An Investigation into the Hardware Abstraction Layer of the Plural Node Architecture for IEEE 1394 Audio Devices

Submitted in fulfilment
of the requirements of the degree
Master of Science
of Rhodes University

by
Nyasha Chigwamba

December 2007

Abstract

Digital audio network technologies are becoming more prevalent in audio related environments. Yamaha Corporation has created a digital audio network solution, named mLAN (music Local Area Network), that uses IEEE 1394 as its underlying network technology. IEEE 1394 is a digital network technology that is specifically designed for real-time multimedia data transmission.

The second generation of mLAN is based on the Plural Node Architecture, where the control of audio and MIDI routings between IEEE 1394 devices is split between two node types, namely an Enabler and a Transporter. The Transporter typically resides in an IEEE 1394 device and is solely responsible for transmission and reception of audio or MIDI data. The Enabler typically resides in a workstation and exposes an abstract representation of audio or MIDI plugs on each Transporter to routing control applications. The Enabler is responsible for configuring audio and MIDI routings between plugs on different Transporters. A Hardware Abstraction Layer (HAL) within the Enabler allows it to uniformly communicate with Transporters that are created by various vendors. A plug-in mechanism is used to provide this capability. When vendors create Transporters, they also create device-specific plug-ins for the Enabler. These plug-ins are created against a Transporter HAL Application Programming Interface (API) that defines methods to access the capabilities of Transporters.

An Open Generic Transporter (OGT) guideline document which models all the capabilities of Transporters has been produced. These guidelines make it possible for manufacturers to create Transporters that make use of a common plug-in, although based on different hardware architectures. The introduction of the OGT concept has revealed additional Transporter capabilities that are not incorporated in the existing Transporter HAL API. This has led to the underutilisation of OGT capabilities.

The main goals of this investigation have been to improve the Enabler's plug-in mechanism, and to incorporate the additional capabilities that have been revealed by the OGT into the Transporter HAL API. We propose a new plug-in mechanism, and a new Transporter HAL API that fully utilises both the additional capabilities revealed by the OGT and the capabilities of existing Transporters.

Acknowledgements

First and foremost, I am grateful to **Richard Foss**, my supervisor, for his invaluable guidance and motivation throughout the course of my research. He has taught me the essence of research and academic writing.

I thank the **Mandela Rhodes Foundation**, the **Centre of Excellence in Distributed Multimedia at Rhodes University**, and the **Joint Research Committee at Rhodes University** for financial support.

I thank the Networked Audio Solutions team - **Harold Okai-Tettey** and **Bradley Klinkradt** for software support, and **Rob Laubscher** for firmware support.

Thanks to **Richard Byrne** of I/One Connects, Canada, for testing my software and for the feedback received.

Thanks to **Mark Olleson** of Yamaha R&D Centre, London, for our fruitful discussions at the onset of my research.

Thanks to mLAN team at Yamaha Corporation, Japan - **Jun-ichi Fujimori** for his work in creating the Plural Node Architecture, and **Ken Kounosu** and **Takashi Furukawa** for clarifying mLAN concepts for me through Richard Foss.

Lastly, I thank my family and friends for their support - my **parents** for reading some of my work and giving me feedback, and my **sisters** and **friends** for the motivation they gave me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As digital audio technologies continue to outplay analog audio technologies, the projected demand for digital audio networks is inevitably increasing. Digital audio networks, among other capabilities, allow remote control over the routing of audio between audio sources and audio receivers. Some of the application areas of digital audio networks include, but are not limited to, broadcast and recording studios, stadiums, houses of worship, hotels, and convention centres. A number of technologies have been developed to meet the audio network requirements in a variety of environments. Some of these technologies are being used in professional audio environments, where the demand for premium quality and functionality is high. This research was initiated because of a need to expose optimal functionality within a professional audio network.

1.1 Analog and Digital Audio Routing

Audio may be represented in two main forms, namely analog and digital. Analog audio representation takes the form of a varying electrical voltage signal. Fluctuations in this signal have a significant impact on the nature of the resulting audio quality. As a result, any unwanted signal induced in transmission wires such as radio frequency interference, and interferences from other sources such as power cables, cannot be distinguished from the desired signal. Noise can be reduced in analog audio transmission by, for example, making use of balanced cables and applying various forms of modulation.

On the other hand, digital audio is mostly derived from analog audio. The varying electrical voltage signals are converted to discrete on/off pulses. This implies that there are two possible

states for digital audio, the on state and the off state, making it easier to remove the effects of unwanted noise. Although Rumsey and Watkinson [1993] suggest that an analog signal's digital counterpart requires more bandwidth, they indicate that another benefit of digital audio is that different types of audio signals may be carried on the same physical link without interfering with each other. In addition, Watkinson [2001] shows how the lower cost of ownership of digital equipment, when compared with that of analog equipment, is leading to a drop in demand for analog equipment.

There are two common alternatives to routing both digital and analog audio from audio sources to audio destinations, namely the use of point-to-point cabling between the source and destination devices, and the use of digital audio networks. These alternatives are described below.

1.1.1 Point-to-Point Cabling

In this form of routing, a predefined physical signal path needs to be set up between the source and destination audio devices, as shown in Figure 1.1. Patch bays may be used to allow for

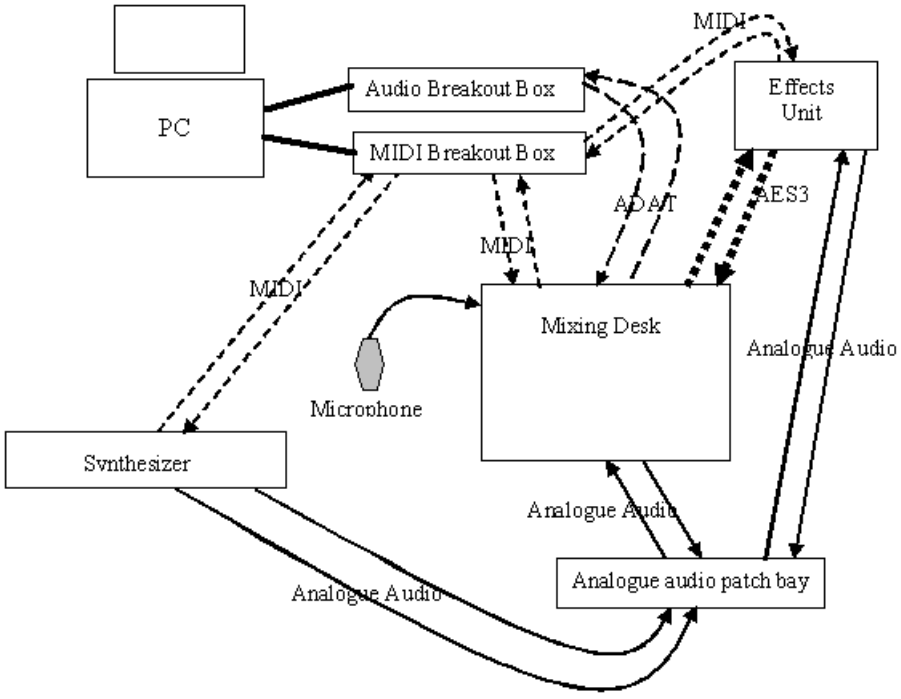


Figure 1.1: Simple Audio Studio Configuration with Point-to-Point Cable Routing

flexible audio path configurations. A patch bay can be viewed as a central routing device which

facilitates the connection of outputs to inputs. Typically, outputs and inputs of the various audio devices are connected to the patch bay. Connections between the outputs and inputs may be done manually using patch leads or via the control of software on a host device, such as a computer. Okai-Tettey [2005] highlights how the myriad of analog and digital audio connectors and their corresponding cable types (as shown in Figure 1.1) leads to complications in management of audio routings. The various types of connectors include: "MIDI, RCA, phone jacks, BNC, AES/EBU, SPDIF, ADAT, TDIF, and MADI" [Okai-Tettey, 2005]. Some of the complications include:

- The cable clutter that results from the use of many different types of cables, which leads to time intensive installations and reconfigurations.
- Signal loss in environments with long analog cable runs, and the costs involved in attempting to minimise such losses.
- The costs incurred in purchasing all the different types of cables for the various connectors.

1.1.2 Digital Audio Networks

Okai-Tettey [2005] discusses the results from a survey conducted by the AES Technical Committee on Networked Audio Systems (AES TC-NAS) [Gross, 2006]. Notably, apart from other important aspects such as the amount of latency and cost requirements, three important points were highlighted by the survey, and these include:

1. The need for open and multi-sourced technology standards in audio networks.
2. The importance of control and monitoring capabilities within audio networks.
3. Lack of preference between the use of standalone infrastructure or ubiquitous data communications infrastructure that is already in place.

A number of digital audio network technologies that resolve some of the problems in point-to-point cable based networks exist. The most common technologies make use of Ethernet as the underlying network technology. In order to ensure guaranteed quality of service (QoS), most manufacturers of Ethernet based solutions have implemented their own proprietary protocols above the Ethernet protocol. As a result, most of these protocols, although they use the same underlying network technology (Ethernet), are not interoperable. Yamaha Corporation introduced

the use of IEEE 1394 [IEEE Std. 1394, 1995; IEEE Std. 1394a, 2000; IEEE Std. 1394b, 2002] in digital audio networks through its music Local Area Network (mLAN) project [Yamaha Corporation, 2007]. Unlike Ethernet, IEEE 1394 was created for multimedia data transmission, which makes it ideal for audio and video networks. A common feature of both IEEE 1394 and Ethernet based networks is that they both require a single cable for device interconnectivity. This thesis focuses on the IEEE 1394 based audio network alternative.

