high shelving filter, and gain of the high shelving filter. This illustrates that the AV/C function block mechanism, while providing a number of appropriate types of function blocks, are more suited to consumer than professional audio devices. In particular the function blocks are not expandable to incorporate the features that are required. Through the examination of similar components such as Phase, Flange, Equalisation, and Reverberation the same conclusions can be drawn.
- The O3D mixer provides digital control over the routings that are established within the mixer. For example, once an analog input is converted into the digital form it is able to be routed along one of the available effects, bus, auxiliary, stereo, or solo lines. This provides flexibility in the establishment of routings, and allows the storage and retrieval of these connections to and from memory, respectively. For example, an audio signal can be routed to an internal effects unit, and then routed onto the stereo output lines. Within AV/C there are no specific function blocks that allow for the establishment of routings. All function blocks are constrained to only one function block output plug. Although this one output plug may contain multiple streams, there is no facility to allow the extraction of these sequences into separate streams. A possible solution is to provide, at the Audio Subunit destination plugs, a stream that contains only one sequence. In other words, to provide to the Audio Subunit, sequences that have already been extracted from the

isochronous transaction in which they were transmitted. This would allow the various function blocks such as the Selection, Up/Down-mix and Mixer function blocks to operate only on one sequence. Consequently, the connection of function block plugs to other function block plugs would be synonymous with the routing of audio signals, as each stream will contain such a signal. The extraction of these sequences from the streams could be performed by the Music Subunit, and the subunit source plugs of the Music Subunit connected to the destination plugs of the Audio Subunit. The reassembly of these streams could be performed by the function blocks provided within the Audio Subunit, such as the Mixer function block, or by the Music Subunit. The limitation of this approach is that only 31 destination plugs are available on a subunit, thereby restricting the number of sequences to 31 [IEC 61883-6, 1998].

As the transmission latency present within a single segment IEEE 1394 bus is predictable, there are no specific facilities specified within AV/C to cater for its management. However, additional latency due to the buffering and processing of audio within devices will be introduced. The management of this latency is required, but there are currently no facilities within AV/C to cater for this. Facilities are provided for synchronisation within the Music Subunit, and through the definition of a rate control specification known as the "Command Set for Rate Control of Isochronous Data Flow" [1394 Trade Association, 2000]. The Music Subunit defines an Audio SYNC Music Input plug which provides access to the synchronisation information transmitted within an isochronous transaction. This plug will be connected to the subunit destination plug that contains the synchronisation information.

The Rate Control specification defines clock-based and command-based methods to allow for the synchronisation of devices. The command-based approach is designed to manage buffer underflows and overflows, and is therefore applicable to a single source and single destination. The clock-based approach allows for the synchronisation of a data stream with a given clock. This is achieved through the provision of a clock source, clock sink, data source, and data sink, as illustrated in figure 4.8.

Source: *AV/C Command Set for Rate Control of Isochronous Data* [1394 Trade Association, 2000]

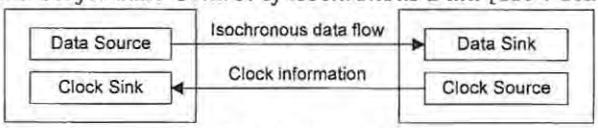

Figure 4.8: Clock-based rate control

The clock information is generated by the clock source and delivered to the clock sink. The clock information is received by a serial bus input plug or external input plug, and routed to a destination plug on the subunit that will be outputting the stream. A source plug of this subunit is connected to a serial bus output plug which carries the timing information within an isochronous transaction. Figure 4.9 illustrates the transmission of timing information through a subunit.

Source: *AV/C Command Set for Rate Control of Isochronous Data* [1394 Trade Association, 2000]

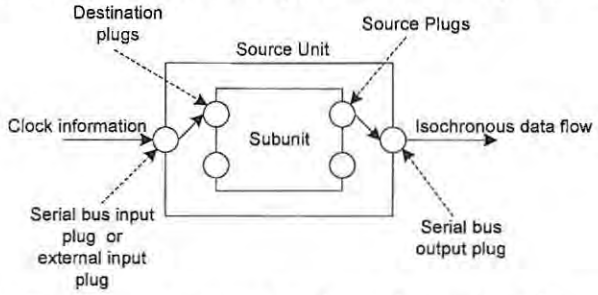

Figure 4.9: The transmission of timing information through a subunit

The combination of the Music Subunit and the defined Rate Control mechanisms provide the required mechanism for the selective extraction and distribution of timing information, respectively.

4.3.2 The MIDI Protocol

The MIDI protocol can be thought of as a master-slave protocol, in that a device can be transmitting on one of sixteen defined channels to all other devices that have been configured to listen to the same channel. These channels do not transmit audio data, but are the transport mechanism for control information. This information is transmitted in the form of 10-bit words, composed of a start bit, 8-bits of data, and a stop bit. The musical data to be transmitted is broken down into messages, which

consist of a status byte followed by one or more data bytes. Status and data bytes are distinguishable as a status byte has its first bit set to 1, and a data byte's first bit is 0.

MIDI messages include Channel, System Common, System Real-Time, and System Exclusive messages. MIDI Channel messages incorporate a MIDI channel number. The two types of Channel messages are Channel Voice and Channel Mode messages. Channel Voice messages are used to manipulate the on-board performance controls of a MIDI enabled instrument, as though these controls were manipulated themselves. This makes the control of one MIDI device from another possible. Channel Mode messages are an extended form of the *control change* Channel Voice messages, and determine whether a MIDI device will respond to all channels or only a selected one, as well as indicating if a device will assign incoming Channel Voice messages to its internal voices polyphonically or monophonically.

System Common messages are targeted at all MIDI nodes within a MIDI system, and therefore do not include a channel number. These messages are used for the transmission of MIDI Time Code (MTC), Song Position Pointer, Song Select, Tune Request, and End of Exclusive messages. System Real-Time messages, like System Common messages, are also targeted at all MIDI nodes present. With the exception of Active Sensing and System Reset messages, System Real-Time messages are utilised for the synchronisation of MIDI devices such as sequencers and drum machines.

System Exclusive messages provide for the communication of device specific information, with the only restriction being that each message must carry a unique manufacturer identification number. The supported formats of these messages provide for the transmission of Universal Non-Real-Time and Real-Time messages. Within these two formats, the transmissions of sample dump data, MTC, and device inquiry messages have been defined. The MTC system exclusive message is used to locate a point at which to begin playing, while the MTC System Common message is utilised for the synchronisation of the master and its connected slaves.

MIDI devices typically provide in, out, and thru MIDI ports. The thru port allows the daisy chaining of MIDI devices, which enables multiple MIDI devices to be

connected to the same network, as illustrated in figure 4.10. Figure 4.11 provides a logical channel layout of the daisy-chained topology presented in figure 4.10. Each of the devices represented will receive MIDI messages on their configured channels, and ignore the rest. All received MIDI messages will however, be forwarded regardless of channel number.

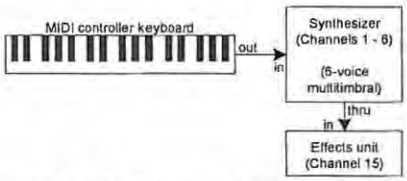

Figure 4.10: Physical view of a MIDI network

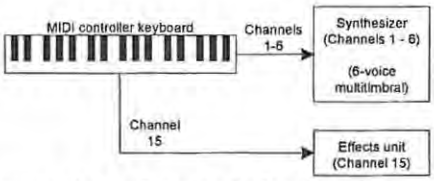

Figure 4.11: Logical channel layout of figure 4.3

Although MIDI was not originally designed as a control and monitoring protocol for professional sound installations, a number of extensions to the original protocol have increased its applicability. For example, the MIDI Show Control (MSC) extensions to the MIDI protocol allow for the peer-to-peer control of various show control devices. MSC has been left as open as possible, thereby leaving room for expansion, but still defines the bare necessities. MSC messages include a device identifier specifying the address of the targeted device, and allows for the addressing of 112 devices. In addition, the MSC messages include a command field that specifies the type of equipment that is to be addressed, such as Music, CD Players or Robots [Huntington, 2000].

Another defined extension to the original MIDI specification is the MIDI Machine Control (MMC) protocol, which provides a standardized way for interfacing with a variety of audio and video devices. MMC provides basic facilities such as play, forward, pause, and eject. Devices complying with MMC fall into the categories of controllers and slaves. Controllers transmit commands to the slaves, which are acknowledged with a response. MMC defines both read-only and read-write registers on the controlled devices, which allows the controller to query their content or, if read-write, to alter them.

4.3.2.1 MIDI and Sound Installation Control

Discovery within a MIDI system can be achieved through the use of the Device Inquiry message, which is part of the Universal Non-Real-Time System Exclusive message format. This inquiry message allows a device such as a controller to request the identity of receivers within the system, by requesting a device to respond according to a device identification number. If this identification number matches that of the local device, a response should be transmitted. Alternatively, if the device identification number contains the value 7Fh, all devices are required to respond. The response contains information that describes the device, such as device and manufacturer identification numbers, device family code, family member code, and device specific information. This inquiry request and response therefore satisfy the evaluation criteria of discovery and description. However, as the description mechanism for devices is largely device specific, this hinders the ability of a central control point to control all of the devices present within a MIDI system.

Although, the MIDI specification provides for control through the use of Channel messages, no mechanism is defined specifically to cater for the monitoring of device parameters. In addition, no facilities are provided to cater for the notification of parameter state changes from a controlled device to a control point or other connected devices.

Although MIDI has a number of shortcomings with regards to the control and monitoring of devices, it has nevertheless proven very successful, and as such mechanisms have already been put in place to allow its transmission over the IEEE 1394 bus through the use of isochronous transactions. Testimony to this fact is the inclusion of MIDI plugs within the Music Subunit specification. However, through the use of a multiplexing mechanism its transmission speed is restricted to 31.25 kbps, which is the transmission speed for traditional MIDI transmissions. This speed restriction hinders MIDI as a possible control and monitoring protocol for professional audio devices present on the IEEE 1394 bus. For example, professional audio installations can have in excess of 20,000 variable parameters [Rosenthal, M., 20/12/2001, pers. com.] which may all require updating due to a scene change. Assume that each of these parameters is required to be updated for a scene change, and each parameter change consisted only of a status byte and one data byte. As each

of the bytes consist of 10-bits, this requires the transmission of 400,000 bits of information. At the defined rate of 31.25 kbps, the parameter updates would require that the device be occupied for a period of 13 seconds.

The MIDI protocol does not satisfy the presentation criteria, as no suitable mechanisms are provided for the presentation of devices. It is, however, feasible that a database of suitable interfaces be maintained that could be utilised based on the identification number associated with a device. This identification number can be retrieved through the use of the inquiry facility.

Although connection management is not catered for directly within the MIDI protocol, the Music Subunit can be utilised in conjunction with the MIDI protocol to provide these facilities. Similarly, the Music Subunit caters for synchronisation as discussed in section 4.3.1.

4.4 An IP-based Approach to IEEE 1394 Sound Installation Control

While the approach of allowing IP-based protocols to be transmitted over the IEEE 1394 bus is applicable to a number of existing audio control and monitoring protocols, two of these have been selected for examination. These are the QSC-24 and Universal Plug and Play (UPnP) architecture protocols. QSC-24 is currently in use within professional audio sound installations, whereas UPnP is an emerging architecture with the potential to provide a wide variety of services. The prevalence of IP-based control and monitoring protocols is demonstrated by the fact that all of the protocols discussed have been developed or adapted to permit their transmission over IP networks. CobraNet is an example of such a network.

4.4.1 QSC-24 over IEEE 1394

As QSC-24 is a scaled down version of AES-24, as seen in chapter 2, a number of assumptions were made by its designers. One of these assumptions is that the Controller has prior knowledge of the address of the devices that it wishes to control. Thus the discovery mechanism within QSC-24 networks is not ideal, although included within the specification as a future extension is the ability for devices to announce themselves to an address resolution server, as well as a resource registry [QSC Audio Products, Inc., 1997b]. This address resolution and registry model is in

line with the AES-24 protocol specification. The QSControl application does, however, provide a service whereby a network can be searched for currently online QSC-24 devices, which is achieved through the transmission of a discovery message to every IP address within a specified block. This is not the ideal approach, as a number of messages are generated and targeted at IP addresses that do not house QSC-24 nodes.

QSC-24 devices are comprised of a collection of objects, which are either standard or device-specific objects. The standard objects are mandatory for all QSC-24 devices, and are essentially those objects required for device, network and session management. The device-specific objects, are dependent on the capabilities of the device, and include objects for audio processing, routing, and monitoring. The set of objects that are supported describes the functionality of a device. However, since the number of devices currently supporting the QSC-24 protocol is limited, a mechanism to discover the list of objects within a device is lacking, and it is noted as a future extension to the protocol.

Through the modification of parameters contained within the objects of a QSC-24 device, a controller can exert control over that device. These parameters are addressed by specifying the device, object, and parameter address, which are all pre-assigned according to an object hierarchy, as is illustrated in section 2.4.3.1. These parameter values are modified or retrieved through the use of defined Get and Set command messages, which are by default acknowledged messages. The level of control specified within the QSC-24 protocol provides a high level of granularity and is very flexible, as individual parameters or the entire device can be addressed.

QSC-24 devices may contain a number of different types of sensor objects, such as an indicator-selector, load-monitor, meter-level, meter-temperature, meter-voltage, or meter-current object. These objects are collectively responsible for monitoring the state of the device, and reporting this state to the controller. Each of these objects contains *ReportCount*, and *ReportInterval* variable fields. The *ReportCount* field is used to indicate the number of reports that this particular object is set to report, which is decremented by one with every report issued. When the *ReportCount* reaches the value 0, a *SetReportCount* message is generated and sent to the controller, allowing

the controller to reinitialize this value as is required. If this value is not reinitialized, no more reports will be issued. This prevents nodes that the controller may have lost from flooding the network with reports. The *ReportInterval* field specifies the time interval between reports. Alternatively, by specifying a negative value (-N), it allows the generation of a report for every Nth time the measured values changes, which is useful for the monitoring of abnormal conditions such as clipping. Although the monitoring facility provided within QSC-24 is relatively simple, it is very flexible and provides adequate device state information.

There are no facilities within the QSC-24 protocol to allow for the presentation of a device to a control point. Instead, a set of plug-ins are made available, whereby a custom application can be written to suit the required layout. The AES-24 specification does define a number of user interface objects that could be used for the presentation of devices to a controller, although this facility would require the formation of a suitable device description mechanism.

As the QSC-24 protocol was originally developed for a network where the audio and control data were separately distributed, there was little need to provide facilities for the management of connections, latency, and synchronization within the QSC-24 protocol. However, due to the extensible nature of this protocol these facilities could easily be added through the addition of appropriate objects.

4.4.2 The Universal Plug and Play (UPnP) Architecture over IEEE 1394

The UPnP architecture is relatively new, and is defined as "an architecture for pervasive peer-to-peer network connectivity of intelligent appliances, wireless devices, and PC's of all form factors" [Microsoft Corporation, 2000]. It is designed to bring flexible, standards based connectivity to ad-hoc or unmanaged networks, by allowing a device to dynamically join a network, obtain an IP address, transmit its capabilities, and discover other devices, and their capabilities. This functionality is not provided through the use of platform specific device drivers, but rather through the standardization of a set of communication protocols, as illustrated in figure 4.12.

The types of devices encountered in the UPnP architecture are controllers (control points) and controlled devices (services), although devices may contain both of these.

The protocols defined allow communication between the controllers and controlled devices, and are compiled into a protocol stack, illustrated in figure 4.12. The defined transport mechanism is IP, with both the TCP and UDP protocols being utilized. Layered above these protocols are the General Event Notification Architecture (GENA), Simple Service Discovery Protocol (SSDP), HyperText Transfer Protocol transmitted using both UDP multicast (HTTPMU) and unicast (HTTPU) mechanisms, and Simple Object Access Protocol (SOAP). These protocols allow for the communication of discovery, description, control, eventing, and presentation information between controllers and controlled devices.

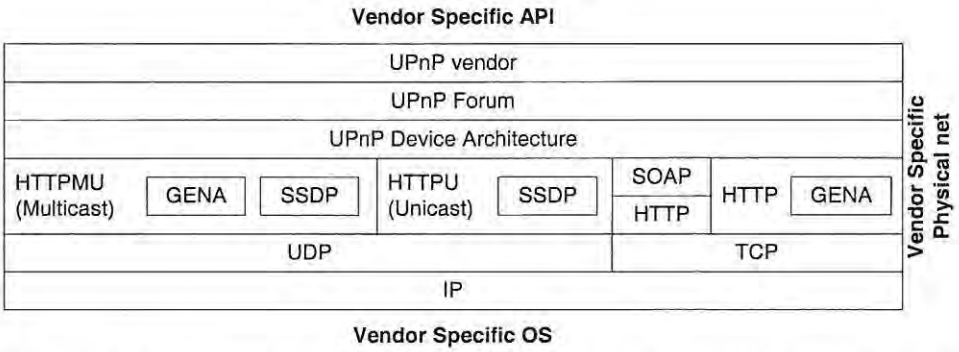

Figure 4.12: Universal Plug and Play (UPnP) architecture protocol stack

The top layer in the stack contains the UPnP vendor specific information, which is supplemented with protocols defined by the UPnP forum. As messages traverse the stack they are formatted according to the GENA, SSDP, or SOAP specifications for transmission, using either a multicast or unicast HTTP format. All messages are transmitted using the IP transmission mechanism, which forms the foundation of the UPnP architecture communications model.

4.4.2.1 The UPnP Architecture Communications Model

The communications model associated with the UPnP architecture provides for the discovery, description, control, eventing, and presentation of devices residing on the network. However, before any communication can commence, an IP address must be obtained. The two methods specified to cater for this are either the Dynamic Host Configuration Protocol (DHCP), or an Auto-IP service. If a DHCP server exists, the DHCP client (the device) will be assigned an IP address. If, however, no DHCP server is present, the Auto-IP mechanism will be used. This mechanism selects a

unique address from a predetermined range of addresses and begins normal operations utilizing this address. The uniqueness of the IP address is determined through the use of an Address Resolution Protocol (ARP) request message containing the selected IP address, to which no responses should be received. Address resolution services will be discussed in chapter 5. The use of a Domain Name System (DNS), which maps a textual name to an IP address, is recommended to allow for the alteration of IP address without adversely affecting the operation of the UPnP services.

Once all devices on the network have obtained a valid IP address on which to operate, the discovery process may begin, which allows control points to find devices of interest. This process can be divided into the advertisement and searching components. The advertisement component includes the advertisement of service descriptions to all other devices on the network, which is initiated as devices join the network, or as instructed by a control point. These descriptions provide essential specifics about a device or service, and include a pointer to a location that contains more detailed information. It is this pointer that provides the information required for the successful operation of the description mechanism within the UPnP architecture. The searching component within the description mechanism allows control points to find nodes that are already operating on the network. This is achieved through the issuing of a request message by a control point, which can be targeted at a specific service or device type, all devices, or at an individual device. The specified target(s) will respond with an advertisement describing the requested facilities.

The description facilities provided within the UPnP architecture allow a control point to query advertised devices and services for more information, which prevents needless information from being transmitted over the network. This is possible, as included within the advertisements is a Uniform Resource Locator (URL), specifying the location that contains the required details. These details are enclosed within a well-formed extensible markup language (XML) document, for which a number of templates have been defined. These documents are referred to as device or service descriptions. Apart from these predefined formats, non-standard vendor extensions can be defined, although these would then be specific to a particular device or service type, as they are not defined within the UPnP architecture.

Once a control point has established the services that are available within a device, and has adequate information regarding those services, it can invoke actions associated with these services and query state variables for their values. This information is gathered during the discovery and description stages, as the supported actions and variables are embedded within the well-formed XML documents. Accompanying the variable descriptions is an eventing field, which specifies whether the value of a particular variable can be evented, where an evented variable is one that publishes an update should its value be altered. However, control points must subscribe with a controlled device in order to receive notifications or event messages about changes that occur to a particular variable.

The UPnP architecture provides a presentation facility, thereby allowing a control device to display an interface that is appropriate for the controlled device. These interfaces are described using the Hypertext Markup Language (HTML) format, commonly associated with the World Wide Web. These HTML documents are housed within the controlled device at a location provided within the description documents associated with the device. Control points can retrieve these documents from the specified URL, and render them in an appropriate application, such as a Web Browser.

As the UPnP architecture was developed primarily for use within home entertainment devices, no explicit mention is made of the management of connections, latency, or synchronization. However, these facilities could be included through the definition of appropriate services to provide this functionality. Chapter 6 provides an example implementation of a UPnP enabled professional audio device, elaborates further on the use of services, and reexamines the facilities supported by the UPnP architecture.

4.5 Summary

The sound installation technologies discussed in chapter 2 were revisited, with a view to utilising these technologies within the IEEE 1394 environment. This included a look at the various topologies and the modifications that would be necessary to provide IEEE 1394 compliance. The analysis of these technologies revealed a move towards IP-based transmission mechanisms.

A set of evaluation criteria was defined against which control and monitoring protocols could be considered. The protocol evaluation included IEEE 1394 based protocols that are currently utilised for the control and monitoring of audio devices, and IP-based protocols that could be utilised over the IEEE 1394 bus. The IEEE 1394 based protocol did not satisfy all of the evaluation criteria identified, while the IP-based protocols, such as QSC-24, faired better.

Chapter Five

An Implementation of QSC-24 above IEEE 1394

In order to cater for legacy protocols within the IEEE 1394 environment, a mechanism is required that will allow these protocols to be transmitted without any major alterations. The motivation behind the transmission of these legacy protocols is to extend the capabilities of IEEE 1394 professional audio nodes, by allowing the control of legacy audio equipment. A number of these control protocols are bound to IP networks, and often depend on an IP address for device identification and addressing. An example of such a legacy protocol is QSC-24. In order for these legacy protocols to be effective in the IEEE 1394 arena, these IP dependencies must remain intact. This can be achieved through the layering of an IP stack above the IEEE 1394 stack.

This chapter describes the implementation approach followed to allow the transmission of QSC-24 messages over the IEEE 1394 bus, through the provision of a suitable IP layer within the IEEE 1394 environment. This approach consisted of:

- The selection of a suitable development board,
- Firmware modifications to the development board, ensuring IP capability,
- The adaptation of an IP stack,
- The implementation of an IP-1394 device driver module to facilitate communication between the target firmware and the IP stack,
- Creating an application to allow a controller access to a legacy, QSC-24 enabled, device.

5.1 Implementation Environment

The development board utilized was a DHIVA (Digital Harmony Interface for Video and Audio), manufactured by the company Digital Harmony Technologies. This board, depicted graphically in figure 5.1, contains an ARM 7 processor, Philips pdil41

link layer with programmable buffer sizes, compatible physical layer, CIP processor, RAM and flash memory.

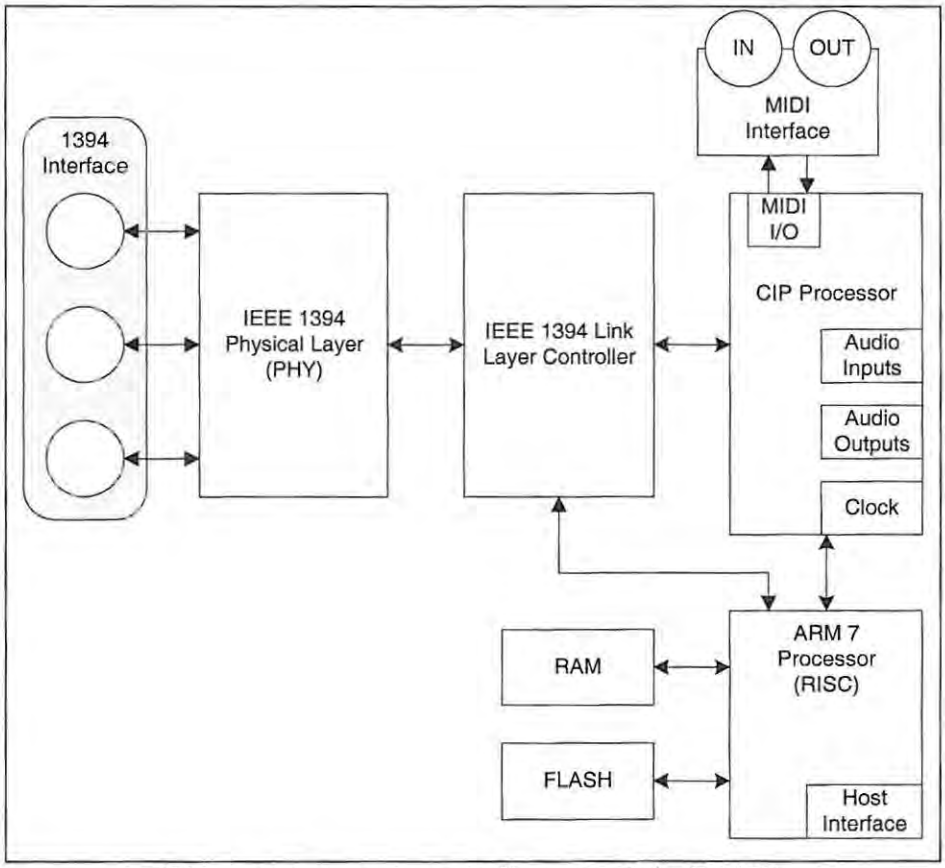

Figure 5.1: A component diagram of the DHIVA

The firmware for the DHIVA, stored within the flash memory, is compliant with IEEE 1394-1995 [IEEE Computer Society, 1996], as well as providing support for a number of 1394a [IEEE Computer Society, 2000] facilities. The capabilities of particular relevance to this study are:

- Configuration ROM in the general format,
- Relevant control and status registers (CSR),
- Support for asynchronous quadlet and block reads, writes, and locks,
- Support for the transmission and reception of data on a single isochronous channel,
- Support for asynchronous stream or loose isochronous transactions,
- Facilities to allow access to the host serial port.

The operating system present on the DHIVA is ThreadX [Express Logic, Inc., 2000], with the Green Hills MULTI application providing an easy-to-use integrated development environment (IDE) [Green Hills Software, Inc., 2001]. Included in the IDE, is the ability to halt execution at any point to examine the state of the device or threads, advanced debugging facilities, and control over the ARM 7 processor.

All firmware code is written in the programming language C, which is then compiled into a suitable image to be downloaded into the RAM on the DHIVA. This download procedure takes place via a parallel port on a PC, running any Microsoft Windows operating system, connected to the JTAG port on the DHIVA. Once the image has downloaded, the program execution can be controlled via the Green Hills IDE.

Also present on the DHIVA is an RS-232 serial debug port, which permits debug information regarding program execution to be displayed on a PC through a suitable terminal emulation application, as well as providing a command line interface (CLI) to allow access to a device's state, and parameter values, without the need to halt program execution. Examples of the use of the CLI facility include the ability to issue a bus reset, and access to topology and speed information of the connected nodes.

The host RS-232 serial port permits the application running on the DHIVA to issue or receive commands, thereby providing an external interface for interaction with other non-1394 devices. It is this facility that allows the DHIVA to function as a legacy adapter, and provide IEEE 1394 capabilities to other devices.

5.2 Approach

The approach adopted for the implementation of an IP stack on the DHIVA is based on RFC 2734 [Johannson, 1999], which describes the necessary message formats and device capability requirements in order for an IEEE 1394 node to be considered IP-capable. It was revealed that by following this approach, the final solution was compatible with the IP-1394 drivers provided by Microsoft Corporation in their Windows Millennium and XP operating systems.

The IP stack chosen is part of a suite of networking utilities, which was developed by the company Pacific Softworks [Pacific Softworks, Inc., 1999]. The stack is designed to be operating system and device independent, as is described in section 5.2.2.

5.2.1 RFC 2734

RFC 2734 specifies one approach to using the IEEE 1394 bus for the transport of Internet Protocol version 4 (IPv4) datagrams. This approach consists of packet formats, an address resolution protocol (1394 ARP[4]), and a multicast channel allocation protocol (MCAP).

There are a number of criteria that a node must satisfy before it can be classified as an IP capable node, which is a node that is physically capable of transporting IP datagrams [Johannson, 1999]:

- Configuration ROM in the general format [ISO/IEC, 1994], as well as the bus information block [IEEE Computer Society, 2000] is required. A unit directory, indicating a node's Internet Capabilities, must also be implemented.
- Support for the transmission and reception of asynchronous streams.
- The MAX_REC field within a nodes bus information block must contain a value no less than 8. This indicates that the node is capable of receiving asynchronous block write requests, transmitting block response packets, as well as transmitting and receiving asynchronous stream packets. These transactions must be capable of handling data payloads of at least 512 bytes.
- The node must be isochronous resource manager (IRM) capable.

5.2.1.1 Encapsulation

All IP datagrams, 1394 ARP requests and responses, and MCAP solicitations and advertisements that are transported over the 1394 bus, are encapsulated within a 1394 packets data payload. The maximum size of the payload is determined by the speed at which the packet is transmitted, as reflected in table 5.1, as well as by the MAX_REC field within the nodes bus information block. For example, a node that is capable of

[4] The acronym "1394 ARP" refers specifically to the address resolution that is specified within RFC 2734, whilst the acronym "ARP" refers to the address resolution that is usually associated with Ethernet.

S200 transmissions supports data payloads of 1024 and 2048 bytes for asynchronous and isochronous transactions, respectively.

Speed	Asynchronous	Isochronous
S100	512	1024
S200	1024	2048
S400	2048	4096
S800	4096	8192
S1600	8192	16384
S3200	16384	32768

Table 5.1 – Maximum Data Payloads

As a result of these restrictions, it can happen that the payload size is less than the default maximum transmission unit (MTU), associated with IP datagrams, of 1500 bytes. To cater for this scenario an encapsulation method, allowing for the fragmentation and re-assembly of IP datagrams at the 1394 link-level, is defined [Johannson, 1999]. Further elaboration of these fragmentation procedures is provided in section 5.2.1.3.

5.2.1.2 1394 Address Resolution

Before IP communication can begin, the IP address to which a message is targeted needs to be matched-up with a corresponding hardware address, this process is known as address resolution. The hardware address is required so that the transmission technology can target individual nodes located on the network. This address resolution service is particularly pertinent to IEEE 1394 nodes, as bus resets can alter the identification number (*node_ID*) associated with a node. It is for this reason that the GUID is utilized for address resolution instead of the *node_ID*, as the GUID for a particular node can not be altered between bus resets. This allows IP communication to continue after a bus reset, without the need to repeat the address resolution procedures.

In general, ARP requests are typically broadcast messages received by all nodes on the network, but is only targeted at the node that is utilizing the IP address contained within the request packet. The targeted node replies with an ARP response packet, containing its hardware address, which permits point-to-point IP communications to

proceed. Figure 5.2 details the packet format of the address resolution associated within Ethernet networks [Comer, 1987].

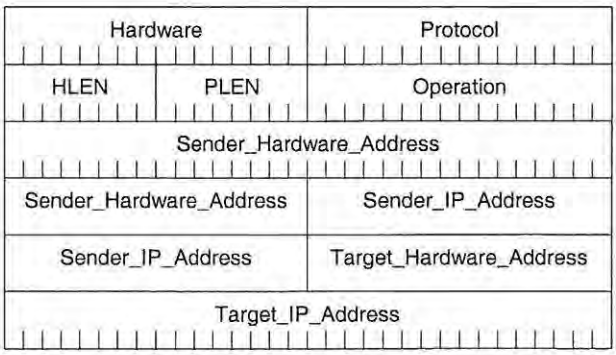

Figure 5.2: ARP packet format associated with Ethernet

This figure illustrates the relation between an IP and a hardware address. When an ARP request message is transmitted the *Target_Hardware_Address* field is the only field that does not contain a valid entry, as this is the value being discovered. The node that is utilizing the IP address denoted by the field *Target_IP_Address*, responds to the ARP request message with all the fields containing the appropriate values. Once the requesting node has received the hardware address via the response packet, IP communication can commence. It must be noted that address resolution is not a requirement for IP communication, but forms part of the link-level communication model associated with a particular transmission technology.

1394 ARP follows this same address resolution model, with the hardware address consisting of a node's 64-bit globally unique identifier (GUID), as specified within that node's bus information block. 1394 ARP requests are sent in the form of broadcast asynchronous stream packets, which are discussed in section 3.2.5. The ARP response to the request can be sent using an asynchronous stream, or through an asynchronous block write message to the requesting node. 1394 ARP packets are 32-bytes long and include the fields, depicted in figure 5.3.

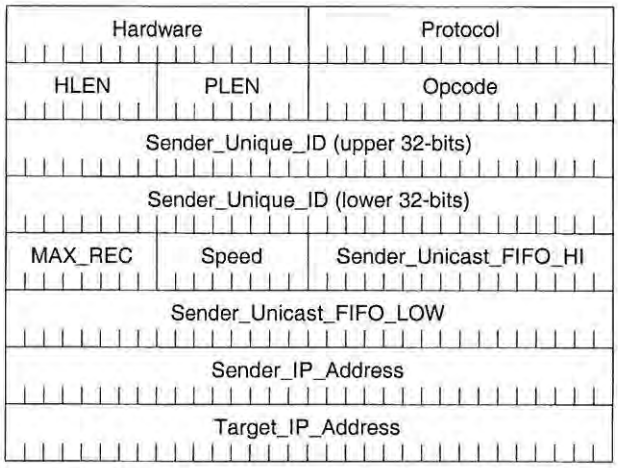

Figure 5.3: 1394 ARP request/response packet format

The meaning of each field is as follows:

- *Hardware*: This field denotes that the packet is 1394 based, and will contain the value 0018h.
- *Protocol*: This field contains the value 0800h, indicating that the protocol addresses contained within the 1394 ARP packet is in IP format.
- *HLEN*: The Hardware Address Length field contains the length, in bytes, of the hardware address associated with an IP address, which according to RFC 2734 is the value 16 [Johannson, 1999].
- *PLEN*: Contained within this field is the value 4, which is the length in bytes of an IP address.
- *Opcode*: The value 1 in this field indicates a 1394 ARP request, while a 2 indicates an ARP response.
- *Sender_Unique_ID*: This field contains the senders GUID, as specified in its bus information block.
- Sender *MAX_REC*: The sender *MAX_REC* field is populated from the MAX_REC field from within the sending nodes configuration ROM.
- Sender *Speed*: This field contains a value indicating the maximum speed at which the sender can operate. This maximum value is determined by taken the lesser of the link and PHY speeds. Possible values are provided in table 5.2. For example, a value of 2 indicates that the sending node is capable of communicating at a speed of S400.

Value	Speed
0	S100
1	S200
2	S400
3	S800
4	S1600
5	S3200

Table 5.2: Defined speed codes

- *Sender_Unicast_FIFO_HI* and *LOW*: These fields combined represent the 48-bit offset within the senders address space that has been set aside for the receipt of IP datagrams. The position of this offset within the address space can only be modified after a power reset, and shall remain constant through bus resets.
- *Sender_IP_Address*: The Sender IP Address field contains the IP address of the sending node.
- *Target_IP_Address*: This field contains the IP address for which the corresponding hardware address is being sought. The node that is using this address will respond with a 1394 ARP response packet.

Once a node initiating communication has received the 1394 ARP response message, communication between those nodes can begin on a point-to-point basis utilizing the discovered hardware address.

5.2.1.3 IP Datagrams

All IP datagrams sent over the IEEE 1394 bus are encapsulated within one of three headers. These different headers allows for the fragmentation of IP datagrams at the 1394 link-level, and allow the reconstruction of the original IP datagram at the destination node. Fragmentation occurs when the IP datagrams are larger than the maximum data payload supported by the sender and all receivers. However, if the entire IP datagram can be sent in one asynchronous block write transaction there is no need for fragmentation, and figure 5.4 illustrates the message format associated with such a packet.

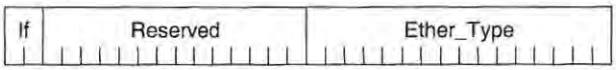

Figure 5.4: Encapsulation header for unfragmented IP datagram

The *lf* field denotes the level of link fragmentation that has occurred. If this packet is unfragmented, the *lf* field contains a value of 0, which can be read from the defined values in table 5.4. The *Ether_Type* field is used to show what type of message is being sent, of which the currently defined types are provided in figure 5.3.

Ether_Type	Datagram
0x0800	IPv4
0x0806	1394 ARP
0x8861	MCAP

Table 5.3: Ether_Type field definitions

lf	Position
0	Unfragmented
1	First Fragment
2	Last Fragment
3	Interior Fragment

Table 5.4: lf field definitions

If link fragmentation is required, two further message formats are required, as shown in figures 5.5 and 5.6.

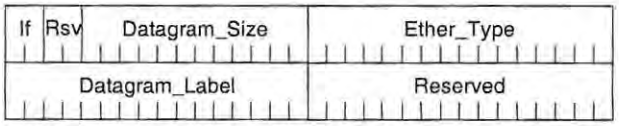

Figure 5.5: Encapsulation for the first fragment of an IP datagram

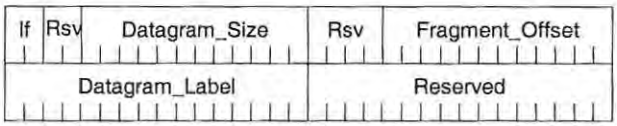

Figure 5.6: Encapsulation header for subsequent fragments of an IP datagram

If fragmentation is required, at least a first and last fragment must be sent. The first fragment will be encapsulated using the header detailed in figure 5.5. The *lf* field will contain the value 1, according to the values of table 5.4. The *Datagram_Size* field will contain the total length of the entire IP datagram. It will be the same for all link fragments, and one less than the value specified within the IP headers Total Length field. The *Ether_Type* field has the same meaning as in the unfragmented header format. The *Datagram_Label* field is used for the reassembly of IP datagrams, as all link fragments from one IP datagram shall have the same datagram label. When this label reaches a value of 65 535, it will wrap back to 0. The field labeled reserved may not be used.

All other link fragments follow the format given in figure 5.5, with the *lf* field either containing the value 2 or 3, indicating that this fragment is the last or an interior link fragment, respectively. The field definitions are the same as for the first link fragment except for the *Ether_Type* field which is replaced with the *Fragment_Offset* field. This field is used in the reconstruction of the IP datagram on the target node, and denotes the position of the fragment, in bytes, from the beginning of the IP datagram, including the IP header.

Reassembling the IP datagram is relatively simple, all fragments with the same sender *Source_ID* and *Datagram_Label* are placed at the offset provided by the *Fragment_Offset* field, in a reconstruction buffer. If a fragment is received that overlaps a previously received fragment or a bus reset occurs, the reassembly process is aborted and a fresh reassembly process started. Should a bus reset occur while a node has link fragments pending transmission, those fragments are to be discarded.

5.2.1.4 Configuration ROM

Configuration ROM entries are specified by RFC 2734 that all IP capable nodes must implement [Johannson, 1999]. These additions include Unit Spec ID and Unit SW Version entries, as specified by ISO/IEC 13213:1994 [ISO/IEC, 1994], and may also contain other permissible entries.

The Unit Spec ID entry specifies the organisation responsible for the definition of the Internet Capabilities of the device. It is an immediate entry in the unit directory and has the value 00 005Eh. The Unit SW Version is also an immediate entry in the unit directory, and together with the Unit Spec ID entry specifies the document that defines the software interface of the unit. The Unit SW Version entry must contain a value of 1, as depicted in figure 5.7.

Textual descriptors within the configuration ROM are optional, but when present provide more intelligible information about the capabilities of a device. RFC 2734 provides textual descriptors that are to be associated with the Unit Spec ID and Unit SW Version entries, and these values are "IANA" and "IPv4", respectively.

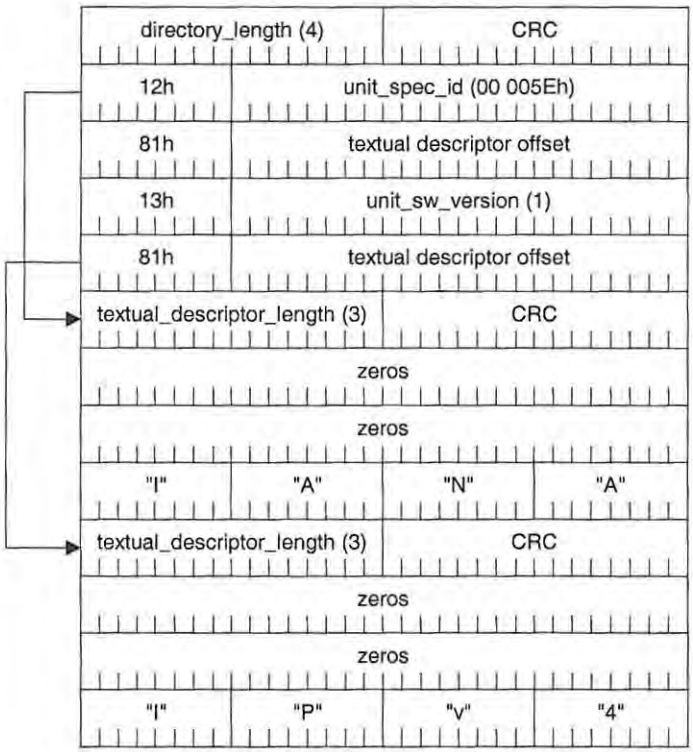

Figure 5.7: Unit directory and textual descriptors

5.2.1.5 Transmission Methods

The two suitable methods for the transmission of unicast IP messages are asynchronous block writes or isochronous stream packets. Asynchronous block writes provide link-level acknowledgement, but provide no quality of service. Isochronous stream packets provide quality of service, but cannot provide link-level acknowledgements. Isochronous streams packets consume a valuable resource, namely a channel number, which leaves asynchronous write messages being the most suitable transaction for IP unicast messages.

Before a unicast IP message can be sent, the maximum supported payload must be determined. This is obtained by analysing the speed map maintained by the IEEE 1394 bus manager, by analysing the Sender MAX_REC field within the 1394 ARP response packet, and taking into account the nodes own hardware restrictions.

