A Proxy Approach to Protocol Interoperability within Digital Audio Networks

Submitted in fulfilment
of the requirements of the degree
MASTERS OF SCIENCE
of Rhodes University

by
OSEDUM P. IGUMBOR

September 2009

Abstract

Digital audio networks are becoming the preferred solution for the interconnection of professional audio devices. Prominent amongst their advantages are: reduced noise interference, signal multiplexing, and a reduction in the number of cables connecting networked devices. In the context of professional audio, digital networks have been used to connect devices including: mixers, effects units, preamplifiers, breakout boxes, computers, monitoring controllers, and synthesizers. Such networks are governed by protocols that define the connection management procedures, and device synchronization processes of devices that conform to the protocols.

A wide range of digital audio network control protocols exist, each defining specific hardware requirements of devices that conform to them. Device parameter control is achieved by sending a protocol message that indicates the target parameter, and the action that should be performed on the parameter. Typically, a device will conform to only one protocol. By implication, only devices that conform to a specific protocol can communicate with each other, and only a controller that conforms to the protocol can control such devices. This results in the isolation of devices that conform to disparate protocols, since devices of different protocols cannot communicate with each other. This is currently a challenge in the professional music industry, particularly where digital networks are used for audio device control.

This investigation seeks to resolve the issue of interoperability between professional audio devices that conform to different digital audio network protocols. This thesis proposes the use of a proxy that allows for the translation of protocol messages, as a solution to the interoperability problem. The proxy abstracts devices of one protocol in terms of another, hence allowing all the networked devices to appear as conforming to the same protocol. The proxy receives messages on behalf of the abstracted device, and then fulfills them in accordance with the protocol that the abstracted device conforms to. Any number of protocol devices can be abstracted within such a proxy. This has the added advantage of allowing a common controller to control devices that conform to the different protocols.

Acknowledgements

Many thanks to my supervisor Prof. Richard Foss for his patience, encouragement, insight and direction throughout the course of this research.

Thanks to my colleagues in the Audio Research Group, who entertained my endless questions with regards to XFN implementation.

I am grateful to my family for their sacrifices, motivation and support throughout the course of my study.

Finally, I will like to thank the *Department of Labour* (DST), and the *Distributed Multimedia Centre of Excellence* at Rhodes University with its sponsors, for their financial assistance.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Professional audio devices are used in many contexts including live concerts, the hospitality industry, recording studios, places of worship, stadiums, and broadcast stations. In such places, more than one (professional) audio device is used to transmit, process and distribute audio. There are many types of commercially available professional audio (*proaudio*) devices such as mixing consoles, effects units, breakout boxes, preamplifiers (preamps), and signal processing units. Often it is necessary to connect multiple proaudio devices to attain a desired effect in the produced sound. For instance, a mixing console might be used to mix and output the audio it 'receives' from a microphone and a synthesizer.

It is desirable to network proaudio devices for routing signals and also for controlling device parameters. For example, a sound engineer might want to perform signal routing and device parameter control while seated stationary in front of a computer. Such a situation is becoming a popular feature of proaudio contexts, such as live concerts, where multiple devices can be networked and controlled from a single point by a sound/audio engineer.

The connection between proaudio devices has typically been achieved by using point-to-point cabling of the devices. In large installations, this entails running long cables, each cable with its own connectors on both ends, in order to connect the devices. There exist different types of cable connectors used to connect proaudio devices. These include RCA, ADAT, S/PDIF, MIDI, stereo jacks, MADI and AES/EBU. Using these connectors with their corresponding cables, to connect proaudio devices, it is possible to send audio, and in the case of MIDI, control instructions to parameters within a device. For example MIDI instructions can be exchanged between a synthesizer and a computer.

One obvious feature of a point-to-point network of proaudio devices, is the huge clutter of cables

that are required to connect the devices. With a large number of devices in a proaudio context, many cables are connected from one device to another in order to route signals amongst the devices. This cluttering of cables, particularly in proaudio installations, where many physical point-to-point connections are required, makes it difficult to trace the signal path of audio as it routes from one device to another. Such a scenario for a simple project studio, is depicted in figure 1.1.

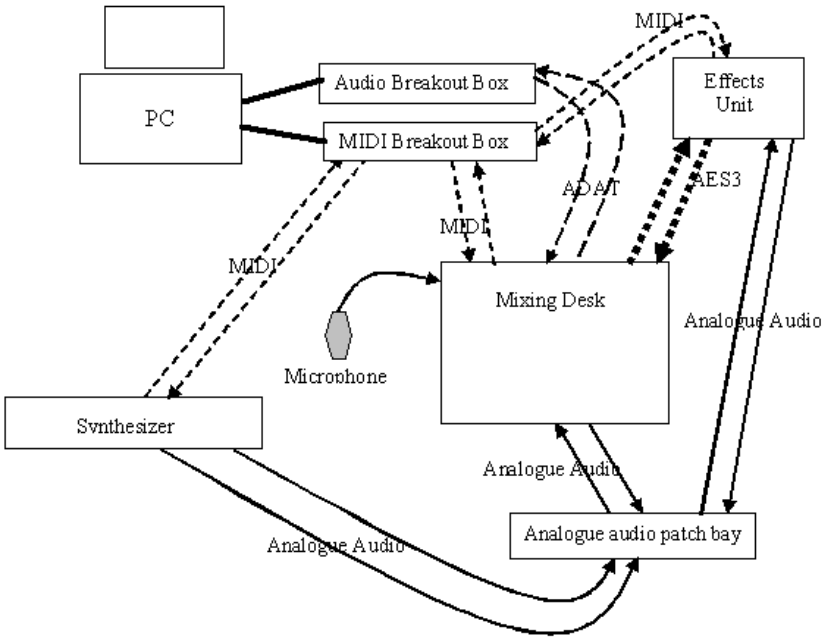


Figure 1.1: Project audio device studio with point-to-point cabling

In figure 1.1, analog audio is input to the *mixing desk* from a *microphone* that is plugged into a *balanced* input on the *mixing desk*. The *mixing desk*, which converts the analog audio from the *microphone* to digital signals, can have some reverberation effects added to its input audio by routing it via *AES/EBU* connectors to the *effects unit*. Audio data is routed from the *PC* through the *audio breakout box* to the mixing desk using *ADAT* cables. MIDI is sent to and from all devices via the *MIDI breakout box*. Audio from the *effects units*, *mixing desk* and *synthesizer* is routed to the *analogue audio patch bay* via *stereo jack* connectors. The *analogue audio patchbay* allows for the patching of audio signals from one audio cable to another, thus providing signal switching capability for analog audio signals.

The use of digital (computer) networks, with their associated routing capability, is gradually becoming a popular solution to overcome the rather cluttered configuration in figure 1.1. Such

networks also allow for easy signal routing and device parameter control from a single point - the computer. It is now possible to network proaudio devices using various network technologies, including IEEE 1394 serial bus, Ethernet, and ATM. There are various network topologies that are used (including star, ring, and mesh network topologies) on the aforementioned network technologies, but figure 1.2 shows a daisy-chain network of proaudio devices.

Figure 1.2 depicts a *PC* running a connection management and device control application, a *mixing desk*, *effects unit*, *synthesizer*, and *breakout box*, all daisy-chained to form a digital audio network.

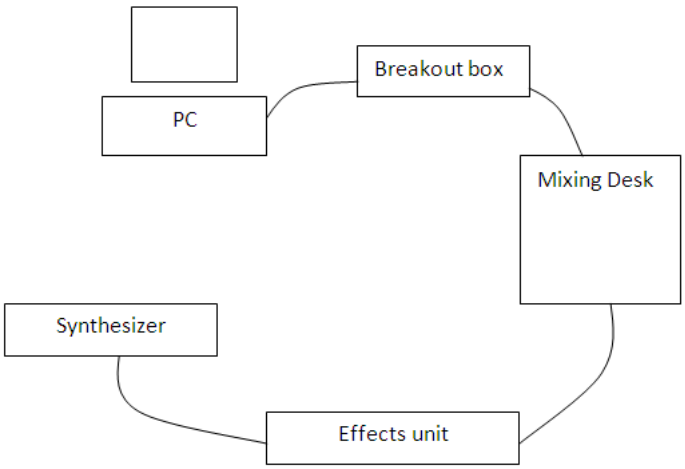


Figure 1.2: Digital audio network

The elegant layout of a digital network (as shown in figure 1.2) presents various other benefits including:

- ease of monitoring networked devices
- remote device control by sending appropriate instructions
- ease of tracing signals, and
- signal routing amongst the networked devices

Within a digital audio network, audio is transmitted in a digital form, and all audio input signals are converted to digital audio for transmission on the network. The digital audio is typically

framed into packets for transmission on the network, for example in IEEE 1394 and Ethernet networks.

Gross (2004) mentions the advantages of a digital audio network over an analog network (similar to that depicted in figure 1.1):

- Analog audio is more susceptible to noise and cross-talk interferences. Such interferences are immensely reduced or completely eliminated (using fiber) in a digital audio network.
- Digital audio networks allow for multiplexing of digital signals. This allows multiple channels of audio over a single physical connection. This reduces the cost of cabling.
- Digital audio networks are capable of providing data integrity by providing error checking and possibly error correction mechanism for transmitted digital media.
- Due to the proliferation of digital end user devices, such as iPods and MP3 players, it becomes attractive to distribute audio digitally (via a digital network) since it reduces the cost of having to provide several analog-to-digital and digital-to-analog conversions of signals. Furthermore, this eliminates possible degradation of signals due to such conversions.
- In the case of multiple access on a digital network, for instance networked IEEE 1394 devices, a centralized control by a user will allow for device configuration and control by sending messages on the network.
- A multiple access digital network also reduces the number of cables that is required to network audio devices in such audio studios as depicted in figure 1.1.

An added advantage is that in a digital network, signal patching can be implemented in software and controlled by the sound/audio engineer from a graphical patchbay running on a PC. This does away with the need for a physical patchbay (similar to that shown in figure 1.1) and the strain of constantly requiring that cables are physically plugged-in or pulled-out from the patchbay. Furthermore, this makes it easy to trace audio signals as they are routed across the network.

A critical aspect of digital networks is the synchronization of the networked devices. This has to do with how timing information is transferred amongst the devices on the network in order to synchronize their clocks. This is important because if the clocks are not properly synchronized distorted signals will be obtained on a receiving device.

The advent of digital audio networks has allowed for the integration of audio devices into a single network, with various networking architectures such as peer-to-peer and client-server network architectures. However, for any such digital network of audio devices, some communication and connection procedure needs to be defined. This is referred to as a protocol. An audio communication/connection protocol will define the nature and criteria for the exchange of information. Indeed every network is governed by some form of agreement, which stipulates the behavior of the members of the network. This is no different for audio networks.

1.1 Digital Audio Network Control Protocols

The streaming of audio over digital networks requires the establishment of audio networking protocols. In some networks both audio and device control instructions are transported on the same network infrastructure. There exist several networking protocols, including:

- *Common Control Interface for Networked Audio and Video Products (IEC 62379)* - uses the simple network messaging protocol (SNMP) for device monitoring, control and configuration (IEC, 2007).
- *music Local Area Network (mLAN)* - an IEEE 1394 serial bus protocol for connection management and networked device synchronization (Yamaha Corporation, 2000).
- *Open Sound Control protocol (OSC)* - a transport layer independent protocol that uses IP packets for the transmission of messages for device control and configuration (Wright, 2002).
- *Audio Video Control protocol (AV/C)* - an IEEE 1394 serial bus protocol that uses a command and response mechanism for device control and connection management (1394 Trade Association, 2001b).
- *XFN protocol* - an IP based protocol for device control, device description and connection management over a peer-to-peer network architecture (Foss, 2008).
- *Application Protocol for Controlling and Monitoring Audio Devices Via Digital Data Networks (AES 24)* - a peer-to-peer network for device control and parameter description (Audio Engineering Society, 1999).

- *Architecture for Control Networks (ACN)* - a suite of interoperable protocols that allow for device description, control, management and monitoring (ESTA, 2005a).

Each of these protocols defines its own procedure for connection management and device control of networked audio devices.

1.2 Problem Statement

Devices that communicate with a particular audio networking protocol can only communicate with other devices that also conform to that specific audio networking protocol. Even when other devices are networked using the same technology (for instance IEEE 1394 serial bus, Ethernet, ATM), there exists no direct interaction amongst networked devices that conform to distinct protocols. This isolation of networked devices is further worsened by audio networking protocols stipulating different procedures for connection management, device configuration, and device control.

This work proposes that audio network protocol interoperability can be achieved using a proxy. The goal of the proxy approach is for a single protocol to be used for connection management and audio device controls amongst networked devices that adhere to disparate protocols. Such a proxy will perform all translations of protocol specific instructions from one protocol to another.

1.3 Proxy Solution

As a product of this investigation, an AV/C and XFN device communication proxy has been created. This proxy called the *AV/C proxy*, models an AV/C device in terms of XFN thus taking advantage of the various merits of the XFN protocols, including the ease of device parameter exploration and user authentication.

The *AV/C proxy* allows for:

- AV/C device discovery
- modification of channels on isochronous stream plugs
- changing sampling frequency

- changing source of synchronization signals
- internal routing
- gain/volume control

The proxy is able to fulfill the afore-mentioned procedures by receiving XFN messages directed at an AV/C device, translating those XFN messages to AV/C commands, and then sending the appropriate (AV/C) commands to the target AV/C device. This allows the XFN proxy to remain an application level solution for network interoperability. An advantage of this approach is the flexibility provided by the proxy, such that different protocols can be integrated into it (proxy) by creating an appropriate translation for each received instruction.

1.4 Chapter Layout

Chapter 2 describes a number of audio device control and networking protocols. It describes a proprietary approach to device interoperability, by explaining the 'plural-node' *Enabler/Transporter* architecture proposed by the music Local Area Network (mLAN). There is a motivation why the mLAN enabler/transporter approach is not suitable for interoperability between AV/C and XFN devices, which are the two candidate protocols in this investigation. This is followed by a description of the proxy approach to network interoperability, by highlighting the overall scheme of interactions between networked devices that use the proxy.

Chapter 3 introduces the IEEE 1394 high speed serial bus that is the physical network upon which the Audio Video Control (AV/C) protocol is built. Various properties of IEEE 1394 are described, followed by a description of the command and response mechanism of Function Control Protocol (FCP) which is utilized by AV/C. This chapter goes on to explain the use of the AV/C protocol for device discovery, exploration (using descriptors and info blocks), control and connection management. The chapter ends by highlighting the nature of two subunits (audio and music subunits) which are of particular relevance to this investigation, since they are implemented in the AV/C device used in this investigation.

Chapter 4 describes an AV/C device that implements the audio and music subunits. This device is an enhanced IEEE 1394 breakout box that allows for the routing of analog, digital, isochronous and MIDI signals. The AV/C device exploration and control highlighted in the previous chapter is implemented in an AV/C Control Panel application that allows for internal device signal

routing, volume control, and isochronous channel modifications. This chapter gives a practical understanding of the nature of AV/C messaging, and emphasizes the complexity of device exploration using the AV/C descriptors and info block mechanism.

Chapter 5 introduces the XFN protocol as an application layer protocol. Here the Internet protocol suite, which categorizes various networking protocols, is described, with emphasis on the various protocols used by XFN. There follows a description of 'IP over 1394' that allows XFN to send IP packets across an IEEE 1394 network. The XFN protocol is then described with an in-depth explanation of the nature of XFN devices, XFN messaging and XFN packet structure. The XFN application programming interface (API) is described and this is followed by a description of XFN device discovery and parameter exploration software (created in this investigation) that utilizes the XFN API.

Chapter 6 discusses the design and implementation details pertaining to the creation of a proxy for AV/C and XFN device interaction and interoperability. It starts by explaining two approaches that could be used in the modeling of AV/C and/or XFN devices by the proxy, highlighting the implementation consequences of using either approach. This is followed by the motivations for the choice of the AV/C proxy approach for the network interoperability implementation. The modeling of an AV/C device's controls as XFN parameters, and the use of an XFN-style parameter referencing scheme for the AV/C parameters are then described. This chapter ends by describing the nature of device control on a graphical user interface, and tests that have been conducted to investigate the efficiency of the AV/C proxy (application) with regards to AV/C device discovery (in XFN), connection management, internal routing and volume controls.

Chapter 7 re-emphasizes the need for interoperability between networked professional audio devices. It highlights some of the benefits of an all-inclusive network of professional audio devices to consumers, and to the professional music industry. Also mentioned is a summary of the implementation constraints, as well as suggestions of desirable features that would allow for more efficient interactions between the audio control protocols investigated in this study (AV/C and XFN).

Chapter 2

Current Protocols and the Interoperability Problem

This research investigates the interoperability of audio control protocols for networked audio devices. It seeks to determine an efficient and practical approach that will allow devices of different protocols to communicate.

In this chapter a number of current audio device networking protocols will be described. The various protocols described here are proprietary networking protocols, and audio/video devices that conform to one protocol are unable to communicate with devices that conform to a different protocol, even though they are on the same network. The description of these protocols will give an indication of the large range of disparate protocols currently being used in the professional audio world.

Some of the professional audio networking protocols that are described in this chapter considered the possibility of interoperability between devices that conform to the protocol and devices of other protocols. One such protocol that provides for device interoperability in professional audio networks is the *mLAN* version 2 architecture. The mLAN *plural-node* architecture (mLAN version 2) is described in this chapter with the intention of giving an insight into how a particular protocol (mLAN) has approached this issue of interoperability between mLAN and non-mLAN devices.

2.1 Networked Audio/Video Device Control Protocols

This section describes various protocols used to control networked audio devices. The intention is to highlight possible network solutions for audio and video device control, monitoring and connection management, and to motivate the choice of the particular protocols chosen for this study. The protocols described here are the Open Sound Control Protocol (Wright, 2002), Architecture for Control Networks (ESTA, 2005a), Common Control Interface for Networked Audio and Video (IEC, 2007), Audio Video Control Protocol (1394 Trade Association, 2001b), music Local Area Network (Yamaha Corporation, 2000), Application Protocol for Controlling and Monitoring Audio Devices via Digital Data Networks (Audio Engineering Society, 1999) and the XFN protocol (Foss, 2008).

2.1.1 Open Sound Control Protocol (OSC)

The Open Sound Control (OSC) protocol was designed for communication amongst computers, sound synthesizers and other multimedia devices (Wright, 2002). It is a (OSI[1]/ISO[2]) transport layer independent protocol that involves the transmission of OSC messages across a UDP or TCP/IP network (Wright, 2005).

Device control or manipulation in OSC involves the transfer of *OSC packets* from an *OSC client* to an *OSC server* (Wright, 2002). The OSC client specifies an OSC *method* within the OSC server that it wishes to trigger using a 'URL-style' addressing scheme. For instance to address an OSC message to the input gain of the first multicore on an OSC device, the OSC client specifies the address as '/input/multicore/1/gain'. This URL/directory style path that is used to address a control within an OSC device is known as an *OSC address pattern*. The address pattern forms a hierarchical tree structure, with the tree leaves referred to as *OSC methods* and the branches being called *OSC containers* (Wright, 2002). The OSC containers are the higher level paths to the OSC methods of the OSC address pattern.

It is the OSC methods that trigger the appropriate procedures required to fulfill a request from an OSC client. Hence a client wishing to instruct a server to perform a particular action addresses its OSC packet to the method that will result in the desired outcome. In some cases, the client may need to instruct more than one method on a device, or it may want to set multiple values for a specified method. Multiple methods at any particular level of the OSC hierarchical

[1] Open Systems Interconnection (OSI) defines a model for communication between devices in a network.
[2] International Organization for Standardization (ISO) is a publisher of international standards (ISO, 2009).

method layout (tree) can be addressed using the asterisk (‘*’) symbol. For instance an address ‘/input/multicore/*’ is used to trigger all input multicore methods on a device. To address different occurrences of a single character at a particular level the question mark (‘?’) symbol is used within the OSC address pattern. To specify a range of values that can be passed to a method, the open and close ‘square’ brackets symbol (‘[’ ‘]’) are used (Wright, 2002).

There are two types of packets defined in OSC, namely *OSC message* and *OSC bundle* (Wright, 2002). The first byte of an OSC packet can be used to differentiate between an OSC message and an OSC bundle.

An *OSC message* consists of an *OSC address pattern*, followed by the *OSC type tag string* and then the *OSC arguments* (Wright, 2002). The OSC address pattern (mentioned earlier) is a sequence of ASCII characters preceded by a forward slash (‘/’), and it is used to specify a particular OSC method that is implemented on an OSC server. An example of an address pattern is ‘/input/multicore/1/gain’. The OSC type tag string is a comma (‘,’) followed by an *OSC symbol* that indicates the type of arguments that follow in an OSC message. For example ‘, f’ indicates that the argument which follows is a floating point number. The OSC arguments are binary representations of each argument indicated in the OSC type tag string.

An *OSC bundle* consists of the string ‘*#bundle*’ in the first byte of the packet, followed by an *OSC time tag* and a number of *OSC bundle elements*. The OSC time tag is a 64-bit fixed point number that indicates when a specified method should be invoked (Wright, 2002). For instance a time tag value of all zeros and the least significant bit set as ‘1’ means that the ‘bundled’ method(s) should be triggered immediately. An OSC bundle element is made up of two parts. The first part is a size field that specifies the number of bytes that make up the second part. The second part is the content field which could be either an OSC message or another OSC bundle nested within. The OSC bundle is used to ‘package’ a number of OSC messages each triggering an OSC method, into one atomic process.

2.1.2 Architecture for Control Networks (ACN)

The Architecture for Control Networks (ACN) includes a number of different but interoperable protocols that allow for device description, control, management and monitoring. This protocol was initially developed as a protocol for lighting equipment to replace the DMX-512[3], but has

[3]DMX-512 is a control protocol for interconnection amongst lighting equipments of different manufacturers and was developed by the ESTA (ESTA, 1996).

now been utilized in audio and video device control networks (ESTA, 2005a). When ACN was being developed by the Entertainment Services and Technology Association (ESTA) , one of the motivations was to create a protocol that allowed for network interoperability. As a result, the development of the ACN suite of protocols followed a modular design process. This allows the protocols to be independent of each other, and allows them to be combined in several different ways (ESTA, 2005a).

On an ACN network, a *component* is any device or process that is capable of transmitting or receiving ACN data (ESTA, 2005a). Each ACN component is uniquely identified by a component identifier (CID). A *device descriptor language* (DDL) is used in ACN to model the controls within a device in a descriptive manner (ESTA, 2005b). This description is an XML schema that describes the device's parameters. The DDL provides an interface that a controller can use to determine the nature of an ACN device. The DDL is comprised of (ESTA, 2005b):

- *device model* - a text file that describes the fundamental control elements within a device. The device model is a high-level description of a device.
- *XML syntax* - an XML file that describes the entire structure of a device. This XML file is very descriptive and lays out the internal structure of a device.
- *protocol interface* - provides a means by which protocols can use the DDL.

The layout of DDL of a device is such that a modular hierarchy is formed (ESTA, 2005b). The modules that are used to describe a device are independent, hence allowing for easy expansion and modifications of individual modules, as well as module reuse.

The interaction with devices on an ACN network is achieved using the *device management protocol* (DMP). The DMP provides for device control, monitoring and management by sending commands according to the *protocol data unit* (PDU) format (ESTA, 2005c). The PDU format as defined by the Architecture for Control Networks specification (2005), is a structural layout of the nature of a message that is transferred between components in an ACN network (ESTA, 2005a). It might be necessary to 'bundle' a number of PDUs into a single group known as the *protocol data unit block* (PDU block). It is the PDU or PDU block that is transmitted in an ACN message over a transport layer protocol. The ACN protocol is independent of the transport layer protocol used for the transfer of ACN data.

In an ACN message, PDUs are packaged in a structured hierarchical manner and are encapsulated within an ACN packet's *preamble* and *postamble*. The hierarchy can be as a result of nested

PDUs with the outermost protocol at level 0 being referred to as the *root protocol* (ESTA, 2005a). The immediate inner PDU is referred to as the level 1 protocol, within this level might exist a level 2 protocol, and so on. It is the root protocol that directly interfaces with the network transport protocol.

Before DMP messages are transmitted, a connection is established between the devices on an ACN network (ESTA, 2005c). This connection allows an ACN controller component to communicate with other components. The DMP relies on the transport layer for the addressing of devices on an ACN network.

2.1.3 Common Control Interface for Networked Audio and Video Products (IEC 62379)

The Common Control Interface for Networked Audio and Video Products (IEC 62379) protocol defines a means for media device control and device parameter enquiry using the Simple Network Messaging Protocol (SNMP). It defines an audio/video device as a *unit* that is made up of functional entities called *blocks* (IEC, 2007). The blocks perform specific signal processing within the unit and are connected to each other to form a signal processing chain. The blocks within a unit have input and output plugs, except for special blocks known as *ports*. Ports are blocks that either input signals to a unit or output signals from the unit. They (ports) are the points of entry for signals into a unit for processing by the unit's blocks, or points of exit from a unit. Any device on the IEC 62379 network is modeled in terms of its blocks and the connections between the input and output plugs of its blocks.

A unit is uniquely identified by an extended unique identifier (EUI-64) which includes the 48-bit media access control (MAC) address of the device's interface. The various controls (parameters) within a unit are referred to as *managed objects*. Each managed object is uniquely identified with an object identifier (OID)(IEC, 2007). The OID is comprised of alphanumeric numbers separated by dots that uniquely addresses a parameter within a device. The use of the OID for parameter addressing is designed to form a hierarchical scheme, with the higher level addresses to the left of the address. For example, an object with OID 'a.b.c', implies that 'a' contains 'b' which in turn contains the object at 'c'.

The IEC 62379 (2007) document defines a *Management Information Base* (MIB) which is a structured hierarchy of parameters (managed objects). Each managed object is identified with its OID in the MIB. The OID describes a particular managed object within a block. In this sense, a

block can be seen as a container of objects.

A *management terminal* is a controller that can discover IEC 62379 devices on a network and manipulate the connections between a unit's blocks. The device discovery process defined by IEC 62379 requires that a management terminal (IEC, 2007):

- determines and creates a list of the units on the network. Each unit is uniquely identified by its EUI-64 identifier.
- determines and creates a list of blocks for each unit.
- determine the connections between the blocks within a unit in order to better understand its processing chain.

The IEC 62379 allows for flexibility of implementation that permits manufacturers to define new types of blocks if required. However the IEC 62379 defines strict requirements for the creation of such blocks. One such requirement is that each newly created block is associated with an MIB table.

In some cases a management terminal seeking control of a management object might require authorization, which are referred to as 'privilege levels'. There are four privilege levels, namely *listener*, *operator*, *supervisor* and *maintenance*. The listener has the least privilege level on unit controls. This is followed by the operator, and then the supervisor. The maintenance level has the highest privilege level for the control of objects on a unit (device).

2.1.4 Audio Video Control Protocol (AV/C)

The Audio Video Control (AV/C) protocol is a command and response protocol that allows for device control on an IEEE 1394 high-speed serial bus. In AV/C, devices are addressed by their IEEE 1394 serial bus node identifiers. An AV/C command is sent from a *controller* node to a *target* node. In response to the command, the target node sends an AV/C response to the controller node.

All AV/C messages are transmitted as IEEE 1394 asynchronous transactions which entail the transfer of asynchronous packets. The nature of asynchronous transactions is that every transmitted packet is acknowledged by the receiver. In an asynchronous packet, the transmitting device specifies its node identifier, and this enables the receiving device to address its response to it.

An AV/C device is composed of logical entities called *units*. These units are subdivided functionally into *subunits*. A subunit might require that a particular processing is performed on its behalf by its *function blocks* (1394 Trade Association, 2001b). There exist different types of units, subunits and function blocks. The 1394 Trade Association (1394 Trade Association, 2009) maintains all standardized specifications that relate to the large range of units and subunits defined in the AV/C general specification (2001) (1394 Trade Association, 2001b).

Within an AV/C command the *unit/subunit/function block* of interest is specified by its type and its identifier. The response returns the same value for the unit, subunit or function block as specified in the command it is responding to. Each AV/C command contains an *opcode* field that specifies the particular action to be performed on the target node. Depending on the *opcode*, a number of *operands* are also specified as fields in an AV/C command, these *operands* are arguments necessary for the execution of the AV/C command. An AV/C response from the target node returns the same value for the *opcode* as was received specified in the command, however, the *operands* may differ.

In AV/C each device presents a layout of its internal structure with an interface known as its *descriptor* and *information block* (*info block*) (1394 Trade Association, 2001c). The descriptor forms a hierarchical layout of the internals of the device with *root descriptor* at the highest level. The root descriptors consist of several *entry descriptors* and *list descriptors* that hold specific information in their associated info blocks (1394 Trade Association, 2001c). The actual storage of data internally within the device is independent of the structure of the descriptors and info blocks. The descriptors form standardized layouts that make it possible for a controller to traverse the properties of a device. However, such a controller must have knowledge of how to parse the descriptors of the specific device. This knowledge is necessary because AV/C allows a manufacturer to define an infinite number of descriptors, using a standard layout.

In AV/C, certain descriptors are defined as ‘recommended’ and others ‘optional’. Most manufacturers implement the ‘recommended’ protocols and not all of the ‘optional’ descriptors, defined in AV/C. This flexibility in the laying out of a device’s descriptors, with its associated info blocks, makes it practically challenging to parse any AV/C device descriptors and info blocks. The AV/C protocol is further discussed in chapter 3.

2.1.5 Music Local Area Network (mLAN)

The music Local Area Network (mLAN) is a proprietary IEEE 1394 serial bus connection management and audio device synchronization protocol that was developed by Yamaha Corporation.

It allows for the transmission of audio and MIDI data amongst networked devices (Fujimori and Foss, 2003). In the first implementation of mLAN the AV/C descriptor and info block mechanism was used for device description, and AV/C vendor specific commands were used to instruct networked mLAN devices. This initial implementation is referred to as the *mLAN version 1* protocol.

In the mLAN version 1 connection management implementation, the abstraction of plugs was located on the networked devices. These plug abstractions were end-points of IEEE 1394 isochronous stream sequences (or MIDI subsequences) and word clock synchronization signals. The isochronous stream sequence concept is explained in chapter 3 section 3.1.7. The mLAN version 1 architecture required that an mLAN device possessed non-volatile memory for the storage of device settings, which was used to set the device to its previous state after every short bus reset or power down. This resulted in an increase in the price of an mLAN (transporter) device. Another drawback with this approach was that changes to the audio/music extraction or transmission requirements depended on a *firmware*[4] upgrade of each transporter device (Fujimori and Foss, 2003).

The mLAN project developed into an mLAN version 2 protocol that sought to overcome the limitations of mLAN version 1. The mLAN version 2 architecture, termed the *Enabler/Transporter* architecture, also referred to as the *plural-node architecture,* relocated the plug abstractions to a digital workstation (PC). In the *Enabler/Transporter* architecture, an *enabler* located on a digital workstation (PC or Macintosh) is in charge of device setup, connection, disconnection and configuration of the networked devices. A *transporter*, implemented on each networked device allowed for the extraction and transmission of isochronous stream data in accordance with the IEC 61883-6 (2005) specification (IEC, 2005). Each transporter can be controlled by only one enabler on a network, although an enabler will typically control more than one transporter device (Okai-Tettey, 2005). As a result, this meant that the enabler which was running on a digital workstation was required to setup the connections between the networked audio devices that implement the transporter (Fujimori and Foss, 2003). However once connections have been made and word clock 'master-slave' relationship established, the networked devices are capable of continuing isochronous transmission and reception of audio and MIDI data without further assistance from the enabler. The mLAN version 2 architecture allows for direct asynchronous reads and writes to address registers within a transporter device. The mLAN enabler implements mLAN plugs that abstract all possible isochronous stream sequences and word clock signals im-

[4]Firmware is software that resides permanently in small size memory within a device, and that controls the hardware component(s) of the device.

plemented in the transporter node. These mLAN plugs allow for connections and disconnections via API methods to the transporter objects within the mLAN enabler. Section 2.3.1.1 on page 21 discusses how connection management is achieved in mLAN version 2.

An mLAN Connection Management Server (mCMS) was developed above the mLAN version 2 architecture. TCP/IP communication is possible between mLAN client devices and mCMS (server) (Chigwamba and Foss, 2007). The client devices communicate with the mCMS using XML.

2.1.6 Application Protocol for Controlling and Monitoring Audio Devices Via Digital Data Networks (AES 24)

The AES 24 protocol is a peer-to-peer application protocol that allows any device on the network to request status information and execute control messages to parameters within networked AES 24 audio devices (Audio Engineering Society, 1999). Device state inquiry and parameter control is achieved by sending commands on the network. The AES 24 protocol models a device in terms of functional elements known as *objects*. The AES 24 commands are exchanged between these objects.

A three level hierarchy of *networks*, *devices* and *objects* is used to structurally describe objects within a device on an AES 24 network. This hierarchy forms an interface that represents in a structured manner the internal layout of functional objects within an audio device. It does not actually represent how those objects and their associated parameters are stored internally within the audio device.

Objects are grouped into classes in the AES 24 protocol. Objects within a class have similar *methods*, *properties* and *events* (Audio Engineering Society, 1999). A method is triggered by an AES 24 command sent from one object to another in order to fulfill a task. Certain conditions known as events can cause an AES 24 device to perform specific functions. These conditions include a change in the state of an object.

Every object within a device is uniquely identified with an identifier and is associated with a parameter that defines its state. This state parameter is a property of the object. A property is a unique attribute of an object that can be monitored and controlled by other objects on the network. When sending a message to an object's property, a *property method* is used to indicate what operation should be performed on the property. The AES 24 protocol defines a number of property methods that can be used to instruct or determine the state of an object's property.

In the AES 24 protocol, devices are modeled as comprising of (Audio Engineering Society, 1999):

- *Management object* - handles device initialization procedures, object addressing, security and signal flow management issues.
- *Intermediate object* - models an internal processing service required for the internal functionality of the device's user interface objects and functional objects.
- *User interface object* - models the user interface functional requirement of a device.
- *Functional object* - models an application function within a device.

Messages are sent from one object to another to cause the target object to execute a particular method. An AES 24 message may request a response or not. In the case of a message that requires a response, the target object will send a response message indicating whether the instruction was successfully fulfilled, or whether it failed. It may return the state of the property that is being accessed by the requesting message.

2.1.7 XFN Protocol

The XFN protocol is a peer-to-peer command and control protocol for networked audio devices (Foss, 2008). Devices on an XFN network are capable of sending and processing request messages.

An XFN device is described in terms of the functional elements it is composed of. These are the *parameters* that can be created, modified or queried by received XFN messages. A 7-level hierarchical description is used to specify a device's parameter. This 7-level hierarchy forms a parameter tree with each of the tree levels viewed as a node on the tree. Each XFN device parameter is associated with a unique identifier known as a *parameter index*.

There are two possible ways of addressing a device's parameter in XFN. In the first instance, a requesting node sends an XFN message to a device's parameter by specifying the 7-level hierarchical description that indicates the parameter of interest. This is called *full data block* parameter addressing. In the second instance, the requesting node sends a message addressed to the parameter's index. This is referred to as *indexed parameter* addressing.

The XFN protocol is a (OSI/ISO) transport-layer independent messaging protocol. XFN messages are transmitted within UDP/IP packets. The device discovery process in XFN entails the requesting node transmitting a broadcast IP packet which is addressed to the IP address parameter. Every XFN device implements an XFN *internal configuration node* which contains among other parameters, the IP address parameter (Klinkradt, 2007). Hence, all XFN devices on the network will respond to the device discovery broadcast message, each with its IP address.

This protocol provides a *wildcard* mechanism that permits multiple parameter addressing. For instance a wildcard at level 3 of an XFN full data block message is addressed to all nodes at that level.

The XFN protocol is capable of ensuring data security and user authentication. It is possible to define user access levels on a parameter using the authentication mechanism. Hence, the permissions for user access to a parameter can be varied.

2.2 Summary of Networked Audio Device Control Protocols

Common to most of the audio device control protocols described in the previous section, is the use of a hierarchical structure to address and describe parameters with a device. Notably, the:

- *OSC* protocol - uses a URL-style *address pattern* that forms a hierarchy, to describe various parameter *methods* within a device.
- *ACN* protocol - uses the *DDL* which forms a hierarchy of *modules*, to model an *ACN* device's controls.
- *IEC 62379* protocol - uses a hierarchy of *blocks* and *objects*, which are addressable using their *OID*s, to describe the parameters within a *IEC 62379* device *(unit)*.
- *AV/C* protocol - uses *(unit/subunit) descriptors* and *info blocks*, which form a hierarchy from *root* descriptors to *entry* and *list* descriptors with their associated info blocks, to describe a device's parameters.
- *AES24* protocol - uses a 3-level parameter hierarchy of *networks*, *devices* and *objects*, to describe parameters (*objects*) within a device.
- *XFN* protocol - uses a 7-level parameter hierarchy to parameters within an XFN device.

However, *mLAN* (protocol) is the only mentioned protocol that does not model networked devices, since it is was not intended for device control. By abstracting isochronous stream plugs and word clock plugs, *mLAN* is able to provide connection management and device synchronization in audio networks.

The audio device control protocols described in the previous section are proprietary solutions to controlling networked digital audio devices. Each protocol defines a particular device description, control and connection management procedure. This presents a problem, in that devices cannot co-exist in a network with a common controller and contribute to the audio system.

The more specific problem of protocol interoperability on an IEEE 1394 bus was considered in this thesis. Currently there are a number of IEEE 1394 devices that implement various protocols, including;

- XFN,
- mLAN, and
- AV/C.

Figure 2.1 shows an IEEE 1394 network with devices that implement the above mentioned protocols.

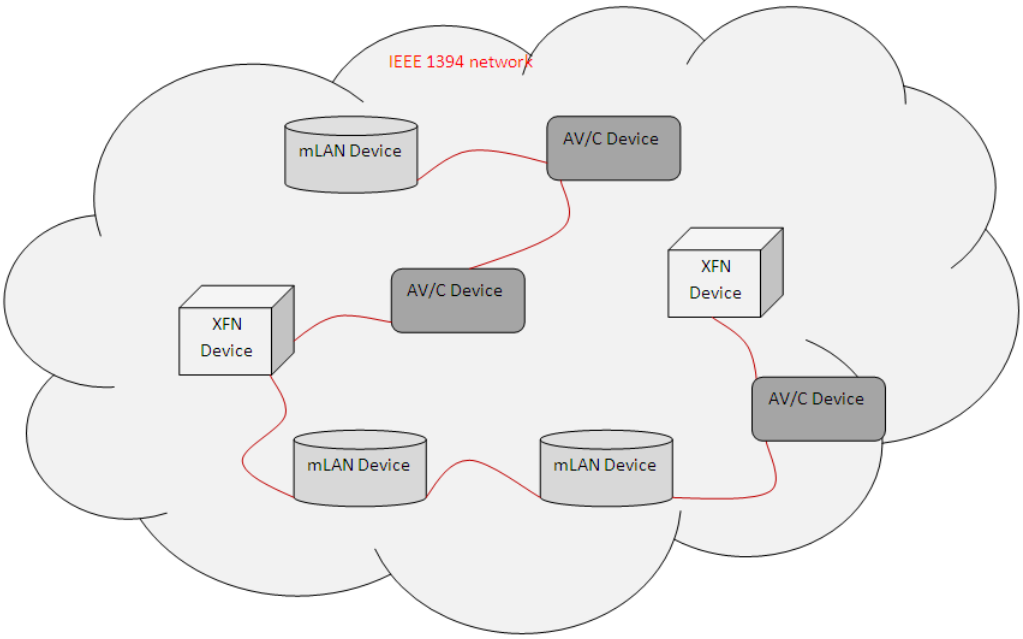


Figure 2.1: IEEE 1394 network with devices that implement different protocols

Although the devices shown in figure 2.1 implement protocols that are based on the same transport protocol (IEEE 1394 serial bus), they are unaware of each other. For example, the AV/C devices are unable to discover either of the XFN devices or the mLAN devices. Furthermore, it is not possible to have a common controller for all these different devices. The next section describes the approaches towards device interoperability on an IEEE 1394 network that was considered in this investigation.

2.3 Possible Solutions to Networked Audio Device Interoperability on the IEEE 1394 Bus

In the course of this investigation, two approaches were considered as possible solutions for interoperability between networked audio devices that implement different audio network protocols. The first was the *mLAN approach to interoperability* and the second was a *proxy approach to interoperability*. These two approaches are described below.

2.3.1 mLAN Approach to Interoperability

Before we can understand the mLAN approach to interoperability, we need to have some background of the mLAN version 2 *plural-node* architecture.

In 2002 Yamaha Co-operation developed version 2 of its IEEE 1394 serial bus audio device networking protocol (*mLAN*) (Fujimori and Foss, 2003). mLAN version 2 considered the possibility of integrating other non-mLAN professional IEEE 1394 devices onto the mLAN network. This allows for interoperability between mLAN and non-mLAN devices.

2.3.1.1 mLAN version 2 Plural-Node Architecture

mLAN version 2 provides connection management and synchronization for the transmission of audio and MIDI data amongst devices on an IEEE 1394 network. The plural-node architecture of mLAN version 2 provides a high level abstraction of the isochronous stream plugs within mLAN devices. It allows for connection management, including connecting and disconnecting mLAN plugs, to be implemented by an *enabler*. For every connection or disconnection request by a user, the appropriate low-level device register instructions are directed at a *transporter*, by

issuing asynchronous read and write transactions. These low-level instructions are intended to configure the transporter device.

The concepts of an *Enabler* and *Transporter* are central to the plural-node architecture. Indeed, the plural-node architecture is also known as the *Enabler/Transporter* architecture. The discussion below described these two concepts in more detail.

mLAN Enabler

The mLAN enabler is a program that resides on a workstation, such as a *Windows PC* or *Mac*. It provides a means by which connections specified by a user, for instance through a graphical user interface, can be implemented. Structurally the mLAN version 2 enabler consists of the following layers (Fujimori and Foss, 2003):

- *mLAN Plug Abstraction Layer* - forms the top layer of the enabler architecture. It interfaces with whatever application requires the services of the enabler and presents the mLAN plugs of networked mLAN devices to the application. All of the device's input and output plugs are abstracted as mLAN plugs that the application can interact with using an application programming interface (API) that has been defined for this layer. The mLAN plugs are virtual end-points of IEEE 1394 isochronous audio sequences or MIDI subsequences.

- *A/M Manager Layer* - is located just below the mLAN plug abstraction layer and interacts with it in order to perform the instructions specified by a user. The A/M layer concerns itself with interactions between the enabler and its associated transporters that pertain to audio and music data parameter manipulations. It is at this layer that the enabler creates a transporter object for each transporter under its control. The transporter objects maintain the state of the transporter devices they represent. The mLAN plug abstraction layer interacts with this layer (A/M Manager layer) via an API that has been defined in mLAN.

- *Hardware Abstraction Layer (HAL)* - forms the bottom of the three layers within the enabler. The HAL layer interacts directly with the A/M manager layer above it and the transporter control interface on the networked transporters (devices). The HAL layer abstracts details of how manufactures have implemented procedures for audio and music data transmission and extraction, within a plug-in. This layer comprises a number of plug-ins that are device specific and whose implementation is dependent on the manufacturer's preferences. A manufacturer wishing to incorporate their device into the mLAN network is required to provide a plug-in for the enabler's HAL layer.

It is possible that a network is comprised of more than one enabler, with each enabler controlling more than one transporter (mLAN device). However, each transporter can only be controlled by a single enabler and it (transporter) is aware of its enabler.

mLAN Transporter

The transporter incorporates an *mLAN Node Controller* and the associated firmware that allows for the extraction and transmission of audio and music data on an mLAN network (Okai-Tettey, 2005). An mLAN transporter is comprised of the following layers (Yamaha Corporation, 2002):

- *IEEE 1394 Layer* - implements the IEEE 1394 serial bus protocol requirements. This includes the implementation of the IEEE 1394 physical layer, link layer and transport layer requirements. It also implements the *mLAN unit directory* as defined in the 'mLAN-2.0 Configuration ROM specification' (2003). This mLAN configuration ROM is used for the identification of a device as an mLAN transporter, and for other software diagnostics that are required for device setup.
- *Transport Control Interface Layer* - maps the control and status registers (CSR) needed for audio and music data processing to the address space within an mLAN transporter. This exposes register space on the transporter to direct IEEE 1394 asynchronous transactions. In the version 1 of mLAN this functionality was not provided by the transporter node, hence all interactions were by AV/C vendor dependent commands.
- *A/M Protocol Layer* - a hardware or software implementation that allows a transporter to extract and package (for transmission) audio and MIDI data in accordance with the 'Audio and Music Data Transmission Protocol' (2005) specification. Hardware application-specific integrated circuit (ASIC) chips such as Yamaha's NC1, PH1 and PH2 chips provide this functionality. BridgeCo's DM1000 chip allows this functionality to be implemented in software (Okai-Tettey, 2005).

One challenge of the above basic requirements of the mLAN transporter is that each transporter will require that an enabler configures its A/M parameters after every power down. To overcome this limitation, mLAN transporters can implement a *connectionless isochronous transmission* (CIT) manager. A transporter that implements the CIT manager saves its A/M layer parameter settings (register values), after each setup by the enabler, to a section of its memory referred to as *boot parameter memory* (Okai-Tettey, 2005). Upon power up, it is the responsibility of the CIT

manager to set the values of the transporter's A/M layer parameters to those stored in the boot parameter memory.

With this understanding of what is required of the enabler and transporter on an mLAN network, the next section describes how non-mLAN devices can communicate with mLAN devices using the mLAN enabler/transporter (plural-node) architecture.

2.3.1.2 Device Interoperability using the mLAN Architecture

The following discussion presents an approach to achieving network interoperability between mLAN devices and AV/C devices, using mLAN's enabler/transporter architecture. Note that these are both based on the IEEE 1394 transport protocol.

The mLAN enabler/transporter architecture was designed with the possibility of permitting communication between mLAN Transporters and non-mLAN devices via the mLAN enabler. Figure 2.2 shows an mLAN enabler (running on a PC) that enables connections with three transporters ('Device 1', 'Device 2', and 'Device 3'). The enabler is connected to two AV/C device ('AV/C Device A' and 'AV/C Device B'), and communicates with them by sending AV/C commands and responses.

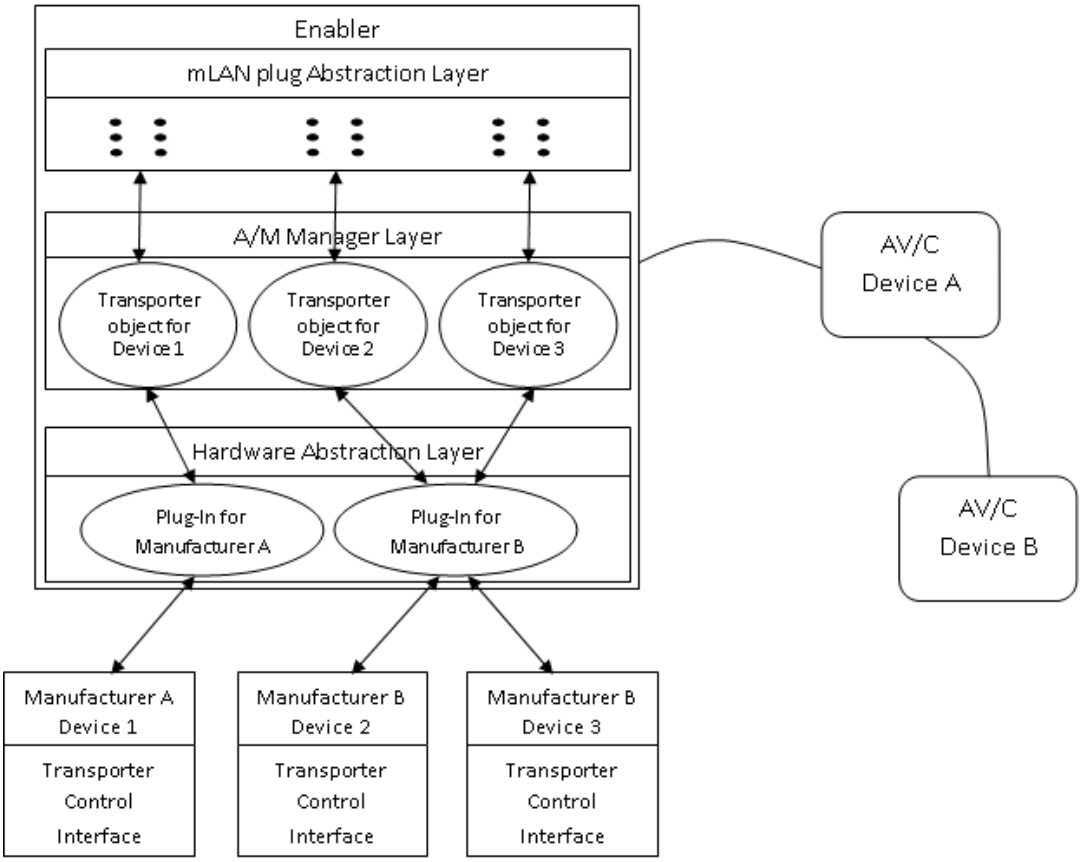


Figure 2.2: mLAN Enabler/Transporter architecture for interoperability

The role of the enabler is to enable communication between transporters (devices). By implication, a transporter relies on the enabler to set up its parameters for the transmission and reception of data. The primary duty of the transporter is to encapsulate data for transmission, and extract data from received packets.