1.2 Problem Statement

IEEE 1394, commonly known as FireWire, is a digital network interface technology that allows professional audio equipment, PCs, and electronic devices to be interconnected using a single cable. Among other features, IEEE 1394 is used in audio networks because of its low latency, deterministic data transmission, and simultaneous transmission of audio data with control data [Anderson, 1999].

The International Electrotechnical Commission (IEC) has created an Audio and Music (A/M) Data Transmission Protocol [IEC, 2005]. This protocol describes the formatting of audio and music[1] data that is transmitted over IEEE 1394. Various manufacturers have created chips (ASICs[2]) that comply with the A/M Data Transmission Protocol requirements. Furthermore, the 1394 Trade Association has documented an architecture known as the "Plural Node Architecture" [1394 Trade Association, 2004], which originated from Yamaha Corporation's mLAN project. The Plural Node Architecture splits connection management of IEEE 1394 audio devices between two nodes types, namely an Enabler and a Transporter.

The Transporter node typically resides in an IEEE 1394 audio device and its sole responsibility is to encapsulate and extract audio and MIDI data in a manner that complies with the A/M Data Transmission Protocol. The Transporter also exposes a Transporter Control Interface which allows an Enabler to access and modify A/M Data Transmission Protocol parameters. The Enabler node resides in a controlling IEEE 1394 device, typically a workstation. It provides a high level abstraction of the audio channel end-points on each Transporter and is responsible for making and breaking connections between these end-points.

When manufacturers create Plural Node audio devices, they provide a Transporter Control Interface on each device, which accepts low level device-specific commands from a device-specific

[1] Data such as MIDI.
[2] Application-Specific Integrated Circuits

Enabler plug-in. The Transporter Control Interface forwards these commands to the A/M Data Transmission Protocol layer of the device. There is variation in the hardware design of the various manufacturers' chips. In order for an Enabler to transparently control Transporters with different hardware designs, a Transporter Hardware Abstraction Layer (HAL) Application Programming Interface (API) is defined for the Enabler. The Transporter HAL API defines a standard set of methods that the Enabler uses to access the capabilities of Transporters. Manufacturers create their device-specific plug-ins for an Enabler against its Transporter HAL API.

The mLAN project created its version of an Enabler, their own chips (ASICs), and defined a Transporter HAL API to access their chip capabilities [Yamaha Corporation, 2002c, 2001, 2003a]. We refer to the mLAN project's Transporter HAL API as the current Transporter HAL API. A new version of the Enabler, known as the Redesigned Enabler, has been created by Okai-Tettey [2005]. The Redesigned Enabler continues to use the current Transporter HAL API.

In an attempt to create an open, standards-based implementation of a Transporter, and also to ease the task of manufacturers, the Audio Engineering Society's SC-02-12-G Task Group has produced an Open Generic Transporter (OGT) guideline document [Audio Engineering Society - Standards Committee, 2005]. This document provides a high level abstract description of a possible Transporter design paradigm, which encapsulates the underlying functions of the A/M Data Transmission Protocol. In particular, if manufacturers design Transporters in accordance with the OGT guidelines, there will be no need to create their own device-specific Transporter plug-ins, since they can make use of a common, open, generic Transporter plug-in.

Although the main motivation for the OGT guideline document was manufacturer convenience, it has also revealed additional Transporter capabilities that were not planned for by the existing mLAN Enablers at the time of their implementation. An Enabler accesses Transporter capabilities via the methods defined in its Transporter HAL API. The current mLAN Enablers have been designed to facilitate connection management between IEEE 1394 audio devices with HAL plug-ins that are written against the current Transporter HAL API.

Our investigation made use of the Redesigned Enabler. An OGT HAL plug-in has been implemented for this Enabler against the current Transporter HAL API. However, the difference in Transporter design philosophies assumed at the time of defining the current Transporter HAL API and those that the OGT guideline is based on have led to underutilisation of OGT capabilities. While the OGT guideline document reveals some additional Transporter capabilities, the Enabler is restricted to only utilising capabilities defined in the current Transporter HAL API. The current Transporter HAL API does not define methods to access these additional Transporter

capabilities. Such a restriction within the Enabler also restricts the capabilities available to high level end-user connection management applications that make use of the Enabler.

This research primarily aimed to investigate how the additional Transporter capabilities revealed by the OGT guideline document can be incorporated into the Enabler, by revealing these capabilities in the Transporter HAL API. However, the current Transporter plug-in mechanism that is used by the Enabler was also found to be problematic. The main problems with this plug-in mechanism are with regards to the existence of binary dependence between the Enabler and its associated Transporter HAL plug-ins, and some inadequacies within the mechanism used for Transporter HAL API versioning.

The outcomes of the research were:

- A new Transporter plug-in mechanism that reduces the level of binary dependence between the Enabler and Transporter HAL plug-ins, whilst allowing for robust Transporter HAL API versioning.
- A new Transporter HAL API that completely reflects the OGT capabilities that were lacking in the mLAN project's Transporter HAL API.
- Proof of concept HAL implementations against our proposed Transporter HAL API in order to demonstrate feasibility of our proposed Transporter HAL API in completely utilising capabilities of both OGT-based and non-OGT-based Transporters.
- Modified versions of the Redesigned Enabler, and a high level connection management application, to demonstrate how users can utilise the additional Transporter capabilities that have been revealed by the OGT guideline document.

1.3 Document Structure

Chapter 2 describes the current state of digital audio networking. A brief background of some existing Ethernet based digital audio network technologies and mLAN (an IEEE 1394 based alternative) is given. The chapter is concluded by highlighting motivations for further investigation of IEEE 1394 audio networking with mLAN.

Chapter 3 provides background information on IEEE 1394. In addition, a description of the work produced by the mLAN project is given. Important sections of the standards on which mLAN is based are also described in detail.

Chapter 4 analyses the designs and implementations of the current Transporter plug-in mechanism and the current Transporter HAL API. These two sections of the HAL comprise the two key investigation areas that are discussed in this thesis. Shortcomings within these two investigation areas are highlighted in this chapter.

Chapter 5 describes our proposal for a new HAL plug-in mechanism, which resolves shortcomings identified for the first of the two key investigation areas that are mentioned in Chapter 4.

Chapter 6 describes our proposal for a new Transporter HAL API, which resolves shortcomings identified for the second of the two key investigation areas that are mentioned in Chapter 4.

Chapter 7 highlights some innovations that have resulted from our proposed Transporter HAL API, which is described in Chapter 6. Here, modifications made to an existing connection management application are discussed and the resulting enhancements to end-user capabilities are shown.

Chapter 8 concludes the thesis by revisiting the context of the investigation, highlighting key motivations for the investigation, highlighting main shortcomings within the current implementations, and how these shortcomings were resolved.

Chapter 2

Current State of Digital Audio Networking

A number of digital audio network technologies exist. Ethernet is the most common network technology that is used by most of the existing digital audio network solutions. The main advantage gained from the use of Ethernet is that of convergence. In particular, the use of Ethernet for audio networks implies that existing data communications infrastructure within homes and professional audio environments can be used, hence no extra costs are incurred in creating new infrastructure. In spite of such benefits, various manufacturers have created their own protocols (proprietary and non-interoperable in most cases) that guarantee the quality of service (QoS) required for real-time audio distribution. Unlike Ethernet based solutions, mLAN is a digital network interface technology that uses IEEE 1394 as its underlying network technology. The main advantage of IEEE 1394 is that it was created for transmission of real-time multimedia data, hence QoS is guaranteed in its standard form. This chapter highlights the existing, or upcoming, audio network technologies and their capabilities. To conclude the chapter, motivations for closer investigation of IEEE 1394 based audio networks are provided.

2.1 Existing Audio Networks

A survey conducted by the Audio Engineering Society's Technical Committee on Networked Audio Systems (AES TC-NAS) revealed that there is a readiness for digital audio networking in all audio related contexts, since digital technologies continue to edge out analog technologies [Gross, 2006]. Because of this, there have been many attempts at audio network solutions. Given below are the most significant products arising from these attempts.

- CobraNet™ by Peak Audio (now owned by Cirrus Logic) [Cirrus Logic, 2007]
- EtherSound by Digigram [Digigram, 2006]
- Livewire™ by Telos Systems (Axia Audio) [Telos Systems/Axia Audio, 2007]
- Media-Accelerated Global Information Carrier (MAGIC) by Gibson [Gibson Musical Instruments, 2003]
- SuperMAC, an implementation of "AES50-2005" [Audio Engineering Society, 2005a] by Sony, Oxford [Oxford Technologies, 2007a]
- SmartBuss by Intelligent Media Technologies [Intelligent Media Technologies, Inc., 2007]
- DANTE Project by Audinate™ [Audinate, 2007], a company founded by researchers at National ICT Australia (NICTA) Research Institute [National ICT Austria, 2007]
- HearBus by Hear Technologies [Hear Technologies, 2007]
- A-Net Pro64 by Aviom Technologies [Aviom, Inc., 2007]
- Roland Ethernet Audio Communication (REAC) by Roland Systems Group [Roland Systems Group, 2007]
- mLAN (music Local Area Network) by Yamaha Corporation [Yamaha Corporation, 2007]

All these products are Ethernet based, with the exception of mLAN which is based on IEEE 1394 (FireWire). We will look at four of the most important Ethernet based audio network technologies and mLAN. We give motivations for a closer investigation of IEEE 1394 audio networking using mLAN.