IP broadcast messages lend themselves to the use of asynchronous stream messages. These messages offer no quality of service or link-level acknowledgement, which suits the nature of broadcast messages. Asynchronous stream messages do consume a channel number, which is specified within the BROADCAST_CHANNEL register of

the isochronous resource manager [IEEE Computer Society, 2000]. IP broadcast messages may only be sent when the valid bit within this BROADCAST_CHANNEL register is set to 1. When a bus reset occurs this bit is set to 0 and all IP broadcast messages cease until such time as the isochronous resource manager allocates a channel number.

Multicast messages are also transmitted via asynchronous stream messages through the use of a cooperative protocol known as the Multicast Channel Allocation Protocol (MCAP). This protocol is adopted by both IP multicast sources and recipients, whenever a channel number other than the broadcast or default channel is used. Multicast groups are advertised by the multicast channel owner which consists of a mapping of group addresses to the corresponding channel number. These advertisements are transmitted periodically or can be requested through the use of a solicitation message. Any further discussion is beyond the scope of this document, but more details can be found in RFC 2734 [Johannson, 1999].

5.2.2 FusionX

The Fusion Network Software can be described as "a collection of networking utilities and library functions designed to support data transfers and inter-process communication over a variety of operating systems" [Pacific Softworks, Inc., 1999]. The facilities of interest provided by the Fusion Network Software at the transport, network, and data link layers include:

- Transport Layers:
  - TCP and UDP
- Network / Internet Layer Protocols:
  - ARP
  - ICMP
  - IP
- Data Link / Network Interface Layer:
  - Ethernet

In order for the Fusion stack to be portable, there is a need for operating system and platform independency. This is achieved through the definition of various interfaces, depicted in figure 5.8, and these interfaces are:

- The Processor Interface
- The Scheduler Interface
- The Communications Device Interface
- The Socket Interface

The porting process involves the redefinition of these interfaces in accordance with the environment in which the IP stack is required to operate. The processor interface deals with issues such as the memory model, byte ordering (big or little-endian), and other issues related to the functioning of the processor.

In order for the IP stack to carry out more than one task simultaneously, some sort of scheduler is required. The services of the scheduler most needed, are the ability to halt the execution of a task, to restart the task, to enable / prevent task pre-emption, and the provision of a relative timing mechanism. TCP/IP makes extensive use of such a scheduler, as timeouts are set and messages placed in a wait state, until a response is received or a retransmission is necessary. The scheduler interface provides a mechanism whereby an already existing scheduler, or a rudimentary scheduler supplied with the IP stack, can be utilised. The approach taken within this study was to adapt these mechanisms to make use of the scheduler supplied with the ThreadX operating system, already present on the DHIVA development board.

The communications device interface provides a number of transmission mechanisms for the transport of datagrams. These mechanisms have traditionally been Ethernet, or some sort of serial communication (RS232) [Pacific Softworks, Inc., 1999]. Before the transmission of IP datagrams is possible, a number of facilities are required, these are the ability to:

- Initialise the device before sending or receiving data
- Bring the device in and out of service
- Modify the run-time behaviour of the device
- Allow the transmission of a frame of data

- Receive a frame of data from the device

These facilities are provided within a device driver specifically written for the appropriate platform and transmission technology being used. An example of which is the IP-1394 device driver written for the DHIVA development board, as is discussed in section 5.3.3.

A socket is the standard mechanism for communicating over the Internet, and is defined by the socket interface. A socket is similar to a file in nature, and supports the same operations such as write, read, open, and close. Possible types of sockets include UDP and TCP sockets. These sockets are bound to a particular port number, and are uniquely addressable through the use of the appropriate IP and port number.

Figure 5.8 is a simple illustration of the components of the Fusion stack and their interactions. An application communicates with the IP stack utilising the interface provided for the sockets. The socket call traverses the IP stack, building the necessary IP datagrams, and initiating the relevant timing mechanisms. The IP datagram is then framed in the device driver, through the device interface, according to the transmission technology employed, such as Ethernet. Once this is complete the encapsulated IP datagram is ready for transmission, and is passed to the relevant hardware. A similar process occurs, except in reverse, when an encapsulated IP datagram is received, the notification of which is typically achieved through the use of an interrupt service routine (ISR).

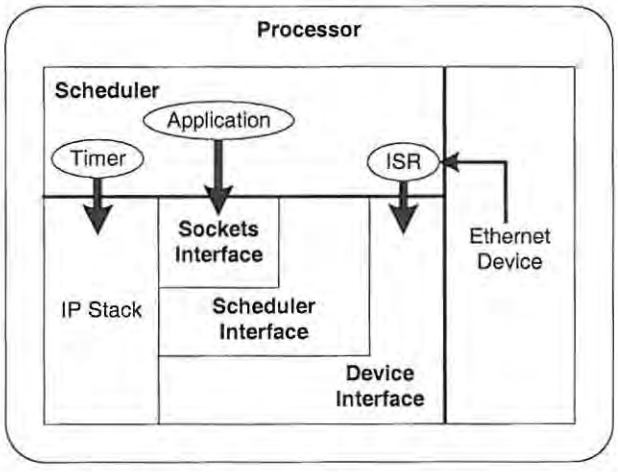

Figure 5.8: IP stack interface components

5.2.2.1 Communications Model

The purpose of the device driver is to bridge the gap between the Fusion stack and the target hardware. For example, the model illustrated in figure 5.9 allows the Fusion stack to communicate with the target firmware, which in turn controls the hardware.

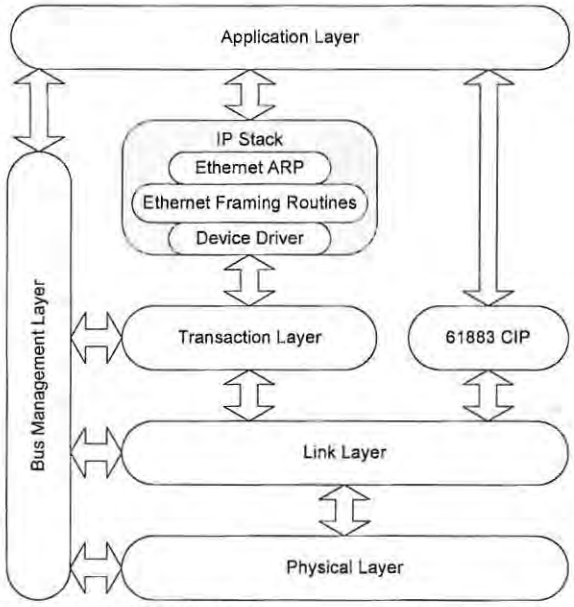

Figure 5.9: DHIVA communication model

The IP stack within the firmware resides above, and utilizes the transaction layer as illustrated. This transaction layer communicates with the link layer, which translates the software requests into hardware instructions. The IP stack is predominantly occupied by the activities of transmitting and receiving IP datagrams, as will be discussed in the sections that follow.

In addition, figure 5.9 illustrates that the IP stack includes an Ethernet ARP, Ethernet Framing Routines, as well as a device driver specific to the DHIVA development board. The reasoning behind the inclusion of the Ethernet ARP and framing routines is expanded in section 5.3. Note however, that although messages received from and passed to the IP stack are Ethernet packets. Ethernet packets are never transmitted over the IEEE 1394 bus, instead the Ethernet headers are stripped before transmission and the relevant RFC 2734 defined headers are appended. Similarly, upon the reception of an IP datagram from the IEEE 1394 bus, formatted according to RFC 2734, the IEEE 1394 specific headers will be stripped and the required Ethernet headers inserted before the packet is passed to the IP stack for processing. The

conversion of IP packets between Ethernet and the RFC 2734 format occurs within the device driver module shown in figure 5.9.

5.2.2.1.1 Transmission of IP related Messages

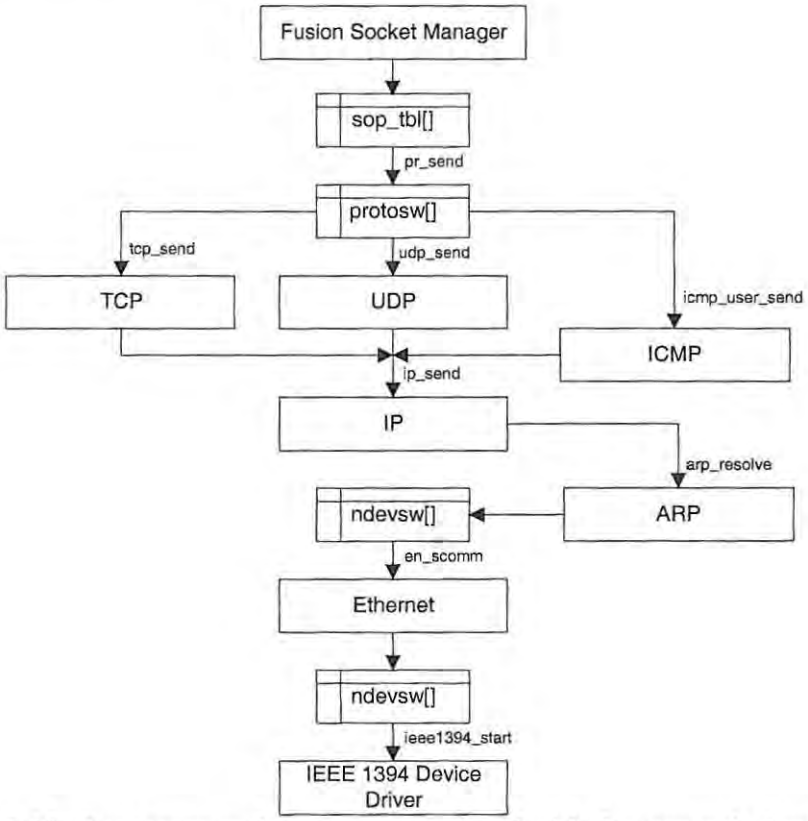

Figure 5.10: Processes and routines associated with the transmission of an IP datagram

The transmission of an IP datagram, illustrated in figure 5.10, is initiated by an application opening a socket, which can be achieved through the use of the udp_open() routine. This will return a socket handle to the application, create an entry in the socket table (sop_tbl[]), and associate the socket with a protocol entry in the protocol switch table (protosw[]). The socket handle provides a means for the application to reference the socket. The transmission of data through the socket, which can be achieved through the use of the sendto() routine illustrates the use of this handle.

The protocol switch table maintains references to the relevant routines associated with a particular protocol. For example, the routine that is to be called for the transmission

of a UDP packet would be the udp_send() routine. Typical entries in the protocol switch table include the UDP, TCP, and ICMP protocols.

The arp_resolve() routine checks the ARP cache for a corresponding hardware address, such as an Ethernet address, which if found is appended to the message structure. If the hardware address cannot be resolved, the message is put in a wait state, and the relevant ARP procedures carried out. These procedures consist of the generation and transmission of an appropriate request ARP message. If a response to this ARP request is received, the ARP cache is updated and the suspended message transmitted. If no ARP request is received within a predetermined time limit, the message will be discarded.

Contained within the network device table (ndevsw[]) are references to the relevant procedures for the selected link layer, such as the send, frame, and updown routines. Through the network device table the procedure responsible for the link level framing of the IP datagram is invoked. For the IEEE 1394 device driver, this would be the en_scomm() routine. At this point the message is ready for transmission and the send routine within the device driver, ieee1394_start(), is called via the network device table. This routine is responsible for the stripping of the Ethernet headers, the repacking of the message according to the frames defined within RFC 2734 [Johannson, 1999], and its transmission over the IEEE 1394 bus. The routines within the device driver are expanded upon later in this chapter.

5.2.2.1.2 Reception of IP related Messages

The two methods typically utilized for the transmission of IP datagrams and ARP requests are asynchronous block writes, and asynchronous stream transactions on the BROADCAST_CHANNEL, respectively. The reception of these messages by the IP stack, depicted in figure 5.11, is achieved through the registration, at initialization time, of a number of routines with the firmware present on the DHIVA development board. These routines provide the firmware with an entry point to the device driver and call the registered routine should one of the following occur:

- An asynchronous stream message is received on the BROADCAST_CHANNEL,

- A bus reset,
- An asynchronous write transaction targets the node address registered by the device driver with the DHIVA firmware.

At initialization time a new receive task is defined, which performs the processing needed on queued IP related packets, and is identified as the "Fusion1394Recv" task. This receive task is in a wait state while there are no packets to be processed, and is restarted by the ieee1394isr() function if packets are appended to the receive queue. The ieee1394isr() routine is the entry point to the device driver for all messages destined for the IP stack, and is called by the code present on the DHIVA development board. An added responsibility of the ieee1394isr() routine is to repackage the received datagrams into Ethernet frames to be passed up the stack, which is undertaken before the receive thread is awoken.

A receive queue is associated with the receive task, and contains all messages pending processing. As messages arrive they are appended to the receive queue, by the ieee1394_isr() routine. The receive thread will process these messages until the queue is empty, at which point it will return to a wait state, until it is restarted to process newly arrived packets.

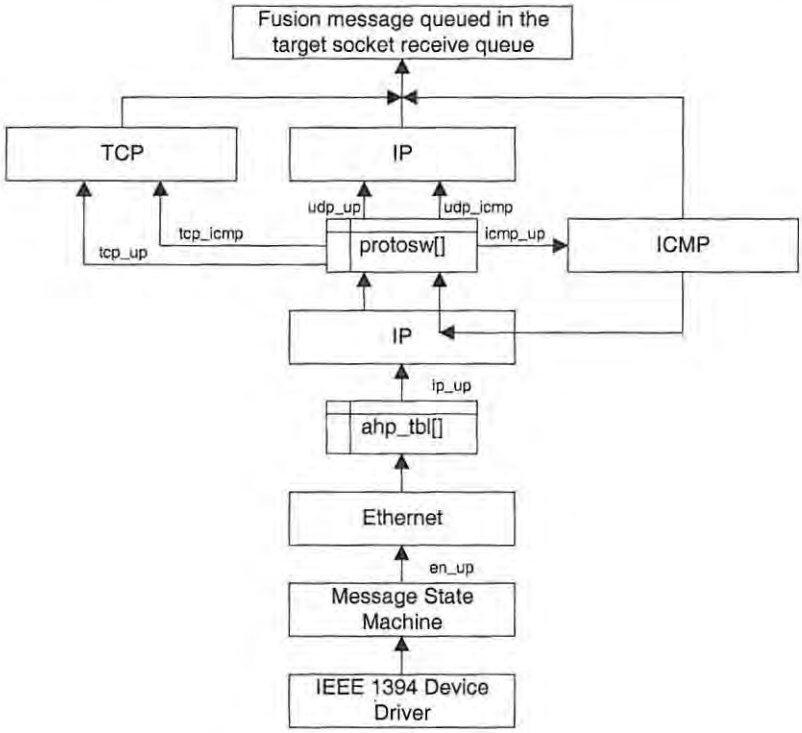

Figure 5.11: Processes and routines associated with the reception of an IP datagram

A packet that has been received by the device driver will be stripped of all 1394 headers, and formulated as an Ethernet packet, which is then placed in the receive queue for processing. This processing is under the control of the finite state machine within the IP stack. This eventually results in the en_up() routine being called which is responsible for the stripping of the Ethernet link-level headers.

The address header pointer table (ahp_tbl[]) is queried for the relevant higher-level protocol and the associated routine called, for example the ip_up() routine. If the message received were an address resolution message, the relevant ARP entry point routine would be extracted from the address header pointer table, and executed. This would result in the updating of the address resolution cache, and eventually the transmission of pending IP datagrams.

The appropriate upper level receive routine is then called via the protocol switch table. Each of the routines involved will strip their appropriate headers before forwarding the message to the next relevant routine. The raw data will eventually be queued at the targeted socket, from which the application can extract the message, and process it.

5.3 Porting Process

The integration of the Fusion networking stack into the development environment proved no trivial task. This was due to restrictions within the IP stack implementation, in particular the dependency of address resolution on a 48-bit hardware address. The initial approach was to provide a new link layer within the IP stack, as depicted in figure 5.12. However, the modification of the IP stack, due to the 48-bit dependencies, would have compromised the integrity of an already well-tested stack.

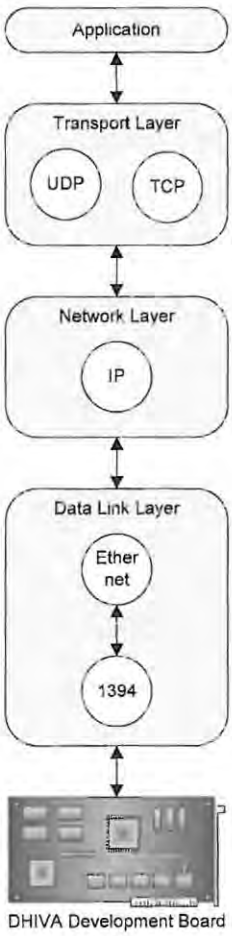

Figure 5.12: Layers within the Fusion networking software

Instead, the approach adopted was to leave the address resolution services intact, and to utilize an existing Ethernet link layer. The main advantages of this approach included maintaining the integrity of the Fusion network software, the modularization

of conversion routines within the device driver, and, as a future extension, the possibility of an IEEE 1394 to Ethernet Bridge.

The steps involved in the above implementation consisted of:

- Porting the operating system interfaces to suit the development environment,
- Editing of the configuration ROM and the implementation of the BROADCAST_CHANNEL register within the DHIVA's firmware,
- The definition of the appropriate data types and structures,
- The implementation of a device driver to allow for IP datagram encapsulation, fragmentation, transmission, and reception as is applicable to the IEEE 1394 bus.

5.3.1 Operating System Interfaces

The Fusion network software provides a number of operating system interfaces, which are essential for the successful operation of the IP stack. These interfaces take the form of functions that are defined to call the relevant operating system dependant routines. This allows the redefinition of these functions for the particular operating system utilized. For example, the function redefinitions that follow are specific to the ThreadX operating system present on the DHIVA development board. These functions include:

- os_critical(),
- os_normal(),
- os_sleep(), and
- os_wakeup().

The os_critical() function, illustrated in listing 5.1, is utilized to prevent preemption of a critical section of code by another task, while os_normal() re-enables preemption. The function, os_sleep(), instructs a task to suspend until a particular event occurs. The suspended task is then restored with a call to os_wakeup(). A number of other operating system dependant functions are also defined, such as error reporting and string handling routines, however they are non-essential and of little consequence to this study.

```
1  int os_critical(void)
   {
3      int status;
       status = tx_interrupt_control(TX_INT_DISABLE); // Disable interrupts
5      return (status);
   }
```

Listing 5.1: Definition of the os_critical() function

Listing 5.1 provides example code implementing the os_critical() function required by the Fusion stack. The function tx_interrupt_control(), called on line 4, is a ThreadX operating system specific call and instructs the operating system to disable interrupts, thereby avoiding task preemption. The remaining operating system interface functions are defined in a similar manner, and are provided in appendix B, section 1.

As previously mentioned, a timing mechanism for the IP stack is essential, and this is implemented by periodically calling the function t_clock(). This function updates the high-resolution timer variables within the IP stack, which are used for the determination of time out values, and the formation of retry messages. The code extract needed to implement the timing mechanism is provided in listing 5.2.

```
1  void TimerTask(DH_UINT32 thread_input)
   {
3      while(1)
       {  //Delay predetermined timer interval
5          os_delay(MS_PER_TICKS/OS_MS_PER_TICK);

7          if (FusionRunning) //Is fusion running?
               t_clock();
9      }
   }
```

Listing 5.2: Implementation of the required timing mechanism

The os_delay() call shown on line 5 delays the thread by a predetermined value which is specified by the argument passed to this function. The call to t_clock(), line 8, instructs the IP stack to update its variables, and perform the necessary processing.

The last operating system dependant aspect to be discussed is the allocation of memory, which is required for the IP stack data structures and sockets. The required

memory is allocated from a heap associated with the IP stack, which consists of either static, fixed, or permanently allocated memory. A static heap is allocated at compile time, with the memory under the management of the Fusion network software. The fixed heap approach specifies a fixed heap size at a known offset within memory, while the permanently allocated approach permits the Fusion network software to utilize operating system dependant routines to manage the allocation of memory.

The approach adopted in this study was the permanently allocated heap, in order to optimize the space required by, and allocated to the heap. The management of the heap is the responsibility of the IP stack, although a few helper routines were defined. These helper routines are responsible for the initialization, and allocation of the memory required for the heap. These helper routines are provided in section 3 of appendix B.

The size of the heap is specified by the constant HEAP_SIZE, which is calculated based on the following:

- 60 Kbytes for the Fusion network software data structures,
- and 4 – 8 Kbytes per open socket.

5.3.2 Firmware Updates

The requirements for a node to be considered IP capable, as is discussed in section 5.2.1, were not all present within the firmware on the development board. However, the hardware was capable of supporting all of these features, with only minor firmware updates. These updates included the modification of the configuration ROM as specified within RFC 2734 [Johannson, 1999], the implementation of the BROADCAST_CHANNEL register [IEEE Computer Society, 2000], enabling support for asynchronous streams, and providing the necessary values within the relevant control and status registers.

The only mandatory modification required to the configuration ROM was the addition of a unit directory, containing entries for *Unit_Spec_ID* and *Unit_SW_Version*. However, the optional textual descriptors were also implemented, as these provide meaningful textual information representing the capabilities of the device to the user.

The inclusion of the unit directory permits other nodes to ascertain the capabilities of the device and utilize its IP capabilities. For example, the IP over 1394 drivers within Microsoft Windows Millennium and XP will not be enabled until another IP-capable node is identified, through the unit directory entries.

The MAX_REC entry within the configuration ROM was updated to specify that the node was capable of transmitting and receiving 1024 bytes per asynchronous transaction, as a value of 512 bytes or greater is required for the transmission of encapsulated IP datagrams. This value is a determining factor when deciding on the level of link level fragmentation required for the transmission of IP datagrams over the IEEE 1394 bus [Johannson, 1999].

The BROADCAST_CHANNEL register is defined as illustrated in figure 5.13 [IEEE Computer Society, 2000]. The value 1 in the first bit indicates that the BROADCAST_CHANNEL register is implemented, and the *V* field indicates whether or not the *Channel* field is valid, with a valid channel being represented by the value 1. The *Channel* field refers to the isochronous channel, typically channel 31, that has been allocated for asynchronous broadcasts.

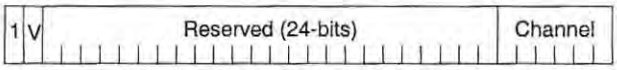

Figure 5.13: BROADCAST_CHANNEL format

The BROADCAST_CHANNEL register was defined within the firmware, and implemented at offset 234h from the base of the CSR register.

Although the hardware supported asynchronous streams transactions, the initial firmware did not. The addition of asynchronous stream capability involved modification to the receive routines within the firmware's link layer and the redefinition of hardware register sizes, which are reserved for the receipt of data, to support the 1024 byte packets defined within the configuration ROM. A later firmware release by Digital Harmony Technologies included the necessary support for asynchronous stream transactions, and was adopted as the base platform upon which to build.

The only other modification required was the inclusion of the IP stack at compilation time to produce a single image for the development board. This was achieved by building the IP stack and device driver into a library, which was then included into the firmware target build along with the required application code.

5.3.3 Device Driver Implementation

As the device driver provides a "glue" layer between the IP stack and the firmware present on the development board, it plays an important role in the successful functioning of the IP over 1394 implementation. This section details the functioning of the device driver, discusses the functions within the device driver, and illustrates the necessary table definitions within the IP stack required for installing and initializing the device driver.

5.3.3.1 Device Installation and Initialization

As described previously, the network device table (ndevsw[]) contains the entries necessary to allow the IP stack to communicate with the device driver. These entries are references to entry points, within the device driver, that have been exposed to allow for the transmission and reception of IP related messages, as well as the provision of device specific information and other miscellaneous device driver routines. Listing 5.3 provides the extract of the network device table that is relevant to the installation of the relevant device driver for IEEE 1394 networks.

```
{ "IP_DHT",     0,          0,          0,          0,
0,              0,          0,          0,
{AF_ETHER},     {0},        ieee1394_init,
ieee1394_updown, en_scomm,  ieee1394_start, ieee1394_ioctl
},
```

Listing 5.3: Network device table entry associated with IEEE 1394 networks

The fields within listing 5.3 which have been assigned the value 0 are typically parameter values that assist in the communication process between the IP stack and the device driver, but are not utilized in this implementation. The *{0}* field denotes a null reference for a possible routine call associated with the reporting of statistical information. The fields highlighted are the fields of relevance. The entry *"IP_DHT"* is a meaningful name assigned to the device driver, and proves useful in the reporting of debugging information. The *{AF_ETHER}* entry specifies the link layer that is

associated with the device, which is Ethernet. Ideally this entry should contain a reference to a link layer specifically designed for IEEE 1394 networks, but for reasons discussed in section 5.3 this is not possible. The remaining entries all specify references to routines which are all, with the exception of the en_scomm() routine, present within the device driver. The code for the entire device driver is included in appendix B, section 4.

The routine responsible for the initialization of the device driver is ieee1394_init(), which takes as an argument a pointer to the relevant network device. The first task within this routine is the validation of this pointer, after which it is assigned to a global variable for future reference. The number of large and small packets are also set, before the fragmentation buffers are initialized to their default values. The fragmentation buffers will be discussed, in detail, later in this chapter.

The ieee1394_updown() routine is used to bring the device driver into and out of operation, through the use of a flag argument indicating the desired operation. A device brought into operation consists of the installation of an Ethernet hardware address, the assignment of link layer type and size within the network device, and the allocation of the necessary buffers. If the device is being taken out of operation, the allocated buffers are released and the receive task is terminated.

The en_scomm() routine resides within the provided Ethernet support routines, and is responsible for the encapsulation of an IP datagram into the relevant Ethernet frame. The ieee1394_start() routine, can be considered as the required transmission mechanism, while the en_scomm() routine handles the Ethernet framing. The ieee1394_start routine(), provided in the device driver, instructs the development board's firmware to transmit a packet. However, as the packet presented to the start routine is framed with an Ethernet header and not the required RFC 2734 headers, additional processing is needed. This processing occurs within the ether_to_1394() method.

The ieee1394_ioctl() routine is intended to perform input and output control on the device, such as switching of the input mode between promiscuous and normal modes.

Due to the prototype nature of the device driver the skeleton of this routine is implemented although no real functionality is included.

5.3.3.2 Data Structures Associated with the IP-1394 Module

A number of data structures are required for the successful operation of the device driver, as they facilitate the formation of headers required for framing, and permit the typecasting of messages into their relevant formats. These data structures include the following:

- Asynchronous stream supplement header,
- 1394 ARP header,
- Fragmentation headers
- Ethernet link-level header, and
- Ethernet ARP header.

The types utilized within these structure declarations consist of a letter followed by a number, as is illustrated in listing 5.4. The letter "a" represents a packed structure, for example a64 denotes a structure consisting of 64-bits, which is declared as an array of eight u8 variables. The "u" indicates that the variable is unsigned, and the "8" indicates the length of the byte, in bits. Therefore the u8 declarations are essentially unsigned characters declarations, which can be redefined according to the target platform and compiler.

```
typedef PACK_STRUCTURE struct a16 {      /* a 16-bit byte-ordered structure */
        u8  a2[2];
} a16;
typedef PACK_STRUCTURE struct a32 {      /* a 32-bit byte-ordered structure */
        u8  a4[4];
} a32;
typedef PACK_STRUCTURE struct a48 {      /* a 48-bit byte-ordered structure */
        u8  a6[6];
} a48;
typedef PACK_STRUCTURE struct a64 {      /* a 64-bit byte-ordered structure */
        u8  a8[8];
} a64;
```

Listing 5.4: Declaration of the types utilized

The asynchronous stream supplement header, defined within RFC 2734, specifies *source_ID*, *specifier_ID*, and version values. The *source_ID* contains the transmitting nodes identity number, while the *specifier_ID* field contains the value 00 005Eh, as assigned by the IEEE Registration Authority and specified within the Configuration ROM of IP-enabled devices. Lastly, the version field shall contain the value 1. This supplemental header is to be appended to every asynchronous stream packet transmitting an IP related message, as illustrated in figure 5.14. The structure declaration is provided in listing 5.5.

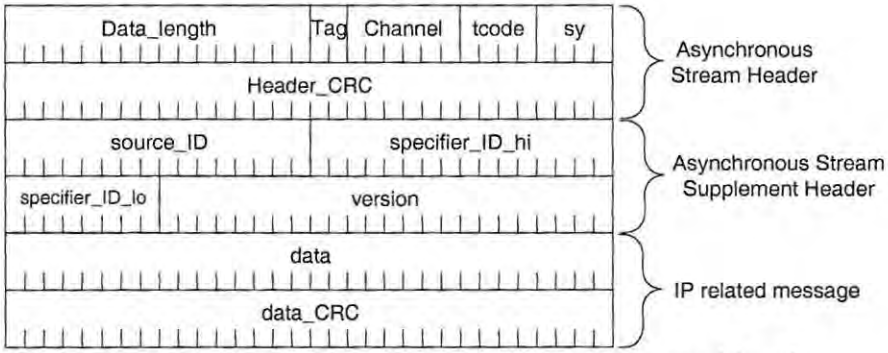

Figure 5.14: Asynchronous stream packet supplemental header

```
/* Asynchronous stream supplement header declaration*/
typedef struct {
        a16 source_id;          /* Source ID */
        a16 spec_id_hi;         /* Spec ID Hi*/
        u8  spec_id_lo;         /* Spec ID Low */
        a16 version_waste;      /* Version field that's not used */
        u8  version;            /* Version field that's used */
} gasp_pre;
```

Listing 5.5: Asynchronous stream supplemental header structure declaration

The Ethernet link-level header, illustrated in figure 5.15, is utilized to format messages appropriate to the link-level within the IP stack, and to extract fields from Ethernet messages framed by the stack. It contains fields for the destination and source Ethernet addresses, as well as a type field, which specifies the format of the encapsulated data. The structure declaration utilized within the implementation is provided in listing 5.6.

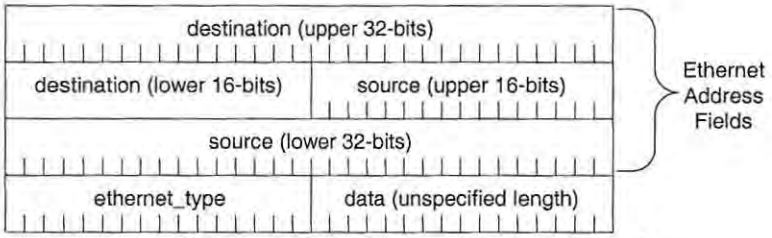

Figure 5.15: Ethernet link-level header

```
/* Format of an ethernet link-level message */
typedef struct {
        a48     ll_daddr;       /* Destination hardware address */
        a48     ll_saddr;       /* Source hardware address */
        a16     ll_type;        /* Type of data to follow */
} en_ll_h;
```

Listing 5.6: Ethernet link-level structure

The packet formats for the 1394 ARP, fragmentation, and Ethernet ARP headers have all been provided in figures 5.3, 5.4 to 5.6, and 5.2, respectively. The data structures associated with the headers required for the successful operation of the device driver, are provided in appendix B, section 4.

The final relevant declaration to be discussed is that of the lookup table to map between IEEE 1394 nodes and Ethernet addresses. Such a table is required by the IP-1394 device driver to facilitate the conversion between Ethernet and RFC 2734 formatted packets. The lookup table is populated by 1394 ARP messages received by the device driver and follows the structure declaration presented in listing 5.7. This declaration consists of two parts, with the first declaring a data structure specifying the format of an entry within the table, and the latter declaring the table.

```
/* Format of the Look up Table to map 1394 to Ethernet */
typedef struct {
        a64 eui;        /* Unique ID                    */
        a16 dna;        /* Destination Node Address     */
        a48 fifo;       /* FIFO Address within dna      */
        u8  mrec;       /* Max Rec                      */
        u8  sspd;       /* Speed                        */
        a48 ena;        /* Ethernet Address             */
} ieee1394_t;

export ieee1394_t ieee1394_tbl[63]; /* Table declaration /*
```

Listing 5.7: Declaration associated with the lookup table

The first field declared is the Extended Unique Identity (*eui*) field, which is a 64-bit value that is unique across all IEEE 1394 devices and contains the GUID of a device. As this field is guaranteed to be unique, it serves as the hardware address within 1394 ARP procedures [Johannson, 1999]. The *dna* field specifies the destination node address (*node_ID*) to which IP related messages are to be sent. In other words the *dna* field contains the *node_ID* of the node whose GUID is specified within the *eui* field. The *fifo* field is a 48-bit field and specifies the offset within the targeted nodes address space to which asynchronous block write messages carrying IP related datagrams, are destined. In order to determine the maximum payload that a targeted node supports, the *mrec* field contains the value specifying its reception capabilities. Similarly, the *sspd* field specifies the maximum speed that the targeted node supports. Finally, the *ena* field specifies the Ethernet address that has been allocated to the targeted node, which is utilized in the conversion from RFC 2734 frames to Ethernet link-level frames. The formation of this Ethernet address is based on the *node_ID* of the targeted node. All entries within the lookup table are indexed according to a targeted nodes physical_*ID*, which guarantees a unique index within the lookup table for each individual node on the local bus.

The last line in listing 5.7 declares the lookup table as *ieee1394_tbl*, and consists of a maximum of 63 entries which corresponds to the maximum number of physical nodes on the local bus. The relevance of these data structure declarations and header formats is discussed in the following section.

5.3.3.3 Transmission and Reception within the Device Driver

A brief overview of the transmission and reception of packets associated with the Fusion IP stack has already been provided. This section, however, describes the inner workings of the device driver, with a particular focus on the IEEE 1394-to-Ethernet mapping mechanism. Appendix C provides two informal flowcharts that illustrate the flow of data through the device driver with regard to the transmission and reception of packet, and provides an overview of the processing logic within the driver.

Assume that there are only two IP capable nodes on the IEEE 1394 bus, each running the ported IP stack, and no IP related communication has yet taken place. One of these nodes, the requester, has been instructed to transmit an IP datagram to the other

node, the responder. However, before this IP datagram can be transmitted the address resolution procedures must be followed to allow the requester to target the responder through the use of a hardware address. Figure 5.16 provides a communication diagram illustrating the transactions that will occur over the IEEE 1394 bus to allow the transmission of the IP datagram.

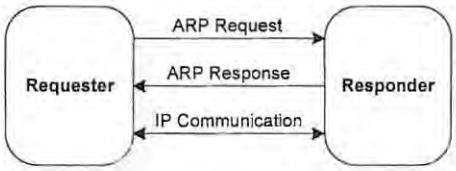

Figure 5.16: Requester and responder communication diagram

The first task is for the requesting node to ascertain whether or not the ARP cache contains a valid entry for the targeted node. As no IP messages have traversed the IEEE 1394 bus yet, there will be no such entry. The IP stack within the transmitting node will form an Ethernet ARP request packet framed by an Ethernet link-level header, as illustrated in figure 5.17. The entire packet is then passed to the device driver for transmission, by calling the ieee1394_start() routine.

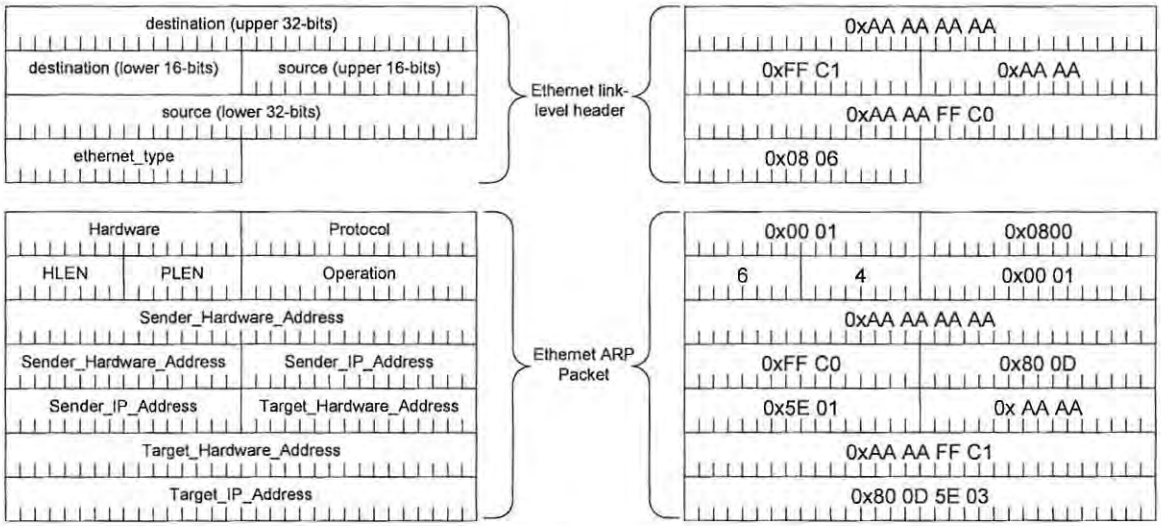

Figure 5.17: Ethernet packet passed to the IP-1394 device driver, with sample data

Associated with the packet structure is a flag field, which will indicate if the packet is a broadcast packet. This flag field is set within the IP stack, and allows the device driver to determine the appropriate transmission mechanism. As all ARP

transmissions are broadcast the ether_to_1394() routine is invoked, with a reference to the packet being passed as an argument.

The purpose of the ether_to_1394() routine, in this instance, is to convert the Ethernet link-level and ARP packet to an RFC 2734 compliant 1394 ARP packet for transmission, framed with the appropriate unfragmented and asynchronous stream supplement headers. This is illustrated in figure 5.18. In order to determine the packet type, the packet, as it was received from the IP stack, is cast using the defined Ethernet link-level structure, which allows easy access to the fields within the header. The *Ethernet_type* field within the link-level header is inspected, which will contain the value 0806h, thereby indicating that the encapsulated message is an ARP message. The remaining portion of the message is typecast to the defined Ethernet ARP structure for the formation of a 1394 ARP request message.

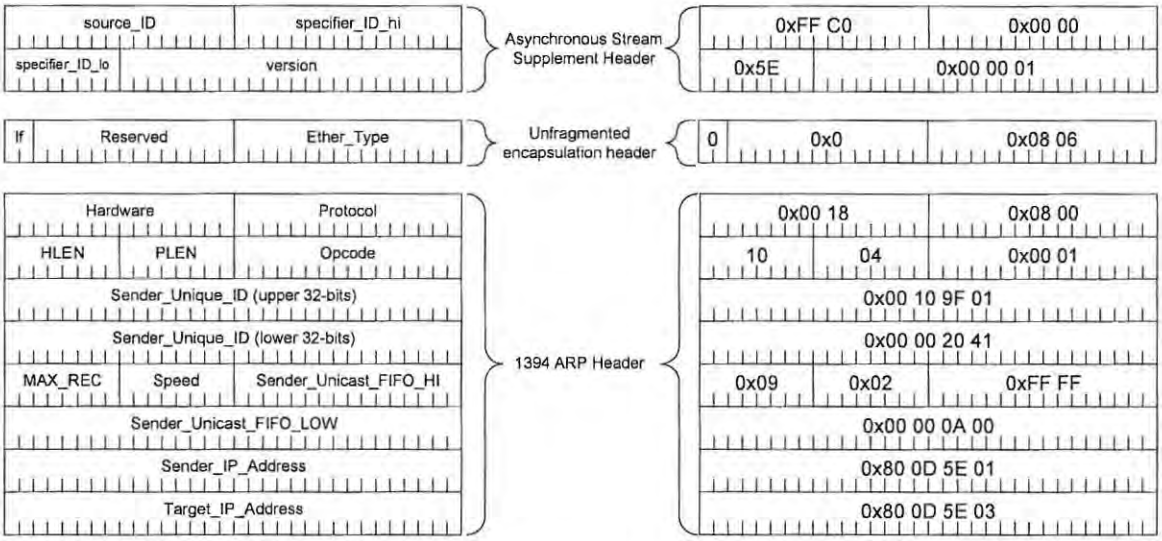

Figure 5.18: 1394 ARP message including the appropriate headers, and sample data

An instance of a 1394 ARP message is declared and the hardware type, protocol type, and hardware length fields are set to contain the values specified within RFC 2734 [Johannson, 1999]. The protocol length, opcode, target IP address, and sender IP address are copied from the Ethernet ARP structure into the 1394 ARP structure. The remaining fields serve to specify the capabilities of the transmitting node, and are updated from the lookup table (ieee1394_tbl). This is possible as the lookup table is

updated to contain the capabilities of the local node after every bus reset, and upon node initialization.

Before the 1394 ARP request can be transmitted it must be encapsulated within an unfragmented datagram and asynchronous stream supplemental header, both of which are shown in figure 5.18. The fields within these headers are set to the values defined within RFC 2734, and the packet is ready for transmission. The transmission happens via a call to the Send1394Packet() routine, which calls the appropriate routines for both broadcast and unicast transactions. At this point the packet leaves the device driver through the IPSendAsyncStream() routine, which rearranges the message according to the quadlet structure required within the firmware, and the message is transmitted over the IEEE 1394 bus.

The firmware within the responder node will receive the asynchronous stream packet, and call the registered callback routine, namely the IPReceiveAsyncStream() routine. This routine validates the payload before calling the ieee1394_isr() entry routine within the device driver, passing the payload length in quads, payload, and a *node_ID* value of 0, indicating an asynchronous stream transaction. The first essential task within the device driver is to format the data received from quadlet order into the byte order expected by the IP stack, and to calculate the payload length in bytes. The next logical step would be to queue the data for the receive task to process, but as the IP stack is expecting Ethernet and not RFC 2734 framed data, a conversion routine IE1394_to_ether() is called. The arguments passed to this routine consist of the reformatted payload, payload length in bytes, and the *node_ID* field containing the value 0.

The IE1394_to_ether() routine first determines whether or not the message was received via a broadcast mechanism by analyzing the *node_ID* argument. As it contains the value 0 which is an invalid *node_ID*, a broadcast packet is assumed. The asynchronous stream supplement header is stripped off, and the values checked for consistency. If one of the fields within this header contains a value other than those specified, the message is discarded.