Each device manufacturer implements a *Hardware Abstraction Layer* (HAL) plug-in that is utilized by the enabler to communicate with transporter devices. The HAL plug-in sends appropriate manufacturer dependent instructions necessary for the extraction and encapsulation of data, to a transporter device. As shown in figure 2.2, two plug-ins are available in the HAL layer of the enabler. 'Manufacturer B' developed the hardware on 'Device 2' and 'Device 3', hence the procedures and associated instructions necessary for configuring the transporters ('Device 2' and 'Device 3') are the same, and are encapsulated within the 'Plug-In for Manufacturer B'.

For each transporter, the enabler creates a transporter object in its *A/M Manager Layer*. In figure 2.2 these can be seen as transporter objects for 'Device 1', 'Device 2', and 'Device 3'. Each

transporter object models the parameters within the corresponding transporter, and communicates with the transporter via an API provided by the hardware abstraction layer. These API methods are implemented within the HAL plug-ins in a manufacturer dependent manner. Any modification of the state of parameters within the created transporter objects results in a corresponding parameter adjustment on the actual transporter device.

The transporter objects present a number of plug abstractions via an API interface, to the *mLAN plug Abstraction Layer* as shown in figure 2.2. These plug abstractions represent end-points of audio sequences, MIDI subsequences, or word clock signals. Connections with any of these plugs trigger API methods that result in a change in the state of parameter(s) within the corresponding transporter object.

To communicate with the AV/C devices, the enabler (in figure 2.2) runs on a workstation that implements the FCP_COMMAND and FCP_RESPONSE registers (these registers are described in chapter 3). This makes it possible for the enabler to receive AV/C commands from an AV/C device, and send AV/C responses to an AV/C device.

The manipulation of isochronous data within AV/C devices is handled by *music plugs* that reside within the music subunit of an AV/C device. The music subunit is discussed in chapter 3. The enabler in figure 2.2 can abstract its transporter plugs as AV/C music plugs, since it is on an AV/C network, and can present these music plugs to a control application. The control application is then capable of making connections between the music subunit plugs of the AV/C devices, and the music subunit plugs of the enabler's transporters. This allows for communication between the AV/C devices and the transporters. Each AV/C command to music subunit plugs within the enabler is eventually implemented by a HAL plug-in as low-level asynchronous writes to registers within the appropriate transporter.

This scheme allows the enabler to abstract plugs on a transporter in such a manner that the plugs conform to the AV/C network protocol. For example in the illustration shown in figure 2.2, the enabler abstracts plugs on its transporters as AV/C music plugs in order to conform with the AV/C network. As a result devices that conform to other protocols, in this case protocols that do not manipulate data sequences using music subunit plugs will be unable to make connections via the enabler.

This approach allows a tranporter device with a HAL plug-in to communicate with a device that has a high level protocol implementation. It does not allow connection between two devices with different high level protocols.

As a result of the above restriction an alternative approach which entails the use of a proxy was

explored. This approach involves the use of a proxy that translates instructions from one protocol to another in an attempt at providing communication between devices that conform to disparate protocols. The proxy approach is described in the next section.

One way of enabling mLAN compliance in a non-mLAN device is to rewrite the entire firmware on the non-mLAN device. This firmware upgrade will enable transporter capability on the non-mLAN device. Such firmware has been created for some devices, for instance TerraTec's Phase 24 FW (Dolphin Music, 2005).

2.3.2 Proxy Approach to Interoperability on the IEEE 1394 Bus

The proxy approach allows devices that conform to different protocols to communicate and to be configured by utilizing a common controller. This approach involves the use of a proxy that:

- abstracts a physical device with a high level protocol implementation, for instance protocol A,
- receives messages of a different protocol on behalf of the abstracted device,
- translates the received messages into instructions that the physical device will understand, and
- relays the translated messages to the physical device according to protocol A

The abstraction of devices as objects within the proxy ensures that any change in the state of a proxy's abstract device parameter, will cause a corresponding change in the state of a parameter within a physical device.

The concept of low-level register manipulation for device control, as prescribed by the mLAN approach, entails a detailed understanding of register mappings within the device and the exposure of a device's address space to the enabler.

The proxy depicted in figure 2.3 enables communication between two devices, each implementing a different high level protocol.

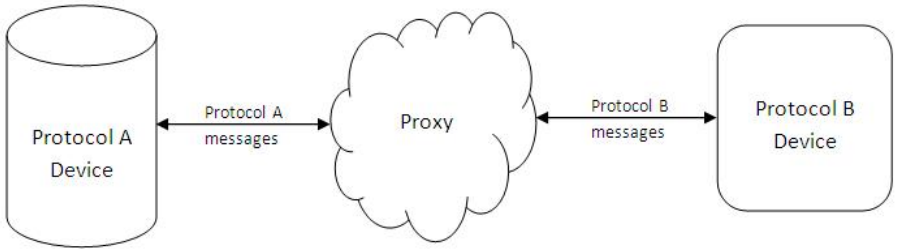


Figure 2.3: Device interactions via a proxy

Protocol A and B devices represent devices of different protocols. 'Protocol A Device' communicates by sending 'Protocol A messages', similarly 'Protocol B Device' can only understand 'Protocol B messages'. This highlights the problem currently experienced in professional audio contexts, where networked devices that conform to different protocols cannot communicate with each other or be configured by a common controller.

With the proxy implemented as a translator between both devices, all communications between 'Protocol A Device' and 'Protocol B Device' are performed via the proxy. The proxy fulfills received messages by translating them from one protocol to another. This approach allows for interoperability between devices of different protocols by ensuring high-level communication (protocol messaging) between networked devices.

In this research, a decision was taken to attempt an interoperability solution between two protocols, namely: AV/C and XFN protocols by using the above proxy approach. The choice of AV/C was largely due to the widespread use of AV/C devices for music production. The XFN protocol was chosen as a contemporary network solution for audio devices, since it provides an easy device description and parameter control mechanism, as well as device and parameter control security. The use of both of these protocols for this investigation was also dependent on the availability and accessibility of AV/C and XFN devices to the researcher. AV/C devices are commercially available, and XFN devices are accessible to the Audio Engineering Research Group (at Rhodes University) of which the researcher is a member.

2.4 Summary

In this chapter several audio and video networking protocols were described. These include IP-based networking protocols such as the Open Sound Control (OSC) protocol, and an IEEE 1394 serial bus networking protocol, the music Local Area Network (mLAN) protocol. The overall

objective of these audio networking protocols is to allow for device control, monitoring, and connection management.

The Enabler/Transporter architecture described in this chapter considered the possibility of interoperability between devices. In this architecture, the enabler abstracts its transporters' plugs as end-point of data sequences in a network dependent manner, and presents those plug abstractions to a control application that wishes to utilize the enabler. The plug abstractions can conform to any protocol. The actual implementation of the connection management procedures are handled by low-level register asynchronous writes that are performed in a manner that is determined by the device's manufacturer.

A different approach to network interoperability was proposed in this chapter. This approach involved the implementation of a proxy that will mediate between devices that conform to different protocols. This research investigates how feasible device control and connection management can be achieved using a proxy between two audio/video network protocols. The two protocols of interest are the AV/C protocol and XFN protocol.

AV/C is a protocol that involves command and response interactions between networked IEEE 1394 devices. The XFN protocol is an IP-based transport layer independent protocol that allows for device control and parameter monitoring of networked audio and video devices.

To achieve device interoperability between AV/C and XFN devices, knowledge of the nature of both protocols is required. Chapter 3 describes the IEEE 1394 high speed serial bus upon which the AV/C protocol is based. This is followed by a description of the AV/C protocol and how it can be used to establish connections amongst networked audio devices. Chapter 5 describes the XFN protocol.

Chapter 3

IEEE 1394 and AV/C

Given that this thesis will focus on two protocols, AV/C and XFN, both currently implemented above IEEE 1394, it is appropriate to provide information regarding IEEE 1394 and the two protocols. This chapter will focus on IEEE 1394 and AV/C.

This chapter introduces a high speed serial bus initially defined in the IEEE 1394-1995 standard (IEEE, 1995). Further refinements to the IEEE 1394-1995 standard are the IEEE 1394a - 2000 standard (IEEE, 2000), the IEEE 1394b-2002 standard (IEEE, 2002) and the IEEE 1394c-2006 standard (IEEE, 2006). The IEEE 1394 serial bus is capable of transmitting data at various speeds (100Mb/s, 200Mb/s, 400Mb/s, and 800Mb/s), it also allows for plug and play, as well as peer-to-peer data transfers. Other standard protocols, such as the *Function Control Protocol* (FCP) defined in the *IEC 61883-1 specification* (IEC, 2008), the *Serial Bus Protocol* (SBP-2) defined by the American National Standard Institute (ANSI) (ANSI, 1998), and the *Internet Protocol*(IP) defined in *RFC 791* document (1981) have been implemented above the IEEE 1394 serial bus (Teener, 2006).

The IEEE 1394 serial bus permits isochronous and asynchronous transactions that each allow for different methods of data transmission. Asynchronous transactions on this serial bus ensure quality of service by guaranteeing packet delivery. In an audio/video network asynchronous transactions are typically used to transmit control messages. Isochronous transactions ensure that the rate at which data is transmitted on the serial bus is constant. Hence, isochronous transactions are used for the transmission of data such as audio and video signals.

The *Function Control Protocol* (FCP) was defined by the International Electrotechnical Commission (IEC) in a specification for consumer digital audio and video equipment (1394 Trade

Association, 2001b). FCP establishes a mechanism that utilizes the IEEE 1394 serial bus asynchronous transaction capability to access memory locations (known as FCP registers) within audio/video devices. A protocol for audio/video device control known as the *Audio Video Control* (AV/C) protocol utilizes the FCP mechanism. The AV/C protocol defines various command sets that at present have over fifty specifications maintained by the 1394 Trade Association (1394 Trade Association, 2009). In an AV/C network, AV/C commands are transferred within FCP frames in IEEE 1394 asynchronous packets from a controller node to a target node.

This chapter explains some of the IEEE 1394 serial bus concepts and capabilities. It also explains some of the IEC 61883-1 specification features, with a focus on FCP. Further on in this chapter the AV/C protocol is discussed in terms of how it utilizes the FCP command and response mechanism. The discussions of AV/C are directed towards explaining the following:

- How to identify an AV/C device.
- What functional entities make up an AV/C device.
- How an application explores the capabilities and features of an AV/C device.

This chapter ends with an insight into the nature of two AV/C subunits, namely audio subunit and music subunit, which are of relevance to this research.

3.1 High Speed serial bus - Firewire

The IEEE 1394 serial bus, referred to hereafter as *firewire* [1], is a high speed serial bus with data transmission speeds of 100Mb/s, 200Mb/s, 400Mb/s, 800Mb/s, and even as high as 3.2 gigabits per second (1394 Trade Association, 2007). However on a firewire network, the transmission speed used by the bus is that of the slowest active device (Pohlmann, 2005). This implies that if a device with transmission speed of 200Mb/s is on a firewire bus with two other devices of 800Mb/s, the data transmission speed on the bus will be 200Mb/s.

Firewire devices are daisy-chained to form a network. This is an added advantage since it reduces the number of cables that is required to network multiple devices. The maximum distance between firewire devices on a bus is 4.5m (IEEE, 2000), with improvements to the IEEE

[1] The term *'firewire'* was first used to describe this serial bus by Apple Computers Inc. Sony Corporation's interpretation of the IEEE 1394 standard is referred to as *'i.LINK'*

1394 specification (IEEE 1394c -2006) allowing for transmission between devices at distances of about 100m apart using CAT5e cables (Teener, 2006). 1023 buses can be connected in a bridged firewire network, with each bus comprising a maximum of 63 nodes (Anderson, 1999). However, a maximum distance between transmitting nodes (*hop count*) of 16 is required for transactions on the firewire bus (Foss, 2007). Hop count refers to the number of times data is relayed on a network from a transmitting device to a destination device. The concept of hops can be explained with figure 3.1 which shows five networked devices labeled A, B, C, D, and E.

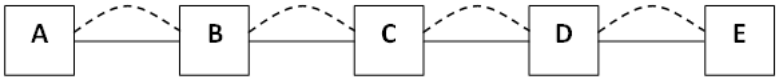


Figure 3.1: Concept of hop count in a digital audio network

In figure 3.1 hops are depicted with dotted lines, and connections are shown as sold lines. The number of hops (hop count) between nodes A and E is 4. The hop count between A and D is 3, the hop count between A and C is 2, and so on.

On a firewire network, devices can transfer data without needing a centralized host controller. When a device is added to the network, a bus reset is triggered which results in the device obtaining a unique identifier. It is this identifier that is used for addressing devices on a bus. The transfer of data on a bus is peer-to-peer in nature, and does not necessarily require a PC to set up.

Firewire devices can be viewed as *modules* that comprise one or more *nodes*. A node is a logical entity within a firewire device. A firewire node may have one or more ports that are physical plugs to which a firewire cable can be connected. Data that is received on any one port is transmitted by the node to all its other plugs, thereby allowing for a daisy-chained network of devices (Anderson, 1999). Each node on a firewire network is uniquely identified by an addressing scheme described in the next section.

3.1.1 Addressing scheme

The IEEE 1394 standard builds on the ISO/IEC 13213 specification (*Information technology - Microprocessor systems - Control and Status Registers (CSR) Architecture for microcomputer buses*) commonly referred to as the CSR architecture (IEC, 1994). The CSR architecture, as implemented in the IEEE 1394 standard, uses a 64-bit addressing scheme. This scheme permits

firewire to have as many as 1023 buses on a bridged network and 63 nodes per bus. Figure 3.2 shows the layout of the 64-bit addressing scheme used in a firewire network.

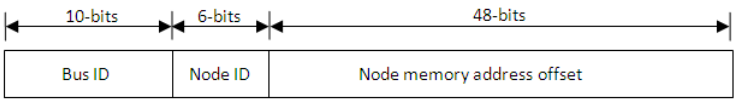


Figure 3.2: CSR addressing scheme used in IEEE 1394 serial bus

The CSR addressing scheme uses 10-bits to address the firewire bus, 6-bits to identify a node on the bus, and 48-bits to address the particular memory offset within the node. Hence this scheme allows for direct memory addressing. A bus ID of 1023 (0x3FF) indicates that the message is addressed to the local bus, that is the bus on which the device transmitting the data is connected. When the node ID is 63 (0x3F), the message is a broadcast packet that is picked up by all nodes on the bus.

The CSR addressing scheme is used to direct packets on a firewire network. These firewire packets could carry either control messages (including commands and responses) or data. The ways in which data can be transferred on a firewire network are discussed in the following section.

3.1.2 Data transfers

Firewire allows for two types of data transfer:

- Asynchronous data transfer, known as an asynchronous transaction - allows for guaranteed data delivery.
- Isochronous data transfer, known as an isochronous transaction - allows for timeous data transmission.

An asynchronous transaction involves the transfer of asynchronous packets and is generally used when the reception of transmitted data is crucial. In an asynchronous transaction the transmitting node is referred to as the *requester,* while the receiving node is referred to as the *responder* (Anderson, 1999). An asynchronous transaction works on the principle that every packet transmitted is acknowledged. This way the requester is able to verify that the packet was delivered. If no acknowledgment is received from the responder, the requester retransmits the asynchronous

packet. This guarantees that the asynchronous packet is delivered. Figure 3.3 illustrates the nature of asynchronous transactions.

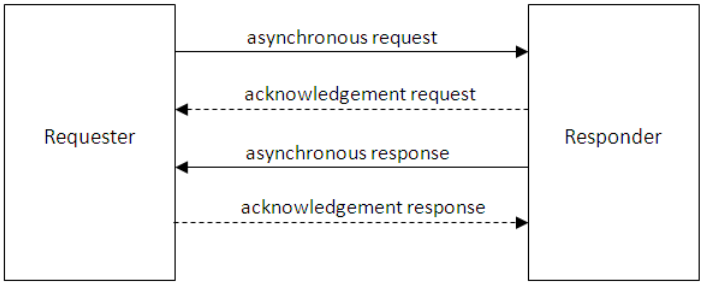


Figure 3.3: Asynchronous transaction

There are three transaction types supported by an asynchronous transaction on firewire. These are read, write and lock transaction types (Anderson, 1999). In an asynchronous read transaction the requester transmits an asynchronous request on the bus to read a value from an address within a responder node, indicating the responder's node ID and the specific memory address. An asynchronous write transaction is used by a requester node to modify the value of an address location on a specified node. An asynchronous lock transaction is used to modify values on a device while handling race conditions. A race condition will occur when more than one process wishes to change the same value. An asynchronous lock transaction involves three operations, namely:

- read the original data from the specified address
- modify the data
- write modified data to address

These three lock transaction operations are atomic in nature. That is they either succeed as a whole or fail as a whole. This means that in the instance that one of the operations fails, the entire lock transaction fails.

Isochronous data transfer ensures that data transmission occurs within a specified time period. Here isochronous packets are transmitted at a constant rate. There are no acknowledgments for received isochronous packets, but there is a guarantee that the required resources (bandwidth and transmission channel) are made available for data transfer. Before a firewire node can transmit an isochronous packet it requests a transmission channel and required bandwidth from the

isochronous resource manager (IRM) (Anderson, 1999). As the name implies, the IRM manages the bus resources for isochronous transactions (see section 3.1.5 on page 38).

A *cycle start* packet, transmitted by a node fulfilling the role of *cycle start master*, is transmitted every 125 microseconds to indicate the start of an isochronous transmission. Isochronous packets are transmitted on a particular channel and only nodes that are set to receive on the indicated channel can pick up the packets.

In the next section, the firewire configuration ROM is described. Every firewire device possesses a configuration ROM that holds information necessary to identify a device. The information obtained from the configuration ROM is used to identify the relevant software necessary for device control.

3.1.3 Configuration ROM

A functional unit within a firewire device is known as a node. The IEC 13213 specification, on which IEEE 1394 standard is built, specifies that each firewire node will possess a configuration ROM (Read Only Memory). Using the information in a device's configuration ROM (also referred to as config ROM), an application is able to identify the software driver and other diagnostic software required to communicate with the device (Anderson, 1999).

The layout of a device's config ROM can be either in the *minimal ROM format* or the *general ROM format* (Anderson, 1999). In the minimal ROM format the most significant 8-bits of the memory address has value 0x01. Following this is a 24-bit *vendor-ID* used to uniquely identify the device manufacturer. Figure 3.4 shows the layout of the minimal config ROM format.

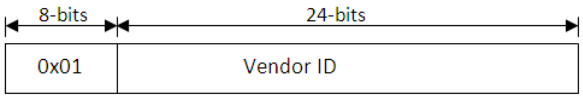


Figure 3.4: Minimal config ROM format layout adapted from Anderson (1999)

The general config ROM format, being more informative, holds information that enables bus management, power management and node device configuration. At start up, an application reads information from various addresses within the config ROM to get an insight into the device it is dealing with (diagnostic information). Figure 3.5, adapted from the IEC 61883-1 specification (2008), depicts the layout of the general config ROM format (IEC, 2008).

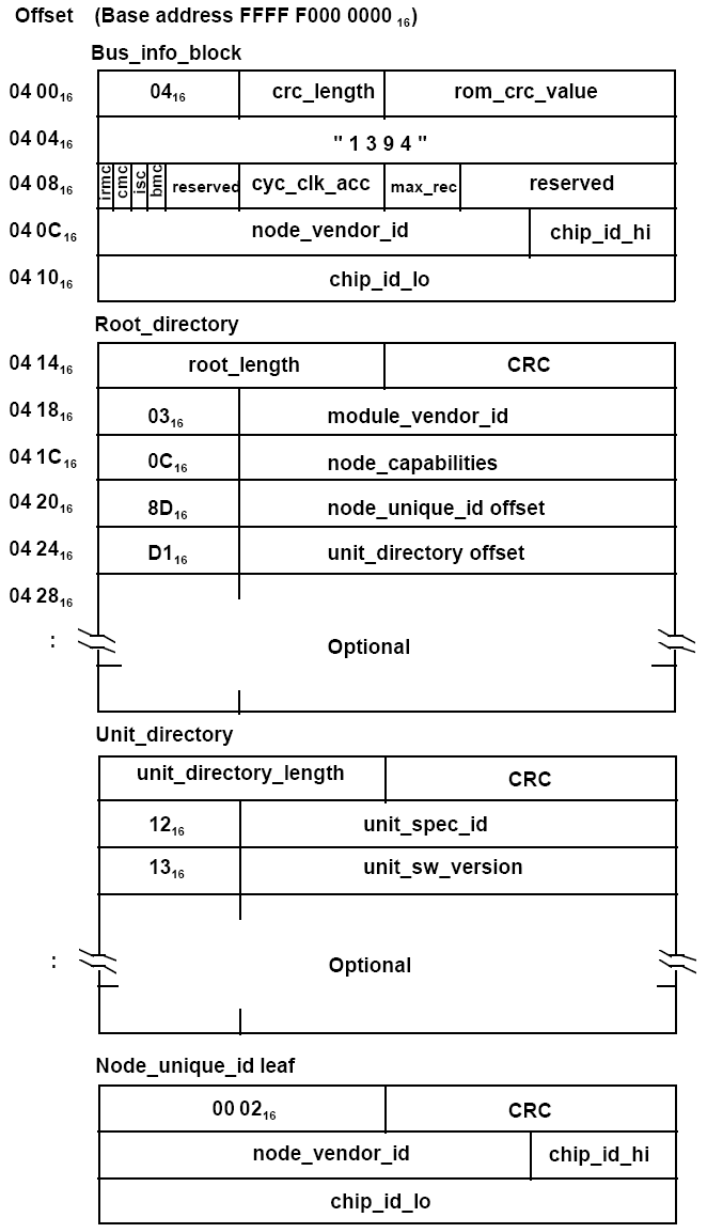


Figure 3.5: General config ROM format layout adapted from IEC (2008)

The bus info block within the config ROM can be used to uniquely identify a firewire device by its *node_vendor_ID*, *chip_ID_hi* and *chip_ID_lo*. These three fields together form what is referred to as a globally unique identifier (GUID) (Foss, 2007). A device's GUID is 64-bit in size.

The config ROM also comprises one or more unit directories as depicted in figure 3.5. Although not mandatory according to the IEEE standards, a firewire device typically possesses as many

unit directories as its number of units. It is within the unit directory that information that assists in determining the relevant software for a device can be found (Anderson, 1999). Specifically, the *unit_spec_ID* and the *unit_sw_version* fields within the unit directory are used for software driver and diagnostic software determination.

The *unit_spec_ID* is a unique number that identifies the organization that established the protocol implemented by the unit. This unique number is obtained from the IEEE Registration Authority (IEEE, 2009). The *unit_sw_version* indicates what protocol is being implemented by the unit.

3.1.4 Bus reset

When a device is removed from or added to a firewire network a combination of processes known as *bus reset* is initiated. A bus reset can also be triggered by an application. Whatever means by which a bus reset is initiated, it results in a possible change in the node IDs of devices that were already on the bus before the reset and hence a possible change in the bus management roles.

A bus reset is itself comprised of the following processes (Anderson, 1999):

- Bus Initialization - a RESET signal is propagated from one node to all the other nodes on the bus with the final result being a clearing of topology information from all nodes on the bus.
- Tree Identification - following the bus initialization process, tree identification results in a new topology being created and one node assuming the role of root. The root node is the node with the highest node ID. The self identification process follows.
- Self Identification - this process results in each node obtaining a node ID by which they can be uniquely identified. This process is initiated by the root node who sends self-ID grant packets to its 'child' nodes.

Anderson (1999) provides details on the bus reset mechanism. After a bus reset the following role players have to be determined:

- *Cycle master* - determines the start of isochronous transmission by sending cycle start packets every 125 microseconds.
- *Isochronous resource manager* - used for the allocation of bandwidth and channel numbers for isochronous data transmission.

- *Bus manager* - keeps bus topology and speed information of connected nodes.

On a firewire network, the above three roles are referred to as *bus management roles*. The isochronous transfer of data on the firewire bus is particularly dependent on the roles played by the cycle master and the isochronous resource manager. The following section describes the role of the isochronous resource manager on a firewire network.

3.1.5 Isochronous Resource Manger (IRM)

Before a node can transmit isochronous packets it requires that sufficient bandwidth be available on the bus, and that it has available to it a channel it can transmit on. A channel can be allocated to only one isochronous stream. The IEEE 1394 standard provides a way of monitoring transmission bandwidth and channels. The monitoring and allocation of bandwidth and channels to transmitting nodes is the responsibility of the isochronous resource manager (IRM). After a bus reset, isochronous resource capable nodes contend for the role of IRM by sending a self-ID packet whose *link layer active* and *contender* fields are set (Anderson, 1999). The contending node with the highest node ID (closest to the root node) wins the competition. Typically, if a root node is IRM capable, it wins the competition and fulfills the role of IRM.

The IRM possesses a CHANNEL_AVAILABLE and a BANDWIDTH_AVAILABLE register (Anderson, 1999). These registers are used in monitoring what channels have been allocated and the amount of bandwidth available for transmission on the bus. A node seeking to transmit on the bus performs an asynchronous lock transaction on the IRM's CHANNEL_AVAILABLE register to obtain a particular channel for transmission. If the lock transaction fails, it implies that the desired channel is already in use by another isochronous stream and therefore cannot be reallocated at that moment. The requester may then choose to obtain another channel by performing another lock transaction on the CHANNEL_AVAILABLE register of the IRM. Similarly, such a node will perform a lock transaction on the IRM's BANDWIDTH_AVAILABLE register, specifying the amount of bandwidth it requires. If the desired bandwidth is less than the current value in the IRM's BANDWIDTH_AVAILABLE register, it is deducted from the current value in the BANDWIDTH_AVAILABLE register and success is signaled to the requester. If the requested bandwidth is more than the available bandwidth in the BANDWIDTH_AVAILABLE register, the lock transaction fails and requester cannot transmit data on the bus. Bandwidth is allocated in *bandwidth allocation units* (Anderson, 1999).

After a device has obtained the isochronous channel and bandwidth it requires, isochronous data transmission can commence.

3.1.6 Data transmission

As mentioned earlier, firewire allows for asynchronous and isochronous transactions. Asynchronous transactions involve the transfer of asynchronous packets on the serial bus, while isochronous transactions involve the transfer of isochronous packets. The nature of an asynchronous packet depends on the type of asynchronous transaction. Possible asynchronous transaction types, each distinguished with its *transaction code* (values in hex), are (Anderson, 1999):

- Asynchronous write transactions - these include *write quadlet request* (0x0), *write block request* (0x1) and *write response* (0x2).
- Asynchronous read transactions - these include *read quadlet request* (0x4), *read block request* (0x5), *read quadlet response* (0x6) and *read block response* (0x7).
- Asynchronous lock transactions - these include *lock request* (0x9) and *lock response* (0xB).
- Cycle start packet - is a broadcast packet transmitted by the firewire cycle master every 125 micro-seconds to signify the start of a transmission cycle.
- Asynchronous stream packet - is a data transmission packet that is transmitted in an asynchronous transmission time interval. Due to its structural resemblance to an isochronous packet it is commonly referred to as a 'loose isochronous packet'. The transmission of asynchronous stream packets may not be permitted by the PHY on some firewire devices (Anderson, 1999).

Figure 3.6 depicts the general structure of an asynchronous packet as adapted from Anderson (1999).

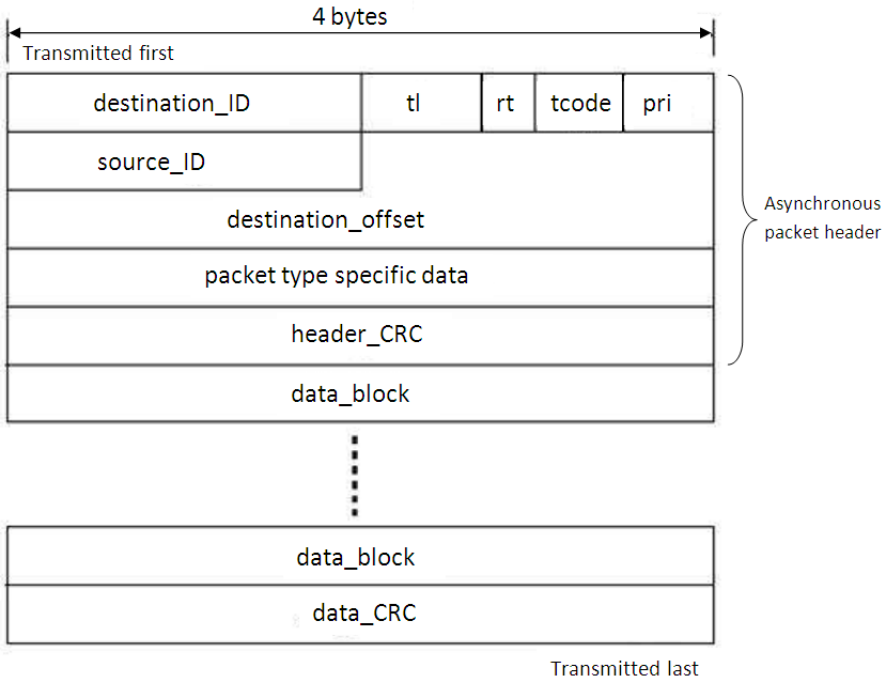


Figure 3.6: Layout of a firewire asynchronous packet adapted from Anderson (1999)

The asynchronous packet has in its header the source and destination device IDs to identify the source of the asynchronous packet and the device to which the packet is addressed. The transaction code (*tcode*) determines what type of asynchronous packet is being transferred. For instance a packet with *tcode* value 0x0 is a request to write an asynchronous data quadlet to the destination device's *destination offset* address (Anderson, 1999). This offset address (*destination_offset* field) resides within the device indicated by the *destination_ID* field.

The size of the data block in an asynchronous quadlet write or asynchronous quadlet read packet is 4 bytes. If the data block size exceeds 4 bytes, an asynchronous block write or asynchronous block read is used instead. If the size of the data block (in bytes) is not divisible by four (size of a quadlet), it becomes necessary to add zeroes to the least significant positions, to fill the data block. The addition of zeroes to complete the size of a quadlet in a data block is referred to as *zero-padding*.

The layout of an isochronous packet is depicted in figure 3.7 on the following page.

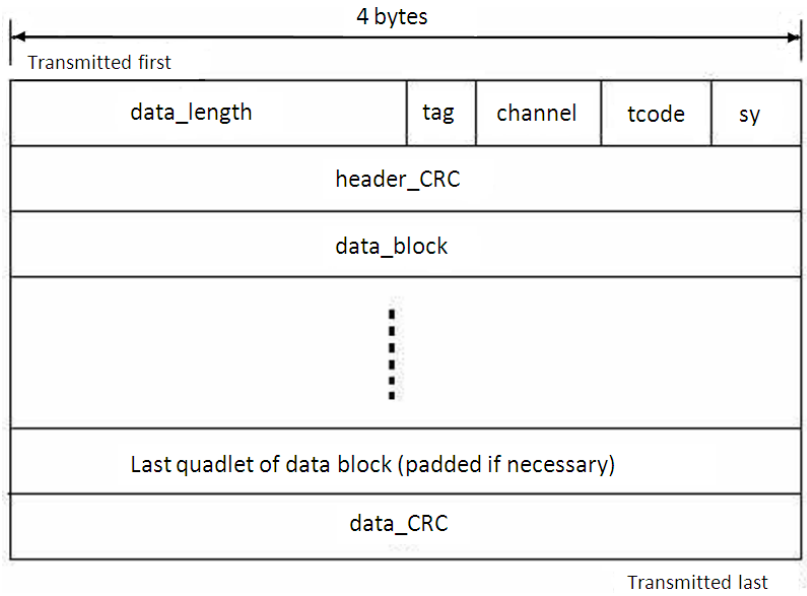


Figure 3.7: Layout of a firewire isochronous stream packet adapted from Anderson (1999)

The *data_length* field specifies the size of data being transmitted in bytes. The *tag* field indicates whether the data being transported in the isochronous packet is formatted or not. A *tag* value of 0x0 indicates that the isochronous data being transported is not formatted. Each isochronous packet specifies what channel it is transmitting on in its *channel* field. An isochronous packet has a *tcode* field with value 0xA. The *sy* field is used for synchronization (Anderson, 1999).

Firewire allocates 80% of the available bus bandwidth for isochronous transactions. An asynchronous transactions, that is transactions involving the transfer of asynchronous packets, is transmitted within the remaining 20% of the serial bus bandwidth. A *cycle start* packet is transmitted by a node known as the *cycle master* which is usually the *root node*, once every 125 microseconds (Anderson, 1999). This period (125 microsecond) is known as a *nominal cycle period* within which packets (isochronous and asynchronous) are transmitted. Figure 3.8 on the next page illustrates the procedure for transmission within a cycle.

Following the transmission of a cycle start packet isochronous transmission can proceed. An *isochronous gap* is a time interval that exists between isochronous packets. A *subaction gap* is a longer time interval than the isochronous gap and it exists before the transmissions of asynchronous packets in a nominal cycle period. The isochronous transmission with its associated isochronous gaps and the asynchronous transmission with its associated subaction gaps must all occur within a nominal time period of 125 microseconds.

On a firewire bus a time interval known as a *fairness interval* exists, within which time all nodes wishing to perform an asynchronous subaction are allowed to do so at least once. An asynchronous subaction refers to either an asynchronous read, write, or lock transaction with its associated acknowledgment. Although asynchronous subactions are atomic, an interval known as an *acknowledgment gap* exists between sending an asynchronous packet and receiving an acknowledgment packet. No bus activity is allowed to take place during an acknowledgment gap (Anderson, 1999). An *arbitration reset gap* is the interval that exists just before the start of a fairness interval.

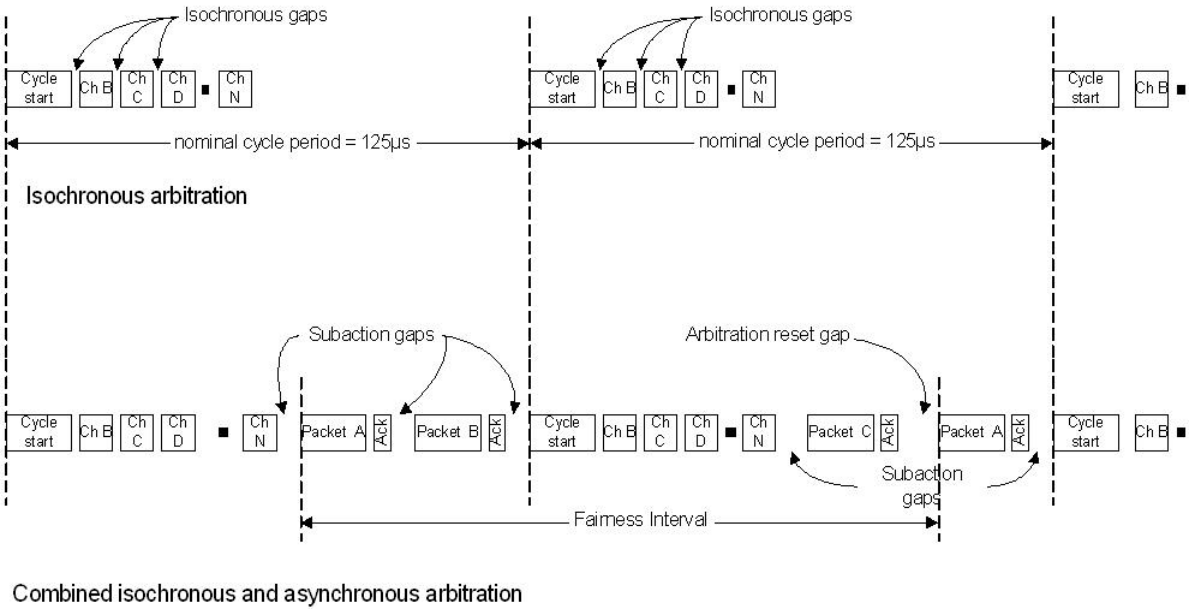


Figure 3.8: Isochronous arbitration and combined asynchronous and isochronous arbitration adapted from Foss (2007)

The transmission of isochronous packets can either be in *blocking mode* or *non-blocking mode*. In a blocking isochronous packet transmission a device waits for a specified number of events to occur before it transmits the isochronous packet. Irrespective of how many nominal cycle periods elapse, the device will keep waiting for the specified number of events to occur. In the case of a non-blocking transmission, the device transmits the isochronous packet within the current isochronous transmission period irrespective of how many events occurred (Foss, 2007).

3.1.7 Isochronous data flow

From figure 3.7 it can be seen that an isochronous packet comprises a *header* and a *data block* section. In the header section (already discussed in section 3.1.6) a channel for transmission of the isochronous packet is specified. The data block section contains the audio samples, MIDI signals, or other digital data being transported.

Isochronous packets with the same channel number form an *isochronous stream* (Foss, 2007), as illustrated in figure 3.9. In figure 3.9, packet A, B and C are isochronous packets being transported on channel 4 as indicated in their packet header. Thus all three packets are conceptually transported as a single isochronous stream over firewire.

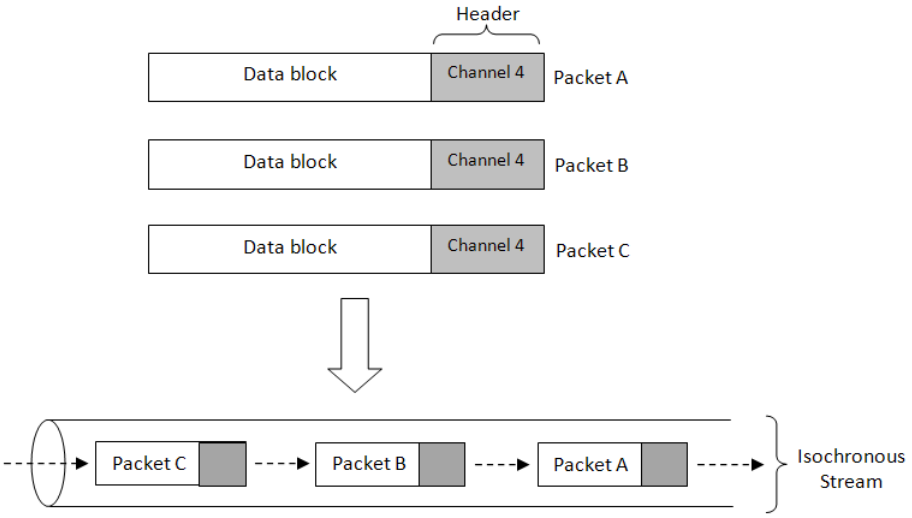


Figure 3.9: Isochronous stream

Digital audio devices (in particular firewire devices) sample audio data at various sampling rates. For instance, a professional audio mixing desk might be sampling at 48 KHz and might be required to transmit the sampled audio over firewire. However the transmission of isochronous packets on a firewire bus is triggered by a cycle start packet from the cycle master every 125 microsecond, that is 8,000 times per second or 8 KHz. As a result the device presents more signals than can be transported by the bus every second. Figure 3.10 depicts this problem.

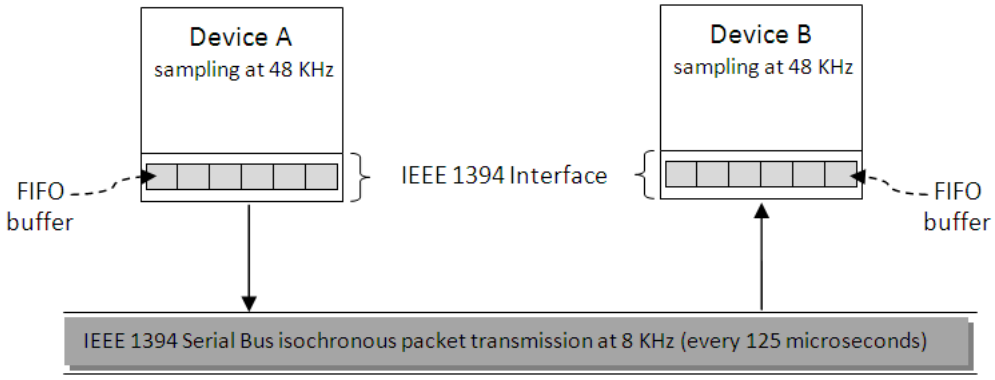


Figure 3.10: Isochronous packet transmission on a firewire network

In figure 3.10, device A and B are both firewire devices, each having a sampling frequency of 48 KHz. Device A is a transmitting device wanting to transmit data to device B through its firewire interface. Device B picks up the transmitted data at the same sampling rate that data is being transmitted by device A, that is at 48 KHz. Device A samples at a much higher rate (48,000 samples per second) than the firewire bus can transmit (at 8,000 samples per second), hence more samples are present at the firewire interface than can be transmitted over firewire every cycle.

To overcome the problem described above, the IEC has defined a *Common Isochronous Packet* (*CIP*) header in a specification titled *Digital Interface for Consumer Audio Equipment - General* (IEC, 2007). An isochronous packet that has the CIP header can be identified with its *tag* field value of 0x01. The CIP header resides as the first two quadlets of the datablock within an isochronous packet (IEC, 2008). The IEC also defines an isochronous data packaging scheme in its *Digital Interface for Consumer Audio Equipment - Audio and Music data transmission* specification (IEC, 2005). The IEC 661883-6 specification defines the following concepts (Foss, 2007).

- *Isochronous stream position* (also referred to as a *sequence*) - is a signal position in an isochronous stream.
- *Data block* - is a cluster comprising of audio/MIDI data that are sampled at a particular time.
- *Element* - is a position within a data block. Audio signals are sampled as elements and grouped to form a data block.

- *Event* - is the audio/MIDI signals available for transmission at a particular time. A data block could be seen as comprising an event.

Using these concepts audio/MIDI data can be clustered into data blocks which are encapsulated within isochronous packets for onward transmission on a firewire bus. The receiving device then extracts the data from the isochronous packet, adhering to the same principles by which the data was encapsulated. In figure 3.11 the isochronous packet represented comprises six data blocks with each data block having five elements. These elements refer to positions within the data blocks, and each element relates to a corresponding position within the isochronous stream. The five audio sequences are *'bundled'* into an isochronous stream. The first element of each data block is associated with *sequence 1* of the isochronous stream.

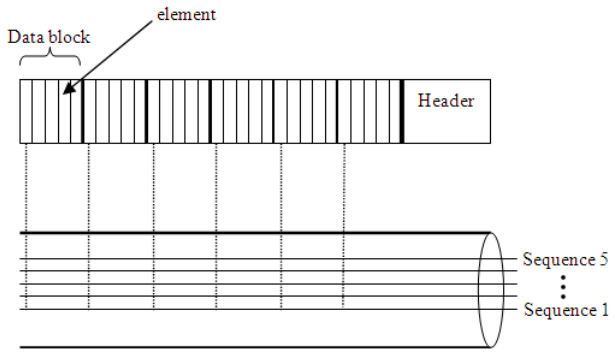


Figure 3.11: Isochronous sequences adapted from Foss (2007)

The number of data blocks transmitted in an isochronous packet is determined by the sampling rate and transmission frequency of the device. In accordance with the IEC 61883-6 specification, each element within a data block is 32-bit in size, and can be formatted according to an *AM824* data format, also defined in the IEC 61883-6 specification. The IEC 61883-6 specification also permits MIDI to be multiplexed and transported using the *AM824* data format. A maximum of eight MIDI streams can be multiplexed into a single sequence (Foss, 2007).

The CIP header of an isochronous packet provides information about the nature of the audio/MIDI data being transported in its isochronous packet. Within the CIP header the *FMT* field with a value of 0x10 implies that the data conforms to the IEC 61883-6 specification. The number of quadlets in the isochronous packet's data block is specified in the CIP header's *DBS* field (IEC, 2008).

A number of chip manufacturers have created *Application Specific Integrated Circuit* (ASIC) chips for the encapsulation and extraction of isochronous packets conforming to the CIP format. Each of these ASIC chips has particular audio/MIDI signal processing capabilities. These ASIC chips possess FIFO buffers (depicted in figure 3.10 on page 44) that are temporary holding places for audio/MIDI signals to be transmitted or received over firewire (Foss, 2007).

3.1.8 Synchronization on a firewire network

On a firewire network the transmission of audio/video involves a transmitting device packaging the data into an isochronous packet and then 'pushing' it onto the (firewire) bus. The receiver 'picks up' this packet and extracts the information within it. For the receiving device to extract the received packet, it is required that the receiver:

- is capable of processing the data format. Isochronous packets that possess the common isochronous packet (CIP) header specify the data format in the FMT field. CIP is discussed in section 3.2.4.
- has a clock that is synchronized with that of the transmitter.

To achieve clock synchronization the firewire cycle master (mentioned in section 3.1.4) transmits a broadcast cycle start packet every 125 microseconds. This cycle start packet holds the value of the cycle master's CYCLE_TIME register, which is incremented (for mLAN devices) at 24.576MHz (Foss, 2007, 117). All devices on the network pick up this cycle start packet, hence update their CYCLE_TIME register values with that of the cycle master.

A transmitting device, time-stamps an isochronous packet by setting the CIP header's SYT field to a value referred to as the *presentation time* (Foss, 2007, 117). The presentation time is a summation of:

- the CYCLE_TIME register of the transmitting device.
- a TRANSFER_DELAY or offset value which compensates for:
  - the time required for the packet to travel from transmitter to receiver
  - short bus reset

The DEFAULT_TRANSFER_DELAY is 354.17 microseconds (Foss, 2007, 119). This time-stamp is placed into the SYT field at various intervals depending on the sampling frequency of the transmitting device. These intervals are referred to as SYT_INTERVALs and are defined in the IEC 61883-6 (2005) specification as shown in table 3.1.

SYT_INTERVAL	Sampling Frequency
8	32kHz
8	44.1kHz
8	48kHz
16	88.2kHz
16	96kHz
32	176.4kHz
32	192kHz

Table 3.1: SYT_INTERVAL for different firewire sampling frequencies (IEC, 2005, 33)

As shown in table 3.1, a transmitting device with a sampling frequency of 48kHz time-stamps in blocking mode, waits for 8 samples before transmitting on the bus.

A receiving device:

- picks up the isochronous packet,
- reads the time-stamp on the packet,
- waits for its CYCLE_TIME register value to be the same as the *presentation time* of the isochronous packet, and
- then sends a match to its phase-locked loop (PLL).

The PLL then generates the sampling frequency of the transmitting device by multiplying the frequency at which it receives a match with the SYT_INTERVAL value which it obtains from the format dependent field (FDF) field of an audio and music data format. FDF is discussed in section 3.2.4.

The next section discusses some IEC 61883-1 requirements for the transmission of audio and MIDI data over firewire.

3.2 Digital and Consumer Audio Equipment

The IEC 61883-1 specification defines a digital interface required for the transfer of isochronous data over firewire. It defines the connection types and various procedures required to set up and manage firewire connections. The IEC 61883-1 defines:

- serial bus *plugs* as virtual end points of isochronous streams that are accessible through registers within the firewire device.
- a *Common Isochronous Packet* (CIP) header that carries isochronous stream format information necessary for the consistent transmission of isochronous packets.
- a command and response mechanism known as the *Function Control Protocol* (FCP) that can form the basis of a high level control protocol such as *AV/C* (Audio/Video Control) protocol.

This section discusses some of the concepts in the IEC 61883-1 specification.

3.2.1 Identifying devices that conform to the IEC 61883-1 standard

Within the configuration ROM of firewire devices is a unit directory that holds information specific to the units that exist within the device. The information within the unit directory can be used to determine what software the *unit*[2] will respond to. Two fields in particular are used for this purpose, namely the *unit specifier ID* and the *unit software version*. These are both 24-bit fields that reside within the unit directory of a device that conforms to the general config ROM format (see figure 3.5 on page 36).

All firewire devices that conform to the IEC 61883-1 specification have a *unit specifier ID* of 0x00A02D (IEC, 2008). To determine whether a device conforms to the IEC 61883-1 specification, an application can read the *unit specifier ID* to confirm whether its value is 0x00A02D. Having this information enables an application to know what to expect of the device in terms of plug registers, connection types and connection management procedure.

The *unit software version* has its most significant 8 bits (of the 24 bits) set to 0x01. This is followed by two bytes that indicate the command/transaction set the unit will respond to. These

[2] A 'unit' is a functional entity within a firewire device

two bytes also appear in the *cts* field of the *Function Control Protocol* (FCP) frame (discussed in section 3.2.5 on page 55) (IEC, 2008). For instance, a device whose unit conforms to the AV/C protocol (see section 3.3 on page 56) has the least significant bit of its *unit software version* set to '1' and will respond to asynchronous packets with FCP *cts* field value of 0x0000. Hence an application wanting to determine whether a device adheres to AV/C can get such information by reading the *unit software version* field within the device's general config ROM.

3.2.2 Plugs and plug registers

Plugs are virtual end points of firewire isochronous streams. These plugs are accessible through registers on the firewire device. The IEC 61883-1 specification defines 32-bit registers that correspond to isochronous stream plugs on a firewire device. A 32-bit master plug control register holds information that pertains to all plugs flowing in a particular direction.