2.1.1 Overview of Existing Digital Audio Network Technologies

2.1.1.1 CobraNet™ by Peak Audio/Cirrus Logic

Cirrus Logic is providing CobraNet™ technology to transport high quality uncompressed digital audio in real-time on a switched Ethernet or dedicated Ethernet repeater Local Area Network (LAN) [Cirrus Logic, 2006]. CobraNet™ technology comprises hardware (a CobraNet™ interface), a network protocol, and firmware. A CobraNet™ interface has standard Ethernet connectors that enable it to be connected to an Ethernet network. Three main communications services are provided by CobraNet™, namely isochronous audio data transport, sample clock distribution, and control and monitoring data transport.

The CobraNet™ protocol operates at the "OSI[1] Layer 2" [McClain, 1991]. It is possible for CobraNet™ devices to co-exist with other data communications devices that make use of the Internet Protocol (IP) for data transmission [Cirrus Logic, 2006]. However, CobraNet™ itself does not use IP to transport audio. Four basic packets types are used by CobraNet™ devices, namely the beat packet, isochronous data packet, reservation packet, and serial bridge packet. The beat packet is a multicast packet, transmitted periodically by only one CobraNet™ device on a network, containing network operating parameters, a clock, and transmission permissions. Isochronous data packets contain the uncompressed audio data and are transmitted upon receipt of the beat packet. A reservation packet is a multicast packet used to allocate bandwidth and establish connections between CobraNet™ interfaces. Serial bridge packets are used to bridge asynchronous serial data between CobraNet™ interfaces.

Routing of audio within a CobraNet™ network occurs in terms of *Bundles* [Gray, 2004]. In particular, a *Bundle* is the basic unit of data which contains up to eight audio sub-channels. No information could be found relating to support for MIDI transportation. With reference to the packets described above, *Bundles* make up the isochronous data packets. Each *Bundle* has a *Bundle* number that is used to identify it, and to determine where and how the data is sent. Audio connections can be made between CobraNet™ devices by assigning the destination CobraNet™ interface the same *Bundle* number as the one that is assigned to a transmitting CobraNet™ interface. CobraNet™ interfaces contain one or more Synchronous Serial Interfaces (SSIs) that carry multiplexed audio data into and out of the interfaces. Depending on the mode in use, a *Bundle* may be sent to all devices on the network (multicast), just one device (unicast), or a selected set of devices (private).

Depending on audio network requirements, there are CobraNet™ integrated circuits that can support up to 32 full-duplex output and input audio channels [Cirrus Logic, 2006]. Sample rates of 48 kHz and 96 kHz, and resolutions of 16, 20, or 24 bits are configurable on a CobraNet™ interface. Although CobraNet™ interfaces at 48 kHz and 96 kHz may co-exist on the same network, audio cannot be exchanged between interfaces with different sample rates. In addition, latency values of 1.33 ms, 2.66 ms, and 5.33 ms may be selected on a CobraNet™ network. Sample rate, resolution, and latency combinations may affect the allowable *Bundle* capacity. For example, the default mode of operation is 5.33 ms latency at 48 kHz. However, changes to latency, sample rate, or resolution may result in restrictions on the *Bundle* capacity as shown in Table 2.1 for a CobraNet™ module known as the *CM-1*. As shown in the table, lower latency modes have fewer restrictions with regards to the number of audio channels allowed in a *Bundle*.

[1] Open Systems Interconnection

Latency	Channels per Bundle					
	16 bit, 48 kHz	**20 bit, 48 kHz**	**24 bit, 48 kHz**	**16 bit, 96 kHz**	**20 bit, 96 kHz**	**24 bit, 96 kHz**
5.33 ms	8	8	7	5	4	3
2.66 ms	8	8	8	8	8	7
1.33 ms	8	8	8	8	8	8

Table 2.1: Bundle Capacity Limits (for CM-1) as a Function of Ethernet Packet Size [Cirrus Logic, 2006]

This is because lower latency is achieved by transmitting smaller audio packets at a higher rate, therefore the main restriction is the maximum size of an Ethernet packet. However, low latency modes place additional demands on network performance such as the need to reduce forwarding delays across the network by the same factor. Such demands require additional network design rules to be complied with.

CobraNet™ over switched Ethernet enables co-existence with ordinary computer data on the same network [Bayburn, 2004]. However, there may be problems if, for example, the computers attached to the network use 10 Mb/s Network Interface Cards (NICs), while packets are sent at 100 Mb/s. Multicast *Bundle*s are sent to all devices on the network. Most Fast Ethernet switches have dual 10/100 Mb/s ports. For example, if eight *Bundles* fill a Fast Ethernet (100 Mb/s) switch port that is connected to a 10 Mb/s NIC, the NIC will be saturated and will start dropping packets. Cirrus Logic have a number of solutions to this problem, which include, but are not limited to, using 100 Mb/s NICs for computers, refraining from use of multicast *Bundles*, using managed switches to exclude certain devices from receiving multicast packets, and using separate networks for either audio or data.

Control, monitoring, and connection management of CobraNet™ devices is done via a high-speed parallel host processor, or via Ethernet using the Simple Network Management Protocol (SNMP) [Cirrus Logic, 2007]. D&R Electronica has created software, "CobraNet™ Manager" [D&R Electronica, 2007], that allows SNMP-based control, monitoring, and connection management on a CobraNet™ network. This software allows for full remote control over the network and can be customised to monitor and control other non-CobraNet™ device-specific parameters.

2.1.1.2 EtherSound by Digigram

EtherSound is a Digigram patented technology that provides high quality audio transport over standard switched Ethernet networks [Digigram, 2006]. There are currently two implementa-

tions, namely ES-100 and ES-Giga, which are 100 Mb/s and 1 Gb/s network implementations respectively. EtherSound technology is fully compatible with IEEE 802.3x standards. The technology comprises a proprietary network protocol (OSI Layer 3 [McClain, 1991]) that has been created by Digigram. This protocol is built above the standard Ethernet (IEEE 802.3) Data Link Layer (OSI Layer 2). EtherSound frames are transported via a dedicated LAN and require an available bandwidth of at least 100 Mb/s full-duplex.

The ES-100 and ES-Giga implementations support up to 64 and 256 channels, respectively, of 24-bit full-duplex PCM[2] digital audio at 48kHz [Digigram, 2006]. Control data is also transported between connected nodes. For the ES-Giga implementation, a data path is also available to carry proprietary control traffic, including Ethernet based protocols such as IP. The typical end-to-end latency time is 5 samples (that is, 104 μs at 48 kHz) excluding analog-to-digital and digital-to-analog conversions. However, the latency time is 1.5 ms when the conversions are included.

Sample rates of 44.1 kHz, 48 kHz, 88.2 kHz, 96 kHz, and 192 kHz are supported. These sample rates have different capabilities, but can be used simultaneously. For example, the ES-100 network has a total audio channel capacity of 64 at 48 kHz, 32 at 96 kHz, and 16 at 192 kHz.

EtherSound networks are fully synchronous and carry their own clock [Digigram, 2006]. One device on the network is designated to be the *Primary Master* and the rest will be *slaves.* In this setup, the *Primary Master* generates the network audio clock using its own local clock or an external clock. The *slave* devices generate their clocks from the EtherSound network. Up to eight consecutive devices can be kept in phase depending on the network architecture.

Three network topologies are supported, namely daisy chain, star, and ring topologies [Digigram, 2006]. The star topology, which may not be used with hubs, requires switches and only supports uni-directional audio transport. Control of EtherSound devices is done via reads and writes to the devices' internal registers. Some registers are reserved for configuring EtherSound protocol parameters while others are left free for manufacturers to add their own customs controls.

Registers may be accessed in one of two possible ways, namely:

1. By an embedded application running on the EtherSound device, such as a digital signal processor (DSP), microcontroller, or field programmable gate (FPGA) array.
2. Through software running on a PC that is connected to the EtherSound device. An SDK is also available which simplifies the development of custom PC-based control applications.

[2] Pulse-code modulation

Digigram provides ESControl [Digigram, 2007], a Microsoft Windows based graphical user interface, to control one or more EtherSound networks. ESControl is based on a client-server approach. A server application is connected to the EtherSound network, while a client application, capable of connecting to the server via TCP/IP from anywhere, provides a matrix based user interface, as shown in Figure 2.1. This allows connections to be made