Next, the level of fragmentation is inspected which reveals that an unfragmented packet has been received. The type field within the unfragmented frame is then examined, as this determines whether an IP packet, or a 1394 ARP packet has been received. A value of 0806h reveals that a 1394 ARP message is to be processed. The payload is typecast into a 1394 ARP structure to allow access to the fields within and a new entry for the lookup table declared. All of the fields within this lookup table entry, with the exception of an Ethernet address, are updated from the 1394 ARP message structure. The Ethernet address is formed by the combination of arbitrary values, which are concatenated with the requesting nodes *physical_ID*, which is extracted from the *source_ID* field of the asynchronous stream supplemental header. For example, in figure 5.17 the *source_ID* field contains the value of FFC0h, where FFCh is the *bus_ID* and 0h is the *physical_ID*. The table entry is then added to the lookup table, at the position specified by the *physical_ID* field. Note that at this point there are two entries within the lookup table, consisting of one for the local node and the other for the requesting node.

The next step is to reformat the ARP message as an Ethernet link-level packet that encapsulates an Ethernet ARP message, and return the new length of the packet to the calling function. All of the fields within the Ethernet packet can be extracted from the 1394 ARP packet, and the lookup table entries. The reformatted message is then appended to the queue associated with the receive task, and the thread invoked to begin the processing required within the IP stack. The reformatted message will resemble the original Ethernet message depicted in figure 5.17.

As this message is an ARP request targeted at the responders IP address, an ARP response will be formed, and transmitted via a call to the ieee1394_start() routine. The ARP response follows a similar path to the ARP request packet, which has been discussed. However, as this message is no longer a broadcast message it will be transmitted via an asynchronous block write message to the offset within the target address space, as was obtained from the 1394 ARP request packet, and stored within the lookup table. Recalling, that the index to the lookup table is via the *physical_ID* value, it must be determined from the Ethernet link-level or ARP packet. This highlights the choice of the *node_ID* value as the basis for the virtual Ethernet address, as it can be easily extracted from the link-level header. Once the *physical_ID*

has been determined from the *node_ID*, the relevant entry in the lookup table is accessed and the address offset within the requesting node is obtained. The same reformatting procedure takes place as described with the ARP request procedure, and the reformatted message is transmitted via an asynchronous block write message to the requesting node via the IPSendAsyncWrite() routine. The ARP response packet including the asynchronous block write header is illustrated in figure 5.19.

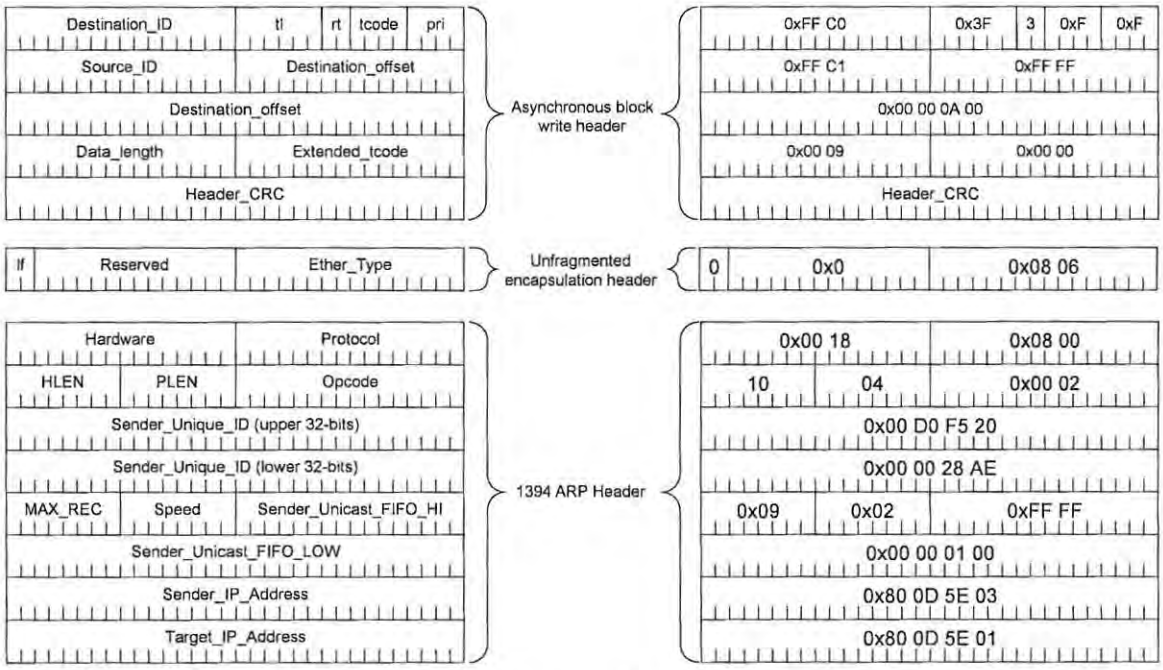

Figure 5.19: ARP response packet with asynchronous block write header, and sample data

The ARP procedure is now complete, and illustrates the transmission and reception facilities present within the device driver. It is these same facilities that are utilized for the transfer of IP datagrams. However, IP datagrams might need to be fragmented at the 1394 link-level, as is described in the following section.

5.3.3.4 IP Datagram Fragmentation and Encapsulation

Fragmentation occurs when the length of an IP datagram and its headers exceeds either the size of the transmission buffers on the transmitting node, or the receive buffer on the targeted node. These buffer sizes are represented by the values contained within the MAX_REC field of the relevant nodes bus information block. However, to facilitate the fragmentation mechanisms required when transmitting IP datagrams, the MAX_REC field is also transmitted within 1394 ARP request and

response messages. The lookup table, associated with the ported IP stack, stores these capability values, allowing quick and easy access to them. These values are used in the determination of the fragmentation level required.

The fragmentation and encapsulation of IP datagrams, destined for transmission over the IEEE 1394 bus, is a relatively trivial process that is carried out by the ether_to_1394() routine. This routine first verifies that the packet to be transmitted is indeed an IP datagram, after which is determines the limiting first-in first-out (FIFO) buffer size. This is done by comparing the MAX_REC value of the destination node, stored within the lookup table, with the MAX_REC value of the local device. The smaller of the two values is taken to be the limiting factor, and is stored in a variable named *mrec*.

The length of the IP datagram permitted within a fragment, is smaller than the limiting MAX_REC value, as the length of the headers need to be catered for. This includes not only those specified within RFC 2734, but also those defined for the encapsulation of messages over the IEEE 1394 bus, such as the header associated with asynchronous block write transactions.

Once the limiting MAX_REC field has been determined, it can be compared to the length of the IP datagram to ascertain if fragmentation is required. If the length of the IP datagram, with the addition of the appropriate transmission headers, is less than the value specified within the limiting MAX_REC field, then no fragmentation is required. In this case an unfragmented packet structure is used, as was the case for the 1394 ARP messages. If, however, the length of the IP datagrams and the required headers exceeds the MAX_REC field, than some level of fragmentation is necessary. The number of packets of length equal to *mrec* less header size determines the level of fragmentation required for the transmission of the entire IP datagram.

The fragmentation procedures within the device driver simply format the first encapsulation header, after which it is transmitted. The length of the IP datagram remaining is then compared to the size of the value within the *mrec* variable. If the remaining IP datagram length is still greater than *mrec*, an inner link fragment is transmitted. This process of transmitting interior fragments occurs until the length of

the IP datagram remaining is less than the value within *mrec*, which indicates that the last fragment may be sent.

The reception of IP fragments is more complex and requires the use of fragmentation reassembly buffers, which is a portion of memory dedicated to the reconstruction of IP datagrams. These buffers are defined by the FRAG_STRUCT structure, depicted in listing 5.8.

```
typedef struct {
        DH_UINT16 snodeid;          /* Node ID */
        DH_UINT32 dgl;              /* Datagram Label */
        DH_UINT32 ttl;              /* Time to Live */
        DH_UINT32 IPLen;            /* Length of the IP packet */
        DH_UINT32 RecLen;           /* Length of the packet we have received */
        DH_BYTE FragBuf[2048];      /* Buffer for Reassembly*/
} FRAG_STRUCT;
```

Listing 5.8: Fragmentation buffer structure

The *snodeid* field contains the *node_ID* of the transmitting node. The length, in bytes, of the entire datagram is stored in the *IPLen* field, while the *RecLen* field is used to keep track of the number of bytes of the IP datagram received so far. The time to live (*ttl*) field is used in the recycling of fragmentation buffers, as it provides a mechanism whereby the fragments within the buffer may be discarded should the entire IP datagram not arrive within a specified period. The *FragBuf* field is the actual data buffer. In order to permit the successful reconstruction of fragmented IP datagrams, all IP fragments need to be identified as belonging to the same fragmented sequence, which is achieved through the use of the datagram label (*dgl*) and *snodeid* fields.

The reassembly of IP fragments occurs within the IE1394_to_ether() device driver routine, as the entire IP datagram must be reconstructed before it can be processed by the IP stack. The reconstruction process is based on the premise that for all encapsulation headers, the first 2-bits indicate the fragmentation level of the encapsulated datagram. This allows for a simple switch statement with catch cases for each level of fragmentation.

The simplest case is if an unfragmented packet such as a 1394 ARP packet is detected, as the IEEE 1394 headers are stripped and the Ethernet link-level header inserted. If, however, the first link-level fragment of a disseminated datagram arrives, denoted by the value 1 within the *lf* field, a fragmentation buffer must be allocated, and the IP datagram fragment stored within it. The allocation of a buffer occurs within the GetFragBufferIndex() routine, which is passed the datagram label, *node_ID*, and length of the IP packet as arguments, all of which are extracted from the associated headers. This procedure searches the currently allocated buffers checking that a buffer is not already allocated for the reconstruction of this datagram. If a buffer was previously allocated for this entry, its index is returned, however such an entry is unlikely to exist, as this is the first link fragment, and an index to the first available buffer is returned. Incidentally, as the buffers are searched sequentially their *ttl* fields are incremented, until a specified ceiling value is reached at which point they are reinitialized, and flagged as unused.

The index returned is used to access the allocated buffer, and initialize all the fields therein to the values specified within the headers framing the packet. Before the data is copied into the buffer, the relevant Ethernet link-level header is generated and placed into the buffer, after which the payload is inserted. The length of data received is provided in the *datagram_size* field of the encapsulation header, and is used to update the *RecLen* field within the allocated buffer.

The processing of the interior and last fragments occurs in a similar fashion to the first, with the exception that the copying of the payload into the fragmentation buffer is instructed to start at the value specified by the *Frag_Offset* field within the encapsulation header. After the completion of the copying process, and the updating of the *RecLen* field within the allocated buffer, the length of the data received is compared to the specified length of the IP datagram. If these fields match, the entire IP datagram has been received and is stored within the buffer, which can be passed to the IP stack for processing and the fragmentation buffer reinitialized.

This concludes the discussion of the device driver, and the workings of the IP stack. However, in order for the stack to be of use, there needs to be an application utilizing its features. As the goal of the project was to provide a mechanism that facilitates the

control and monitoring of audio devices within IEEE 1394 networks, the application selected was based on the QSC-24 protocol, which was introduced in chapter 2.

5.4 QSC-24 over IEEE 1394

Chapter 4 provided an overview of the system configurations that are suitable for QSC-24 networks. This section provides the implementation specifics for these configurations allowing the transmission of QSC-24 messages.

The two types of nodes involved in this scenario are the controller and the controlled audio device. The controller is a PC that is capable of issuing QSC-24 control commands, while the controlled audio device is able to receive these commands, and respond in an appropriate manner. The controlled audio device consists of a Digital Harmony DHIVA, functioning as a legacy adapter for a QSC Audio Products CM16 device. Both the controller and the controlled device are IEEE 1394 compliant, and have an application utilizing a TCP/IP stack for communication. Therefore the QSControl Application can be used as the control application. This illustrates the use of an existing application to control devices over the IEEE 1394 bus, without the need to alter the application or the control protocol used.

Three plausible approaches were identified for the design of the application on the controlled audio device. The initial approach, depicted in figure 5.20, was to provide a QSC-24 message processor above the IP stack, but within the firmware of the DHIVA. This would allow the immediate processing of QSC-24 messages. The drawback of this method is that the IEEE 1394 interface device is intended as a legacy adapter to a CM16 device, and if the processing occurs within the legacy device, the CM16 device firmware is largely redundant, resulting in significant changes to it. This is partly due to an already existing QSC-24/IP stack within the CM16 device, which utilizes an Ethernet interface.

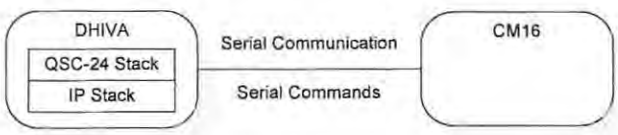

Figure 5.20: Serial command approach

To allow the CM16 device to utilize its existing QSC-24 stack, an application was developed that would supply the raw QSC-24 messages to the device for processing. Stripping off the IP headers of received datagrams and transmitting the encapsulated QSC-24 message to the CM16 device, via the host serial port, achieved this. This would allow the QSC-24 message processor within the CM16 device, to function via both the IEEE 1394 and Ethernet interfaces, although some modification to the CM16 stack would be necessary to cater for these changes. This was the second approach adopted and is shown in figure 5.21.

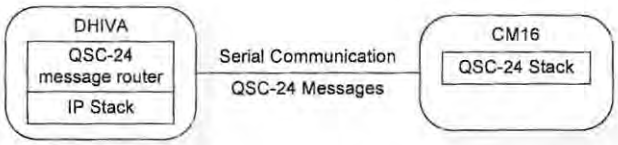

Figure 5.21: QSC-24 message approach

The last approach, termed the socket proxy approach, was to provide some means of exposing the socket interface via a serial port to the CM16 device. This approach is illustrates in figure 5.22. The motivation behind this approach was to allow the controller access to IP based applications running on the CM16, without completely overhauling its embedded IP stack. It was speculated that by exposing the socket interface of the IP stack running on the IEEE 1394 device, minimal changes to the CM16 would be required. This facility could also be used for QSC-24 datagram transmission, as even fewer changes would be necessary.

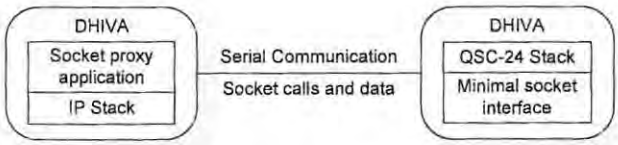

Figure 5.22: Socket interface approach

Unfortunately, there was no CM16 devices available to function as a legacy device, resulting in the simulation of a CM16 using a PC. The purpose of the simulated device was not to allow the control of audio, but to demonstrate the transmission and reception of control related data over the IEEE 1394 network, in a way that could easily be ported to a real CM16 device. With this goal in mind, the first approach identified was ruled out, as it did not lend itself immediately to the real world

scenario, although it is ultimately the desired solution. Approaches two and three were implemented, as will be described in the following sections.

5.4.1 Raw QSC-24 Packets via the Serial Port

This approach required the implementation of two applications. One for the embedded environment and the other for the simulated device within the PC. The application on the embedded legacy adapter is responsible for binding to UDP port 2501, which is the specified QSC-24 data communications port, and simply receives and transmits QSC-24 messages through this port. All packets received on this UDP port, are transmitted over the host serial port. Similarly, all messages that are received via the host serial port by the embedded application, are transmitted over the IEEE 1394 bus through this UDP port.

The simulated CM16 device interacts with the legacy device through a serial port, as is described above. The PC application must be able to form and transmit, as well as receive and decode QSC-24 messages. The application utilized, depicted in figure 5.23, is able to receive and display any QSC-24 message received via TCP/IP or through a serial port. Support is also provided for the transmission of QSC-24 messages utilizing both the TCP/IP and serial interfaces. QSC-24 messages that are received are displayed according to the columns shown in figure 5.23, including fields for Version, Type, Reply Address, Destination Object, Destination Method, and Parameter list. The Version and Type field contain defined values, which are currently restricted to the value of 1. The Reply Address provides the return address for responses. The destination of the message is specified by the Destination Object and Method fields which identify the targeted object and the method within that object. The Parameter field displays the parameter list that is to be passed to the targeted method.

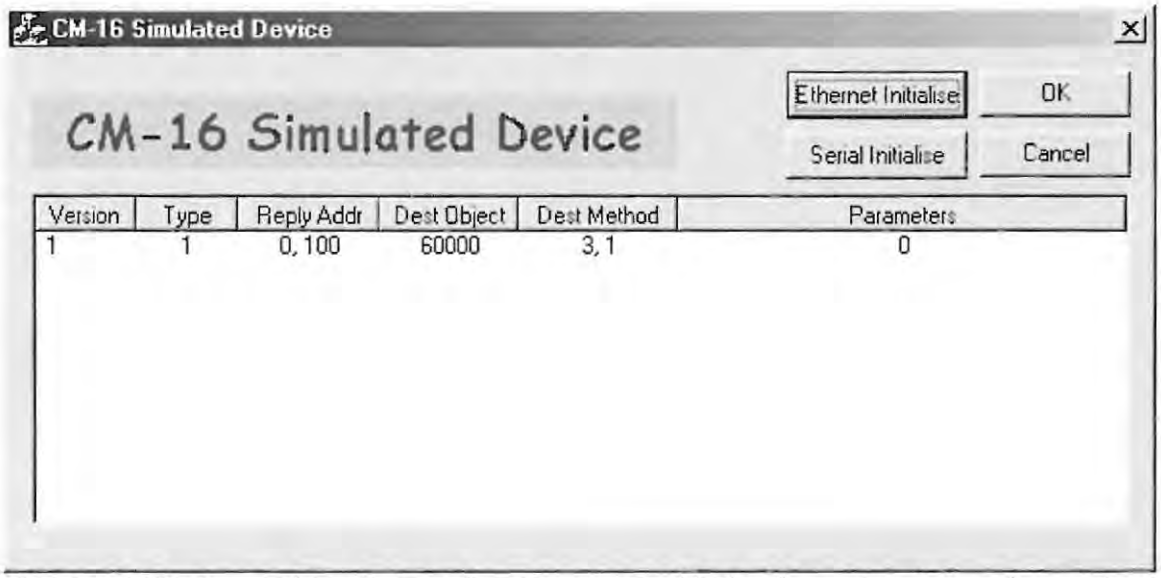

Figure 5.23: Screenshot of an application responsible for the generation and reception of QSC-24 data messages

The Initialise buttons, in the above screenshot, are used for the transmission of a QSC-24 initialize message through either the Ethernet or the serial interface. If this message is sent via the serial interface, it is intercepted by the embedded application, and transmitted over the IEEE 1394 network. The list control is used to provide details of the QSC-24 messages that have been received, by decoding the message and displaying the encapsulated values within the relevant fields.

The PC application is very rudimentary, but it serves to illustrate that the transmission and reception of QSC-24 messages via the serial port can be easily achieved. For example, any messages sent by the QSControl system, over the IEEE 1394 bus, will be routed, via the serial port, to this application and will displayed in a meaningful format.

5.4.2 A Socket Interface Approach

The socket proxy facility provides a means for legacy devices to make use of the socket calls already stack present within the firmware of the embedded legacy adapter. This is achieved by providing an interface to a number of selected socket calls through the host serial port. All of the arguments to these exposed socket calls are provided via the serial port, and error information is reported back to the user through the serial port, as is required.

The need for this facility has arisen due to legacy devices having been tailored to run certain IP based applications, such as Telnet or TFTP. These applications should be run within the IEEE 1394 device, allowing direct access to the socket calls. However, an intermediate development step has prompted the development of a socket proxy, allowing these legacy devices to maintain control over their applications but still gain the benefits offered by IEEE 1394 technology.

The implementation approach adopted was as follows. Sequences of characters are used to identify the targeted routine, and parameter values to be used. If an error occurs while a socket call is being processed, this error is reported back the host through the serial port. Appendix D provides a complete listing of the defined characters and their associated socket call.

At present only a small subset of the socket calls have been exposed, all of which cater for the use of UDP messages. Table 5.5 provides the subset currently exposed, as well the format of the messages that are used. Socket calls are identified by a leading 'S' character, followed by a ',', and then the length field, followed by another ','. The commas are used to denote end of logical field groupings, and serve as check tokens to ensure consistency when extracting data for the socket calls. The length field indicates the length of the actual socket data, excluding the comma immediately following the length field.

Typically the first call would be to open a UDP socket, which is done using the udp_open() socket call. A port field is associated with this call, which indicates the port that is to be opened, and consists of a 32-bit integer broken down into its relevant 8-bit bytes, as is illustrated in table 5.5. It is important that all 4 of these 8-bit groupings are sent, and this rule should be applied to all 32-bit integer transmissions.

An error or success status will be returned to the host. This is done by means of an 'S' and ',' characters followed by a length field and another ','. Following this header information is either a 'P' or an 'E' character, with P indicating success and E failure. This is followed by another ',' and then a 32-bit integer value whose value is dependent on whether the socket call succeeded or not. If the socket call was a success the integer value contains the identifier of a socket, which is used in all future

socket calls to that specific socket. However, if an error occurred while opening the socket, the relevant error code is contained within the integer. The error code returned is the same code as is returned by the failed socket call.

A socket that has been opened can be utilized through the send and receive calls exposed by passing in the appropriate parameters. The socket field within these calls must contain a valid socket identifier as returned by the socket proxy facility. A number of different flags can also be passed into these socket calls. These flags are identified by a sequence of characters, as defined in table 5.5. Simply transmitting the desired character sequence can use more than one of these flag variables. For example, transmitting the sequence 'B' 'U', would result in the flags MSG_BLOCKING and MSG_URGENT being used in the procedure call.

All Socket calls preceded by:	'S'	','	Length of data (excluding the next comma)	','

recv

'R'	'E'	','	socket (bits 32-24)	socket (bits 24-16)	socket (bits 6-8)	socket (bits 8-0)
','	n[flags]	','	BufLen (bits 32-24)	BufLen (bits 24-16)	BufLen (bits 6-8)	BufLen (bits 8-0)

Note:
flags are defined as follows:

'B'	MSG_BLOCKING
'N'	MSG_NONBLOCKING
'P'	MSG_PEEK
'T'	MSG_TRUNCATE
'U'	MSG_URGENT

sendto

'S'	'T'	','	socket (bits 32-24)	socket (bits 24-16)	socket (bits 6-8)	socket (bits 8-0)
','	n[flags]	','	IP	IP	IP	IP
','	Port (bits 32-24)	Port (bits 24-16)	Port (bits 6-8)	Port (bits 8-0)		

Note:
flags are defined as follows:

'B'	Non-Blocking
'D'	MSG_DONTROUTE
'F'	MSG_FDBROADCAST
'N'	MSG_NONBLOCKING
'O'	MSG_OOB

udp_open

'U'	'O'	','	Port (bits 32-24)	Port (bits 24-16)	Port (bits 6-8)	Port (bits 8-0)

Table 5.5: Current subset of the socket interface exposed via the serial interface

There are a number of limitations to the approach adopted, of which some are inherent to the chosen design, and others to the hardware platform. The most crippling of these is the communications link between the host and the DHIVA. This communications link is through the host serial port, which is severely limiting when high data transfer rates are required. The maximum buffer size that can be transmitted and received by the serial port is also a possible limitation, as at present an "all or nothing" solution is used. That is, all the data is written using one buffer, and if the buffer is too small no data is transmitted or received.

A design limitation is that the socket proxy runs on a single thread. This may become a bottleneck when more than one application wishes to use the socket proxy interface simultaneously. For example, when a Telnet application issues a receive socket command to the socket proxy thread, the thread blocks until that data is received or an error occurs. If a TFTP application wishes to send data, a send command is issued. The socket proxy thread cannot handle the send command as it is blocked waiting to receive data. This limitation can be overcome by creating intelligent applications on the host side that are aware of the socket proxy thread's status.

It must be emphasised that although this approach has been successful in its implementation so far, it is not the desired solution. Ultimately having the application reside directly above the IP stack will provide the most effective, and efficient solution. This solution has been provided merely as a proof of concept facility in order to demonstrate the advantages that are to be gained from using an IP over 1394 solution.

5.5 Summary

This chapter set the scene for the implementation environment utilized to provide an IP stack above an embedded IEEE 1394 device. Included, was an overview of the porting process, and the mechanisms involved in the transmission and reception of messages through the stack. In particular, the device driver was detailed, and the Ethernet-to-1394 mapping mechanism explained.

However, in order to allow control and monitoring data to flow over IEEE 1394 using IP, a control application was needed within the embedded environment. The two

implemented application approaches were discussed, and the implications of each highlighted.

The next chapter provides an overview of Universal Plug and Play (UPnP) architecture, which provides mechanisms for the discovery, control, monitoring, and presentation of devices. The chapter will establish that with the addition of the IP stack to the IEEE 1394 environment, a wide variety of applications and protocols can now be utilized over the IEEE 1394 bus.

Chapter Six

An Implementation of the Universal Plug and Play Architecture

The UPnP architecture was introduced in section 4.4.2 as an appropriate protocol for the control and monitoring of devices. This is due to the UPnP architecture providing facilities to cater for discovery, description, control, monitoring, and presentation.

This chapter examines the use of the UPnP architecture for the control and monitoring of professional audio devices in sound installations through the implementation of an example audio device. This device is modelled on an existing professional audio device. Additionally, the possibility of utilising the UPnP architecture over IEEE 1394 networks within sound installations is discussed.

6.1 UPnP Architecture

Section 4.4.2 introduced the UPnP architecture as a peer-to-peer network comprised of intelligent devices. The design of the UPnP architecture brings standards-based connectivity to unmanaged networks through the use of a standardised set of communication protocols. The types of devices encountered within UPnP networks consist of controllers or control points and controlled devices. Figure 6.1 illustrates a simple UPnP network consisting of two control points and three controlled devices. As the UPnP architecture is, by definition, not restricted by the transmission technology utilised, the connections between the devices shown in figure 6.1 illustrate that communication can take place between any devices present on the network. For instance, the device labelled as *Controller 1* can exert control over any of the controlled devices present on the network, such as *Controlled Device 3*.

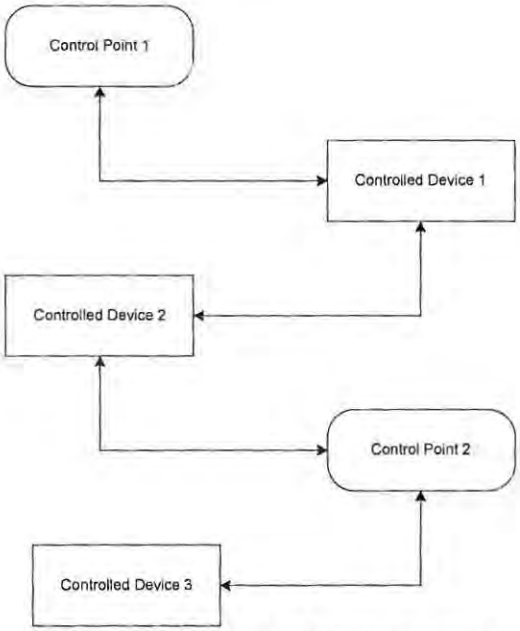

Figure 6.1: Simple UPnP network

Intercommunication between UPnP devices occurs through the use of standardised protocols which are listed in figure 6.2. The blocks labelled GENA, SSDP and SOAP represent the General Event Notification Architecture, Simple Service Discovery Protocol and Simple Object Access Protocol, respectively. These protocols are utilised for the discovery and eventing procedures utilised by the UPnP architecture and are discussed in sections 6.1.2, 6.1.5 and 6.1.4. Section 4.4.2 briefly introduces the other protocols present within the stack.

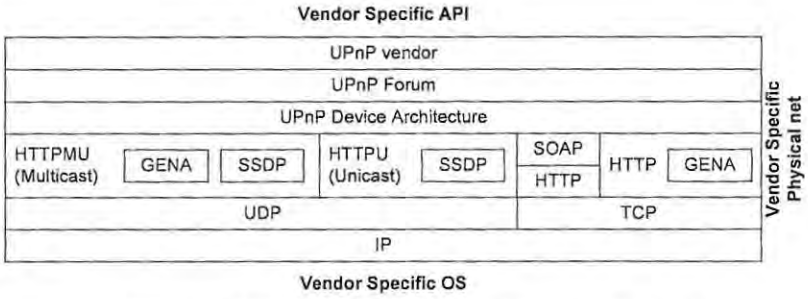

Figure 6.2: UPnP architecture protocol stack

Illustrated in figure 6.2 is the reliance of the UPnP architecture upon IP-based protocols such as TCP/IP, UDP/IP, and other web-based protocols such as HTTP. Consequently UPnP transmission technologies are currently restricted to those that support the transfer of IP datagrams. This illustrates the dependency of many control and monitoring protocols on IP-based networks, and provides motivation for the

implementation of an IP-stack on IEEE 1394 compliant nodes. Apart from the UPnP architectures dependency on IP-based networks, the facilities that are provided by the architecture include the following:

- Addressing,
- Discovery,
- Description,
- Control,
- Eventing, and
- Presentation.

These facilities are provided through the definition of currently available protocols that satisfy a particular task. For example, the GENA architecture is utilised for all notifications within the UPnP architecture. However, if an alternative protocol is developed that is better suited to the UPnP architecture; the currently specified protocols can be replaced with only minimal alterations to the other facilities. For example, the discovery protocol currently utilised is SSDP, which can easily be replaced with a more suitable protocol should one be developed. These alterations to the UPnP architecture specification are subject to approval by the UPnP Forum and its members.

Devices within the UPnP architecture are described in terms of embedded or logical devices and services. Such a device is illustrated in figure 6.3. It is these embedded devices and services that are advertised to control points, and that permit their control and monitoring.

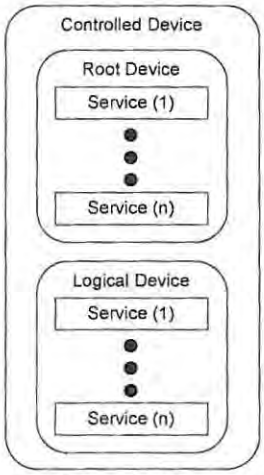

Figure 6.3: A UPnP device

6.1.1 Addressing

As the UPnP architecture is currently IP dependant it follows that the addressing scheme utilised is based on the use of IP addresses. The UPnP Architecture document recommends the use of a Domain Name Service (DNS) allowing the mapping of volatile IP addresses to static textual names for devices. This allows devices to be referred to by name regardless of changes to the IP address assigned to that device.

The determination of an IP number for a device follows one of two paths. The first path attempted is through the use of the Dynamic Host Configuration Protocol (DHCP), which requests an IP address from a DHCP server. If no server exists on the network, or no response is received within a specified time limit, the second path known as auto-configuration is attempted. This path utilises implementation specific automatic IP address configuration routines, which allocates an IP address from a predetermined range. The IP address selected from either of these paths is then tested to ensure that no other device on the network is currently operating on this address. This is achieved through the use of an ARP probe, which is an ARP request message with the sender IP address set to 0 and the target IP address set to the above selected address [Microsoft Corporation, 2000]. If no response to this probe is received, the address can be assumed to be unique and normal node activity may continue.

A device that has determined its IP address via auto-configuration is required to periodically check for the presence of a DHCP server on the network. If such a server is present, the device must utilise the IP address assigned by the server.

6.1.2 Discovery

The discovery phase follows the completion of the address allocation phase, and is responsible for the discovery of other devices and services available on the network. This phase consists of controlled nodes advertising their services to control points upon joining the network, and permits control points to search for devices of interest on the network. The discovery message utilised in both instances consists of a few device specifics, such as its type, identifier, and a reference to more detailed information [Microsoft Corporation, 2000].

A number of multicast messages are transmitted upon the insertion of a new device onto the network. These messages contain information advertising the root device, embedded devices and services that are available. Similarly, the messages transmitted by control points searching for devices on the network are multicast messages to which all controlled devices are required to listen. If the search criteria specified within these messages corresponds with that of a controlled device the controlled device is required to transmit a response.

If a device is in the process of being removed from the network, the device should multicast a number of messages revoking its previous advertisements. This notifies control points, and other devices utilising these services that these will no longer be available.

The discovery messages and the associated multicast addresses are defined by the SSDP. The SSDP consists of discovery requests and presence or service announcements. Discovery requests, as illustrated in figure 6.4, are utilised by control points for the discovery of available services present within controlled devices. These requests contain a search pattern or target which corresponds to a type or identifier for a device or service. If a controlled device should receive such a request and it contains an embedded device or service that satisfies the search criteria, a response message is to be transmitted.

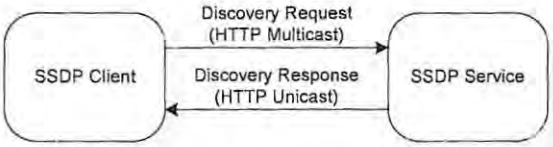

Figure 6.4: SSDP discovery request and response

All discovery requests and responses utilise HTTP messages that are transmitted via UDP/IP, with requests utilising multicast transactions and responses being a unicast message targeted at the requesting device. Included within a discovery request message is a field that specifies a maximum wait period in which controlled devices are required to respond. This permits the controller to receive and process numerous responses, as each controlled device selects a random number within this wait period. The selected value corresponds to a wait period in seconds that devices must delay before transmitting responses. This delay avoids the transmission of multiple responses simultaneously by devices on the network. A search target is also specified within the discovery request, for which the following are defined [Microsoft Corporation, 2000]:

- *ssdp: all* – Search for all devices and services
- *upnp:rootdevice* – Search for root devices only
- *uuid:device-UUID* – Search for a particular device containing the Universally Unique Identifier (UUID) as specified by the UPnP vendor
- *urn:schemas-upnp-org:device:deviceType:v* – Search for a device of this type, where the device types and version are defined by the UPnP working committee
- *urn:schemas-upnp-org:services:serviceType:v* – Search for a service of this type, where the service types and version are defined the UPnP working committee

Discovery responses resemble service announcements and provide the same information, with service announcements providing the facilities required for the advertising of devices, embedded devices and services. These service announcements are illustrated in figure 6.5 and utilise multicast HTTP messages while discovery responses are unicast messages.

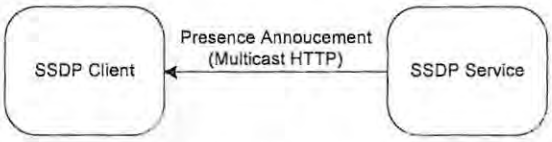

Figure 6.5: SSDP presence announcements

The two service announcements defined are *alive* and *byebye* notifications, which are comprised of the following four major components [Microsoft Corporation, 2000]:

- A potential search target,
- A composite identifier for the advertisement,
- A Uniform Resource Locator (URL) containing a UPnP description of the root device, and
- A duration for which the advertisement is valid.

The *alive* messages announce the presence of a device or service, while the *byebye* message is utilised for the removal of a device or service from the network. The number of *byebye* messages transmitted must correspond to the number of *alive* messages transmitted.

6.1.3 Description

After the discovery phase is complete the description phase may commence, where control points can extract further information about controlled devices. This is possible as the responses to discovery messages and service advertisements contain a URL to the UPnP description of a device, which control points can retrieve. This description is composed of device and service descriptions. The device descriptions describe the physical and logical containers within a device, and the service descriptions provide the capabilities exposed by the device [Microsoft Corporation, 2000]. These descriptions are provided in Extensible Markup Language (XML) syntax, and are based on the UPnP device and service templates, respectively.

XML is a restricted form of the Standard General Markup Language (SGML), whose documents are made up of entities. These entities contain either parsed or unparsed data. Parsed data is made up of characters which form either character data or markup. Markup resembles HTML tags and is utilised to encode a description of a document's storage layout and logical structure, upon which constraints may be imposed [World Wide Web Consortium, 2000]. XML documents are read by an

XML parser which is able to provide access to both content and structure. An example of an XML document is provided in figure 6.6.

```
<?xml version="1.0" ?>
- <root xmlns="urn:schemas-upnp-org:device-1.0">
  - <specVersion>
      <major>1</major>
      <minor>0</minor>
    </specVersion>
    <URLBase>Base URL for all relative URLs</URLBase>
  - <device>
      <deviceType>urn:schemas-upnp-org:device:deviceType:v</deviceType>
      <friendlyName>Short user-friendly name</friendlyName>
      <manufacturer>Manufacturer name</manufacturer>
      <manufacturerURL>URL to manufacturer site</manufacturerURL>
      <modelDescription>Long user-friendly title</modelDescription>
      <modelName>Model name</modelName>
      <modelNumber>Model number</modelNumber>
      <modelURL>URL to models site</modelURL>
      <serialNumber>Manufacturer assigned serial number</serialNumber>
      <UDN>uuid:Unique Device Name</UDN>
      <UPC>Universal Product Code</UPC>
    + <iconList>
    + <serviceList>
      <deviceList>Embedded device descriptions, if required</deviceList>
      <presentationURL>URL for presentation</presentationURL>
    </device>
  </root>
```

Figure 6.6: XML device description document

The device description includes vendor specific information such as a model name, serial number, and manufacturer name. A sample of these and other fields present within the device description are illustrated in figure 6.6 [Microsoft Corporation, 2000]. The addition and subtraction signs alongside the entities displayed in figure 6.6, allow these entities to be expanded and contracted respectively. This illustrates the hierarchical structures present within XML documents. For example, the *serviceList* entity can be expanded to reveal a number of defined services as shown in figure 6.7.

```
- <serviceList>
  - <service>
      <serviceType>urn:schemas-upnp-org:service:serviceType:v</serviceType>
      <serviceID>urn:upnp-org:serviceID:serviceID</serviceID>
      <SCPDURL>URL to service descriptions</SCPDURL>
      <controlURL>URL for control</controlURL>
      <eventSubURL>URL for eventing</eventSubURL>
    </service>
  + <service>
  + <service>
    Other service declarations, if required
  </serviceList>
```

Figure 6.7: Expanded serviceList entity

The *serviceList* entity contains one or more *service* entity entries, which describe the services that are available for the device that is being described. The *service* entity includes fields that provide the URLs for control and eventing, as well as specifying where additional description information may be obtained. This additional information is provided in the form of an XML service description document, as illustrated in figure 6.8.

```
  <?xml version="1.0" ?>
- <scpd xmlns="urn:schemas-upnp-org:service-1-0">
  + <specVersion>
  - <actionList>
    + <action>
      Other action declarations, if required
    </actionList>
  - <serviceStateTable>
    + <stateVariable sendEvents="yes">
    + <stateVariable sendEvents="yes">
      Other state variable declarations, if required
    </serviceStateTable>
  </scpd>
```

Figure 6.8: Service description document

The two main entities within the service description document are the *actionList* and *serviceStateTable* entities. The *actionList* entity, as described by figure 6.9, consists of a number of actions with associated arguments, with these actions comprising a particular service supported by a device. Associated with each of these arguments is the name of a related state variable, which is to be defined in the *serviceStateTable* entity.

```
- <actionList>
  - <action>
      <name>Action name</name>
    - <argumentList>
      - <argument>
          <name>Formal parameter name</name>
          <direction>in or out</direction>
          <retval />
          <relatedStateVariable>State variable name</relatedStateVariable>
        </argument>
        Other argument declarations, if required
      </argumentList>
    </action>
    Other action declarations, if required
  </actionList>
```

Figure 6.9: actionList service entity

The *serviceStateTable*, which is represented in figure 6.10, provides the variables that represent the current state of a service, and consequently the state of the device. State variables are exposed to control points through services, and may be altered by the invocation of these services. It is for this reason that arguments are required to correspond with a defined state variable. Predetermined values may be assigned to these variables, and minimum and maximum values specified, as illustrated in figure 6.10. The definition of a step value is particularly useful for audio applications, as it allows the resolution of rotary controllers or sliders to be advertised.

```
- <serviceStateTable>
  - <stateVariable sendEvents="yes">
      <name>Variable name</name>
      <dataType>Variable data type</dataType>
      <defaultValue>Default value</defaultValue>
    - <allowedValueList>
        <allowedValue>Enumerated value</allowedValue>
        Other allowed value declarations, if required
      </allowedValueList>
    </stateVariable>
  - <stateVariable sendEvents="yes">
      <name>Variable name</name>
      <dataType>Variable data type</dataType>
      <defaultValue>Default value</defaultValue>
    - <allowedValueRange>
        <minumum>Minimum value</minumum>
        <maximum>Maximum value</maximum>
        <step>Increment value</step>
      </allowedValueRange>
    </stateVariable>
    Other state variable declarations, if required
  </serviceStateTable>
```

Figure 6.10: serviceStateTable entity

In order for control points to discover more about the services that are advertised by a device, the control point is required to retrieve the XML documents provided within the URL locations described. This retrieval occurs via an HTTP GET request over TCP/IP to the controlled device, which will return a response containing the XML document at the requested URL. Once the retrieval of this document is complete, the control point can parse the document to extract the information regarding the services and actions supported by the controlled device.

6.1.4 Control

Subsequent to control points discovering the services that are available within controlled devices, these services can be requested to invoke actions and control points can poll these services for the values of their state variables. This action invocation resembles a remote procedure call, with the control point sending a suitably formatted message to a specified service. The control URL to which these requests are sent is advertised to control points in the description documents, as is illustrated in figure 6.7.

The protocol utilised for the invocation of actions is the Simple Object Access Protocol (SOAP), which is delivered via HTTP over TCP/IP. SOAP defines the use of XML and HTTP for remote procedure calls, and is utilised by the UPnP architecture for the delivery of control messages and for the return of results or error codes. A SOAP request is composed of XML data enclosed within a SOAP envelope, as illustrated in figure 6.11. The XML name space (*xmlns*) and *encodingStyle* elements contain values that are defined within the SOAP specification, indicating conformance with this specification. For example, the *encodingStyle* element contains *http://schemas.xmlsoap.org/soap/enconding/*, which indicates compliance with the SOAP encoding rules.

```
- <s:Envelope xmlns:s="http://schemas.xmlsoap.org.soap/envelope/"
    s:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  - <s:Body>
    - <u:actionName xmlns:u="urn:schema-upnp-org:service:serviceType:v">
        <argumentName>in argument value</argumentName>
        Other in arguments and their values, if required
      </u:actionName>
    </s:Body>
  </s:Envelope>
```

Figure 6.11: Example SOAP control request envelope

The *actionName* field specifies the action that is to be invoked. This action resides within the service identified by the *serviceType* attribute. The XML name space (*xmlns*) for this action is defined to be the concatenation of the *serviceType* field and the uniform resource name (*urn*) assigned to the UPnP architecture, which are enclosed within double quotes. The *argumentName* provides the name of the *in* arguments and values that are to be assigned to these arguments. Associated with every argument is a direction, which indicates whether or not this argument is passed into (*in*) or out of (*out*) the action. SOAP requests and responses contain *in* and *out* arguments, respectively.