There are two possible directions for the flow of isochronous streams with regards to a firewire device, as illustrated in figure 3.12. An input plug permits the flow of data into the device while an output plug permits data flow out of a device. In figure 3.12, the firewire device possesses two input and two output plugs. Input plug 1 and input plug 2 receive an isochronous stream on a particular channel and allow data to enter the device. Output plugs 1 and output plug 2 are used to transmit data from the device onto the bus.

Figure 3.12: Direction of isochronous stream flow

The data flows referred to here are isochronous streams. An isochronous stream flows on a particular channel from an input plug to an output plug. Figure 3.13 adapted from the IEC 61883-1 specification, depicts the layout of IEC 61883 compliant device plug registers.

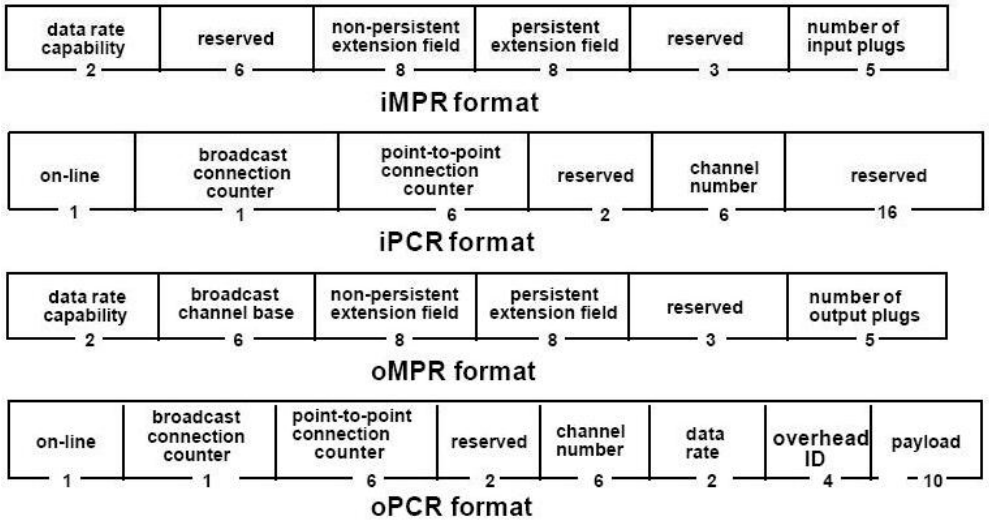


Figure 3.13: Input and output plug registers

In terms of a device's input, an input master plug register (iMPR) holds information that pertains to all the input plugs on a device. As depicted in figure 3.13 the iMPR has a field (*data rate capability*) that holds information about the maximum speed at which isochronous streams can be received by the device. It also holds information about the *number of input plugs* on the device. Thus one way for an application to determine the number of input plugs on a device is to read the iMPR's *number of plugs* field. The iMPR is mapped to memory offset address 0xFFFF F000 0980 (IEC, 2008).

A device's input plug is accessible through its input plug control register (iPCR). The iPCR (depicted in figure 3.13) specifies:

- the channel on which the plug is set to receive data in its *channel* field
- whether or not the plug is capable of transmitting (*online*)
- the number of broadcast connections to the input plug (*broadcast connection counter*)
- the number of point-to-point connections to the input plug (*point-to-point connection counter*).

Since the *number of input plugs* field of a device's iMPR can have a maximum value of 31, the maximum number of isochronous stream input plugs a device can possess is 31. Hence a device can have a maximum of 31 iPCRs each corresponding to an isochronous stream plug.

In terms of a device's output, an output master plug register (oMPR) holds information that pertains to all the output plugs on a device. As depicted in figure 3.13, the oMPR holds information about the maximum speed at which isochronous streams can be transmitted by the device (*data rate capability* field) and the number of isochronous stream output plugs on the device (in its *number of output plugs* field). The oMPR is mapped to memory offset address 0xFFFF F000 0900 (IEC, 2008).

A device's output plug is accessible through its output plug control register (oPCR). The oPCR (depicted in figure 3.13) specifies (IEC, 2008):

- whether or not the plug is capable of transmitting (*online* field)
- the number of broadcast connections from the output plug (*broadcast connection counter* field)
- the number of point-to-point connections from the output plug (*point-to-point connection counter* field)
- the channel on which the plug is set to transmit data *(channel* field)
- the bit-rate at which the plug transmits isochronous packets.

Similar to what pertains to the input, the *number of output plugs* field of a device's oMPR can have a maximum value of 31. This implies that a device can have a maximum of 31 isochronous stream output plugs, hence a maximum of 31 oPCRs.

3.2.3 Connection types

There are two types of connections that exist between firewire devices, as defined by IEC 61883-1. These are *point-to-point* connections and *broadcast* connections.

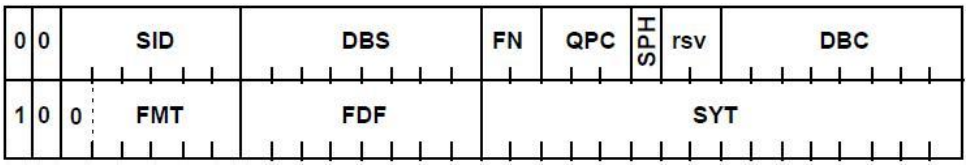
A *point-to-point* connection is a secure connection between the isochronous stream plugs of two firewire devices. In a point-to-point connection, one of the plugs is an output plug and the other an input plug. This type of connection is a secure connection between two devices. The point-to-point counter field of the output plug and the input plug is increased for each point-to-point connection. An input or output plug can be involved in multiple point-to-point connections. Only the initiator of a point-to-point connection can break it (IEC, 2008).

There are two types of *broadcast* connections. A *broadcast-in* connection is established between an input plug and the serial bus. In a broadcast-in connection the isochronous stream input plug receives isochronous streams on a broadcast channel from the bus. A *broadcast-out* connection is established between an output plug and the serial bus. In a broadcast-out connection the output plug transmits isochronous stream packets on a broadcast channel. These packets can be picked up by any device with the same channel set in its broadcast-in connection to the serial bus. A broadcast-in and broadcast-out together are referred to as a broadcast connection. A broadcast connection can be broken by either of the devices involved in the connection. An isochronous stream plug can only be involved in one broadcast connection (IEC, 2008). When a broadcast connection (broadcast-in or broadcast-out) has been established on a plug, the plug's control register broadcast connection counter is increased.

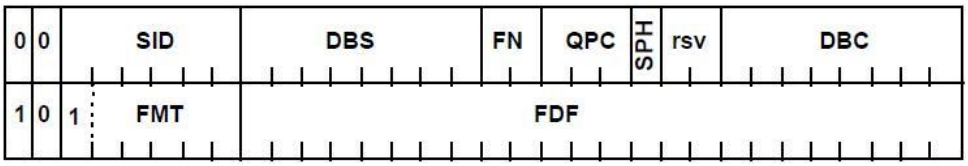
3.2.4 Common Isochronous Packet (CIP)

The Common Isochronous Packet (CIP) header is defined in the IEC 61883-1 (2008) specification. This header is used to pass information pertaining to the arrangement of data within a firewire isochronous packet. It resides in the first two quadlets within an isochronous packet's data block. An isochronous packet that implements CIP has a *tag* value of '0x01' in its isochronous packet header (IEC, 2008). CIP also provides for synchronization between a transmitting device and a receiving device with its *SYT* field.

The two-quadlet CIP header defined in IEC 61883-1 has two forms. The distinction between both forms is the presence or absence of an SYT field. Figure 3.14 shows the two forms of a two-quadlet CIP header.



(a) Form 0



(b) Form 1

Figure 3.14: Common Isochronous Packet header adopted from IEC (2008)

Described below are the CIP header fields as defined in IEC 61883-1 (2008):

- SID - is the identifier (node ID) of the device transmitting the isochronous packet.
- DBS - specifies the size (in quadlets) of the data blocks that make up the isochronous packet.
- FN - specifies the number of data blocks into which an isochronous data from a transmitting device is divided.
- QPC - specifies the number of zero padding (in quadlets) within an isochronous packet.
- SPH - indicates whether the isochronous packet includes a source packet header.
- DBC - is a counter appended to an isochronous packet and is used in detecting packet loss.
- FMT - indicates the format of the data in the isochronous packet.
- FDF - is a format dependent field whose meaning and definition is dependent on the FMT field.
- SYT - is used for data synchronization. A transmitting device places a time stamp from the lower 16-bits of its CYCLE_TIME into the SYT field.

As can be seen in figure 3.14, a CIP header that transmits timing information in its SYT field has the most significant bit of its FMT field set to ‘0’. This SYT field is derived from the

CYCLE_TIME register of a transmitting device with an offset value added to it (see section 3.1.8 on page 46 for a discussion of synchronization on a firewire network). All firewire devices capable of isochronous transactions implement the CYCLE_TIME register.

An FMT value of *0x10* indicates that the data within the isochronous packet follows the audio and music (AM) format. The IEC 61883-6 (2005) defines the AM824 data format. AM824 is an 8-bit label and 24-bit data format. Conformance to the AM824 format is dependent on the EVT field of the generic FDF field layout shown in figure 3.15.

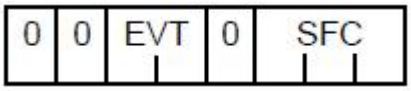


Figure 3.15: Generic FDF field layout adapted from IEC (2005, 30)

IEC 61886-6 (2005) defines the following EVT values shown in table 3.2.

EVT Value	Description
0	AM824 Data
1	24-bit * 4 Audio pack
2	32-bit floating point data

Table 3.2: EVT values for the generic FDF field (IEC, 2005)

The *sampling frequency code* (SFC) field specify the sampling frequency of the transmitting device. The values of the SFC field as defined in IEC 61883-6 (2005) are shown in table 3.3.

SFC value	Sampling frequency
0	32kHz
1	44.1kHz
2	48kHz
3	88.2kHz
4	96kHz
5	176.4kHz
6	192kHz

Table 3.3: SFC values for the generic FDF field (IEC, 2005)

3.2.5 Function Control Protocol (FCP)

The IEC 61883-1 specification defines a *Function Control Protocol* (FCP) that forms the basis of other higher level protocols. FCP capitalizes on the ability of firewire to perform asynchronous transactions, hence it involves the transfer of FCP frames within firewire asynchronous packets. This establishes a command and response mechanism that is implemented as asynchronous transactions. An FCP frame is located within the data block section of an asynchronous packet. Figure 3.16 the layout of an asynchronous packet with its FCP frame is shown.

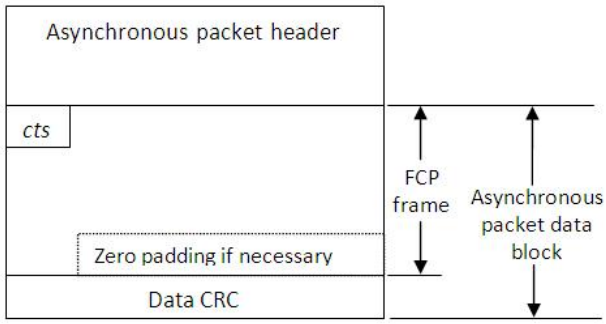


Figure 3.16: FCP frame within an asynchronous packet

FCP provides a mechanism which can be used by various protocols. The command/transaction set of the protocol that utilizes FCP is indicated by the value of the FCP frame's *cts* field. For instance an asynchronous packet that encapsulates an AV/C message will have the value of its FCP *cts* field set to all zeros.

A firewire device that conforms to the IEC 61883 specification possesses two FCP registers known as FCP_COMMAND and FCP_RESPONSE registers. These are 512 bytes in size and start at memory offset address 0xFFFF F000 0B00 for the FCP_COMMAND register and 0xFFFF F000 0D00 for the FCP_RESPONSE register (IEC, 2008). The FCP command and response mechanism is achieved by asynchronous writes to these registers as depicted in figure 3.17.

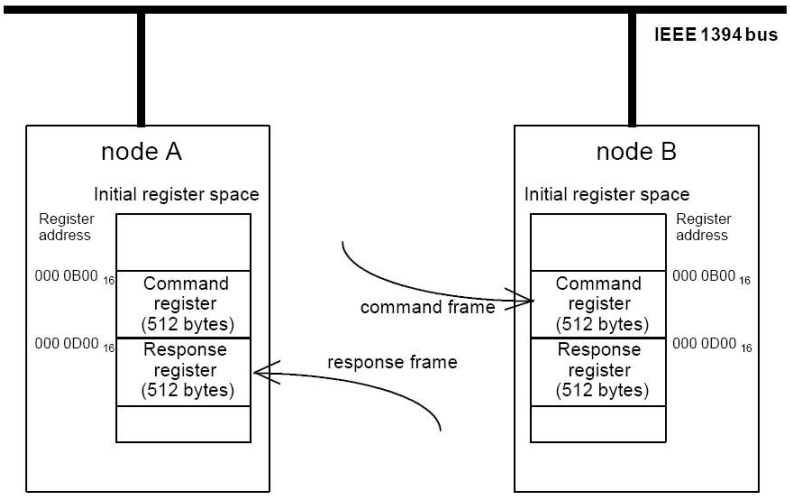


Figure 3.17: FCP registers within IEC 61883 devices adapted from IEC (2008)

A firewire node known as a *controller* node writes a command to the FCP_COMMAND register of another node, referred to as a *target* node. In response, the target node writes a response to the FCP_RESPONSE register of the controller node.

The AV/C protocol utilizes this command and response mechanism established by FCP to control audio and video devices. The next section explains the various components of AV/C.

3.3 Audio/Video Control (AV/C) Protocol

AV/C (Audio/Video Control) is a standard protocol for firewire device control. It is established as a command/transaction set that uses the FCP command and response mechanism (described in section 3.2.5 on the preceding page) for the control of digital audio and video devices. An AV/C command is transmitted asynchronously from a controller node to a target node. The target node responds by asynchronously transmitting an AV/C response to the controller node. This command and response mechanism involves the transmission of AV/C commands/responses within the FCP frame of an asynchronous firewire packet. The nature of an AV/C command/response is discussed in this section.

In the interaction between an AV/C control application and an AV/C device, two things are required by the application namely;

- to determine whether a particular firewire device is AV/C compliant

- to determine the features (isochronous stream plugs and unit/subunit information) that the AV/C device possesses.

As described in section 3.2.1 on page 48, the unit software version of a device gives an indication of what command/transaction set the device will respond to. In an AV/C device the last bit of the *unit software version* field is set to '1'. Hence a device that only supports AV/C transactions will have a unit software version value of 0x010001.

An AV/C device can be seen logically as composed of *units* that have within them *subunits*. A unit describes the entire functionality performed by an AV/C device. Within an AV/C unit are subunits that describe specific functionalities. For instance, a firewire breakout box can be seen as possessing an audio unit. Within this audio unit there could exist a music subunit that processes isochronous stream signals and an audio subunit that processes audio signals from *isochronous stream plugs*, *external plug*s or *asynchronous plugs*. An *isochronous stream plug* inputs or outputs isochronous signals from a firewire bus while an *external plug* inputs or outputs a signal from a signal source other than firewire. A device's *asynchronous plugs* inputs or outputs asynchronous streams to or from a firewire bus (1394 Trade Association, 2001b).

In the following subsections, there will be a description of:

- the structuring of an AV/C device in terms of its units and subunits.
- a description of the AV/C device exploration in an attempt to determine the features that an AV/C device possesses. This discussion will include how to
  - read a device's descriptors and information blocks
  - send general AV/C commands to obtain information about device features that the application seeks to control, and
  - send AV/C vendor specific commands to a device.
- an in depth investigation of two subunits, namely the audio and music subunits. These two subunits are of particular interest to this research and the reason for this will become obvious in chapter 4.

Given the varied number of available AV/C devices, including printers, digital cameras, breakout boxes and so on, there exists many AV/C standards documents for AV/C subunits implemented in

these devices. The 1394 Trade Association [3] maintains these AV/C standards documents (1394 Trade Association, 2009).

3.3.1 AV/C Command and Response Mechanism

As described earlier, AV/C is a command and response protocol for digital audio/video equipment control. This protocol is formalized by the 1394 Trade Association as a standard that allows for device discovery, exploration and control. AV/C is a command/transaction set that involves the transfer of AV/C commands from a controller node to a target node and the reception of AV/C responses (for corresponding commands) from a target node by a controller node. This command and response scheme is in compliance with the FCP protocol. Thus for every AV/C command that is sent to a target device's FCP_COMMAND register, a corresponding AV/C response will be received in the controller's FCP_RESPONSE register. AV/C commands and responses are encapsulated within the FCP frame. AV/C commands and responses are sent as asynchronous write request packets (1394 Trade Association, 2001b). Figure 3.18 shows the layout of an AV/C command.

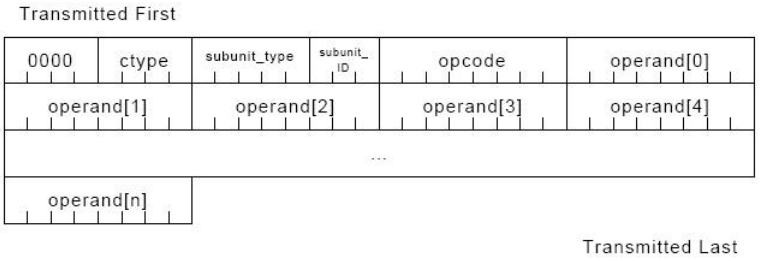


Figure 3.18: AV/C command packet adapted from 1394 Trade Association (2001b)

Within an asynchronous packet, the first 4-bits, which is the *cts* field of an FCP frame, are all '0's indicating that the supported protocol is AV/C. Following this field is a *ctype (command type)* field, which is also 4-bits in size. When an AV/C command is issued from a controller to a target, the *ctype* field is used to indicate the particular type of command that the controller seeks to send to the target node. When an AV/C response is issued from the target node to the controller, this field is known as a *response* field. The response field indicates the completion state of the command sent by the controller.

[3] 1394 Trade Association publishes and maintains standards relating to the IEEE1394 serial bus. For more information see http://www.1394ta.org

Two examples of commands are the AV/C *control* command and the AV/C *status* command. The *control* command with value '0' is used by a controller to instruct a target node to perform a particular operation (1394 Trade Association, 2001b), for instance when a controller wishes to set the value of a target's volume control. The *status* command with value '1' is used when a controller wishes to determine the state of a target node (1394 Trade Association, 2001b), for instance when a controller wishes to determine the current value of a target's volume control. Other *ctype* fields are *specific inquiry*, *notify*, and *general inquiry*. See (1394 Trade Association, 2001b) for a detailed explanation.

When a target node responds with an *accepted* or *implemented/stable* response, it is an indication that the command was successful and that the target implemented the command. A *rejected* or *not-implemented* response signifies an unsuccessful command, or that the target is still busy with another transaction. Other response types are *in-transition*, *changed* and *interim* responses. See (1394 Trade Association, 2001b) for a detailed explanation.

The *subunit type* and *subunit ID* fields uniquely identify a unit or subunit within an AV/C device. The *subunit type* is 5-bits in size while the *subunit ID* is a 3-bit field. By identifying a particular subunit, an AV/C command can be directed to specific controls within a device. Thus, the *subunit type* and *subunit ID* forms an addressing scheme that is used by the controller to specify what unit or subunit (within a target device) it intends to control or query.

The *opcode* is an 8-bit field that specifies the particular type of operation to be performed on the target node (1394 Trade Association, 2001b). An AV/C response will always return the same opcode value as indicated in the command for which it is a response. The *operands* are the arguments that the *opcode* will require to attain its purpose. Typically every opcode has more than one operand, and the number of operands is dependent on the particular *opcode*.

In AV/C, firewire device nodes contain addressable *units* and *subunits*. These units/subunits are instructed using AV/C commands to either change a control state or to return the status of a control. Controls refer to parameters within the device whose values can be modified to fulfill a particular function. The following section examines the structural layout of an AV/C device in terms of its units and subunits.

3.3.2 AV/C Unit/Subunit

An AV/C device comprises one or more logical entities known as *units* that each describe a functionality within the device. Within an AV/C unit are *subunits* that have specific roles, and

complement each other to fulfill the overall functionality of the AV/C unit. A subunit is a functional entity within a unit that performs a specific service for the unit it resides in. A unit may possess one or more subunits. Some subunits such as the audio subunit have *function blocks* within them (1394 Trade Association, 2001b). These function blocks can be seen as subunits within a subunit that each perform a specific action on behalf of their 'parent' subunit.

It is possible for an AV/C device to possess more than one subunit, for instance a unit might have an audio subunit, a music subunit, a video subunit and a tape recorder/player subunit [4]. Hence an AV/C command that is addressed to a target node needs to specify the particular unit/subunit it seeks to control in its *subunit type* and *subunit ID* fields. Figure 3.19 depicts an AV/C unit that contains two subunits.

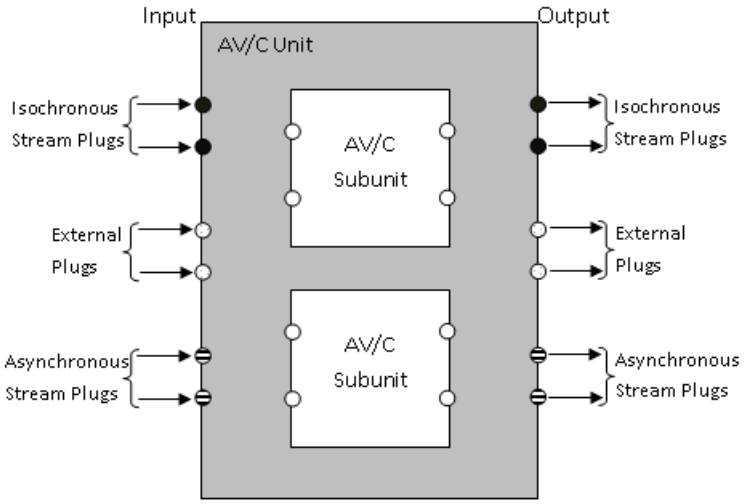


Figure 3.19: AV/C unit structural layout

An AV/C unit has a number of unit input plugs and unit output plugs. The unit input plugs are used to receive signals into an AV/C unit, while the unit output plugs are for transmitting signals out of an AV/C unit. These unit plugs are classified as either *serial bus asynchronous plugs*, *serial bus isochronous plugs* or *external plugs*. A maximum of 31 plugs can be implemented within a device (1394 Trade Association, 2001b).

The *serial bus asynchronous plugs* are virtual end points of asynchronous firewire data flow. The *serial bus isochronous plugs* are used for the transfer of isochronous streams. A unit's *external plugs* represents all non-firewire plugs that transmit signals to or from the AV/C unit. If a unit

[4]The subunit types; audio, music, video, and tape recorder/player, are some of the subunits defined by the 1394 trade association.

does not use a particular type of plug, it is not required to implement it (1394 Trade Association, 2001b).

An AV/C subunit possesses subunit plugs that serve as points of entry or exit depending on the direction of data flow. The *subunit destination plugs* are used for inputting signals into the subunit, while the *subunit source plugs* are used for outputting signals from the subunit. Similar to the unit plugs, the maximum number of source or destination plugs implemented on a subunit is 31 (1394 Trade Association, 2001b). Figure 3.20 depicts an AV/C subunit.

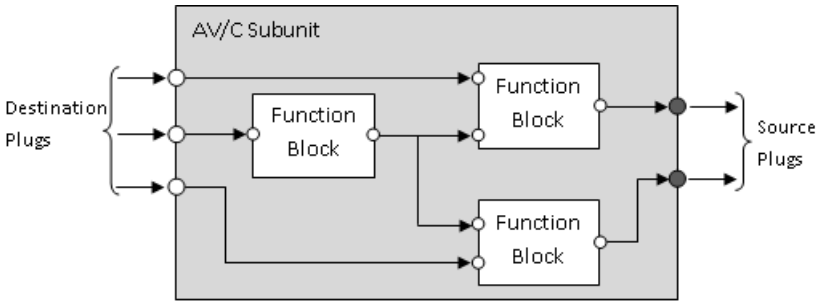


Figure 3.20: AV/C subunit

Each unit or subunit plug is uniquely identified by its *plug ID*. The function blocks within subunits also possess *function block plug*s. These function block plugs can be either function block input plugs or function block output plugs depending on the direction of signal flow.

A unit input plug receives signal from the firewire bus, a non-firewire signal source or from a unit output plug. A unit output plug receives signal from either a subunit source plug or from a unit input plug. A subunit destination plug receives signal from either a unit input plug or a subunit source plug. A subunit source plug transmits signal to either a unit output plug or subunit destination plug (1394 Trade Association, 2001b).

Having this understanding of the logical and functional entities that make up an AV/C unit, the next section discusses ways of exploring an AV/C device.

3.3.3 Exploring AV/C devices

In the previous section, an AV/C device was described as a node whose unit(s) may possess one or more subunits. This section describes some methods that have been used by control applications to obtain information about the various units, subunits, plugs and signal routing that exists within

an AV/C device. Firstly the *descriptor* and *information block* concept is explained. Then there is a discussion of the possible AV/C *status* commands that can be used to obtain information about an AV/C device. This section ends with a vendor specific approach to obtaining information about an AV/C device by sending certain vendor dependent AV/C commands.

3.3.3.1 Descriptors and Information Blocks Mechanism

In AV/C there exists a scheme that allows a device to present its internal information in a structured manner. This scheme also allows a controller to read and modify the information within a device by sending appropriate AV/C commands. It is known as the *descriptor* and *information block* (common referred to as *info block*) mechanism and is defined in the *AV/C Descriptor Mechanism Specification* (1394 Trade Association, 2001c).

Descriptors are structured data interfaces that a device presents to other firewire devices about its internal features and data structures. A descriptor is a blueprint of a device's features. The actual internal memory storage is dependent on the device, but the interface that it presents is standardized by descriptors (1394 Trade Association, 2001c).

The *AV/C Descriptor Mechanism Specification* defines three types of descriptors namely:

- *Unit/Subunit identifier descriptor* - which when implemented hold information that pertains to the entire unit/subunit.
- *List descriptor* - holds the lists of entries (*entry descriptors*) that are implemented in the unit/subunit.
- *Entry descriptor* - is an addressable set of information within a device's *descriptor hierarchy*. Entry descriptors may themselves contain other list descriptors.

A device's *descriptor hierarchy* refers to a structured ordering used in the presentation of descriptor information. At the top of a descriptor hierarchy is the *unit/subunit identifier descriptor* that hold general information pertaining to the unit/subunit. The *unit/subunit identifier descriptor* also holds a list of *root list descriptors*. A root list descriptor is a parent descriptor to other *list descriptors* and *entry descriptors* at lower levels of the hierarchy. A device may implement more than one *root list descriptor*. An *entry descriptor* contains other *list descriptors,* and is addressable by its type, ID or position within a list. This hierarchy forms a structured way of presenting a device's descriptors (1394 Trade Association, 2001c).

Within each descriptor are data structures known as *information blocks* (*info blocks*) that hold specific information (1394 Trade Association, 2001c). The contents of the info blocks can be modified by a controller. Figure 3.21 depicts a descriptor and its associated info blocks as an interface presented by the target to the controller node.

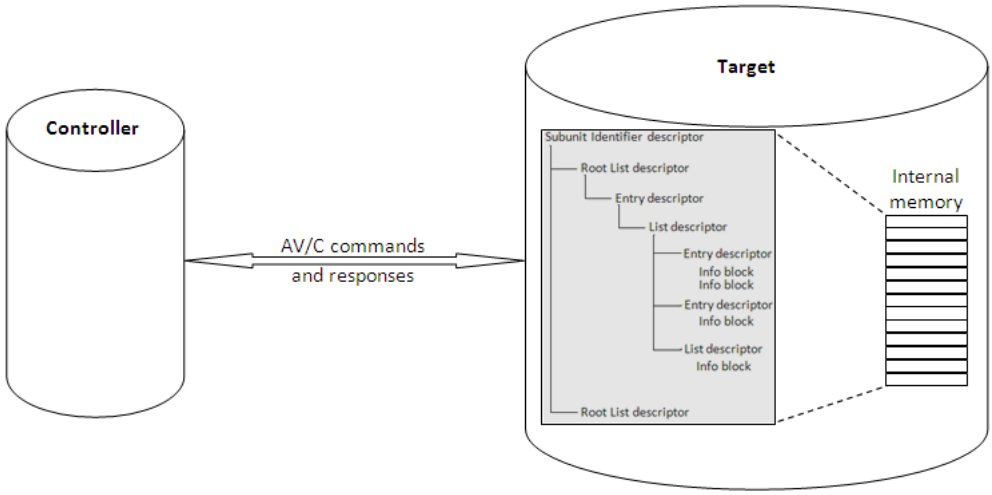


Figure 3.21: AV/C descriptor and Info block mechanism

In figure 3.21 the target node exposes its internal information through its *unit identifier descriptor*. While the actual storage of the device's attributes/information are device specific, the target node presents the information according to the format defined by the unit identifier descriptor (1394 Trade Association, 2001c).

AV/C provides a set of commands that a controller can use to read, write, create or delete descriptors and info blocks. These commands can be found in the *AV/C Descriptor Mechanism Specification* (1394 Trade Association, 2001c) and *AV/C Information Block Types Specification (1394 Trade Association, 2001a)*. In the *AV/C Descriptor Mechanism Specification* a set of commands are defined that allow for reading, writing, creating or deleting a descriptor and hence its associated info blocks. For instance to read an AV/C descriptor

- a controller sends an AV/C command to a target node indicating it intends to read from a specific descriptor.
- the target responds indicating whether or not the descriptor exists and the state of the descriptor, that is whether it can be read.

- if the descriptor exists and its state will allow for a read (in which case another operation has not locked it), the controller then sends another AV/C command to read the descriptor and then locks the descriptor.
- after reading the descriptor, the controller sends an AV/C command (to the target) indicating that it has finished reading and hence wishes to unlock the descriptor.

The use of descriptors and info blocks provide a comprehensive scheme for exploring AV/C devices. However it involves the parsing of large amounts of data received from a target node which increases the chances of misinterpretation. Instead of this approach, some control applications send AV/C status commands to obtain device information. This latter method is discussed in the following section.

3.3.3.2 AV/C Unit and Subunit Commands

The descriptor and info block mechanism requires the reading of entire descriptors to determine their capabilities and infer controls that reside within an AV/C device. A read of the entire descriptor within an AV/C device results in large amounts of data being returned by the target, and this requires the controller application to traverse back and forth in order to properly parse the information returned. A controller can avoid this by inquiring about the controls it seeks to command. To do this it sends a number of *AV/C status commands* to the device to inquire about the state of specific controls. If the specified control does not exist within the target node, a *not implemented* AV/C response is returned by the target node.

Some of the unit/subunit commands that are used by controller applications to obtain device specific information include:

- *Unit Info status command* - returns a functional description of the device's unit (*unit type)* and a *company ID*. The company ID is a unique identifier assigned to the manufacturer of the device by the IEEE Registration Authority [5] (1394 Trade Association, 2001b).
- *Plug Info status command* - returns the number of input/destination plugs and output/source plugs of the specified unit/subunit. When this command is addressed to a unit, the controller is required to specify the type of unit plug (asynchronous plug, isochronous plug or

[5] IEEE Registration Authority is an internationally recognized body that registers unique identifiers to manufacturers.

external plug). In response the target will return the corresponding number of input and output unit plugs. If the controller addresses this command to a subunit, the number of destination and source plugs of the specified subunit is returned (1394 Trade Association, 2001b).

- *Subunit Info status command* - returns the types of subunit and the number of each type of subunit that is implemented within the unit that this command is addressed to (1394 Trade Association, 2001b).

- *Signal Source status command* - returns the source of the signal to either a unit output plug or a subunit destination plug, depending on which is specified in the command. Possible signal sources are the unit input plugs and the subunit source plugs (1394 Trade Association, 2003).

Using these status commands and others specified in various AV/C subunit specifications, a controller is able to obtain information about a target node. The connections between the unit plugs and subunit plugs can be determined using the *signal source command* defined in the *AV/C Connection and Compatibility Management Specification* (1394 Trade Association, 2003). Specific subunit information can be obtained by sending subunit dependent status commands. An AV/C device is not required to implement all possible status info commands. Details about the nature of the unit info, subunit info and plug info commands can be obtained from the *AV/C Digital Interface Command Set General Specification* (1394 Trade Association, 2001b).

AV/C allows a manufacturer to define additional vendor dependent commands and even additional vendor dependent unit types. The following section discusses device exploration based on an enhancement to the general AV/C commands.

3.3.3.3 Enhancements to AV/C commands

While the use of the general AV/C commands described in section 3.3.3.2 on the previous page will enable a controller to determine the overall information pertaining to a device's plugs, they are not adequate for a controller to determine the internal connections of a device's subunit plugs. This shortcoming may require a controller to parse the subunit's descriptors and info blocks in order to obtain the internal connections within a subunit. Vendor dependent AV/C commands can be used to obtain such information. Two such enhancement to the AV/C commands are (BridgeCo, 2007):

- *Extended Plug Info command* - this AV/C command, when sent as a status command, allows for more specific plug information. This status command has been implemented to return the plug type, number of channels on a plug, information about the plug inputs, and plug outputs.

- *Extended Subunit Info command* - this AV/C command, when sent as a status command, returns information specific to an audio subunit. This status command has been implemented to return information about the various function blocks that exist within an audio subunit, as well as function block input and output plug information.

The above mentioned enhancements are not generally implemented in all AV/C devices. They are examples of an implementation by BridgeCo[6] to create a reduced and efficient set of AV/C commands by adding functionality to the general AV/C commands.

The following section describes two AV/C subunits (audio and music subunit) that are of interest to this research.

3.3.4 AV/C Subunits

As mentioned in section 3.3.2, an AV/C device is composed of logical entities known as units and one or more functional blocks called subunits. These subunits fulfill a specific role within the device. Figure 3.20 on page 61 shows the structure of a subunit as comprising some functional blocks, destination subunit plugs and source subunit plugs. While a destination plug inputs signal into the subunit, a source plug outputs signal from the subunit.

In this section two subunit types (audio subunit and music subunit) are described. The particular relevance of these two subunits to this study will become apparent in chapter 4.

3.3.4.1 Audio Subunit

Details of the audio subunit can be obtained from the *AV/C Audio Subunit Specification* (1394 Trade Association, 2000a). The AV/C audio subunit concerns itself with the processing and routing of audio signals. Within a device's audio subunit exists function blocks (1394 Trade Association, 2000a). Function blocks are audio signal processing blocks that fulfills a specific purpose. A function block possesses input and output plugs. A plug that routes a signal into

[6]BridgeCo is a producer of entertainment equipment platforms for electronic devices

a function block is known as a *function block input plug*, while a *function block output plug* routes a signal out of a function block. A device's audio subunit consists of four possible types of function blocks, namely (1394 Trade Association, 2000a):

- *Selector function block* - this function block allows only one of its input signals to be output through the function block output plug. Hence this function block is used to choose between more than one possible input signals.
- *Feature function block* - is used to adjust a device's audio control parameter. For instance, a device will require a feature function block in order to allow for adjusting the device's gain levels on its volume control.
- *Processing function block* - this function block manipulates input signals according to some algorithm and then outputs it from its output function block plug. It clusters input signals, manipulates them, and then channels the resultant signal out through its function block output plug. For instance a mix of all possible input signals (analog, digital or isochronous stream signals) can be clustered together in a *mixer* processing function block and the resultant signal output through the output function block plug.
- *CODEC function block* - this function block allows for specific signal CODEC manipulations. It is capable of decompressing non-linear audio signals such as the *MPEG-2 AAC (Advanced Audio Coding)* format.

Within an AV/C audio subunit, audio signals might be routed between its various function blocks. Figure 3.22 depicts an AV/C audio subunit that contains a selector function block, a feature function block and a mixer processing function block. The selector function block depicted is capable of outputting either of its two input signals. The feature function block is used to manipulate a particular feature (for example a volume control) and its output is routed to form one of the inputs of the selector function block. All three signals entering the processing function block through its input plugs are being clustered and sent to the selector function block.

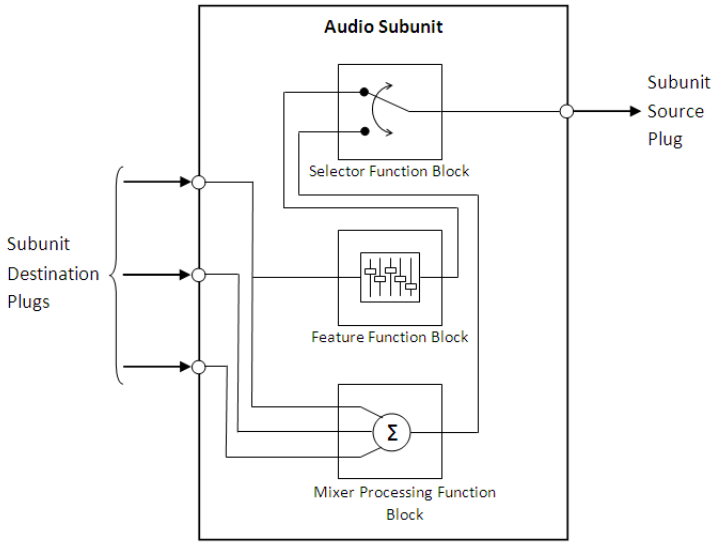


Figure 3.22: AV/C audio subunit

The *AV/C Audio Subunit Specification* (1394 Trade Association, 2000a) defines audio subunit specific AV/C commands that will allow a controller application to enquire about and control audio subunit plugs, function blocks, and function block plugs. It also provides a suite of commands that will allow for signal routing within the audio subunits.

3.3.4.2 Music Subunit

Within an AV/C device the audio subunit is capable of handling audio signals from external input plugs. However, the music subunit handles only firewire audio/MIDI signals. These firewire input signals can then be sent to the audio subunit for further manipulations or output through the unit's isochronous stream output plugs. The music subunit receives its input signals from a subunit source plug and/or from the unit input plug. The music subunit is detailed in the *AV/C Music Subunit Specification* (1394 Trade Association, 2005).

A music subunit destination plug gives a full description of the signal it receives (1394 Trade Association, 2005). This makes it possible for a control application to determine and manipulate the signals that enter a music subunit through its destination subunit plugs.

A music subunit as implemented in commercial devices might conform to either of the following two AV/C music subunit standards (1394 Trade Association, 2005).

- *AV/C Music Subunit version 1.0* - introduces the concepts of *input music plug*s and *output music plugs* within a music subunit. The *input music plugs* receive signals from the music subunit destination plugs. The *output music plugs* transmit signals to the music subunit source plugs. This is an earlier approach of implementing the music subunit. Figure 3.23 depicts this approach.

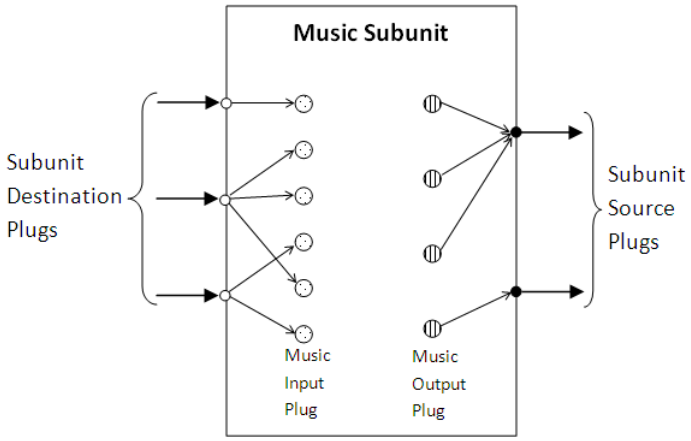


Figure 3.23: AV/C music subunit - legacy model

In this version of the music subunit, a *subunit destination plug* can be connected to multiple *music input plugs*, but a *music input plug* only receives signal from one *subunit destination plug*. In a similar manner, a *subunit source plug* can get its source signal from more than one *music output plug*, but a *music output plug* only outputs a single audio/MIDI signal.

- *AV/C Music Subunit version 1.1* - combines the concept of input music plug and output music plug into one plug known as a *music plug*. Each *music plug* possesses an *input end* and an *output end*. In this approach, data signals are clustered and then routed to a music plug for processing. This is a recent approach of implementing the music subunit. Figure 3.24 depicts this concept.

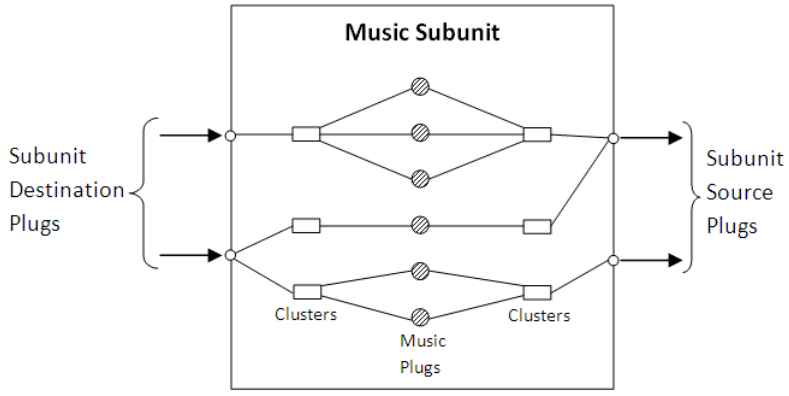


Figure 3.24: AV/C music subunit - modern model

The *music plugs* give a full description of the data signals and allows for the routing of these signals within the music subunit. In some devices this routing is fixed and can not be manipulated by a controller. Signals entering a music subunit are grouped together into clusters based on their data format, for example different AM824 data formats. Each audio signal in the cluster is controlled and described by one music plug. A music subunit can have a maximum of 65534 music plugs (1394 Trade Association, 2005). See the IEC specification titled *Digital Interface for Consumer Audio Equipment - Audio and Music data transmission*(1394 Trade Association, 2005) for details on the various AM824 data formats.

The *AV/C Music Subunit Specification*(1394 Trade Association, 2005) defines AV/C music subunit commands that can be used to obtain music plug information and manipulate internal routing within the music subunit. It also defines the nature of the music subunit descriptors.

3.4 Summary

In this chapter a high speed serial bus called firewire (IEEE 1394) was introduced. Firewire allows for asynchronous and isochronous transactions. An asynchronous transaction guarantees data delivery, while an isochronous transaction ensures consistent delivery of data. The IEC standard defines a Function Control Protocol (FCP) that is implemented as firewire asynchronous transactions. FCP involves an asynchronous write to a target's FCP_COMMAND register and a corresponding asynchronous write to the controller's FCP_RESPONSE register. This establishes a command and response mechanism that has been capitalized on by a command/transaction set known as Audio/Video Control (AV/C) protocol. AV/C devices can be controlled, and their

state determined, by sending AV/C commands that are encapsulated within asynchronous (FCP) firewire packets. In the AV/C discussions, it was indicated that every AV/C command triggers (as is the nature of FCP) an AV/C response from the target node.

The nature of AV/C devices was revealed as consisting of logical entities that are described as units. These units consist of subunits that perform specific processing within the unit. The nature of two types of subunits, the audio and music subunits was discussed. In this chapter, three methods of exploring an AV/C device to discover the nature of its units and subunits, as well as the its unit plugs and subunit plugs were described.

The next chapter describes an application that discovers AV/C devices on a network, determines the number of isochronous stream plugs (input and output plugs) on the device and controls the AV/C device.

Chapter 4

An Implementation of the AV/C Protocol

In the previous chapter, the AV/C protocol was described as a protocol that uses the FCP command and response mechanism to inquire about the state of a firewire device. AV/C was also described as being capable of modifying device parameters. Every AV/C device conforms to the IEC 61883-1 specification (*Digital Interface for Consumer Audio Equipment - General*) and implements FCP_COMMAND and FCP_RESPONSE registers. The FCP command and response mechanism describes a process that entails a controller node (node sending a request/command on a firewire bus) sending a command to the FCP_COMMAND register of a target node (node receiving/responding to a request/command on a firewire bus). On receiving a command, a corresponding response is triggered from the target node to the FCP_RESPONSE register of the controller. These commands and responses are implemented as firewire asynchronous write transactions.

As was mentioned in the previous chapter, the *unit_software_version* within the general config ROM of a firewire device can be used to determine whether the device conforms to the AV/C protocol. If a device implements AV/C the last bit of its *unit software version* is set to '1'. This implies that if the device only supports AV/C, its *unit software version* will have a value of 0x010001.

The concept of plugs as virtual end points of data streams was described in the previous chapter. It was mentioned that there are three types of unit plugs; isochronous stream plugs, asynchronous plugs and external plugs. A unit has been described as a logical entity that describes a particular functionality within an AV/C device. Within a unit there might exist several functional entities known as subunits that perform specific signal processing. These subunits possess destination subunit plugs that receive signals, and source subunit plugs that transmit signals.

In this chapter, the details of the implementation of an AV/C controller application called *AV/C Device Control Panel*, is described. The AV/C Device Control Panel will demonstrate the use of the AV/C protocol for device control, this being one of the protocols investigated in this research. The AV/C Device Control Panel was developed as part of this research and it performs firewire asynchronous reads and writes to:

- discover AV/C devices on a firewire network
- determine the number of isochronous stream input and output plugs
- determine and modify the channel set on an isochronous stream input or output plug
- force bus resets on the local firewire bus

The AV/C Device Control Panel (application) also executes various AV/C commands to:

- perform internal signal routing within an AV/C device
- control volume of audio signals outputted by an AV/C device
- select volume control type (line or headphone knob) for the headphone output on an AV/C device

This chapter starts by describing an AV/C compliant device known as the Phase 24 FW. It is a firewire audio breakout box with analog and digital audio input and output plugs, MIDI input and output plugs, and firewire input and output plugs. The Phase 24 FW is an AV/C compliant device that allows for asynchronous reads and writes to its FCP registers. Further on in this chapter the AV/C Device Control Panel application, which has been developed and tested on the Phase 24 FW, is described.

4.1 AV/C device - Phase 24 FW

The Phase 24 FW is an enhanced firewire breakout box manufactured by TerraTec (TerraTec, 2004). It allows audio and MIDI data that is streamed into it from its firewire plugs to be routed to its analog and digital output plugs. The Phase 24 FW also allows for internal signal processing such as mixing of audio signals and volume control. It utilizes a BridgeCo firewire chip, the

DM1000 (TerraTec, 2009) and implements connection management using BridgeCo's enhanced AV/C commands, described in section 3.3.3.3 on page 65.

(a) Front view (b) Rear view

Figure 4.1: TerraTec's Phase 24 FW

The Phase 24 FW has two firewire plugs which are visible on its rear. Also visible on the rear of the device are two monaural analog input plugs and two monaural analog output plugs to the right. There is an external connector that connects to the 9 pin plug and that provides the S/PDIF and MIDI 5 pin DIN connectors. On the front is a headphone output that outputs stereo audio signals.

Phase 24 FW is capable of routing signals from its analog or digital input plugs to its analog (including headphone stereo) and/or digital output plugs. It is also capable of routing audio from its firewire plugs to any of the analog, digital, or headphone output plugs. Routing from the analog and digital input plugs to the firewire plugs is also possible. The routing of signals are implemented internally within the device. The *Phase 24*, as it is commonly known, allows for volume control of all audio signals that it outputs. A volume control knob exists for the headphone (stereo) output that can be used to physically adjust the volume.

TerraTec ships a device driver and control panel application with each Phase 24. This driver can only be installed per device on a PC, hence allowing a PC to control only one Phase 24 per installation. At start up of TerraTec's own Phase 24 control panel, the transmission and reception channels of its firewire plugs are set automatically. The control panel does not allow a user to select the channels on either of the device's input or output isochronous stream plugs.

The next section gives a technical description of the internal nature of the Phase 24 in AV/C terms. This description was obtained by probing the device with various AV/C commands.

4.1.1 Exploring the Phase 24 FW

This section explores the Phase 24 by performing various firewire, and particularly AV/C, transactions on the device. The procedure used in the exploration of the Phase 24 entails determining its:

- number and type of units
- types of subunits, and the number of each subunit
- types of unit plugs, and the number of each type of unit plug
- number of subunit destination and source plugs for each type of subunit

The Phase 24 comprises an AV/C *unit*. In order to determine its unit type, an AV/C *unit info status command* was sent to the device (refer to Appendix B figure B.1). The response obtained from this status command described the Phase 24 as possessing an audio unit. The type and number of subunits that are implemented within the Phase 24 was obtained by sending a *subunit info status command* to the Phase 24 (refer to Appendix B figure B.2). The device responded by indicating that it had one audio subunit and one music subunit implemented.

Knowing the number and type of units and subunits within the device, the next step in exploring the Phase 24 was to determine the number and type of plugs that exist within it. Using the *plug info status command* addressed to the unit, the number of unit plugs of a specific type was determined (refer to Appendix B figure B.3). A *plug info status command* addressed to an AV/C unit takes as an argument the type of plug to query for. The possible types of plugs to query for are isochronous and external plugs, or asynchronous plugs. When the *plug info status command* is addressed to a subunit it will return the number of subunit source and destination plugs. Refer to Appendix B figures B.4 and B.5, for the layout of the *plug info status command* addressed to an AV/C audio subunit and music subunit, respectively.