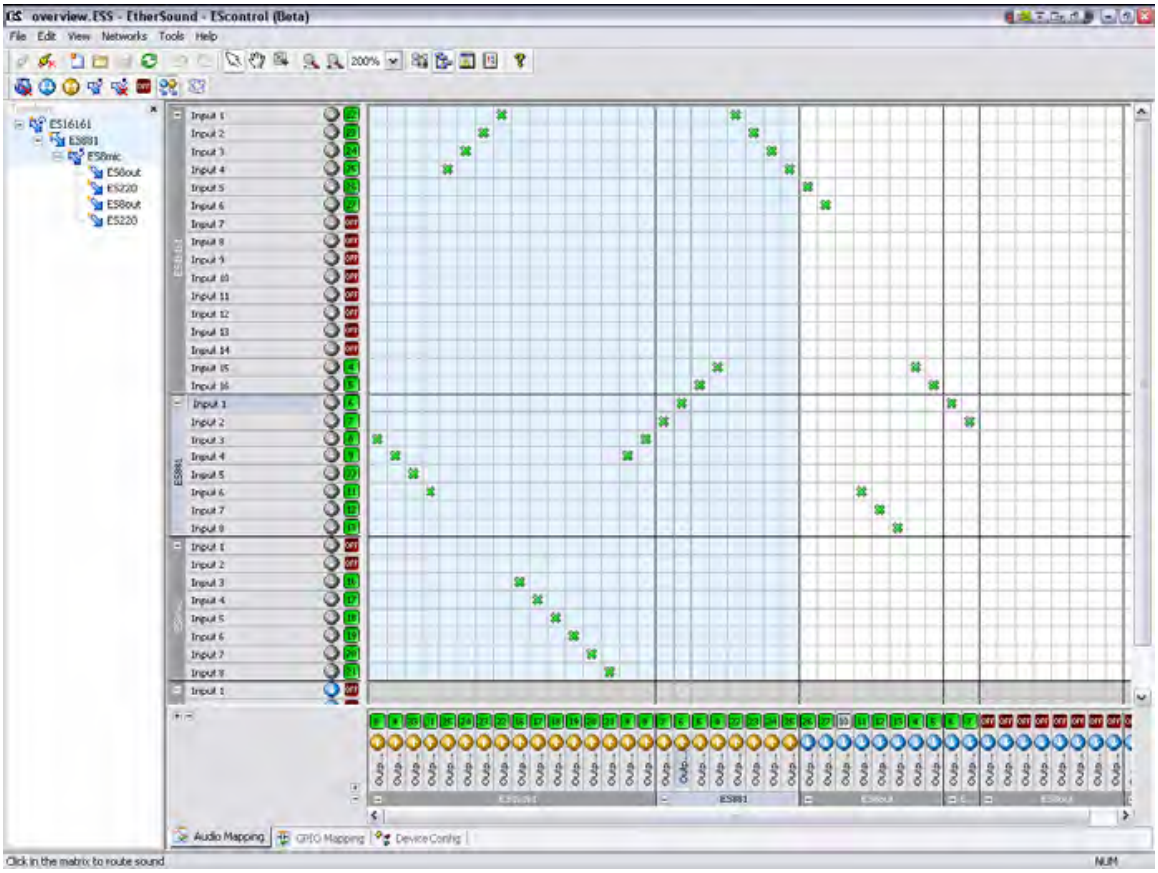


Figure 2.1: ESControl Graphical User Interface [Digigram, 2006]

between devices, in addition to controlling and monitoring device parameters. The diagram shows devices and their inputs on the vertical axis, and devices and their outputs on the horizontal axis. A connection between an input of one device and output of another can be made by clicking on the intersection between the output and the input.

2.1.1.3 Livewire™ by Axia Audio

Axia Audio, a division of Telos Systems, has created its patented Livewire™ technology for use in broadcast studios [Telos Systems/Axia Audio, 2007]. This technology transmits multiple

channels of real-time uncompressed digital audio, device control messages, programme associated data, and routine network traffic, using a single CAT-6 Ethernet cable or fibre link over switched Ethernet. A 100Base-T segment can carry 25 stereo channels of 48 kHz, full-duplex, 24-bit PCM audio. Livewire™ networks make use of Ethernet switches in order to guarantee the QoS that is required for real-time audio distribution. The advantage of switches is that audio data can be prioritised, allowing it to take precedence over other data. A proprietary transport protocol ensures that input-to-output latency is less than 1 ms per network hop.

The building blocks of Livewire™ networks are called "audio nodes" [Telos Systems/Axia Audio, 2007]. There are five node variants, namely an Analogue Node, an AES/EBU node, a Microphone node, a Router Selector node, and a GPIO node. These nodes provide an interface between conventional audio equipment and a Livewire™ network. In order to ensure reliability of network operations and low delays, audio nodes run a version of Linux on an embedded processor, and have built-in web servers that allow for remote configuration and control via a web browser. An Axia Windows driver is also available to make a PC appear as an "audio node" on a Livewire™ network. This adds the capability of exchanging audio between PCs and other "audio nodes" on the network.

Signal routing, control, and monitoring may be done within a Livewire™ network by using a Microsoft Windows PC client-server software package called PathfinderPC [Telos Systems/Axia Audio, 2004]. One or more servers communicate with all the nodes in a Livewire™ network, offering redundant common points of control. Multiple clients can connect to each of the servers from where presets can be set up to allow for quick global or local studio changes. A virtual patch bay user interface, as shown in Figure 2.2, ensures that the task of signal routing is intuitive, and allows users to make routings by, for example, clicking on the destination to be routed to, and then selecting the source. Route points can also be locked to prevent other users of the system from mistakenly changing routings that are in use. PathfinderPC can also combine audio and machine logic into a single "virtual router", allowing simultaneous full-duplex routing of audio and GPIO. A "watch" can also be placed on a destination device in order to switch to a backup source in the event that the currently routed source fails.

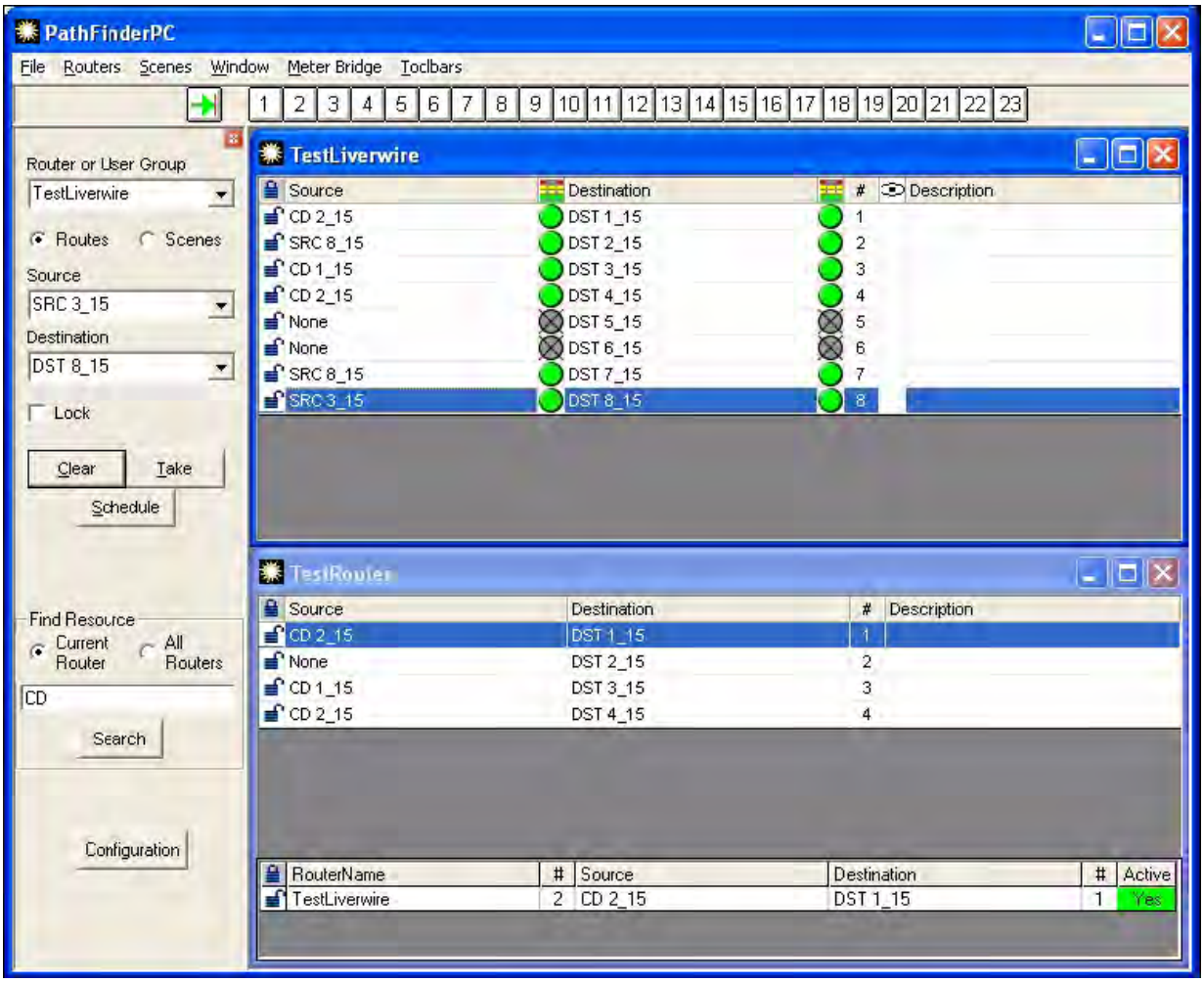


Figure 2.2: PathfinderPC User Interface for Audio Routing [Telos Systems/Axia Audio, 2004]

2.1.1.4 SuperMAC and HyperMAC by Oxford Technologies

SuperMAC and HyperMAC are complementary technologies that have been created by Sony Pro Audio Lab, Oxford, for sample clock and multi-channel audio interconnection using CAT-5/6 cables [Oxford Technologies, 2007a]. The technologies are optimised for applications requiring low latency and high reliability via Ethernet physical layer audio data transmission. Latency is as low as three cycles at 44.1 kHz and 48 kHz sample rates, that is, 68 μs and 63 μs respectively.

SuperMAC is an implementation, and also the origin, of the AES50-2005 standard [Audio Engineering Society, 2005a]. The standard specifies the capability to transmit up to 48 channels (at 44.1 kHz and 48 kHz) of full-duplex 24-bit digital audio in a variety of formats using a single CAT-5 (or better) structured data cable. This technology is designed for use in studio environ-

ments. The 100Base-TX physical layer of Fast Ethernet is used to transport digital audio data frames. A $64f_s$[3] audio clock signal is transmitted in parallel with the audio data via the extra signal pairs of the structured data cable. SuperMAC also provides a 5 Mb/s full-duplex packet-switched Ethernet data channel for generic data communication, such as control and status data [Oxford Technologies, 2007b].