The targeted service on the controlled devices will receive this control request and is required to transmit a response within 30 seconds, which includes the time taken for transmission. The transmitted response is another SOAP envelope as detailed in figure 6.12. The SOAP request and response envelopes are similar, with the exception that the arguments specified are in the outward direction, and the *actionName* field is tagged with the static text of "Response". The *xmlns* element is identical to that illustrated in the SOAP request.

```
- <s:Envelope xmlns:s="http://schemas.xmlsoap.org.soap/envelope"
    s:encodingStyle="http://schemas.xmlsoap.org/soap/encoding">
  - <s:Body>
    - <u:actionNameResponse xmlns:u="urn:schema-upnp-org:service:serviceType:v">
        <argumentName>out argument value</argumentName>
        Other out arguments and their values, if required
      </u:actionNameResponse>
    </s:Body>
  </s:Envelope>
```

Figure 6.12: Example SOAP control response envelope

In addition to the invocation of services, a facility is provided to allow for the querying of a state variable's value. This facility is, however, restricted to one variable per SOAP envelope, which results in additional requests for the querying of multiple variables. The envelope associated with such a query is illustrated in figure 6.13, where the *varName* entity is the name of the state variable that is to be queried. The XML name space associated with the *QueryStateVariable* entity is defined within the UPnP architecture specification [Microsoft Corporation, 2000].

```
- <s:Envelope xmlns:s="http://schemas.xmlsoap.org.soap/envelope"
    s:encodingStyle="http://schemas.xmlsoap.org/soap/encoding">
  - <s:Body>
    - <u:QueryStateVariable xmlns:u="urn:schemas-upnp-org:control-1-0">
        <u:varName>Name of variable to query</u:varName>
      </u:QueryStateVariable>
    </s:Body>
  </s:Envelope>
```

Figure 6.13: Example SOAP state variable query request envelope

An additional facility is provided that allows state variables to notify control points should their values be modified, which does not form part of the control phase of the UPnP architecture and is discussed in section 6.1.5.

The response envelope from a state variable query request resembles a control response, and is illustrated in figure 6.14. The value contained within the state variable query is contained within the *return* entity.

```
- <s:Envelope xmlns:s="http://schemas.xmlsoap.org.soap/envelope"
    s:encodingStyle="http://schemas.xmlsoap.org/soap/encoding">
  - <s:Body>
    - <u:QueryStateVariableResponse xmlns:u="urn:schemas-upnp-org:control-1-0">
        <return>State variable value</return>
      </u:QueryStateVariableResponse>
    </s:Body>
  </s:Envelope>
```

Figure 6.14: Example SOAP state variable query response envelope

If an error is encountered by the service through the invocation of the requested method or if a service cannot provide a value for the request state variable, an appropriate response must be returned reflecting the error. This is achieved through

the definition of a SOAP envelope specifically for the reporting of errors, which is illustrated in figure 6.15.

```
- <s:Envelope xmlns:s="http://schemas.xmlsoap.org.soap/envelope"
    s:encodingStyle="http://schemas.xmlsoap.org/soap/encoding">
  - <s:Body>
    - <s:Fault>
        <faultcode>s:Client</faultcode>
        <faultstring>UPnPError</faultstring>
      - <detail>
        - <UPnPError xmlns="urn:schemas-upnp-org:control-1-0">
            <errorCode>Appliable error code</errorCode>
            <errorDescription>Textual description of error</errorDescription>
          </UPnPError>
        </detail>
      </s:Fault>
    </s:Body>
  </s:Envelope>
```

Figure 6.15: Example SOAP error envelope

The *faultstring* entity is defined to contain the value "UPnPError", as the details of the error are provided within the *UPnPError* entity. An integer value is utilised to represent an error code, with the textual description being provided within the *errorDescription* entity.

6.1.5 Eventing

As detailed within the discussion of the description mechanism utilised by the UPnP architecture, section 6.1.3, a UPnP service description includes a list of actions a particular service implements and a list of state variables that represent the current state of the service. These variables can be defined as evented variables, and any modification to the values of these variables will be published. However, in order for control points to receive this updated variable information, they are required to subscribe for such notification with the service controlling the variable.

The subscription and event messages that are transmitted and received within the eventing phase of UPnP based networks are based on GENA, which extends the HTTP protocol through the definition of SUBSCRIBE and NOTIFY facilities. These extended HTTP messages are transmitted via TCP/IP.

Control points or subscribing devices must register with a controlled or publishing device through the use of a SUBSCRIBE message. The three possible SUBSCRIBE message variants defined are subscription, renewal, and cancellation messages. The subscription message is utilised for the creation of subscription entries within the list maintained by the publisher. This list contains the relevant information for the transmission of event messages. The list is kept updated through the use of renewal messages. The cancellation message informs the publishing device of a subscription that is to be removed from the list. The subscription list is required to maintain the following information for each subscription [Microsoft Corporation, 2000]:

- Unique subscription identifier – This identifier is generated by the publisher in response to a subscription message and uniquely identifies the subscription.
- Delivery URL for event messages – The URL to which published messages will be transmitted, and is provided by the subscriber in the subscribe message.
- Event key – This key is numbered sequentially for each event message and is initially set to zero. This allows subscribers to verify that no event messages have been lost.
- Subscription duration – The duration of time until this subscription expires.

Figure 6.16 illustrates the communication model associated with eventing. The first step of this model requires the subscribing device to transmit a subscription message which includes the following information:

- Publisher path – The path component of the eventing URL specified within the XML description documents.
- Callback – One or more URLs to which event messages are to be sent.
- Timeout – Requested duration until the subscription expires.

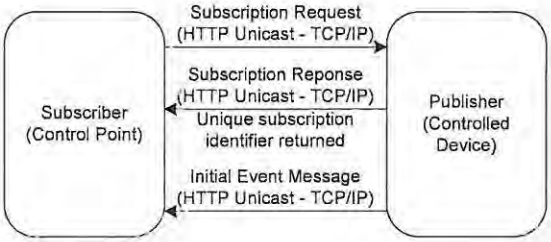

Figure 6.16: Subscription communication model

The information transmitted with the subscription message is utilised by the publisher to construct an entry for the subscription list, provided that the publisher is not over subscribed. If the publisher has sufficient resources available to satisfy the request, the subscriber list will be updated with the created entry specifying a unique subscription identifier and duration as determined by the publisher. This will result in the transmission of a subscription response message to the subscriber, which includes the unique service identifier and timeout value allotted for the subscription. As soon as possible after the response has been transmitted, an initial event message will be transmitted by the publisher, containing values for all of the evented variables supported by the targeted service.

The renewal message is transmitted to renew a subscription before it expires, the duration of which is determined by the publisher and communicated via previous subscription or renewal response messages. The renewal message is a subscribe message that includes the unique subscription identifier returned by the publisher and a request duration. If the publisher accepts the renewal, a message identical to the original subscription response will be generated containing the new duration value. The subscription list entry for this particular subscription will also be updated with this new value. If the renewal message cannot be accepted by the publisher an error message is to be returned to the subscriber with an appropriate error code. The transmission of a subscription renewal request and response is illustrated in figure 6.17.

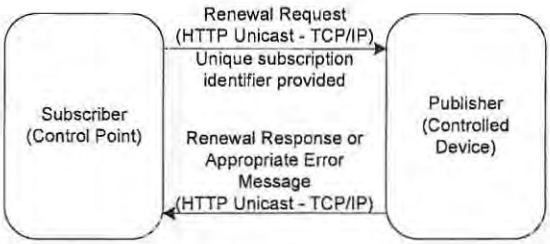

Figure 6.17: Subscription renewal request and response

The unsubscribe message is responsible for the removal of subscription information from a subscription list when eventing information is no longer required by a subscriber. The communication model associated with unsubscribe messages is the same as the request-response model illustrated for subscription renewal messages.

The service that is to be unsubscribed is identified to the publisher by the inclusion of the unique service identifier originally issued to the subscriber. Upon receipt of such a message, the identified service will be removed from the subscription list to avoid any further events being transmitted to the subscriber. The response to an unsubscribe message is simply an HTTP 200 OK message.

Event messages that are generated due to state variable changes are transmitted to all devices with valid entries in the subscription list maintained by the publishing device. These change notifications are required to be transmitted as soon as possible after the actual change, as this allows subscribers to maintain accurate information and user interfaces present on subscribers appear responsive. Should a number of variables be altered simultaneously, all of these values may be specified within one eventing message in order to minimise the amount of network traffic and processing.

NOTIFY request messages specify the new values of evented variables using the XML notation, as illustrated in figure 6.18. The *variableName* element provides the name of the altered variable with the new value provided within the tag. Included within the headers for the NOTIFY request is a sequence or event key field which is incremented for every event sent to a particular subscriber. Subscribers are required to respond to each NOTIFY request message, typically with a 200 OK message or an appropriate error code. If no response is received by the publisher, the event message will be abandoned for this particular subscriber, but further event messages will be sent until the subscription expires.

```
- <e:propertyset xmlns:e="urn:schemas-upnp-org:event-1-0">
  - <e:property>
      <variableName>New value</variableName>
    </e:property>
    Other variables names and values, if any
  </e:propertyset>
```

Figure 6.18: Evented variable notation

6.1.6 Presentation

The presentation phase exposes an HTML-based user interface to control points through which a device may be controlled. This allows a controlled device to specify its own user interface, including the messages that will be sent to its services for the

invocation of actions. As these actions are invoked through the use of SOAP messages, an embedded scripting language, such as VB Script, may be included inside the HTML-based user interface. This allows a script enabled web browser control over the device. Additional non-UPnP facilities may be embedded within the HTML document for the utilisation of specific device capabilities not catered for by the UPnP architecture. The location of the HTML document is specified by the *presentationURL* element present within the device description document. The transmission of the HTML document is via HTTP over TCP/IP, through the use of an HTTP GET operation.

6.2 Implementation of a Professional Audio Device using the UPnP Architecture

This section provides an example UPnP architecture implementation of a simulated professional audio device. The audio device simulated is a FireBob or FB-88 device, which was developed by Peavey Electronics with the IEEE 1394 interface provided by Digital Harmony Technologies [Digital Harmony Technologies, Inc., 2000]. Section 6.2.1 provides specific details of the facilities and operating modes provided by the FireBob.

The ideal solution would have been to modify the Digital Harmony DHIVA device, introduced in chapter 5, to provide it with UPnP capability. This would allow the DHIVA to function as a legacy adapter to a host of currently available audio devices, providing them with UPnP capability and allow a network of professional UPnP enabled audio devices to be constructed and tested. However these modifications were not attempted due to memory restrictions on the DHIVA. The modifications required to provide such a device with UPnP architecture compatibility are the introduction of a suitable web server, XML parser, and the UPnP architecture stack illustrated in figure 6.2.

The UPnP architecture simulation entails the implementation of a simulated FireBob that is able to respond to UPnP requests. It also entails the formation of relevant description documents, and the creation of an appropriate HTML document with embedded scripting to allow control over the simulated device. The simulated device was created using the 'UPnP Development Kit 1.0' released by the Microsoft Corporation [Microsoft Corporation, 2001]. This development kit included examples

of other simulated devices as well as the code required for the Internet Information Services (IIS) web server extensions to allow the UPnP architecture specified protocols of SOAP and GENA to operate. The simulated device provides an appropriate UPnP controlled device, while a standard web browser is utilised as a control point. Figure 6.19 illustrates the communication that takes place between the web browser and the simulated device during each phase of the UPnP architecture. Each of these phases is discussed in turn in the subsequent sections.

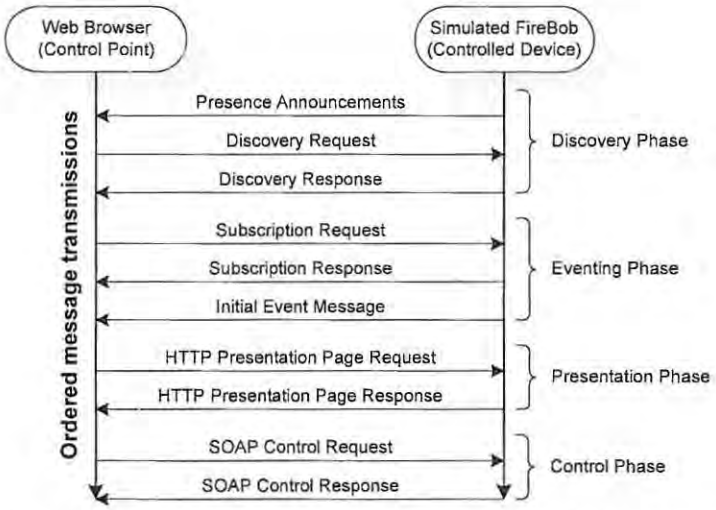

Figure 6.19: Control point and simulated device communication

As UPnP devices are highly self-descriptive, the code within the development kit remains fairly consistent regardless of the application. This is illustrated in figure 6.20, where the modules that required alterations for the implementation of the simulated FireBob are represented by the shaded blocks. This figure also displays the logical flows within the development kit, and illustrates the modular approach adopted. Further information regarding the standard modules can be found in the documentation provided with the 'Microsoft Development Kit 1.0' [Microsoft Corporation, 2001].

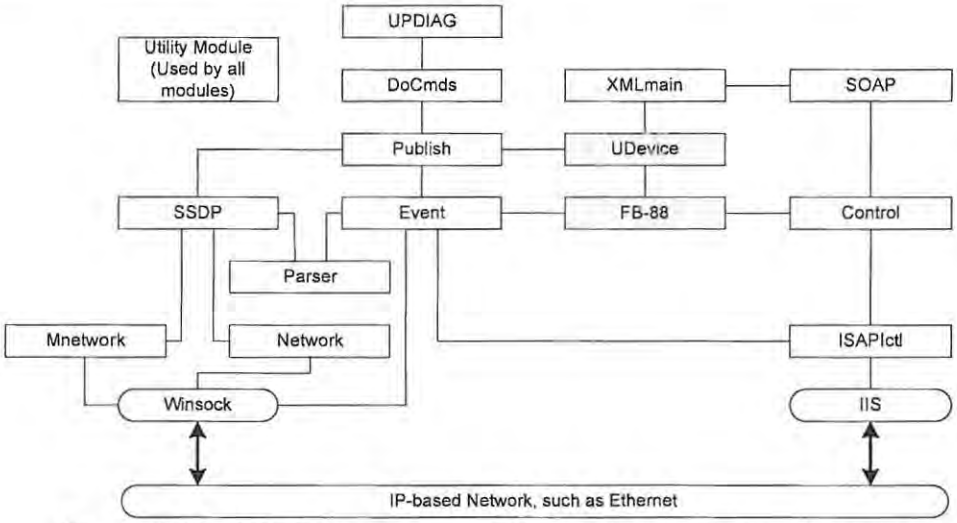

Figure 6.20: UPnP Development Kit module interaction diagram

The *UDevice* module provides the function prototypes for the simulated device and service control table. This table defines the services that are supported by the device, and associates these services with textual strings that will be used to invoke them. The service control table also illustrates the hierarchical layout of the implemented device and must correspond with the service description documents created. This is illustrated in figure 6.21, where a service entry for the service called *channel1* is shown. Included below the service declaration, are the actions supported by the device and the textual string that is associated with a particular action. For example, should the string "*IncreaseInputVolume*" be received as an action name, the *Do_IncreaseInputVolume* method will be called.

```
const SERVICE_CTL c_rgSvc[c_cDemoSvc] =
{
        {
        "channel1",                     //Service control identifier.
        (PFNAI)Do_Channel_Init,         //Device-specific initialization function.
        (PFNAC)Do_Channel_Cleanup,      //Device-specific cleanup function.
        8,                              //Number of actions this service implements.
        {
                { "IncreaseInputVolume",        (PFNAS)Do_IncreaseInputVolume   },
                { "DecreaseInputVolume",        (PFNAS)Do_DecreaseInputVolume   },
                { "IncreaseInputTrim",          (PFNAS)Do_IncreaseInputTrim     },
                { "DecreaseInputTrim",          (PFNAS)Do_DecreaseInputTrim     },
                { "IncreaseOutputVolume",       (PFNAS)Do_IncreaseOutputVolume  },
                { "DecreaseOutputVolume",       (PFNAS)Do_DecreaseOutputVolume  },
                { "IncreaseOutputTrim",         (PFNAS)Do_IncreaseOutputTrim    },
                { "DecreaseOutputTrim",         (PFNAS)Do_DecreaseOutputTrim    },
    },
  },
};
```

Figure 6.21: A service entry from the service control table for the FB-88 device

The FB-88 module provides the implementation of the function prototypes illustrated in figure 6.21 mentioned above, and provides default values for the state variables associated with a device. These state variables are declared in a property list which maps a textual string to a default value, in a similar way to the service entry table illustrated in figure 6.21. Also included within the property list table is an indication of whether or not a particular variable is evented. All of the evented variables and their default values will be included within the initial event message upon the successful receipt of a subscription request message. Appendix E provides the code required for the implementation of this module.

As the implementation is based directly on the examples presented within the development kit, little discussion of the code associated with the implementation is included. Instead the interactions that occur between the control point and controlled devices are examined, and the specific messages illustrated. This provides a high-level overview of the workings of an implemented UPnP device, as well as providing examples of device and service description documents, and a presentation page. The code utilised for the implementation is, however, included in Appendix E.

6.2.1 FireBob Specification

The FireBob permits two modes of operation, providing varied configurations of the available analog audio input and output, MIDI input and output, optical input and output, monitor output, and word clock input and output ports. These ports and modes are illustrated by the front and rear panels of the FireBob which are shown in figures 6.22 and 6.23, respectively. The modes are known as the analog and ADAT modes, and are selectable via software. When the device is configured to work in analog mode the following facilities are available [Digital Harmony Technologies, Inc., 2000]:

- The 8 analog audio inputs are available,
- The 8 analog audio outputs are available, and
- The optical output port is configured for SPDIF output.

The ADAT operating mode provides the following facilities:

- The 8 ADAT audio inputs are available,
- The optical output is configured for ADAT output, providing 8 outputs channels, and
- The 8 analog outputs mirror the 8 ADAT outputs.

Source: FireBob Functional Specification [Digital Harmony Technologies, Inc., 2000]

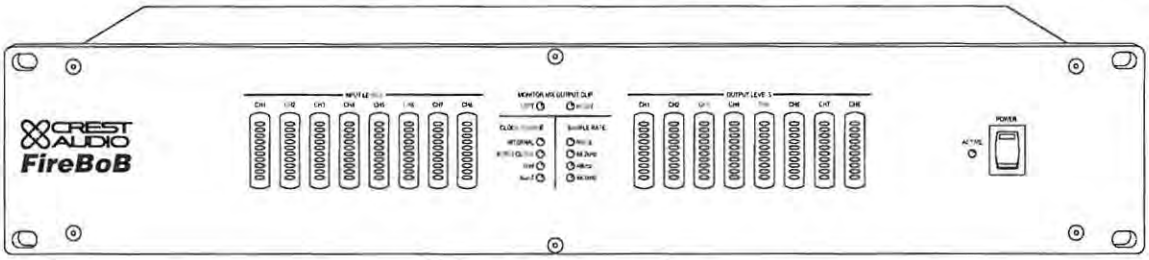

Figure 6.22: FireBob front panel

Source: FireBob Functional Specification [Digital Harmony Technologies, Inc., 2000]

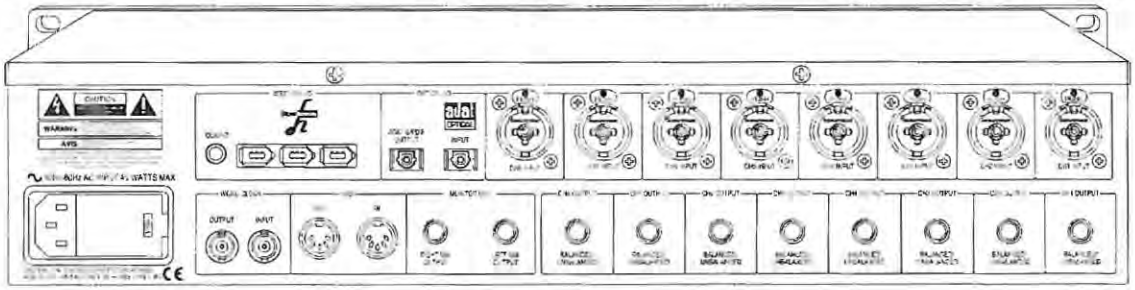

Figure 6.23: FireBob rear panel

The FireBob supports sampling rates for analog audio of 44.1 KHz, 48 KHz, 88.2 KHz, and 96 KHz at a resolution of 24 bits. Control of volume and trim levels over audio inputs and outputs for each channel is also provided. To facilitate synchronisation the FireBob supports the use of the following software-selectable clock sources depending on the current mode of operation:

- ADAT (ADAT mode only),
- Word clock (ADAT and analog modes),
- 1394 (analog mode only), and
- Internal 44.1 KHz, 48 KHz, 88.2 KHz, or 96 KHz (analog mode only).

6.2.2 Addressing

As both the control point and controller were PC-based applications utilising IP over Ethernet as the transport mechanism, the IP address assignment was via a DHCP server attached to the network. In addition, a DNS server was present on the network which provided a textual name to IP address lookup service, thereby allowing all networked devices to be targeted via their assigned textual names. The advantages of utilising a DHCP and DNS server are discussed in section 6.1.1. It was therefore unnecessary to provide any custom code for the IP addressing.

6.2.3 Description

The description phase requires the formation of the appropriate device and service description documents. The description document provides static information about the simulated device, such as the manufacturer and device serial number, as well as providing location pointers to further information pertaining to the services supported by the device. These location pointers provide the URLs for the service descriptor documents which provide the specifics about the services supported. A separate service descriptor document is provided for each service supported by the device. As the simulated device consists primarily of audio channels, it was modelled using this structure with every channel being declared as a service. Extracts from the device and service descriptor documents illustrating these declarations are provided in figure 6.24 and 6.25, respectively. There is a separate service descriptor document for each one of the eight audio channels supported by the FireBob. Defined within these services are the actions, as shown in figure 6.26, that are supported by the channels, and these include the incrementing and decrementing of:

- Input volume,
- Output volume,
- Input trim,
- Output trim,
- Monitor volume,
- Monitor trim, and
- Monitor mix.

Although there are a number of other controllable facilities provided by the FireBob, this implementation is considered as a proof of concept and the additional features can easily be added to the established framework.

```
- <service>
    <serviceType>urn:schemas-upnp-org:service:channel1:1</serviceType>
    <serviceId>urn:upnp-org:serviceId:channel1</serviceId>
    <controlURL>../control/isapictl.dll?channel1</controlURL>
    <eventSubURL>../control/isapictl.dll?channel1</eventSubURL>
    <SCPDURL>../SCPD/FB88Channel1-SCPD.xml</SCPDURL>
  </service>
```

Figure 6.24: Channel service element found in device description document

```
- <stateVariable>
    <name>InputVolume</name>
    <dataType>i4</dataType>
  - <allowedValueRange>
      <minimum>0</minimum>
      <maximum>255</maximum>
      <step>1</step>
    </allowedValueRange>
    <defaultValue>0</defaultValue>
  </stateVariable>
```

Figure 6.25: InputVolume state variable declaration for a particular channel

```
- <action>
    <name>IncreaseInputVolume</name>
  </action>
- <action>
    <name>DecreaseInputVolume</name>
  </action>
```

Figure 6.26: Action declarations to increase and decrease the InputVolume state variable for a particular channel

The full device and service description documents are provided in appendix E.

6.2.4 Discovery

The initialization of the controlled device involves the parsing of the device description document as the event sources are constructed for each declared service and the default values initialized. These event sources are utilized for the transmission of change notifications should a variable within the event source be modified. Once this initialization phase is complete, the controlled device transmits a number of presence announcements, advertising the root device and the supported services. An example of one of these announcements is provided in figure 6.27.

```
NOTIFY * HTTP/1.1
Host:239.255.255.250:1900
NT:urn:schemas-upnp-org:service:channel1:1
NTS:ssdp:alive
Location:http://128.13.94.8/fb88/description/fb88-orig.xml
USN:uuid:D9FD5726-5310-4127-9F12-4459367F0F94::urn:schemas-upnp-org:service:channel1:1
SERVER: Windows NT/5.1 UPnP/1.0 DevKit Sample/1.0
Cache-Control:max-age=600
```

Figure 6.27: Example service presence announcement for channel 1

Control points that are inserted into the network subsequent to these announcements, send multicast discovery request messages to all UPnP devices attached to the network. Figures 6.28 and 6.29 illustrate discovery request and response messages respectively, which are based on the unique identification number assigned to the simulated FireBob device. For example, the discovery request message specifies the unique identification number (*uuid*) of the targeted device in the search target (ST) field. Other permissible search target values can be specified according to device type, service type, all root devices, or all devices.

```
M-SEARCH * HTTP/1.1
Host:239.255.255.250:1900
ST:uuid:D9FD5726-5310-4127-9F12-4459367F0F94
Man:"ssdp:discover"
MX:3
```

Figure 6.28: Discovery search request

The discovery response message is an HTTP 200 OK message, and contains the search target that was specified in the request. It should be noted that the discovery response closely resembles an advertisement message, and can be interpreted by the control point as such. Therefore, the duration for which the advertisement is valid is specified by the *max-age* directive.

```
HTTP/1.1 200 OK
ST:uuid:D9FD5726-5310-4127-9F12-4459367F0F94
USN:uuid:D9FD5726-5310-4127-9F12-4459367F0F94
Location:http://128.13.94.8/fb88/description/fb88-orig.xml
SERVER: Windows NT/5.1 UPnP/1.0 DevKit Sample/1.0
EXT:
Cache-Control:max-age=600
DATE: Thu, 09 Jan 2003 08:44:53 GMT
```

Figure 6.29: Discovery search response

6.2.5 Eventing

Should the control point require notification of changes to the state variables present within the controlled device, the control point will transmit subscription requests, which are targeted at the controlled device. An example of a subscribe message is provide in figure 6.30, where notifications are requested for alterations to state variables associated with the service named *channel 1*.

```
SUBSCRIBE /fb88/control/isapictl.dll?channel1 HTTP/1.1
NT: upnp:event
Callback: <http://146.231.126.19:5000/notify>
Timeout: Second-1800
User-Agent: Mozilla/4.0 (compatible; UPnP/1.0; Windows NT/5.1)
Host: 146.231.126.18
Content-Length: 0
Pragma: no-cache
```

Figure 6.30: Example subscription request message for notifications of alterations to state variables associated with the service name channel 1

The controlled device will, upon the receipt of a valid subscription request, update the event list associated with the service and provide an appropriate subscription response message, as illustrated in figure 6.31, which is followed by the initial event message. The subscription response message contains the subscription identification number assigned by the controlled device and the timeout value associated with this subscription.

```
HTTP/1.1 200 OK
DATE: Windows NT/5.1 UPnP/1.0 DevKit Sample/1.0
SERVER: Thu, 09 Jan 2003 12:49:53 GMT
SID: uuid:015a7289_d28_2
Timeout: Second-1800
```

Figure 6.31: Subscription response message

The initial event message transmitted, which is illustrated in figure 6.32, provides the currently assigned values of all of the state variables provided by the service. The body of the message is formatted using XML notation, as discussed in section 6.1.5, and the sequence field (*SEQ*) contains the value 0 indicating an initial event message.

```
NOTIFY /notify HTTP/1.1
Content-Type: text/xml
NT: upnp:event
NTS: upnp:propchange
SID: uuid:015a7289_d28_2
SEQ: 0
User-Agent: SSDP CD Events
Host: 146.231.126.19:5000
Content-Length: 614
Cache-Control: no-cache

<e:propertyset xmlns:e="urn:schemas-upnp-org:event-1-0">
 <e:property>
  <e:InputVolume>
   0
  </e:InputVolume>
 </e:property>
 <e:property>
  <e:InputTrim>
   0
  </e:InputTrim>
 </e:property>
</e:propertyset>
```

Figure 6.32: An extract of an initial event message

Any alterations to the state variables within the service will result in event messages being transmitted from the controlled device to all control points that have registered to receive these notifications. These notifications resemble the initial event message, with the exception that the sequence number is incremented accordingly, and the body of the message contains only the values of the altered variables, as illustrated in figure 6.33.

```
<e:propertyset xmlns:e="urn:schemas-upnp-org:event-1-0">
 <e:property>
  <e:InputVolume>
   1
  </e:InputVolume>
 </e:property>
</e:propertyset>
```

Figure 6.33: The body of an event message

6.2.6 Control

Control is exerted over the controlled device through the invocation of actions associated with services. These actions control state variables and can alter the values stored within them, thereby controlling the service or device. As discussed in section 6.1.4, control messages utilise SOAP for the invocation of actions, an example of

which is illustrated in figure 6.34. This example illustrates the SOAP envelope associated with a request to invoke the *IncreaseInputVolume* action associated with the *channel1* service.

```
Request URI: channel1
Request Body: <?xml version="1.0"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
        <m:IncreaseInputVolume xmlns:m="urn:schemas-upnp-org:service:channel1:1"/>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 6.34: Example SOAP request

If the SOAP request is successful, the actual method within the FB88 module associated with the action will be executed and an appropriate response returned. If this method updates any of the state variables, a change notification message will be sent to all registered control points. This change notification message will resemble the event messages discussed in section 6.2.4 and depicted in figure 6.33.

6.2.7 Presentation

Provided within the device description document is a URL specifying the location of a presentation page, in HTML format, that is associated with the simulated device. This page provides all of facilities required by a control point allowing a basic web browser to act as the control point for the FireBob. The presentation page created for the FireBob is illustrated in figure 6.35. The HTML code and embedded script can be found in appendix E.

The presentation page contains embedded VB script that is able to utilise UPnP architecture components present within Microsoft Windows operating systems that are UPnP enabled, for example Windows XP. The components provided are the SSDP and the Control Point Component. The SSDP component listens for UPnP presence announcements. When an announcement is received the SSDP component retrieves the device description document from the advertising device, and notifies the user of the availability of a new UPnP device. The user is then able to view the details extracted from the description document, or invoke the presentation page

associated with the device. The URL of this page is present within the description document.

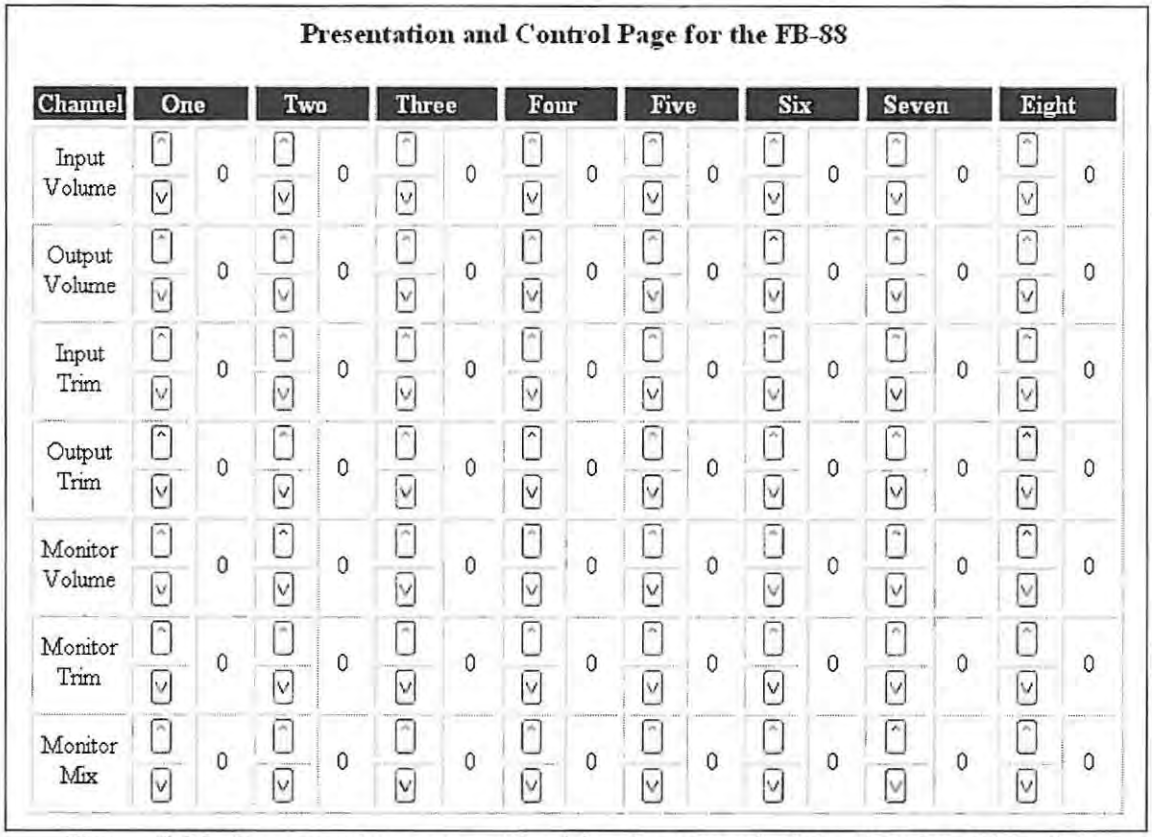

Figure 6.35: Presentation page utilised for the control of the Simulated FireBob

The Control Point Component is utilised within the web browser, although it can be utilised within a custom written application. The Control Point Component is accessed through the use of defined interfaces which are provided in table 6.1. The entries in bold indicate the interfaces that were utilised in this implementation.

Source: Microsoft Developer Network – Platform SDK [Microsoft Corporation, 2000]

Interface	Description
IUPnPDescriptionDocument	Enables an application to load a device description.
IUPnPDescriptionDocumentCallback	Enables an application to receive the results of an asynchronous load operation.
IUPnPDevice	Enables an application to retrieve information about a specific device.
IUPnPDeviceDocumentAccess	Enables an application to obtain the URL of a device description document.
IUPnPDeviceFinder	Enables an application to find a device.
IUPnPDeviceFinderCallback	Enables an application to receive asynchronous search results from UPnP.
IUPnPDevices	Enumerates a collection of devices.
IUPnPService	Enables an application to retrieve state information and invoke actions for a service.
IUPnPServiceCallback	Enables an application to receive notification from UPnP when events occur.

IUPnPServices	Enumerates a collection of services.

Table 6.1: Control Point Component Interfaces

The facilities provided by the Control Point Component include the following:

- Find available UPnP devices on the network (*discovery*)
- Determine which of these devices are of interest (*description*)
- Issue control requests and receive events (*control and eventing*)

The first task undertaken by the web browser upon the loading of the presentation page, as indicated by the script embedded within the HTML page, is the instantiation of an instance of the UPnP Device Description object. The device description document is retrieved and loaded into this object. Figure 6.36 provides the extract of script required to create this object and load the document.

```
' Download Description Document
Dim ChannelDesc
Set ChannelDesc = CreateObject("UPnP.DescriptionDocument.1")
ChannelDesc.Load("..\description\FB88-orig.xml")
```

Figure 6.36: Device Description object creation

From the Device Description object the root device associated with the simulated device can be obtained, which provides access to the services supported by the device. The script required to retrieve a reference to the root device is illustrated in figure 6.37.

```
'Get the Root Device
Dim FB88
Set FB88 = ChannelDesc.RootDevice
```

Figure 6.37: Obtaining a reference to the root device

Services are accessed according to a service identifier, which is specified as a string. This is illustrated in figure 6.38 where the *urn:upnp-org:serviceId:channel1* service is referenced. Although the entire string identifies the service, the *channel1* portion of the string corresponds to the declaration present within the *UDevice* module, as illustrated in figure 6.21. In order to handle notifications from the simulated device an event handler is required. This event handler is implemented as a callback routine which will be called upon the arrival of any notification message. An example of a notification is the initial event message, which will set the default values of the variable parameters on the web page.

```
'Attach the event handler to this service
```

```
Dim ChannelService
set ChannelService=FB88.Services("urn:upnp-org:serviceId:channel1")
ChannelService.AddCallback GetRef("eventHandler")
```

Figure 6.38: Service selection and event handler registration

Two buttons are provided for each modifiable parameter, allowing for the relevant increase or decrease action to be invoked. Associated with the service selected is an *InvokeAction* method which generates the necessary SOAP request message to invoke the correct method within the controlled device. The *InvokeAction* method, illustrated in figure 6.39, can accept input and output parameters allowing for a specific value to be passed, as an argument, to the action invoked. The controlled device will respond to the SOAP request and generate an event notification message, as the value of one of the state variables is likely to be altered by the action invoked. This message will be received by the registered callback, and the appropriate field on the presentation page updated. This mechanism allows for multiple control points to be utilised, as each one will receive a similar event notification message, and their presentation pages will be updated.