Table 4.1 shows the results of *plug info status commands* addressed to the unit, audio subunit and music subunit of the Phase 24.

	Number of Input Plugs	Number of Output Plugs
Isochronous Unit Plugs	2	2
External Unit Plugs	3	4
	Number of Destination Plugs	Number of Source Plugs
Audio Subunit	4	5
Music Subunit	5	5

Table 4.1: Plug Info of the Phase 24 FW

Knowing the type of subunits implemented within the Phase 24 and the number of unit and subunit plugs implemented in the device, the next section describes how the internal signal routing within this enhanced breakout box was determined. Firstly, determination of the routing between the unit plugs and subunit plugs is described. This is followed by an analysis of the internal routing within each subunit, starting with the music subunit and then the audio subunit.

4.1.1.1 Phase 24 FW - Unit

Figure 4.2 shows the unit and its associated subunits as implemented in the Phase 24. Also visible are the unit and subunit plugs. A clear distinction is made between the unit's isochronous stream plugs and the unit's external plugs. The connections that indicate the signal flows between these plugs are also visible in the figure.

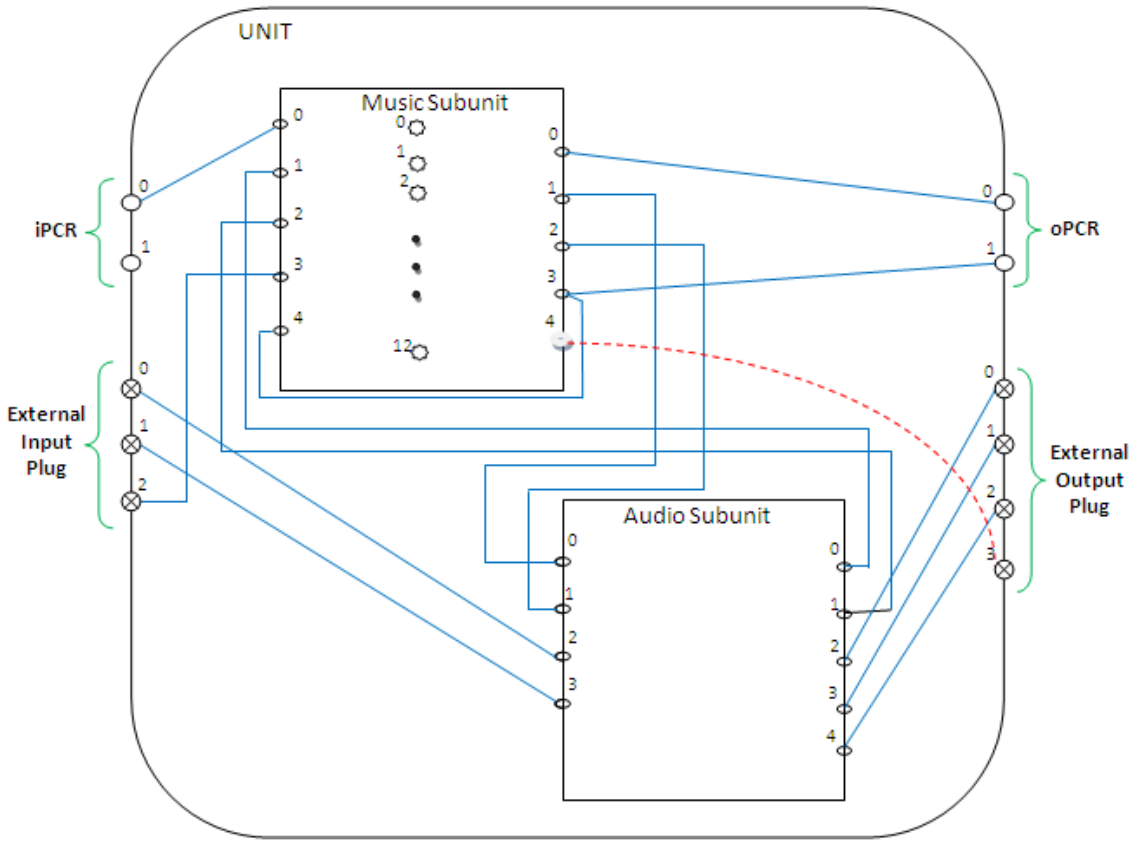


Figure 4.2: Signal routing in a Phase 24 FW

The isochronous input plug control registers (iPCRs) and isochronous output plug control registers (oPCRs), as shown in figure 4.2, corresponds to the isochronous input plugs and isochronous output plugs respectively. Asynchronous reads were performed on the input master plug register (iMPR) and the output master plug register (oMPR) (refer to Appedix A, figures A.1 and A.2) and the results confirmed the response obtained from the plug info status command (shown in table 4.1). These results verified that within the Phase 24 there exist two input isochronous stream plugs and two output isochronous stream plugs.

Each of the isochronous plugs transmits or receives on a particular channel. The channel on which an isochronous stream output plug transmits can be determined from the channel field of the oPCR that corresponds to that plug. Similarly the channel on which an isochronous stream input plug receives (isochronous streams) can be determined from the channel field of the iPCR that corresponds to that plug. Refer to figure 3.13 on page 50 for the layout of oPCR and iPCR.

Within the Phase 24 the unit plugs and subunit plugs are connected. These include connections

between unit input and subunit destination plugs, between subunit source and unit output plugs, and between subunit source and subunit destination plugs. These connections were obtained by sending a *signal source status command* to a specific unit/subunit plug. For instance, to determine the signal source to music subunit destination plug '0', a *signal source status command* was addressed to the music subunit, indicating that the subunit destination plug of interest has ID '0' (refer to Appendix B figure B.6). In response the device will return the unit input plug or subunit source plug that transmits signal to the specified destination plug. Refer to section 3.3.3.2 on page 64 for information about the AV/C *signal source status command.*

Another approach to obtain information about signal routes within a device is to parse the device's descriptors and info blocks. The descriptors and info blocks outline all the connections that exist within the Phase 24, and indicate other attributes about a plug's connection. For example they indicate whether or not a connection exists to that plug and whether the connections between plugs are fixed. The descriptor and info block mechanism is described in section 3.3.3.1 on page 62.

4.1.1.2 Phase 24 FW - Music Subunit

Figure 4.3 shows the music subunit as implemented in the Phase 24. The internal clusters, music plugs and stream positions are obtained from the music subunit descriptor (see section 3.3.4.2 on page 68 for information about the music subunit). The layout of the Phase 24's *music subunit status* and *music subunit identifier* descriptors are shown in Appendix C, figure C.2 and figure C.1, respectively.

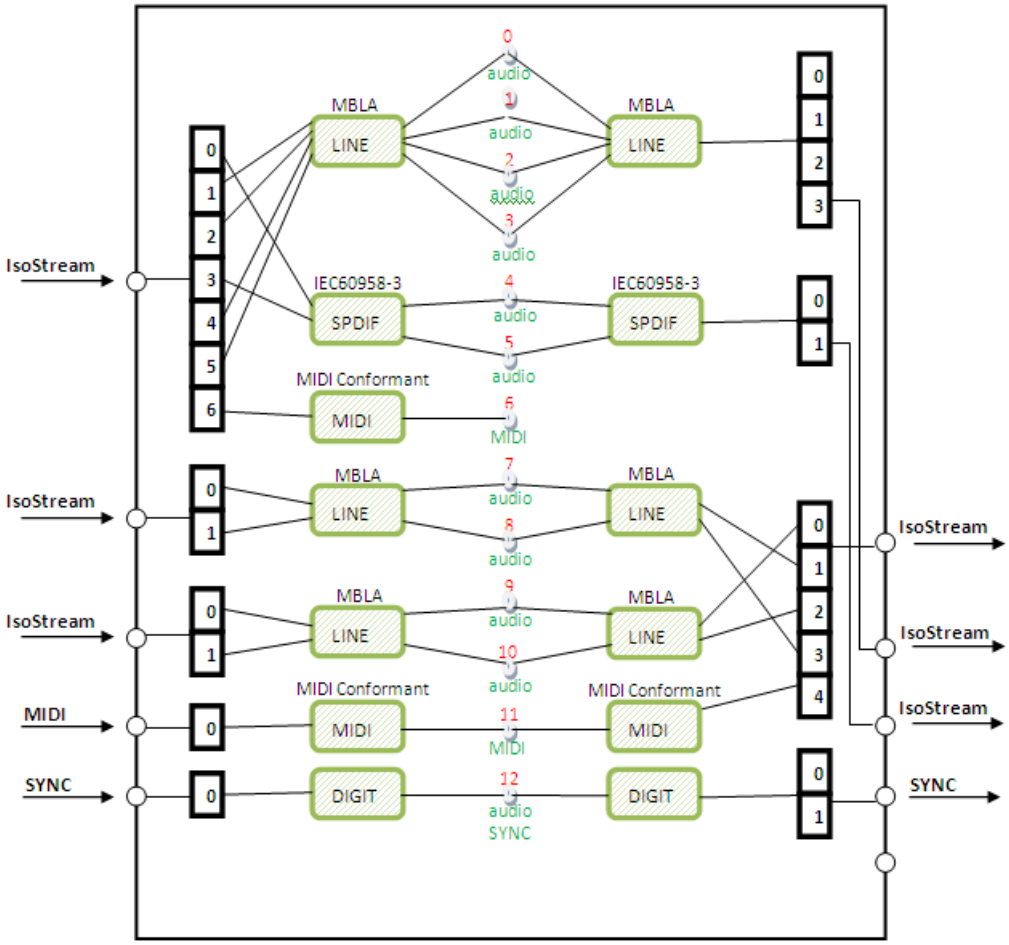


Figure 4.3: Music Subunit as implemented in Phase 24 FW

The music subunit within the Phase 24 has five destination and five source subunit plugs, each allowing a particular type of signal through. In figure 4.3 the top three destination subunit plugs receive isochronous audio stream inputs. The fourth (counting top-down) destination plug receives a MIDI input signal and the fifth destination plug is connected to an audio synchronization signal source. On the source subunit plug side, the top three source plugs output isochronous streams, while the fourth source plug outputs an audio synchronization signal.

An isochronous audio stream enters the music subunit through its destination subunit plug. This input stream may comprise a number of sequences each occupying a particular stream position. Refer to section 3.1.7 on page 43 for an explanation of the concepts of isochronous streams and sequences. In figure 4.3, the isochronous stream that enters through the first music subunit destination plug of the Phase 24 is shown as consisting of seven sequences. These sequences are

grouped into different input clusters on the basis of the signal's data format, in this case AM824 data format. Each cluster has a particular port type, such as LINE or S/PDIF, that specifies its data format. The signals are then routed to various music plugs. Each music plug routes a signal to an output cluster and then to a source music subunit plug.

A music plug completely describes the signals that it routes, and could allow for various signal manipulations and possible dynamic routing capability. However, the music subunit descriptor of the Phase 24, which was obtained in this investigation, indicates that the audio format and routing of the music plugs are fixed. Thus a controller is not allowed to modify routing by using the Phase 24's music plugs. Hence, when an *AV/C connect control command* is sent to the music subunit within a Phase 24, with the intention of making connections between music plugs, a *rejected* response is received from the device.

4.1.1.3 Phase 24 FW - Audio Subunit

A unit's external plugs are its non-firewire plugs, including the analog, digital and MIDI plugs. In the Phase 24, these external plugs (with their associated signals), except for the MIDI inputs, are routed to the audio subunit. The audio subunit within a Phase 24 has

- 8 feature function blocks responsible for volume controls
- 3 processing function blocks that allow for various mixing of signals
- 4 selector function blocks that allow for the selection of input signals

These results were obtained by parsing the *audio subunit descriptor* of the Phase 24. Refer to Appendix C, figure C.3, for the layout of the *audio subunit identifier* descriptor. Figure 4.4 on the next page depicts the audio subunit as it is implemented in the Phase 24.

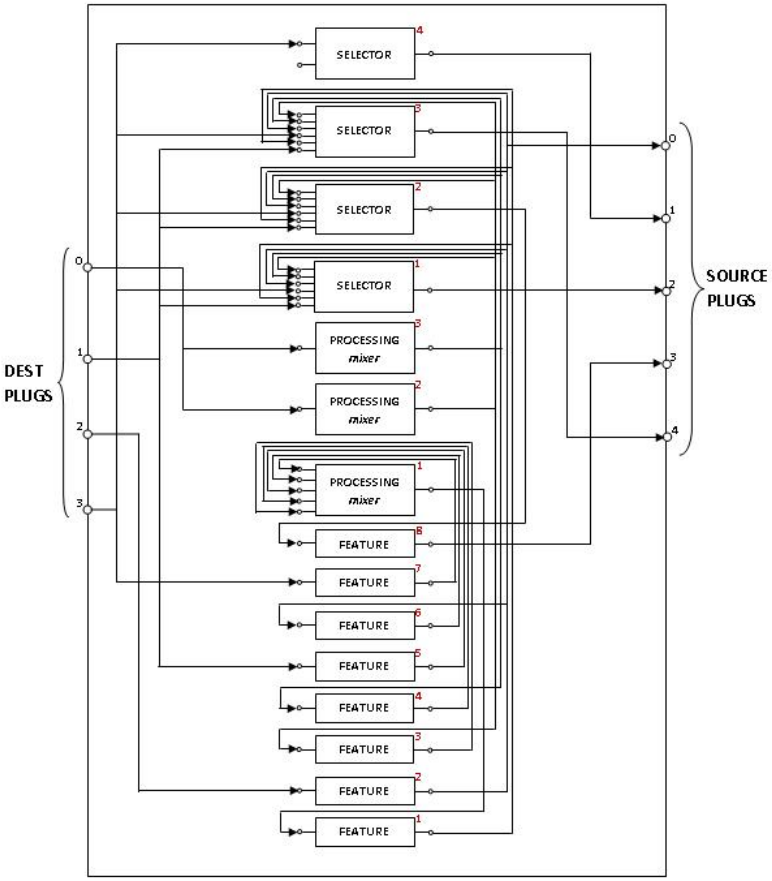


Figure 4.4: Audio subunit as implemented in Phase 24 FW

In figure 4.4, the selector function blocks labelled 1, 2 and 3 each have six function block input plugs. Selector function block labelled 4 has one function block input plug. Processing function blocks 2 and 3 have one input each, while processing function block 1 has five inputs plugs. Each of the eight feature function blocks have one input and provide volume and mute controls over any signals that they receive. This information was obtained by parsing the audio subunit's descriptors and info blocks.

The Phase 24's audio subunit plugs have fixed connections with the function block input plugs. The connection between the audio subunit source plugs and the function block output plugs of figure 4.4 are tabulated in table 4.2.

Source Plug (from figure 4.4)	Function Block (from figure 4.4)
0	Feature 2
1	Selector 4
2	Selector 1
3	Feature 8
4	Selector 3

Table 4.2: Phase 24 FW audio source plug to function block relationship

Notably, most of the output signals of the source plugs are obtained from selector function blocks, except for source plugs 0 and 3 which come from feature function block 2 and 8 respectively. When the inputs of feature function blocks 2 and 8 are traced (refer to figure 4.4 and figure 4.2), they are seen to come from destination subunit plug 2 and selector function block 2 respectively. The audio subunit's descriptor indicates that feature function blocks 2 and 8 implement only the volume control feature.

With an understanding of the subunits and function blocks that reside within the Phase 24, an AV/C Device Control Panel was implemented that allows for the control of the Phase 24. The control panel identifies any AV/C device on the firewire network. This has the advantage of overcoming the "installation per device" issue associated with the original TerraTec control panel (which uses a *device driver*), since the AV/C Device Control Panel uses a firewire *bus driver*.

The AV/C Device Control Panel developed in this research allows a user to determine the number of isochronous stream plugs on any AV/C device, and allows a user to allocate channels to any of the determined isochronous stream plugs. This control panel application also allows for volume level adjustment on the Phase 24 as well as for audio signal routing.

4.2 AV/C Device Control Panel

The AV/C Device Control Panel is a windows application that utilizes a number of AV/C commands. Its user interface is designed on the *JUCE* platform, using the *JUCER* toolkit. JUCE (Jules' Utility Class Extension) is a C++ library that allows for cross-platform graphical user interface (GUI) creation. JUCER, which is shipped with the JUCE package, facilitates quick creation of components using a visual GUI interface. All terms used to describe the AV/C Device Control Panel GUI features are as defined by the JUCE library (Raw Materials Software, 2005).

To interact with firewire devices a bus driver, *raw1394*, and its associated application program interface (API) is used. This API allows for generic firewire operations such as bus reset handling, asynchronous reads and writes, and extracting topology information of the firewire network. The *raw1394* bus driver allows access to any device on the bus. The next section describes the interaction between the AV/C Control Panel application and the *raw1394* bus driver.

4.2.1 AV/C Device Control Panel interaction with firewire drivers

In AV/C a controller node sends an AV/C command to a target node. This AV/C command is fulfilled by performing an asynchronous write transaction on the target node's FCP_COMMAND register. In response, the target node performs an asynchronous write to the FCP_RESPONSE register of the controller node. A *raw1394* API has been created that exposes a set of methods that are sufficient to fulfill these asynchronous transactions via the *raw1394* driver. See *'A Comparative Study of the Linux and Windows Device Driver Architectures with a focus on IEEE1394 (high speed serial bus) drivers'* (Tsegaye, 2002) for details on the *raw1394* API.

Applications such as the AV/C Device Control Panel use the API methods to perform firewire driver calls that are hidden to the application. Figure 4.5 depicts the interaction between the AV/C control panel application, the *raw1394* client driver through its API, and the windows kernel space *1394bus* driver.

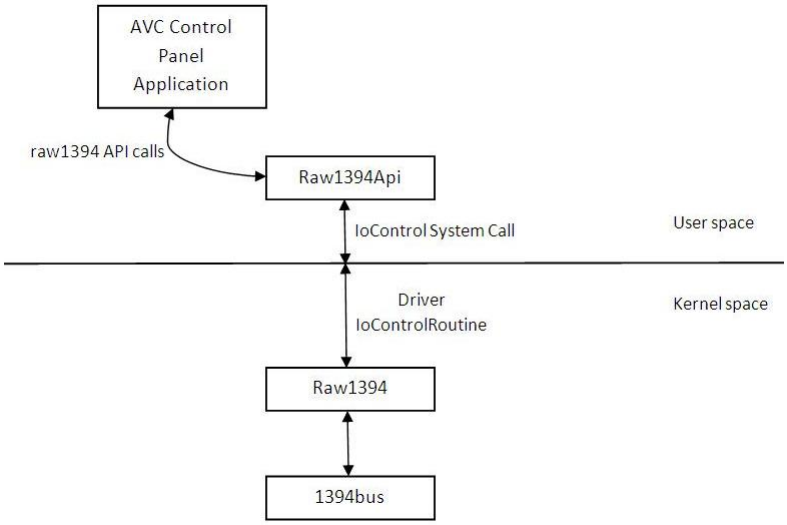


Figure 4.5: AV/C control panel (application) interaction with *1394bus* driver via *raw1394* API adapted from Tsegaye (2002)

In figure 4.5 on the preceding page, the interaction between an application using the API and the actual operating system firewire drivers is divided into two parts. The upper part, being the user-space, comprises the application and the *raw1394* driver API. The lower part is the kernel space, and it includes kernel modules used by the kernel to fulfill the client driver's requests. The kernel space includes the *raw1394* driver that interacts with the *1394bus* driver. The API hides the various *IO Control* interactions from the application, but presents a simple interface to the application.

To use the API, the AV/C application is required to obtain a handle to the *raw1394* driver by calling the *raw1394_Open* method. This handle gives the application access to all the other *raw1394* API methods. The *raw1394_close* method is used to perform a clean up of memory allocated to the *raw1394* handler and is called during the application shut down.

The FCP command and response mechanism is implemented by registering the device's FCP_ RESPONSE register. The API's *raw1394_register_address_range* is used to register an address range. This method causes a callback to be triggered whenever a write is made to the registered FCP_RESPONSE address. The *raw1394* driver passes to this callback the data that was written to the registered address. The application picks up the data whenever the callback is triggered. The registered address range is de-registered after use by calling the *raw1394_free_address_range* method. This unlocks the registered address so that it is available to other applications/processes that might require it.

All AV/C commands are fulfilled by the API's *raw1394_async_write* method. It uses this 'write' method to perform an asynchronous write to the target node's FCP_COMMAND register. The application ensures that it properly packages the data sent in the asynchronous write method in such a way that it conforms to the intended AV/C command. To obtain information about isochronous stream plugs and the channels set on these plugs the application uses the *raw1394_ async_read* provided by the *raw1394* API.

The API also provides a *raw1394_async_lock* method that allows for lock transactions. For instance, when the AV/C Device Control Panel application seeks to obtain a transmission channel, it performs a lock transaction on the IRM's CHANNEL_AVAILABLE register. A similar lock transaction to the BANDWIDTH_AVAILABLE register of the IRM is used to obtain transmission bandwidth. See section 3.1.5 on page 38 for further information about channel and bandwidth allocations.

With an understanding of the interactions between the control panel application and the *raw1394* driver, the next section describes the features of the AV/C Control Panel.

4.2.2 AV/C Device Control Panel Application

The AV/C Device Control Panel is an application that performs device control on the Phase 24 by using various AV/C commands. It has been implemented under *Windows (XP* and *Vista)* and uses the *raw1394* API (described earlier in section 4.2.1) for firewire device access. Figure 4.6 shows the AV/C Device Control Panel application.

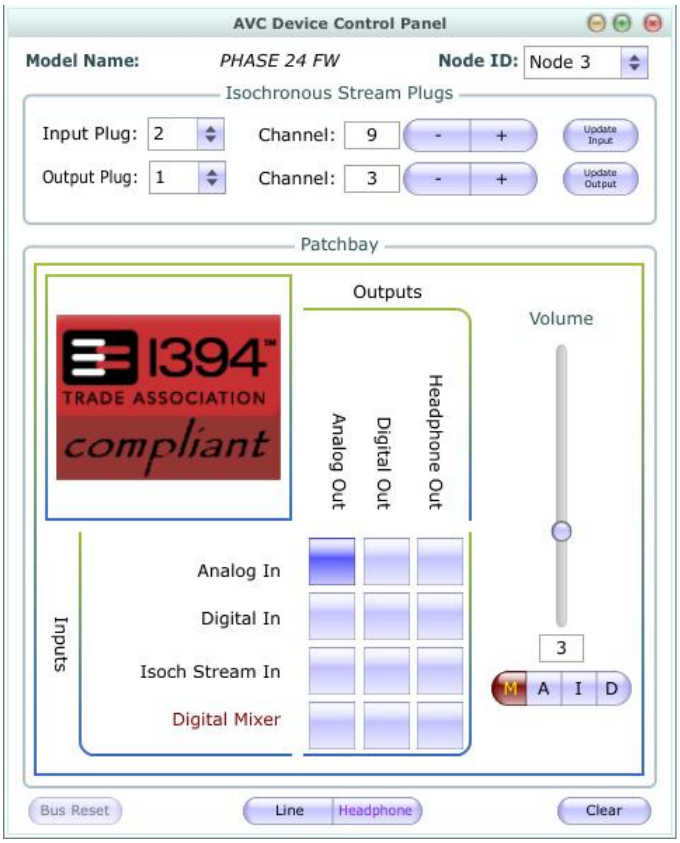


Figure 4.6: AV/C Device Control Panel

The layout of this control panel comprises a:

- device discovery area at the top
- plug information and channel modification area
- patchbay area for signal routing
- general controls and user interface management area at the bottom.

The following subsections explain the features of this application based on the above functional layouts.

4.2.2.1 Device discovery area

The device discovery area comprises a Model Name label and Node ID combo box on the GUI. Device discovery is the first task performed by the application. At start up, the application obtains the number of devices on the firewire network. For each device, the application reads its *unit software version* (within the device's config ROM) in order to determine whether it is an AV/C device. Refer to section 3.2.1 on page 48 for information on *unit software version*. Every AV/C device discovered is added to an AV/C device node list. Each AV/C device in the AV/C node list is identified by its node ID and its globally unique identifier (GUID). A GUID is a 64-bit address obtained from the bus info block of the device's config ROM.

The discovered devices are displayed in the application according to their unique node ID. This is visible in the combo box on the top right labeled *'Node ID'*. In the absence of any AV/C node the AV/C device node list is empty. Node IDs of networked devices on a firewire bus are numbered from node '0'.

On selecting a node from the node ID list, an asynchronous read of the node's config ROM is triggered. The 1394TA document titled *Configuration ROM for AV/C Devices 1.0* (1394 Trade Association, 2000b) specifies that an AV/C device conforms to the general config ROM format (described in section 3.1.3 on page 35). Hence, from the config ROM the *'Model Name*', which is stored as ASCII[1] characters, is obtained and displayed on the application. This textual description (model name) gives the user an indication of the device that corresponds to a particular node ID. The model name may not be sufficient information to identify a device, for instance when more than one device of the same type and hence same model name present on the network. One way to resolve this would be to display the GUID of the device. However, since a user can not tell what GUID corresponds to a particular physical device by simply looking at the devices on the network, this approach might not be beneficial. The *general control area* of this application (described in section (4.2.2.4)) presents a technique that can assist in identifying a particular Phase 24.

Figure 4.7 shows the *device discovery* area of the AV/C Device Control Panel. In the figure, the device with node ID of '1' is selected and the model name when retrieved from the device is

[1] American Standard Code for Information Interchange (ASCII) encodes the English alphabets using characters

shown to be *PreSonus FIREPOD*. This application has not been designed to incorporate internal routing features of the PreSonus FIREPOD, for reasons given in subsection 4.2.2.3 on page 89. However, it (AV/C Device Control Panel) is capable of identifying the FIREPOD as an AV/C device, determining the number of isochronous (input and output) stream plugs on the FIREPOD and modifying the channels on the FIREPOD's isochronous stream plugs.

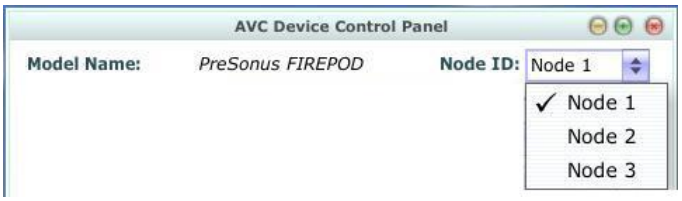


Figure 4.7: AV/C device list on the AV/C Device Control Panel

Note that this node is different from that selected in figure 4.6. Also nodes 1-3 are the only nodes visible on the node ID list. This is an indication that node 0 is not an AV/C device, since firewire node count starts from '0'.

4.2.2.2 Plug Information and Channel modification area

This is the area indicated as *'Isochronous Stream Plugs'* on the control panel application. It comprises an *'Input Plug'* combo box, *'Output Plug'* combo box, *'Channel'* sliders (with increment and decrement button) for input and output plugs, *'Update Input*' text button and *'Update Output'* text button.

When an AV/C device is selected from the *Node ID* combo box, the application sends an asynchronous read packet to the input master plug register (iMPR) to determine the number of isochronous (stream) input plugs. This follows the read of the device's config ROM performed in the previous section. In the same manner an asynchronous read packet is performed on the output master plug register (oMPR) to determine the number of isochronous output plugs (see section 3.2.2 on page 49 for information on iMPR and oMPR). The *'Input Plug*' combo box displays a list of the available isochronous stream input plugs on a device with plug IDs starting from '1'. Similarly the *'Output Plug*' combo box displays a list of the available isochronous stream output plugs on a device with plug IDs starting from '1'.

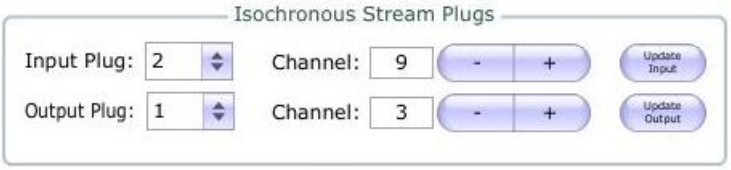


Figure 4.8: AV/C Device Control Panel's isochronous stream plug area

When an input or output plug is selected from the corresponding input or output combo box, an asynchronous read is executed to the corresponding plug control register (PCR) of the selected plug. Of interest is the *channel* field that specifies the current channel that is set on the selected plug. The retrieved channel is displayed in the channel (increment/decrement) slider. This value field can be changed by clicking on the increase (+) or decrease (-) button. However, this will not result in any actual change of the channel set on the plug until the corresponding update button is clicked. To modify a channel on a particular isochronous stream input plug click the '*Update Input*' button. The same pertains to the output channel on a selected output plug when the '*Update Output*' button is clicked.

When the appropriate modification button is clicked (Update Input or Update Output) to change an isochronous stream plug's channel, the following sequence is executed in order.

- The application requests for the specified channel from the isochronous resource manager (IRM) by performing a lock on the CHANNEL_AVAILABLE register of the IRM. If this channel has already been allocated, a pop-up message indicates to the user that the update has failed . See pop up window in figure 4.9.
- The application then requests sufficient bandwidth from the IRM by performing an asynchronous lock transaction on the BANDWIDTH_AVAILABLE register of the IRM. If this lock transaction fails, the user is presented with a 'pop-up' window that indicates the update has failed. See pop up window in figure 4.9.
- If the above steps succeed, the application breaks all connections on the selected plug. This includes broadcast and point-to-point connections on the selected plug. This is achieved by clearing the *broadcast counter* field and *point-to-point counter* fields in the PCR that corresponds to the selected plug.
- Next the channel is set on the selected plug by executing an asynchronous write to the PCR corresponding to that plug. This asynchronous write will set the *channel* field and

increment the *point-to-point counter* field on the PCR. If this write is successful, a window pops up indicating that the update was successful. See pop up window in figure 4.9.

(a) Update failed (b) Update successful

Figure 4.9: Update button status pop-up windows

There can be only one channel set on a plug. This implies that whenever the channel value changes and the update button is clicked, the application will go through the above steps.

However, in order to prevent a situation in which the available network resources (channels and bandwidth) become exhausted, each node is required to de-allocate previously allocated resources, before requesting for new transmission resources. For instance, when a node wishes to change the transmission channel on an input plug, it will firstly perform an asynchronous write transaction to release the channel and bandwidth it was previously allocated, before requesting for a new channel and adequate bandwidth from the IRM.

4.2.2.3 Patchbay area

This area in the AV/C Device Control Panel comprises a routing matrix shown in figure 4.10. The signal sources on the left of the matrix are indicated as *inputs* and signal destinations on the top of the matrix are indicated as *outputs*. While this routing matrix model can be adapted for most AV/C devices, in its present state this routing matrix is specific to the Phase 24. To incorparate any other AV/C device, for example PreSonus FirePod mentioned in subsection 4.2.2.1 on page 86, would require that the nature of the internal routing within the AV/C device is obtained. This will require that the entire unit and subunits descriptors and info blocks within each AV/C device is parsed, in order to determine such routing.

In its current implementation, the AV/C Device Control Panel application verifies the model name of the selected node, and if it is a Phase 24 FW it then enables the routing matrix.

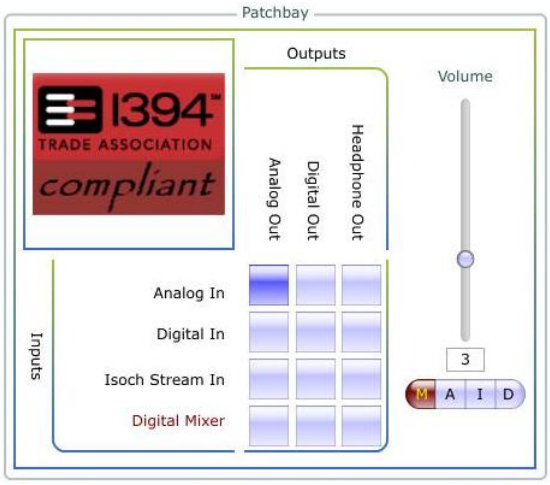


Figure 4.10: AV/C Device Control Panel Patchbay area

The analog, digital and isochronous stream inputs refer to physical inputs on the Phase 24. Similarly, the analog, digital and headphone output represent physical plug outputs on the Phase 24. The 'Digital Mixer' on the input side (left side of the matrix) allows for a mixed output of all the input signals to the specified output.

The cross-points of inputs and outputs are toggle buttons. When any cross-point is clicked, the corresponding input signal is routed to the output plug. All cross-points cause a *selector function block control command* to be executed. Refer to Appendix B figure B.7 for the layout of the *selector function block control command*. These particular selector function block control commands will instruct a selector function block, within the selected nodes audio subunit, to output from a particular input function block plug.

At any instant, only one connection can be made between an input and output signal flow. However an input plug can be routed to all three output plugs. To allow for a mix of all active inputs through an output plug, the 'Digital Mixer' input should be selected.

Also visible in the patchbay area is a volume adjustment slider with four volume control buttons below it. These buttons are indicated as 'M' for master volume control, 'A' for analog input control, 'I' for isochronous input volume control and 'D' for digital input volume control. In figure 4.10 the master volume control has been pressed and as a result all volume adjustments are for the master control. A master control will adjust the volume of the analog, digital and isochronous inputs collectively.

Adjusting the volume slider causes the slider value to change, each time sending a *feature function block control command* to the Phase 24's audio subunit feature function block. Refer to

Appendix B figure B.8 for the layout of the *feature function block control command*. This volume adjustment has been designed to be Phase 24 specific, as a specific feature function block is targeted by the command using its function block ID. This feature function block ID may not exist in other AV/C devices or could be used to adjust some other control such as bass or treble control.

The implementation of the selector function block, and feature function block commands uses the *raw1394* API. These commands are executed by sending an AV/C control command packet to the Phase 24 using the *raw1394_async_write* method. In the AV/C control command packet, the targeted node's function block type and its function block ID are indicated.

4.2.2.4 General control area

This is the bottom area of the control panel. It comprises a *Bus Reset* button, a *Line* button, a *Headphone* button and a *Clear* text button.

Figure 4.11: General control area

When the *'Bus Reset'* text button is clicked, a firewire bus reset is triggered. This is accomplished by a call to the *raw1394* API's *raw1394_generate_bus reset* method. A bus reset callback is triggered after the bus reset. As implemented in this application, the bus reset callback will re-enumerate devices on the firewire network to determine the AV/C nodes on the serial bus and then populate the AV/C node list.

The middle of this area holds two connected text buttons. These buttons are used to enable a volume control type feature for the headphone output plug on the front side of the Phase 24 (see figure 4.1). When the *'Line'* text button is clicked, a line signal output is channeled through the headphone output plug. This line output indicates that the gain level of the volume is *'sent out'* of the Phase 24 without internal adjustment by the Phase 24. This allows the device that inputs the audio signal to have absolute control of the volume adjustment. When the *'Headphone'* text button is clicked, the volume is adjusted by rolling the knob on the front left corner of the Phase 24. When one of these two buttons ('Line' or 'Headphone') is clicked, a *feature function block control command* is sent to the Phase 24's audio subunit's feature function block (feature function block with ID '8' in figure 3.20 on page 61).

Whenever the *'Headphone'* text button is pressed, an LED (Light-emitting diode) is lit on the front left (view) of the Phase 24 (refer to figure 4.1 on page 74), that indicates that the volume adjustment knob is active. Hence the headphone output (stereo output) volume control can be adjusted using the volume adjustment knob at the front left corner of the Phase 24. When the *'Line'* text button is clicked, the LED goes off, which is an indication that the adjustment knob is inactive. This can be used to identify a Phase 24 by toggling between the *'Headphone'* and *'Line'* text buttons, the LED goes ON and OFF. This makes it easy to physically distinguish between one Phase 24 and another.

When the *'Clear'* button is clicked, all selected fields on the control panel application are cleared, and enabled cross-point buttons (in the patchbay) are disabled.

4.2.3 Summary

This chapter demonstrates a practical implementation of the AV/C protocol. Firstly an enhanced breakout box which implements the AV/C protocol was introduced. The Phase 24 FW manufactured by TerraTec is explored, carefully identifying the unit and various subunits that comprise the device.

An exploration of the music subunit gives a detailed description of how the firewire (audio) isochronous streams, MIDI signals and audio SYNC signals are routed. This is followed by an exploration of the audio subunit and its function blocks. The internal routing within the audio subunit is also determined.

Having understood the internal structure of the unit, subunits, the unit plugs and subunit plugs, an AV/C Device Control Panel that allows for AV/C device discovery, isochronous stream plug discovery and channel modifications was discussed. The other features of the control panel application include signal routing and volume control.

The implementation of the AV/C Device Control Panel demonstrates the ability for an application to control a professional AV/C audio device (Phase 24 FW) on a firewire network. In particular, this AV/C control capability demonstrated the feasibility of creating a proxy for AV/C devices. Such a proxy will receive instructions and fulfill them by sending AV/C commands to the AV/C device.

Chapter 5

IP over firewire and XFN

In chapter 3 firewire was described as a high speed serial bus that conforms to the IEEE 1394 standard. The IEEE 1394 standard establishes a protocol for asynchronous and isochronous data transfers. Chapter 4 demonstrated how firewire has been used by the Function Control Protocol (defined in the IEC 61883-1 specification) for device control. The IEEE 1394 serial bus has found popularity in the networking of audio and video devices where its high speeds and deterministic data transfer are important. Firewire has also been used in the networking of computers to enable data (file) transfers at high speeds. In this chapter another device and computer networking protocol, is introduced. This protocol is the Internet Protocol version 4 (IPv4) that is defined in the RFC 791 document (Postel, 1981a). The Internet Protocol (IP) is a popular computer networking protocol that enables the transfer of data over a packet-switched network irrespective of the physical transmission medium.

IP provides an addressing scheme that allows for routing of packets from a source device to a destination device. This addressing scheme, when used on a firewire network, is known as *'IP over 1394'*. The IP over 1394 concept is defined in the RFC 2734 document (Johansson, 1999). Firewire devices that implement the IP protocol are able to transfer User Datagram Protocol (UDP) datagrams and Transmission Control Protocol (TCP) packets on a firewire network. The User Datagram Protocol is a connectionless transport layer protocol used to transfer relatively small amounts of data in what is referred to as *datagrams* and is defined in the RFC 768 document (Postel, 1980). The Transmission Control Protocol is a transport protocol that guarantees the delivery of packets and is defined in the RFC 793 document (Postel, 1981b). A connection-oriented protocol such as TCP establishes a communication path between a source and destination device before it starts data transfer. In a connectionless protocol such as UDP,

the source device transmits data addressed to a destination device without determining that the destination device is ready to receive the data (Olifer and Olifer, 2006).

UDP is a faster data transfer protocol when compared with TCP since it does not perform error checking on transferred datagrams. It (UDP) has been used in the establishment of a command and control protocol known as XFN. XFN is an application layer protocol for peer-to-peer networked audio/video device control. Devices that implement the XFN protocol incorporate an XFN stack which handles all XFN messaging. In XFN, devices are functionally defined in terms of their parameters in a 7-level hierarchy (Foss, 2008). The various concepts necessary for an understanding of the XFN protocol are described in this chapter. These include the:

- XFN stack
- XFN application node
- XFN 7-level parameter hierarchy, and
- layout of an XFN packet

XFN devices communicate by exchanging XFN messages. These messages are picked up and processed by the XFN stack. All communication with an XFN device is handled by the XFN stack. To enable a control application to implement XFN and hence communicate with an XFN device, an XFN application programming interface (API) has been developed (Klinkradt, 2007).

An XFN Level Explorer application which utilizes the XFN API was developed in the course of this research to detect XFN nodes on a network and explore their XFN parameters. This level explorer application uses various XFN API functions to obtain information from an XFN device.

In the following section, the RFC 1122 document (Braden, 1989) known as *Requirements for Internet Hosts – Communication Layers* defines an *Internet Protocol Suite* that forms the basis for the discussion on XFN as an application layer protocol.

5.1 The Internet Protocol Suite

The Internet Protocol (IP) is a protocol for communication between devices on a packet-switched network (Postel, 1981a). The Internet Protocol was initially implemented for computer networking, but has now been used in the networking of various devices to provide services such as Voice

over IP (VoIP) used in telephony, radio broadcasts, IP-TV and audio/video device networks. IP is concerned with the routing of packets from a source device to a destination device and forms the basis upon which other higher level network protocols are built.

On a packet-switched network of devices the transfer of IP packets entails collaboration between various network protocols. These network protocols have been categorized into four layers in an architectural framework known as the *Internet Protocol Suite* (Braden, 1989). The network protocols belonging to the internet protocol suite perform various functions such as device addressing and routing of data from a source device to a destination device. Some of the network protocols ensure quality of service (QoS) for transmitted data. Figure 5.1 shows the four layers of the internet protocol suite with examples of protocols that belong to each layer.

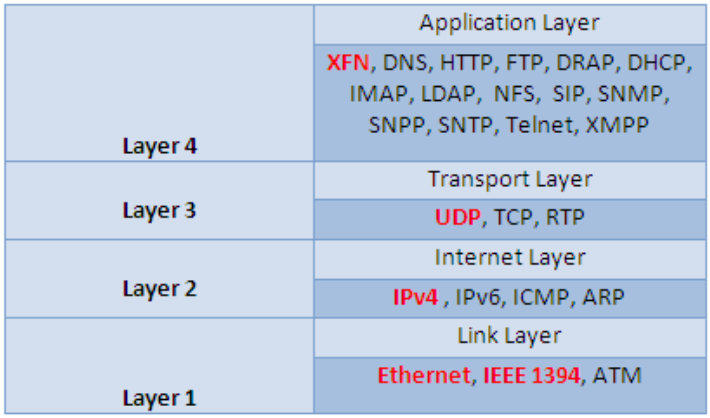


Figure 5.1: Internet protocol suite

With regard to the above architecture, XFN is situated at layer 4 as an application layer protocol and is built on other protocols at the lower layers. At layer 1 of the internet protocol model is the IEEE 1394 standard for high speed serial bus communications, and Ethernet which incorporates the IEEE 802.3 standards for local area networks (LANs) [1]. These network standards (Ethernet and firewire) provide the link layer protocol for XFN. At layer 2 is the Internet Protocol version 4 (IPv4), which forms the internet layer protocol for XFN. XFN utilizes the user datagram protocol (UDP) as its transport layer protocol at layer 3 of the above architecture.

The transmission of data using the XFN protocol requires that packet headers[2] and in some cases

[1]Local Area Network (LAN) refers to an interconnection of computers and devices over a relatively small area. This could be within a building or small housing complex.

[2]Headers are fields attached to the front of a data packet to provide protocol specific information.

tailers[3] of the lower level protocols, and of XFN itself, are added to the data. This implies that data transmitted by an XFN device will be encapsulated in the:

- XFN packet (layer 4)
- UDP datagram (layer 3), and
- IPv4 packet (layer 2)

before being transmitted over Ethernet or firewire. At the receiving XFN device, these protocols will also be implemented, so that as data traverses from layer 1 to layer 4, the various protocol headers and associated tailers are stripped off by the appropriate protocol.

The following sections describe the various network protocols utilized by XFN. The discussion is structured according to the internet protocol suite model. It starts by describing the layer 1 protocol used by XFN, to layer 4 where XFN is described as a protocol for audio/video device control.

5.1.1 Link Layer

This layer comprises the network interface protocols such as firewire and Ethernet. Firewire has been described in section 3.1 on page 31 as a standard for the transmission of data on a high-speed serial bus. Ethernet incorporates a host of IEEE 803.2 standards for link layer protocols used in computer networks. The link layer protocols handles the framing of packets or datagrams for onward transmission onto a network (Comer, 2006).

While firewire nodes are uniquely identified by their GUID (see section 3.1.3 on page 35), Ethernet nodes are uniquely identified by their media access control (MAC) addresses. A device's MAC address is a 48-bit number that is 'hard-coded' into the Ethernet card by its manufacturer (Washburn and Evans, 1996). The current implementation of XFN runs on Ethernet and firewire networks.

[3]Depending on the protocol, it might be necessary to attach some check bits (typically cyclic redundancy check bits) at the end of the data. These are referred to as tailers.

5.1.2 Internet Layer

The internet layer uses the Internet Protocol (IP), which is a protocol for data transmission over a packet-switched communications network (Postel, 1981a). IP allows data to be routed across a network from a source device to a destination device, their addresses being indicated in the packet header.

There are two IP standards, IPv4 and IPv6. The more widespread IPv4 defines a 32-bit addressing scheme for devices on a network. This addressing scheme is popularly represented in a *'dotted decimal'* format because of its readability. For instance, a 32-bit IP address *'10011101 11100111 10010010 00010010'* is represented as '157.231.146.18' in dotted decimal format. IPv6 addresses devices with a 128-bit addressing scheme thus allowing more devices to be addressed (Deering and Hinden, 1998).

Another function of the IP protocol is the fragmentation and reassembly of data packets. In a packet-switched network, packet fragmentation is used when the data to be transmitted is larger than the maximum transmittable data size in one packet of the transport protocol. IP defines a scheme for a source device to divide such large data into fragments that are reassembled at the destination device (Postel, 1981a). The division of data into smaller bits of data is known as fragmentation and is performed by the source device on an IP network. The gathering together of these data bits to form the original large chunk of data is referred to as data reassembly and is performed by the destination device on an IP network. The current implementation of XFN uses IPv4 as its internet layer protocol.

5.1.3 Transport Layer

The protocols in this layer interact with the application layer protocols (above it) and the internet layer protocols (below it) (seen in figure 5.1). The protocols that fall into this category (transport layer) are implemented in software on a network node (Olifer and Olifer, 2006). They provide end-to-end communication from one application program to another (Comer, 2006).

Olifer (2006) highlights the possible functions of the transporter layer protocols as:

- capability of restoring broken connections
- allowing multiple applications requesting to use the same transport layer protocol
- transmission error detection and correction.

However, the choice of transport layer protocol depends on the extent to which the application layer (at the top), and internet layer and link layer (at the bottom) provide reliable data delivery (Olifer and Olifer, 2006). Two of the most popular protocols in this category are the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP).

UDP provides an unreliable connectionless packet delivery service for applications to communicate with. It is the responsibility of the application program using UDP to handle problems associated with UDP, such as packet loss, data duplication and data delay, and out-of-order arrival of packets (Comer, 2006). The UDP protocol is defined in the RFC 768 document (Postel, 1980).

TCP provides reliable end-to-end data delivery irrespective of the reliability of the internet layer protocol. It ensures that a connection exists between a *socket* on the transmitting machine and a *socket* on the receiving machine. A socket is a virtual end-point of data streams on an IP network. A TCP connection is full-duplex in nature, which implies that data can flow in both directions between a transmitter and a receiver (Tanenbaum, 2003). TCP is defined in the RFC 793 document (Postel, 1981b).

XFN uses UDP for the fast transfer of XFN messages. Since UDP does not guarantee data delivery, XFN is capable of ensuring data delivery by assigning a sequence identifier to each XFN message.

5.1.4 Application Layer

An application layer protocol such as XFN relies on the lower level protocols for data transmission. XFN messages are encapsulated within UDP/IP packets thus taking advantage of the fast data delivery of UDP, IP packet routing, and the fragmentation/reassembly functionality provided by IP. XFN adds an XFN header to the data before passing it on to the transport layer protocol. XFN implements error checking by requesting a response from the destination device. The XFN protocol and message structure is described in section 5.3 on page 102.

XFN is used to control, modify and inquire about the state of audio/video devices. It is an IPv4 based protocol that can be transmitted over Ethernet as well as firewire. Hereafter IPv4 is referred to as IP. The transfer of IP packets over firewire is known as IP over 1394 and it is described in the RFC 2734 document titled *'IPv4 over IEEE 1394'* (Johansson, 1999). The next section describes some elements of the IP over 1394 protocol.

5.2 IP over 1394

IP over 1394 refers to the encapsulation and transmission of IP datagrams within firewire packets as defined by the RFC 2734 (Johansson, 1999). The RFC 2734 document spells out the criteria necessary for a device to transfer IP datagrams on a firewire network. These criteria include:

- that the device will include a config ROM that conforms with the IEEE 1394a specification (IEEE, 2000) and incorporates an IP unit directory
- that within the device's config ROM bus info block, the *max_rec* field (which specifies the maximum allowable payload size) indicates a payload value not less than 8 bytes.
- that the device is isochronous resource manager capable.
- that the device is able to receive and transmit IEEE 1394 asynchronous stream packets.

All firewire devices that implement this protocol (IP over 1394) possess within their config ROM a unit directory whose *unit_spec_ID* is registered to the Internet Assigned Numbers Authority (IANA) and whose *unit_sw_version* indicates that it is IP over 1394 capable. This unit directory is referred to as an IP unit directory. An IP unit directory has a unit_spec_ID value of *0x00 005E* and unit_sw_version of *'1'* (Johansson, 1999). See section 3.1.3 on page 35 for details regarding a device's unit directory.

IP datagrams can be transmitted within asynchronous packets or asynchronous stream packets. The packet of choice depends on the transaction that an application intends to perform. These packets can be transmitted as unicast, multicast or broadcast transmission. In a unicast transaction the packet is addressed to an individual (single) device on the network. A multicast transaction involves the addressing of a group of devices on the network. While a broadcast transaction is addressed to all devices on the network (Olifer and Olifer, 2006).