Complementary to SuperMAC, HyperMAC uses Gigabit Ethernet physical layer technology via conventional CAT-6 or fibre optic cable. HyperMAC effectively provides the same service as SuperMAC, although with much higher capacity. This technology has not been standardised yet, but Sony intends submitting the HyperMAC protocol specification for standardisation. In addition, HyperMAC makes it possible to use standard Gigabit Ethernet switching hardware for signal distribution. There are eight streams, as shown in Figure 2.3, each with the same capacity as a SuperMAC link.

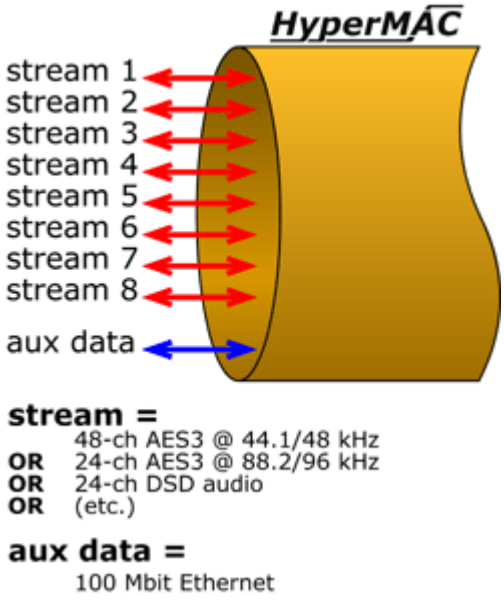


Figure 2.3: HyperMAC Channel Capacity [Oxford Technologies, 2007b]

Each stream may be associated with one of four independent asynchronous sample clocks and may carry different audio formats at different sample rates. This allows multiple audio streams of different asynchronous audio sample rates to be carried at the same time. The formats shown in Figure 2.3 are "AES3" [Audio Engineering Society, 2003] and Sony Corporation's "DSD™" (or

[3] f_s refers to a base audio sampling frequency [Audio Engineering Society, 2005b]. This may be 44.1 kHz or 48 kHz regardless of other sample frequency multipliers used in high resolution digital audio.

Direct Stream Digital™) [Audio Engineering Society, 2005b]. HyperMAC also provides a 100 Mb/s full-duplex packet-switched Ethernet data channel for generic data communication that is independent of the audio. This is used for control and status monitoring, or sending compressed video signals [Oxford Technologies, 2007b].

Sony has created a SuperMAC/HyperMAC router that acts as a studio hub for interconnecting a number of AES50/SuperMAC or HyperMAC-equipped devices [Oxford Technologies, 2007a]. Figure 2.4 shows an example of a layout with one HyperMAC link and two SuperMAC links, although more links are possible. Generic Ethernet traffic for control applications is packet-

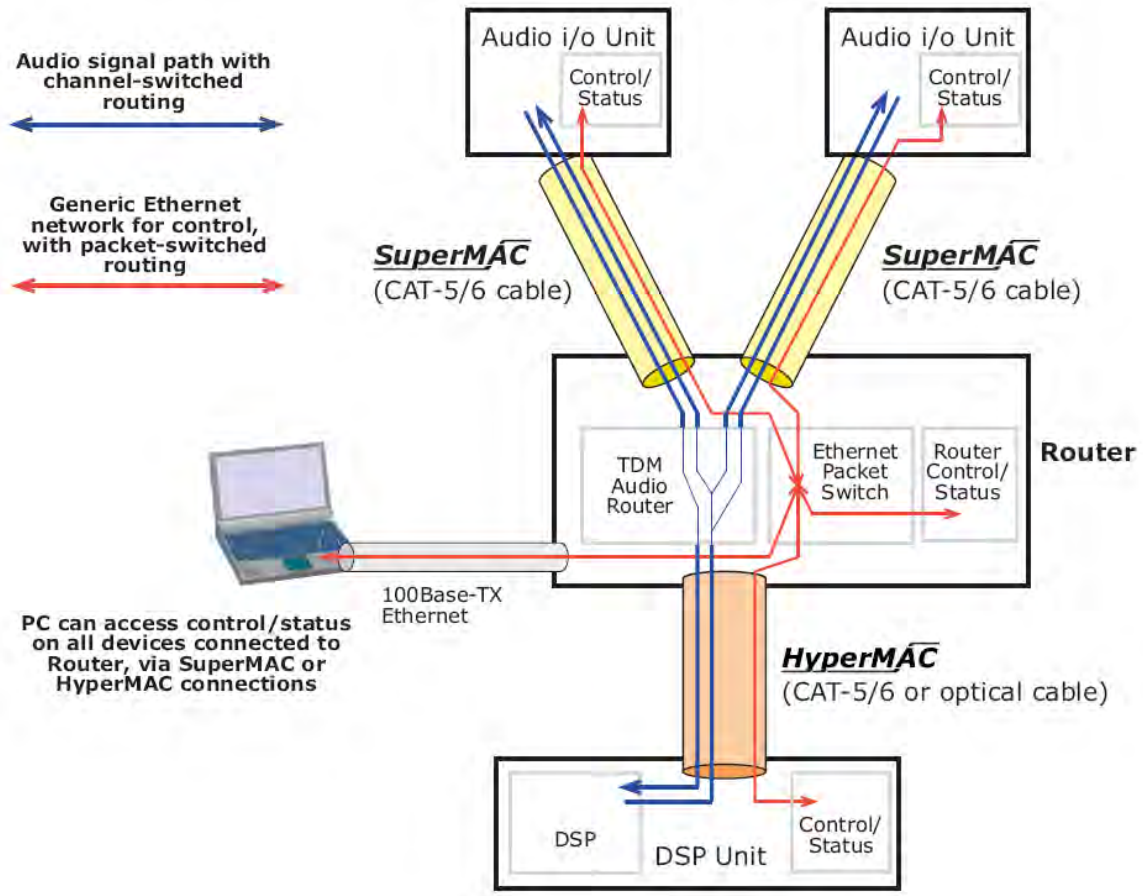


Figure 2.4: Schematic Example of SuperMAC/HyperMAC Interconnected System [Oxford Technologies, 2007a]

switched and also accessible from any connected device, while audio is channel-switched and has low latency.

The router may be controlled from another workstation or studio device via a TCP/IP-over-

Ethernet interface. A snapshot of an application which shows the number of channels available on each router port to allow control over routing is shown in Figure 2.5.

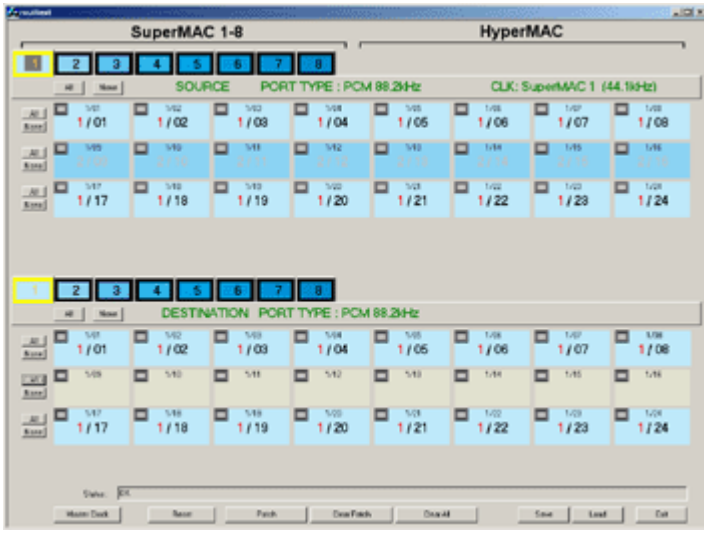


Figure 2.5: Sony's SuperMAC/HyperMAC Router Control Application [Oxford Technologies, 2007b]

"AES47-2006" [Audio Engineering Society, 2006] is a standard for transmission of professional audio across metropolitan- and wide-area networks over public asynchronous transfer mode (ATM) networks. Sony Pro-Audio Lab are working in collaboration with Nine Tiles Networks to develop "Audiolink 2" [Nine Tiles Networks Ltd., 2007], a SuperMAC-to-AES47 bridge. This bridge will provide an integrated local/long-haul solution for audio routing and device control. Discussion of metropolitan- and wide-area audio networks is beyond the scope of this thesis.

2.1.1.5 mLAN by Yamaha Corporation

mLAN (music Local Area Network) technology has been created by Yamaha Corporation. It is a digital audio network interface technology that uses the IEEE 1394 high performance serial bus standard and its amendments [IEEE Std. 1394, 1995; IEEE Std. 1394a, 2000; IEEE Std. 1394b, 2002]. The IEEE 1394 serial bus offers high-speed communications and isochronous (real-time) data transmission. mLAN is also based on the IEC 61883-6 specification [IEC, 2005].

IEEE 1394, commonly known as FireWire, allows PCs and a variety of high performance multimedia devices to be interconnected using a single cable [Anderson, 1999]. In its standard form,

IEEE 1394 is simple, allows for low cost implementations, and supports isochronous data transmission between devices. It is based on, and adds serial bus specific extensions to, the international ISO/IEC 13213 specification, which is also formally known as "Information Technology - Microprocessor Systems - Control and Status Registers (CSR) Architecture for Microcomputer Buses" [IEC, 1994]. Theoretically, the architecture supports up to 1024 buses with up to 64 nodes on each bus. Therefore, the maximum number of nodes supported on an IEEE 1394 network is 65536. One of the node addresses is reserved for broadcast. This reduces the maximum number of possible physical nodes per bus to 63, resulting in a maximum network capacity of up 64512 nodes.