```
Dim inArgs(1)
Dim outArgs(0)

inArgs(0) = 1
ChannelService.InvokeAction "IncreaseInputVolume", inArgs, outArgs
```

Figure 6.39: Action invocation

6.2.8 Evaluation of the UPnP Architecture

Due to the large amounts of textual data transmitted during the various phases defined within the UPnP architecture, it is most likely not immediately suitable for use within the professional audio industry where large amounts of data, such as the transmission of automation information to a mixer, are required. For example, professional audio installations can have in excess of 20,000 variable parameters [Rosenthal, M., 20/12/2001, pers. com.]. A simple UPnP SOAP control message for the simulated FireBob consists of 618 bytes of data, and a TCP/IP header length of 40 bytes. This amounts to a total of 658 bytes for the control message, without the required transportation level headers, such as Ethernet or IEEE 1394. The length of these headers for the IPv4 over IEEE 1394 implementation, discussed in chapter 5, consists of 4 bytes for an unfragmented header and 20 bytes for the asynchronous block write

header. Therefore the total amount of data required to be transmitted for every control message is 682 bytes.

One of the requirements for IEEE 1394 nodes to provide IP capability is the ability to transmit and receive at least 512 bytes of data. As the length of the control message is greater than this value, control messages transmitted to these nodes will have to be fragmented. These fragmentation procedures, discussed in chapter 5, require an additional 4 bytes of header for every transmission, and would require two transmissions per control message. The first transmission could be 512 bytes in length, and the subsequent transmission 200 bytes. Therefore, the total amount of data to be transmitted, if fragmentation is required, is 714 bytes, as illustrated in figure 6.40.

Header	Length in Bytes	
1394 asynchronous block write header	20	First fragment
First fragment encapsulation header	8	
Data payload	484	
1394 asynchronous block write header	20	Last fragment
Last fragment encapsulation header	8	
Data payload	172	
Total length of data transmitted	**714**	

Figure 6.40: Fragmented control message length

Assuming, however, the control message could be transmitted encapsulated within an unfragmented packet, this would require approximately 20,000 control messages to achieve a scene update that affected all variable parameters. This would require the transmission of approximately 13 megabytes of information. Considering that currently available IEEE 1394 networks currently operate at around 400 Mbps, with only 20% of the available bandwidth (10 Megabytes per second) allocated to asynchronous transactions, it would take approximately 1.3 seconds for the transmission of all update messages. This theoretical value does not account for the delays encountered by control points and the controlled device to generate and process these messages.

This delay could be reduced by allowing multiple control messages to be embedded within one SOAP envelope, as this is currently restricted to one message. This would reduce the overhead associated with each transmission. The textual strings within the SOAP envelopes could also be transmitted as binary data instead of text. This would

require the formation of suitable encoding rules, and reduce the amount of data transmitted. New IEEE 1394 developments are improving the transmission speeds, which would allow a greater amount of data to be transmitted in real-time.

6.3 Summary

This chapter examined the UPnP architecture in detail, and provided a high-level overview of the implementation of an UPnP enabled controlled device. The device simulated was a professional audio device known as the FireBob or FB-88. This implementation demonstrated the features associated with the UPnP architecture and its applicability for use within audio installations. The simulated device was able to operate over an IP-capable network, such as Ethernet or IEEE 1394.

The UPnP architecture satisfies all of the criteria for a professional installation, which was identified in chapter 4, although a large amount of textual data is required to be transmitted. These large transmissions hinder the applicability of the UPnP architecture to professional installations as significant delays are introduced. Improvements to the protocols utilised within the UPnP architecture and the ability to transmit binary encoded data could reduce the amount of data transmitted and the associated overheads.

Chapter Seven
Conclusion

7.1 Overview

A number of currently available sound system technologies have been analysed and discussed, each providing their own unique approach to sound installations. However, the majority of these approaches are proprietary and will not interoperate with devices from other manufacturers. This is especially evident in the discussions on the control and monitoring of these devices. The analysis of currently available technologies provided an object model of the facilities common to all protocols examined. This model yielded a number of evaluation criteria against which control and monitoring protocols could be measured. These criteria include description, discovery, control, monitoring, presentation, connection management, latency management, and synchronisation.

The introduction of IEEE 1394 to professional audio devices has alleviated some of these interoperability problems, as standards have been specified for the transport of audio between devices. However, no adequate control and monitoring mechanisms have been provided for the IEEE 1394 bus that are suited to professional audio devices. Current control mechanisms provided over the IEEE 1394 bus include the AV/C and MIDI protocols.

As there are a number of IP-based control and monitoring protocols defined, which are currently in use within the sound installation industry, the approach adopted was to provide a mechanism to allow these protocols to be transported over the IEEE 1394 bus. Not only does this provide sufficient control and monitoring capabilities to IEEE 1394 networks, but allows the reuse of existing applications based on these protocols. An example of such a protocol is QSC-24. This led to the implementation of an IP stack above the firmware present on a Digital Harmony DHIVA device and the creation of an appropriate communication interface between the two.

It was desirable that this implementation be consistent with the IPv4 over IEEE 1394 implementation provided by the Microsoft Corporation in their Windows Operating System products, as the reuse of Windows IP-based applications was required. The approach adopted for the Windows implementation is based on RFC 2734, which provides:

- Packet encapsulation formats,
- Fragmentation and reassembly procedures,
- Multicasting advertisement and solicitations,
- ARP formats, and
- Defines appropriate IEEE 1394 transactions for the transfer of IP datagrams.

The same approach was adopted for the IPv4 over IEEE 1394 implementation on the embedded DHIVA device, ensuring that the DHIVA and PC can communicate using IP datagrams.

Although the implementation of an IP stack on the embedded device was primarily motivated by the provision of a control and monitoring mechanism suitable for professional audio devices, a vast number of other applications are possible using the same stack. For example, bridges can be created to bridge Ethernet and IEEE 1394 networks, allowing devices located on the IEEE 1394 bus access to the World Wide Web.

The implementation procedures required the definition of an IP-1394 device driver to facilitate communication between the firmware present on the DHIVA and the IP stack. This device driver was also responsible for the formation of RFC 2734 compliant packets, and contained the necessary fragmentation and reassembly routines.

The protocol selected to demonstrate the control and monitoring of audio devices using IP over the IEEE 1394 bus was QSC-24. A number of possible configurations were discussed, each with their distinct advantages and disadvantages. Example applications were developed to simulate a real audio device and the transmission of QSC-24 messages over the IEEE 1394 bus. However, the existing QSControl

software could be utilised for the transmission of control messages, and would function as if it were operating over an Ethernet network.

Although QSC-24 is based on public documentation, not many manufacturers utilise it for the control and monitoring of their devices. An alternative to this protocol is the UPnP architecture, and its viability for the control and monitoring of audio devices was explored. As the UPnP architecture is web server based and messages are transmitted in a textual format, the associated processing overhead is undesirable for professional audio devices. This would be evident should a device require that all of its parameters be altered or uploaded in real time. However, with improvements in the embedded environment, such as the increases in processing power and transmission speeds, the UPnP architecture might become a viable option for the transmission of control and monitoring data within sound installations. The protocols currently utilised within the UPnP architecture may be exchanged for more stream-lined versions that are suited to the IEEE 1394 bus.

7.2 The Hypothesis Revisited

The original hypothesis was that IP-based protocols could be utilised in conjunction with IEEE 1394 networks for the control and monitoring of professional audio devices within sound installations. The approach taken was to successfully implement an IP stack above an IEEE 1394-enabled development board. The IP-based protocols of QSC-24 and those present within the UPnP architecture were selected and compared to the AV/C and MIDI control and monitoring protocols currently available on the IEEE 1394 bus. It was found that the IP-based protocols were more suited to professional sound installations, and software already existed that could be reused within these 1394-based installations. Therefore the addition of IP-capability to IEEE 1394 devices does provide an adequate solution to control and monitoring within sound installations. In addition, this solution allows other IP-based control and monitoring protocols access to the IEEE 1394 bus. The solution does not, however, provide a unified control and monitoring protocol that all audio device manufacturers are expected to adopt.

7.3 The Unrealisable Ideal

The evaluation criteria identified provide the facilities that are desirable within control and monitoring protocols, and therefore should be taken into consideration in the design of new protocols. The IEEE 1394 architecture provides a number of these facilities inherently, although they are currently not exploited to their full potential. For example, IEEE 1394 devices are able to automatically discover other devices on the network (plug-and-play capabilities) and describe their capabilities through the use of the configuration ROM. It seems appropriate that these facilities be utilised to their full potential in the creation or selection of a control and monitoring protocol for professional audio devices that is suited specifically to the IEEE 1394 bus. However, the effective use of the IEEE 1394 bus is at the expense of interoperability with devices utilising other transport technologies. For example, sound installations utilising IEEE 1394-based technologies are not easily bridged to Ethernet-based installations.

The control and monitoring protocol utilised should be devised with the sole purpose of providing the facilities required by professional audio devices. This would entail defining the fundamental criteria required by these devices, while still allowing extensibility within the protocol to cater for future developments. This would require that manufacturers agree on a standardised protocol for use within IEEE 1394 networks, thereby providing unhindered interoperability between various devices. Unfortunately, history has proven that manufacturers are reluctant to abandon proprietary protocols for standardised efforts. AES-24 is a prime example of a standardised protocol that provided the required capabilities but was not widely adopted.

This fundamental approach is anticipated to provide for the most effective and efficient use of the bus. However, if the defined protocol is not adopted, it will provide no benefits. An alternative approach is to allow devices to describe themselves, and their capabilities. The UPnP architecture attempts to provide these capabilities, and succeeds in terms of discovery and description. Perhaps this is the approach that should be adopted for IEEE 1394 networks; where a control and monitoring architecture should be defined, and protocols either developed or selected that satisfy the evaluation criteria for each component of the architecture. The

challenge however is not the definition or selection of appropriate protocols but rather the motivation for audio device manufacturers to provide support for a standardised approach to control and monitoring, and the abandonment of proprietary solutions.

7.4 A Workable Approach

From a manufacturer point of view, an ideal solution would be one where devices are able to interoperate regardless of the control and monitoring protocol utilised. There are a number of approaches that could provide this facility. One approach is the provision of a central repository, as depicted in figure 7.1, which is connected to every device on the network. The repository abstracts these devices and exposes them as virtual devices to all control points. All control and monitoring related messages are then targeted at these virtual devices on the central repository and not the actual device. The advantage of this approach is that, provided the central repository supports the control and monitoring protocol utilised by a specific device, it can control that device. This approach could be abstracted to a higher level, by providing control and monitoring facilities to a subnet of similar devices that are able to interoperate and not for every individual device. In this context the repository is acting as a bridge between varying protocols, as illustrated in figure 7.2.

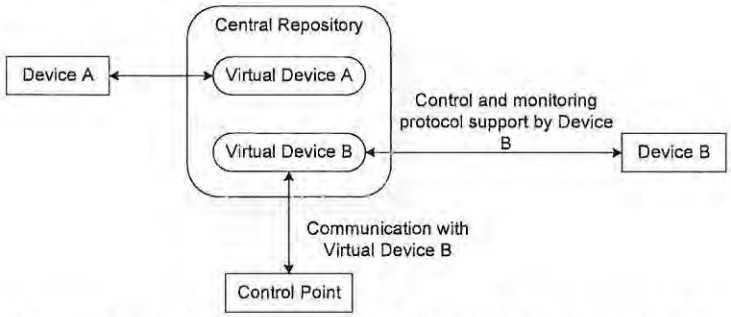

Figure 7.1: Central repository device based approach

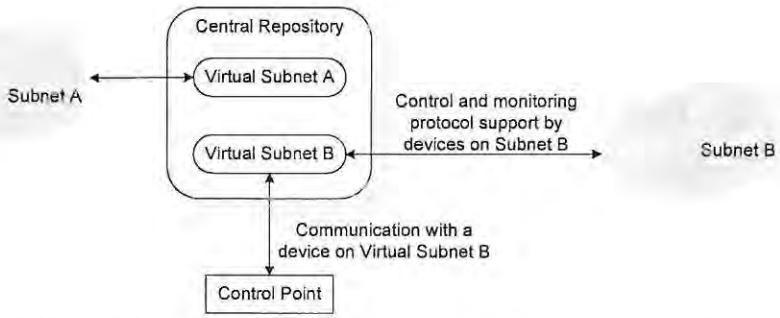

Figure 7.2: Central repository subnet based approach

A logical extension to this approach leads back to the proposal of an IEEE 1394 based control and monitoring protocol developed specifically for professional audio devices, provided that manufacturers wishing to provide interoperability support this protocol. This would permit manufacturers to utilise their current control and monitoring protocols within their subnet of devices, but would require them to convert this protocol to a standardised protocol for inter-subnet communication. Such a standardised protocol could simply be XML based, allowing for easy description, although these documents would require binary encoding to minimise overheads and length of transmissions.

It is possible that the repository or a control point support pluggable-modules for specific devices or a range of devices. These modules can be retrieved from a manufacturer's web site or from the actual device. This would require the definition of a standard API for interfacing with these modules, but the modules would provide all of the protocol specific messaging that would be required. A mechanism such as this would allow communication between the repository or control point and any attached device, provided an appropriate module is available.

A few suggestions are provided here, but there are many more possibilities. The solution finally adopted will be one where there is sufficient room for manufacturers to expand as new developments are made, and where competitive advantages can be maintained. This implies that manufacturers are to continue to utilise their proprietary technologies, and mechanisms should be devised to allow these technologies to be transported over the IEEE 1394 bus. This was the motivation for the provision of the IP over IEEE 1394 facilities described within this document. The way forward is to allow the existing control and monitoring protocols to interoperate.

References

1394 Trade Association, 1998: *AV/C Digital Interface Command Set – General Specification*. 1394 Trade Association.

1394 Trade Association, 1999: *AV/C Monitor Subunit Model and Command Set*. 1394 Trade Association.

1394 Trade Association, 2000: *AV/C Audio Subunit Specification 1.0*. 1394 Trade Association.

1394 Trade Association, 2000: *AV/C Command Set for Rate Control of Isochronous Data 1.0*. 1394 Trade Association.

1394 Trade Association, 2000: *AV/C Connection and Compatibility Management Specification 1.0*. 1394 Trade Association.

1394 Trade Association, 2000: *AV/C Connection and Compatibility Management Specification 1.0*. 1394 Trade Association.

1394 Trade Association, 2000: *AV/C General – Descriptor and Info Block Mechanism*. 1394 Trade Association.

1394 Trade Association, 2001: *AV/C Music Subunit 1.0 – Draft*. 1394 Trade Association.

1394 Trade Association, 2001: *AV/C Panel Subunit Specification 1.1*. 1394 Trade Association.

Anderson, D., 1999: *FireWire System Architecture: Second Edition*. MindShare Inc., USA.

Audio Engineering Society, 1997: *AES Standard for Sound System Control – AES-24 Parts 1 and 2.*

Comer, D., 1987: *Internetworking with TCP/IP.* Prentice-Hall, USA.

Crown Audio, Inc., 1996a: *FE-FF Protocol.*

Crown Audio, Inc., 1996b: *IQ Development Standards.*

Crown Audio, Inc., 2001a: *IQ for Windows 4.1.* Online: available at http://www.iqaudiosystems.com/.

Crown Audio, Inc., 2001b: *IQ for Windows Help.*

Crown Audio, Inc., 2001c: *IQNet Server Help.*

Crown Audio, Inc., 2001d: *IQNet Server Product Description.* Online: available at http://www.iqaudiosystems.com/IQNetServer.htm.

Crown Audio, Inc., 2001e: *The IQ Bus.* Online: available at http://www.iqaudiosystems.com/IQBus.htm.

Digital Harmony Technologies, Inc., 2000: *FireBob 1394 Interface Specification.* Digital Harmony Technologies, Inc.

Digital Harmony Technologies, Inc., 2000: *FireBob Functional Specification – version 0.96.* Digital Harmony Technologies, Inc., USA.

Express Logic, Inc., 2000: *ThreadX – The High-performance Embedded kernel – User Guide.* Express Logic, Inc., USA.

Fujimori J. and Foss R., 2002: *mLAN: Current Status and Future Directions.* 113th AES Convention, Los Angeles.

Green Hills Software, Inc., 1999: *Using MULTI 2000 with the ThreadX Real-Time Kernel*. Green Hills Software, Inc., USA.

Green Hills Software, Inc., 2001: *The MULTI 2000 and AdaMULTI Integrated Development Environments*. Green Hills Software, Inc., USA. Online: available at http://www.ghs.com/products/ide.html.

Gross, K., 2000: *Digital Audio Distribution Systems*.

Harmony Central, 1999: *Midiman Ships DigiPatch 12x6 Audio Patchbay*. Online: available at http://www.harmony-central.com/Newp/WNAMM99/Midiman/DigiPach-12x6.html.

Huntington, J. 1994: *Control Systems for Live Entertainment*. Boston, Mass.: Focal Press.

Huntington, J., 2000: *Control Systems for Live Entertainment – Second Edition*. Focal Press, USA.

IEC 61883-1, 1998: *Digital Interface for Consumer Audio/Video Equipment – Part1: General*. International Electrotechnical Commission.

IEC 61883-6, 1998: *Consumer Audio/Video – Digital Interface – Part 6: Audio and music data transmission protocol*. International Electrotechnical Commission.

IEEE Computer Society, 1996: *IEEE Standard for a High Performance Serial Bus (IEEE Std 1394-1995)*. Institute of Electrical and Electronics Engineers, Inc., USA.

IEEE Computer Society, 2000: *Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications*. Institute of Electrical and Electronics Engineers, Inc., USA.

IEEE Computer Society, 2000: *IEEE Standard for a High Performance Serial Bus – Amendment 1 (IEEE Std 1394a-2000).* Institute of Electrical and Electronics Engineers, Inc., USA.

IEEE Computer Society, 2002: *IEEE Standard for a High Performance Serial Bus Amendment 2.* Institute of Electrical and Electronics Engineers, Inc., USA.

IMA, 1989: *MIDI 1.0 Detailed Specification, Document Version 4.1.* The International MIDI Association.

ISO/IEC, 1994: *Control and Status Registers (CSR) Architecture for microcomputer buses (ISO/IEC 13213:1994).* International Standards Organisation/International Engineering Consortium.

Johannson, P., 1999: *IPv4 over IEEE 1394 (RFC 2734).* The Internet Society.

Laubscher, R., Moses, B., Foss, R., 2000: *A 1394-Based Architecture for Professional Audio Production.* AES-109th Convention, Los Angeles.

Microsoft Corporation, 2000: *Microsoft Developer Network Library – Platform SDK.* Microsoft Corporation, USA.

Microsoft Corporation, 2000: *Universal Plug and Play Architecture.* Microsoft Corporation, USA.

Microsoft Corporation, 2001: *Microsoft UPnP Development Kit.* Microsoft Corporation, USA.

Pacific Softworks, Inc., 1999: *Fusion Porting Guide.* Pacific Softworks, Inc., USA.

Peak Audio Inc., 2001a: *CobraCad Help.*

Peak Audio Inc., 2001b: *CobraNet Competitive Analysis.* Online available at http://www.peakaudio.com.

Peak Audio Inc., 2001c: *CobraNet Technology Datasheet.*

Peavey Electronics Corporation, 1996: *MediaMatrix MM-8800 Series.* Online: available at http://mediamatrix.peavey.com/.

Peavey Electronics Corporation, 1999: *CAB 8i and CAB 8o.* Online: available at http://mediamatrix.peavey.com/.

Peavey Electronics Corporation, 2001a: *External Control via the Break-Out-Box.* Online: available at http://mediamatrix.peavey.com/pdf_library.html.

Peavey Electronics Corporation, 2001b: *How MediaMatrix Works.* Online: available at http://mediamatrix.peavey.com/how_mediamatrix_works.html.

Peavey Electronics Corporation, 2001c: *MediaMatrix.* Online: available at http://mediamatrix.peavey.com/.

Peavey Electronics Corporation, 2001d: MediaMatrix 3.2 On-line Help. Online: available at http://www.mediamatrix.peavey.com/mmhelp/mmhelp.htm.

Peavey Electronics Corporation, 2001e: *MediaMatrix MM-DSP Series DPU Boards.* Online: available at http://mediamatrix.peavey.com/mm-dsp-cn.html.

Peavey Electronics Corporation, 2001f: *MediaMatrix X-Frame.* Online: available at http://mediamatrix.peavey.com/pdf_library.html.

Peavey Electronics Corporation, 2001g: *MM-8802 MediaMatrix Break-Out-Box (Bob) – User Manual.* Online: available at http://mediamatrix.peavey.com/mm-8802.html.

QSC Audio Products, Inc., 1997a: *QSControl2 Help.*

QSC Audio Products, Inc., 1997b: *QSC-24 Application Protocol.*

QSC Audio Products, Inc., 1998: *Rave Application Guide*. Online: available at http://www.qscaudio.com/support/library/guides/ravegide.pdf.

QSC Audio Products, Inc., 2000a: *QSControl Software V.3.0*. Online: available at http://www.qscaudio.com/pdfs/cm16soft.pdf.

QSC Audio Products, Inc., 2000b: *CM16a Amplifier Network Monitor*. Online: available at http://www.qscaudio.com/pdfs/cm16spc.pdf.

QSC Audio Products, Inc., 2001a: *QSControl*. Online: available at http://www.qscaudio.com/products/amps/qscontrol/qsintro.htm.

QSC Audio Products, Inc., 2001b: *CM16a Features*. Online: available at http://www.qscaudio.com/products/amps/qscontrol/cm16.htm.

Rosenthal, M., 20/12/2001: *Control and Monitoring Thread*. Personal Communication.

Rumsey, F. and Watkinson, J., 1995: *The Digital Interface Handbook – Second Edition*. Focal Press, Great Britain.

World Wide Web Consortium, 2000: *Extensible Markup Language (XML) 1.0 (Second Edition)*.

Yamaha Corporation, 1997: *Yamaha O3D Digital Mixing Console – Owners Manual*. Yamaha Corporation.

Appendix

A

UML Model of a typical Sound System

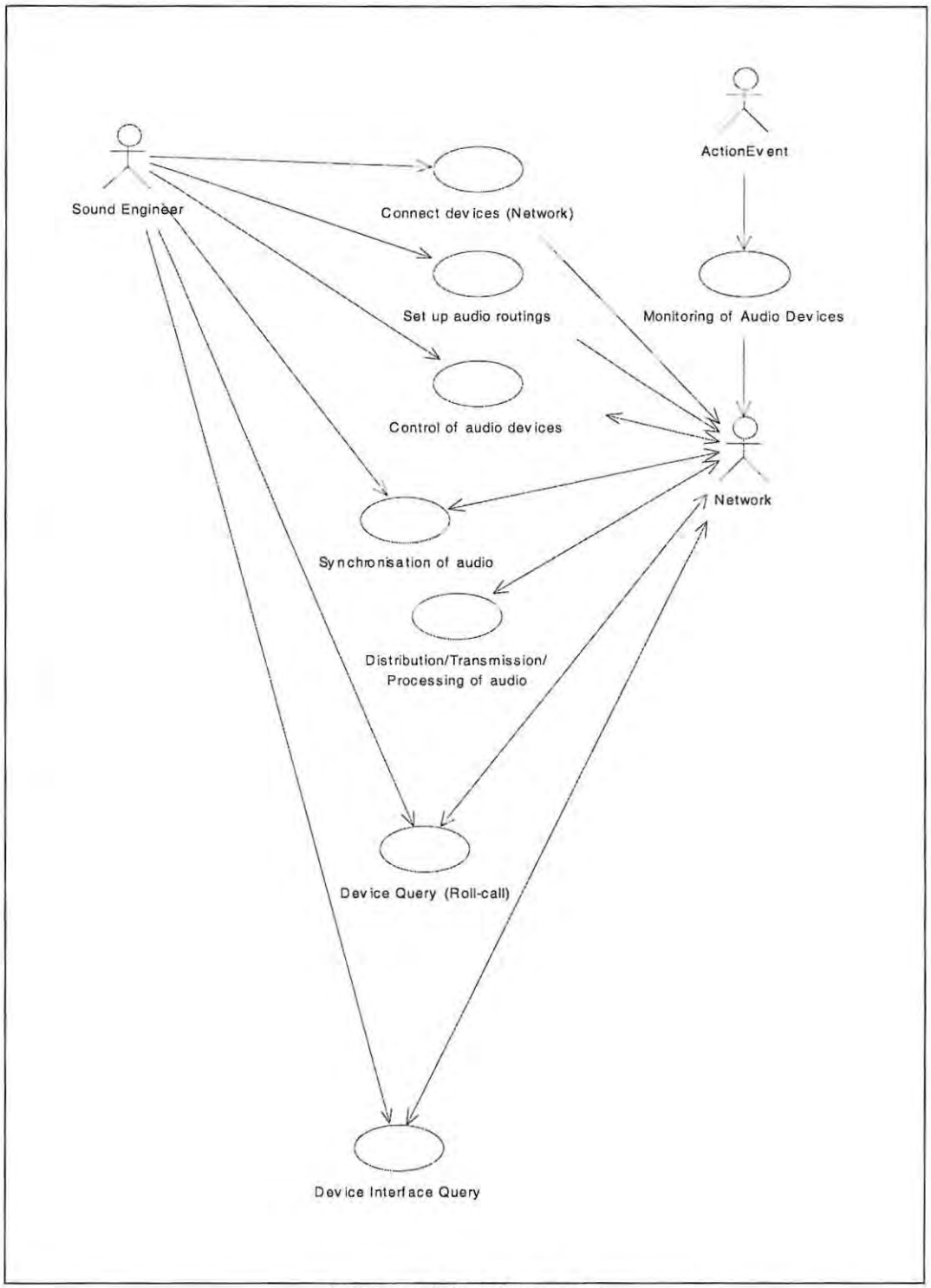

Figure A.1 Use Case view

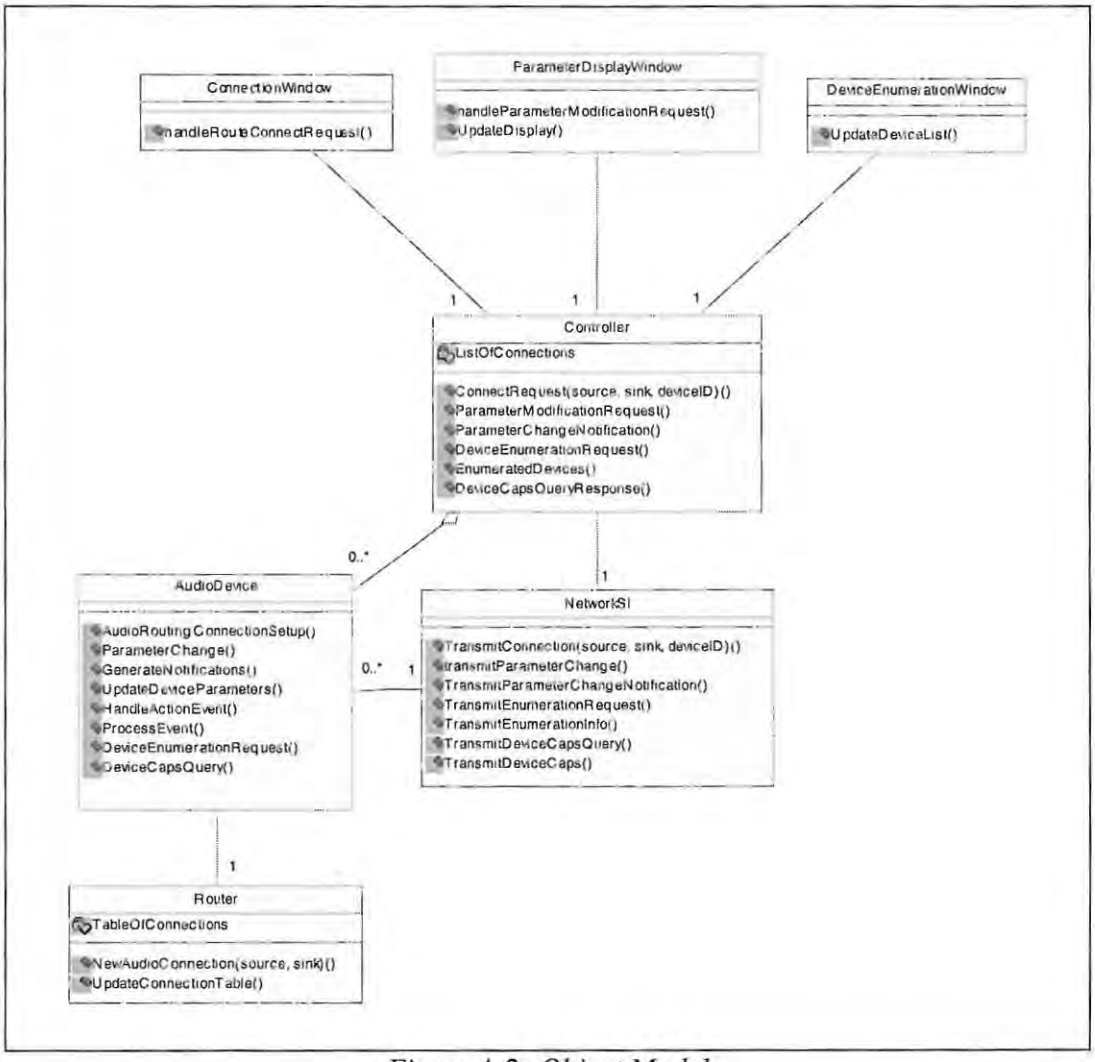

Figure A.2: Object Model

Textual Scenarios

Network Roll-call

- The user instructs the controller to perform a system wide roll-call in order to determine all of the device connected to the network.
- The roll call message is received by all connected device, and is processed.
- The device generates an appropriate response, and transmits the response to the controller.

Device Capabilities Query

- The controller may issue a device capabilities query for 3 reasons:
    - Device startup/network initialisation
    - Bus reset
    - At the users request

- The device capabilities query message is generated by the controller, and directed at the appropriate device.
- The message is received and deciphered by the device
- The device queries itself (units/subunits) for a list of device, and forms an appropriately formatted message.
- The device transmits this message to the controller.

Set up Audio Routings (IEEE 1394)
- The user selects a source port, and a sink port (device independent) and clicks a connect button.
- A connect message (Eg. AV/C) is formed, containing the selected ports, and is sent to the associated devices.
- The devices process this message and perform the necessary internal registers updates to initiate the routing.

Control of Devices (Single)
- The user is presented with an appropriate interface that is supported by a device that is to be controlled.
- A user-controllable parameter presented on this interface is altered.
- A parameter-change message is formulated and sent to the relevant device.

Control of Devices (Multiple)
- The user is presented with an appropriate interface that is supported by a number of devices that are to be controlled.
- A user-controllable parameter presented on the interface, and linked to multiple devices, is altered.
- A parameter-change message is formulated and is:
    - Broadcast to all devices
    - Multicast to the relevant devices listening on the multicast channel
    - Unicast to the relevant devices (multiple message each modified appropriately)

Monitoring of Devices
- A device has been setup (through an appropriate control message, or by default) to report monitoring information to the controller.
- The device formulates the appropriate message gathering the required information.
- This message is sent to the controller.

User Interface Query
- The controller issues a user interface query message to a selected device
- The device receives this message, and checks to see if this facility is supported.
- The device forms an appropriate response message, which is transmitted to the controller.

NB: The response could be in the form of a bitmap detailing the interface, but more preferably in the form of an xml document (possibly accompanying an html document) describing the interface.

Sequence Diagrams

All of the sequence diagrams can be broken down into the action from the viewpoint of the controller and the audio device.

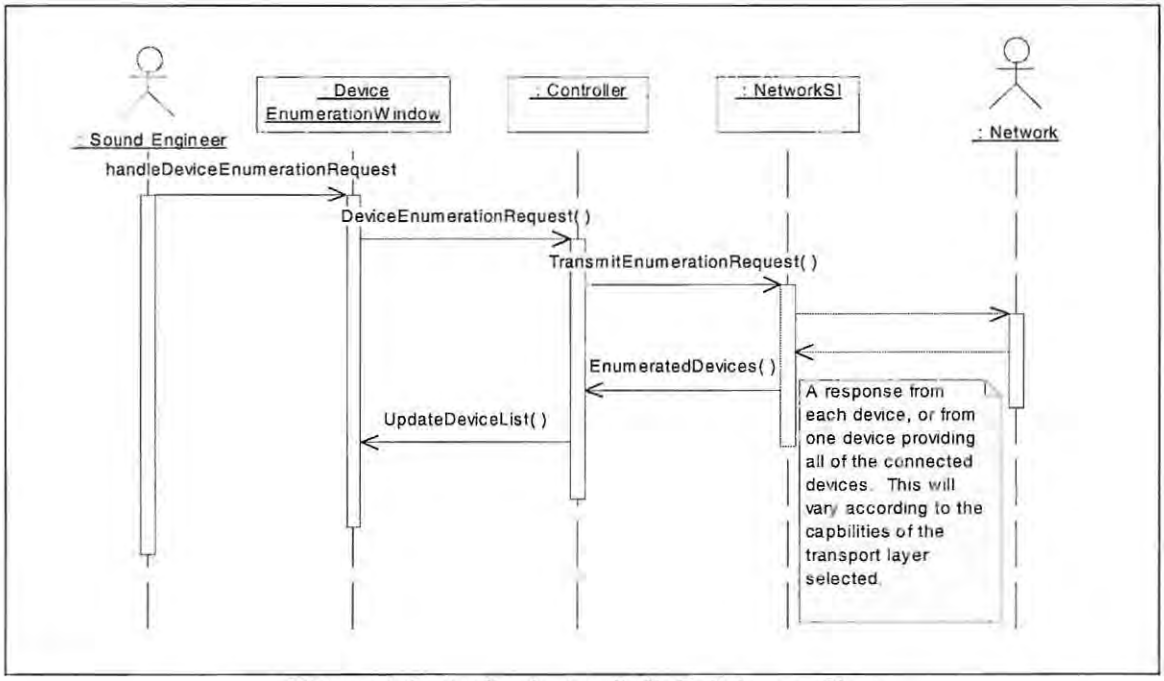

Figure A.3: Audio device Roll-Call (controller)

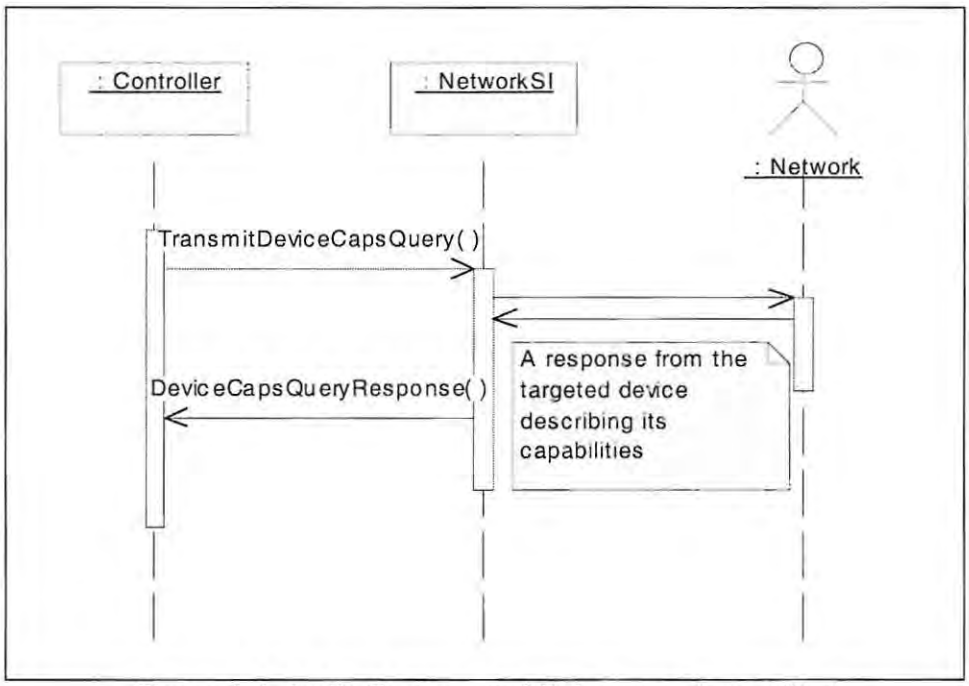

Figure A.4: Audio Device capabilities query (controller)

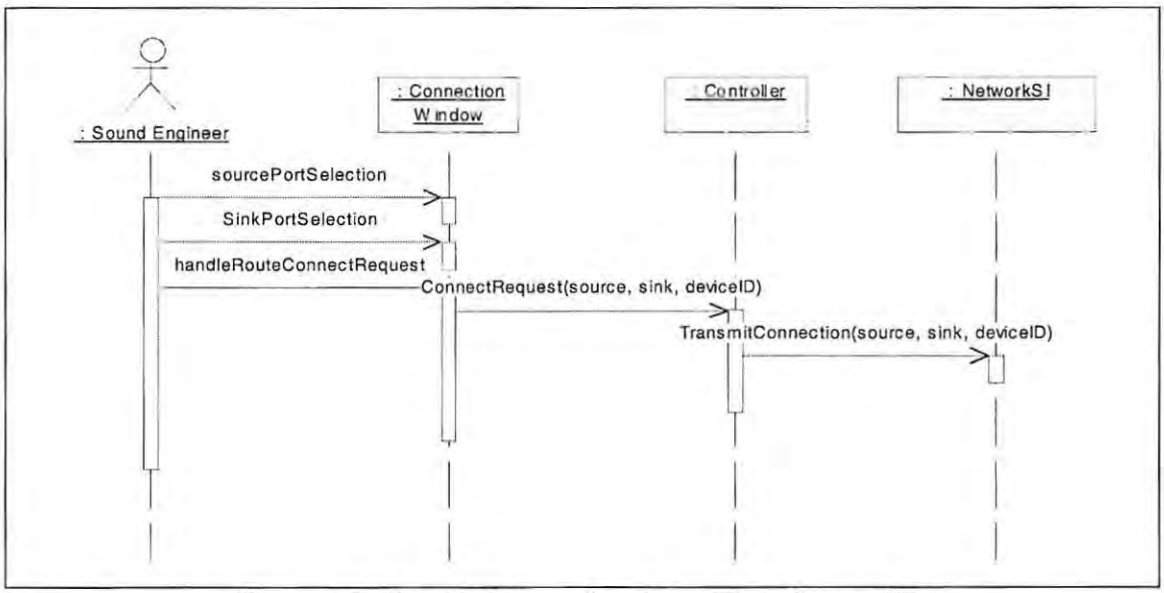

Figure A.5: Establishment of audio routings (controller)

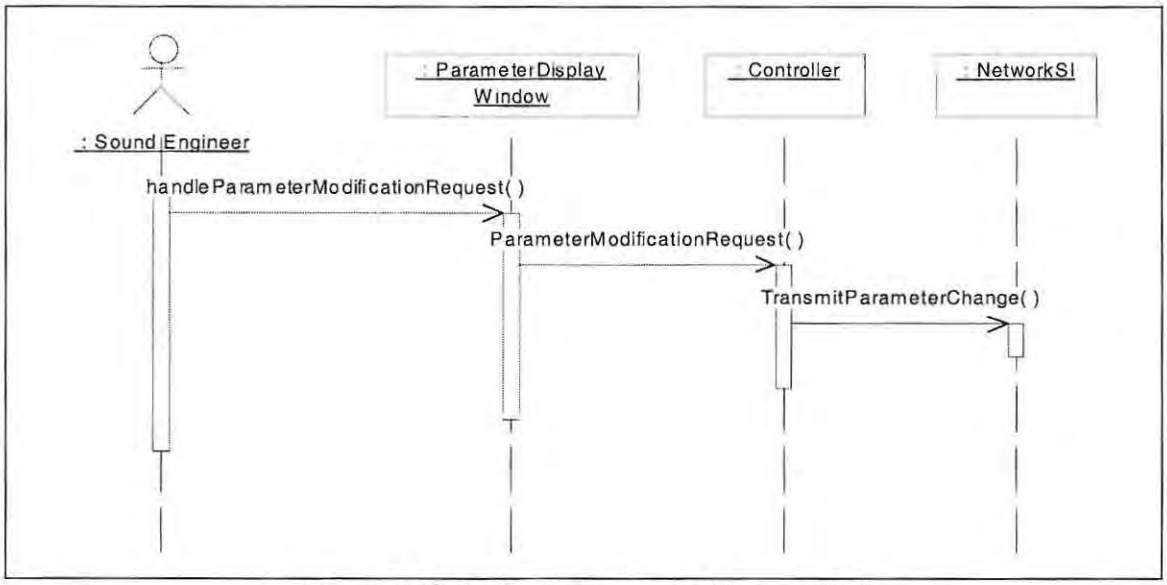

Figure A.6: Audio device control (controller)

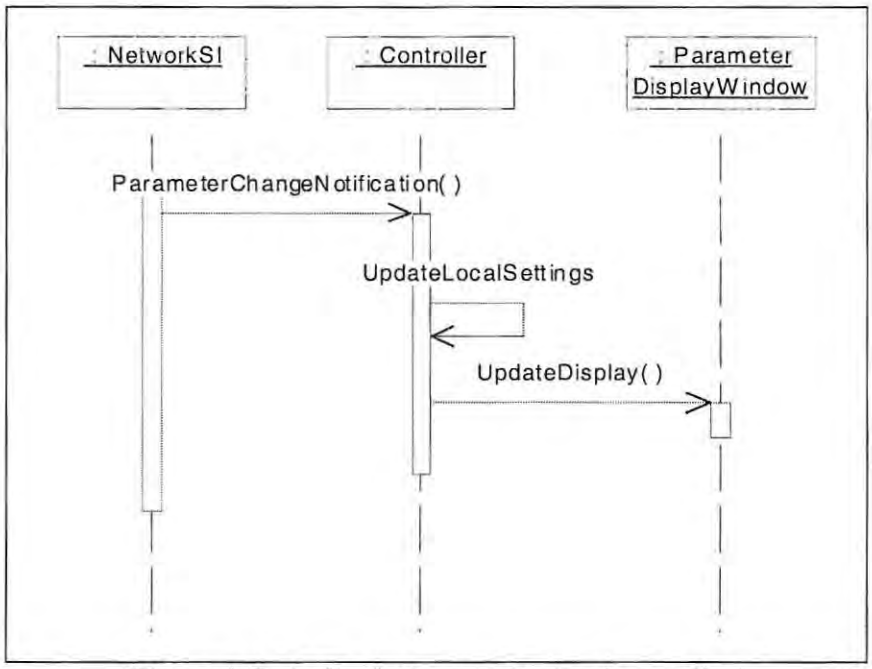

Figure A.7: Audio device monitoring (controller)

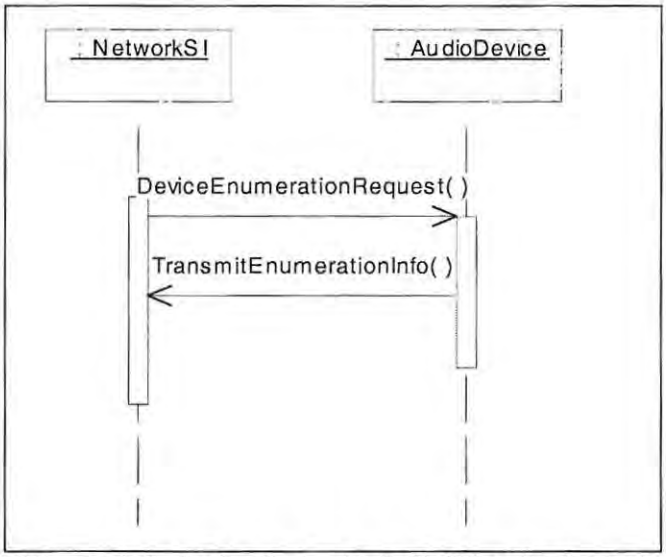

Figure A.8: Audio device Roll-Call (audio device)

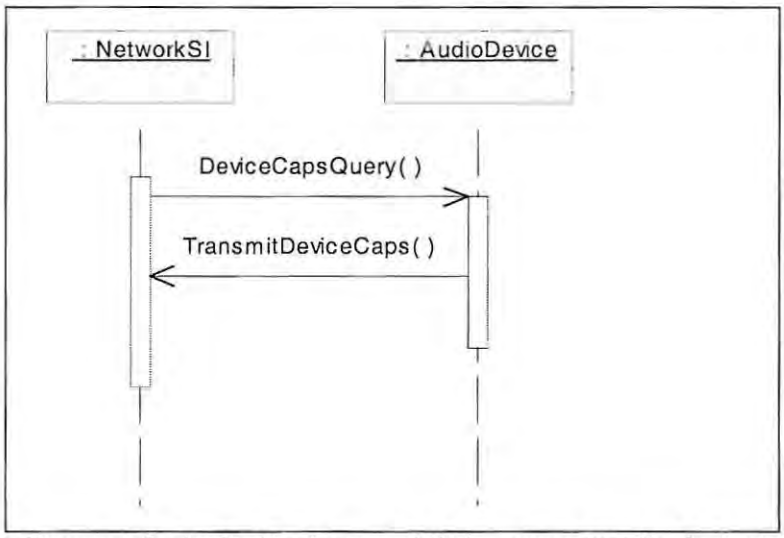

Figure A.9: Audio Device capabilities query (audio device)

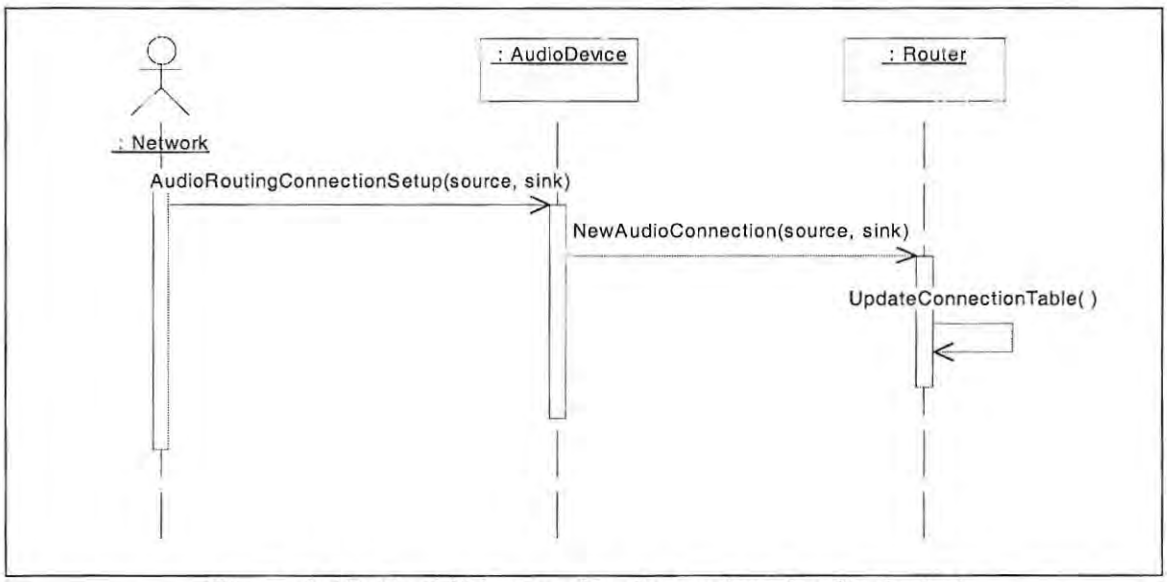

Figure A.10: Establishment of audio routings (audio device)

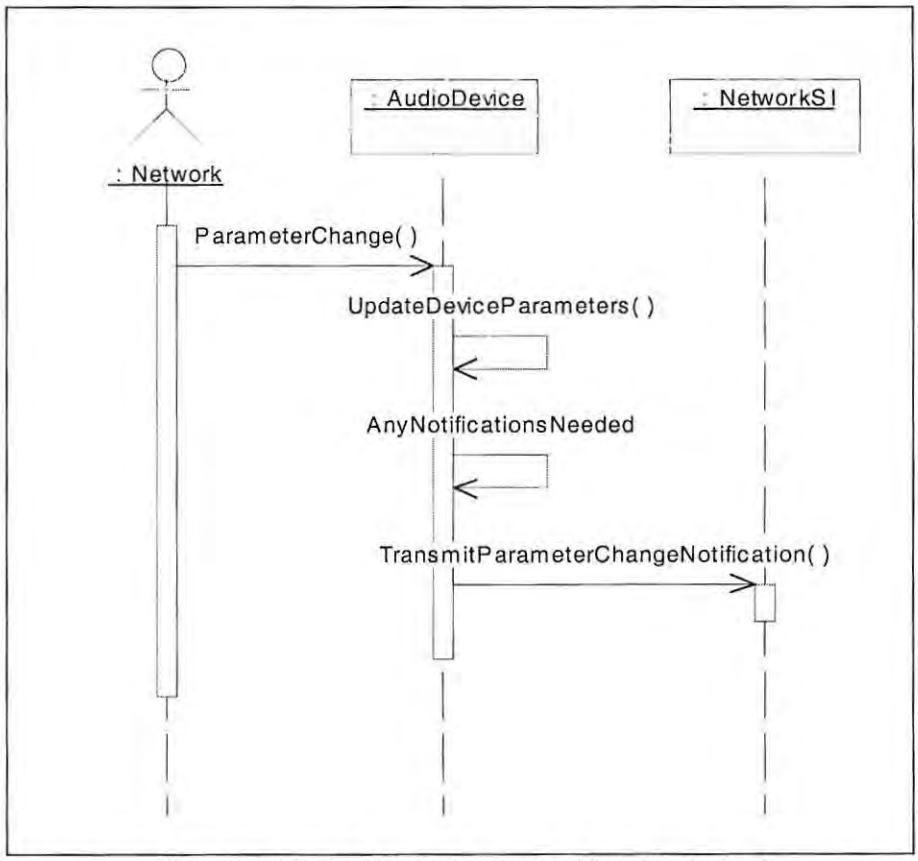

Figure A.11: Audio device control (audio device)

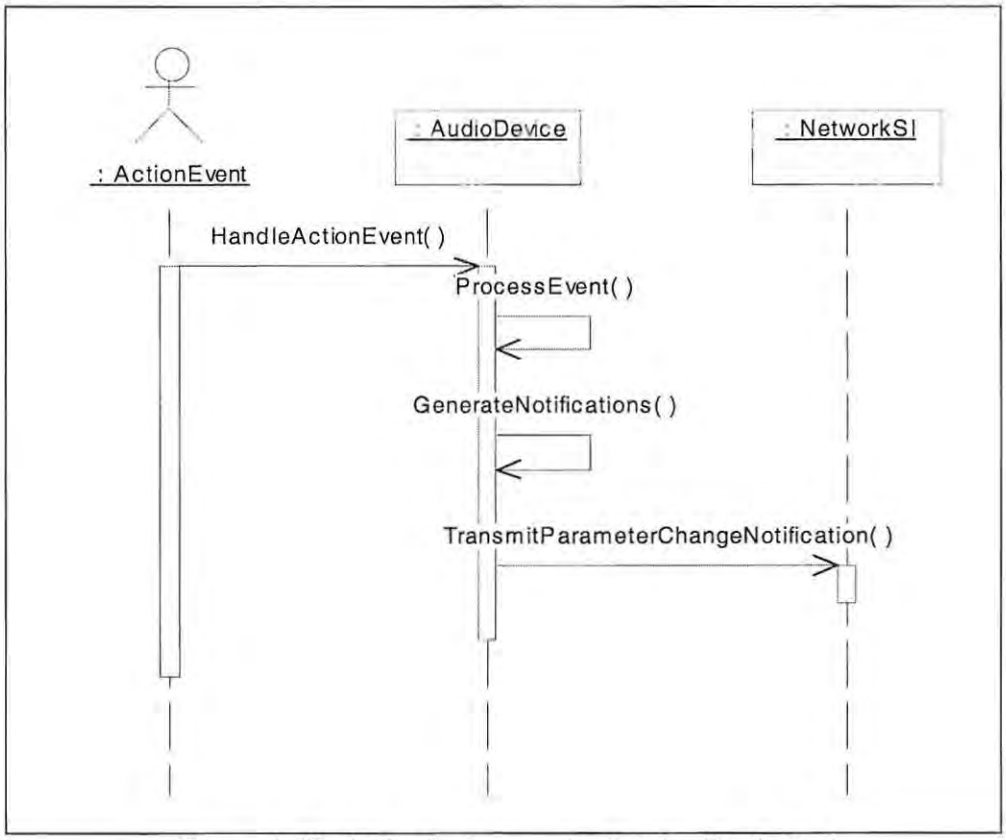

Figure A.12: Audio device monitoring (audio device)

Appendix

B

B.1 Operating System Interface Functions – osdep.c