The RFC 2734 document defines an encapsulation header for IP datagrams that resides within firewire packets as well as an address resolution protocol called the '1394 ARP' that allows for firewire nodes to be identified on a network. The IP encapsulation header resides just above the data payload of a firewire IP packet. This header allows for the fragmentation and reassembling of IP datagrams (Johansson, 1999). The 1394-ARP is explained below.

Address Resolution Protocol (1394-ARP)

Communication between any two devices on an IP network depends on each device knowing the physical network interface address of the other (Comer, 2006). On such a network (IP-network), devices are identified by their IP addresses, which is assigned to the device statically or by a DHCP[4] server. IP relies on the constant IP address that is assigned to a device on the network (Birk, 2003).

On an Ethernet network, a device's IP address is mapped to its unique media access control (MAC) address. A device's MAC address is a 48-bit address also referred to as the *physical address* of the device's interface card. This address (MAC) is configured into the interface card by its manufacturers and is stored in non-volatile memory. However, certain devices may allow a user to define its MAC address (Washburn and Evans, 1996).

On a firewire network, devices are addressed by their node identifiers (node IDs). A device's node ID may change when a bus reset occurs. As a result, there are no guarantees that a device will retain its node ID after another device has been added or removed from the firewire bus. To address this problem the RFC 2724 specification defines a *'1394 (Address Resolution Protocol) (ARP)'* (Johansson, 1999). The 1394-ARP is not mapped to a device's node ID, but rather to its GUID which is specified in the *bus info block* of the device's config ROM. Refer to section 3.1.3 on page 35 for details on the device's config ROM and GUID.

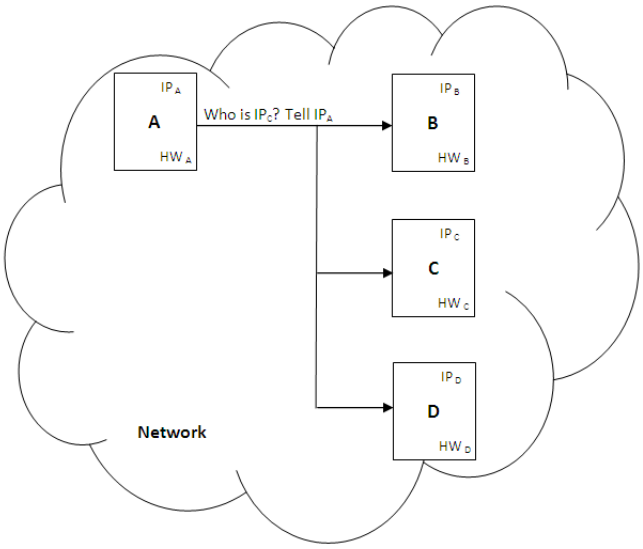


Figure 5.2: Node A broadcast ARP-request to all nodes

[4]Dynamic Host Configuration Protocol (DHCP) automatically assigns IP addresses to network interfaces, while maintaining an address database and eliminates address duplication (Olifer and Olifer, 2006).

Consider the four nodes on the firewire network shown in figure 5.2, Node A, B, C and D. Each node has its IP-address (IP) and physical *(hardware)* address (HW) which is equivalent to the node's GUID in 1394-ARP. If node A wishes to transmit packets to node C, first of all it broadcasts a *1394-ARP* request on the network. This *ARP-request* is received by every node on the bus including node C. The ARP-request will amongst other fields indicate the node A's IP-address (IP_A), nodes A's GUID (HW_A) and node C's IP-address (IP_C). This message as illustrated in figure 5.2, says "Who is IP_C? Tell IP_A".

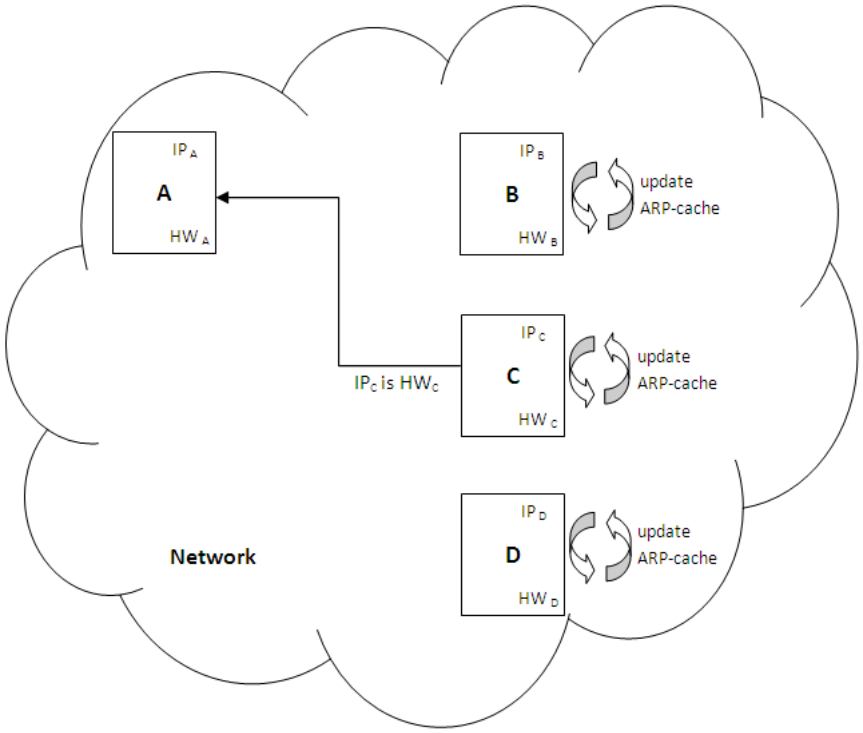


Figure 5.3: Node C sends ARP-response

In response to the ARP-request, only node C, whose IP-address is IP_C, will respond with an ARP-response packet containing its GUID (physical address) (HW_C). This is shown in figure 5.3. Each node on the network keeps an *ARP-table* within volatile memory known as *ARP-cache*. This table maps the IP address to GUID of nodes on the network.

Each node updates its ARP-table with the values it receives from an ARP-request, as illustrated in figure 5.3. Hence, reducing the number of broadcast ARP-requests on the network. For instance, if node B wishes to transmit to node A, it no longer will be required to broadcast an ARP-request considering that it already knows node A's physical address.

In the next section, an audio/video device control protocol known as XFN is described. XFN is an IP-based application layer protocol. As a result, it is possible to transmit XFN messages (within IP packets) over firewire or Ethernet. XFN messages are addressed to physical devices using the device's IP address, irrespective of the medium of transmission.

5.3 XFN

XFN is a peer-to-peer network protocol that facilitates network devices to obtain information about the state of a device's parameters and to modify such parameters (Foss, 2008, 3). These interactions involve the transfer of XFN messages encapsulated within UDP/IP packets. Figure 5.4 illustrates the layout of an IP packet that 'wraps' an XFN frame within a UDP data block.

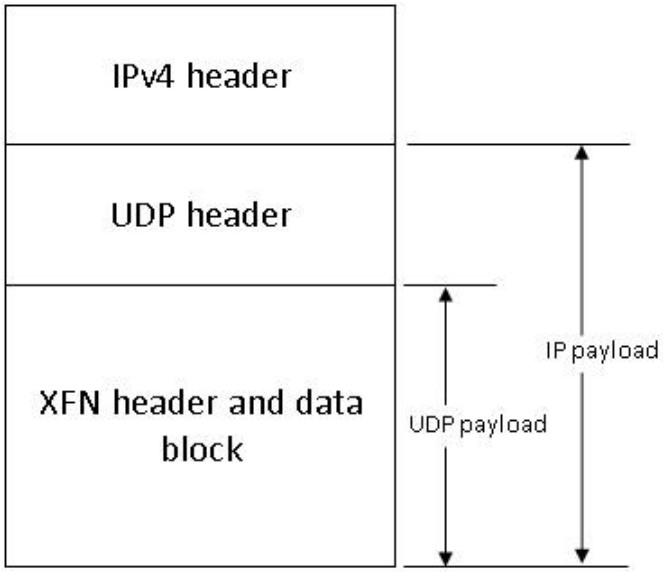


Figure 5.4: UDP/IP packet with XFN frame

Figure 5.4 shows an IPv4 header as the outermost header of the packet. Currently XFN has only been implemented for the IPv4 protocol, which provides a 32-bit addressing scheme for XFN devices on a network. The packet source address field and destination address field are indicated in the IP header.

From the IP header, moving inward is the UDP header. UDP specifies the source port and destination port of an XFN message. With limited error checking capabilities, UDP eliminates the time necessary for appending, decrypting and validating packets as is the case with TCP.

This ensures that XFN messages are transmitted much faster than if they were implemented using TCP.

The XFN header and data block are found within the data section of a UDP datagram, as illustrated in figure 5.4. The XFN header provides for XFN node identification and other XFN protocol dependent attributes. The various fields of the XFN header are discussed in section 5.3.2 on page 109.

Within each XFN capable device is an XFN stack. When an XFN message reaches the destination device, the XFN stack parses the XFN frame and fulfills the corresponding XFN command on the device. The layout of an XFN device is discussed in section 5.3.1.

In XFN a device is modeled on the basis of its various functionalities. Each property within a device is described in XFN as a parameter. It is these parameters that are used to model the device. XFN defines a 7-level hierarchy for the description of a device's parameter. The 7-level hierarchy used to describe device parameters is discussed in section 5.3.1.3 on page 106. In section 5.3.3 on page 110 the nature of XFN commands (requests) and responses are described. A platform independent application programming interface (API) for XFN has been developed (Klinkradt, 2007). This XFN API exposes some functionalities of the XFN stack needed for development of an application that interacts with the stack. The XFN API is discussed in section 5.3.4 on page 112.

5.3.1 XFN device and the XFN stack

An XFN device is any device that implements the XFN stack. This stack is an XFN protocol layer that sits above the IP stack on a device. The layout of an XFN device is illustrated in figure 5.5.

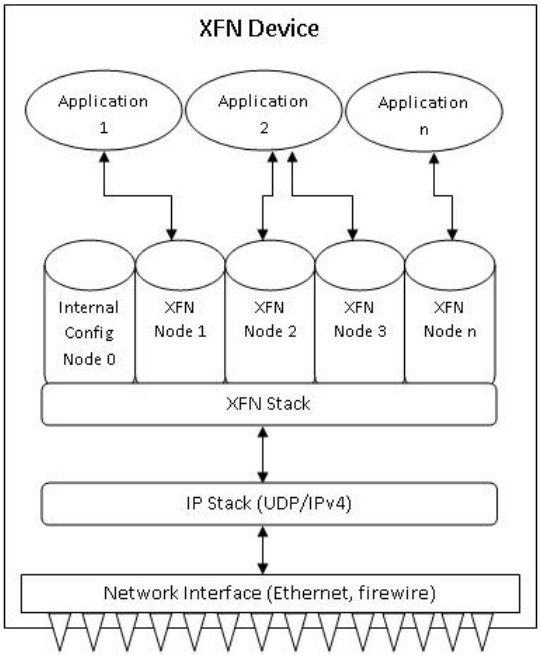


Figure 5.5: XFN device with its associated stack and nodes, adapted from Klinkradt (2007)

In figure 5.5, the network interface refers to the Ethernet or firewire interface on the device. This network interface layer interfaces with the transmission medium currently supported by XFN. The IP stack which interacts with the network interface is an IPv4 implementation that handles UDP and IP processing for XFN. In the case where the device is sending out an XFN command, the IP stack receives the XFN message and encapsulates it within a UDP/IP packet for onward transmission to the network interface. When an XFN message is received from the network interface, the IP stack is responsible for removing the UDP and IP headers from the packet and passing what is left (the XFN message) to the XFN stack.

The XFN stack interacts directly with the IP stack. It is capable of handling, receiving and transmitting XFN messages. The current implementation of the XFN stack is a singleton program that processes all XFN communication. Above the XFN stack are nodes, which are each a conceptual container that incorporates the parameters within the XFN device. The XFN stack creates a node at start up known as the *Internal Configuration Node*. This node is identified with an XFN node ID of '0'. Subsequently, all applications that utilize the XFN stack create their own nodes, which are identified with XFN node IDs '1' and increasing (Klinkradt, 2007). However, there are instances where an application creates more than one node above the XFN stack as exhibited by Application 2 of figure 5.5. In such a case, each node may represent specific units within a device, or as is the case with an XFN proxy for AV/C devices, each node represents a

single device (Foss, 2008, 6). The following sections describe various components of the XFN stack.

5.3.1.1 XFN Node

A node as described above is an instance of a functional entity within an XFN device. It is created above the XFN stack as a virtual representation of an XFN device's unit. An XFN node is uniquely identified by an XFN node ID and incorporates the XFN device's node parameters. The *Internal Configuration Node*, also referred to as 'node zero', has a node ID of '0'. It contains diagnostic and general configuration information about the device. The information incorporated in node zero includes the device's IP address, subnet, device GUID, a textual name of the device, device type and firmware revision. The information in node zero is common to all the other nodes on the XFN stack. Every XFN device creates, in addition to the node zero, at least one other node, which contains the device's parameters.

There is a difference between a firewire node and an XFN application node. A firewire node refers to a logical entity within a firewire device as described in section 3.1 on page 31. An XFN application node refers to a device's parameter 'container' built above the XFN stack that represents an instance of an XFN device as a functional unit.

5.3.1.2 XFN Parameter

An XFN parameter represents a particular attribute or control within a device. For example an audio mixer could have an output volume control, which is represented as an output gain parameter. Associated with every parameter is a value that indicates the state of the parameter. XFN allows for the determination and modification of parameter values. Every parameter can be addressed in either of two ways:

- Full data block - this is the addressing of a parameter using its 7-level hierarchical address to identify it.
- Parameter index - this is an identifier that is unique to every XFN parameter. A device's parameter index is associated with a parameter when the parameter is created.

The following section explains the 7-level hierarchy as defined by XFN. Bear in mind that the 7-level hierarchy is a parameter addressing scheme that defines a particular parameter. Hence, at the bottom of a 7-level hierarchy, is the actual value of a parameter.

5.3.1.3 7-level Hierarchy

XFN describes every parameter according to a 7-level hierarchy. Conformance to this 7-level description of a parameter results in a consistent way of addressing XFN parameters. . This makes XFN device parameter addressing unambiguous. As a result, a requesting device can send an instruction to modify a parameter on any target device indicating the 7-level address of the parameter. If the parameter of interest exist on the target, the target will respond to the modify instruction. Furthermore, this implies that a requesting device does not need to identify all parameters within a target device. The requester sends an XFN message specifying the parameter of interest, and if that parameter exists the target responds.

Foss (2008, 9-10) defines the 7-levels that form this hierarchy as:

- *Section Block* - is a high level classification of parameters within a device into the various functional units that exist on the device. For example, a mixing console might have an input section, output section and a matrix section.
- *Section Type* - is a further sub-grouping of a section block into its functional subunits that differentiates the various components of a section block. For instance, the mixing console might have, within its section block input, a section type 'audio' that represents all the analog audio input signals to the device.
- *Channel Number* - represents the processing or routing path of a signal. This allows for a signal's channel to be traced throughout a multi-device network.
- *Parameter Block* - is a grouping of parameters that allow for the processing and routing of audio channels. For instance, a mixing console might possess a block of equalizers that allow for the equalization of an audio channel.
- *Parameter Block Index* - allows for differentiating similar components within a parameter block. For example, a mixing console might have equalization sub-groupings of parameters related to gain, within its equalizer parameter block.
- *Parameter Type* - this level specifies the type of parameter being accessed. For instance a mixing console might possess gain, clock source and PAN parameters.
- *Parameter Index* - this is used to uniquely identify a parameter, from other parameters of the same parameter type, within a device. For instance, a mixing console might possess

multiple gain parameters within a single channel, each of these gain parameters possess a unique parameter index.

XFN provides for the use of *dummy* values when, for instance, a parameter can be described with less than the 7-levels. In such a case, the particular level (of the 7-level parameter description) that is not necessary for the description of a device's parameter is assigned a dummy value. This dummy value is defined in XFN as a '1'. For instance, if only a single 'gain' parameter exist on a particular input channel on a mixing console, the parameter block index will be assigned a dummy value.

At any level, a '*wildcard*' can be used to indicate all parameters at that level (Foss, 2008, 13). For instance, if a wildcard is placed at the section type level (level 2), the XFN device returns all section types of the indicated section block that exist on the device. This wildcard mechanism is particularly important for device parameter exploration, such as implemented by the XFN Level Explorer described in section 5.4 on page 115. The use of the wildcard allows a controller to address all tree-nodes or parameters, at a particular level, by sending a single XFN message.

The 7-level hierarchy for any device's parameters forms a tree structure with each level being a node on the tree. The tree nodes referred to here are different from the XFN application node referred to in section 5.3.1.1. An XFN tree node refers to a particular level in a 7-level hierarchy that describes a device's parameter. Figure 5.6 shows the tree structure for an XFN device.

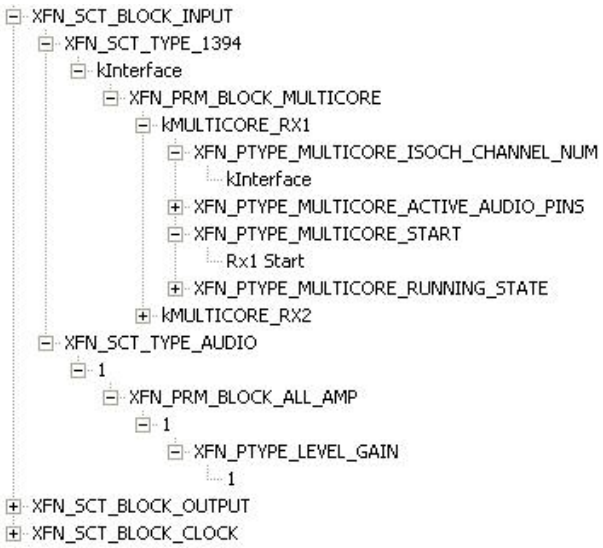


Figure 5.6: An example of XFN parameter tree structure

Figure 5.6 depicts the following:

- three section blocks at level 1. These are:
  - XFN_SCT_BLOCK_INPUT
  - XFN_SCT_BLOCK_OUTPUT, and
  - XFN_SCT_BLOCK_CLOCK
- the input section block has two section block types at level 2. These are:
  - XFN_SCT_TYPE_1394, and
  - XFN_SCT_TYPE_AUDIO
- the XFN_SCT_TYPE_1394 of the input section block has one channel at level 3 which is represented as 'kInterface' in the figure.
- the channel (kInterface) of the XFN_SCT_TYPE_1394 has one input parameter block at level 4, namely XFN_PRM_BLOCK_MULTICORE.
- the XFN PRM BLOCK MULTICORE has two parameter block indices at level 5, namely:
  - kMULTICORE_RX1, and
  - kMULTICORE_RX2
- the kMULTICORE_RX1 has four parameter types at level 6. These parameter types are:
  - XFN_PTYPE_MULTICORE_ISOCH_CHANNEL_NUM
  - XFN_PTYPE_MULTICORE_ACTIVE_AUDIO_PINS
  - XFN_PTYPE_MULTICORE_START, and
  - XFN_PTYPE_MULTICORE_RUNNING_STATE
- the XFN_PTYPE_MULTICORE_START has a unique parameter index at level 7, represented as 'Rx1 Start'.

The names of the XFN tree nodes depicted in the figure 5.6 on the preceding page are aliases that refer to XFN defined values. The 7-levels form the various fields of an XFN data block. The XFN header and data block are discussed in the following section.

5.3.2 XFN header and data block

The XFN header and data block are encapsulated within a UDP/IP payload. Refer to figure 5.4 for a layout of UDP/IP packet with an XFN message. Figure 5.7 shows an XFN message without the IP and UDP headers.

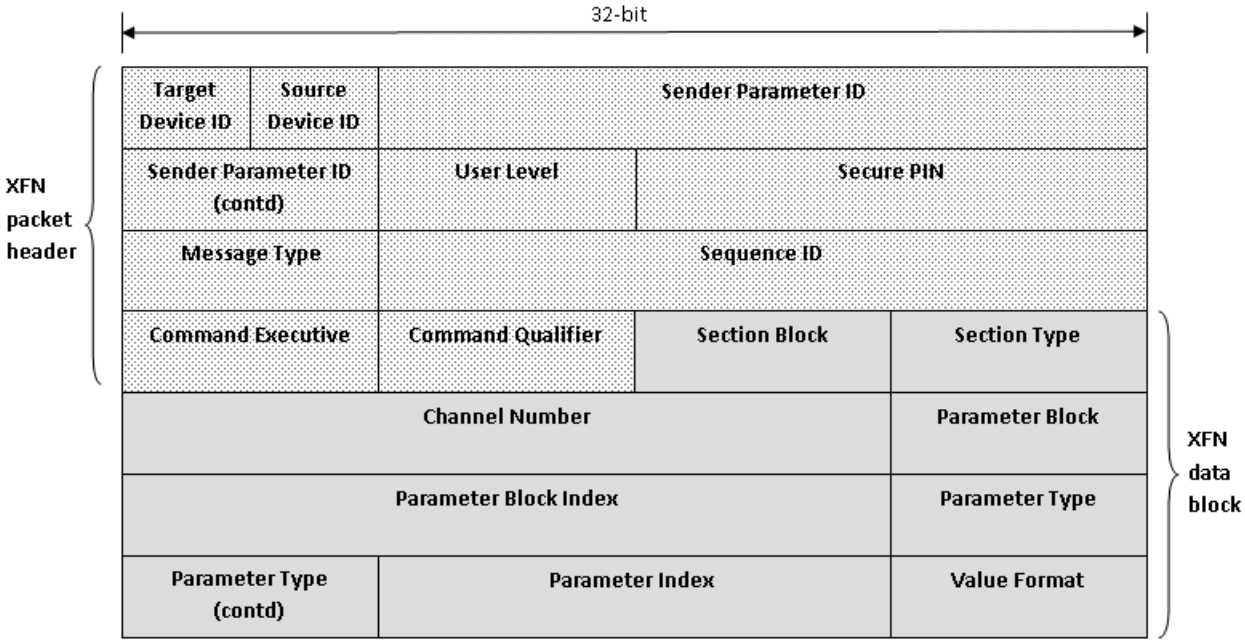


Figure 5.7: The structure of an XFN message adapted from Foss (2008)

According to Foss (2008, 6-7), the XFN header consists of the following fields:

- Target Device ID - is the XFN node ID of the XFN node that the XFN message is directed at.
- Sender Device ID - is the XFN node ID of the XFN node that sent the XFN message.
- Sender Parameter ID - is the identifier of the XFN parameter on the sender device.
- User Level - is used as an authorization mechanism that allows users to be grouped into various categories for device control, with each category having varying privileges.
- Secure PIN - is an authentication mechanism used to validate whether a user is authorized to carry out the desired operation.

- Message Type - determines the nature of the XFN message. That is whether the XFN message is a request or a response, and if the XFN parameter is addressed using a full data block or parameter index addressing scheme.
- Sequence ID - is used to identify an XFN message, thus enabling an application to associate a response with a particular request.
- Command Executive - is used to specify the kind of action that should be performed on the parameter. For example 'GET' or 'SET'.
- Command Qualifier - specifies what attribute the command executive is directed at. For instance, a GET command executive could have a command qualifier VALUE. This will imply that the XFN message seeks to obtain the value of the XFN parameter it is addressed to.

These header fields provide security and integrity for an XFN message, and instructions to an XFN device. The XFN data block incorporates the XFN 7-levels parameter addressing scheme already described in section 5.3.1.3 on page 106. Immediately following the parameter index field is the *value format* field. The value format indicates the type of value and the length of the value field. The actual value of the parameter follows this value format field.

5.3.3 XFN messaging

XFN messaging involves the transfer of either an XFN request or an XFN response packet. In an XFN transaction, an XFN request is sent indicating whether it (the request) requires a response from the target node. In the case where a response is required the XFN header's *sequence ID* is used to associate a response to a particular request packet. Otherwise, if no response is required from the target, the XFN request does not specify a sequence ID.

XFN request messages are of two types, namely:

- Full datablock request - this includes a 7-level address of the XFN parameter in the XFN datablock. The structure of this command is shown in figure 5.7 on the preceding page.
- Indexed message request - this message incorporates a 32-bit parameter ID of the parameter it addresses. Each parameter ID is associated with a device's parameter (by its XFN stack) during parameter creation. The structure of an XFN indexed message is shown in figure 5.8 on the next page.

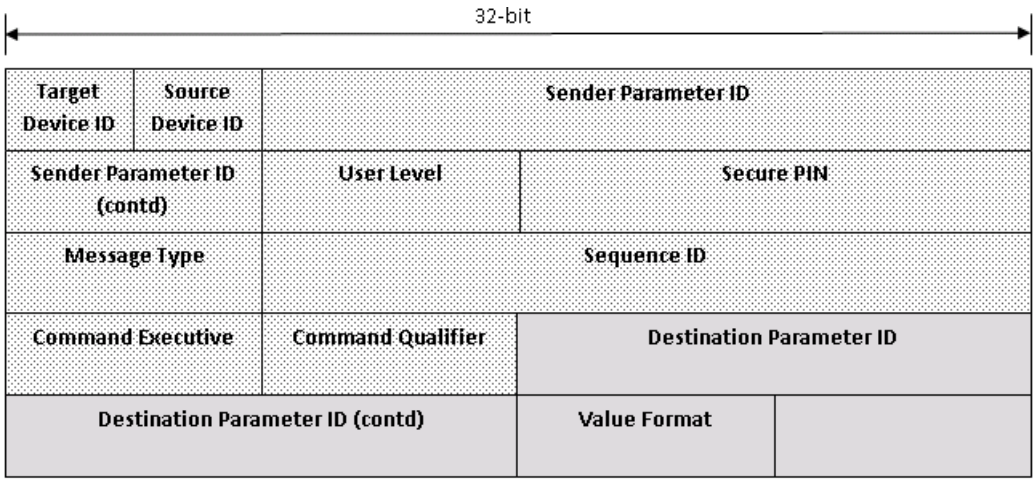


Figure 5.8: XFN indexed message (Foss, 2008)

In all XFN messages the *Message Type* field determines whether the message is

- a full datablock with response command
- a full datablock with with no response command
- an indexed message with response command
- an indexed message with no response command, or
- a response message.

Two other fields that determine the particular action a request should perform are the *Command Executive* and *Command Qualifier* fields. The command executive indicates the essence of a command. Two such 'executive' commands are the GET and SET requests. A GET request returns the value of the parameter specified by the 7-level address or parameter ID of an XFN request message. The SET request changes the value of the specified parameter.

The command qualifier specifies what type of parameter should be returned in the case of a GET command executive or modified in the case of a SET command executive. With regards to this thesis, the command targets of interest are:

- VALUE - this refers to a parameter's value specified by the value format field in an XFN message.

- ID - this refers to a unique numerical value that identifies a parameter.
- NAME - this refers to the parameter name and is of type character string.

An XFN response takes on a different structure as shown in figure 5.9. The XFN header of an XFN response is similar to that of an XFN request except for the absence of the COMMAND EXECUTIVE and COMMAND QUALIFIER fields. The data block of an XFN response does not specify any parameter, in contrast to an XFN request, but contains the value format. The *sequence ID* is used to match an XFN response to an XFN request (Foss, 2008, 20).

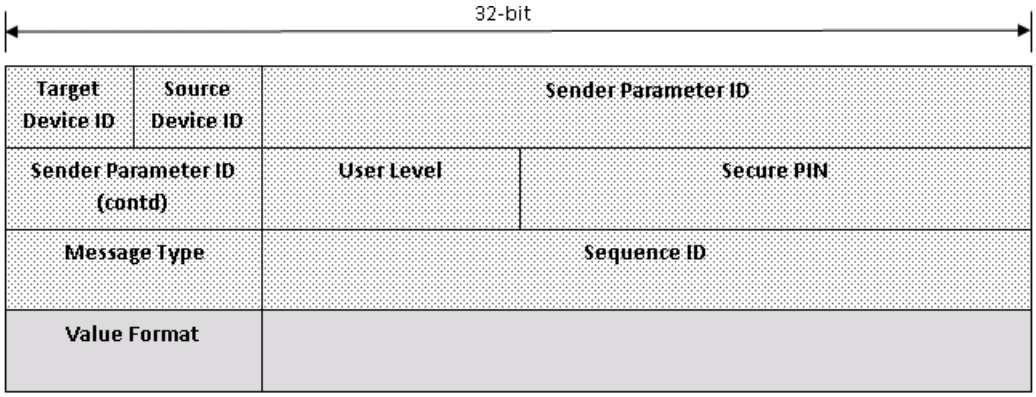


Figure 5.9: XFN response message (Foss, 2008)

XFN parameters and XFN messages can be created by a software developer using the XFN API described in the following section.

5.3.4 XFN API

The XFN API provides an interface for software developers to communicate with the XFN stack. As mentioned in section 5.3.1 on page 103, the XFN stack forms the basis of XFN protocol messaging. Applications are built above this stack. At start up, the XFN stack initializes the various message processing queues necessary for its functioning. These include a transmit queue for messages that are to be transmitted, a receive queue for messages that have been received but not yet processed, and a retransmit queue for sent messages whose response have not yet been received.

Also created at start up of the XFN stack is the internal configuration node (node zero) described in section 5.3.1.1 on page 105. Node zero contains diagnostic and general information param-

eters that pertain to the XFN stack and are common to all other nodes created above the XFN stack. This is depicted in figure 5.5 on page 104.

Klinkradt (2007) has divided the XFN API into the following categories:

- XFN Stack Initialization and Cleanup - this category consists of API calls to create the threads and initialize the queues necessary to start up the XFN stack. This capability is exposed by the *XFNInitialiseTasks( )* method of the API. An application will call the *XFN_Cleanup( )* method of the API in order to properly clean up the XFN stack. This releases previously allocated memory for queues and other data structures and also stops all threads.
- XFN Hierarchy Creation - this category encompasses all API functions that are involved in the creation of an XFN parameter. These include:
  - the creation of an XFN node using the API's *addXFNDeviceNode( )* method
  - the creation of each of the 7-levels defined by the XFN protocol using the *createXFNLevel( )* method
  - defining aliases for the created levels using the *setXFNLevelAlias( )* method
  - creating a parent-child relationship using the *addChildXFNLevel( )* method to form a hierarchy
  - adding the above created level to a particular XFN node with the *addLevelToXFNDevNode( )* method
  - creating the XFN parameter and registering the parameter callback using the *createXFNParameter( )* method
  - associating the created parameter with the seventh level (parameter index) of the level hierarchy using the *addParameterToXFNLevel( )* method
  - incorporating a parameter into a node with the *addParameterToXFNDevNode( )* method.
- XFN Messaging - these are API functions used by an application to send XFN messages through the XFN stack. Hence an application is able to get or set a parameter value on a remote device using the appropriate GET or SET command provided by the API. These commands can be performed as blocking or non-blocking requests. The API provides the following full datablock (get and set) blocking and non-blocking methods:

  - *getRemoteParamValue_block_fdb( )*
  - *getRemoteParamValue_nonb_fdb( )*
  - *setRemoteParamValue_block_fdb( )*
  - *setRemoteParamValue_nonb_fdb( )*

  If the application wishes to change a parameter value of its local host the *setParamValue()* method is used.

- XFN Level Commands - the XFN API provides a *getChildLevelAliases( )* method to explore any level of the 7-level hierarchy that addresses a parameter. A level alias is a textual descriptor that is associated with each of the nodes of an XFN 7-level parameter tree. This method uses the wildcard value of *0xEE* to indicate the 'child' level that it is inquiring about. For instance, if an application wishes to obtain all the section types (level 2) of a particular section block (level 1) that exist within a device, it uses the *getChildLevelAliases* method. To use this method the application passes a seven element array (each element corresponds to one of the 7-levels of a parameter hierarchy) with the value of the first element indicating the section block of interest (for example, input section block) and the remaining six elements having a wildcard value of *'0xEE*'. On receiving this command the target (XFN) node returns the aliases for all section block types (level 2) it implements. Since the other lower levels (levels 3-7) have value *0xEE* the target node also returns the aliases of each of those levels that are implemented in the device. The possible values of an XFN tree node at a particular level are defined by the XFN protocol. See *XFN Protocol Overview* (Foss, 2008, 41-71) for currently defined XFN parameter values.

- XFN Helper Functions - a *createDatablock( )* method takes an integer array and packages it into an XFN full datablock.

Associated with every parameter that is created is a *callback* implemented in the application space (Klinkradt, 2007). When the XFN stack receives a message addressed to a parameter, the appropriate callback is triggered to perform the instruction within the XFN message. For instance, a gain parameter of an audio mixer will have a *gainCallback* function that is associated with it. The association between a parameter and its callback is established during the creation of the parameters using the API *createXFNParameter( )* method.

XFN messages can either be *blocking* or *non-blocking* in nature. If a blocking transaction is used, the sender waits until it gets a response before it sends any further request. In a non-blocking

transaction requests can be sent indiscriminately, with the responses all queued in a response buffer. It is up to the application to pick the response up from the response buffer.

The next section describes an XFN application that explores the XFN 7-level parameter hierarchy of discovered XFN device nodes, on an XFN network. This will give an indication of how the XFN API can be utilized.

5.4 XFN Level Explorer

The XFN Level Explorer is currently implemented on the *Windows XP* platform. It utilizes the JUCE library (mentioned in section 4.2 on page 82) for its graphic display. Figure 5.10 shows the XFN Level Explorer.

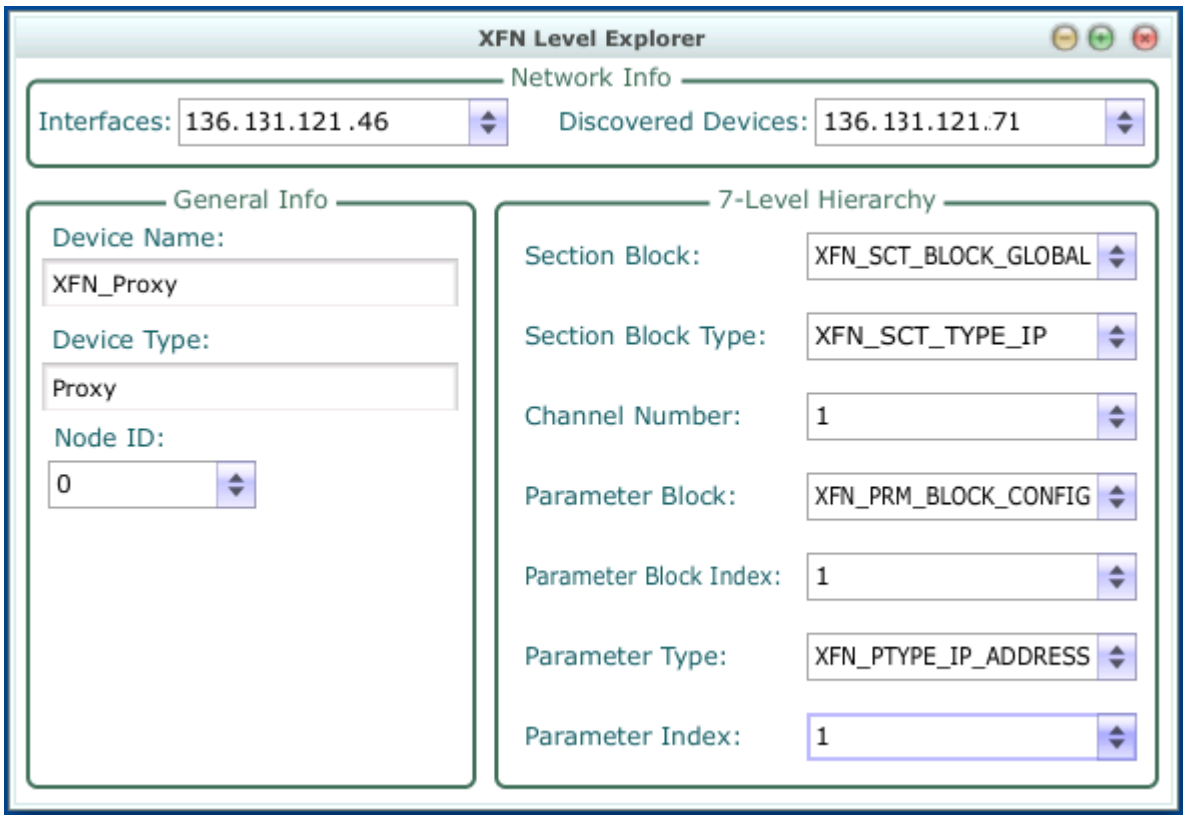


Figure 5.10: XFN Level Explorer

As can be seen in figure 5.10, the XFN Level Explorer uses a number of combo boxes to display the various options for each field. The XFN Level Explorer comprises three groupings, and

each provides different information. The Network Info, General Info and 7-Level Hierarchy are explained in the following subsections.

At start up, the XFN Level Explorer initializes the XFN stack, by calling the *XFNInitialiseTasks* method, thus making the stack available for all subsequent interactions with XFN devices on the network. To communicate with other devices and their associated nodes, the XFN Level Explorer uses various XFN API methods. The first process undertaken by this application is to obtain the IP-address of its local host. This is achieved by interactions between the XFN stack and the IP stack. Refer to figure 5.5 on page 104 for the layout of an XFN device, which indicates the interaction between the XFN and IP stacks. Once a handle to an IP interface (hence an IP-address) has been obtained, the application discovers other XFN devices on the network.

When the 'close button' (top-right corner) of the XFN Level Explorer application is pressed, the application makes a call to the *XFN_Cleanup* method of the XFN API. This causes the XFN stack to destroy and clean all the messaging processing queues and threads.

5.4.1 Network Info

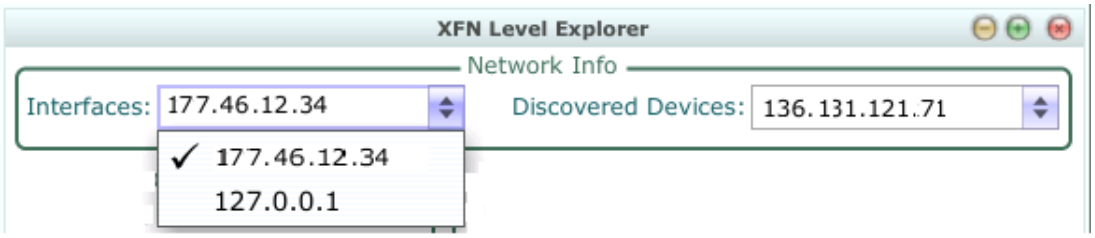


Figure 5.11: XFN Level Explorer network information

To address XFN devices on the network, two functions are performed by the application:

1. It obtains the IP addresses on the local host on which the application is running. This gives the application a handle to any of the IP interfaces on its host by using the interface's IP address. In turn, this makes the host XFN device addressable on the IP network. Since IP-addresses are used to address packets to devices on an IP network.

2. It discovers XFN devices on the network by obtaining their IP addresses. Each XFN device on a network is uniquely identified by its IP-address.

The IP addresses of the host machine running the XFN stack are obtained from the IP stack on the host machine. The combo box labeled 'Interfaces' will display all the IP addresses on the host machine, including its loop back address (127.0.0.1), firewire, LAN and wireless LAN IP addresses, if applicable.

The combo box labeled *Discovered Devices* displays the IP addresses of all devices on the network that implement the XFN stack. This application discovers the IP addresses of XFN devices by broadcasting an XFN message addressed to the 'IP address' parameter. This message is picked up by the IP stack of all devices on the network. However, only XFN devices respond, since they are the only devices that understand the XFN message. To achieve this, the XFN Level Explorer calls the XFN API's *getRemoteParamValue_nonb_fdb()* method. As the name implies, this method sends a non-blocking full datablock request to get the value of the parameter indicated in the XFN datablock. This 'get' command takes a callback function as one of its arguments. The callback is triggered when a response is received. All XFN devices on the network return their IP addresses on receiving the 'get IP address' command, thus triggering the associated callback implemented by the application.

5.4.2 General Info

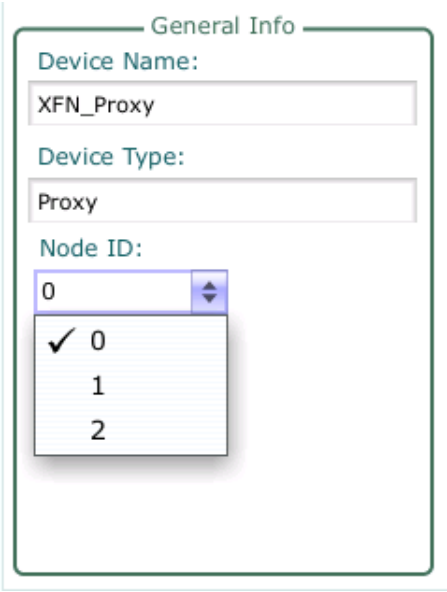


Figure 5.12: XFN Level Explorer general information

The *General Info* area displays information obtained from the internal configuration node (Node zero, see section 5.3.1.1 on page 105) of the XFN device selected in the *Discovered Device*s combo box of figure 5.11. The *Device Name* is obtained by sending a 'get remote parameter value' command addressed to the device name parameter of the selected device (in the *Discovered Devices* combo box). The device name is a textual description that describes what device the XFN stack is implemented on. This value is set by the application that runs the XFN stack.

The *Device Type* is obtained by sending a non-blocking full datablock 'get value' command addressed to the device type parameter of the selected device (in the *Discovered Devices* combo box). For the purpose of this research, an *XFN PROXY DEVICE* was defined as one of the device types recognized by the XFN stack. Other device types defined in the XFN stack include: *WINDOWS PC*, *ROUTER*, *X1*, *FIREWIRE/ETHERNET TUNNEL,* and *AVB GATEWAY*.

The *Node ID* combo box displays the XFN node IDs of all XFN nodes implemented in an XFN device. For the purpose of this research, an *XFN Device Node Count* parameter has been defined to indicate the number of XFN nodes that have been created above a device's XFN stack. Hence if an application creates an XFN node above its host XFN stack, the XFN Device Node Count parameter will have a value of two. This is because node zero is always implemented by the stack and the application implements one further node. The device whose XFN node IDs are shown in figure 5.12 implements three nodes on its XFN stack. One node is implemented by the device's XFN stack (Node 0) and there are two other application specific nodes (Nodes 1 and 2). In the same manner as was used to obtain the device name and device type parameters, a 'get remote parameter value' command is sent to the selected device. In this instance, the XFN message is addressed to the XFN device node count parameter. All subsequent requests to obtain information about a node's parameter, is required to indicate the node of interest with its XFN node ID.

Once a node is selected, the application begins to explore the 7-level parameter hierarchy of the selected node.

5.4.3 7-Level Hierarchy

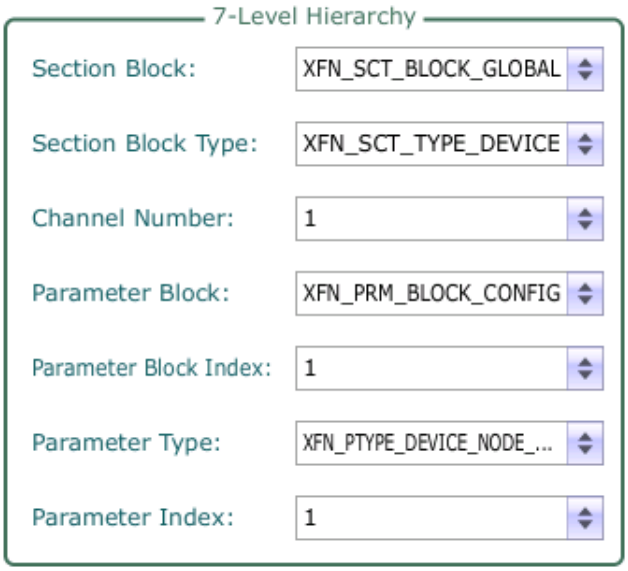


Figure 5.13: XFN Level Explorer 7-level parameter hierarchy

The 7-level hierarchy that describes a device's parameter is explained in section 5.3.1.3 on page 106. The level explorer displays the aliases of each of the 7-levels in this *7-Level Hierarchy* area. Two aspects of the XFN protocol that allow for the exploration of a device's parameter 7-level hierarchy are:

1. *Wildcard* mechanism - allows for all XFN tree node alias at a particular level to be returned. When the XFN stack parses the XFN datablock and at any of its 7 levels it detects a 'get alias' wildcard (of value all *'0xE*'s), it returns the XFN tree node alias at that level.

2. XFN API *getChildLevelAlias()* method - refer to section 5.3.4 on page 114 for a description of this method. It is used to get the XFN tree node parameter aliases by specifying a wildcard at the child-level it seeks to obtain. For instance, to obtain all parameter blocks (at level 4) implemented in a device's input section block (level 1), the 7-level hierarchy from level 1 to level 3 are specified and a wildcard is placed at level 4 to level 7. This will cause the device to return all its child XFN tree node aliases from level 4 to level 7.

When any level of the combo box is selected, there is an API call to *getChildLevelAlias()* which then sends an XFN message to a targeted device node. When the stack on the target parses the

XFN datablock and detects an alias wildcard, it returns aliases of all the XFN tree nodes at the level where the wildcard is detected. If the wildcard is at the section block level (level 1), the target will respond by returning all the section blocks it implements.

5.5 Summary

This chapter introduced the Internet Protocol version 4 (IPv4) and how it has been used as an addressing scheme for networked devices. UDP was mentioned as a fast, connectionless transport layer protocol with no datagram delivery guarantees. The use of IP over firewire was described with mention of the 1394 address resolution protocol (ARP).

XFN, a peer-to-peer protocol that has been implemented for the command and control of audio and video devices was introduced. The concept of parameters as variables and controls within an XFN device was explained. An XFN device as a functional entity that implements the XFN stack was described. The nature of the XFN stack, XFN packet structure, and XFN messaging were explained. An insight was provided into the XFN API that allows software developers to communicate with the XFN stack on host and remote devices.

This chapter ends by describing an application known as the XFN Level Explorer that implements the XFN protocol. The XFN Level Explorer utilizes the XFN API for device discovery and tree node exploration. The various features of the XFN Level Explorer were also described.

With the understanding of the XFN protocol which was presented in this chapter, and the AV/C protocol that was presented in chapter 3, this research proposes a proxy as a means to achieving network interoperability between both protocols (XFN and AV/C). In the next chapter the design and implementation details of the XFN/AVC proxy created as a product of this research are discussed.

Chapter 6

Proxy Design and Implementation

This chapter builds on the discussions of the AV/C protocol in chapter 3 and the XFN protocol in chapter 5. The AV/C device controller described in chapter 4 and the XFN level explorer described in chapter 5 laid the foundation for the implementation of an AV/C and XFN network interoperability scheme described in this chapter. This scheme entails the use of an 'interpreter' known as the *AV/C Proxy*.

The AV/C Proxy is an application that facilitates communication between networked AV/C and XFN devices. The proxy (AV/C proxy) models AV/C devices and receives XFN instructions on their behalf. It then translates those messages to AV/C commands that are targeted at AV/C devices on a firewire bus.

In the development of the AV/C proxy, there were several design considerations and implementation decisions. In this chapter these design and implementation details are described. This is followed by a description of a practical test situation in which the AV/C proxy was used.

6.1 XFN Controller for AV/C devices

In the design of the proxy for AV/C device interoperability with XFN, two approaches were considered. These are the *AV/C proxy* and the *XFN proxy* approaches. These two approaches require that

- a particular type of networked device conforming to a protocol is discovered by a proxy

- the device's parameters are modeled within a device object that conforms to a different protocol
- the device object resides within the proxy.

The difference between the two approaches lies in the type of device object that is created, and hence the protocol the device object can use for communication. These device modeling approaches are described in the following subsections.

6.1.1 XFN Proxy Approach

In this approach, device objects created within the proxy are modeled in terms of AV/C. This implies that all XFN devices on an IP-network will be discovered, and for each XFN device, a virtual AV/C device object is created within the proxy. Other AV/C devices can communicate with the XFN devices by directing AV/C commands to the proxy. For each AV/C command received by the proxy, an appropriate XFN message is sent to the indicated XFN device. After the execution of the XFN message, the proxy then sends an AV/C response to the AV/C controller node, indicating the status of the received command.

Figure 6.1 portrays the interaction between an AV/C device and an XFN proxy that models two XFN devices.