There are two main forms of transfers defined for an IEEE 1394 serial bus, namely asynchronous and isochronous transfers [Anderson, 1999]. Asynchronous transfers are periodic data transfers with guaranteed delivery. *Read, write* and *lock* transactions comprise the asynchronous transfers. Read and write transactions return data from and write data to a specified memory address within a node, respectively. Lock transactions are atomic read-modify-write operations. On the other hand, the rate of data transfer, as opposed to guaranteed delivery, is crucial for isochronous transfers. These transfers involve, for example, audio or video data which needs to be transferred at a constant rate in order for it to be reproduced without distortions. Consequently, in isochronous transactions, data is sent in the form of *isochronous packets* from a source node to a target node at regular intervals (every 125 μs). The most important component of an isochronous packet is the isochronous channel number. All isochronous packets from a source node bear an isochronous channel number (which is not in use by any other source node on the network) in the range 0 to 63 inclusive. Isochronous packets bearing the same channel number comprise an *isochronous stream*. More than one target node may receive the same isochronous stream.

The original IEEE 1394 [IEEE Std. 1394, 1995] specification and its first amendment, IEEE 1394a [IEEE Std. 1394a, 2000], support speeds of 100, 200, and 400 Mb/s in any acyclic network topology. The second IEEE 1394 amendment, IEEE 1394b [IEEE Std. 1394b, 2002], remains backwards compatible with previous versions, but defines speeds of 800 Mb/s, 1.6 Gb/s, and 3.2 Gb/s, and single hop distances of up to 100m in any network topology. The IEEE P1394c Working Group is working on an amendment that defines features and mechanisms which provide gigabit speed using CAT-5 cabling over single hop distances of up to 100m [IEEE P1394c Working Group, 2007].

As mentioned in the first paragraph of this section, mLAN is based on the IEC 61883-6 specification. This specification, also known as the "Audio and Music (A/M) Data Transmission

Protocol", describes the formatting of audio and music[4] data [IEC, 2005]. Audio and/or music data in AM824[5] quadlet (32-bit) format is packaged into a *"Common Isochronous Packet (CIP)"* [IEC, 2003]. This CIP makes up the payload of the standard IEEE 1394 isochronous packet. Each quadlet occupies a particular stream position, known as a *sequence*, within the packet cluster.

Various manufacturers have created application-specific integrated circuits (ASICs) that comply with the IEC 61883-6 specification. Yamaha Corporation created their own chips, namely mLAN-NC1 and mLAN-PH2. The mLAN-NC1 is capable of simultaneously transmitting and receiving eight sequences of audio data, two sequences of non-audio data, and four sequences of MIDI data streams, or transmitting and receiving one sequence of non-audio data and eight sequences of MIDI data streams [Yamaha Corporation, 2001]. On the other hand, the mLAN-PH2 chip is capable of simultaneously transmitting and receiving 32 audio data sequences [Yamaha Corporation, 2003a]. Up to four mLAN-PH2 chips may be used simultaneously, in addition to an mLAN-NC1 chip, giving a maximum full-duplex audio channel count of 128 at 48 kHz and eight MIDI data streams. The IEC 61883-6 specification also defines a mechanism for the transmission of audio clock information and a maximum transfer latency of 354.17 μs [IEC, 2005].

The 1394 Trade Association has documented an architecture known as the "Plural Node Architecture" [1394 Trade Association, 2004] (also known as the Enabler/Transporter Architecture) which splits connection management of IEEE 1394 audio devices over two node types, namely a Transporter and an Enabler . The Transporter resides in the audio device and is responsible for the encapsulation and extraction of audio/MIDI data in accordance with the A/M Data Transmission Protocol. The Transporter also exposes a Transporter Control Interface that allows it to be controlled by an Enabler. The Enabler, which typically resides in a workstation, is responsible for making or breaking connections between Transporters. A plug-in mechanism is used by the Enabler to facilitate hardware independent Transporter control by the Enabler. In particular, vendors typically create their Transporters using hardware architectures of their choice. In order to get mLAN compliance, software plug-ins are provided, for each different type of Transporter, to allow the Enabler to communicate with the Transporters in a device-specific manner. Second generation mLAN devices are based on the Plural Node Architecture.

Yamaha Corporation provides Audio Stream Input/Output (ASIO) drivers for Mac OS 9 and Windows operating systems [Holton, 2003]. Windows Driver Model (WDM) drivers are also provided for Windows. Yamaha Corporation and Apple Inc. have collaborated to produce low

[4]Data such as MIDI.

[5] 8-bit label and 24-bit data

latency drivers for Mac OS X as an integral part of the operating system. The use of these drivers makes mLAN compatible with any sequencer[6] or digital audio workstation (DAW) that supports ASIO, WDM, or Mac OS X Core Audio and Core MIDI.

Audio routings on an mLAN network may be configured via a graphical user application such as the one shown in Figure 2.6. Each of the five devices shown in the diagram has two plug lists, one labelled "IN" and the other labelled "OUT". To make a connection, a user selects a source plug from the source device's "OUT" plug list and a destination plug from the destination device's "IN" plug list. After the selections are made, a virtual cable is drawn between the source and destination plugs, as shown in Figure 2.6. When a connection is made, the destination plug is set to receive the audio data at the sequence position at which the source plug is transmitting within the source device's isochronous stream. Connection state may also be saved to a file to allow for quick reloading of setups. MIDI messages, encapsulated in isochronous packets, are used for device control in an mLAN network.

[6] A device or piece of software that allows the user to record, play back, and edit audio or MIDI data.

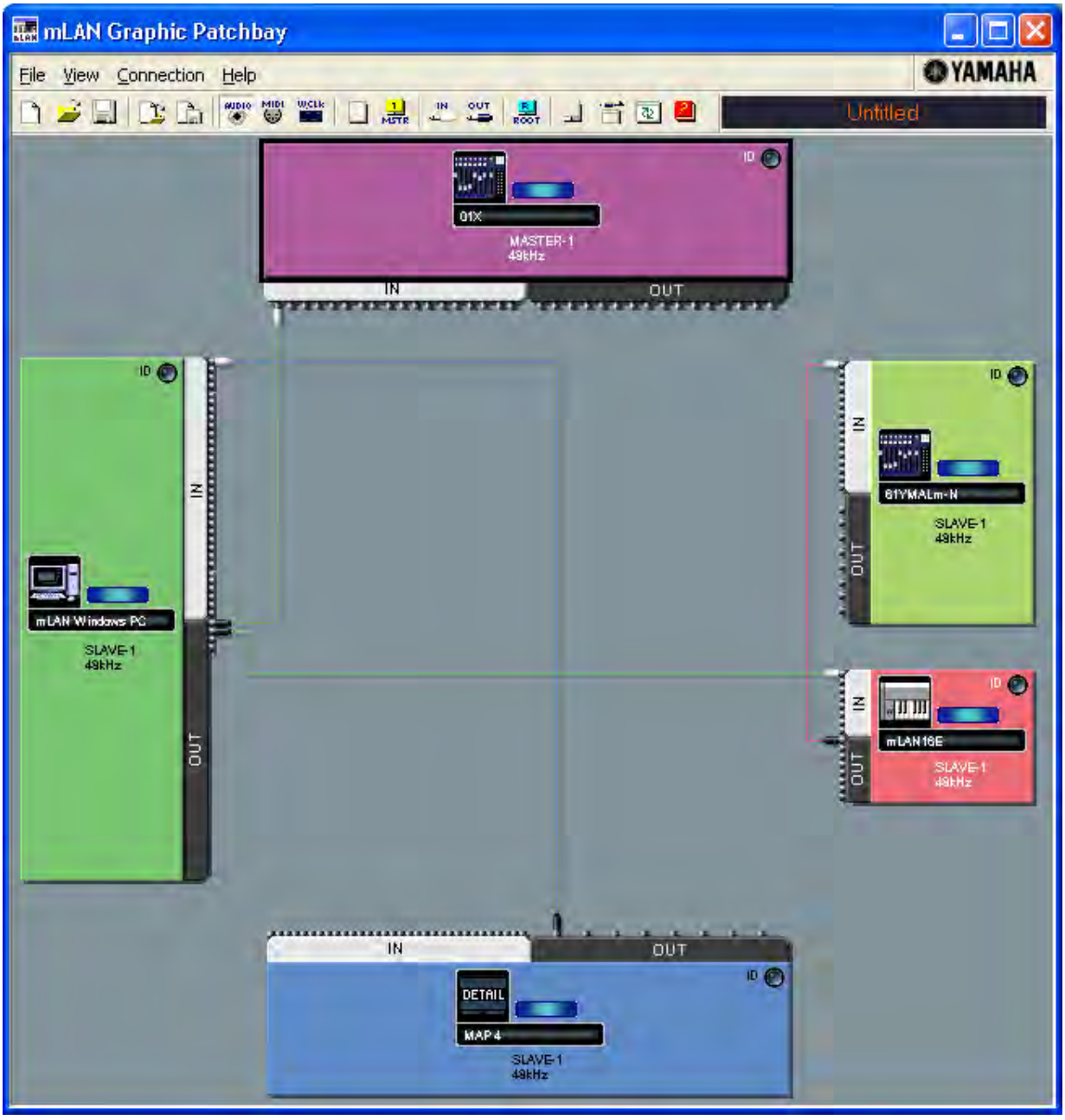


Figure 2.6: Snapshot of mLAN Graphic Patchbay

2.2 Motivation for Further Investigation of IEEE 1394 Audio Networking with mLAN

The rest of this thesis focuses on mLAN and its Plural Node Architecture implementation. The IEEE 1394 high performance serial bus, in its standard form, provides all the required capabilities for real-time audio transmission. In particular, quality of service is guaranteed, since IEEE 1394 was specifically designed for multimedia data transmission. As a result, mLAN devices can co-exist with other IEEE 1394 devices on the same network without need for any special constraints to ensure that network performance is not compromised. Although IEEE 1394 is a specialised network infrastructure, results from a survey conducted by the AES TC-NAS show that there is lack of preference between the use of standalone infrastructure or ubiquitous data communications infrastructure already in place [Gross, 2006].