```
/**************************************************************************
*  Fusion(tm) by Pacific Softworks Inc.                                   *
**************************************************************************
*  Copyright (C) 1993,1994,1995 by                                        *
*  Pacific Softworks Inc., Camarillo, California                          *
*  All rights reserved.  No part of this software may be disclosed or     *
*  distributed in any form or by any means without the prior written      *
*  consent of Pacific Softworks Inc.                                      *
**************************************************************************
*  Version:               $Revision: 1 $                                  *
*  Date of Last Revision: $Date: 7/21/00 10:03a $
*  Filename:              $Source: OSDEP.C $

/**************************************************************************
*                                                                         *
*                  PRELIMINARY - THREADX INTERFACE ROUTINES               *
*                                                                         *
**************************************************************************/
#include <config.h>
#include <std.h>
#include <so.h>
#include <inp.h>
#include <socket.h>
#include <node.h>
#include <netdev.h>
#include <fnsproto.h>

#include <fr.h>
#include   <tx_api.h>


local boolean     t_initialized = false;
u32   interval;          /* really a local, but see misc.s */


typedef struct {
    char        *event;          /* FUSION event pointer */
      TX_THREAD*  p_thread;        /* ptr. to sleeping thread */
} FUSION_EVENT_TABLE;


#define MAX_EVENTS  32

TX_EVENT_FLAGS_GROUP fusion_event_group;

FUSION_EVENT_TABLE fusion_events[MAX_EVENTS];

typedef struct {
    char in_use;
    TX_THREAD thread;
} FORK_TABLE;

FORK_TABLE fork_table[OS_MAX_THREADS];
```

```
TX_SEMAPHORE fusion_critical_semaphore;          /* fusion critical section
control */
TX_THREAD *p_active_thread;
int critical_status;                            /* current critical / normal
status */


/************************************************************************
*                                                                       *
*                                                                       *
*  Function :  os_critical                                              *
*                                                                       *
*  Description :                                                        *
*                                                                       *
*  Parameters :                                                         *
*                                                                       *
*  Return :                                                             *
*                                                                       *
************************************************************************/
int os_critical (void)
{
#ifdef CRITICAL_AS_SEMAPHORE
    register int status;
    register int ret_status;
    register TX_THREAD *p_thread;

    status = tx_interrupt_control(TX_INT_DISABLE);  /* disable interrupts */
    if ( p_active_thread != (p_thread = tx_thread_identify()) ) /* different
thread executing critical */
    {
        tx_semaphore_get (&fusion_critical_semaphore, TX_WAIT_FOREVER); /*
*/
        critical_status = 1;                        /* current status is
critical */
        p_active_thread = tx_thread_identify();     /* this thread now
executing critical */
        ret_status = 0;                             /* status was normal
before call */
    }
    else
        ret_status = 1;                             /* status was critical
before call */
    tx_interrupt_control (status);                  /* restore interrupts */
    return (ret_status);                            /* return previous status
*/
#else
    register int status;
    status = tx_interrupt_control(TX_INT_DISABLE);  /* disable interrupts */
    return (status);
#endif
}

/************************************************************************
*                                                                       *
*                                                                       *
*  Function :  os_normal                                                *
```

```
*                                                                          *
*  Description :                                                           *
*                                                                          *
*  Parameters :                                                            *
*                                                                          *
*  Return :                                                                *
*                                                                          *
****************************************************************************/
void os_normal (int interrupt_status)
{
#ifdef CRITICAL_AS_SEMAPHORE
    register int status;

    if ( interrupt_status == critical_status )      /* status change ? */
        return;                                     /* no, just return */
    status = tx_interrupt_control(TX_INT_DISABLE);  /* disable interrupts */
    if ( (interrupt_status == 0)                    /* return to normal
status ? */
        && (critical_status == 1)
        && (p_active_thread == tx_thread_identify()) )
    {
        critical_status = 0;
        p_active_thread = 0;
        tx_semaphore_put (&fusion_critical_semaphore);  /* release control */
    }
    tx_interrupt_control (status);                  /* restore interrupts */
#else
    tx_interrupt_control(interrupt_status);  /* disable interrupts */
#endif
}


/****************************************************************************
*                                                                          *
*                                                                          *
*  Function :  os_sleep                                                    *
*                                                                          *
*  Description :                                                           *
*                                                                          *
*  Parameters :                                                            *
*                                                                          *
*  Return :                                                                *
*                                                                          *
****************************************************************************/
int os_sleep (char *event)
{
      register int index;
    unsigned int interrupt_status;
      unsigned long events;

    interrupt_status = tx_interrupt_control(TX_INT_DISABLE); /*disable
interrupts*/
    index = 0;
    while ( (fusion_events[index].event != 0) && (index < MAX_EVENTS) )
        index++;
```

```
    if ( index < MAX_EVENTS )              /* any empty entries ? */
    {                                       /* yes, use it */
        fusion_events[index].event = event;    /* set event we are waiting on
*/
        fusion_events[index].p_thread = tx_thread_identify();
                                            /* suspend task waiting for event
*/
        tx_event_flags_get (&fusion_event_group, (1L << index), TX_AND_CLEAR,
                        &events, TX_WAIT_FOREVER);
       fusion_events[index].event = 0;
        tx_interrupt_control(interrupt_status);            /* restore
interrupts */
        return 0;                           /* normal return */
    }
      tx_interrupt_control(interrupt_status);            /* restore
interrupts */
      return (-1);            /* error, more threads running than
OS_MAX_THREAD setting */
}



/************************************************************************
*                                                                       *
*                                                                       *
*  Function :  os_wakeup                                                *
*                                                                       *
*  Description :                                                        *
*                                                                       *
*  Parameters :                                                         *
*                                                                       *
*  Return :                                                             *
*                                                                       *
************************************************************************/
void os_wakeup(char *event)
{
      register int index;

    index = 0;
    while ( index < MAX_EVENTS )
    {
            if (fusion_events[index].event == event)
            {
            tx_event_flags_set(&fusion_event_group, (1L << index), TX_OR);
            return;
            }
            index++;
      }

}



/************************************************************************
```

```
*                                                                          *
*                                                                          *
*  Function :  os_clock_init                                               *
*                                                                          *
*  Description :                                                           *
*                                                                          *
*  Parameters :                                                            *
*                                                                          *
*  Return :                                                                *
*                                                                          *
***************************************************************************/
void os_clock_init (void)
{
      interval = 0;
      t_initialized = true;
}



/***************************************************************************
*                                                                          *
*                                                                          *
*  Function :  os_panic                                                    *
*                                                                          *
*  Description :                                                           *
*                                                                          *
*  Parameters :                                                            *
*                                                                          *
*  Return :                                                                *
*                                                                          *
***************************************************************************/
void os_panic (char * msg,...)
{
   os_printf("\n\rIP PANIC: ");
   os_printf("\r\n%s %i %i", msg, *((int *) &msg+1), *((int *) &msg+2));
    while ( 1 );
}  /* end os_panic routine */



/***************************************************************************
*                                                                          *
*                                                                          *
*  Function :  os_warn                                                     *
*                                                                          *
*  Description :                                                           *
*                                                                          *
*  Parameters :                                                            *
*                                                                          *
*  Return :                                                                *
*                                                                          *
***************************************************************************/
void os_warn (char * msg,...)
{
      char  buf[128];
```

```
      os_printf("FNS WARN: ");
      os_printf(msg, *((int *) &msg+1), *((int *) &msg+2));
}  /* end os_warn routine */


/*************************************************************************
*                                                                        *
*                                                                        *
*  Function :  os_sopen                                                  *
*                                                                        *
*  Description :                                                         *
*                                                                        *
*  Parameters :                                                          *
*                                                                        *
*  Return :                                                              *
*                                                                        *
*************************************************************************/
int os_sopen (so_t * sop)
{
      return (0);
}




/*************************************************************************
*                                                                        *
*                                                                        *
*  Function :  os_sclose                                                 *
*                                                                        *
*  Description :                                                         *
*                                                                        *
*  Parameters :                                                          *
*                                                                        *
*  Return :                                                              *
*                                                                        *
*************************************************************************/
void os_sclose (so_t * sop)
{
      sop->so_osp->os_p1 = (u32)0;    /* reset the task id */
      sop->so_osp->os_p2 = (u32)0;
}




/*************************************************************************
*                                                                        *
*                                                                        *
*  Function :  os_set_err                                                *
*                                                                        *
*  Description :                                                         *
```

```
*                                                                          *
*  Parameters :                                                            *
*                                                                          *
*  Return :                                                                *
*                                                                          *
****************************************************************************/
int os_set_err (int     err_value)
{
      return err_value;
}


/****************************************************************************
*                                                                          *
*                                                                          *
*  Function :  os_priv_user                                                *
*                                                                          *
*  Description :                                                           *
*                                                                          *
*  Parameters :                                                            *
*                                                                          *
*  Return :                                                                *
*                                                                          *
****************************************************************************/
boolean os_priv_user (void)
{
      return 0;
}


/****************************************************************************
*                                                                          *
*                                                                          *
*  Function :  os_fork                                                     *
*                                                                          *
*  Description :                                                           *
*                                                                          *
*  Parameters :                                                            *
*                                                                          *
*  Return :                                                                *
*                                                                          *
****************************************************************************/
int os_fork (char *name, void (*task)(ULONG p_data), void *p_parameters,
             void *p_stack, int stack_size, int priority)
{
    int index;
    unsigned int ret_code;

    index = 0;                    /* find first unused slot in fork table */
    while ( (index < OS_MAX_THREADS) && (fork_table[index].in_use == 1) )
        index++;
    if ( index == OS_MAX_THREADS )          /* table full ? */
        return (-1);                        /* yes, return error */
    if ( fork_table[index].in_use == 2 )    /* was this thread terminated ?
*/
```

```
        ret_code = tx_thread_delete (&fork_table[index].thread); /* yes,
delete it, then re-use */
    fork_table[index].in_use = 1;           /* flag in use */
                                            /* start new thread */
    ret_code = tx_thread_create(&fork_table[index].thread, name, task,
                (ULONG)p_parameters, p_stack, stack_size,
                  priority, priority, TX_NO_TIME_SLICE, TX_AUTO_START);
    return (index);                         /* return "handle" */
}


/*************************************************************************
*                                                                        *
*                                                                        *
*  Function :  os_kill                                                   *
*                                                                        *
*  Description :                                                         *
*                                                                        *
*  Parameters :                                                          *
*                                                                        *
*  Return :                                                              *
*                                                                        *
*************************************************************************/
int os_kill (int handle)
{
     int index;
     TX_THREAD *p_dead_thread;
     FUSION_EVENT_TABLE *p_event;

    index = handle;
    if ( handle == 0 )
    {
        p_dead_thread = tx_thread_identify();
        index = 0;
        while ( (index < OS_MAX_THREADS) && (p_dead_thread !=
&fork_table[index].thread) )
            index++;
        if ( index == OS_MAX_THREADS ) return -1;
    }
    p_dead_thread = &fork_table[index].thread;
    tx_thread_terminate (p_dead_thread);
    fork_table[index].in_use = 2;

    for ( index = 0, p_event = fusion_events; index < OS_MAX_TASKS; index++,
p_event++ )
    {
        if ( p_event->p_thread == p_dead_thread )
            p_event->event = 0;
    }
}


/*************************************************************************
*                                                                        *
*                                                                        *
```

```
*  Function :  os_delay                                                    *
*                                                                          *
*  Description :                                                           *
*                                                                          *
*  Parameters :                                                            *
*                                                                          *
*  Return :                                                                *
*                                                                          *
***************************************************************************/
void os_delay(unsigned short ticks)
{
    tx_thread_sleep(ticks);
}


/***************************************************************************
*                                                                          *
*                                                                          *
*  Function :  init_osdep                                                  *
*                                                                          *
*  Description :                                                           *
*                                                                          *
*  Parameters :                                                            *
*                                                                          *
*  Return :                                                                *
*                                                                          *
***************************************************************************/
void init_osdep (void)
{
    int index;
                                    /* create critical section semaphore */
    tx_semaphore_create (&fusion_critical_semaphore, "Fusion_critical", 1);
    critical_status = 0;            /* current status = normal */
    p_active_thread = 0;
    for ( index = 0; index < OS_MAX_THREADS; index++)
        fork_table[index].in_use = 0;
    tx_event_flags_create (&fusion_event_group, "FusionEvents");
}


/***************************************************************************
*                                                                          *
*                                                                          *
*                                                                          *
*  Function :                                                              *
*                                                                          *
*  Description :                                                           *
*                                                                          *
*  Parameters :                                                            *
*                                                                          *
*  Return : 0                                                              *
*                                                                          *
***************************************************************************/
void *os_malloc(int size)
{
    register void *p_memory;
```

```
    register int i;

    if ((p_memory = (void *)h_alloc((i32)size, (int *)0, 0)) != 0)
    {
/*        bzero(p_memory, size);*/
        return ( p_memory );
    }
    else
        return (0);
}

/*************************************************************************
*                                                                        *
*                                                                        *
*                                                                        *
*  Function :                                                            *
*                                                                        *
*  Description :                                                         *
*                                                                        *
*  Parameters :                                                          *
*                                                                        *
*  Return : 0                                                            *
*                                                                        *
*************************************************************************/
void os_free(void *p_memory)
{
    h_free( (u32 *)p_memory);
}


int MaskInterrupt()
{
    return (tx_interrupt_control(TX_INT_DISABLE)); /*disable interrupts*/
}

void RestoreInterrupt (int interrupt_status)
{
    tx_interrupt_control(interrupt_status);
}

int os_disintr(void)
{
    return (tx_interrupt_control(TX_INT_DISABLE)); /*disable interrupts*/
}

void os_resintr(int savpri)
{
    tx_interrupt_control(savpri);
}
```

B.2 Timing Mechanism – timetask.c

```
/*****************************************************************************
*  File: timetask.c                                                          *
*  Original Author: Fusion(tm) by Pacific Softworks Inc.
*
*  Modified by: B.Klinkradt
*****************************************************************************

/*****************************************************************************
*                                                                            *
*                    TIMER TASK                    *
*                                                                            *
*****************************************************************************/
#include <config.h>
#include <osdep.h>
#include <fnsproto.h>

/* DHT Inludes */
#include "system.h"
#include "utils.h"


/*****************************************************************************
*                                                                            *
*                         EXTERNAL VARIABLES                                 *
*                                                                            *
*****************************************************************************/
extern char FusionRunning;


/*****************************************************************************
*                                                                            *
*                         EXTERNAL FUNCTIONS                                 *
*                                                                            *
*****************************************************************************/
#ifdef ETHER_STATISTICS
extern void EtherStatisticsAverging (void);
#endif

unsigned long udp_performance_timer;
unsigned long tcp_performance_timer;

extern DH_FWSTATUS DHfusion_init(void);

/*****************************************************************************
*                                                                            *
*  Function :  TimerTask                                                     *
*                                                                            *
*  Description :                                                             *
*           Provides the timing mechanism required by the stack,             *
*           and provides statistical information                             *
*  Parameters : thread_input                                                 *
*                                                                            *
*  Return : None.                                                            *
*                                                                            *
*****************************************************************************/
```

```
void TimerTask(DH_UINT32 thread_input)
{
#ifdef ETHER_STATISTICS
    int statistics_timer;

    statistics_timer = 0;
#endif

     blockUntilAllThreadsRunning(thread_input);

    while ( 1 )
    {
        os_delay(MS_PER_TICK/OS_MS_PER_TICK); /* delay 50 ms            */
        udp_performance_timer++;
        tcp_performance_timer++;
        if ( FusionRunning )                  /* is Fusion running ?    */
        {
            t_clock();                        /* yes, do Fusion timer
processing */
        }
#ifdef ETHER_STATISTICS
        if ( ++statistics_timer >= 160 )      /* 8 second timer (50 ms * 160)
*/
        {
            statistics_timer = 0;             /* reset timer            */
            EtherStatisticsAverging ();       /* do statistics average calc. */
        }
#endif
    }
}
```

B.3 Heap initialization and allocation helper routines – heapinit.c

```
/*****************************************************************************
*  File: heapinit.c                                                          *
*  Original author: Fusion(tm) by Pacific Softworks Inc.                     *
*  Modified by: B.Klinkradt                                                  *
*****************************************************************************/

#include <config.h>
#include <std.h>

/* DHT Includes */
#include "system.h"
#include "utils.h"

/*****************************************************************************
*                                                                            *
*                        LOCAL VARIABLES                                     *
*                                                                            *
*****************************************************************************/
unsigned char *next_free;

/* HEAP Cheat */
static unsigned char heap_cheat[HEAP_SIZE];
static int heap_index;


/*****************************************************************************
*  Function : InitStaticMalloc                                               *
*                                                                            *
*  Description : Intialise pointers used in the heap allocation              *
*                procedures                                                  *
*  Parameters : p_unused - not used                                          *
*                                                                            *
*  Return : 0                                                                *
*                                                                            *
*****************************************************************************/
void InitStaticMalloc (char *p_unused)
{
    heap_index = 0;
    next_free =  &heap_cheat[0];
}


/*****************************************************************************
*  Function : StaticMalloc                                                   *
*                                                                            *
*  Description : Allocate a static portion of the heap, and update the       *
*                necessary pointers                                          *
*  Parameters : The size of the portion of heap to be allocated              *
*                                                                            *
*  Return : Reference to allocated heap                                      *
*                                                                            *
*****************************************************************************/
unsigned char *StaticMalloc (unsigned long size)
{
    unsigned char *p_allocated;
```

```
    p_allocated = next_free;
    heap_index += size;
    next_free = &heap_cheat[heap_index];
    return (p_allocated);
}

/*****************************************************************************
*                                                                            *
*                PLATFORM SPECIFIC FUSION HEAP INITIALIZATION                *
*                                                                            *
*  Function : fusion_heap_init                                               *
*                                                                            *
*  Description :                                                             *
*           This function is called from Fusion to initialize the            *
*           Fusion heap.                                                     *
*                                                                            *
*  Parameters :                                                              *
*           p_heap_size - Pointer to Fusion heap start pointer               *
*           p_heap_base - Pointer to Fusion heap size variable               *
*                                                                            *
*  Return : None.                                                            *
*                                                                            *
*****************************************************************************/
void fusion_heap_init(u32 *p_heap_size, u32 *p_heap_base)
{
    unsigned char *p_buffer;
    int i = 0;

    p_buffer = StaticMalloc (HEAP_SIZE);
    *p_heap_size = HEAP_SIZE;
    *p_heap_base = (u32)p_buffer;

    heap_index = 0;
}
```

B.4 Device Driver Data Structures and Declarations – 1394Dev.h

```
/*****************************************************************************
*  File: 1394Dev.h                                                           *
*  Original author: B.Klinkradt                                              *
*****************************************************************************/

/*****************************************************************************
 *           IEEE 1394 Device Driver Data Structures and Declarations        *
 *****************************************************************************/
#ifndef _1394Dev_
#define _1394Dev_

#ifndef _STD_
#include <std.h>
#endif

#include "lal.h"
```

```
/*************************************************************************
*  Required persistent values - As defined within RFC 2734               *
*************************************************************************/

#define     IE_ARP_HWTYPE           0x0018
#define IE_ARP_PTYPE          0x0800 /* IPv4 */
#define IE_ARP_HWLEN          8
#define IE_ARP_PLEN           4
#define IE_ARP_REQUEST        1
#define IE_ARP_RESPONSE       2
#define IE_ARP                0x0806
#define IE_IP                 0x0800
#define IE_MACP               0x8861 /* Defined, but not currently supported
*/
#define IP_VERSION            1
#define IP_NOFRAG       0
#define IP_FIRSTFRAG          1
#define IP_LASTFRAG           2
#define IP_INTERIORFRAG       3

/*************************************************************************
*  1394 to Ethernet Lookup table                                         *
*************************************************************************/
/* Format of the Look up Table to map 1394 to ethernet */
typedef PACK_STRUCTURE struct {
      a64 eui     PACK_MEMBER_MACRO;  /* Unique ID */
      a16 dna PACK_MEMBER_MACRO;  /* Destination Node Address */
      a48 fifo PACK_MEMBER_MACRO; /* FIFO Address within dna */
      u8  mrec PACK_MEMBER_MACRO; /* Max Rec */
      u8  sspd PACK_MEMBER_MACRO; /* Speed */
      a48 ena PACK_MEMBER_MACRO;  /* Ethernet Address */
} PACK_STRUCTURE_MACRO ieee1394_t;

export ieee1394_t ieee1394_tbl[63];

#define     tbl_index(a)      a&0x3f

/*************************************************************************
*  Header structure declarations                                         *
*************************************************************************/
/* GASP Preamble */
typedef PACK_STRUCTURE struct {
      a16 source_id PACK_MEMBER_MACRO; /* Source ID */
      a16 spec_id_hi PACK_MEMBER_MACRO; /* Spec ID Hi*/
      u8  spec_id_lo PACK_MEMBER_MACRO; /* Spec ID Low */
      a16 version_waste PACK_MEMBER_MACRO; /* Version field that's not used
*/
      u8  version PACK_MEMBER_MACRO; /* Version field that's used */
} PACK_STRUCTURE_MACRO gasp_pre;

/* 1394 ARP message */
typedef PACK_STRUCTURE struct {
      a16 arph_hwtype PACK_MEMBER_MACRO;  /* Hardware Type */
      a16 arph_ptype PACK_MEMBER_MACRO;   /* Protocol Type */
      u8  arph_hwlen PACK_MEMBER_MACRO;   /* Haredware Len */
      u8  arph_plen PACK_MEMBER_MACRO;    /* Proto Len */
      a16 arph_opcode PACK_MEMBER_MACRO;  /* Opcode (1=Request 2=Response) */
```

```
      a64 arph_eui PACK_MEMBER_MACRO;      /* Sender Unique ID */
      u8  arph_smrec PACK_MEMBER_MACRO;    /* Sender Max Rec */
      u8  arph_sspd PACK_MEMBER_MACRO;     /* Sender Max Speed */
      a48 arph_fifo PACK_MEMBER_MACRO;     /* Sender Unicast FIFO */
      a32 arph_sip PACK_MEMBER_MACRO;          /* Sender IP */
      a32 arph_tip PACK_MEMBER_MACRO;          /* Target IP */
} PACK_STRUCTURE_MACRO arp_h;

/* Unfragmented encapsulation header */
typedef PACK_STRUCTURE struct {
      a16   unfh_lf PACK_MEMBER_MACRO;     /* Link Fragment field (unfraged
packet so set to 0) */
      a16   unfh_type PACK_MEMBER_MACRO;   /* Ether Type Field */
} PACK_STRUCTURE_MACRO unf_h;

/* First Fragment encapsulation header */
typedef PACK_STRUCTURE struct {
      a16   ffh_lf_size PACK_MEMBER_MACRO;       /* Link and datagram size
field */
      a16   ffh_type PACK_MEMBER_MACRO;    /* Ether Types Field */
      a16   ffh_dgl PACK_MEMBER_MACRO;     /* Datgram Label */
      a16   ffh_reserved PACK_MEMBER_MACRO;      /* Reserved */
} PACK_STRUCTURE_MACRO ff_h;

/* Subsequent Fragment Encaps Header */
typedef PACK_STRUCTURE struct {
      a16 sfh_lf_size PACK_MEMBER_MACRO;   /* Link and Datagram size */
      a16 sfh_offset PACK_MEMBER_MACRO;    /* Fragment Offset */
      a16 sfh_dgl PACK_MEMBER_MACRO;           /* Datagram Label */
      a16 sfh_reserved PACK_MEMBER_MACRO; /* Reserved */
} PACK_STRUCTURE_MACRO sf_h;

/* Format of an ethernet link-level message */
typedef PACK_STRUCTURE struct {
      a48   ll_daddr PACK_MEMBER_MACRO;
      a48   ll_saddr PACK_MEMBER_MACRO;
      a16   ll_type PACK_MEMBER_MACRO;
} PACK_STRUCTURE_MACRO en_ll_h;

/* Format of an ethernet arp message */
typedef PACK_STRUCTURE struct {
      a16     ar_ltype PACK_MEMBER_MACRO;
      a16     ar_ptype PACK_MEMBER_MACRO;
      u8      ar_llen PACK_MEMBER_MACRO;
      u8      ar_plen PACK_MEMBER_MACRO;
      a16     ar_opcode PACK_MEMBER_MACRO;
      a48   ar_shw_addr PACK_MEMBER_MACRO;
      a32   ar_sip PACK_MEMBER_MACRO;
      a48   ar_dhw_addr PACK_MEMBER_MACRO;
      a32   ar_dip PACK_MEMBER_MACRO;
} PACK_STRUCTURE_MACRO en_arp_h;

/*************************************************************************
*  Prototype defintions                                                  *
*************************************************************************/
import void clear1394EtherTableAndFragBufs(void);
import void add1394_entry (ieee1394_t *ie_t, int index);
```

```
import int IE1394_to_ether(DH_BYTE **mp_buf, int len, a16 snodeid);
import void ether_to_1394 (m *mp, DH_OFFSET_1394 fifo_offset, lalNodeHandle
hHandle);
#endif
```

B.5 Device Driver Routines – 13494Dev.c

```
/*************************************************************************
*  File: 1394Dev.c                                                       *
*  Original author: B.Klinkradt                                          *
*************************************************************************/

/*************************************************************************
 *           IEEE 1394 Driver - network interface driver                 *
 *************************************************************************/

/* include files */
#include <config.h>
#include <std.h>
#include <fnsproto.h>
#include <socket.h> /* for llaf_ix */
#include <flags.h>
#include <1394Dev.h>
#include <enet.h>
#include <m.h>
#include <netdev.h>
#include <netioc.h>
#include <boot.h>

#include <lwq.h>
#include <fr.h>  /* for llahp_tbl[] */
#include <IPDHT.h>
#include <osdep.h> /* os dependant functions eg os_fork (for ThreadX) */

/* DHT INCLUDES */
#include "utils.h"
#include "lal.h"

#include "IPtxr.h"

#ifdef DHT_TRACE
      import boolean dht_trace;
#endif
/* globals */

static netdev  * ieee1394_ndp;
extern void PDS_RECEIVER(void); //in asm file, calls pack_isr whan a packet
is received
void ieee1394_stat(void);


extern a48 host_addr;
static int FragDgl=0;

static FRAG_STRUCT IPFrag[IPFRAG_LIMIT]; /* No. of fraged packets we can
handle */
```

```
m *lwq_getbuf(netdev *ndp, int len);

int   offset;
m     *ieee1394mp;
char  *ieee1394recbuf;
a16   nodeid;
static unsigned long ieee1394_send_count=0;
static unsigned long ieee1394_recv_count=0;
static unsigned long ieee1394_packets_dropped = 0;

/*************************************************************************
*  Function : Send1394Packet                                             *
*                                                                        *
*  Description : Sends a prepared IP datagram over the 1394 bus          *
*  Parameters : mp - pointer to the message                              *
*               fifo_offset - target fifo offset                         *
*               hHandle - Handle of target node                          *
*  Return : 0                                                            *
*                                                                        *
*************************************************************************/
void Send1394Packet(m * mp,  DH_OFFSET_1394 fifo_offset, lalNodeHandle
hHandle)
{   netdev  * ndp;
    int len=(u16)(m_dsize(mp));

    if ((ndp = mp->m_ndp) == (netdev *)0)
        return;

    ieee1394_send_count++;

    /*call send procedure here*/
    /* Only stream broadcats */
    if (mp->m_flags & F_M_BCAST)
       IPSendAsyncStream(mp->m_hp, len);
    else /* Send Async Write */
       IPSendAsyncWrite(hHandle, fifo_offset, mp->m_hp, len);

}

/*************************************************************************
*  Function : GetLocalNodeID                                             *
*                                                                        *
*  Description : Returns the local node ID (minus the bus ID)            *
*                                                                        *
*  Parameters : None                                                     *
*                                                                        *
*  Return : Node ID                                                      *
*                                                                        *
*************************************************************************/
u8 GetLocalNodeID()   /* Returns the local node ID minus the bus ID */
{
   DH_UINT32 ThisNode;

   lalGetThisNodeAddr(&ThisNode);

   //sysDebugPrintf("\r\nGetLocalIndex found local node id to be 0%c", (u8)
ThisNode >> 16);
```

```
   return (u8) ThisNode;

}

/*************************************************************************
*  Function : fixbuf                                                     *
*                                                                        *
*  Description : The order in which quadlets are stored vary between     *
*                the IP stack and the firmware.  This procedure          *
*                rearranges the data as is neccessary.                   *
*  Parameters : len - Length of the data to copy                         *
*               qb - Reference to the data                               *
*  Return : Reference to the altered data                                *
*                                                                        *
*************************************************************************/
DH_BYTE *fixbuf(int len, DH_UINT32 *qb)
{ int i = 0;
  int j = 0;
  int c = 0;
  DH_UINT32 tmpbuf;
  DH_BYTE *pbuf = (DH_BYTE* ) qb;

  for (c = 0; c < len; c++)
  {
        /* Fix a quad */
        tmpbuf = qb[c];
        j = 3;
          for (i=0; i <= 3; i++, j--)
            { pbuf[(c*4)+i] = tmpbuf>>(j*8);
              }
  }
  return pbuf;
}

/*************************************************************************
*  Function : clear1394EtherTableAndFragBufs                             *
*                                                                        *
*  Description : Resets the lookup tables, and fragmentation buffers     *
*  Parameters : None                                                     *
*  Return : None                                                         *
*                                                                        *
*************************************************************************/
void clear1394EtherTableAndFragBufs(void)
{ int i;
  for (i = 0; i < 63; i++)
   ieee1394_tbl[i].dna[1] = 0xff;

  for (i=1; i < IPFRAG_LIMIT; i++)
    IPFrag[i].dgl = -1;

}

/*************************************************************************
*  Function : add1394_entry                                              *
*                                                                        *
*  Description : Add an entry into the lookup table, at the specified    *
*                index                                                   *
```

```
*  Parameters : ie_t - Reference to the entry to add                        *
*               index - Position within the table to add the entry          *
*  Return : None                                                            *
*                                                                           *
****************************************************************************/
void add1394_entry (ieee1394_t *ie_t, int index)
{
      os_move(ie_t, &ieee1394_tbl[index], sizeof(ieee1394_t));

      //#ifdef DHT_TRACE
 for (index = 0; index < 5; index++)
  {    sysDebugPrintf("\r\nIndex is %d", index);
      if (ieee1394_tbl[index].dna[1] != 0xff)
            {

                  sysDebugPrintf("\r\nEUI address is %02X %02X %02X %02X %02X
%02X %02X %02X ",
                  ieee1394_tbl[index].eui[0], ieee1394_tbl[index].eui[1],
ieee1394_tbl[index].eui[2],
                  ieee1394_tbl[index].eui[3], ieee1394_tbl[index].eui[4],
ieee1394_tbl[index].eui[5],
                  ieee1394_tbl[index].eui[6], ieee1394_tbl[index].eui[7]);

                  sysDebugPrintf("\r\nHost Addr address is %02X %02X %02X
%02X %02X %02X",
                  ieee1394_tbl[index].ena[0], ieee1394_tbl[index].ena[1],
ieee1394_tbl[index].ena[2],
                  ieee1394_tbl[index].ena[3], ieee1394_tbl[index].ena[4],
ieee1394_tbl[index].ena[5]);
             }
             else
             {
                  sysDebugPrintf("\r\nNode does not contain a valid entry!");
             }
 }
      //#endif
}

/****************************************************************************
*  Function : GetAllocatedIndex                                             *
*                                                                           *
*  Description : Search through the frag buffer, and return the index       *
*                of the specified node                                      *
*  Parameters : dgl - Datagram label to search for                          *
*               snodeid - Node ID of the source                             *
*               IPLen - Length of the IP packet                             *
*  Return : The index or 0 if not found                                     *
*                                                                           *
****************************************************************************/
int GetAllocatedIndex(DH_UINT32 dgl, DH_UINT16 snodeid, DH_UINT32 IPLen)
{
      int i;
       /* Check for an Existing Entry in the Frag Buffers */
       for (i = 1; i <= IPFRAG_LIMIT; i++)
       {
         if ((IPFrag[i].dgl == dgl) && (IPFrag[i].snodeid == snodeid) &&
(IPFrag[i].IPLen == IPLen)) /* already in our buf! */
```

```
            return i;
        }
 return 0;
}

/*************************************************************************
*  Function : GetFragBufferIndex                                         *
*                                                                        *
*  Description : Search through the frag buffer, and if the entry is not *
*                found create a new frag buffer for the IP datagram      *
*  Parameters : dgl - Datagram label to search for                       *
*               snodeid - Node ID of the source                          *
*               IPLen - Length of the IP packet                          *
*  Return : The index or 0 if none available                             *
*                                                                        *
*************************************************************************/
int GetFragBufferIndex(DH_UINT32 dgl, DH_UINT16 snodeid, DH_UINT32 IPLen)
{
       int FragIndex =1;

       if (!GetAllocatedIndex(dgl, snodeid, IPLen)) /* Don't have a valid
entry yet */
       {

      /* Add this entry into a new frag buffer */
      for (FragIndex = 1; FragIndex <= IPFRAG_LIMIT; FragIndex++)
      {
       if (IPFrag[FragIndex].dgl == -1)
           {  IPFrag[FragIndex].dgl = dgl;
            IPFrag[FragIndex].RecLen = 0;
              IPFrag[FragIndex].ttl = 0;
              IPFrag[FragIndex].IPLen = IPLen;
              IPFrag[FragIndex].snodeid = snodeid;
            return FragIndex;
           }

       if (IPFrag[FragIndex].ttl == IPFRAG_LIMIT) /* Free it */
        { IPFrag[FragIndex].ttl = 0;
          IPFrag[FragIndex].dgl = dgl;
          IPFrag[FragIndex].RecLen = 0;
          IPFrag[FragIndex].IPLen = IPLen;
          IPFrag[FragIndex].snodeid = snodeid;
          return FragIndex;
        }
       else
        IPFrag[FragIndex].ttl++;
      }
      }
 return 0;
 }
/*************************************************************************
*  Function : IE1394_to_ether                                            *
*                                                                        *
*  Description : Converts a 1394 packet recieved on the bus to an        *
*            ether packet to send up the stack                       *
*  Parameters : mp_buf - Reference to message buffer                     *
*               len - length of message                                  *
```

```
*                snodeid - node ID of source node                          *
*  Return : The length of message to send, 0 indicates not to send         *
*                                                                          *
****************************************************************************/
int IE1394_to_ether(DH_BYTE **mp_buf, int len, a16 snodeid)
{ //a16 snodeid;
  int buf_entry;
  DH_BYTE *buf = *mp_buf;
      //unf_h *unfh = &buf[0];
  gasp_pre *gp;
  unf_h *unfh;

  if ((snodeid[0] == 0) && (snodeid[1] == 0)) /* Called via an async Stream
*/
  {
    gp = (gasp_pre *) &buf[0];
    unfh = (unf_h *) &buf[8];
    /* Strip off source id, spec id and version */
    os_move(&gp[0], &snodeid[0], sizeof(a16));
    buf_entry = 12;
    if ((gp->spec_id_hi[0] != 0x0) || (gp->spec_id_hi[1] != 0x0) || (gp-
>spec_id_lo != 0x5e) || (buf[7] != 0x1))
      return 0;
  }
  else /* called via an async write */
  {
    unfh = (unf_h *) &buf[0];
    buf_entry = 4;
  }

    switch (unfh->unfh_lf[0]>>6)
   { case IP_NOFRAG:  { /* Unfragmented Packet Received */
            //os_printf("\r\nUnfragmented Packet Received\n");

            /* Expect ARP packets here */
            if ((unfh->unfh_type[0] == 8) && (unfh->unfh_type[1] == 6))
            {  int i; /* Tmp Counter variable  */
               int j; /* Remove Later */
               en_arp_h en_arp; /* Ethernet ARP Packet */
               arp_h *arh = (arp_h *)&buf[buf_entry]; /* 1394 ARP Packet */
               ieee1394_t ie_t; /* Lookup table entry */
               en_ll_h enh; /* Ethernet Link Header */

               /* Update the Lookup table to map 1394 to ethernet*/
               /* Nodes are always added according to node ID (snodeid) */
            /* EUI */
            os_move(&arh->arph_eui[0], &ie_t.eui[0], sizeof(a64));
            /* Dest Node Address */
            os_move(&snodeid[0], &ie_t.dna[0], sizeof(a16));
            /* Ethernet Address */
            for(i = 0; i < 4; i++)
               ie_t.ena[i] = 0xaa;

            ie_t.ena[4] = snodeid[0];
            ie_t.ena[5] = snodeid[1];
            sysDebugPrintf("\r\nNode is to add is %02x %02x", ie_t.ena[4],
ie_t.ena[5]);
```

```
            /* Fifo for async transactions */
            os_move(&arh->arph_fifo[0], &ie_t.fifo[0], sizeof(a48));

            ie_t.mrec = arh->arph_smrec;
            ie_t.sspd = arh->arph_sspd;

            /* Add the new entry to our table */
            add1394_entry(&ie_t, tbl_index(snodeid[1]));

            os_printf("\r\nARP PACKET RECEIVED!!!\n");