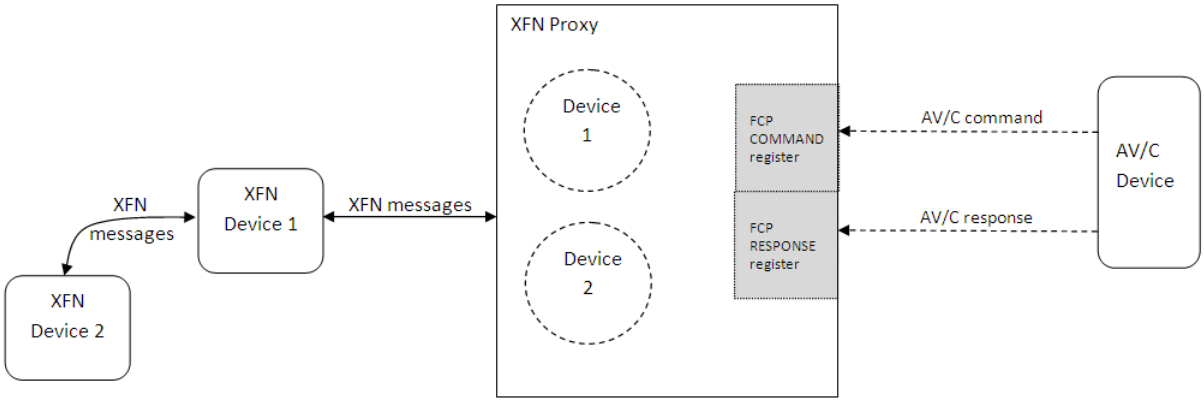


Figure 6.1: XFN proxy interaction with AV/C device

In figure 6.1 the 'AV/C Device' represents a physical AV/C device on a firewire network that acts as a controller node. The XFN proxy interacts with two XFN devices ('XFN Device 1' and 'XFN Device 2') that are possible target nodes for an AV/C command directed at the proxy.

Each of the AV/C device objects ('Device 1' and 'Device 2' in figure 6.1) can be represented as units within the proxy. Hence each device object will be uniquely identified by a *unit ID* assigned to it by the proxy. This ensures that the entire proxy network of devices (to the left of the proxy in figure 6.1), including the proxy, is seen by the AV/C device as a single AV/C node with multiple units.

If 'Device 1' models 'XFN Device 1' and is assigned unit ID '1' by the proxy, any AV/C command intended for 'XFN Device 1' is required to specify this unit ID ('1').

The proxy will be required to model the units that it creates for the XFN devices in terms of descriptors and info blocks. These descriptors and info blocks are structured interfaces that the proxy will present to any AV/C controller node that seeks to explore the capabilities of the AV/C device objects. In AV/C, the layout of descriptors and info blocks are dependent on the unit/subunit type that they model. 1394 Trade Association (2001b) defines values for AV/C unit/subunit types. Each AV/C subunit specification defines the nature of the descriptors and info blocks that are implemented within the subunit.

There exist a relatively small number of possible unit/subunit types, and hence descriptors and info blocks, available to describe the wide range of possible XFN parameters. This presents a challenge when considering a professional (XFN) audio device that possesses a wide range of parameters. Such a device might not be adequately modeled by the relatively few AV/C subunits and their associated descriptors and info blocks. This descriptor and info block creation process might also require a lot of processing time on the part of the proxy, depending on how many parameters exist on the XFN device being modeled.

6.1.2 AV/C Proxy Approach

In the AV/C proxy approach, the device objects created are modeled in terms of the XFN protocol. This will require that all AV/C devices are modeled as XFN devices, and communication will involve the transfer of XFN messages. Refer to section 5.3.3 on page 110 for a discussion of XFN messaging. Figure 6.2 shows the interaction between an XFN device and a proxy that models two AV/C devices.

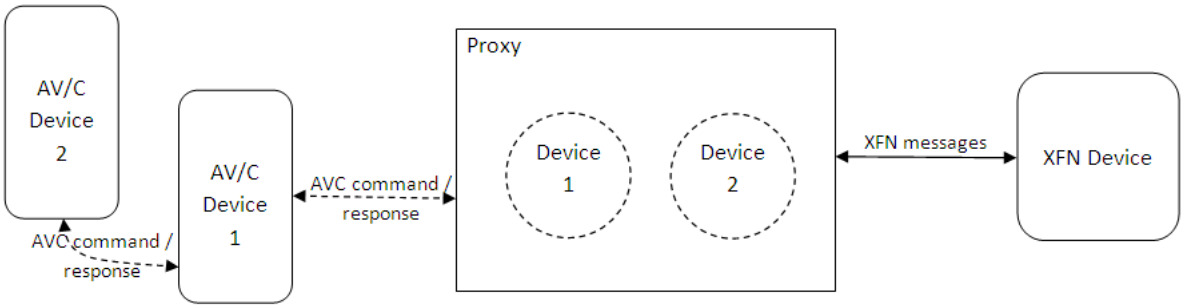


Figure 6.2: AV/C proxy interaction with an XFN device

The proxy discovers the AV/C devices on the network and for each AV/C device a corresponding XFN device object is created. In the figure, two AV/C devices are discovered by the proxy, hence the proxy creates two XFN device objects (Device 1 and Device 2) that model 'AV/C Device 1' and 'AV/C Device 2' respectively. Each of the created XFN device objects is uniquely identified by its XFN node ID. If the 'XFN Device' of figure 6.2 intends to send a message to 'AV/C Device 1', it specifies the XFN node ID of the proxy's 'Device 1' within an XFN message, and then sends the XFN message to the proxy. On receiving an XFN message addressed to a proxy device object, the proxy translates the XFN message to an appropriate AV/C command, and then sends the AV/C command to 'AV/C Device 1'.

Using this approach, the various parameters within an AV/C device can be explored by sending XFN messages to the device object (within the proxy) that models the AV/C device.

6.1.3 Motivation for AV/C Proxy Implementation

The AV/C proxy approach was chosen over the XFN proxy approach. The following reasons influenced this decision:

- XFN provides an easy way of exploring device parameters, since XFN device parameters are modeled in a standardized 7-level hierarchy. In AV/C, device parameters are modeled using descriptors and info blocks which:
  - have an infinite hierarchy from a root descriptor to its leaf descriptor(s) with their associated info blocks. The descriptor and info block hierarchy is explained in 'Descriptors and Information Blocks Mechanism' of chapter 3 (section 3.3.3.1 on page 62).

  - depends on the particular unit or subunit that is modeled. For instance, the descriptors and info blocks of the audio subunit are different from those of the music subunit.

- The (OSI/ISO) link layer independent nature of XFN ensures that by using the AV/C proxy approach, an AV/C device can be communicated with using any IP supported link layer protocol such as Ethernet and firewire.
- AV/C device control is able to take advantage of the security features present in XFN. These include authentication and access level privileges that are implemented in the XFN protocol. The proxy is able to provide strict adherence to XFN level privileges by restricting unauthorized users from performing privileged tasks.
- There is a comprehensive control and connection management application available for XFN devices.

The XFN model of an AV/C device allows the AV/C device to perform its various functions while taking advantage of the XFN network. Each of the AV/C devices are modeled as XFN nodes within the AV/C proxy.

6.2 Requirements of the AV/C Proxy

The AV/C proxy implemented in this investigation is expected to allow for communication between AV/C and XFN networked devices. The AV/C proxy is required to receive XFN instructions targeted at an AV/C device, and then translate such instructions into AV/C commands. The proxy will reside on a workstation. At startup, it will discover all AV/C devices on the network. For each discovered AV/C device, an XFN node with its associated parameter hierarchy will be created to model the parameters that exist on the device.

The XFN node parameters that are created by the proxy will depend on the type of units and subunits that exist within the discovered AV/C node. Thus, the XFN proxy is required to determine the unit/subunit type and parse the unit/subunit descriptors and info blocks in order to determine its capabilities and attributes. Based on the information that the proxy obtains about the device, it creates parameters within the XFN node that correspond to the device. These parameters represent specific controls within the AV/C devices, whose values can be queried or modified. For example, a ‘gain parameter’ could represent the volume on a device which can be increased, decreased or muted.

All XFN communications with AV/C devices will be directed at the proxy. On receiving an XFN message, the proxy will parse the message to identify what parameter is addressed. Once the parameter has been determined, the proxy then translates the XFN message by determining the action indicated in the COMMAND EXECUTIVE and COMMAND QUALIFIER fields within the XFN message. Refer to section 5.3.2 on page 109 for a description of the various fields within an XFN message. The proxy then sends the appropriate AV/C commands to the AV/C node that corresponds to the XFN node ID indicated in the received XFN message.

The figure below shows a typical setup that includes two AV/C devices, the XFN proxy running on a workstation, an XFN device, and an XFN connection manager application running on a workstation.

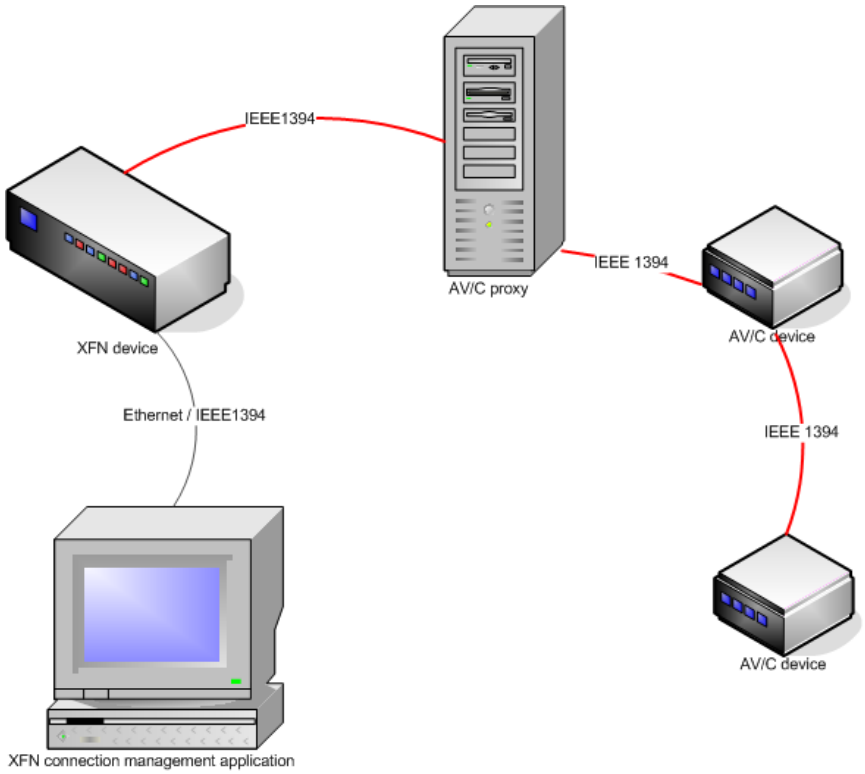


Figure 6.3: Physical layout of network with AV/C proxy

Figure 6.4: Logical layout of AV/C proxy interactions with networked devices

In figure 6.3, the AV/C proxy resides on a workstation and is connected to the AV/C devices. The XFN device is connected to the AV/C proxy via firewire, and is connected to the XFN connection management application via Ethernet or firewire. Some XFN devices possess both Ethernet and firewire interfaces. Figure 6.4 shows the message types used to communicate with the proxy. The proxy interacts with XFN devices by sending XFN messages, which are transmitted over IP on an IEEE 1394 network. The proxy communicates with AV/C devices by sending AV/C commands on the IEEE 1394 serial bus.

With this proxy on a network, XFN devices will communicate with AV/C devices as if they were speaking to other XFN devices. For example, an XFN device that seeks to modify the channel on an AV/C device's isochronous input plug, will direct a channel change command to the XFN proxy indicating the target device's XFN node ID. Upon receiving this message, the proxy will parse the command to determine the target XFN node ID. Then the proxy identifies the parameter (multicore input plug channel), and action to be performed on that parameter (for example a 'get' or 'set' command). The proxy will then execute the appropriate AV/C command(s) that will result in a channel change on an input plug.

The AV/C proxy will allow for:

- AV/C device discovery and identification - the proxy will discover AV/C devices on a network and model each AV/C device as an XFN node. Each of the XFN nodes will be uniquely identified by its XFN node ID.

- AV/C device parameter exploration - the proxy will create the necessary parameters that model discovered AV/C devices in accordance with the 7-level parameter hierarchy. These parameters can be explored by other XFN devices on the network.

- Connection management - the proxy will allow a user to set isochronous input and output plug channels, clock (synchronization) sources and sampling rates on an AV/C device by using an XFN connection management application. The connection management application will interact with the AV/C devices by sending XFN messages to the proxy.

- Internal signal routing - routing signals between the input and output plugs that exist on an AV/C device.

- Volume control - allows for volume adjustment on an AV/C device. Each volume control will be modeled as a gain parameter within an XFN node. A gain parameter will be associated with a *deskItem* in an XFN connection management application. DeskItems are explained in section 6.7.6 on page 161.

The above functionality of the AV/C proxy is summarized in the use-case diagram shown below.

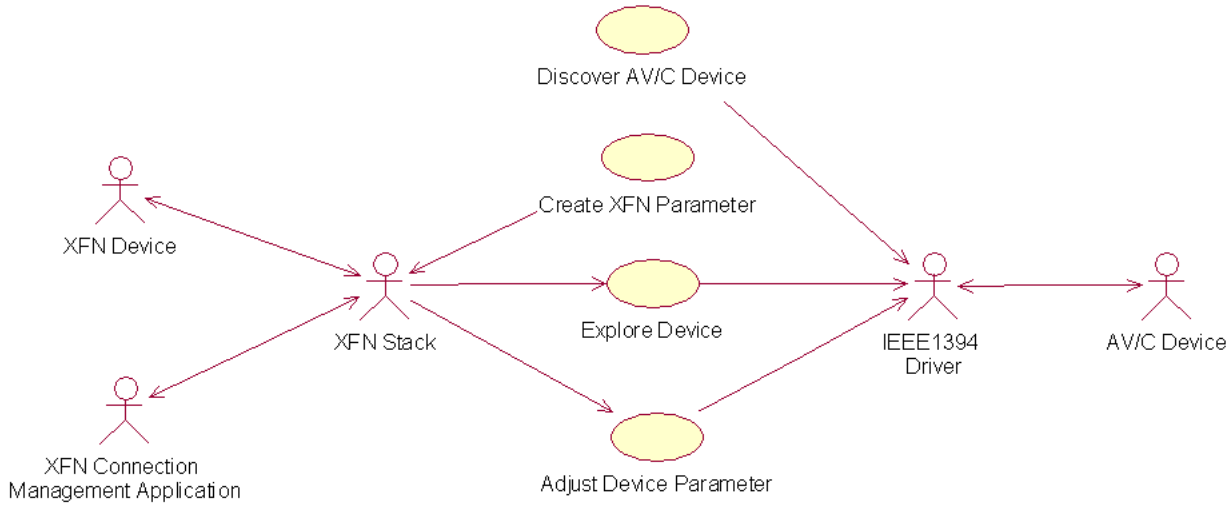


Figure 6.5: Use-case for the AV/C proxy application

The XFN connection management application is capable of:

- discovering XFN devices on a network,
- exploring a discovered XFN device,
- obtaining and monitoring the status of a parameter on an XFN device,
- modifying the value of a parameter,
- synchronizing networked XFN devices by modifying the sampling frequency and clock source of discovered devices.

As far as the XFN connection management application is concerned, all AV/C devices that it discovers with the aid of the proxy are seen as XFN devices. All XFN interactions with the AV/C proxy should be via the XFN stack, and all AV/C interactions with the proxy should be via the

FCP command/response mechanism. An important requirement of the AV/C proxy application is that it should respond to all XFN messages as any other XFN device would.

To fulfill the afore-mentioned requirements, the AV/C proxy interacts with device drivers on its host via various low-level kernel modules. The following section describes the interactions that occur within a workstation running the AV/C proxy.

6.3 AV/C Proxy Overview

The current implementation of the AV/C proxy for network interoperability between XFN devices and AV/C devices utilizes Terratec's Phase 24 firewire-capable breakout boxes, and has been developed for the *Linux*[1] platform.

Platform information:

- *Linux kernel 2.6.24-24-generic*
- *Ubuntu Linux 8.04*
- *libraw1394 version 1.3.0-2*

Figure 6.6 depicts the interaction between the various Linux modules (used by the proxy application), the XFN stack, a Linux firewire library (*libraw1394*) and the AV/C proxy application.

The *ieee1394 bus driver* is a Linux kernel module that manages the interaction between high-level (such as *raw1394*) and low-level (such as *ohci1394*) firewire kernel modules. It is also responsible for managing firewire event triggering within the modules (Linux 1394, 2006). Also situated in the 'Kernel Space' are the *eth1394* and *raw1394* modules that both depend on the *ieee1394* bus driver module. The *eth1394* module provides IP over 1394 support (discussed in section 5.2 on page 99) utilized by the *XFN stack*, which is situated in the 'User Space'. The *raw1394* module provides support for firewire asynchronous and isochronous transactions as well as bus management support for the *libraw1394* library that resides in the 'User Space'. The *XFN stack* and *libraw1394* library both present an application programming interface (API) that allows the *AV/C Proxy Application* in the 'Application Layer' to interact indirectly with the kernel middle-level *ieee1394* bus driver.

[1] *Linux* is a UNIX-based kernel upon which various computer operating systems, such as *Ubuntu*, have been built (Linux Online Inc, 1994).

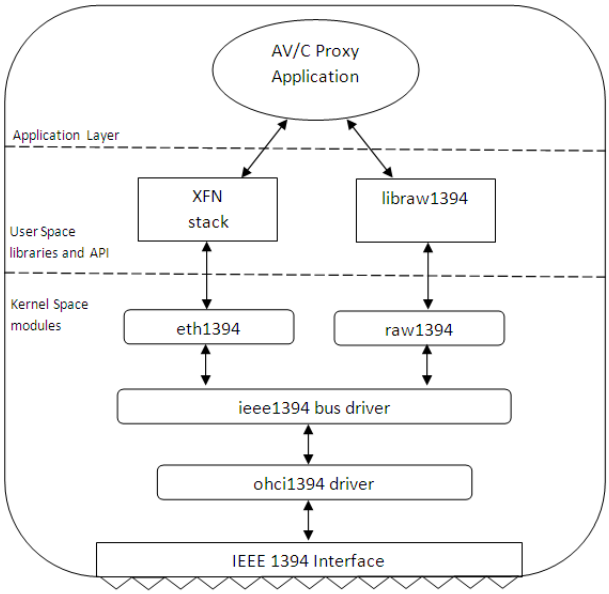


Figure 6.6: AV/C proxy application interaction with Linux modules

The next section describes the nature of XFN nodes that are implemented in the AV/C proxy.

6.4 AV/C device representation in AV/C Proxy

Within the AV/C proxy, AV/C devices are modeled as XFN nodes. Each XFN node is a container that holds the controls (parameters) that exist within a device. An XFN node is uniquely identified by its XFN node identifier. Figure 6.7 depicts the structural layout of a host device running the AV/C proxy.

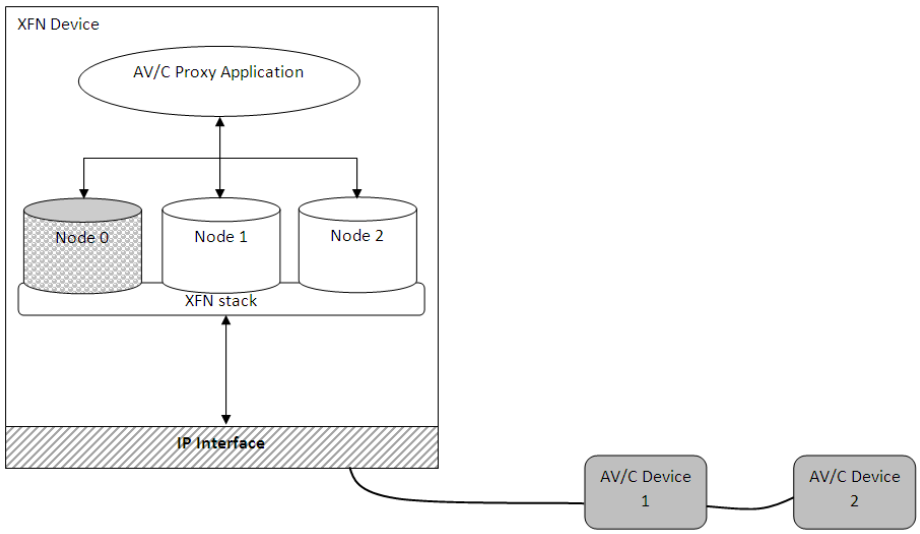


Figure 6.7: AV/C Proxy running on a digital workstation

In figure 6.7, the AV/C proxy is running on a host XFN device that implements the XFN stack. The XFN stack relies on the IP stack of its host for the transmission of XFN messages. The 'AV/C Proxy Application' shown in the figure, discovered two AV/C devices that were networked with the AV/C proxy. For each AV/C device that is discovered, the AV/C proxy application creates an XFN node, identified as 'Node 1' and 'Node 2' in the figure that models an AV/C device in terms of XFN. There is a one-to-one relationship between an AV/C proxy device created within the proxy and a physical AV/C device. A change in the state of an XFN node's parameters (within the proxy) causes a corresponding change in the actual control within the AV/C device modeled by that node. The *internal configuration node* ('Node 0') holds general diagnostic information that pertains to all XFN nodes on the XFN stack. Node 0 is created by the XFN stack during the XFN stack initialization process. Refer to section 5.3.1.1 on page 105 for a discussion of the *internal configuration node* (*Node 0*).

In the current implementation of the AV/C proxy, an XFN_NODE_COUNT parameter has been created within node 0. This parameter exposes the number of XFN nodes, including the internal configuration node, and hence is one node more than the number of AV/C devices modeled by the proxy. A control application, such as a graphical user interface connection management program, is able to determine the number of AV/C devices that the proxy has control over by obtaining the value of this XFN_NODE_COUNT parameter.

The XFN_DEVICE_TYPE parameter is implemented in an XFN device's *internal configuration node*. An XFN_DEVICE_TYPE_PROXY device type has been defined in the XFN stack for

the purpose of the AV/C proxy. This device type parameter is used to indicate that the device running the XFN stack is a proxy. Other possible values of the *device type* parameter include the XFN_DEVICE_TYPE_WIN_PC that indicates the device is a PC running the *Windows* operating system and the XFN_DEVICE_TYPE_ROUTER that indicates the device is a router.

The XFN device discovery process is used to gain knowledge of the XFN devices on a network. During this process, an XFN broadcast message is sent to all XFN devices on the network. The XFN broadcast message is directed at the 'IP Address' parameter within Node 0 of all the XFN devices. When this message is received by an XFN device, it is obliged to respond to the sender of the XFN message indicating it's (the responding XFN device's) IP address. Following this response, a controller (sending device) then seeks to determine the *device type* of each XFN device. If the controller receives a value that indicates that the device is a proxy (XFN_DEVICE_TYPE_PROXY), it becomes its responsibility to obtain the XFN_NODE_COUNT parameter from the proxy.

Any XFN message addressed to an XFN device's parameter specifies the identifier (XFN node ID) of an XFN node (within the XFN device) that 'contains' the parameter. Whenever the proxy obtains an XFN message, it uses the indicated XFN node ID to direct the message to the appropriate XFN node. The XFN node triggers an AV/C command to fulfill the instruction specified within the received XFN message. This AV/C command is directed at the AV/C device modeled by the XFN node. An XFN node refers to a node created above the XFN stack as shown in figure 6.7.

A globally unique identifier (GUID) parameter has been created within each XFN node that models an AV/C device. This parameter enables an XFN device to gain access to a device's GUID, hence making it easy to identify a firewire device. The GUID is also used to obtain a device's firewire node ID after a bus reset. This ensures that if a device was already modeled before a bus reset, the proxy only updates the device's node ID upon bus reset, thus eliminating the need to create the device parameters all over again.

The layout of the AV/C proxy application is shown in figure 6.8 in the form of an object model.

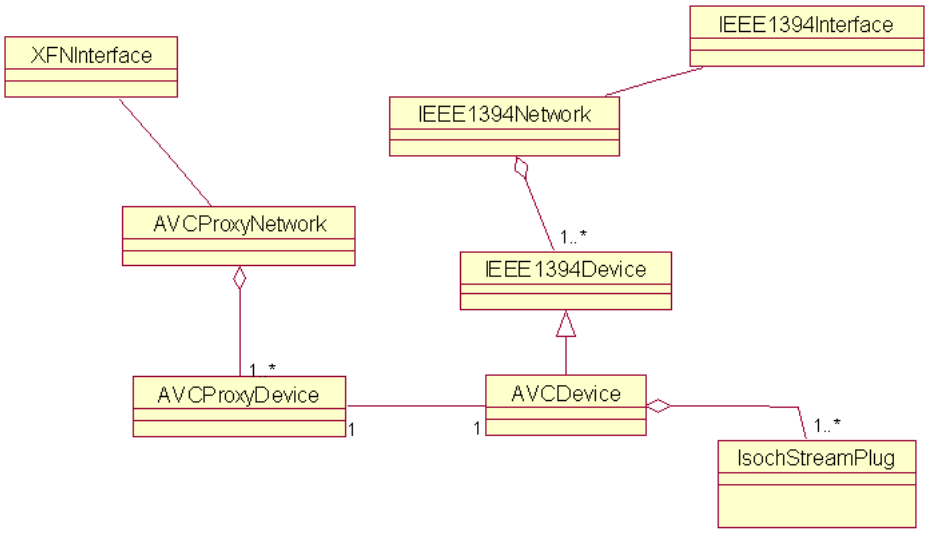


Figure 6.8: Object model of the AV/C proxy application

The *IEEE1394Interface* class abstracts all calls to the *raw1394* bus driver (refer to section 6.3). It initializes the raw1394 bus driver, and obtains information (such as the interface name) about the host computer's firewire interface. All interactions between the proxy and AV/C devices on the network are via the methods defined in this class. All implementations (methods) within this class are platform specific.

The firewire network of devices is abstracted within the *IEEE1394Network* class. The number of firewire nodes, number of AV/C nodes, and the node ID of the isochronous resource manager are stored in this class. The *IEEE1394Network* class also holds a list of all firewire devices on the network.

The *IEEE1394Device* class abstracts firewire devices on the network. Each object of this class is uniquely identified by its GUID (refer to section 3.1.3 on page 35 for description of a device's GUID), and has a firewire node ID.

An object of the *AVCDevice* class represents and abstracts methods of an AV/C device. Each *AVCDevice* object inherits from the *IEEE1394Device* class. Hence, each *AVCDevice* object has a node ID and is uniquely identified by its GUID. It holds a list of isochronous input and output plugs. There is a one-to-one relationship between an *AVCDevice* object and an *AVCProxyDevice* object.

Objects of the *IsochStreamPlug* class abstract the isochronous plugs on a firewire device. Each plug is identified by its plug ID, and is either an input or output plug. An object of this class has

a plug control register (refer to section 3.2.2 on page 49) that holds information about the state of the corresponding isochronous stream plug.

The *AVCProxyDevice* object models a corresponding AV/C device in terms of XFN. Each *AVCProxyDevice* object represents an XFN node, and is uniquely identified by its XFN node ID. It contains the XFN parameters created for the AV/C device it models, and implements the appropriate callbacks for each parameter (refer to section 5.3.1.1 on page 105).

The *AVCProxyNetwork* class creates the *AVCProxyDevice* objects which it stores in a list. The *XFNInterface* class initializes the XFN stack, and is utilized by the *AVCProxyNetwork* class and *AVCProxyDevice* class for XFN related functions.

6.5 AV/C Proxy for AV/C Device Control

Within the AV/C proxy, AV/C devices are modeled as XFN nodes. Each node is a container that holds the parameters that exist on an AV/C device. This implies that the internal nature of the XFN nodes will depend on the particular AV/C device that it models.

An AV/C device possesses various descriptors and info blocks that portrays in a standardized manner the controls and capabilities of the device. This layout of descriptors can have any number of levels from the root descriptor to the leaf descriptor(s) with its associated info blocks. An AV/C *subunit identifier descriptor* and *subunit dependent descriptor* defines a number of list and entry descriptors (list and entry descriptors are explained in section 3.3.3.1 on page 62). However, some of the list and entry descriptors defined in AV/C are optional, meaning that their implementation is dependent on the preference of the manufacturer of the AV/C device. This makes it challenging to determine all of the capabilities and functionality within an AV/C device, by simply parsing the descriptors of an AV/C device's subunit (for example audio subunit).

As a result of the above mentioned challenge, a particular AV/C device was used in this investigation. A firewire - enhanced audio breakout box that conforms to the AV/C protocol, Terratec's Phase 24 (described in chapter 4), was modeled in accordance with the XFN protocol. A process of reverse-engineering was used to determine the various AV/C commands that the Phase 24 responded to. The internal nature of the Phase 24 was obtained, by a combination of responses from AV/C commands and parsing of the AV/C audio and music subunit descriptors. The reverse-engineering involved the use of Terratec's control panel for the Phase 24 to trigger AV/C commands to the device, and then the 'sniffing' of packets that were transmitted on the firewire network.

In these discussions, the procedures and commands that are implemented by (or can be directed to) any AV/C device on the network are indicated as being addressed to an 'AV/C device'. However, commands that depend on a particular implementation of AV/C are described as being implemented by (or directed at) the 'Phase 24'.

The AV/C proxy, implemented in this investigation, modeled the Phase 24. This allowed for communication between Phase 24s and other XFN devices on the same network. This section discusses the various concerns that pertain to the implementation of the AV/C proxy for communication with the Phase 24.

A similar implementation will allow the proxy to communication with other AV/C devices, for example PreSonus *firepod* (PreSonus, 2007). The distinction in the proxy's interaction with other AV/C devices is in the nature of the parameters created within each XFN node. Each device is modeled (as an XFN node) differently depending on the various parameters that it possesses. For example to utilize this proxy for the firepod, the proxy will require knowledge of the parameters within the firepod, and then model those parameters within an XFN node. An XFN node will be created for a firepod whenever the proxy discovers a firepod on the network. It is the created XFN nodes that all XFN messages will be addressed to in order to query or instruct the firepod. This can be achieved for any AV/C device.

6.5.1 AV/C Proxy initialization

The initialization of the AV/C proxy for communication between Terratec's Phase 24 and XFN devices entails that at start up, the AV/C proxy will:

- determine the number of AV/C devices on the network, and for each discovered AV/C device, create an AV/C device object which it places in a list. It determines whether a device is AV/C-compliant by verifying that a device's unit software version indicates compliance with the AV/C protocol. This requires that the AV/C proxy ascertains that a device implements an AV/C unit directory. Refer to section 3.2.1 on page 48 for discussion on how to identify an AV/C device.
- check that each AV/C device object in the proxy's AV/C device list is a Phase 24. It performs this check by reading the model name from the configuration ROM of the AV/C device.

- determine the number of multicores (input and output) for each Phase 24 in the list. It obtains this information by reading the master plug registers (input and output) within the Phase 24.
- create an XFN device object for each Phase 24 device. The XFN device object is created as an XFN node above its XFN stack. This device object will have a one to one relationship with a Phase 24.
- create a Phase 24 parameter tree within the created XFN node. This parameter tree models all the controls within the Phase 24.

The creation of an XFN parameter tree entails the use of the XFN API's methods. To model an AV/C device using the XFN API, the application has to have knowledge of all the parameters that exist on the device - in this context the Phase 24. Once that has been determined, the XFN parameter tree is created using the following procedures:

1. Create an XFN node - each node models a Phase 24 device within the proxy. The XFN node is created using the XFN API's *addXFNDeviceNode* method. This method adds an XFN node above the XFN stack.
2. Create the tree nodes of the 7-level hierarchy and assign an alias to each tree node - the tree nodes form a 7-level hierarchy that is used to address a device's parameter in a standardized manner. Refer to section 5.3.1.3 on page 106 for a description of XFN 7-level hierarchy. The levels are created using the *createXFNLevel* method of the XFN API. Each level is assigned an alias using the XFN API's *setXFNLevelAlias* method.
3. Create parameter tree parent-child relationship - the parent-child relationship used to create the 7-levels that form a parameter tree is established using the XFN API's *addChildXFN-Level* method. This results in the top-level tree node (level 1 node) remaining without a parent node.
4. Add the parameter tree to an XFN node - the 7-level parameter tree is created within an XFN node by adding the top-level (the level with no parent tree node) to an XFN node. This is accomplished with the XFN API's *addLevelToXFNDevNode* method.
5. Create the XFN parameter - the XFN parameter is created with the *createXFNParameter* method of the XFN API. Each parameter is assigned a unique index, and is associated with a callback that implements all possible operations that can be performed on the parameter.

6. Add the parameter to the created 7-level hierarchy - the *addParameterToXFNLevel* method of the XFN API is used to add a parameter to the lowest level of the (7-level) parameter hierarchy.

7. Add the parameter to an XFN node - the XFN API's *addParameterToXFNDevNode* method is used to associate the created parameter with an XFN node.

The above steps are followed for the creation of each of the parameters in an XFN device. In the process of creating device parameters, it is often the case that certain levels that describe successive parameters might already have been created. For example levels 1 to 5 of the 7-levels that describes a particular parameter might have already been created within the same XFN node. In this case, the procedure will be to create levels 6 and 7 (for that parameter), and then link level 6 to the already existing level 5 using the *addChildXFNLevel* method which is used to establish a parent-child relationship.

6.5.2 XFN node for Phase 24

An XFN node created for a Phase 24 abstracts the various controls of the device. Within the AV/C proxy, there is a one-to-one relationship between an XFN node and an AV/C device. The correct node ID is maintained by keeping the globally unique identifier (GUID) of the AV/C device. The relationship between an *AVCProxyDevice* object and an *AVCDevice* object as implemented in the AV/C proxy is shown in figure 6.9.

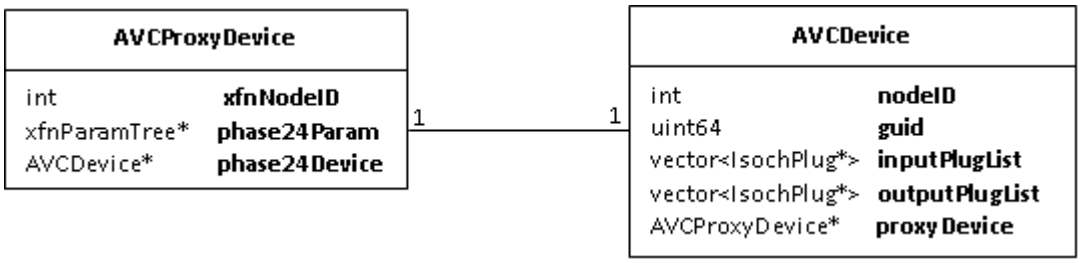


Figure 6.9: AVCProxyDevice object and AVCDevice object relationship

The *AVCDevice* object of figure 6.9 abstracts a physical AV/C device on the network, in this instance the Phase 24. It holds a list of input and output firewire isochronous stream plugs, the device's GUID and its node identifier.

In figure 6.9, the *AVCProxyDevice* object models an *AVCDevice* object in accordance with the XFN protocol. Each *AVCProxyDevice* is uniquely identified by its *xfnNodeID*. A parameter tree (*xfnParamTree*) is created for the AV/C device. This tree will model the controls within the AV/C as parameters in accordance with the XFN 7-level hierarchy for parameter description and addressing. An XML file is also created to model the device specific controls, known in XFN as *deskItems*. For example, the master volume control within the Phase 24 is modeled as a 'gain parameter' whose control is modeled as a *fader deskItem* in an XFN connection management application. This will be discussed further in section 6.7.6.

Upon a firewire bus reset, the proxy updates the:

- Node IDs of each AV/C device in its AV/C device list.
- If a new AV/C device is detected, it is added to the AV/C device list.

In order to update the node IDs of previously discovered AV/C devices, the proxy compares the GUID of an AV/C device to that of the *AVCDevice* object. If there is a match between both GUIDs it assigns the AV/C device's node identifier to the *AVCDevice* object. Otherwise, it creates a new *AVCDevice* object for the discovered AV/C device.

6.5.3 XFN parameter modeling

In the AV/C proxy, the controls within a Phase 24 correspond to parameters in an XFN node within the proxy. The XFN parameters created for the Phase 24 were allocated to the following XFN section blocks (refer to section 5.3.1.3 on page 106 for a discussion of section blocks):

- XFN_SCT_BLOCK_MATRIX - models all parameters necessary for the internal routing of signals within an Phase 24. These include the various audio, digital and isochronous input and output signals. This section block models these signals as a matrix that patches an input signal to an output.
- XFN_SCT_BLOCK_INPUT - models input parameters within the Phase 24. These include parameters that relate to firewire inputs such as the channel on an isochronous input multicore (stream) and the number of audio pins (sequences) on an input multicore, as well as parameters that relate to non-firewire audio inputs such as input gain (volume) parameter.

- XFN_SCT_BLOCK_OUTPUT - models the output parameters within the Phase 24. These include firewire output parameters such as the channel on an isochronous output multicore and the number of audio pins (sequences) on an output multicore, as well as non-firewire audio output parameters such as output audio gain (volume) parameter.
- XFN_SCT_BLOCK_CLOCK - models parameters that are necessary for the synchronization of a Phase 24 with other networked devices. These include its sampling frequency and current clock source.
- XFN_SCT_BLOCK_GLOBAL - holds the globally unique identifier parameter (GUID) of a Phase 24.

Each of the above-mentioned section blocks, form the top-level of the 7-level hierarchy used to describe the parameters within an XFN node. Together, the various entries on the levels of the hierarchy that describe parameters within a node, form a parameter tree. An example of the hierarchy used to describe the isochronous channel on the second input multicore of the Phase 24 is shown in figure 6.10.

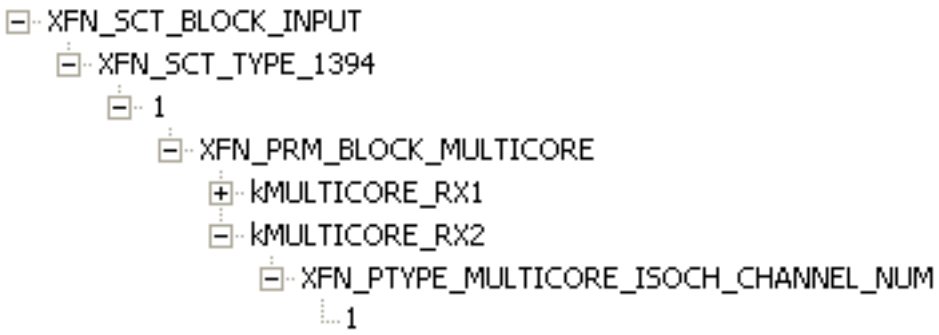


Figure 6.10: XFN parameters for Phase 24 FW

The level ‘names’ shown in figure 6.10 are symbols which have specific values that are defined in the XFN stack. These symbols and their values can be found in the XFN specification (Foss, 2008). An XFN message (refer to section 5.3.3 on page 110) addressed to the isochronous channel depicted in figure 6.10 will have the values indicated in table 6.2.

Level Number	Level Description	Level Alias	Level Value
1	Section Block	XFN_SCT_BLOCK_INPUT	0x01
2	Section Type	XFN_SCT_TYPE_1394	0xE1
3	Channel Number	1	0x000001
4	Parameter Block	XFN_PRM_BLOCK_MULTICORE	0xD1
5	Parameter Block Index	kMULTICORE_RX2	0x000002
6	Parameter Type	XFN_PTYPE_MULTICORE_ISOCH_CHANNEL_NUM	0x0D04
7	Parameter Type Index	1	0x0001

Table 6.2: Full data block address value of an isochronous channel parameter

The parameters created for the Phase 24 are detailed in Appendix D. Each parameter within the Phase 24 is associated with a callback, the callback responds to any XFN message addressed to the parameter. The callback mechanism implemented for the Phase 24 is discussed in the next subsection.

6.5.4 Callback mechanism for the Phase 24's XFN parameters

In XFN, the callback mechanism is used to perform an operation on a device's parameter. A callback could be used to get the value of a parameter or to set a parameter's value. Any message addressed to a parameter causes the callback associated with that parameter to be triggered. The use of the callback mechanism for an AV/C device is illustrated in the flow diagram of figure 6.11.

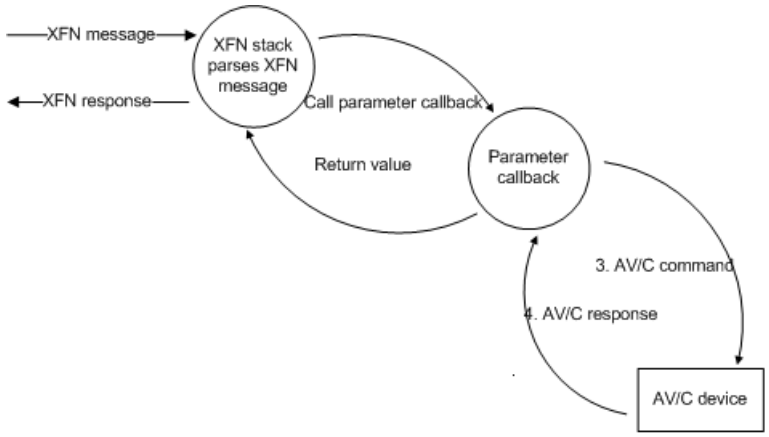


Figure 6.11: Diagram depicting the callback mechanism of an AV/C device

An XFN message is addressed to a parameter within a device. When an XFN message arrives, the XFN stack (of the receiving XFN device) parses the message to determine which amongst its parameters, the message is addressed to. To identify the parameter, the XFN stack firstly determines whether the message contains a:

- full-datablock parameter address - in this case the 7-level parameter hierarchy is used to specify the parameter of interest, or
- indexed parameter address - in which case a unique parameter index is specified in the message. This parameter index is used to identify the parameter of interest.

It then determines the callback of the XFN parameter. The XFN stack also determines whether or not a response is required for the received message. In either case, a callback associated with the parameter is invoked. The callback triggers appropriate firewire asynchronous transactions to fulfill the received instruction. If a response is required, the callback passes the response to its XFN stack.

While the callback requires particular arguments, the actual procedures that it requires to fulfill the instruction it receives are not restricted to XFN. This allows for interoperability between XFN and other networking protocols, since a protocol specific instruction can be embedded within a callback (method). It is this capability that the AV/C proxy capitalizes on to control AV/C devices. As shown in figure 6.11 the *parameter callback* causes firewire asynchronous transactions to be executed.

Figure 6.12 illustrates the XFN parameter callback implementation to change the master volume (gain) on the headphone output of a Phase 24.

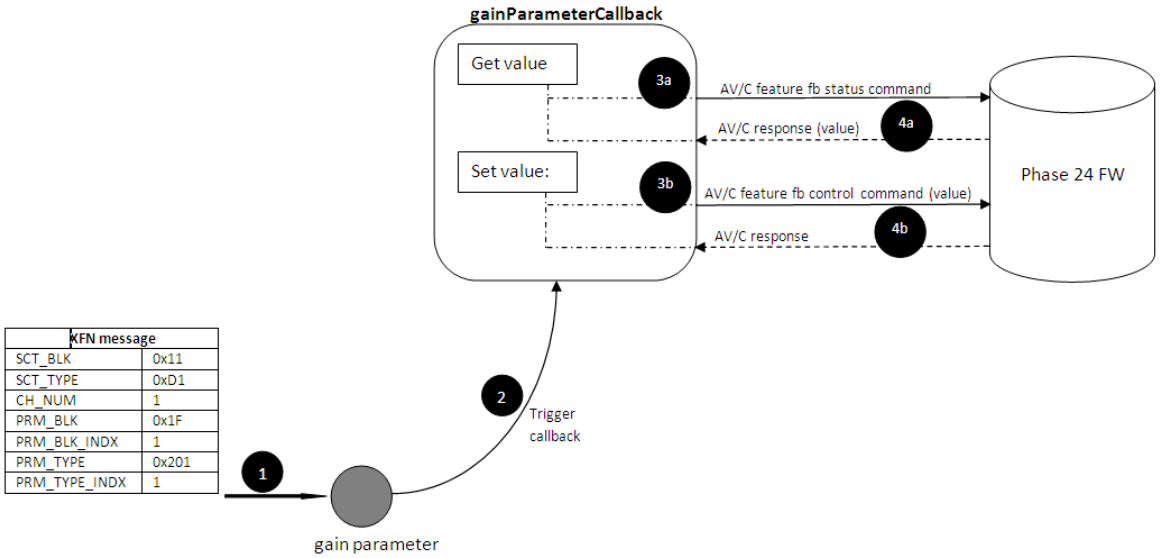


Figure 6.12: Phase 24 gain parameter callback

In figure 6.12 an XFN message is addressed to the gain parameter of the Phase 24 (1). This triggers the 'gainParameterCallback' (2). Depending on the *command executive* and *command qualifier* fields of the XFN packet, it could either be a *Get Value* or *Set Value* message. Refer to section 5.3.2 on page 109 for discussion on command executive and command qualifier fields within an XFN message header.

In the case of a *Get Value*, an AV/C status command is sent to the feature function block within the Phase 24's audio subunit that is responsible for the master volume of the headphone output (3a). In response to this, the Phase 24 sends an AV/C response with the current value of the master volume to the proxy (4a).

If the XFN message is a *Set Value*, an AV/C control command (with the new value as argument) is sent to the appropriate feature function block within the Phase 24's audio subunit (3b). In response, the Phase 24 sends (to the proxy) an AV/C response indicating the status of the received command (4b). The response indicates whether or not the command was successful.

The code synopses of the XFN parameter creation and callback implementation, are shown in figure 6.13 and figure 6.14, respectively.

```
// create an XFN node
// create the tree nodes of the 7-level hierarchy and assign an alias to each tree node
// create parameter tree parent-child relationship
// add the parameter tree to an XFN node
//create the XFN parameter
struct XFN_PARAM_STRUCT* gainParamter = createXFNParameter(paramIndex,valIt,callback,voidPtr);
// add the parameter to the created 7-level hierarchy
// add the parameter to an XFN node
```

Figure 6.13: XFN parameter creation

```
int callback(PVOID data, UInt8 command, UInt8 *valft, UInt32 *valueBufferLength, UInt8 **valueBuffer)
{
        switch (command)
        {
                case XFN_CB_COMMAND_GET_VAL:
                {
                  // valueBuffer = parameter value
                  // valueBufferLength = length of parameter value( in bytes)
                }
                break;
                case XFN_CB_COMMAND_SET_VAL:
                {
                  //parameter value =  valueBuffer
                }
                break;
        }
        return true;
}
```

Figure 6.14: XFN callback implementation

In the next subsection, the layout of an XFN graphical user interface application that can be used to communicate with the AV/C proxy is described.

6.6 The User Interface

An XFN graphical user interface application suite that runs on a *Windows*, *Linux* or *Mac* workstation has been developed as part of the Universal Media Access Network (UMAN) (UMAN, 2009). This application, known as the UMAN connection management application suite (UCMAN), currently allows for:

- discovery of networked XFN devices on multiple network subnets,
- XFN device parameter exploration,
- connection management of networked XFN devices,
- internal routing of signals within an XFN device,
- device specific controls, including gain and metering, using graphical user interface (GUI) controls known as *deskItems*, and
- networked device synchronization, including the manipulation of a device's sampling frequency and clock signal source.

To allow a user to manage the connections and control networked XFN devices, UCMAN presents a number of matrices each with a specific functionality. The various views that can be seen in UCMAN are described below.

6.6.1 Device View

The *Device View* is the initial screen display of UCMAN. It shows all the discovered networked devices, and groups devices within the same network subnet together. Figure 6.15 shows UCMAN's device view window for two XFN devices on the same subnet.

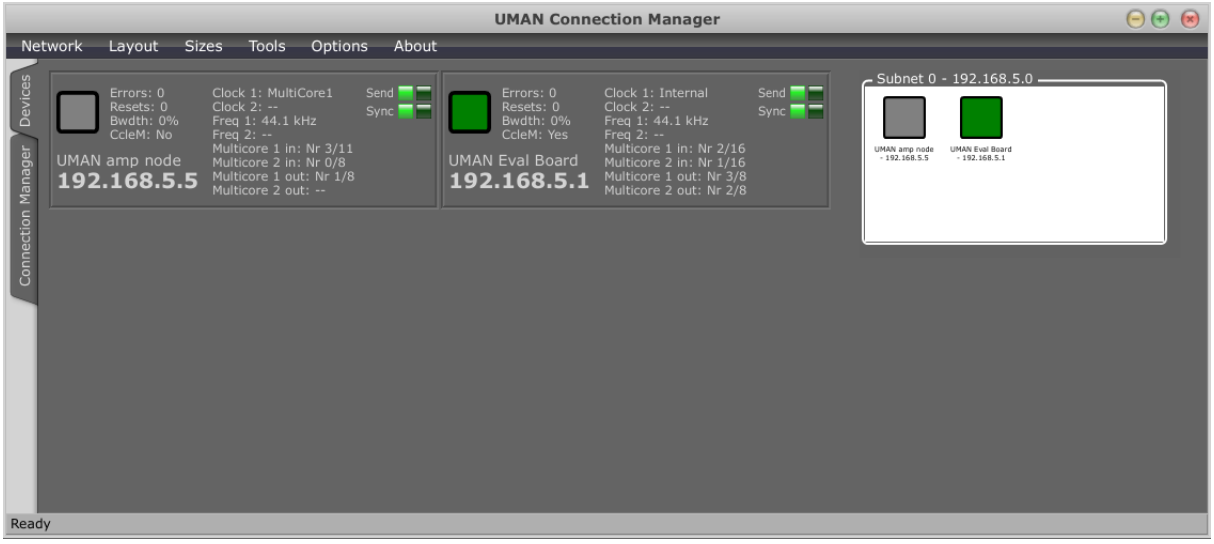


Figure 6.15: UCMAN Device View

To the right (in figure 6.15) is the subnet discovered by UCMAN. In the figure UCMAN has discovered only one subnet (‘192.168.5.0’). To the left are the properties of the devices within a subnet that have been discovered by UCMAN. In figure 6.15, two XFN devices were discovered with IP addresses ‘192.168.5.5’ and ‘192.168.5.1’. The visible properties of the devices include the device’s clock source, multicores, sampling frequency and the channel set on each of its input and output multicores.