As mentioned above, mLAN is based on open industry standards. Some companies, other than Yamaha Corporation, that are providing mLAN devices include: PreSonus Audio Electronics, TerraTec Electronic, Apogee Electronics Corporation, and Kurzweil Music Systems. The use of a plug-in mechanism by the Enabler, as described in section 2.1.1.5, abstracts the underlying hardware specifics from the Enabler, while providing a means for interoperability across vendors.

The capability of viewing a workstation as an "mLAN device", through the use of drivers such as WDM and ASIO, greatly enhances the use of mLAN in recording studios. Unlike most existing solutions, mLAN networks also facilitate the transmission of MIDI data in addition to audio and word clock synchronisation data.

This investigation was carried out within the Audio Engineering Research Group in the Department of Computer Science at Rhodes University, South Africa. This research group has, for some time, been working in collaboration with Yamaha Corporation, Japan, in enhancing the mLAN audio network technology.

2.3 Chapter Summary

This chapter has shown how the readiness for audio networking in all audio related contexts has resulted in a number of digital audio network solutions from various vendors. Most of these solutions build upon existing Ethernet data communications infrastructure. Above Ethernet, most vendors have introduced their own proprietary techniques that guarantee QoS and provide interfaces with conventional audio equipment. Such proprietary techniques have resulted in the loss

of interoperability across vendors. Four Ethernet based solutions have been described in order to highlight their capabilities. mLAN, an IEEE 1394 based audio network technology that was created by Yamaha Corporation, has also been described.

Motivations have been provided to show why we chose to investigate mLAN audio networking. These motivations include the favourable nature of IEEE 1394 with regards to audio and video transmission, an open-standards-based approach, capability for transmission of both audio and MIDI data, capability to stream audio between a PC and any of the devices in an mLAN network, and the on-going collaborative relationship with Yamaha Corporation, the creators of mLAN, in improving the technology.

The next chapter, Chapter 3, provides a more detailed overview of IEEE 1394 and mLAN technologies. The concepts introduced in section 2.1.1.5 are explored further.

Chapter 3

IEEE 1394 and music Local Area Network Technology

Chapter 2 has introduced the underlying concepts of mLAN technology. In particular, mLAN is primarily based on the IEEE 1394 and IEC 61883-6 specifications. This chapter focuses on IEEE 1394 and mLAN in more detail. An overview of the IEEE 1394 architecture and its features is given. These features include the nature of data transmission and the underlying transmission protocol. A description of mLAN will follow, where the first and second generations of mLAN are explored, together with reasons for moving from the first to the second generation. Second generation mLAN is the foundation of our investigation and therefore warrants a more detailed description. In this regard, the Plural Node Architecture (introduced in section 2.1.1.5) will be highlighted. In the context of the Plural Node Architecture, the Open Generic Transporter (OGT) concept is introduced. The OGT concept sets the context for the next chapter, Chapter 4, which highlights the shortcomings that our investigation aimed to resolve.

3.1 IEEE 1394 Architecture Overview

Section 2.1.1.5 on page 18 has given a brief introduction to some of the capabilities of the IEEE 1394 high performance serial bus that make it desirable for use in real-time audio transmission. Some of these capabilities include, but are not limited to [Anderson, 1999]:

- Scalable performance through support of speeds of 100 Mb/s, 200 Mb/s, 400 Mb/s, 800

Mb/s, 1.6 Gb/s, and 3.2 Gb/s.[1]

- Plug and play support through the self-configuration process that is done by the nodes.
- Hot insertion and removal of devices on the network without requiring the system to be powered down.
- Support for isochronous and asynchronous transfers which ensure that isochronous applications are guaranteed constant bandwidth.
- Peer-to-peer communication which eliminates host processor/memory bottlenecks.

This section focuses on some of the internals of the IEEE 1394 high performance serial bus that allow for the abovementioned capabilities. An overview of the ISO/IEC 13213 specification and the IEEE 1394 protocol layers is given.

3.1.1 The ISO/IEC 13213 (ANSI/IEEE 1212) Specification

This specification, more formally known as "Information Technology - Microprocessor Systems - Control and Status Registers (CSR) Architecture for Microcomputer Buses" [IEC, 1994], defines a set of core features that are implemented by a variety of buses, including IEEE 1394 [Anderson, 1999]. For simplicity, this specification is referred to as the "CSR architecture". The specification defines a number of features that attempt to reduce the amount of customised software required to support a bus standard, enable interoperability between bus nodes based on different platforms, provide support between different bus types, and improve software transparency between multi-bus implementations.

Some of the features defined by the CSR architecture and implemented by the IEEE 1394 specification include: node architectures, address space, common transaction types, Control and Status Registers (CSRs), configuration ROM structure, and message broadcast mechanisms to all nodes. Each of these functions is described below, as adapted from Anderson [1999].

3.1.1.1 Node Architecture

The logical and physical organisation of devices attached to a CSR architecture compliant bus is represented by *modules, nodes*, and *units*, as shown in Figure 3.1. A module contains one or

[1] At the time of this writing, there were no known Plural Node devices that implemented speeds of 1.6 Gb/s and 3.2 Gb/s

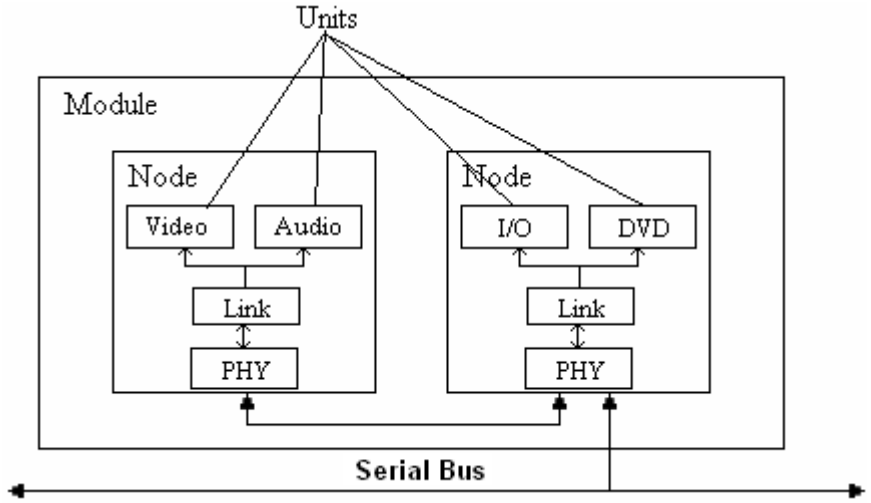


Figure 3.1: Module, Node, and Unit Architecture

more nodes and represents a physical device attached to the bus. Nodes are logical entities within a module that are visible to initialisation software and contain CSRs and ROM entries that are mapped into the *initial node address space* (see section 3.1.1.2 below). Most of the CSRs and ROM entries are only required for system initialisation. Units are functional subcomponents of a node that identify processing, memory, or I/O functionality. Units typically operate independently and are controlled by their own software drivers. Unit registers are mapped into the *node address space* and are accessible via unit specific software drivers. A description of the *node address space* follows.

3.1.1.2 Node Address Space

32- and 64-bit addressing models are defined by the CSR architecture. However, the IEEE 1394 specification only supports the 64-bit addressing model. Up to 1024 buses with up to 64 nodes on each bus are supported. Figure 3.2 shows the 64-bit address space that the IEEE 1394 specification implements. As shown in Figure 3.3, the most significant 10 bits (bits 0 to 9) identify the target bus, while the next six bits (bits 10 to 15) identify the target node on the specified bus. The least significant 48 bits (bits 16 to 63) define the 256 terabytes (TB) of memory address space for each node. The 256 TB address space for each node is divided into the following blocks (as shown in Figure 3.2):

- Initial memory space

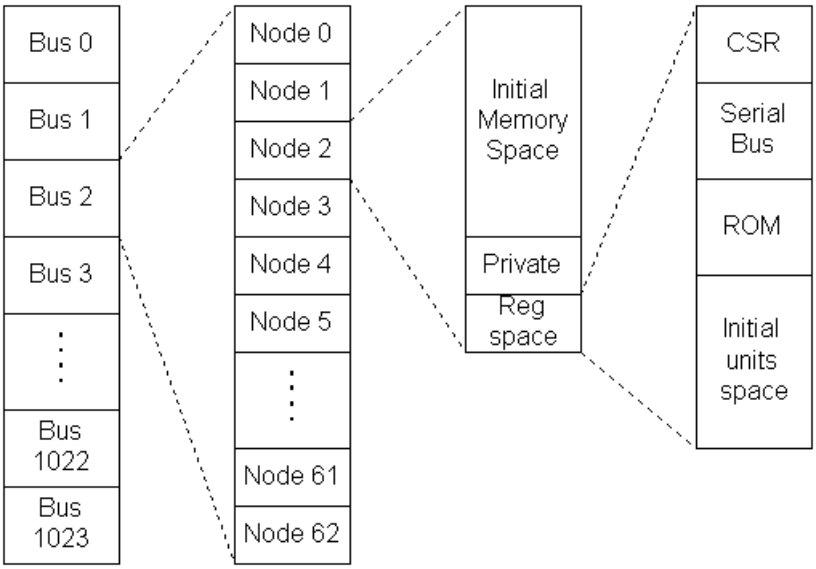


Figure 3.2: 1394 Node Address Space

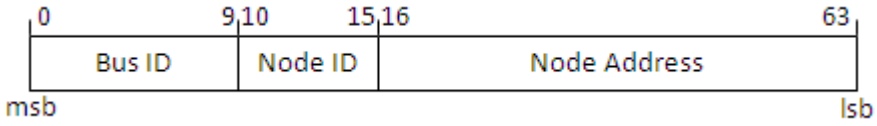


Figure 3.3: 64-bit Address Components

- Private space
- Initial register space (shown as Reg space in Figure 3.2)
    - CSR architecture register space
    - Serial bus space
    - ROM
    - Initial units space

The *private space* is reserved for a node's local use while the *initial register space* defines standardised locations used for serial bus configuration and management [Anderson, 1999].