               /* Covert Packet back to ethernet ARP packet */
               en_arp.ar_ltype[0] = 0; /* For Ethernet and completeness! */
               en_arp.ar_ltype[1] = 1; /* For Ethernet */
               os_move(&arh->arph_ptype[0], &en_arp.ar_ptype[0],
sizeof(a16)); /* Proto Type */
               en_arp.ar_llen = 6; /* 48 bit address lengths */
               en_arp.ar_plen = arh->arph_plen; /* Proto Address Length */
               os_move (&arh->arph_opcode[0], &en_arp.ar_opcode[0],
sizeof(a16)); /* Opcode */
               os_move (&ieee1394_tbl[tbl_index(snodeid[1])].ena[0]
,&en_arp.ar_shw_addr[0], sizeof(a48)); /* Sender Ether Address */
               os_move (&ieee1394_tbl[tbl_index(GetLocalNodeID())].ena[0] ,
&en_arp.ar_dhw_addr[0], sizeof(a48)); /* Dest Ether Address (ourselves) */
               os_move (&arh->arph_sip[0], &en_arp.ar_sip[0], sizeof(a32));
/* Sender IP Address */
               os_move (&arh->arph_tip[0], &en_arp.ar_dip[0], sizeof(a32));

              /* Ethernet Link Level Header */
              /* ARP requests are broadcast, responses not */
              if (en_arp.ar_opcode[1] == IE_ARP_REQUEST)
                for (i = 0; i < 6; i++)
                  enh.ll_daddr[i] = 0xff;
              else /* Response - our local ethernet address */
                os_move(&ieee1394_tbl[tbl_index(GetLocalNodeID())].ena[0]
,&enh.ll_daddr[0], sizeof(a48));

              /* Ethernet Source Address */
              os_move(&en_arp.ar_shw_addr[0], &enh.ll_saddr[0], sizeof(a48));

              /* Type is ARP */
              enh.ll_type[0] = 8;
              enh.ll_type[1] = 6;

              /* Place this new structure in buf */
              os_move(&enh.ll_daddr[0], &IPFrag[0].FragBuf[0],
sizeof(en_ll_h));
              os_move(&en_arp.ar_ltype[0],
&IPFrag[0].FragBuf[sizeof(en_ll_h)], sizeof(en_arp_h));
              *mp_buf = IPFrag[0].FragBuf;

              return (sizeof(en_ll_h) + sizeof(en_arp_h));
            }

            /* Unfragmented IP packet */
            if ((unfh->unfh_type[0] == 8) && (unfh->unfh_type[1] == 0))
```

```
            {  /* Must have a valid entry in the lookup table */
              os_printf("\r\nUnfragmented IP packet");
               if (ieee1394_tbl[tbl_index(snodeid[1])].dna[1] != 0xff)
               {  /* Simply format the Ethernet Header, and pass it up */
                    en_ll_h enh;

os_move(&ieee1394_tbl[tbl_index(GetLocalNodeID())].ena[0], &enh.ll_daddr[0],
sizeof(a48));  /* Dest is ourselves */
                    os_move(&ieee1394_tbl[tbl_index(snodeid[1])].ena[0],
&enh.ll_saddr[0], sizeof(a48)); /* Sender ethernet address - from table */
                    os_move(&unfh->unfh_type[0], &enh.ll_type[0],
sizeof(a16));

                    /* Generate new data to replace buf */
                    /* Copy ethernet Header */
                    os_move(&enh.ll_daddr[0], &IPFrag[0].FragBuf[0],
sizeof(en_ll_h));
                    /* Copy data back */
                    os_move(&buf[sizeof(unf_h)],
&IPFrag[0].FragBuf[sizeof(en_ll_h)], len - sizeof(unf_h));

                    *mp_buf = IPFrag[0].FragBuf;

                    return (sizeof(en_ll_h) + len - sizeof(unf_h));

               }
            }
            break;
            }
     case IP_FIRSTFRAG: /* First Fragment recieved */
            { ff_h *first_frag = (ff_h*)&unfh->unfh_1f[0]; /* Cast to First
Frag Type */
              int FragIndex;
              sysDebugPrintf("\r\nFirst Fragment Received");

              if ((first_frag->ffh_type[0] != 8) && (first_frag->ffh_type[1]
!= 0)) /* Only care about IP */
               return 0;

               /* Only First Fragments can Assign buffers */
               FragIndex = GetFragBufferIndex((first_frag->ffh_dgl[0] << 8) +
first_frag->ffh_dgl[1], (snodeid[0]<<8)+snodeid[1], ((first_frag-
>ffh_1f_size[0]&0x0F ) << 8)  +  first_frag->ffh_1f_size[1] + 1);
              if (!FragIndex) /* No Available Frag Buffers */
                return 0;

             /* We have a buffer available, pointed at by index, lets fill it
in */
             //IPFrag[FragIndex].dgl = (first_frag->ffh_dgl[0] << 8) +
first_frag->ffh_dgl[1]; /* Datagram Label */
             IPFrag[FragIndex].RecLen += len-sizeof(ff_h);

             /* Place the ethernet header in first */
             if (ieee1394_tbl[tbl_index(snodeid[1])].dna[1] != 0xff)
               {  /* Simply format the Ethernet Header, and pass it up */
                    en_ll_h enh; /* Ethernet Link Level Header */
```

```
os_move(&ieee1394_tbl[tbl_index(GetLocalNodeID())].ena[0], &enh.ll_daddr[0],
sizeof(a48));  /* Dest is ourselves */
                    os_move(&ieee1394_tbl[tbl_index(snodeid[1])].ena[0],
&enh.ll_saddr[0], sizeof(a48)); /* Sender ethernet address - from table */
                    os_move(&first_frag->ffh_type[0], &enh.ll_type[0],
sizeof(a16));

                    /* Copy ethernet Header */
                    os_move(&enh.ll_daddr[0], &IPFrag[FragIndex].FragBuf[0],
sizeof(en_ll_h));
                    /* Copy data */
                    os_move(&buf[sizeof(ff_h)],
&IPFrag[FragIndex].FragBuf[sizeof(en_ll_h)], len - sizeof(ff_h));

                    /* If our len is == IPLen, we have to whole packet, so
pass it up */
                    if (IPFrag[FragIndex].IPLen == IPFrag[FragIndex].RecLen)
                    { *mp_buf = &IPFrag[FragIndex].FragBuf[0];
                        IPFrag[FragIndex].dgl = -1;
                          return (IPFrag[FragIndex].IPLen + sizeof(en_ll_h));
                      } else if (IPFrag[FragIndex].IPLen <
IPFrag[FragIndex].RecLen)
                          { IPFrag[FragIndex].dgl = -1; /* Not a valid packet!
*/
                            return 0;
                          }

            }
            return 0;
            break;
          }
     case IP_LASTFRAG: /* Last Link Fragment Received */
            {
            sf_h *last_frag = (sf_h*) &unfh->unfh_lf[0]; /* Cast to Last Frag
Type */
            DH_BYTE not_found = 0;
              char *dgl;
              int i;
              int FragIndex;

            sysDebugPrintf("\r\nLast Fragment Received");

             FragIndex = GetAllocatedIndex((last_frag->sfh_dgl[0] << 8) +
last_frag->sfh_dgl[1], (snodeid[0]<<8)+snodeid[1], ((last_frag-
>sfh_lf_size[0]&0x0F ) << 8)  +  last_frag->sfh_lf_size[1] + 1);
               if (!FragIndex) /* No Available Frag Buffers */
                 return 0;

             IPFrag[FragIndex].RecLen += len-sizeof(ff_h);

                /* Just copy back data into buffer */
                    os_move(&buf[sizeof(sf_h)],
&IPFrag[FragIndex].FragBuf[(last_frag->sfh_offset[0] << 8) + last_frag-
>sfh_offset[1] + sizeof(en_ll_h)], len - sizeof(sf_h));
```

```
                    /* If our len is == IPLen, we have to whole packet, so pass
it up */
                    if (IPFrag[FragIndex].IPLen == IPFrag[FragIndex].RecLen)
                    {    *mp_buf = &IPFrag[FragIndex].FragBuf[0];
                         IPFrag[FragIndex].dgl = -1;
                           return (IPFrag[FragIndex].IPLen + sizeof(en_ll_h));
                    } else if (IPFrag[FragIndex].IPLen <
IPFrag[FragIndex].RecLen)
                           { IPFrag[FragIndex].dgl = -1; /* Not a valid packet!
*/
                             return 0;
                           }

                 return 0;
                 break;
               }
      case IP_INTERIORFRAG: /* Interior Link Fragment Received */
             {
                 sf_h *int_frag = (sf_h*)&unfh->unfh_lf[0]; /* Cast to Interior
Frag Type */
                 DH_BYTE not_found = 0;
                 char *dgl;
                 int i;
                 int FragIndex;
               sysDebugPrintf("\r\nInterior Fragment Received");

                FragIndex = GetAllocatedIndex((int_frag->sfh_dgl[0] << 8) +
int_frag->sfh_dgl[1], (snodeid[0]<<8)+snodeid[1], ((int_frag-
>sfh_lf_size[0]&0x0F ) << 8)  +  int_frag->sfh_lf_size[1] + 1);
                 if (!FragIndex) /* No Available Frag Buffers */
                   return 0;

                IPFrag[FragIndex].RecLen += len-sizeof(ff_h);

                 /* Just copy data into buffer */
                 os_move(&buf[sizeof(sf_h)],
&IPFrag[FragIndex].FragBuf[(int_frag->sfh_offset[0] << 8) + int_frag-
>sfh_offset[1] + sizeof(en_ll_h)], len - sizeof(sf_h));

                    /* If our len is == IPLen, we have to whole packet, so
pass it up */
                if (IPFrag[FragIndex].IPLen == IPFrag[FragIndex].RecLen)
                { *mp_buf = &IPFrag[FragIndex].FragBuf[0];
                  IPFrag[FragIndex].dgl = -1;
                  return (IPFrag[FragIndex].IPLen + sizeof(en_ll_h));
                } else if (IPFrag[FragIndex].IPLen <
IPFrag[FragIndex].RecLen)
                           { IPFrag[FragIndex].dgl = -1; /* Not a valid packet!
*/
                             return 0;
                           }

             return 0;
             break;
             }
      default: /* No other packet expected...so ignore */
       break;
```

```
   }

}


/*************************************************************************
*  Function : ether_to_1394                                              *
*                                                                        *
*  Description : Convert Ether pack to IEEE 1394 packet for transmission*
*                                                                        *
*  Parameters : mp - message                                             *
*               fifo_offset - target fifo address                        *
*               hHandle - Handle of target node                          *
*  Return : None                                                         *
*                                                                        *
*************************************************************************/
void ether_to_1394 (m *mp, DH_OFFSET_1394 fifo_offset, lalNodeHandle hHandle)
{  int f;
   char tmpbuf[IEEE1394_MAX_SEND+1024];

  en_ll_h  *en_l = (en_ll_h*) mp->m_hp; /* Link Level Headers */
  unf_h         uf_h;   /* RFC 1734 Unfragmented Header */
  DH_UINT32 ThisNode;

  /* Check if ether broadcast */
  if (mp->m_flags & F_M_BCAST)
  {   //os_printf("\r\nEthernet Broadcast detected\n");
  }

  if ((int) mp->m_hp[12] == 8 && (int) mp->m_hp[13] == 6) /* An ARP packet */
      { /* AN ARP PACKET */
        en_arp_h *en_a; /* Ether ARP Header */
        arp_h     ar_h; /* RFC 2734 ARP Header */
        gasp_pre  g_pre;      /* Gasp Preamble */

        en_a = (en_arp_h*) &mp->m_hp[14]; /* Type cast the m structure that
we are concerned with */
        if ((en_a->ar_ptype[0] != 8) && (en_a->ar_ptype[1] != 0)) /* IP
Protocol */
        {
            /* For Future use! */
        }
        else
        { /* AN IP ARP PACKET */
           /* Build up the ARP packet that will be passed out onto the bus */
           /* ARP Hardware Type code (from RFC 2734) */
           ar_h.arph_hwtype[0] = 0x00;
           ar_h.arph_hwtype[1] = 0x18;

           /* Proto Type - only catering for IP here */
            os_move(&en_a->ar_ptype[0], &ar_h.arph_ptype[0], sizeof(a16));
           /* HW address length */
           ar_h.arph_hwlen = 0x10; /* RFC states 16...but probably should be
8 */
           /* IP address length */
           ar_h.arph_plen = en_a->ar_plen;
           /* Opcode */
```

```
            os_move(&en_a->ar_opcode[0], &ar_h.arph_opcode[0], sizeof(a16));
            /* Sender Unique ID */
            os_move(&ieee1394_tbl[tbl_index(GetLocalNodeID())].eui[0],
&ar_h.arph_eui[0], sizeof(a64));
            /* Sender Max Rec */
            ar_h.arph_smrec = ieee1394_tbl[tbl_index(GetLocalNodeID())].mrec;
            /* sspd */
            ar_h.arph_sspd = ieee1394_tbl[tbl_index(GetLocalNodeID())].sspd;
            /* Sender FIFO address */
            os_move(&ieee1394_tbl[tbl_index(GetLocalNodeID())].fifo[0],
&ar_h.arph_fifo[0], sizeof(a48));
            /* Sender IP address */
            os_move(&en_a->ar_sip[0], &ar_h.arph_sip[0], sizeof(a32));
            /* Target IP address */
            os_move(&en_a->ar_dip[0], &ar_h.arph_tip[0], sizeof(a32));

            /* Fill mp structure with zero's */
            for (f = 0; f < (u16)(m_dsize(mp)); f++)
                  mp->m_hp[f] = 0;

            /* Create Encapsulation Header - No Fragmentation */
            /* Format the new unfragmented header */
            uf_h.unfh_lf[0] = uf_h.unfh_lf[1] = 0;
            /* Type is ARP 0x0806 */
            uf_h.unfh_type[0] = IE_ARP>>8;
            uf_h.unfh_type[1] = IE_ARP&0xff;

                  /* Create Gasp Preamble */
                  lalGetThisNodeAddr(&ThisNode);
                  //sysDebugPrintf("\r\n THis node has has the address %08x",
ThisNode);

                  g_pre.source_id[0] = ThisNode >> 8;
                  g_pre.source_id[1] = ThisNode & 0xFF;
                  //sysDebugPrintf("\r\n THis node has has the address %02x
%02x", g_pre.source_id[0], g_pre.source_id[1]);
                  g_pre.spec_id_hi[0] = g_pre.spec_id_hi[1] = 0;
                  g_pre.spec_id_lo = 0x5e;
                  g_pre.version_waste[0] = g_pre.version_waste[1] = 0;
                  g_pre.version = IP_VERSION;

                  //sysDebugPrintf("\r\n THis node has has the address %02x
%02x", g_pre.source_id[0], g_pre.source_id[1]);
            /* Change Existing mp structure */
            /* Fill in Gasp Preamble */
            os_move (&g_pre.source_id[0], &mp->m_hp[0], sizeof(gasp_pre));

            /* fill in Encapsulation Header */
            os_move(&uf_h.unfh_lf[0], &mp->m_hp[sizeof(gasp_pre)],
sizeof(unf_h));

            /* Change Existing mp structure - fill in ARP Packet*/
            os_move(&ar_h.arph_hwtype[0], &mp->m_hp[sizeof(unf_h) +
sizeof(gasp_pre)], sizeof(arp_h)); /*(u16)(m_dsize(mp))); */

            /* Set the new Tail Pointer */
```

```
          mp->m_tp = &mp->m_hp[sizeof(gasp_pre) + sizeof(arp_h) +
sizeof(unf_h)];

          /* Send the Packet */
          Send1394Packet(mp, fifo_offset, hHandle);

        } /* AN IP ARP PACKET */
      } /* AN ARP PACKET */
   else
   { /* not an arp packet */
      if ((int) mp->m_hp[12] == 8 && (int) mp->m_hp[13] == 0) /* An IP Packet
follows the Ethernet Header */
      { /* Simply Change Ethernet Header to 1394 header */
        /* Handle Fragmentation here */
      int mrec = 1;
      int c;
      for (c = 0; c < (ieee1394_tbl[tbl_index(en_1->ll_daddr[5])].mrec)+1;
c++)
         mrec *= 2;

      if (mrec > IEEE1394_MAX_SEND) mrec = IEEE1394_MAX_SEND-32; /* Limited
by our FIFO */

        if (m_dsize(mp) - sizeof(en_ll_h) < mrec)
        {
             /* Format the new unfragmented header */
             uf_h.unfh_lf[0] = uf_h.unfh_lf[1] = 0;
             os_move(&mp->m_hp[12], &uf_h.unfh_type[0], sizeof(a16));

             /* Move the header into the m structure */
             os_move(&uf_h.unfh_lf[0], &mp->m_hp[0], sizeof(unf_h));
             /* Move the rest of the hp data down */
             os_move(&mp->m_hp[14], &mp->m_hp[4], (u16) (m_dsize(mp) - 10));


             /* Set the old area of hp to zero */
             for (f = 0; f < 10; f++)
                 mp->m_hp[(u16) (m_dsize(mp) - f)] = 0;

             /* Update the tail pointer */
             mp->m_tp = &mp->m_hp[(u16) (m_dsize(mp)-10)];
             /* Send the Packet */
               Send1394Packet(mp, fifo_offset, hHandle);
        }
        else
        {
          /* Form the First Link Fragment Header */
          ff_h first_frag;
          sf_h *next_frag;
          int TotalLength = (mp->m_hp[sizeof(en_ll_h)+2] << 8) + mp-
>m_hp[sizeof(en_ll_h)+3];
          int AmtDataCopied = mrec - sizeof(ff_h);
          int FirstDataSize = AmtDataCopied;

          char tmp_buf[2048];
          sysDebugPrintf("\r\nFragmentation Needed");
```

```
            first_frag.ffh_lf_size[0] = (IP_FIRSTFRAG << 6) + (mp-
>m_hp[2+sizeof(en_ll_h)]&0x0F);
            first_frag.ffh_lf_size[1] = mp->m_hp[3+sizeof(en_ll_h)] - 1;
            first_frag.ffh_type[0] = 0x08;
            first_frag.ffh_type[1] = 0x00;
            FragDgl++;
            first_frag.ffh_dgl[0] = FragDgl >> 8;
            first_frag.ffh_dgl[1] = FragDgl & 0x00FF;

            /* Copy Data not sent in this fragment */
            os_move(&mp->m_hp[sizeof(en_ll_h)+AmtDataCopied], &tmp_buf[0],
TotalLength-AmtDataCopied);

            /* Copy the header for this fragment */
            os_move(&first_frag.ffh_lf_size[0], &mp->m_hp[0], sizeof(ff_h));
            /* Copy the data for this fragment */
            /* Note: eth. link header is larger than frag header, so the below
is safe */
            os_move(&mp->m_hp[sizeof(en_ll_h)], &mp->m_hp[sizeof(ff_h)],
AmtDataCopied);

            /* Update the Tail pointer */
            mp->m_tp=&mp->m_hp[AmtDataCopied+sizeof(ff_h)];

            /* Send this Fragment */
           Send1394Packet(mp, fifo_offset, hHandle);

           /* Form the last and inner fragments */

          while (TotalLength-AmtDataCopied+sizeof(sf_h) > mrec) /* more than
we can send in one go - Interior fragments*/
           {
             next_frag = (sf_h*) &first_frag.ffh_lf_size[0];

             next_frag->sfh_lf_size[0] &= 0x0F; /* clear the type field */
             next_frag->sfh_lf_size[0] = (IP_INTERIORFRAG << 6) + next_frag-
>sfh_lf_size[0];
             next_frag->sfh_offset[0] = AmtDataCopied >> 8;
             next_frag->sfh_offset[1] = AmtDataCopied & 0x00FF;
             /* The other fields are taken from above, and should be correct
*/

             /* Copy Header into mp */
             os_move(&next_frag[0], &mp->m_hp[0], sizeof(sf_h));
             /* Copy Data into mp */
             os_move(&tmp_buf[AmtDataCopied-FirstDataSize], &mp-
>m_hp[sizeof(sf_h)], mrec-sizeof(sf_h));
             AmtDataCopied += mrec-sizeof(sf_h);

             /* Update Tail Pointer */
             mp->m_tp=&mp->m_tp[AmtDataCopied+sizeof(sf_h)];

             /* Send this Frag */
             Send1394Packet(mp, fifo_offset, hHandle);
           }

            /* Last Fragment */
```

```
            next_frag = (sf_h*) &first_frag.ffh_lf_size[0];

             next_frag->sfh_lf_size[0] &= 0x0F; /* clear the type field */
             next_frag->sfh_lf_size[0] = (IP_LASTFRAG << 6) + next_frag-
>sfh_lf_size[0];
             next_frag->sfh_offset[0] = AmtDataCopied >> 8;
             next_frag->sfh_offset[1] = AmtDataCopied & 0x00FF;
             /* The other fields are taken from above, and should be correct
*/

             /* Copy Header into mp */
            os_move(&next_frag[0], &mp->m_hp[0], sizeof(sf_h));
            /* Copy Data into mp */
            os_move(&tmp_buf[AmtDataCopied-FirstDataSize], &mp-
>m_hp[sizeof(sf_h)], TotalLength - AmtDataCopied);

            /*Update the Tail Pointer */
            mp->m_tp=&mp->m_hp[TotalLength - AmtDataCopied + sizeof(sf_h)];

            /* Finally send this last frag */
            Send1394Packet(mp, fifo_offset, hHandle);
          }
         //os_printf("\r\nEthernet IP packet detected (ether_convert)\n");
       } /* an IP packet */

       else
         os_printf("\r\nUnknown Packet Type detected\n");
    } /* not an arp packet */

}

/*************************************************************************
*  Function : ieee1394_init                                              *
*                                                                        *
*  Description : Network interface initialisation routine                *
*                                                                        *
*  Parameters : ndp - Reference to the network device                    *
*                                                                        *
*  Return : 0                                                            *
*                                                                        *
*************************************************************************/
int ieee1394_init (netdev * ndp)
{  DH_UINT32 ThisNode;
   lalDeviceInfo DevInfo;
   int i;

    ieee1394_ndp=ndp;

    if ((ieee1394_ndp) == (netdev *)0)
      sysDebugPrintf("\r\nInit proc dectected null ndp...fix it!");

#define IEEE1394_NUMBER_LARGE_PACKETS     4
#define IEEE1394_NUMBER_SMALL_PACKETS     10
    ndp->nd_lpkts = IEEE1394_NUMBER_LARGE_PACKETS;
    ndp->nd_spkts = IEEE1394_NUMBER_SMALL_PACKETS;
    ndp->nd_flags |= F_N_MAINT_BUF;
```

```
 /* Initialise Frag structures */
 for (i = 0; i < IPFRAG_LIMIT; i++)
  { IPFrag[i].dgl = -1;
    IPFrag[i].ttl = 0;
  }

 return 0;
}

/*************************************************************************
*  Function : ieee1394_updown                                            *
*                                                                        *
*  Description : Network interface updown routing, brings the interface *
*                into or out of operation                                *
*  Parameters : ndp - network device                                     *
*               flags - updown flag                                      *
*               options - unused                                         *
*  Return : Status (up or down)                                          *
*                                                                        *
*************************************************************************/
int ieee1394_updown(netdev * ndp, u16 flags,  char *options)
{  fast llafh * llafhp;
   int hdr_size;
   int i;
   a64 eui_addr;

   a48 macp;

   int result;
   ieee1394_t ie_t;

   if (!ndp)
      return ENETDOWN;

    if (flags)
    {          /* DO NOT ALTER THIS ADDRESS! UNLESS YOU ALTER IT IN IPTXR.C
AS WELL */
                        host_addr[0] = 0x00;
               host_addr[1] = 0xbe;
               host_addr[2] = 0xae;
               host_addr[3] = 0x03;
               host_addr[4] = 0x02;
               host_addr[5] = 0xd0;

             llafhp = llahp_tbl[llaf_ix(ndp->nd_lladdr.a_type)];
           hdr_size = (int) llafhp->lah_hsize;
           offset = (int) nc_hsize - hdr_size;

           stass(ndp->nd_lladdr.a_ena, host_addr);
           ndp->nd_lladdr.a_type = AF_ETHER; /* AF_L3;*/
             ndp->nd_lladdr.a_len = sizeof(a48);/*a64);*/

      #ifdef LWQ
           lwq_alobuf(ndp);
           //os_printf("\r\nBuffers have been allocated\n");
      #endif  /* LWQ */
```

```
      ieee1394mp = lwq_getbuf(ndp, IEEE1394_MAX_SEND+1024);
      ieee1394recbuf = ieee1394mp->m_hp;

    }
    else
    {    os_wakeup((char *)&ndp->nd_recvq); /*kill the receive task
(F_N_ONLINE is off)*/
      #ifdef LWQ
            os_printf("\r\nBuffers have been released\n");
            lwq_rlsbuf(ndp);
      #endif /* LWQ */
   }

#ifdef DEBUG
    /*os_printf( "\r\nIEEE 1394 driver updown routine called" );
    sysDebugPrintf("\r\n IP Device Driver updown routine called and local
details initialised in ether/1394 lookup table");
    /*os_printf( "MAC address set to 0x%x 0x%x 0x%x 0x%x 0x%x 0x%x 0x%x 0x%x
\n", host_addr.a6[0],
      host_addr.a6[1], host_addr.a6[2], host_addr.a6[3], host_addr.a6[4],
host_addr.a6[5]);*/
#endif

    return (0);
}

/*************************************************************************
*  Function : ieee1394_malloc                                            *
*                                                                        *
*  Description : Allocate a mp for the ieee 1394 driver       n          *
*                                                                        *
*  Parameters : handle - not used                                        *
*               len - length                                             *
*  Return : 0                                                            *
*                                                                        *
*************************************************************************/
/*************************************************************************
*
*
* ieee1394_malloc - Allocate a mp for the ieee 1394 driver
*
* RETURNS: data portion of mp
*/
char * ieee1394_malloc (int handle, int len)
{
      m *  mp;
      os_printf( "\r\nIEEE 1394 driver malloc routine called\n" );

      if ((mp = lwq_getbuf(ieee1394_ndp, len)) != (m *)0)
      return mp->m_hp;
      else {
            ieee1394_packets_dropped++;
            return (char *)0;

       }
return (char *)0;
}
```

```
/*************************************************************************
*  Function : ieee1394_start                                             *
*                                                                        *
*  Description : Output packet to network interface device               *
*                                                                        *
*  Parameters : mp - message to be sent                                  *
*                                                                        *
*  Return : 0                                                            *
*                                                                        *
*************************************************************************/
int ieee1394_start (m * mp)
{
    u16 len;            /* length           */
    int result;
    int err;
    lalNodeHandle hHandle;
    DH_UINT32 NodeAddr;
    DH_OFFSET_1394 fifo_offset;
    a48 tmp_fifo;

    int i,j;
    //os_printf( "\r\nIEEE 1394 driver start routine called" );

    /* Get FIFO address of this node if its not a broadcast */
    if (!(mp->m_flags & F_M_BCAST))
    {
      /* Get Dest Handle from packet before we send it */
      NodeAddr = (mp->m_hp[4] << 8) + mp->m_hp[5];
      lalGetHandleFromNodeAddr(NodeAddr, &hHandle);

      //sysDebugPrintf("\r\nHandle is %08x", hHandle);

      os_move(&ieee1394_tbl[tbl_index(mp->m_hp[5])].fifo[0], &tmp_fifo[0],
sizeof(a48));

      /*for (i = 0; i < 6; i++)
         sysDebugPrintf("\r\nfifo from table is %02x ", tmp_fifo[i]);
      */
      fifo_offset.High = (tmp_fifo[0] << 8) +  tmp_fifo[1];

      j = 3;
      for (i = 0; i <= 3; i++, j--)
      {
              fifo_offset.Low = tmp_fifo[i+2] << (8*j);
            //  sysDebugPrintf("\r\ntmp_fifo[i+2] is %02x", tmp_fifo[i+2]);
      }

      //sysDebugPrintf("\r\nFifo high is %04x and low is %08x",
fifo_offset.High, fifo_offset.Low);
    }

   ether_to_1394(mp, fifo_offset, hHandle); /* Convert Ether packet to RFC
2734 compliany packet */
   return 0;
}
```

```
/*************************************************************************
*  Function : ieee1394_ioctl                                             *
*                                                                        *
*  Description : I/O control (Promiscous/Normal Mode)                    *
*                                                                        *
*  Parameters : ndp, cmnd, addr                                          *
*                                                                        *
*  Return : 0 or Not Supported                                           *
*                                                                        *
*************************************************************************/
int ieee1394_ioctl(netdev * ndp, int cmnd, char * addr)
{
  int rc=EOPNOTSUPP;

  os_printf("\r\nIEEE 1394 I/O control procedure called\n");

   switch(cmnd)
   {
     case ENIOCALL:
     case ENIOCPROMISC:
       break;

      case ENIOCNORMAL:
      default:
        rc=EOPNOTSUPP;
    }
    return(rc);
}

/*************************************************************************
*  Function : ieee1394recv                                               *
*                                                                        *
*  Description : Receive task (Thread).  Processes packet.               *
*                                                                        *
*  Parameters : thread_input                                             *
*                                                                        *
*  Return : None                                                         *
*                                                                        *
*************************************************************************/
void ieee1394recv (DH_UINT32 thread_input)
{
  use_critical;

  blockUntilAllThreadsRunning(thread_input);

  //os_printf( "\r\nIEEE 1394 driver receive routine called\n" );
   while(ieee1394_ndp->nd_flags & F_N_ONLINE)
  {
      critical;
      while (q_empty(&ieee1394_ndp->nd_recvq)
                  && ieee1394_ndp->nd_flags & F_N_ONLINE)
      {
      //os_printf( "\r\nIEEE 1394 driver receive routine called: Sleep on
nd_recvq\n" );
      os_sleep((char *)&ieee1394_ndp->nd_recvq);
      }
      normal;
```

```
      //os_printf( "\r\nIEEE 1394 driver receive routine called: ieee1394_up
called\n" );
      lwq_recv_task(ieee1394_ndp, en_up);
  }
#ifdef DEBUG
      os_printf("\r\nIEEE 1394 Driver Interface Recv Task KILLED\n");
#endif
  os_idle();
}

/*************************************************************************
*  Function : ieee1394isr                                                *
*                                                                        *
*  Description : Network interface interrupt service routine             *
*                                                                        *
*  Parameters : len - length of data                                     *
*               quadbuf - data (Quad format)                             *
*               NodeID - Node ID                                         *
*  Return : 0                                                            *
*                                                                        *
*************************************************************************/
/*************************************************************************
*
*
* ieee1394isr - network interface interrupt handler
*
* RETURNS: N/A
*/

void ieee1394_isr (int len, DH_UINT32 *quadbuf, DH_UINT32 NodeID)
{
      /*call this when a packet arrives
        ie. The entry point to the stack */
    m               *mp;// = ieee1394mp;
    int i, j;
    int length = len;
    int level;
    char *buf;
    DH_BYTE **mp_buf;
    a16 n_id;

    n_id[0] = NodeID >> 8;
    n_id[1] = NodeID & 0xFF;

    //fix the buf! */
   *mp_buf = fixbuf(len, quadbuf);
   length = length*4;

    /* Format packet as ethernet packet, and pass up the stack */
    length = IE1394_to_ether(mp_buf, length, n_id);
    if (length == 0)
      return;

    mp = lwq_getbuf(ieee1394_ndp, length );

    mp->m_ndp=ieee1394_ndp;
    mp->m_hp=(char*) *mp_buf;
```

```
    mp->m_tp=mp->m_hp+length;

    sysDebugPrintf("\r\nMP size is %i", (u16)(m_dsize(mp)));

    level = os_critical();
    lq_in(&mp->m_q, &ieee1394_ndp->nd_recvq);
    os_wakeup((char *)&ieee1394_ndp->nd_recvq);
    os_normal(level);

    ieee1394_recv_count++;
}

/*************************************************************************
*  Function : ieee1394_maint                                             *
*                                                                        *
*  Description : Maintenance routine                                     *
*                                                                        *
*  Parameters : None                                                     *
*                                                                        *
*  Return : None                                                         *
*                                                                        *
*************************************************************************/
void ieee1394_maint(void)
{
    if ((ieee1394_ndp) == (netdev *)0)
      { os_printf( "\r\nIEEE 1394 driver maint routine reports that driver
not intialised\n" );
            return;
      }

    if (!(ieee1394_ndp->nd_flags & F_N_ONLINE))
      { os_printf( "IEEE 1394 driver maint routine reports that driver is
online\n" );
            return;
      }

    lwq_alobuf(ieee1394_ndp);
}

/*************************************************************************
*  Function : ieee1394_bringdown                                         *
*                                                                        *
*  Description : Release the packet types from the 1394 driver           *
*                                                                        *
*  Parameters : None                                                     *
*                                                                        *
*  Return : None                                                         *
*                                                                        *
*************************************************************************/
void ieee1394_bringdown (void)
{ os_printf( "\r\nIEEE 1394 driver bring down routine called\n" );

    if ((ieee1394_ndp != (netdev *)0) && (ieee1394_ndp->nd_flags &
F_N_ONLINE))
    {
 /*#ifdef DEBUG
          ieee1394_stat();
```

```
#endif*/
            ieee1394_updown(ieee1394_ndp,0,(char *)0);
        }
}

/*************************************************************************
*  Function : ieee1394_stat                                              *
*                                                                        *
*  Description : Statistics gathering routine - NOT IMPLEMENTED          *
*                                                                        *
*  Parameters : None                                                     *
*                                                                        *
*  Return : 0                                                            *
*                                                                        *
*************************************************************************/
void ieee1394_stat (void)
{
      os_printf( "\r\nIEEE 1394 driver stat routine called\n" );
}
```

Appendix C

Appendix

D

Function	Key
accept	AC
bind	BD
blocking	BL
caccept	CA
close	CL
connect	CO
getpeername	GP
getsockname	GS
getsopt	GO
ip_bind	IB
listen	LT
nonblocking	NB
nselect	NS
pioctl	PI
recv	RE
recvfrom	RF
recvh	RH
reject	RJ
send	SE
sendh	SH
sendto	ST
setsopt	SO
shutdown	SD
socket	SC
socketpair	SP
tcp_open	TO
udp_open	UO

Appendix E

E.1 FB-88 Device Description Document

```
<?xml version="1.0"?>
<root xmlns="urn:schemas-upnp-org:device-1-0">
  <specVersion>
    <major>1</major>
    <minor>0</minor>
  </specVersion>
  <device>
    <UDN>uuid:D9FD5726-5310-4127-9F12-4459367F0F94</UDN>
    <friendlyName>SAMPLE DEVICE - FB88</friendlyName>
    <deviceType>urn:schemas-upnp-org:device:fb88:1</deviceType>
    <presentationURL>../presentation/fb88.html</presentationURL>
    <manufacturer>Crest</manufacturer>
    <manufacturerURL>http://www.crestaudio.com/</manufacturerURL>

    <modelName>Pro Model</modelName>
    <modelNumber>PM1</modelNumber>
    <modelDescription>FB88 - Professional Audio
Device</modelDescription>
    <modelURL>http://www.microsoft.com/</modelURL>
    <UPC>000000000001</UPC>
    <serialNumber>0000001</serialNumber>
    <iconList>
       <icon>
          <mimetype>image/png</mimetype>
          <width>16</width>
          <height>16</height>
          <depth>2</depth>
          <url>../images/16-2.png</url>
       </icon>
    </iconList>
    <serviceList>
      <service>
        <serviceType>urn:schemas-upnp-
org:service:channel1:1</serviceType>
        <serviceId>urn:upnp-org:serviceId:channel1</serviceId>
        <controlURL>../control/isapictl.dll?channel1</controlURL>
        <eventSubURL>../control/isapictl.dll?channel1</eventSubURL>
        <SCPDURL>../SCPD/FB88Channel1-SCPD.xml</SCPDURL>
      </service>


      <service>
        <serviceType>urn:schemas-upnp-
org:service:channel2:1</serviceType>
        <serviceId>urn:upnp-org:serviceId:channel2</serviceId>
        <controlURL>../control/isapictl.dll?channel2</controlURL>
        <eventSubURL>../control/isapictl.dll?channel2</eventSubURL>
        <SCPDURL>../SCPD/FB88Channel2-SCPD.xml</SCPDURL>
      </service>

      <service>
        <serviceType>urn:schemas-upnp-
org:service:channel3:1</serviceType>
        <serviceId>urn:upnp-org:serviceId:channel3</serviceId>
        <controlURL>../control/isapictl.dll?channel3</controlURL>
```

```
        <eventSubURL>../control/isapictl.dll?channe31</eventSubURL>
        <SCPDURL>../SCPD/FB88Channel3-SCPD.xml</SCPDURL>
      </service>

      <service>
        <serviceType>urn:schemas-upnp-
org:service:channel4:1</serviceType>
        <serviceId>urn:upnp-org:serviceId:channel4</serviceId>
        <controlURL>../control/isapictl.dll?channel4</controlURL>
        <eventSubURL>../control/isapictl.dll?channel4</eventSubURL>
        <SCPDURL>../SCPD/FB88Channel4-SCPD.xml</SCPDURL>
      </service>

      <service>
        <serviceType>urn:schemas-upnp-
org:service:channel5:1</serviceType>
        <serviceId>urn:upnp-org:serviceId:channel5</serviceId>
        <controlURL>../control/isapictl.dll?channel5</controlURL>
        <eventSubURL>../control/isapictl.dll?channel5</eventSubURL>
        <SCPDURL>../SCPD/FB88Channel5-SCPD.xml</SCPDURL>
      </service>

      <service>
        <serviceType>urn:schemas-upnp-
org:service:channel6:1</serviceType>
        <serviceId>urn:upnp-org:serviceId:channel6</serviceId>
        <controlURL>../control/isapictl.dll?channel6</controlURL>
        <eventSubURL>../control/isapictl.dll?channel6</eventSubURL>
        <SCPDURL>../SCPD/FB88Channel6-SCPD.xml</SCPDURL>
      </service>

      <service>
        <serviceType>urn:schemas-upnp-
org:service:channel7:1</serviceType>
        <serviceId>urn:upnp-org:serviceId:channel7</serviceId>
        <controlURL>../control/isapictl.dll?channel7</controlURL>
        <eventSubURL>../control/isapictl.dll?channel7</eventSubURL>
        <SCPDURL>../SCPD/FB88Channel7-SCPD.xml</SCPDURL>
      </service>

      <service>
        <serviceType>urn:schemas-upnp-
org:service:channel8:1</serviceType>
        <serviceId>urn:upnp-org:serviceId:channel8</serviceId>
        <controlURL>../control/isapictl.dll?channel8</controlURL>
        <eventSubURL>../control/isapictl.dll?channel8</eventSubURL>
        <SCPDURL>../SCPD/FB88Channel8-SCPD.xml</SCPDURL>
      </service>

      <service>
        <serviceType>urn:schemas-upnp-org:service:device:1</serviceType>
        <serviceId>urn:upnp-org:serviceId:device</serviceId>
        <controlURL>../control/isapictl.dll?device</controlURL>
        <eventSubURL>../control/isapictl.dll?device</eventSubURL>
        <SCPDURL>../SCPD/FB88device-SCPD.xml</SCPDURL>
      </service>
```

```
    </serviceList>
   </device>
</root>
```

E.2 Service Description Document for Channel 2

Note that all other service description documents follow the same format.