6.6.2 Network View

This view contains detailed information about the networked devices per subnet. The network view is shown in figure 6.16.

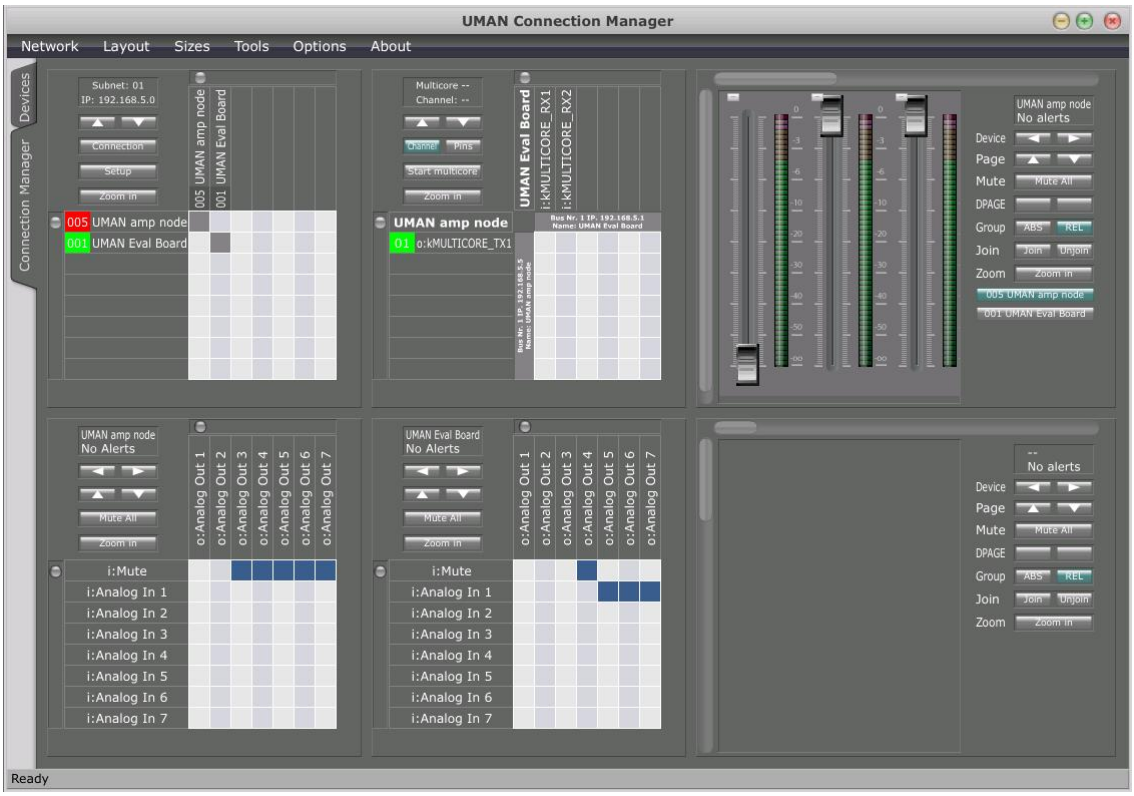


Figure 6.16: UCMAN Network View

The top-left of UCMAN's network view displays the discovered XFN devices on a specific subnet. This is known as the *device matrix* in UCMAN. The top-middle of the network view is UCMAN's *multicore matrix* and it displays the multicores of the devices that have been selected in the *device matrix*. The top-right section of the network view displays the *deskItems* for controls within the XFN device that has been selected in the *device view*. The bottom-left and bottom-middle matrices display the *internal routing matrix* of a source and destination device respectively. A source device is the transmitting device and is listed on the left section of the *device matrix* (top-left matrix). The destination device is the receiving device and is listed at the top section of the *device matrix*. Whenever a cross-point in the *device matrix* is clicked, the source and destination device will change in the other three matrixes to reflect the selected devices. And whenever a source device is selected (in the device view) by clicking on the device's name, its deskItems are loaded in the top-right section (*deskItem area*).

The AV/C proxy has been designed such that each AV/C proxy device responds to XFN messages in a manner similar to any other XFN device on the network. The next section discusses how UCMAN interacts with the AV/C proxy in order to communicate with networked devices.

6.7 AV/C Proxy for Network Interoperability using UCMAN

This section describes the various device controls and communication between XFN and AV/C devices that have been achieved using the AV/C proxy. The XFN communication with networked devices including the AV/C proxy and its proxy devices was accomplished using the UCMAN graphical interface. Figure 6.17 lays out the configuration of the networked devices used in this investigation.

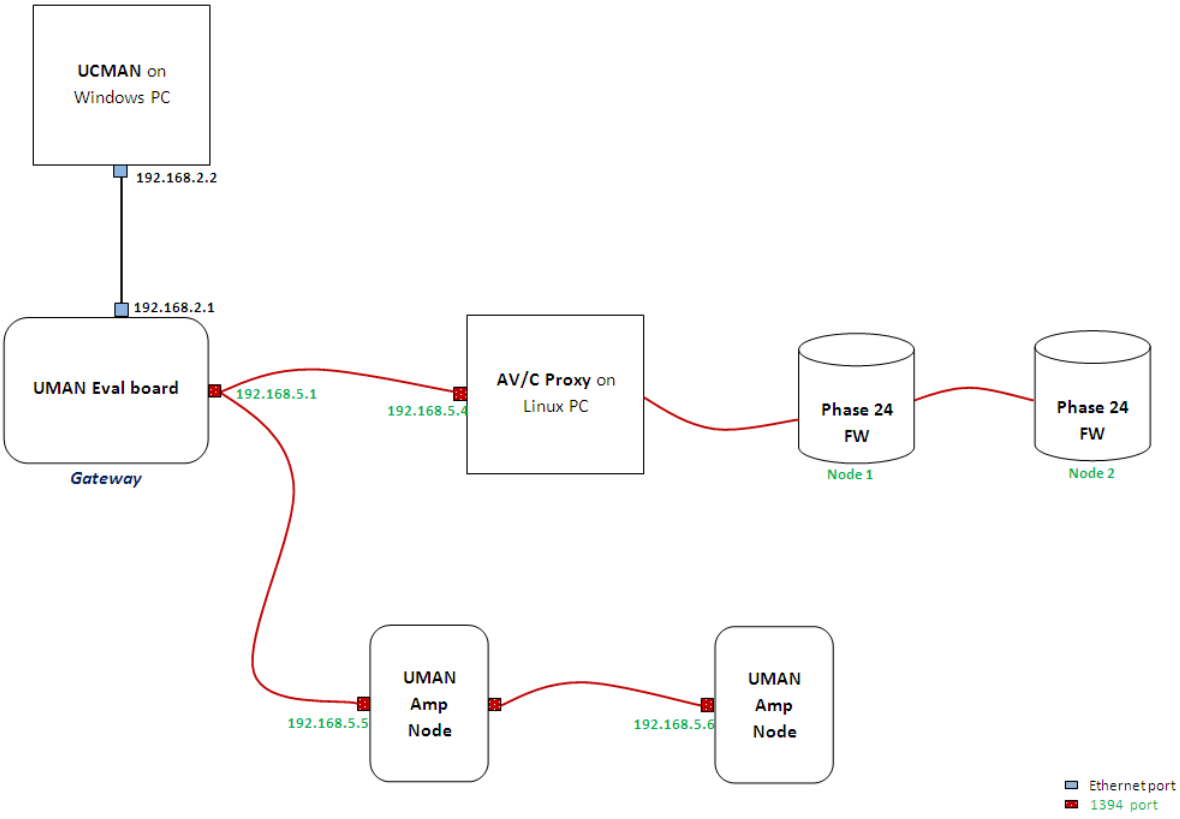


Figure 6.17: Network topology

In the figure, the XFN connection manager and device control application (UCMAN) is running on a *Windows (XP)* PC. It is connected via Ethernet to a *UMAN evaluation board (Eval board)* which serves as a gateway to the firewire network. The Eval board is connected via firewire to the *AV/C Proxy* and to two *UMAN amplifier nodes (amp nodes)*. The AV/C proxy is running on a *Linux (Ubuntu 8.04)* PC and is connected to two Phase 24 FWs. All networked XFN devices and the AV/C proxy reside within the same subnet (192.168.5.0).

The following investigations were carried out with UCMAN using the above network topology:

6.7.1 AV/C device discovery

In the device discovery process, UCMAN sends a broadcast to all networked devices. The AV/C proxy responds with its IP address just like any other XFN device on the network. The discovered devices as seen in UCMAN's device view are shown in figure 6.18.

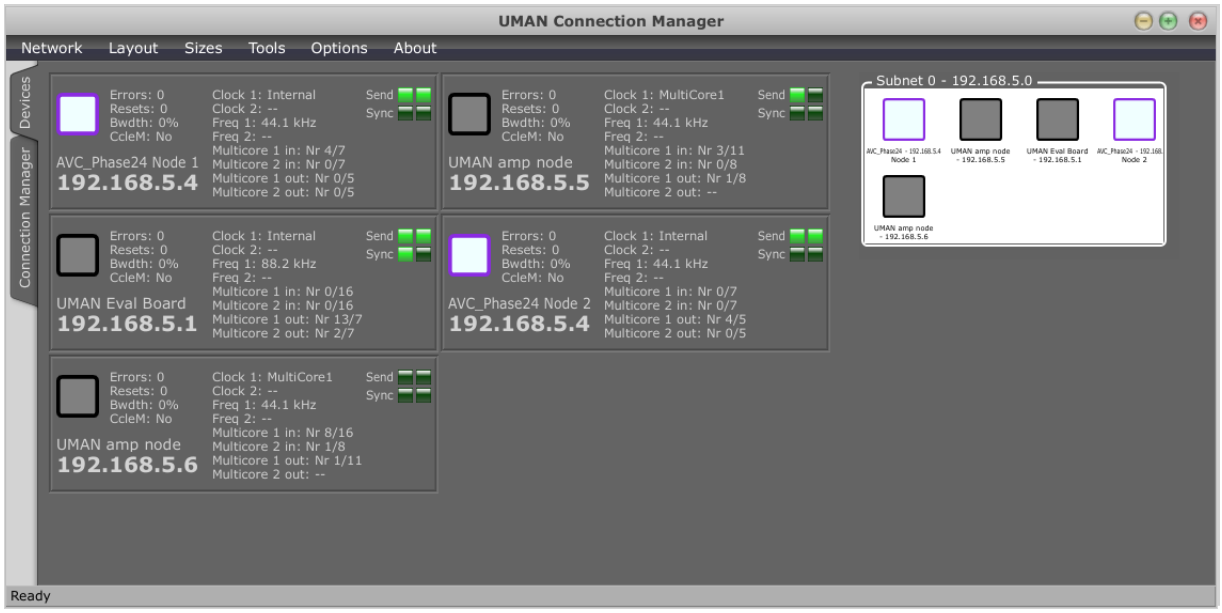


Figure 6.18: UCMAN discovers AV/C proxy devices

For each response to the device discovery message, UCMAN determines the device type in order to appropriately represent it on the graphical interface. If UCMAN receives a device type of XFN_DEVICE_TYPE_PROXY, it then requests the number of XFN nodes from the proxy. By knowing the number of XFN nodes on the network, and hence the number of AV/C devices, UCMAN is able to represent them in its device view as shown in figure 6.18.

The AV/C proxy devices are the purple colored nodes in the device view. Since all of the XFN nodes within the proxy will have the same IP-address, that is the IP-address of the proxy '192.168.5.4', each XFN node can be distinguished by its textual description. In figure 6.18 the XFN nodes are distinctly recognized as 'AVC_Phase24 Node 1' and 'AVC_Phase24 Node 2'.

This AV/C device discovery is represented by the sequence diagram in figure 6.19.

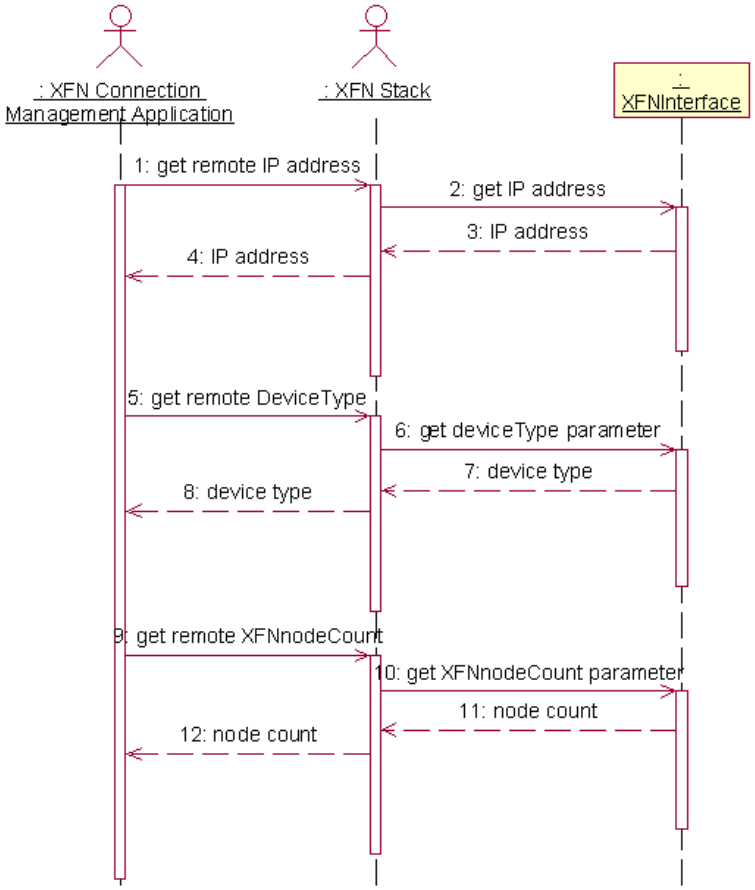


Figure 6.19: UCMAN discovers AV/C devices on the network

The XFN stack is initialized by the AV/C proxy application, and resides on the same host as the AV/C proxy. It is the *XFNInterface* class, within the proxy application, that:

1. initializes the XFN stack,
2. sets the value of the 'nick name' parameter (within the internal configuration node) to 'AVC_Phase24' and
3. sets the value of the 'device type' parameter (within the internal configuration node) to 'XFN_DEVICE_TYPE_PROXY'.

For each Phase 24 ('AVC_Phase24 Node 1' and 'AVC_Phase24 Node 2') seen in the device view of UCMAN, its multicores are visible, each indicating the channel set on it. Also visible for each multicore are the number of audio pins, which is the number of end-points of the audio

sequences. The sampling frequency and clock sources of the Phase 24 can also be seen in the device view.

For each discovered AV/C device, UCMAN obtains the device's GUID. Figure 6.20 shows UCMAN's *device Info* display for AV/C devices. This can be viewed by right-clicking on a device, under 'Device Information' click on 'Display'.

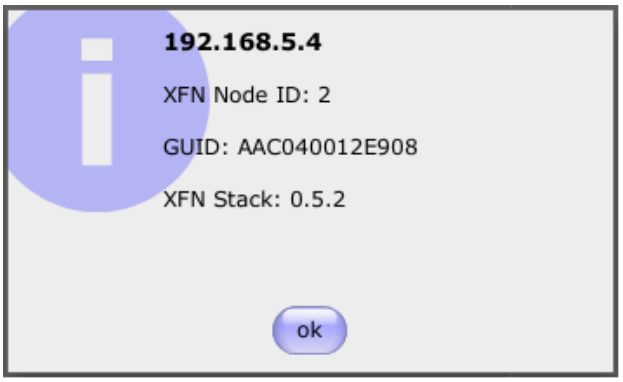


Figure 6.20: UCMAN displays information about a Phase 24

A GUID parameter is created by the proxy for each XFN node that corresponds to an AV/C device. The AV/C proxy determines a device's GUID (from the device's config ROM) when creating the *AVCProxyDevice* and holds its value within the GUID parameter. An XFN message can be sent to the parameter address to acquire the GUID value.

6.7.2 Synchronizing devices

In UCMAN device synchronization is performed in the device matrix. It is here that a user can verify or select a clock source for a particular XFN device. Figure 6.21 shows the current clock source of a Phase 24.

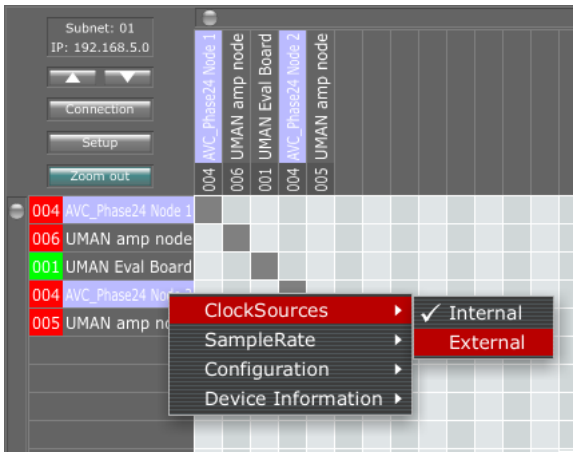


Figure 6.21: Clock source of a Phase 24 AV/C proxy device

UCMAN determines the clock sources of all discovered XFN devices, by sending a 'get clock source' parameter message to each XFN device. As shown in figure 6.21 the Phase 24 is using its internal clock, and will transmit timing information (*time stamps*) within isochronous packets. This ensures that the Phase 24 can be the word clock master, and all other devices on the firewire network should be word clock slaves.

Synchronization of digital audio devices ensures that digital audio is reproduced at the same rate at which it is transmitted. If the networked devices are not synchronized, glitches will be heard in the audio produced on the receiving device. In an extreme case, only clicks are heard on the receiving device. Refer to section 3.1.8 on page 46 for a discussion of synchronization on a firewire network.

The sequence diagram in figure 6.22 depicts the interaction between UCMAN and the proxy, in order to change the Phase 24's sync source.

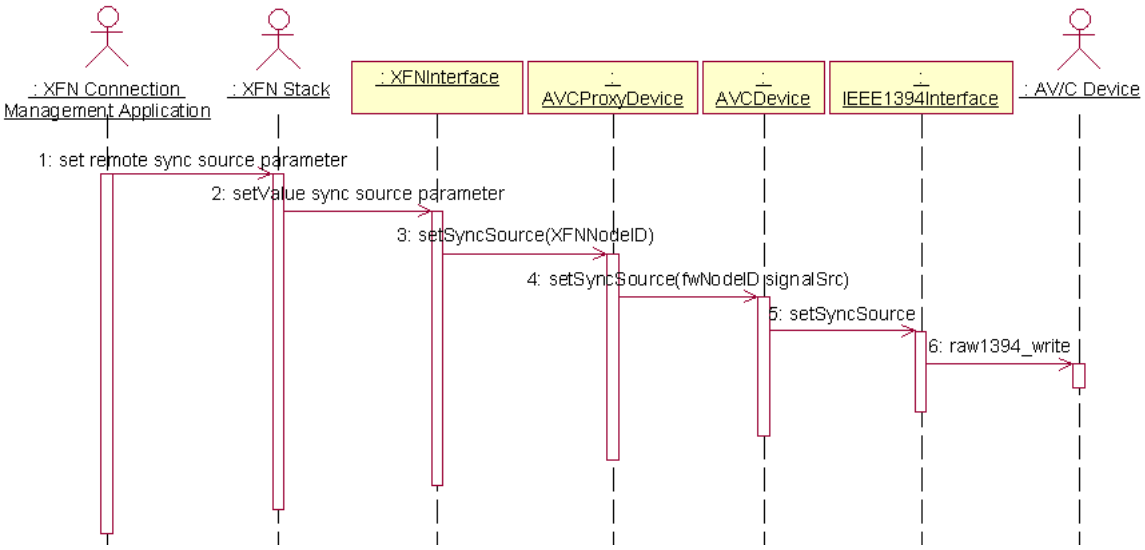


Figure 6.22: Sync source change in Phase 24

In figure 6.22, the connection management application sends an XFN message to change the sync source of a Phase 24. This instruction is parsed by the *XFN Stack* that is utilized by the proxy. From section 6.4 that it is the proxy's *XFNInterface* class that initializes the XFN stack. The received instruction is sent to the 'sync source' parameter callback of the appropriate *AVCProxyDevice* object. The callback is implemented by sending a 'set sync source' AV/C command to a corresponding *AVCDevice* object. This (*AVCDevice* object) in turn performs asynchronous writes to the appropriate AV/C device (Phase 24), via the firewire bus driver implemented in the *IEEE1394Interface* class.

By design, the Phase 24 cannot receive word clock from the firewire bus but rather via S/PDIF. This stems from its role as a breakout box. By default it sets its word clock source to 'internal' and transmits timing information to all other nodes on the network. As a result, multiple Phase 24s on the same network, will always be out of *sync* (not synchronized) if no other provision is made to assign only one Phase 24 as word clock master, and all the others as word clock slaves. A scenario that allows for device synchronization via digital (S/PDIF) signals is described in section 6.9 on page 166.

6.7.3 Changing Sampling Frequency

Changing the sampling frequency on a transmitting device can be performed from the device matrix in UCMAN. Figure 6.23 shows the device view for sample frequency change, as seen on

UCMAN.

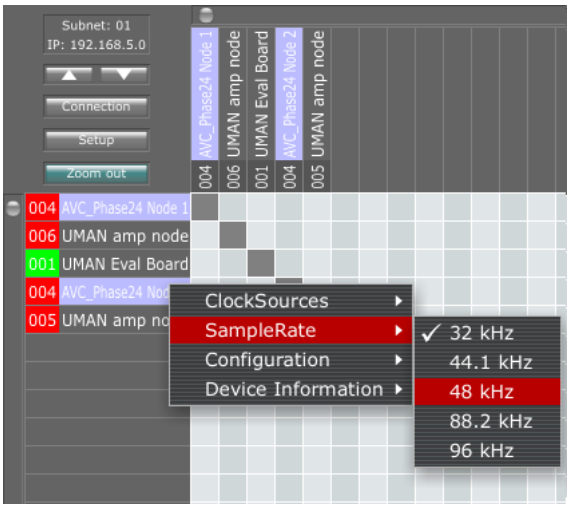


Figure 6.23: Changing the sampling frequency of the Phase 24 AV/C device

In the figure, the ticked sampling frequency is the current sampling frequency of the device. When a frequency value is clicked in the menu shown in figure 6.23, UCMAN sends an XFN message to the AV/C proxy to change the value of the *sampling frequency* parameter value of a specified XFN device. This causes the AV/C proxy to send AV/C control commands to the input and output plugs on the corresponding Phase 24. The Phase 24 device then responds with a success or failure indication of whether its sampling frequency has changed.

The interactions between UCMAN, the AV/C proxy, and the firewire bus driver, are represented in the sequence diagram of figure 6.24.

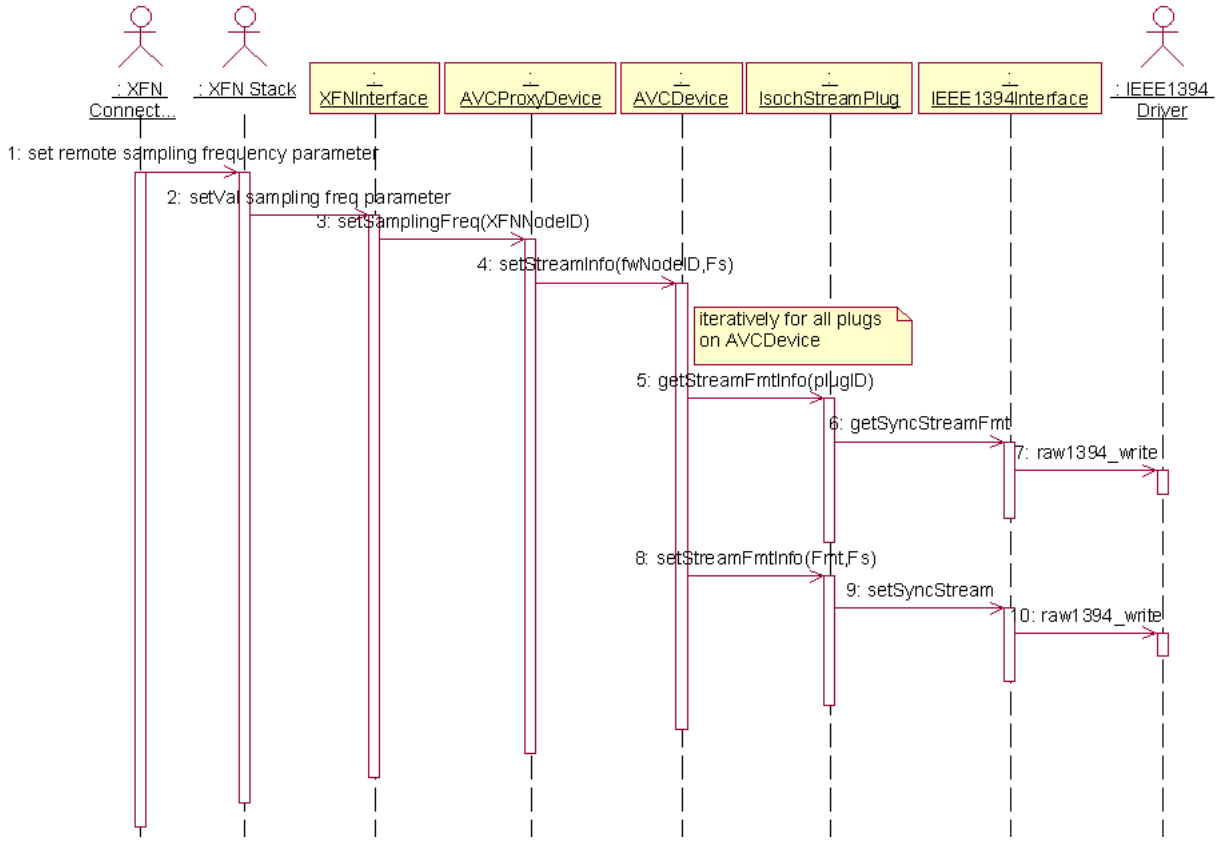


Figure 6.24: Interactions that result in a change in the sampling frequency on a Phase 24

The proxy receives the sampling frequency parameter change request via its *XFN Stack* which is initialized by its *XFNInterface* class. The callback implemented for the 'sampling frequency' parameter within the *AVCProxyDevice* object calls an *AVCDevice* object's *setStreamInfo()* method. For each isochronous stream plug on the AV/C device (Phase 24), the *setStreamInfo()* method:

- obtains the current stream format by sending an AV/C sync stream status command
- sets the sampling frequency to the new value

The AV/C proxy sends *stream format commands* to a Phase 24, in order to obtain or modify the device's sampling frequency. To set the sampling frequency on the Phase 24's input or output plugs, two variant of the *stream format control command* are used. These are the *'AM824 compound single request'* and the *'sync streams single request'* commands, and they are defined in BridgeCo (2007). The *AM824 compound* variant is used to set the sampling frequency on a plug with multiple sequences. Hence in the case of the Phase 24, this *AM824 compound* command

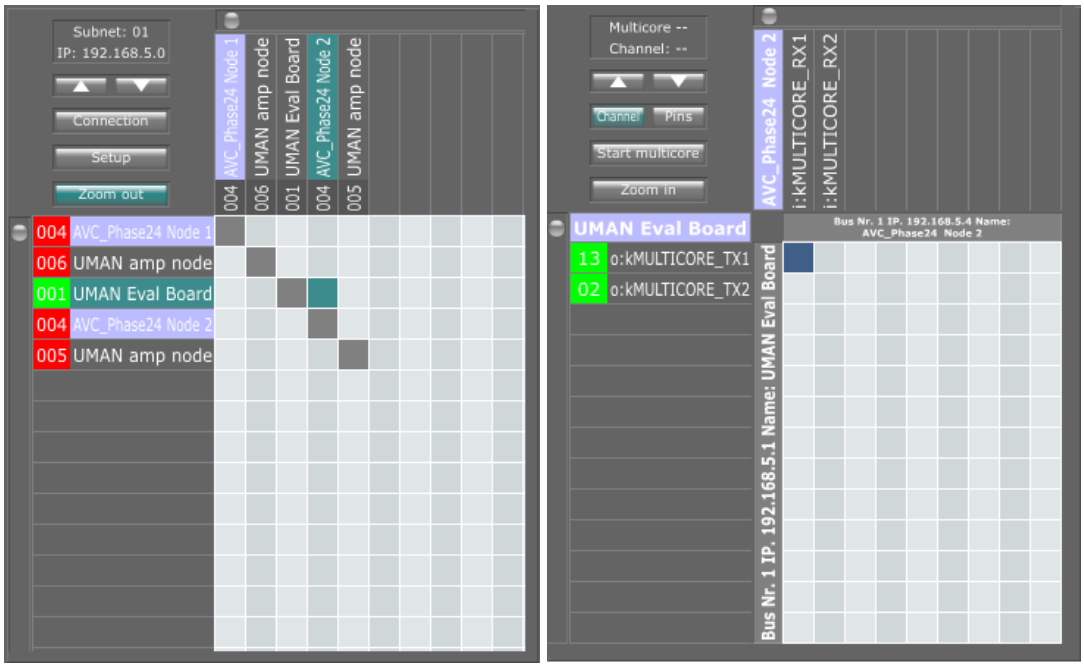
is used for input and output plugs '0', which receive and transmit (respectively) multiple audio sequences.

6.7.4 Making a multicore connection

Connections can be made in UCMAN's network view (network view is described in section 6.6). The device matrix and the multicore matrix are used for making multicore connections in UCMAN. The process followed by a user to stream audio from a source device's output multicore to a destination device's input multicore requires that the user will:

- select a source device and a destination device, by clicking on a cross-point in the device matrix.
- synchronize the source and destination devices in the device matrix by selecting appropriate clock sources for the source and destination device.
- set the sampling frequency of a source device in the device matrix.
- set the number of multicore sequences (audio pins) in the multicore matrix.
- set the transmission channel on the source device in the multicore matrix.
- make a connection between the source device's output multicore and the destination device's input multicore in the multicore matrix.

Figure 6.25 shows the device matrix when a user attempts to select the 'UMAN Eval board' as a source device and the 'AVC_Phase24 Node 2' as the destination device. In the multicore matrix the user has made a connection between the first multicore of the evaluation board and the first multicore of the AV/C proxy device by selecting the appropriate cross point.



(a) Device Matrix (b) Multicore Matrix

Figure 6.25: Making a connection between a UMAN Eval board and a Phase 24 using UCMAN

When a cross point in the multicore view is selected, an XFN message is sent to the AV/C proxy to set the channel on the input multicore of the Phase 24 to that of the source device's output multicore. The AV/C proxy then directs the message to Node 2 which represents a Phase 24 device. As a result, the callback associated with the *isochronous channel* parameter causes an asynchronous write transaction (with the indicated channel value) on the plug control register (PCR) that corresponds to the specified multicore. The asynchronous write sets the *channel* field on the plug control register.

The nature of the transactions to change the multicore channel on the Phase 24 is shown in the form of a sequence diagram in figure 6.26.

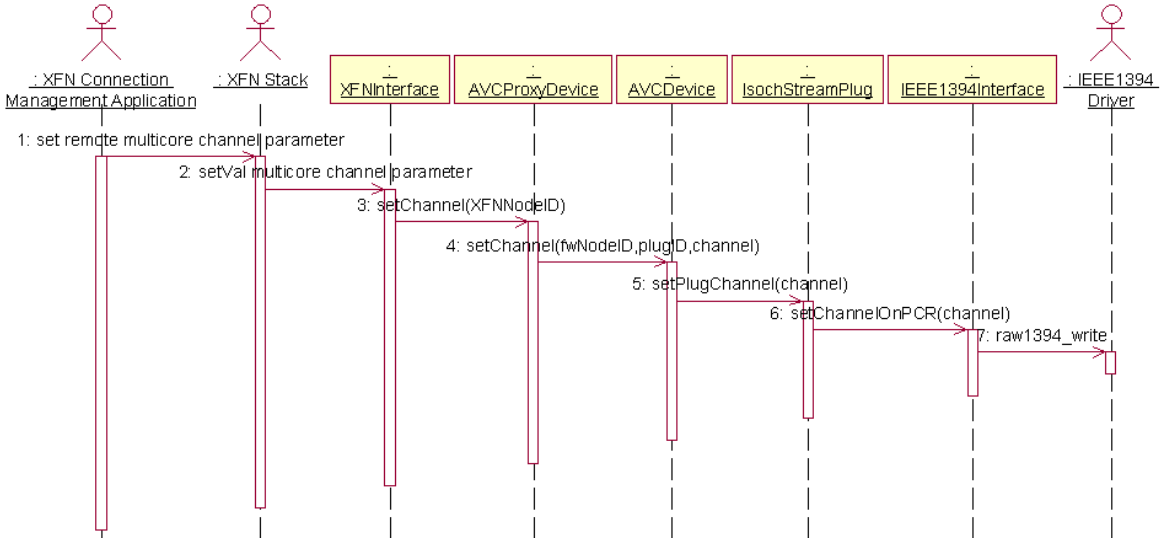


Figure 6.26: Changing the Phase 24's multicore channel via the AV/C proxy

When the proxy receives the 'set multicore channel' parameter instruction from its *XFN Stack*, it executes the 'multicore channel' callback within the *AVCProxyDevice* object. This callback is fulfilled by executing a *setPlugChannel()* method of an *IsochStreamPlug* object within the *AVCDevice* object. The *setPlugChannel()* method calls the *setChannelOnPCR()* method of the *IEEE1394Interface* class (shown in the figure 6.26). The *setChannelOnPCR()* method is implemented by an asynchronous write transaction on the plug's PCR within the Phase 24. This write transaction modifies the *channel* field within the PCR, in order to cause a corresponding change on the channel set on the plug.

If the Phase 24 was to be connected to the Eval board such that the Phase 24 is the transmitting device, the *IEEE1394Interface* class obtains the necessary bandwidth and channel for the isochronous resource manager, before modifying the channel field within its PCR.

For a selected output multicore in the multicore view (refer to figure 6.25 (b)), UCMAN allows a user to start or stop the streaming of data, depending on whether or not the multicore was already streaming. When the 'Start multicore' button (which transforms into a 'Stop multicore' button as shown in figure 6.25 since device in the figure is already streaming) is clicked by a user, UCMAN sends an XFN message to the Eval board instructing it to start the transmission of data.

UCMAN then sets the value of the Eval board's *running state* parameter, such that it indicates that the multicore has started streaming isochronous packets. This is followed by an XFN message

to the proxy instructing it to set the isochronous channel of the input multicore of the destination device (Phase 24) to that of the source device (Eval board).

If the multicore is already streaming and the 'Stop multicore' button is clicked by the user, UCMAN sends an XFN message addressed to the Eval board instructing it to stop transmitting data on the specified multicore, then it modifies the *running state* parameter associated with the multicore to indicate that it is no longer streaming.

In a situation where an AV/C proxy device such as the 'AVC_Phase24 Node 2' is set as the source device, a similar set of procedures as mentioned for the Eval board is followed. The *running state* parameter has also been implemented within the proxy for each XFN node that models a Phase 24 device.

To modify the *running state* of an AV/C device, an asynchronous write transaction is performed on the plug control register that is modeled by that output multicore. This asynchronous write sets the *online*, *broadcast counter* and *point-to-point counter* fields of the output PCR. Refer to section 3.2.2 on page 49 for a description of the various fields of PCRs.

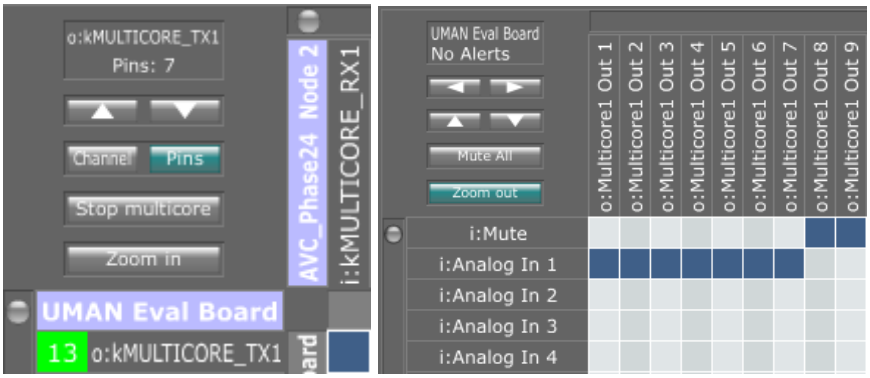
6.7.5 Routing signals

The streaming of audio on a firewire network requires that a source device output the audio on a number of sequences, and that the destination device pick up the audio from the specified sequences. The exact number of audio sequences sent by the source device should be picked up by the destination device. For audio to be streamed by the UMAN Eval board (source device) and received by a Phase 24 (destination device), the Eval board is required to send seven sequences of audio. This is the number of sequences that has been hard-wired into the Phase 24's isochronous input plug. As a result, the number of audio sequences to an isochronous input plug is fixed, and hence cannot be increased or decreased by a remote controller. A similar scenario exists at the output plugs, where the fixed number of audio sequences output by the isochronous plug of a Phase 24, is five.

In XFN, the end-point of an audio sequence is referred to as an *audio pin*. In order, to transmit audio from the Eval board to the Phase 24, the number of active pins on the output multicore is set to '7'. After setting the number of active audio pins to seven, audio is then routed from the audio source to the first seven audio pins of the output multicore.

Figure 6.27(a) shows the number of active audio pins set to '7', on the first output multicore on the Eval board. In figure 6.27(b), the output multicore pins ('Multicore1 Out 1' to 'Multicore1

Out 7') get their input from the analog input signal 'Analog In 1' on the Eval board.



(a) Set active audio pins (b) Output audio through active audio pins

Figure 6.27: UMAN Eval board routing 'Analog In 1' to 'Multicore Outs 1 - 7'

To output the audio signal received in the isochronous stream through the stereo headphone output of the Phase 24, the audio is routed from the isochronous input plug to the headphone stereo output plug of the Phase 24, as shown in figure 6.28.

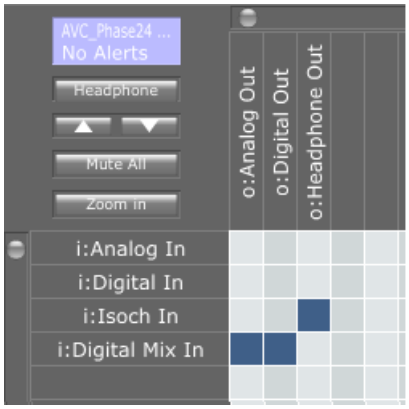


Figure 6.28: Phase 24 AV/C proxy device routing 'Isoch In' to 'Headphone Out'

A similar connection can be made between a Phase 24 as a source device and the Eval board as the destination device. Since the Phase 24 outputs five sequences of audio, the Eval board should be set to receive on five of its input multicore's audio pins.

The various outputs in figure 6.28, can receive input from only one of the input sources. An input-to-output routing is made by clicking on a cross-point between the input and output signals. For example, clicking on the cross-point between the 'Analog In' and 'Digital Out' signals, results

in the signal at the analog input being routed to the digital output plug. The routing matrix is modeled in XFN by associating each output plug with ‘patch input ID’ parameter. The 7-level hierarchy that describes this ‘patch input ID’ parameter is shown in the table 6.3 below.

Level	Description	Alias	Value
1	Section Block	XFN_SCT_BLOCK_MATRIX	0x80
2	Section Type	XFN_SCT_TYPE_AUDIO	0xD1
3	Channel Number	1	1
4	Parameter Block	XFN_PRM_BLOCK_MATRIX_OUTPUT_AXIS	0xD3
5	Parameter Block Index	1	1
6	Parameter Type	XFN_PTYPE_MATRIX_MATRIX_PATCH_INPUT_ID	0x0E01
7	Parameter Type Index	1	1

Table 6.3: 7-level hierarchy of the ‘patch input ID’ parameter

The ‘parameter type index’ value starts at ‘1’ and will depend on the number of outputs. In this case three outputs (Analog, Digital, Headphone) are modeled, hence the ‘parameter type index’ is created with values 1, 2 and 3.

The interactions that results in the internal routing of a signal (within the Phase 24) is represented in the form of a sequence diagram in figure 6.29.

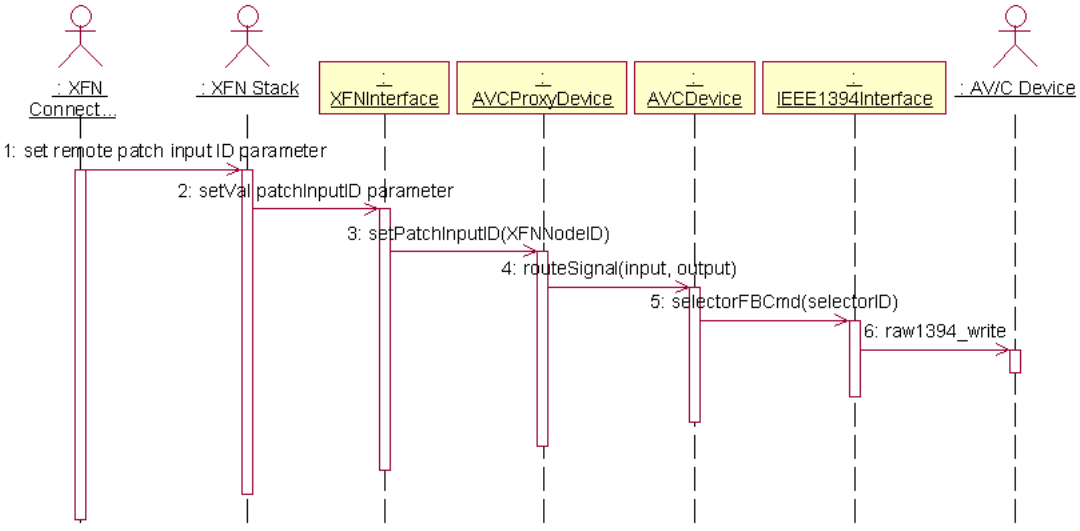


Figure 6.29: Interactions with the AV/C proxy to enable signal routing within the Phase 24

On receiving an XFN message to set the value of a ‘patch input ID’ parameter, the parameter’s

callback within the specified *AVCProxyDevice* object calls on the *AVCDevice* object's *routeSignal()* method. The *routeSignal()* method is implemented by sending an AV/C *selector function block control command* to a specific selector within the Phase 24's audio subunit. This is achieved by calling the *selectorFBCmd()* method of the *IEEE1394Interface* class, which is fulfilled via asynchronous write transactions (libraw's API's *raw1394_write*).

The choice of the particular selector function block that an AV/C command is addressed to, depends on the output plug on the Phase 24. Table shows the output plugs and the corresponding ID of the selector that controls its output signal.

Output Plug	Selector ID
Analog	1
Digital	3
Headphone	2

Table 6.4: Output plug with corresponding control selector ID

6.7.6 DeskItem for Phase 24

In XFN, device specific controls are stored in a zipped file - known as a *deskItem* file - within the device, and can be requested by a remote device. The zipped file (deskItem) contains an:

- xml folder - that contains the XML file that describes the controls within the device
- images folder - that contains the images that could be loaded for the control

The use of deskItems allow a manufacturer to provide (device specific) graphic controls that can be loaded in UCMAN, and each deskItem (in UCMAN) is *joined* with an actual parameter on a remote device.

The XFN protocol allows for grouping of parameters, by a process referred to as a *join*. When two or more parameters are joined, within the same or different devices, an XFN command to modify one of the parameters also adjusts all the other parameters in the (join) group. To facilitate this, each parameter keeps a list of all other parameters it is joined with (Foss, 2008).

A 'volume gain' parameter, that is specific to the Phase 24, was created. This parameter resides within the (AV/C) proxy's XFN node and it models the master volume control of a Phase 24. To

manipulate this device specific control, UCMAN loads the *deskItem* file that control the 'volume gain' parameter.

For the Phase 24, an XML was created to model the 'volume gain' parameter. Figure 6.30 shows the XML file used to load a fader in UCMAN.

```
<UMANFADER name="Line In" XFNid="" XFN_Level_1="0x01" XFN_Level_2="0xD1"
XFN_Level_3="1" XFN_Level_4="0x1F" XFN_Level_5="1" XFN_Level_6="0x201"
XFN_Level_7="1" Control_Param_ID="2" id="15fcaa301b7990fd" explicitFocusOrder="0"
pos="8 7 44 296" backimage_plain="" backimage_plainrect="0 0 0 0"
backimage_scaled="" backimage_scaledrect="0 0 0 0" trackimage="" trackimagerect="0 0
0 0" thumbimage="" thumbimagerect="0 0 0 0" minValueDB="-60" maxValueDB="0"
xfnUnitRangeMinValueDB="-120" xfnUnitRangeMaxValueDB="18" trackX="16"
trackY="16" trackHeight="263" trackWidth="12" thumbX="10" thumbHeight="42"
thumbWidth="23" topScaleYPos="24" bottomScaleYPos="272" backWidthWithScale="55"
backWidthNoScale="44"/>
```

Figure 6.30: XML for a fader deskItem

The XML of figure 6.30 describes the *UMANFADER* with an alias 'Line In'. It describes the 7-level hierarchy of the parameter it controls, with the *'XFN_Level_1'* to *'XFN_Level_7'* attributes. The image associated with this deskItem is specified with the *'backimage_plain'*, *'trackimage'*, and *'thumbimage'* attributes, representing the image-background, the slider-track, and the thumb-slider images, respectively. Each deskItem is uniquely identified by its *'Control_Param_ID'*.

A modification of the deskItem parameter, results in a corresponding change in the value of the 'volume gain' parameter. Figure 6.31 shows a fader in UCMAN's deskItem area that models the Phase 24's master volume control.

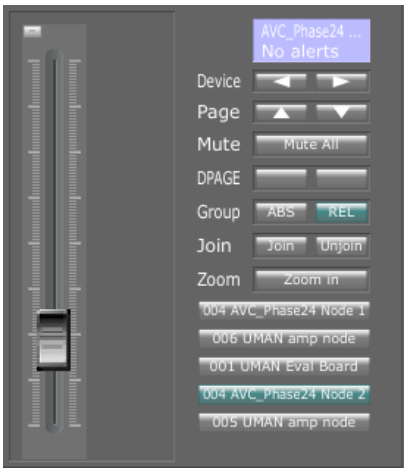


Figure 6.31: DeskItem for Phase 24's master volume control

When the Phase 24 AV/C (proxy) device is clicked in the device matrix, UCMAN loads the fader shown in figure 6.31. This fader deskItem is intended to control the master volume of the Phase 24. Whenever the value of the fader is adjusted in UCMAN, an XFN message is sent to the appropriate XFN node within the AV/C proxy. The proxy then sends an AV/C control message to the feature function block within the Phase 24's audio subunit that is responsible for master volume control. The value of the fader sent by UCMAN is in terms of *XFN units*. It is the responsibility of the AV/C proxy to convert the XFN units to values that will cause a corresponding gain change on the Phase 24.

An XFN unit provides a standardized unit for parameter values, and is defined in the XFN specification (Foss, 2008). An XFN device can define an *XFN units table* which it uses to convert the XFN units it receives, into (its) parameter dependent units. For example, if three parameters A, B, and C residing in three different XFN devices, are joined. All three devices will have created their own XFN units table. If the deskItem that controls A is adjusted, the XFN unit value received by A is relayed to B and C as well. It is possible that these three parameters have different parameter dependent units; A might use decibel unit (dB), B millisecond unit (ms), and C hertz unit (Hz). When a parameter receives an XFN unit value, it will convert the XFN unit to its parameter unit, then makes the necessary parameter adjustment.

In the implementation of the AV/C proxy for communication and control of Terratec's Phase 24 FW, a number of restrictions prevented absolute XFN device control and parameter manipulation. These challenges and constraints are mentioned in the next section.

6.8 Implementation Constraints

There were several constraints encountered in the implementation of the AV/C proxy for Terratec's Phase 24 enhanced breakout box. These constraints restricted a number of XFN controls over the device parameters. They include:

- *Fixed number of audio pins* - there is a fixed number of audio pins that is hard wired into the Phase 24. For any of the Phase 24's input isochronous plugs, exactly seven audio pins (stream sequences) can be picked up by the plug. For any of its output isochronous plugs, exactly five audio pins are transmitted by a Phase 24. Hence a device wanting to transmit to the Phase 24 must send seven sequences of audio on a particular channel. Any device that wishes to receive audio from the Phase 24, can only receive audio on the five sequences that are transmitted by the Phase 24.

- *Fixed stream format* - the audio format of a Phase 24's isochronous stream sequences (audio pins) are fixed. Although the audio format of the audio pins conforms to *AM824* they cannot be altered by a controller. AM824 is a format for packaging audio and MIDI data and is defined in the IEC 61883-6 specification (2005). The ordering of AM824 stream formats expected by a Phase 24's isochronous stream plug cannot be altered. For example, the Phase 24 outputs five sequences with MIDI at stream position 5. The output plug will not transmit the MIDI in any other stream position.