```
<?xml version="1.0"?>
<scpd xmlns="urn:schemas-upnp-org:service-1-0">
<specVersion>
<major>1</major>
<minor>0</minor>
</specVersion>

  <serviceStateTable>

    <stateVariable>
      <name>InputVolume</name>
      <dataType>i4</dataType>
        <allowedValueRange>
          <minimum>0</minimum>
          <maximum>255</maximum>
          <step>1</step>
        </allowedValueRange>
      <defaultValue>0</defaultValue>
    </stateVariable>

    <stateVariable>
      <name>OutputVolume</name>
      <dataType>i4</dataType>
        <allowedValueRange>
          <minimum>0</minimum>
          <maximum>255</maximum>
          <step>1</step>
        </allowedValueRange>
      <defaultValue>0</defaultValue>
    </stateVariable>

    <stateVariable>
      <name>InputTrim</name>
      <dataType>i4</dataType>
        <allowedValueRange>
          <minimum>0</minimum>
          <maximum>1</maximum>
          <step>1</step>
        </allowedValueRange>
      <defaultValue>0</defaultValue>
    </stateVariable>

    <stateVariable>
      <name>OutputTrim</name>
      <dataType>i4</dataType>
        <allowedValueRange>
          <minimum>0</minimum>
          <maximum>1</maximum>
```

```
        <step>1</step>
      </allowedValueRange>
    <defaultValue>0</defaultValue>
  </stateVariable>

  <stateVariable>
    <name>MonitorTrim</name>
    <dataType>i4</dataType>
      <allowedValueRange>
        <minimum>0</minimum>
        <maximum>1</maximum>
        <step>1</step>
      </allowedValueRange>
    <defaultValue>0</defaultValue>
  </stateVariable>

  <stateVariable>
    <name>MonitorVolume</name>
    <dataType>i4</dataType>
      <allowedValueRange>
        <minimum>0</minimum>
        <maximum>255</maximum>
        <step>1</step>
      </allowedValueRange>
    <defaultValue>0</defaultValue>
  </stateVariable>

  <stateVariable>
    <name>MonitorMix</name>
    <dataType>i4</dataType>
      <allowedValueRange>
        <minimum>0</minimum>
        <maximum>16777215</maximum>
        <step>1</step>
      </allowedValueRange>
    <defaultValue>0</defaultValue>
  </stateVariable>
</serviceStateTable>

<actionList>

  <action>
    <name>IncreaseInputVolume</name>
  </action>

  <action>
    <name>DecreaseInputVolume</name>
  </action>

  <action>
    <name>IncreaseOuputVolume</name>
  </action>

  <action>
    <name>DecreaseOutputVolume</name>
  </action>
```

```
    <action>
      <name>IncreaseInputTrim</name>
    </action>

    <action>
      <name>DecreaseInputTrim</name>
    </action>

    <action>
      <name>IncreaseOutputTrim</name>
    </action>

    <action>
      <name>DecreaseOutputTrim</name>
    </action>

  </actionList>


</scpd>
```

E.3 Presentation Page

```
<!DOCTYPE HTML PUBLIC "-//W3C/DTD HTML 4.0 Transitional//EN"
<HTML>
<HEAD>
      <TITLE> Presentation and Control Page for the FB-88 </TITLE>
</HEAD>

<BODY> <BR>

<H3 ALIGN=center>Presentation and Control Page for the FB-88! </H3>

<TABLE BORDER=1 VALIGN="top" ALIGN=center CELLPADDING=1 CELLSPACING=3>
<TR>
     <TD BGCOLOR=black VALIGN=center ALIGN=middle WIDTH="60">
        <B>
          <FONT SIZE="3" COLOR=whitesmoke>Channel</FONT>
        </B>
     </TD>
     <TD BGCOLOR=black VALIGN=center ALIGN=middle WIDTH="60"
colspan="2">
        <B>
          <FONT SIZE="3" COLOR=whitesmoke>One</FONT>
        </B>
     </TD>
     <TD BGCOLOR=black VALIGN=center ALIGN=middle WIDTH="60"
colspan="2">
        <B>
          <FONT SIZE="3" COLOR=whitesmoke>Two</FONT>
        </B>
     </TD>
     <TD BGCOLOR=black VALIGN=center ALIGN=middle WIDTH="60"
colspan="2">
        <B>
          <FONT SIZE="3" COLOR=whitesmoke>Three</FONT>
```

```
        </B>
      </TD>
      <TD BGCOLOR=black VALIGN=center ALIGN=middle WIDTH="60"
colspan="2">
        <B>
          <FONT SIZE="3" COLOR=whitesmoke>Four</FONT>
        </B>
      </TD>
      <TD BGCOLOR=black VALIGN=center ALIGN=middle WIDTH="60"
colspan="2">
        <B>
          <FONT SIZE="3" COLOR=whitesmoke>Five</FONT>
        </B>
      </TD>
      <TD BGCOLOR=black VALIGN=center ALIGN=middle WIDTH="60"
colspan="2">
        <B>
          <FONT SIZE="3" COLOR=whitesmoke>Six</FONT>
        </B>
      </TD>
      <TD BGCOLOR=black VALIGN=center ALIGN=middle WIDTH="60"
colspan="2">
        <B>
          <FONT SIZE="3" COLOR=whitesmoke>Seven</FONT>
        </B>
      </TD>
      <TD BGCOLOR=black VALIGN=center ALIGN=middle WIDTH="60"
colspan="2">
        <B>
          <FONT SIZE="3" COLOR=whitesmoke>Eight</FONT>
        </B>
      </TD>
</TR>


<TR>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        Input Volume
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseInputVolume(1) type=button value=^ DESIGNTIMESP="19963">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=InVolumeChannelOne></P>
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseInputVolume(2) type=button value=^ DESIGNTIMESP="19967">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=InVolumeChannelTwo></P>
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseInputVolume(3) type=button value=^ DESIGNTIMESP="19971">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=InVolumeChannelThree></P>
      </TD>
```

```
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseInputVolume(4) type=button value=^ DESIGNTIMESP="19975">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=InVolumeChannelFour></P>
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseInputVolume(5) type=button value=^ DESIGNTIMESP="19979">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=InVolumeChannelFive></P>
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseInputVolume(6) type=button value=^ DESIGNTIMESP="19983">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=InVolumeChannelSix></P>
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseInputVolume(7) type=button value=^ DESIGNTIMESP="19987">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=InVolumeChannelSeven></P>
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseInputVolume(8) type=button value=^ DESIGNTIMESP="19991">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=InVolumeChannelEight></P>
      </TD>
</TR>
<TR>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseInputVolume(1)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseInputVolume(2)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseInputVolume(3)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseInputVolume(4)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseInputVolume(5)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseInputVolume(6)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseInputVolume(7)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseInputVolume(8)" value="v">
      </TD>
</TR>
```

```
<TR>
     <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
       Output Volume
     </TD>
     <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseOutputVolume(1) type=button value=^
DESIGNTIMESP="20014">
     </TD>
     <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
       <P ID=OutVolumeChannelOne></P>
     </TD>
     <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseOutputVolume(2) type=button value=^
DESIGNTIMESP="20018">
     </TD>
     <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
       <P ID=OutVolumeChannelTwo></P>
     </TD>
     <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseOutputVolume(3) type=button value=^
DESIGNTIMESP="20022">
     </TD>
     <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
       <P ID=OutVolumeChannelThree></P>
     </TD>
     <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseOutputVolume(4) type=button value=^
DESIGNTIMESP="20026">
     </TD>
     <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
       <P ID=OutVolumeChannelFour></P>
     </TD>
     <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseOutputVolume(5) type=button value=^
DESIGNTIMESP="20030">
     </TD>
     <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
       <P ID=OutVolumeChannelFive></P>
     </TD>
     <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseOutputVolume(6) type=button value=^
DESIGNTIMESP="20034">
     </TD>
     <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
       <P ID=OutVolumeChannelSix></P>
     </TD>
     <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseOutputVolume(7) type=button value=^
DESIGNTIMESP="20038">
     </TD>
     <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
       <P ID=OutVolumeChannelSeven></P>
     </TD>
     <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseOutputVolume(8) type=button value=^
DESIGNTIMESP="20042">
```

```
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=OutVolumeChannelEight></P>
      </TD>
</TR>
<TR>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseOutputVolume(1)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseOutputVolume(2)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseOutputVolume(3)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseOutputVolume(4)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseOutputVolume(5)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseOutputVolume(6)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseOutputVolume(7)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseOutputVolume(8)" value="v">
      </TD>
</TR>

<TR>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        Input Trim
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseInputTrim(1) type=button value=^ DESIGNTIMESP="20065">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=InTrimChannelOne></P>
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseInputTrim(2) type=button value=^ DESIGNTIMESP="20069">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=InTrimChannelTwo></P>
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseInputTrim(3) type=button value=^ DESIGNTIMESP="20073">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=InTrimChannelThree></P>
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseInputTrim(4) type=button value=^ DESIGNTIMESP="20077">
      </TD>
```

```
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=InTrimChannelFour></P>
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseInputTrim(5) type=button value=^ DESIGNTIMESP="20081">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=InTrimChannelFive></P>
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseInputTrim(6) type=button value=^ DESIGNTIMESP="20085">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=InTrimChannelSix></P>
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseInputTrim(7) type=button value=^ DESIGNTIMESP="20089">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=InTrimChannelSeven></P>
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseInputTrim(8) type=button value=^ DESIGNTIMESP="20093">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=InTrimChannelEight></P>
      </TD>
</TR>
<TR>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseInputTrim(1)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseInputTrim(2)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseInputTrim(3)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseInputTrim(4)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseInputTrim(5)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseInputTrim(6)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseInputTrim(7)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseInputTrim(8)" value="v">
      </TD>
</TR>

<TR>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
```

```
      Output Trim
    </TD>
    <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseOutputTrim(1) type=button value=^ DESIGNTIMESP="20116">
    </TD>
    <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
      <P ID=OutTrimChannelOne></P>
    </TD>
    <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseOutputTrim(2) type=button value=^ DESIGNTIMESP="20120">
    </TD>
    <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
      <P ID=OutTrimChannelTwo></P>
    </TD>
    <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseOutputTrim(3) type=button value=^ DESIGNTIMESP="20124">
    </TD>
    <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
      <P ID=OutTrimChannelThree></P>
    </TD>
    <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseOutputTrim(4) type=button value=^ DESIGNTIMESP="20128">
    </TD>
    <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
      <P ID=OutTrimChannelFour></P>
    </TD>
    <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseOutputTrim(5) type=button value=^ DESIGNTIMESP="20132">
    </TD>
    <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
      <P ID=OutTrimChannelFive></P>
    </TD>
    <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseOutputTrim(6) type=button value=^ DESIGNTIMESP="20136">
    </TD>
    <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
      <P ID=OutTrimChannelSix></P>
    </TD>
    <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseOutputTrim(7) type=button value=^ DESIGNTIMESP="20140">
    </TD>
    <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
      <P ID=OutTrimChannelSeven></P>
    </TD>
    <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseOutputTrim(8) type=button value=^ DESIGNTIMESP="20144">
    </TD>
    <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
      <P ID=OutTrimChannelEight></P>
    </TD>
</TR>
<TR>
    <TD VALIGN=center ALIGN=middle WIDTH=60>
      <INPUT type="button" onclick="DecreaseOutputTrim(1)" value="v">
    </TD>
    <TD VALIGN=center ALIGN=middle WIDTH=60>
      <INPUT type="button" onclick="DecreaseOutputTrim(2)" value="v">
```

```
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseOutputTrim(3)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseOutputTrim(4)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseOutputTrim(5)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseOutputTrim(6)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseOutputTrim(7)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseOutputTrim(8)" value="v">
      </TD>
</TR>

<TR>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        Monitor Volume
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseMonitorVolume(1) type=button value=^
DESIGNTIMESP="20167">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=MonitorVolumeChannelOne></P>
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseMonitorVolume(2) type=button value=^
DESIGNTIMESP="20171">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=MonitorVolumeChannelTwo></P>
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseMonitorVolume(3) type=button value=^
DESIGNTIMESP="20175">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=MonitorVolumeChannelThree></P>
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseMonitorVolume(4) type=button value=^
DESIGNTIMESP="20179">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=MonitorVolumeChannelFour></P>
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseMonitorVolume(5) type=button value=^
DESIGNTIMESP="20183">
      </TD>
```

```
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=MonitorVolumeChannelFive></P>
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseMonitorVolume(6) type=button value=^
DESIGNTIMESP="20187">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=MonitorVolumeChannelSix></P>
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseMonitorVolume(7) type=button value=^
DESIGNTIMESP="20191">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=MonitorVolumeChannelSeven></P>
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseMonitorVolume(8) type=button value=^
DESIGNTIMESP="20195">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=MonitorVolumeChannelEight></P>
      </TD>
</TR>
<TR>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseMonitorVolume(1)"
value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseMonitorVolume(2)"
value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseMonitorVolume(3)"
value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseMonitorVolume(4)"
value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseMonitorVolume(5)"
value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseMonitorVolume(6)"
value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseMonitorVolume(7)"
value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseMonitorVolume(8)"
value="v">
```

```
      </TD>
</TR>

<TR>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        Monitor Trim
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseMonitorTrim(1) type=button value=^ DESIGNTIMESP="20218">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=MonitorTrimChannelOne></P>
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseMonitorTrim(2) type=button value=^ DESIGNTIMESP="20222">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=MonitorTrimChannelTwo></P>
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseMonitorTrim(3) type=button value=^ DESIGNTIMESP="20226">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=MonitorTrimChannelThree></P>
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseMonitorTrim(4) type=button value=^ DESIGNTIMESP="20230">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=MonitorTrimChannelFour></P>
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseMonitorTrim(5) type=button value=^ DESIGNTIMESP="20234">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=MonitorTrimChannelFive></P>
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseMonitorTrim(6) type=button value=^ DESIGNTIMESP="20238">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=MonitorTrimChannelSix></P>
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseMonitorTrim(7) type=button value=^ DESIGNTIMESP="20242">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=MonitorTrimChannelSeven></P>
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseMonitorTrim(8) type=button value=^ DESIGNTIMESP="20246">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=MonitorTrimChannelEight></P>
      </TD>
</TR>
<TR>
```

```
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseMonitorTrim(1)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseMonitorTrim(2)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseMonitorTrim(3)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseMonitorTrim(4)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseMonitorTrim(5)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseMonitorTrim(6)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseMonitorTrim(7)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseMonitorTrim(8)" value="v">
      </TD>
</TR>

<TR>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        Monitor Mix
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseMonitorMix(1) type=button value=^ DESIGNTIMESP="20269">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=MonitorMixChannelOne></P>
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseMonitorMix(2) type=button value=^ DESIGNTIMESP="20273">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=MonitorMixChannelTwo></P>
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseMonitorMix(3) type=button value=^ DESIGNTIMESP="20277">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=MonitorMixChannelThree></P>
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseMonitorMix(4) type=button value=^ DESIGNTIMESP="20281">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=MonitorMixChannelFour></P>
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseMonitorMix(5) type=button value=^ DESIGNTIMESP="20285">
      </TD>
```

```
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=MonitorMixChannelFive></P>
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseMonitorMix(6) type=button value=^ DESIGNTIMESP="20289">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=MonitorMixChannelSix></P>
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseMonitorMix(7) type=button value=^ DESIGNTIMESP="20293">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=MonitorMixChannelSeven></P>
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60"><INPUT
onclick=IncreaseMonitorMix(8) type=button value=^ DESIGNTIMESP="20297">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH="60" rowspan=2>
        <P ID=MonitorMixChannelEight></P>
      </TD>
</TR>
<TR>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseMonitorMix(1)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseMonitorMix(2)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseMonitorMix(3)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseMonitorMix(4)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseMonitorMix(5)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseMonitorMix(6)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseMonitorMix(7)" value="v">
      </TD>
      <TD VALIGN=center ALIGN=middle WIDTH=60>
        <INPUT type="button" onclick="DecreaseMonitorMix(8)" value="v">
      </TD>
</TR>
</TABLE>
<P>


      <SCRIPT language=VBScript>

            Sub eventHandler (callbackType, svcObj, varName, value)
            Dim output
            output = output & "callbackType " & callbackType & vbCrLf
```

```
            output = output & "varName " &varName & vbCrLf
            output = output & "value " & value & vbCrLf
            output = output & "svcObj " & svcObj.Id & vbCrLf
            'MsgBox output
                  If (callbackType = "VARIABLE_UPDATE") Then
                     select case svcObj.Id
                      case "urn:upnp-org:serviceId:channel1"
                        select case varName
                              Case "InputVolume"
                                    InVolumeChannelOne.innerText = value
                        Case "OutputVolume"
                                    OutVolumeChannelOne.innerText =
value
                        Case "InputTrim"
                                    InTrimChannelOne.innerText = value
                        Case "OutputTrim"
                                    OutTrimChannelOne.innerText = value
                        Case "MonitorVolume"
                                    MonitorVolumeChannelOne.innerText =
value
                        Case "MonitorTrim"
                                    MonitorTrimChannelOne.innerText =
value
                        Case "MonitorMix"
                                    MonitorMixChannelOne.innerText =
value
                        end select
                     end select
                  End If
            End Sub

            function IncreaseInputVolume(StrChannel)

            Dim inArgs(1)
            Dim outArgs(0)
            inArgs(0) = 1
                  ChannelService.InvokeAction "IncreaseInputVolume",
inArgs, outArgs
                  'MsgBox("Channel " & StrChannel & " ->
IncreaseInputVolume")
            end function

            function DecreaseInputVolume(StrChannel)
            Dim inArgs(0)
            Dim outArgs(0)
                  ChannelService.InvokeAction "DecreaseInputVolume",
inArgs, outArgs
                  'MsgBox("Channel " & StrChannel & " ->
DecreaseInputVolume")
            end function

            function IncreaseInputTrim(StrChannel)
            Dim inArgs(0)
            Dim outArgs(0)
                  ChannelService.InvokeAction "IncreaseInputTrim",
inArgs, outArgs
```

```
                'MsgBox("Channel " & StrChannel & " ->
IncreasInputTrim")
            end function

            function DecreaseInputTrim(StrChannel)
            Dim inArgs(0)
            Dim outArgs(0)
                ChannelService.InvokeAction "DecreaseInputTrim",
inArgs, outArgs
                'MsgBox("Channel " & StrChannel & " ->
DecreaseInputTrim")
            end function

            function IncreaseOutputVolume(StrChannel)
            Dim inArgs(0)
            Dim outArgs(0)
                ChannelService.InvokeAction "IncreaseOutputVolume",
inArgs, outArgs
                'MsgBox("Channel " & StrChannel & " ->
IncreaseOutputVolume")
            end function

            function DecreaseOutputVolume(StrChannel)
            Dim inArgs(0)
            Dim outArgs(0)
                ChannelService.InvokeAction "DecreaseOutputVolume",
inArgs, outArgs
                'MsgBox("Channel " & StrChannel & " ->
DecreaseOutputVolume")
            end function

            function IncreaseOutputTrim(StrChannel)
            Dim inArgs(0)
            Dim outArgs(0)
                ChannelService.InvokeAction "IncreaseOutputTrim",
inArgs, outArgs
                'MsgBox("Channel " & StrChannel & " ->
IncreaseOutputTrim")
            end function

            function DecreaseOutputTrim(StrChannel)
            Dim inArgs(0)
            Dim outArgs(0)
                ChannelService.InvokeAction "DecreaseOutputTrim",
inArgs, outArgs
                'MsgBox("Channel " & StrChannel & " ->
DecreaseOutputTrim")
            end function

            function IncreaseMonitorVolume(StrChannel)
            Dim inArgs(0)
            Dim outArgs(0)
                ChannelService.InvokeAction "IncreaseMonitorVolume",
inArgs, outArgs
                'MsgBox("Channel " & StrChannel & " ->
IncreasMonitorVolume")
            end function
```

```
            function DecreaseMonitorVolume(StrChannel)
            Dim inArgs(0)
            Dim outArgs(0)
                  ChannelService.InvokeAction "DecreaseMonitorVolume",
inArgs, outArgs
                  'MsgBox("Channel " & StrChannel & " ->
DecreaseMonitorVolume")
            end function

            function IncreaseMonitorTrim(StrChannel)
            Dim inArgs(0)
            Dim outArgs(0)
                  ChannelService.InvokeAction "IncreaseMonitorTrim",
inArgs, outArgs
                  'MsgBox("Channel " & StrChannel & " ->
IncreasMonitorTrim")
            end function

            function DecreaseMonitorTrim(StrChannel)
            Dim inArgs(0)
            Dim outArgs(0)
                  ChannelService.InvokeAction "DecreaseMonitorTrim",
inArgs, outArgs
                  'MsgBox("Channel " & StrChannel & " ->
DecreaseMonitorTrim")
            end function

            function IncreaseMonitorMix(StrChannel)
            Dim inArgs(0)
            Dim outArgs(0)
                  ChannelService.InvokeAction "IncreaseMonitorMix",
inArgs, outArgs
                  'MsgBox("Channel " & StrChannel & " ->
IncreasMonitorMix")
            end function

            function DecreaseMonitorMix(StrChannel)
            Dim inArgs(0)
            Dim outArgs(0)
                  ChannelService.InvokeAction "DecreaseMonitorMix",
inArgs, outArgs
                  'MsgBox("Channel " & StrChannel & " ->
DecreaseMonitorMix")
            end function

            '************************************
            ' Download Description Document
            Dim ChannelDesc
            Set ChannelDesc = CreateObject("UPnP.DescriptionDocument.1")
            ChannelDesc.Load("..\description\FB88-orig.xml")

            '************************************************
            'Get the Root Device
            Dim FB88
            Set FB88 = ChannelDesc.RootDevice
```

```
            '************************************************
            'Output Some useful properties!
            Dim output
            output = "Found: " & vbCrLf
            output = output & "DisplayName: " & FB88.FriendlyName &
vbCrLf
            output = output & "UDN: " & FB88.UniqueDeviceName & vbCrLf
            output = output & "IsRootDevice: " & FB88.IsRootDevice
&vbCrLf

            MsgBox output

            'bob = FB88.Services.Item( "urn:upnp-org:serviceId:channel1"
)

            '************************************************
            'Attach the event handler to this service
            Dim ChannelService
            set ChannelService=FB88.Services("urn:upnp-
org:serviceId:channel1")
            ChannelService.AddCallback GetRef("eventHandler")

      ' Get the service objects
'Dim services
'services = ChannelDesc.RootDevice.Services

'Set ChannelService = FB88.Services( "urn:upnp-org:serviceId:channel1" )
'Set xportService = services( "urn:upnp-org:serviceId:AVTransport" )


      </SCRIPT>
</P>
<P><STRONG><U>Device Control
Parameters</U>  </STRONG></P>
<P>Power</P>
<P>Synchronisation</P>
<P>Frequency</P>
<P>Input mode    </P>
<P>Optical mode</P>
</BODY>
</HTML>
```

E.4 FB-88 Module for Microsoft UPnP Development Kit

```
// Include all the necessary header files.
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include "global.h"
#include "event.h"
#include "util.h"
#include "udevice.h"

#define MAX_IN_VOLUME         255
#define MIN_IN_VOLUME         0
#define MAX_IN_TRIM                 1
#define MIN_IN_TRIM                 0
```

```
#define MAX_OUT_VOLUME          255
#define MIN_OUT_VOLUME          0
#define MAX_OUT_TRIM            1
#define MIN_OUT_TRIM            0
#define MAX_MONITOR_TRIM        1
#define MIN_MONITOR_TRIM        0
#define MAX_MONITOR_VOLUME      255
#define MIN_MONITOR_VOLUME      0
#define MAX_MONITOR_MIX         16777215
#define MIN_MONITOR_MIX         0
#define MAX_SYNC                4
#define MIN_SYNC                0
#define MAX_FREQUENCY           4
#define MIN_FREQUENCY           1
#define MAX_INPUT_MODE          1
#define MIN_INPUT_MODE          0
#define MAX_OPTICAL_MODE        1
#define MIN_OPTICAL_MODE        0

typedef struct
{
      DWORD dwInVolume;             /* InputVolumeLevel */
      DWORD dwInTrim;               /* InputTrimLevel */
      DWORD dwOutVolume;            /* OutputVolumeLevel */
      DWORD dwOutTrim;              /* OutputTrimLevel */
      DWORD   dwMonitorTrim;        /* Monitor Trim */
      DWORD dwMonitorVolume;  /* Monitor Volume */
      DWORD dwMonitorMix;           /* Monitor Mix */
}  FB88_Channel;

struct FB88
{
      FB88_Channel      Ch1;        /* Channel One */
      FB88_Channel      Ch2;        /* Channel Two */
      FB88_Channel      Ch3;        /* Channel Three */
      FB88_Channel      Ch4;        /* Channel Four */
      FB88_Channel      Ch5;        /* Channel Five */
      FB88_Channel      Ch6;        /* Channel Six */
      FB88_Channel      Ch7;        /* Channel Seven */
      FB88_Channel      Ch8;        /* Channel Eight */
//    DWORD       dwSync;                 /* Sync Select */
//    DWORD       dwFrequency;      /* Frequency Select */
//    DWORD       dwInputMode;      /* Input Mode Select */
//    DWORD       dwOpticalMode;    /* Optical Mode */

};

/* Binary Copy of above Device */

FB88 InsFB88 =
{
      {0, 0, 0, 0, 0, 0, 0},  /* Channel 1 */
      {0, 0, 0, 0, 0, 0, 0},  /* Channel 2 */
      {0, 0, 0, 0, 0, 0, 0},  /* Channel 3 */
      {0, 0, 0, 0, 0, 0, 0},  /* Channel 4 */
      {0, 0, 0, 0, 0, 0, 0},  /* Channel 5 */
      {0, 0, 0, 0, 0, 0, 0},  /* Channel 6 */
```

```
        {0, 0, 0, 0, 0, 0, 0},  /* Channel 7 */
        {0, 0, 0, 0, 0, 0, 0},  /* Channel 8 */

//      1,                                  /* Sync */
//      1,                                  /* Frequency */
//      0,                                  /* Input Mode */
//      0                                   /* Optical Mode */
};

// Number of state variables.
static const int nChannelStateVariables = 7;

// String copy of state variables for initializing eventing.
E_PROP_LIST eChannelPropList =
{
    nChannelStateVariables,
    {
        {"InputVolume",       "0", FALSE, TRUE},
        {"InputTrim",         "0", FALSE, TRUE},
            {"OutputVolume",  "0", FALSE, TRUE},
            {"OutputTrim",        "0", FALSE, TRUE},
            {"MonitorTrim",       "0", FALSE, TRUE},
            {"MonitorVolume", "0", FALSE, TRUE},
            {"MonitorMix",        "0", FALSE, TRUE},
    }
};

// Number of state variables.
static const int nDeviceStateVariables = 4;

// String copy of state variables for initializing eventing.
E_PROP_LIST eDevicePropList =
{
    nDeviceStateVariables,
    {
        {"Sync",        "1", FALSE, TRUE},
            {"Frequency",     "1", FALSE, TRUE},
        {"InputMode",   "0", FALSE, TRUE},
            {"OpticalMode", "0", FALSE, TRUE},
    }
};

DWORD Do_Channel_Init    (CHAR* StrEventUrl)
{
   printf("Channel init called");
      BOOL bResult = InitializeUpnpEventSource(StrEventUrl,
&eChannelPropList);

   return bResult ? 1 : 0;
}

DWORD Do_Channel_Cleanup (CHAR* StrEventUrl)
{
  if (!CleanupUpnpEventSource(StrEventUrl))
      puts ("Do_Channel_Cleanup failed to clean up the Upnp Event
Source");
```

```
  return 1;
}

DWORD Do_Device_Init    (CHAR* StrEventUrl)
{
   BOOL bResult = InitializeUpnpEventSource(StrEventUrl,
&eDevicePropList);

   return bResult ? 1 : 0;
}

DWORD Do_Device_Cleanup (CHAR* StrEventUrl)
{
  if (!CleanupUpnpEventSource(StrEventUrl))
      puts ("Do_Channel_Cleanup failed to clean up the Upnp Event
Source");

  return 1;
}

DWORD Do_IncreaseInputVolume (
                          CHAR*          StrEventUrl,
                          DWORD          cArgs,
                          ARG*           rgArgs,
                          PDWORD         pArgsOut,
                          ARG_OUT*       rgArgsOut
                         )
{

      printf("%i input params containing Name: %s, and value %s", cArgs,
rgArgs->szName, rgArgs->szValue );
      InsFB88.Ch1.dwInVolume = atoi(rgArgs->szValue);

      CHAR szValue[32];
      sprintf(szValue, "%u", InsFB88.Ch1.dwInVolume);

      ChangeProp(StrEventUrl, "InputVolume", szValue);
      SubmitPropEvents(StrEventUrl, NULL);
      /*if (InsFB88.Ch1.dwInVolume < MAX_IN_VOLUME) /* Our Upper Limit
*/
      /*{

            InsFB88.Ch1.dwInVolume++;
            CHAR szValue[32];
            sprintf(szValue, "%u", InsFB88.Ch1.dwInVolume);
            ChangeProp(StrEventUrl, "InputVolume", szValue);
            SubmitPropEvents(StrEventUrl, NULL);
      }*/


      return 0;
}


DWORD Do_DecreaseInputVolume (
                          CHAR*          StrEventUrl,
                          DWORD          cArgs,
```

```
                        ARG*            rgArgs,
                        PDWORD          pArgsOut,
                        ARG_OUT*        rgArgsOut
                       )
{
      if (InsFB88.Ch1.dwInVolume > MIN_IN_VOLUME) /* Our Upper Limit */
      {
            InsFB88.Ch1.dwInVolume--;
            CHAR szValue[32];
            sprintf(szValue, "%u", InsFB88.Ch1.dwInVolume);
            ChangeProp(StrEventUrl, "InputVolume", szValue);
            SubmitPropEvents(StrEventUrl, NULL);
      }
      return 0;
}

DWORD Do_IncreaseInputTrim (
                        CHAR*           StrEventUrl,
                        DWORD           cArgs,
                        ARG*            rgArgs,
                        PDWORD          pArgsOut,
                        ARG_OUT*        rgArgsOut
                       )
{
      if (InsFB88.Ch1.dwInTrim < MAX_IN_TRIM) /* Our Upper Limie */
      {
            InsFB88.Ch1.dwInTrim++;
            CHAR szValue[32];
            sprintf(szValue, "%u", InsFB88.Ch1.dwInTrim);
            ChangeProp(StrEventUrl, "InputTrim", szValue);
            SubmitPropEvents(StrEventUrl, NULL);
      }
      return 0;
}

DWORD Do_DecreaseInputTrim (
                        CHAR*           StrEventUrl,
                        DWORD           cArgs,
                        ARG*            rgArgs,
                        PDWORD          pArgsOut,
                        ARG_OUT*        rgArgsOut
                       )
{
      if (InsFB88.Ch1.dwInTrim > MIN_IN_TRIM) /* Our Upper Limie */
      {
            InsFB88.Ch1.dwInTrim--;
            CHAR szValue[32];
            sprintf(szValue, "%u", InsFB88.Ch1.dwInTrim);
            ChangeProp(StrEventUrl, "InputTrim", szValue);
            SubmitPropEvents(StrEventUrl, NULL);
      }

      return 0;
}

DWORD Do_IncreaseOutputVolume (
                        CHAR*           StrEventUrl,
```

```
                        DWORD           cArgs,
                        ARG*            rgArgs,
                        PDWORD          pArgsOut,
                        ARG_OUT*        rgArgsOut
                       )
{
      printf("%i input params containing Name: %s, and value %i", cArgs,
rgArgs->szName, rgArgs->szValue );
      if (InsFB88.Ch1.dwOutVolume < MAX_OUT_VOLUME) /* Our Upper Limit
*/
      {
            InsFB88.Ch1.dwOutVolume++;
            CHAR szValue[32];
            sprintf(szValue, "%u", InsFB88.Ch1.dwOutVolume);
            ChangeProp(StrEventUrl, "OutputVolume", szValue);
            SubmitPropEvents(StrEventUrl, NULL);
      }

      return 0;
}

DWORD Do_DecreaseOutputVolume (
                        CHAR*           StrEventUrl,
                        DWORD           cArgs,
                        ARG*            rgArgs,
                        PDWORD          pArgsOut,
                        ARG_OUT*        rgArgsOut
                       )
{
      return 0;
}

DWORD Do_IncreaseOutputTrim (
                        CHAR*           StrEventUrl,
                        DWORD           cArgs,
                        ARG*            rgArgs,
                        PDWORD          pArgsOut,
                        ARG_OUT*        rgArgsOut
                       )
{
      return 0;
}


DWORD Do_DecreaseOutputTrim (
                        CHAR*           StrEventUrl,
                        DWORD           cArgs,
                        ARG*            rgArgs,
                        PDWORD          pArgsOut,
                        ARG_OUT*        rgArgsOut
                       )
{
      return 0;
}
```

E.5 Alterations to the UDevice Module for the FB-88 Simulated Device

```
// FB88 Device function prototypes
DWORD Do_Channel_Init    (CHAR* StrEventUrl);

DWORD Do_Channel_Cleanup (CHAR* StrEventUrl);

DWORD Do_Device_Init     (CHAR* StrEventUrl);

DWORD Do_Device_Cleanup (CHAR* StrEventUrl);

DWORD Do_IncreaseInputVolume (
                          CHAR*          StrEventUrl,
                          DWORD          cArgs,
                          ARG*           rgArgs,
                          PDWORD         pArgsOut,
                          ARG_OUT*       rgArgsOut
                         );

DWORD Do_DecreaseInputVolume (
                          CHAR*          StrEventUrl,
                          DWORD          cArgs,
                          ARG*           rgArgs,
                          PDWORD         pArgsOut,
                          ARG_OUT*       rgArgsOut
                         );

DWORD Do_IncreaseInputTrim (
                          CHAR*          StrEventUrl,
                          DWORD          cArgs,
                          ARG*           rgArgs,
                          PDWORD         pArgsOut,
                          ARG_OUT*       rgArgsOut
                         );

DWORD Do_DecreaseInputTrim (
                          CHAR*          StrEventUrl,
                          DWORD          cArgs,
                          ARG*           rgArgs,
                          PDWORD         pArgsOut,
                          ARG_OUT*       rgArgsOut
                         );

DWORD Do_IncreaseOutputVolume (
                          CHAR*          StrEventUrl,
                          DWORD          cArgs,
                          ARG*           rgArgs,
                          PDWORD         pArgsOut,
                          ARG_OUT*       rgArgsOut
                         );

DWORD Do_DecreaseOutputVolume (
                          CHAR*          StrEventUrl,
                          DWORD          cArgs,
                          ARG*           rgArgs,
                          PDWORD         pArgsOut,
                          ARG_OUT*       rgArgsOut
                         );
```

```
DWORD Do_IncreaseOutputTrim (
                          CHAR*          StrEventUrl,
                          DWORD          cArgs,
                          ARG*           rgArgs,
                          PDWORD         pArgsOut,
                          ARG_OUT*       rgArgsOut
                         );

DWORD Do_DecreaseOutputTrim (
                          CHAR*          StrEventUrl,
                          DWORD          cArgs,
                          ARG*           rgArgs,
                          PDWORD         pArgsOut,
                          ARG_OUT*       rgArgsOut
                         );


// Add SERVICE_CTL entries for each device.

// c_cDemoSvc must be set to the number of control IDs in c_rgSvc
(below).
// Change this number if you add entries to c_rgSvc.
#define c_cDemoSvc 9

// The table that follows is a method of associating UPDIAG control
identifiers with
// device-specific function calls.

const SERVICE_CTL c_rgSvc[c_cDemoSvc] =
{
      {
        "device",                                   //Service control
identifier.
            (PFNAI)Do_Device_Init,                  //Device-specific
initialization function.
            (PFNAC)Do_Device_Cleanup,               //Device-specific
cleanup function.
            8,                                      //Number of actions this
service implements.
        {
            { "IncreaseInputVolume",
      (PFNAS)Do_IncreaseInputVolume      },
            { "DecreaseInputVolume",
      (PFNAS)Do_DecreaseInputVolume     },
            { "IncreaseInputTrim",
      (PFNAS)Do_IncreaseInputTrim   },
            { "DecreaseInputTrim",
      (PFNAS)Do_DecreaseInputTrim   },
                  { "IncreaseOutputVolume",
(PFNAS)Do_IncreaseOutputVolume      },

            { "DecreaseOutputVolume",
(PFNAS)Do_DecreaseOutputVolume     },
            { "IncreaseOutputTrim",
      (PFNAS)Do_IncreaseOutputTrim   },
```

```
            { "DecreaseOutputTrim",
      (PFNAS)Do_DecreaseOutputTrim   },
        },
    },
          {
          "channel1",                                  //Service control
identifier.
           (PFNAI)Do_Channel_Init,                    //Device-specific
initialization function.
           (PFNAC)Do_Channel_Cleanup,                  //Device-specific
cleanup function.
           8,                                         //Number of actions this
service implements.
        {
            { "IncreaseInputVolume",
(PFNAS)Do_IncreaseInputVolume      },
            { "DecreaseInputVolume",
      (PFNAS)Do_DecreaseInputVolume     },
            { "IncreaseInputTrim",
      (PFNAS)Do_IncreaseInputTrim   },
            { "DecreaseInputTrim",
      (PFNAS)Do_DecreaseInputTrim   },
                { "IncreaseOutputVolume",
(PFNAS)Do_IncreaseOutputVolume      },
            { "DecreaseOutputVolume",
(PFNAS)Do_DecreaseOutputVolume     },
            { "IncreaseOutputTrim",
      (PFNAS)Do_IncreaseOutputTrim   },
            { "DecreaseOutputTrim",
      (PFNAS)Do_DecreaseOutputTrim   },
        },
    },

      {
        "channel2",                                   //Service control
identifier.
           (PFNAI)Do_Channel_Init,                    //Device-specific
initialization function.
           (PFNAC)Do_Channel_Cleanup,                 //Device-specific
cleanup function.
           8,                                         //Number of actions this
service implements.
        {
            { "IncreaseInputVolume",
(PFNAS)Do_IncreaseInputVolume      },
            { "DecreaseInputVolume",
      (PFNAS)Do_DecreaseInputVolume     },
            { "IncreaseInputTrim",
      (PFNAS)Do_IncreaseInputTrim   },
            { "DecreaseInputTrim",
      (PFNAS)Do_DecreaseInputTrim   },
                { "IncreaseOutputVolume",
(PFNAS)Do_IncreaseOutputVolume      },
            { "DecreaseOutputVolume",
(PFNAS)Do_DecreaseOutputVolume     },
            { "IncreaseOutputTrim",
      (PFNAS)Do_IncreaseOutputTrim   },
```

```
            { "DecreaseOutputTrim",
      (PFNAS)Do_DecreaseOutputTrim   },
        },
    },
      {
        "channel3",                                  //Service control
identifier.
            (PFNAI)Do_Channel_Init,                  //Device-specific
initialization function.
            (PFNAC)Do_Channel_Cleanup,               //Device-specific
cleanup function.
            8,                                      //Number of actions this
service implements.
        {
            { "IncreaseInputVolume",
(PFNAS)Do_IncreaseInputVolume      },
            { "DecreaseInputVolume",
      (PFNAS)Do_DecreaseInputVolume     },
            { "IncreaseInputTrim",
      (PFNAS)Do_IncreaseInputTrim   },
            { "DecreaseInputTrim",
      (PFNAS)Do_DecreaseInputTrim   },
                  { "IncreaseOutputVolume",
(PFNAS)Do_IncreaseOutputVolume      },
            { "DecreaseOutputVolume",
(PFNAS)Do_DecreaseOutputVolume     },
            { "IncreaseOutputTrim",
      (PFNAS)Do_IncreaseOutputTrim   },
            { "DecreaseOutputTrim",
      (PFNAS)Do_DecreaseOutputTrim   },
        },
    },
      {
        "channel4",                                  //Service control
identifier.
            (PFNAI)Do_Channel_Init,                  //Device-specific
initialization function.
            (PFNAC)Do_Channel_Cleanup,               //Device-specific
cleanup function.
            8,                                      //Number of actions this
service implements.
        {
            { "IncreaseInputVolume",
(PFNAS)Do_IncreaseInputVolume      },
            { "DecreaseInputVolume",
      (PFNAS)Do_DecreaseInputVolume     },
            { "IncreaseInputTrim",
      (PFNAS)Do_IncreaseInputTrim   },
            { "DecreaseInputTrim",
      (PFNAS)Do_DecreaseInputTrim   },
                  { "IncreaseOutputVolume",
(PFNAS)Do_IncreaseOutputVolume      },
            { "DecreaseOutputVolume",
(PFNAS)Do_DecreaseOutputVolume     },
            { "IncreaseOutputTrim",
      (PFNAS)Do_IncreaseOutputTrim   },
```

```
            { "DecreaseOutputTrim",
      (PFNAS)Do_DecreaseOutputTrim   },
        },
    },
      {
        "channel5",                                  //Service control
identifier.
            (PFNAI)Do_Channel_Init,                  //Device-specific
initialization function.
            (PFNAC)Do_Channel_Cleanup,               //Device-specific
cleanup function.
            8,                                      //Number of actions this
service implements.
        {
            { "IncreaseInputVolume",
(PFNAS)Do_IncreaseInputVolume      },
            { "DecreaseInputVolume",
      (PFNAS)Do_DecreaseInputVolume     },
            { "IncreaseInputTrim",
      (PFNAS)Do_IncreaseInputTrim   },
            { "DecreaseInputTrim",
      (PFNAS)Do_DecreaseInputTrim   },
                { "IncreaseOutputVolume",
(PFNAS)Do_IncreaseOutputVolume      },
            { "DecreaseOutputVolume",
(PFNAS)Do_DecreaseOutputVolume     },
            { "IncreaseOutputTrim",
      (PFNAS)Do_IncreaseOutputTrim   },
            { "DecreaseOutputTrim",
      (PFNAS)Do_DecreaseOutputTrim   },
        },
    },
      {
        "channel6",                                  //Service control
identifier.
            (PFNAI)Do_Channel_Init,                  //Device-specific
initialization function.
            (PFNAC)Do_Channel_Cleanup,               //Device-specific
cleanup function.
            8,                                      //Number of actions this
service implements.
        {
            { "IncreaseInputVolume",
(PFNAS)Do_IncreaseInputVolume      },
            { "DecreaseInputVolume",
      (PFNAS)Do_DecreaseInputVolume     },
            { "IncreaseInputTrim",
      (PFNAS)Do_IncreaseInputTrim   },
            { "DecreaseInputTrim",
      (PFNAS)Do_DecreaseInputTrim   },
                { "IncreaseOutputVolume",
(PFNAS)Do_IncreaseOutputVolume      },
            { "DecreaseOutputVolume",
(PFNAS)Do_DecreaseOutputVolume     },
            { "IncreaseOutputTrim",
      (PFNAS)Do_IncreaseOutputTrim   },
```

```
            { "DecreaseOutputTrim",
      (PFNAS)Do_DecreaseOutputTrim    },
        },
    },
      {
        "channel7",                                  //Service control
identifier.
            (PFNAI)Do_Channel_Init,                  //Device-specific
initialization function.
            (PFNAC)Do_Channel_Cleanup,               //Device-specific
cleanup function.
            8,                                      //Number of actions this
service implements.
        {
            { "IncreaseInputVolume",
(PFNAS)Do_IncreaseInputVolume       },
            { "DecreaseInputVolume",
      (PFNAS)Do_DecreaseInputVolume      },
            { "IncreaseInputTrim",
      (PFNAS)Do_IncreaseInputTrim    },
            { "DecreaseInputTrim",
      (PFNAS)Do_DecreaseInputTrim    },
                  { "IncreaseOutputVolume",
(PFNAS)Do_IncreaseOutputVolume       },
            { "DecreaseOutputVolume",
(PFNAS)Do_DecreaseOutputVolume      },
            { "IncreaseOutputTrim",
      (PFNAS)Do_IncreaseOutputTrim    },
            { "DecreaseOutputTrim",
      (PFNAS)Do_DecreaseOutputTrim    },
        },
    },
      {
        "channel8",                                  //Service control
identifier.
            (PFNAI)Do_Channel_Init,                  //Device-specific
initialization function.
            (PFNAC)Do_Channel_Cleanup,               //Device-specific
cleanup function.
            8,                                      //Number of actions this
service implements.
        {
            { "IncreaseInputVolume",
(PFNAS)Do_IncreaseInputVolume       },
            { "DecreaseInputVolume",
      (PFNAS)Do_DecreaseInputVolume      },
            { "IncreaseInputTrim",
      (PFNAS)Do_IncreaseInputTrim    },
            { "DecreaseInputTrim",
      (PFNAS)Do_DecreaseInputTrim    },
                  { "IncreaseOutputVolume",
(PFNAS)Do_IncreaseOutputVolume       },
            { "DecreaseOutputVolume",
(PFNAS)Do_DecreaseOutputVolume      },
            { "IncreaseOutputTrim",
      (PFNAS)Do_IncreaseOutputTrim    },
```

```
            { "DecreaseOutputTrim",
      (PFNAS)Do_DecreaseOutputTrim    },
        },
    },
};


// Define the product name that will appear in HTTP SERVER header.
const CHAR c_szProduct[] = "DevKit Sample";
const CHAR c_szProductVersion[] = "1.0";
```