  Table 6.5 and table 6.6 show the fixed audio format of the Phase 24's input and output streams.

Pin Number	**AM824 Format**
1	IEC 60958 Conformant
2	Multi-bit Linear Audio (MBLA)
3	IEC 60958 Conformant
4	Multi-bit Linear Audio (MBLA)
5	Multi-bit Linear Audio (MBLA)
6	Multi-bit Linear Audio (MBLA)
7	MIDI

Table 6.5: AM824 format of a Phase 24's input audio pins

Pin Number	AM824 Format
1	Multi-bit Linear Audio (MBLA)
2	Multi-bit Linear Audio (MBLA)
3	Multi-bit Linear Audio (MBLA)
4	Multi-bit Linear Audio (MBLA)
5	MIDI

Table 6.6: AM824 format of a Phase 24's output audio pins

In order to transmit audio from the UMAN Eval board to the Phase 24, audio was routed to the first seven sequences of the transmitting output multicore. The Eval board can extract the audio from the second sequence of the received multicore audio pins.

- *Imposes clock master role* - a Phase 24 insists on being the clock master whenever it is 'purely networked' with other firewire devices. 'Purely networked' via firewire implies that no other digital synchronization is input through its S/PDIF digital input plug. Hence each Phase 24 will insist on using its internal clock as its clock source. As a result of this, multiple Phase24s on a firewire network will have their clocks out of phase. The direct impact of this is that the audio heard from any of the Phase 24's will have glitches in audio. However a Phase 24 can be connected via S/PDIF to another clock source ad have its clock set to external.

- *Audio streaming between multiple Phase24s* - it is currently not possible to stream audio from one Phase 24 and have another Phase 24 pick up the audio. This is due to the following:

  - the Phase 24 expects a fixed audio format in its input isochronous stream plug, and this audio format is different from that of its output isochronous stream plug. Compare the audio format of the input pins and output pins shown in table 6.5 and table 6.6.
  - Phase 24s insist on using their internal clocks as clock source and hence all want to be clock master.

These factors disallowed more than one Phase 24 to be on the same firewire network. The next section proposes a remedy for synchronizing multiple Phase 24s using the AV/C proxy.

6.9 Clock synchronization between multiple Phase24s on a network

Terratec's Phase 24 enhanced breakout box is shipped with a control panel for single Phase 24 device control. On installation of the control panel, the Phase 24 firewire driver is installed in the *Windows XP* platform. The installation of the control panel and its associated drivers are 'tight-bound' to a Phase 24. This prevents the control panel from gaining control over any other Phase 24 (with a different GUID). A further restriction is that each Phase 24 insists on being the word clock master by using its internal clock and transmitting time stamps in the SYT field when connected only via firewire (refer to section 3.1.8 on page 46 for a discussion on clock synchronization).

Figure 6.32 lays out a connection scenario that provides for device synchronization of multiple Phase24s on the same network.

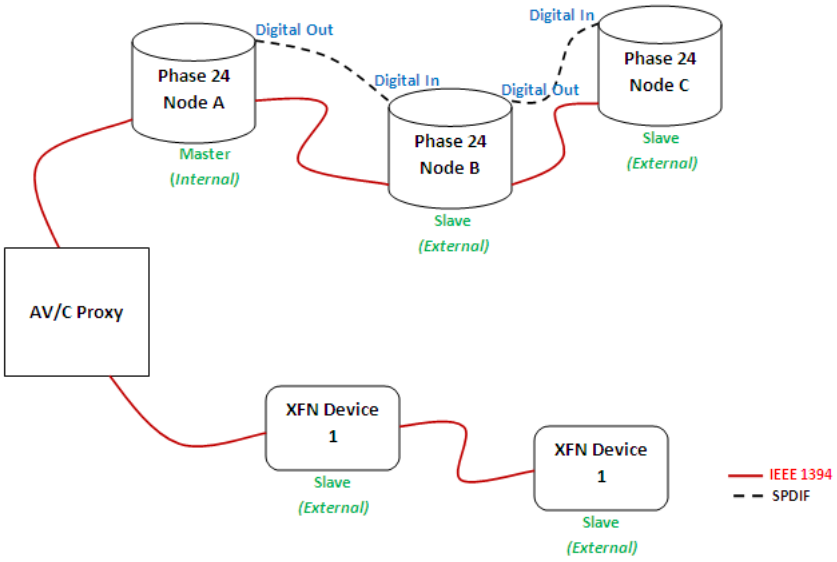


Figure 6.32: Clock synchronization for multiple Phase24s on a firewire network

In the figure, the XFN devices and the Phase24s communicate via the proxy. One of the Phase24s fulfills the role of clock master and sets its clock source to 'Internal'. The others (Phase24s) are slaves that use the clock signal obtained from their digital input plugs and hence their clock source is set to 'External'. The external clock signal is derived from the word clock master. The XFN devices in the above configuration would set their clock source to receive from a multicore,

hence using the time stamps transmitted by the clock master (Phase 24) device. This can only be achieved using the AV/C proxy, since Terratec's control panel prohibits such a connection in which more than one Phase 24 is on the same bus. In this case, the proxy is used to specify the clock source of each of the networked Phase24s - one set to 'Internal', the others to 'External'.

6.10 Summary

This chapter highlighted the benefits of modeling an AV/C device as an XFN node. These include:

- ease of exploring a device's parameters in XFN
- transport layer independence of the XFN protocol makes it easy for AV/C device controls to be manipulated by an IP-messaging protocol
- authentication and user privilege security features of XFN can be implemented for AV/C device controls.

An AV/C proxy was developed for interoperability between an AV/C device (Terratec's Phase 24) and any XFN device. The various design considerations for the implementation of the AV/C proxy were discussed. These include the:

- AV/C proxy initialization procedure
- relationship between an AV/C proxy device object and a Phase 24
- nature of the parameters created for the Phase 24
- callback mechanism implemented for the Phase 24 device's parameters

The AV/C proxy is capable of translating any XFN message addressed to a Phase 24's parameters. These XFN messages could be to:

- discover AV/C proxy devices (Phase24s) on the network
- make connections between an XFN device's isochronous stream plugs (multicores) and a Phase 24's multicores

- change the channel on a Phase 24's multicore
- change the sampling frequency on a transmitting Phase 24
- select a word clock source for a Phase 24
- adjust the master output volume (gain) on a Phase 24 using deskItems

An XFN connection management graphical interface, UCMAN was described with emphasis on how it can be used to communicate with a Phase 24 via the AV/C proxy.

A number of constraints that prevent full XFN control of the Phase 24 were discussed. Core to these constraints is that the Phase 24 does not fully comply with the AV/C procedure and limits the possible AV/C commands implemented in the device.

A connection scheme that will allow for more than one Phase 24 to be synchronized on a firewire network was proposed. This scheme entails the use of the proxy to set one of the Phase 24's as word clock master and the others as word clock slaves.

Chapter 7

Conclusion

This investigation was aimed at providing a solution that allows for communication between professional audio devices that conform to different digital audio network protocols. The proposed approach involved the use of a proxy that will allow for high-level protocol messaging between the networked devices and the proxy. The proxy approach was investigated in this work using two candidate protocols (AV/C and XFN), and has proven to be a practical solution that allows for communication between networked devices of disparate protocols within the context of professional audio.

An introduction into the world of networking professional audio devices, revealed that there exist a range of digital audio network protocols. Each protocol has its own message structure and defines a particular procedure for connection management, device synchronization, and parameter addressing. The protocols that were described include: Open Sound Control (OSC), Architecture for Control Networks (ACN), Common Control Interface for Networked Audio and Video Products (IEC 62379), Audio Video Control (AV/C), music Local Area Network (mLAN), Application Protocol for Controlling and Monitoring Audio Devices Via Digital Data Networks (AES 24), and the XFN protocol.

While some of these protocols are research oriented and open (for example OSC), others are proprietary (for example mLAN), while still others are not widely used (IEC 62379). The AV/C protocol is available in commercial professional audio devices, and its standards documents are accessible from the 1394 Trade Association (1394 Trade Association, 2009). The XFN protocol was accessible to the researcher by virtue of the researcher's affiliation with the Audio Engineering Research Group at Rhodes University, which is involved in the development and standardization of the XFN protocol. The XFN resources available to the researcher included:

XFN evaluation boards, amplifer nodes, and documentation. Since there was access to AV/C and XFN devices, these two protocols were chosen as candidates for this investigation.

The creation of a proxy for interoperability between AV/C and XFN devices required a practical understanding of both protocols. Such an understanding considered how:

- devices are modeled
- parameters within devices are addressed
- protocol messages are structured
- synchronization is achieved

With regards to AV/C, of particular interest were AV/C devices that implement the music and audio subunits. The AV/C music subunit has been described as responsible for the routing of firewire audio signals, while the AV/C audio subunit with its various function blocks is capable of performing several signal routing and non-firewire signal processing function within an AV/C device.

It was mentioned that an AV/C device presents a structured layout of its internal capabilities and functionality using descriptors and info blocks. Hence, a controller is able to explore an AV/C device by parsing the device's unit and subunit descriptors with their associated info blocks. This exploration of the internal nature of a device gives the controller knowledge of the capabilities of device it is interacting with, as well as knowledge about what functionality can be performed by the device. The various AV/C subunit specifications define the nature of the particular subunit's identifier and status descriptors. Some of the descriptors and info blocks defined in AV/C subunits, for example the audio subunit, provide various alternatives to the structuring of a subunit's descriptors. These alternatives are meant to allow for flexibility in the implementation of subunit descriptors. However, this flexibility makes it challenging for a controller to determine the internal nature of an AV/C device's subunit by parsing the device's descriptors and info blocks. Typically controllers hold some prior knowledge of a manufacturer's implementation which eases the parsing of a device's descriptors.

The existence of a range of alternatives in the implementation and structural layout of an AV/C device's descriptors and info blocks, required that a specific AV/C device be used for this investigation. The AV/C device used was Terratec's Phase 24 FW, particularly because it was available in the audio engineering lab (Rhodes University) at the time this investigation was conducted.

The Phase 24 (as it is commonly called) is a firewire enhanced breakout box that implements a music subunit, and an audio subunit. The internal nature of the Phase 24 in terms of its subunits, plugs, and routing was obtained by a combination of parsing the descriptors and info blocks, and sending various AV/C status commands. In the course of this investigation, it was discovered that there exist several AV/C constraints in the Phase 24's implementation of AV/C. These include:

- Fixed routing in the music subunit. This makes it impossible for a controller to modify connections and music plug attributes, by sending AV/C music subunit control commands to the Phase 24.
- Fixed number of sequences on unit input and output plugs. The Phase 24 implements a fixed number of sequences on its firewire input and output plugs. As a result, a controller is unable to make any modifications to the number of sequences received or transmitted by the Phase 24.
- Fixed input and output stream formats at the isochronous stream plugs. Each sequence of the Phase 24's input and output plugs has a defined AM824 stream format.

Similar constraints exist on other music subunit implementations within AV/C devices, for example PreSonus *firepod* (PreSonus, 2007). It would be desirable to have a common controller that can efficiently parse descriptors and info blocks, and control any AV/C device that implements the music subunit. Being able to parse any AV/C music subunit implementation, such a controller will be capable of interacting with any firewire AV/C device irrespective of the manufacturer's design preference. However, such a controller does not currently exist.

The proxy implemented in this investigation modeled AV/C devices as XFN application nodes. The concept of an XFN application node allows any application or device that utilizes the XFN stack to create a node above the XFN stack. The node created is referred to as an application XFN node, and it contains the parameters that pertain to the application or device that implements the XFN stack. The XFN stack itself creates an initial node, referred to as the internal configuration node. It is the internal configuration node that holds generic information, which comprises parameters, and that are necessary for diagnostics and device identification. Such information includes: IP-address, firewire device GUID, XFN device name, and XFN device type.

In XFN, parameters are modeled in a 7-level hierarchy, from a general grouping to a more specific functionality-based definition of the parameter. A controller is able to explore the parameters within an XFN device by sending various XFN messages that return the 7-level parameter

definition. The 7 levels can be used to address a device's parameter. Various XFN messages defined by the XFN protocol allow a remote controller to enquire about and change the state of a parameter. It is also possible to assign values to parameters using the appropriate XFN command.

In the implementation of the proxy for AV/C devices, each XFN node that was created modeled a particular AV/C device. Hence, the signal routing capability, volume control, and isochronous plug channels were modeled as XFN parameters. This allowed an XFN controller to ascertain the nature of an AV/C device by exploring the parameters within the corresponding XFN application node. However, the current implementation of the XFN stack does not present the XFN node ID of each XFN node to a remote controller. Furthermore, the isolation of certain device information, such as device IP-address and device GUID, from the application node implied that the controller was required to obtain such information from the internal configuration node. Hence a controller is unable to determine the number of AV/C devices that the proxy interacts with, and it is also unable to determine the GUID of each AV/C device. The number of AV/C devices gives an indication of the possible node IDs of the application nodes created by the proxy, hence making it possible to direct an XFN message to each node. The GUID is necessary since it uniquely identifies an AV/C device.

The current implementation of the proxy approached the above mentioned constraints by creating:

- an XFN 'node count' parameter, that exposes (to a remote controller) the number of application nodes created by the proxy
- a read-only GUID parameter within each application node, that holds the AV/C device's GUID. This enables the determination of the node ID after a bus reset.

As a result, when a controller determines that a device running the XFN stack is of type 'proxy', it is required that it determines the XFN node count in other to know the number of AV/C devices that the proxy interacts with. However, it is desirable to have each application node possess its own IP-address parameter and XFN node ID parameter. Further investigation can be conducted into this modeling structure. This will enable each application node to respond to an XFN device discovery broadcast message in the same manner as any other XFN device.

The proxy created in this investigation is capable of receiving XFN messages on behalf of AV/C devices, and then fulfilling those messages in terms of AV/C. It proved to be capable of fulfilling the following requirements for multiple Phase 24s via XFN commands:

1. Multicore discovery - entailed the discovery of the available isochronous stream plugs on networked devices.

2. Channel change on multicores - makes it possible to set the channel on a device's isochronous stream plug. It was also possible to determine the channel set on a multicore via the proxy.

3. Sampling Frequency determination and variation - makes it possible to obtain the supported sampling frequencies of a networked device, and to change the current rate at with data is sampled by the device.

4. Synchronization signal source change - is able to determine and set the word-clock source of a networked Phase 24.

5. Internal routing - makes it possible to modify the internal routing within networked networked Phase 24s. It was also possible to determine the current signal routing within the Phase 24.

6. Device specific parameter controls - makes it possible to determine and modify Phase 24 specific parameters, with the assistance of proxy. One such parameter is the volume control within the Phase 24.

The above capabilities form the core functionality of the Phase 24 as a firewire enhanced breakout box.

The proxy approach presented here entails high-level abstraction of devices, with each abstraction modeling a device according to a preferred network protocol. In such a scenario, all communication involves the transfer of protocol messages on the network. The proxy acts as a translator that receives a protocol message on behalf of a target device. It then translates the received message into a message that conforms to the same protocol as the target device. The common controller used to communicate with the networked devices is required to be of the same protocol as the device abstractions. This enables the device abstractions, within the proxy, to understand the messages they receive from the controller.

The implication of this study is that with the aid of a proxy, a common controller can be used to configure parameters that enable communication between devices of different digital audio network protocols. And hence, network interoperability is achieved.

While this study has investigated two digital audio network protocols, it is possible to create such a proxy between and amongst any other audio network protocols, particularly if they are both

hierarchical. Furthermore, it is possible to use this proxy approach for any number of protocols on a digital audio network.

In professional audio, digital audio networks are becoming a necessary solution for large scale audio distribution. A proxy can be used to network devices that conform to different protocols, and this could result in enormous benefits as a range of professional audio devices conforming to varying protocols can be connected to contribute to the audio system. With the use of a proxy, the user's preference for audio devices can be met, since device communication restrictions are eliminated by the proxy. Professional audio devices of different network protocols can be integrated into a single controller network.

References

1394 Trade Association. *"TA Document 1999008: AV/C Audio Subunit Specification 1.0"*, 2000a.

1394 Trade Association. *"TA Document 1999027: Configuration ROM for AV/C Devices 1.0"*, 2000b.

1394 Trade Association. *"TA Document 1999045: AV/C Information Block Types Specification Version 1.0"*, 2001a.

1394 Trade Association. *"TA Document 2001012: AV/C Digital Interface Command Set General Specification Version 4.1"*, 2001b.

1394 Trade Association. *"TA Document 1999025: AV/C Descriptor Mechanism Specification Version 1.0"*, 2001c.

1394 Trade Association. *"TA Document 2002010: AV/C Connection and Compatibility Management Specification 1.1"*, 2003.

1394 Trade Association. *"TA Document 2004007: AV/C Music Subunit Specification 1.1"*, 2005.

1394 Trade Association. "1394 Trade Association Press Release". 2007. URL `http://www.1394ta.org/press/TAPress/2007_1212.html`. Accessed 17 August 2009.

1394 Trade Association. *"1394 Technology - Specifications"*, 2009. URL `http://www.1394ta.org/Technology/Specifications/specifications.htm`. Accessed: 17 August, 2009.

Anderson, D . *"FireWire System Architecture"*. Mindshare Inc., New Jersey, 2nd edition, 1999.

ANSI. "Serial Bus Protocol 2 (SBP-2)". American National Standards Institute - ANSI, 1998.

Audio Engineering Society. *"AES24-1999: AES standard for sound system control - Application protocol for controlling and monitoring audio devices via digital data networks - Part 1: Principles, formats, and basic procedures"*, 1999.

Birk, A . "Firewire - Future Applications", 2003. URL `http://www.faculty.iu-bremen.de/birk/lectures/PC101-2003/13firewire/super_fast_net_ip_1394.htm`. Accessed: 27 April, 2009.

Braden, R . "Requirements for Internet Hosts - Communication Layers". RFC 1122 (Standard), October 1989. URL `http://www.ietf.org/rfc/rfc1122.txt`. Updated by RFCs 1349, 4379.

BridgeCo. *"Additional AVC commands: AV/C Unit and Subunit - SDD Products"*, 2007.

Chigwamba, N and Foss, R . *"An Investigation into the Hardware Abstraction Layer of the Plural Node Architecture for IEEE 1394 Audio Devices"*. SATNAC, 2007.

Comer, D . *"Internetworking with TCP/IP"*, volume 1. Prentice-Hall, 5th edition, 2006.

Deering, S and Hinden, R . "Internet Protocol, Version 6 (IPv6) Specification". RFC 2460 (Draft Standard), 1998. URL `http://www.ietf.org/rfc/rfc2460.txt`. Updated by RFC 5095.

Dolphin Music. "Terratec PHASE 24 FW Upgraded with mLAN Capability", 2005. URL `http://www.dolphinmusic.co.uk/article/238-terratec-phase-24-fw-upgraded-with-mlan-capability.html`. Accessed: 27 April, 2009.

ESTA. "Welcome to the DMX-512 mini-FAQ ". Entertainment Services and Technology Association - ESTA, 1996. URL `http://www.lighting-association.com/links/dmx-faq.htm`. Accessed 18 May 2009.

ESTA. "Entertainment Technology - Architecture for Control Networks". Entertainment Services and Technology Association - ESTA, 2005a.

ESTA. "Entertainment Technology - Architecture for Control Networks. Device Description Language". Entertainment Services and Technology Association - ESTA, 2005b.

ESTA. "Entertainment Technology - Architecture for Control Networks. Device Management Protocol". Entertainment Services and Technology Association - ESTA, 2005c.

Foss, R . *"Audio Engineering - Computer Science Honours Level Course Notes"*. Rhodes University, Department of Computer Science, 2007.

Foss, R . *"XFN Protocol Overview"*. Universal Media Access Networks, 2008.

Fujimori, J and Foss, R . *"A New Connection Management Architecture for the Next Generation of mLAN"*. Presented at the 114th Audio Engineering Society Convention, Amsterdam, 2003.

Gross, K . *"Digital Audio Distribution Systems"*. Cirrus Logic, Inc, 2004.

IEC. *"ISO/IEC 13213: Information Technology - Microprocessor Systems - Control and Status Registers (CSR) Architecture for Microcomputer Buses"*. International Electrotechnical Commission - IEC, 1994.

IEC. *"IEC 61883-6: Consumer Audio/Video Equipment - Digital Interface - Part 6: Audio and Music Data Transmission Protocol"*. International Electrotechnical Commission - IEC, 2nd edition, 2005.

IEC. *"IEC 62379-1: Common Control Interface - Part 1: General"*. International Electrotechnical Commission - IEC, 1st edition, 2007.

IEC. *"IEC 61883-1: Consumer Audio/Video Equipment - Digital Interface - Part 1: General"*. International Electrotechnical Commission - IEC, 3rd edition, 2008.

IEEE. "Standard for a High Performance Serial Bus". IEEE Std. 1394, 1995.

IEEE. "Standard for a High Performance Serial Bus - Amendment 1". IEEE Std. 1394a, 2000.

IEEE. "Standard for a High Performance Serial Bus - Amendment 2". IEEE Std. 1394b, 2002.

IEEE. "Standard for a High Performance Serial Bus - Amendment 3". IEEE Std. 1394c, 2006.

IEEE. "IEEE Registration Authority". Institute of Electrical and Electronics Engineers - IEEE, 2009. URL `http://standards.ieee.org/regauth/index.html`. Accessed: 27 April, 2009.

ISO. "About ISO ". International Organization for Standardization - ISO, 2009. URL `http://www.iso.org/iso/about.htm`. Accessed 18 May 2009.

Johansson, P . "IPv4 over IEEE 1394". RFC 2734 (Proposed Standard), 1999. URL `http://www.ietf.org/rfc/rfc2734.txt`.

Klinkradt, B . *"XFN Stack Overview"*. Universal Media Access Networks, 2007.

Linux 1394. *"IEEE 1394 for Linux"*. 2006. URL `http://www.linux1394.org`. Accessed: 27 April, 2009.

Linux Online Inc. *"What is Linux?"*, 1994. URL `http://www.linux.org/`. Accessed: 27 April, 2009.

Okai-Tettey, H . *"High Speed End-To-End Connection Management in a Bridged IEEE 1394 Network of Professional Audio Devices"*. PhD thesis, Rhodes University, Grahamstown, 2005.

Olifer, N and Olifer, V . *"Computer Network - Principles, Technologies and Protocols for Network Design"*. John Wiley & Sons, 2006.

Pohlmann, K . *"Principles of Digital Audio"*. McGraw-Hill Professional, 5th edition, 2005.

Postel, J . "User Datagram Protocol". RFC 768 (Standard), 1980. URL `http://www.ietf.org/rfc/rfc768.txt`.

Postel, J . "Internet Protocol". RFC 791 (Standard), 1981a. URL `http://www.ietf.org/rfc/rfc791.txt`. Updated by RFC 1349.

Postel, J . "Transmission Control Protocol". RFC 793 (Standard), 1981b. URL `http://www.ietf.org/rfc/rfc793.txt`. Updated by RFCs 1122, 3168.

PreSonus. *"FP10 - 10x10 Firewire Recording System"*, 2007. URL `http://www.presonus.com/products/Detail.aspx?ProductId=3`. Accessed: 2009.07.04.

Raw Materials Software. *"JUCE Overview"*, 2005. URL `http://www.rawmaterialsoftware.com/juce/`. Accessed: 27 April, 2009.

Tanenbaum, A . *"Computer Networks"*. Prentice Hall PTR, 4th edition, 2003.

Teener, M . "1394 Standards and Specifications Summary", 2006.

TerraTec. *"Description - PHASE 24 FW"*, 2004. URL `http://ftp.terratec.net/Producer/PHASE/PHASE24FireWire/TechnicalData/PHASE24FireWire_TechnicalData_GB.pdf`. Accessed: 27 April, 2009.

TerraTec. *"Sound Products"*, 2009. URL `http://linux.terratec.de/sound_en.html`. Accessed: 27 April, 2009.

Tsegaye, M . "A Comparative Study of the Linux and Windows Device Driver Architectures with a focus on IEEE1394 (high speed serial bus) drivers". Master's thesis, Rhodes University, 2002.

UMAN. "Universal Media Access Network", 2009. URL `http://umannet.com/`. Accessed: 27 April, 2009.

Washburn, K and Evans, J . *"TCP/IP Running a Successful Network"*. Addison-Wesley, 2nd edition, 1996.

Wright, M . "Open Sound Control 1.0 Specification". 2002. URL `http://opensoundcontrol.org/spec-1_0`. Accessed: 18 May, 2009.

Wright, M . "Open Sound Control: an enabling technology for musical networking". volume 10, pages 193–200, 2005/12/01 2005. URL `http://cnmat.berkeley.edu/publications/open_sound_control_enabling_technology_musical_networking`. Accessed: 18 May, 2009.

Yamaha Corporation. *"mLAN-1.0 Connection Control Specification - Draft Version 0.5 (Rev. 47)"*, 2000. Confidential.

Yamaha Corporation. *"mLAN-1.0 Transporter Specification"*, 2002. Confidential.

Appendix A

IEEE 1394 asynchronous packets

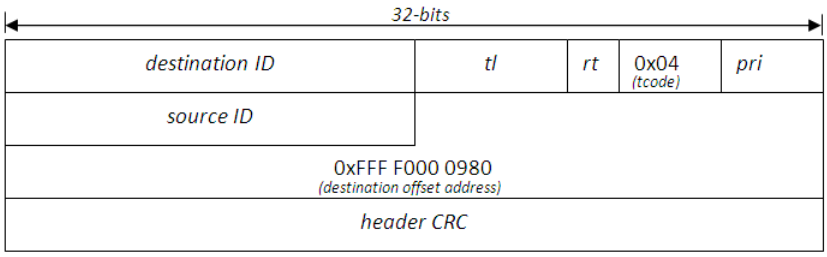


Figure A.1: IEEE 1394 asynchronous read packet for input master plug register (iMPR)

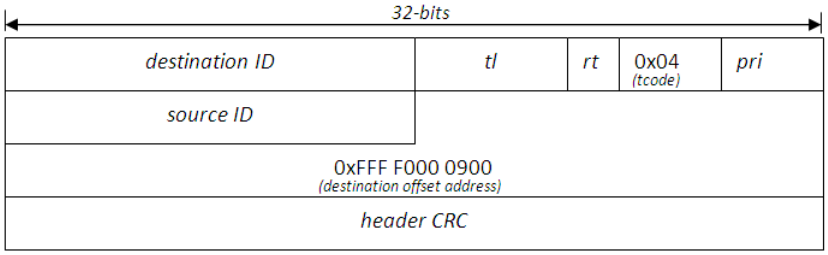


Figure A.2: IEEE 1394 asynchronous read packet for output master plug register (oMPR)

Appendix B

Relevant AV/C commands

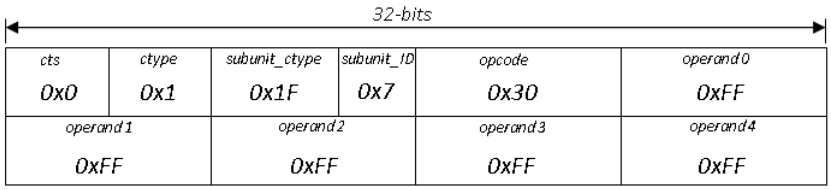


Figure B.1: Unit Info status command

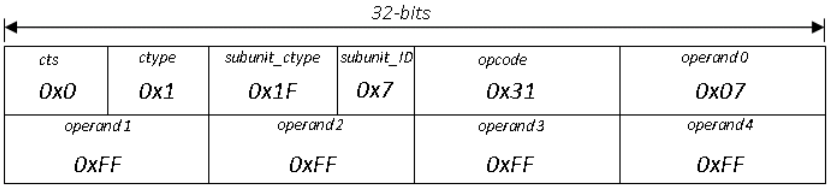


Figure B.2: Subunit Info status command

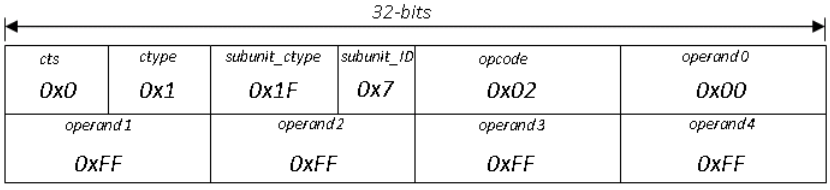


Figure B.3: Plug Info status command addressed to unit

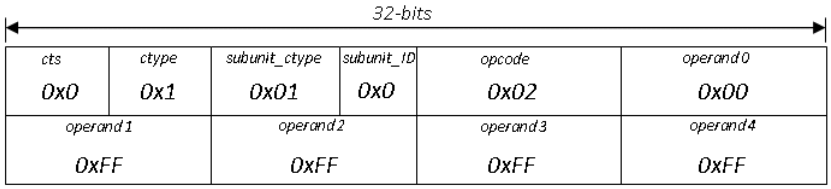


Figure B.4: Plug Info status command addressed to audio subunit with ID ‘0’

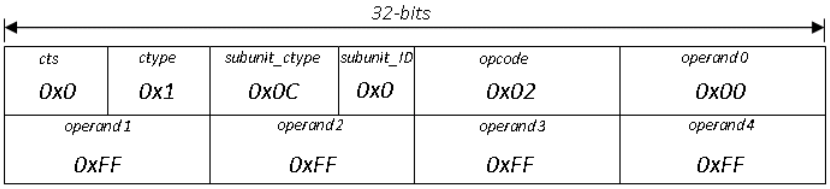


Figure B.5: Plug Info status command addressed to music subunit with ID ‘0’

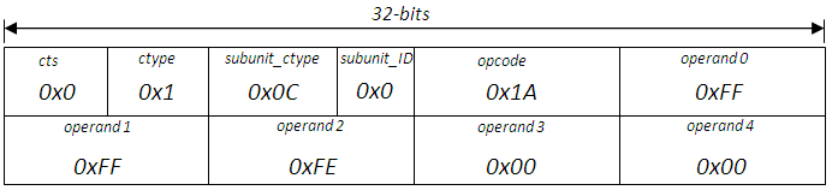


Figure B.6: Signal Source status command addressed to music subunit with ID ‘0’

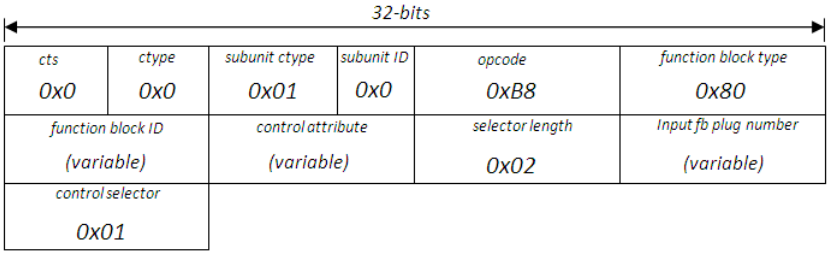


Figure B.7: Selector function block control command

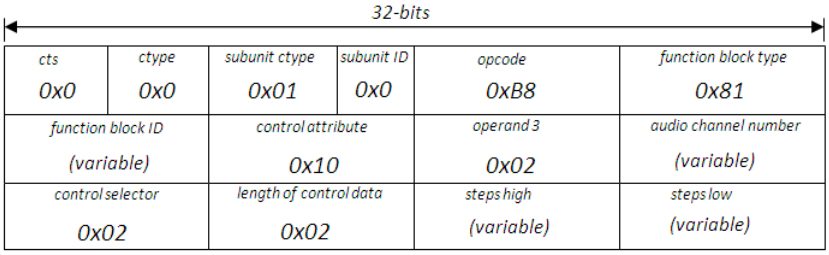


Figure B.8: Feature function block control command

Appendix C

Relevant Subunit Descriptors

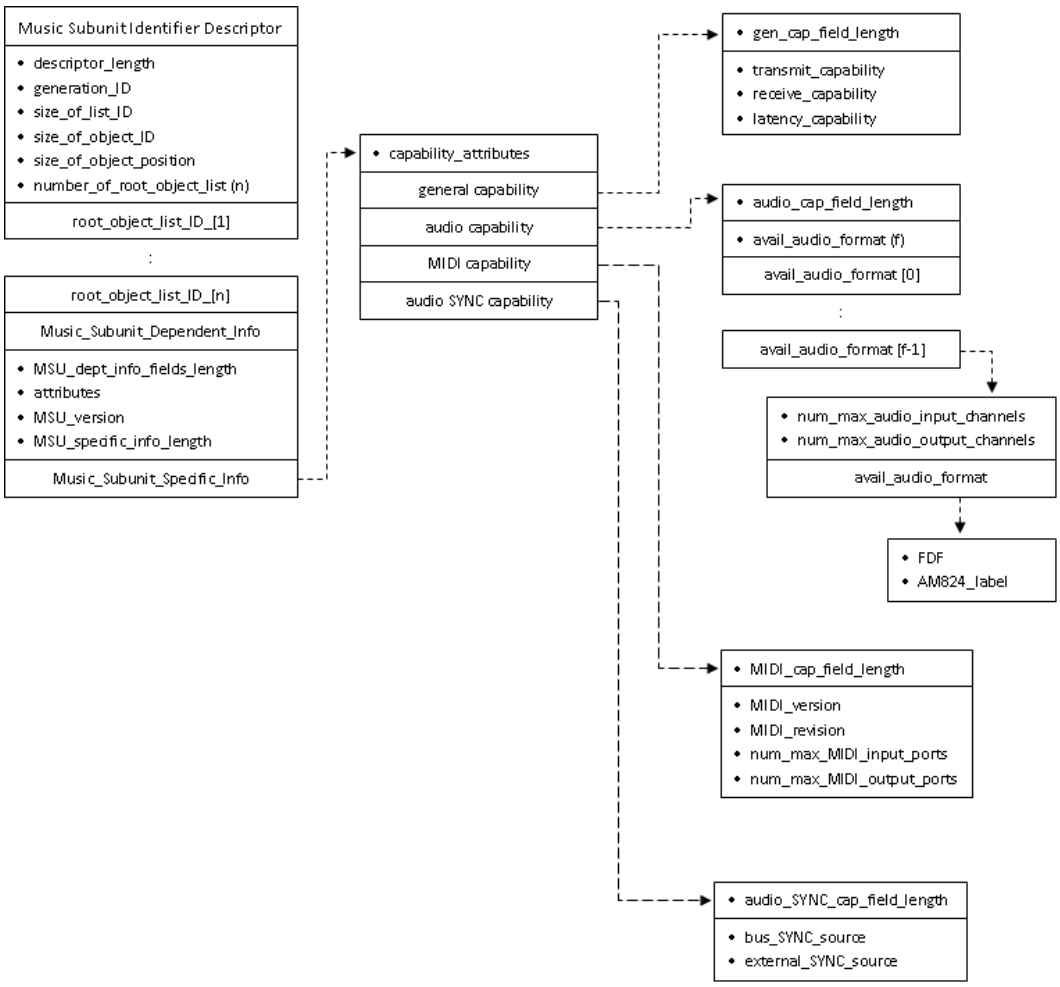


Figure C.1: Music Subunit Identifier Descriptor

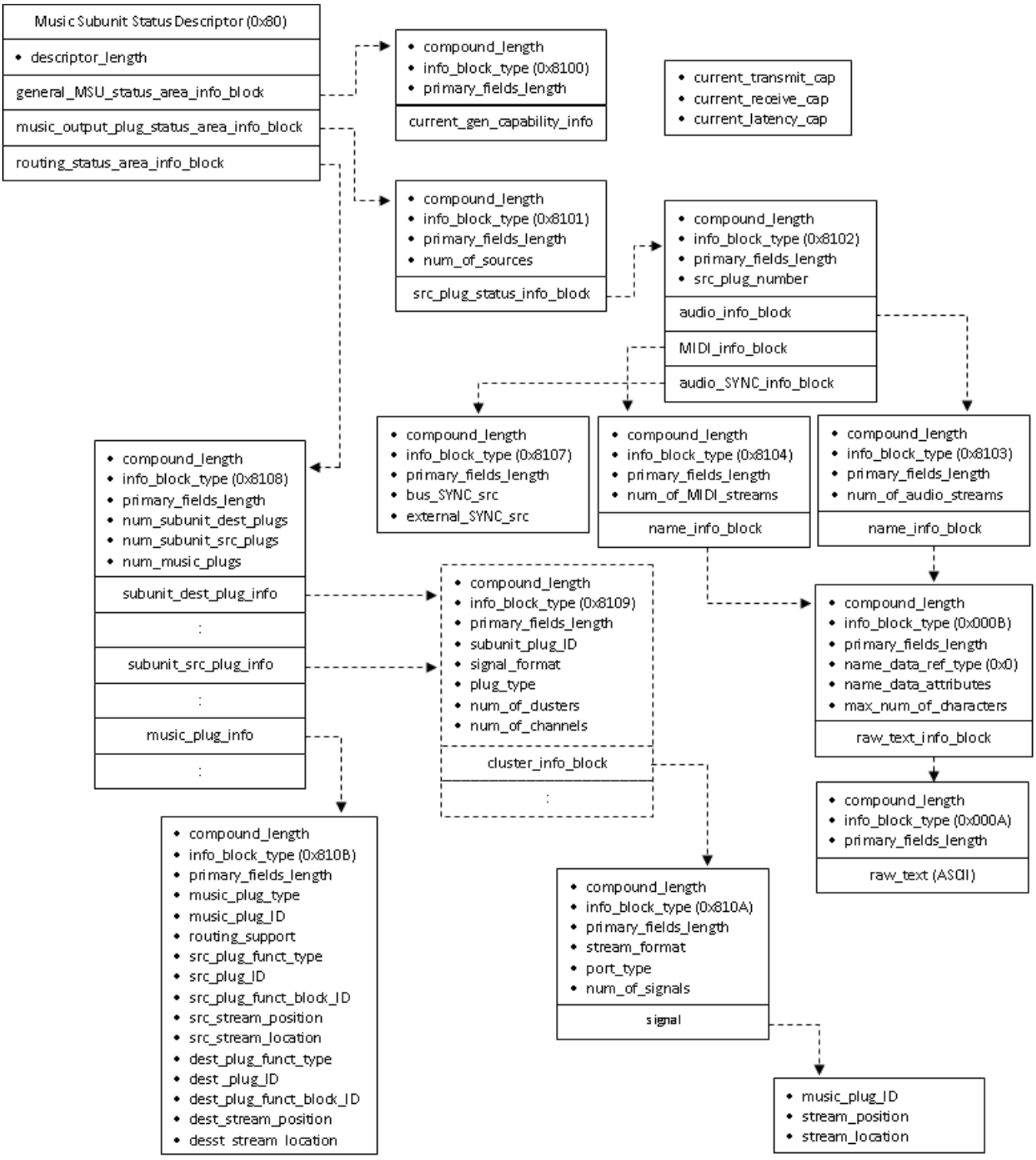


Figure C.2: Music Subunit Status Descriptor

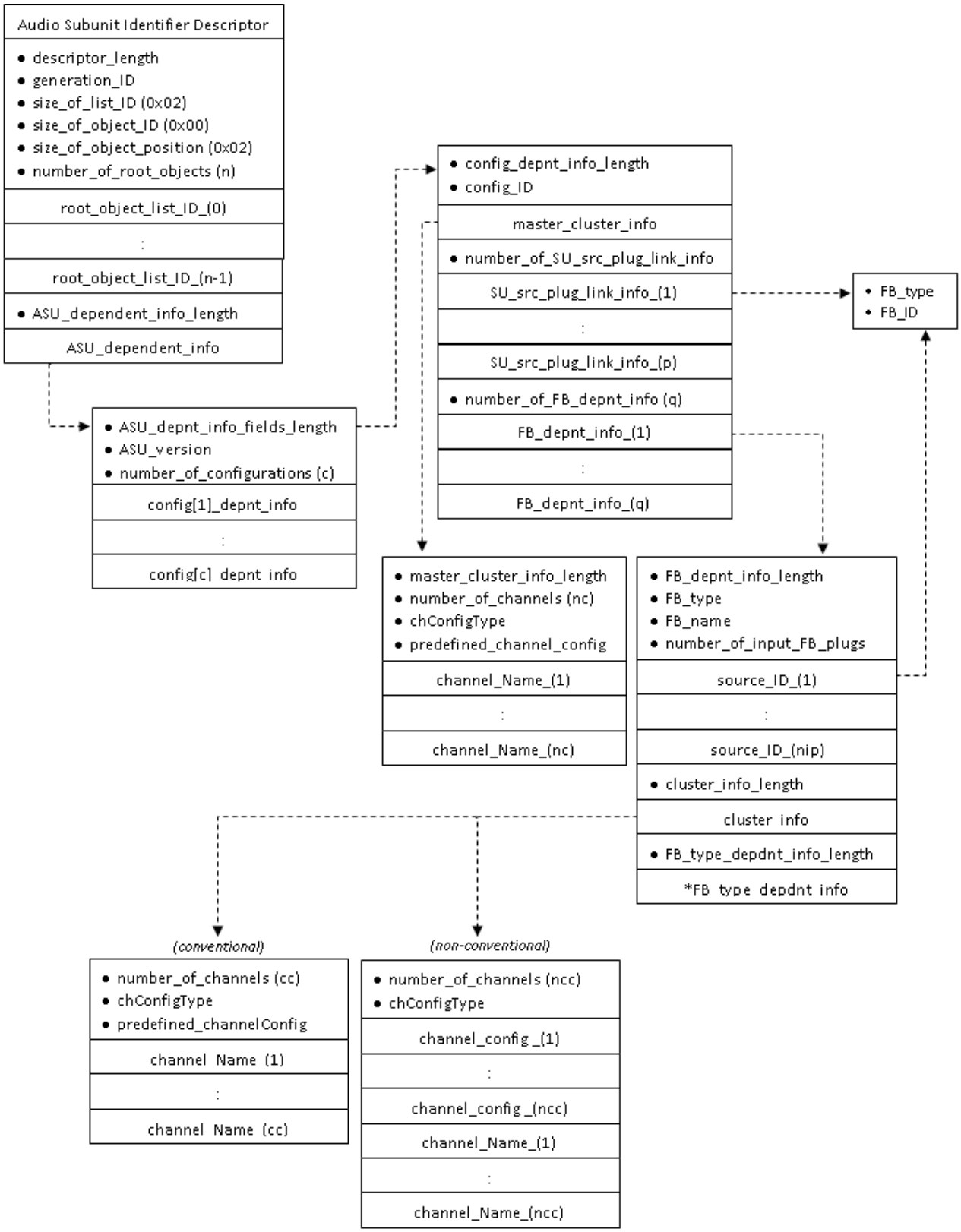


Figure C.3: Audio Subunit Identifier Descriptor

Appendix D

XFN Parameters for Phase 24 FW

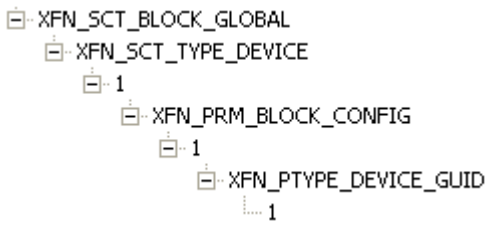


Figure D.1: Phase 24 Section Block Global

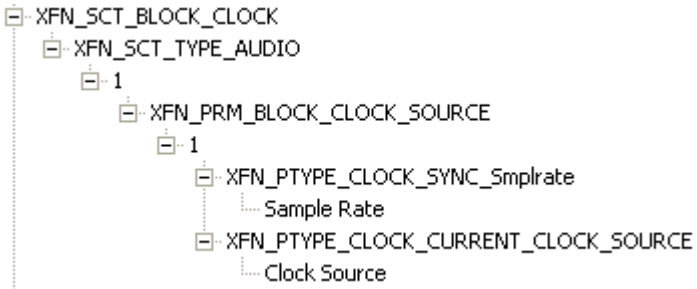


Figure D.2: Phase 24 Section Block Clock

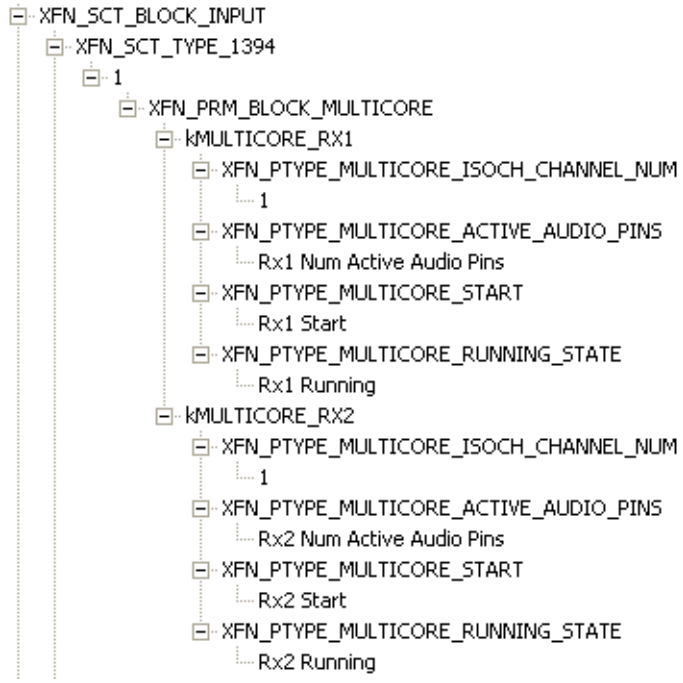


Figure D.3: Phase 24 Section Block Input

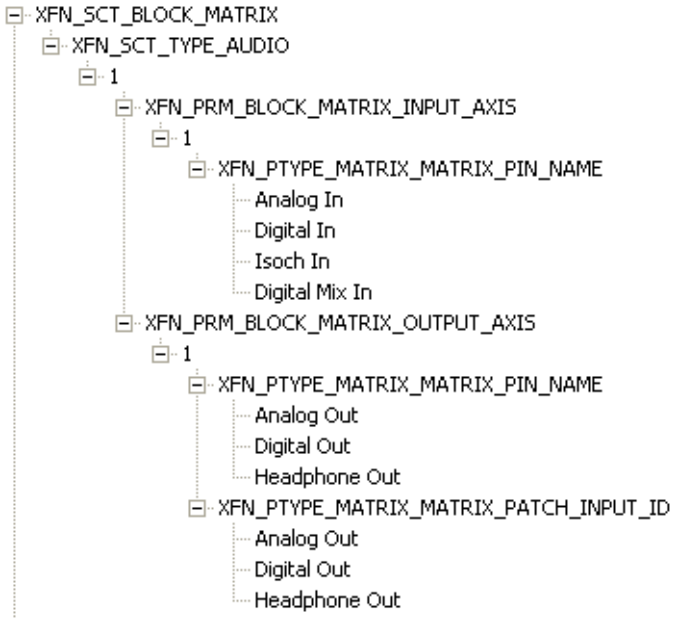


Figure D.4: Phase 24 Section Block Matrix

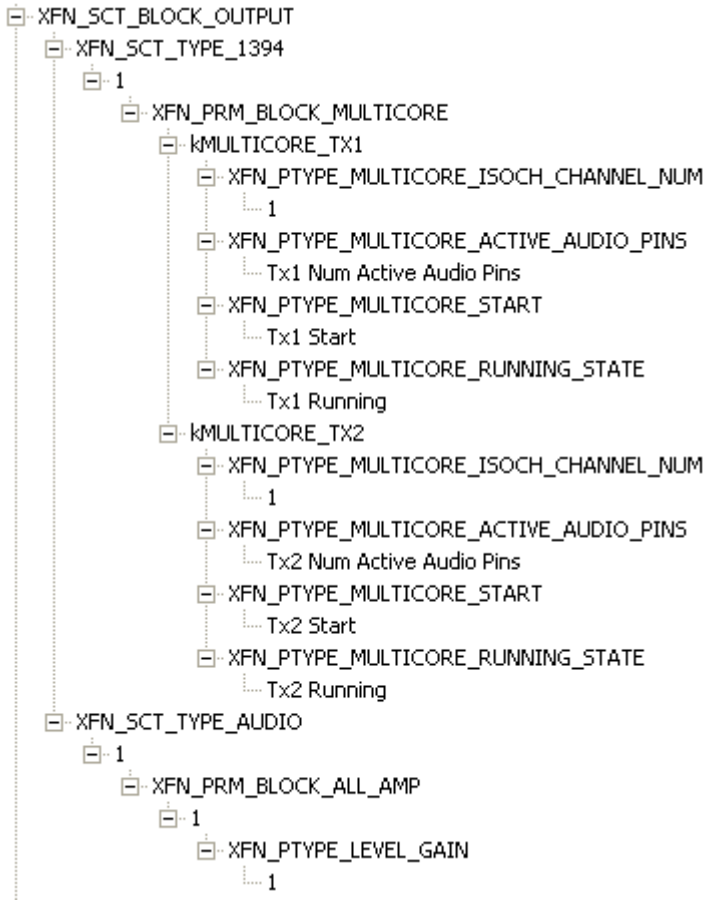


Figure D.5: Phase 24 Section Block Output