3.1.1.3 Transfers and Transactions

As mentioned in section 2.1.1.5 on page 18, there are two main forms of transfers, namely asynchronous and isochronous transfers. The former applies to transactions where guaranteed

delivery is of prime importance, while the latter applies to transactions where data needs to be transmitted at a constant rate. Asynchronous and isochronous transfers are performed simultaneously by sharing the overall bus bandwidth. Bandwidth allocation is based on 125 μs intervals that are known as *cycles.*

Asynchronous transfers are packaged into asynchronous packets that use an explicit 64-bit address to target a particular node. *Read, write* and *lock* transactions, as described in section 2.1.1.5 on page 18, comprise asynchronous transfers. All asynchronous transfers are guaranteed at least 20% of the overall bus bandwidth. Nodes wishing to initiate asynchronous transactions are not guaranteed any specific amount of bus bandwidth but are rather guaranteed fair access to the bus via a *fairness interval*. Here, each node with pending asynchronous transfers gets access to the bus exactly once during during a single fairness interval. The maximum asynchronous packet payload sizes are shown in Table 3.1. The serial bus also provides CRC error checking

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

Table 3.1: Maximum Payload Size for Asynchronous Transfers [Anderson, 1999]

mechanisms and response codes that are used to verify the integrity of data that is transferred. These error checking mechanisms are useful in generating confirmation responses (acknowledgements), directed to the initiator of a transfer, regarding the success or failure of the transfer. Acknowledgements may result in the initiator retrying the transfer in the event of failure.

Isochronous transfers are packaged into isochronous packets that target one or more devices based on a 6-bit channel number associated with the transfer. The channel number is used by target nodes (*isochronous listeners*) to access a memory buffer, optionally residing in the node's 256 TB address space, within the host application. Section 3.1.2 describes the role of the host application within an IEEE 1394 node. *Isochronous talkers* (transmitting devices) are required to request bus bandwidth, allocated on a per cycle basis, from an *isochronous resource manager (IRM)* node. If bandwidth is available for an isochronous transfer, the corresponding channel receives a guaranteed timeslice during each 125 μs cycle. Up to 80% (100 μs) of the overall bus bandwidth may be allocated to isochronous transfers. Unlike asynchronous transfers,

isochronous transfers are not acknowledged by target nodes, since the rate of transfer is more important than guaranteed delivery. The maximum isochronous packet payload sizes are limited by either the available bus bandwidth or the sizes specified in Table 3.2.

Transmission Speed	Maximum Data Payload Size (Bytes)
100 Mb/s	1024
200 Mb/s	2048
400 Mb/s	4096
800 Mb/s	8192
1.6 Gb/s	16384
3.2 Gb/s	32768

Table 3.2: Maximum Payload Size for Isochronous Transfers [Anderson, 1999]

Figure 3.4 shows the structure of an isochronous packet. The *data length* field specifies the pay-

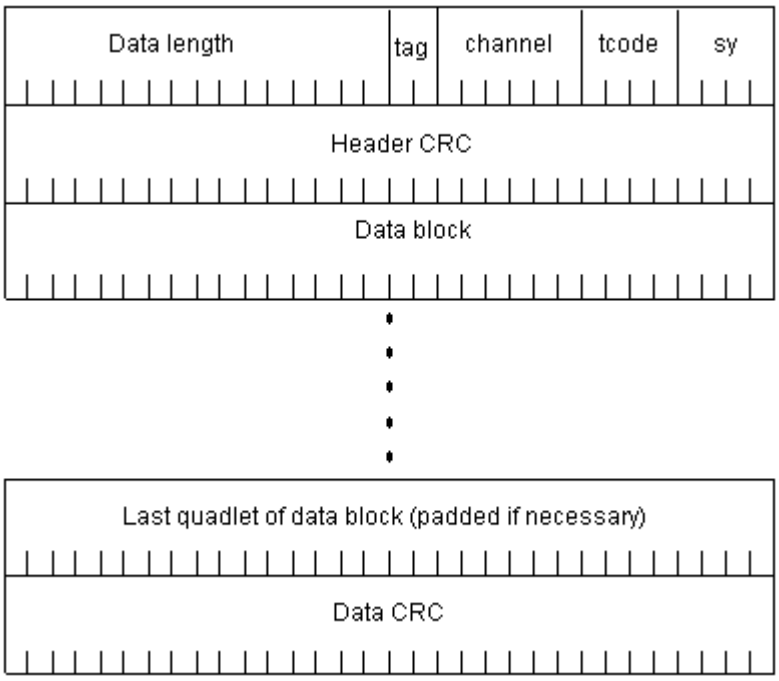


Figure 3.4: Isochronous Packet Structure

load size of the packet in bytes. The isochronous data format *tag* is used to indicate the format of the packet's data, such as the Common Isochronous Packet (CIP) format that is used as the general packet format for audiovisual data over IEEE 1394 [IEC, 2003]. The *channel* field specifies a 6-bit channel number (ranging from 0 to 63 inclusive) mentioned above. The *tcode* field represents the transaction code. This transaction code conveys information that assists isochronous

listeners in determining the nature of the transaction and hence, in determining the format of the remainder of the packet. The *sy* field represents an application-specific synchronisation code.

3.1.1.4 Control and Status Registers (CSRs)

CSRs are a group of core registers that support functions common to all CSR architecture compliant buses. They provide standardised offset locations within the initial register space as described in section 3.1.1.2. Only a subset of the CSRs are required for the IEEE 1394 serial bus as shown in Table 3.3.

Offset (hex)	**Register Name**	**Description**
000	STATE_CLEAR	State and control information.
004	STATE_SET	Sets state clear bits.
008	NODE_IDS	Specifies the 16-bit node ID value.
00C	RESET_START	Resets the state of a node.
	:	
018-01C	SPLIT_TIMEOUT_HI SPLIT_TIMEOUT_LO	Implemented by transaction capable nodes to timeout split transaction retries.
	:	
200-3FC	Serial Bus Dependent	Registers unique to the IEEE 1394 bus.

Table 3.3: CSR Registers Implemented by the IEEE 1394 Bus

As shown above, registers from offset 200 (hex) to 3FC (hex) are reserved for the sole purposes of IEEE 1394. Table 3.4 on page 32 shows a list of these registers. Of importance to us are the CYCLE_TIME, BUS_TIME, BUS_MANAGER_ID, BANDWIDTH_AVAILABLE, and CHANNELS_AVAILABLE registers. We now describe the roles of these registers.

Nodes capable of isochronous transfers implement the CYCLE_TIME and BUS_TIME registers. These registers are initialised by the *bus manager* node, or, in the absence of the bus manager node, the *isochronous resource manager* node . After initialisation, the registers are updated via a 24.576 MHz clock within the node. The bus manager node is responsible for verifying that

Offset (hex)	Register Name	Description
200	CYCLE_TIME	Used by isochronous capable nodes as a common time reference.
204	BUS_TIME	Used by cycle-master capable nodes to keep track of bus time.
208	POWER_FAIL_IMMINENT	Used to notify that power is about to fail.
20C	POWER_SOURCE	Used to validate power failure notifications.
210	BUSY_TIMEOUT	Timeout on transaction retries.
214-218	Not used	Reserved for future use.
21C	BUS_MANAGER_ID	The physical ID of the bus manager node.
220	BANDWIDTH_AVAILABLE	Used to manage isochronous bandwidth.
224-228	CHANNELS_AVAILABLE	A 64-bit mask that keeps track of isochronous channel usage.
22C	MAINT_CONTROL	Used for diagnostics.
230	MAIN_UTILITY	Used for debugging.
234-3FC	Not used	Reserved for future use.

Table 3.4: Serial Bus Dependent Registers for the IEEE 1394 Bus

the *root node* is *cycle master capable* (via the node's config ROM space). The manner in which the root node is determined is described in section 3.1.2.1. A *cycle start packet* is broadcast by the *cycle master* every 125 μs. The cycle master delivers the contents of its CYCLE_TIME register in the cycle start packet and each node will update its CYCLE_TIME register based on this value.

The BUS_MANAGER_ID register is implemented within nodes capable of being the isochronous resource manager. The register contains the node ID of the node that is the bus manager. Non-isochronous resource manager nodes are able to access this register within the isochronous resource manager, and then request bus manager facilities from the bus manager node.

As mentioned in section 3.1.1.3, nodes wishing to perform isochronous transfers ought to acquire the required bandwidth prior to transmission. The BANDWIDTH_AVAILABLE register keeps track of the available bandwidth in terms of *allocations units.* Each allocation unit corresponds to the time required to transfer 32 bits of data when the data rate is 1600 Mb/s, this being, 20 ns. The maximum usable bandwidth on an mLAN network is 4915.20 allocation units [Yamaha Corporation, 2002d]. In addition to obtaining bandwidth, a node must also acquire a free isochronous channel number via the CHANNELS_AVAILABLE register of the isochronous resource manager node before isochronous transmission proceeds.

3.1.1.5 Configuration ROM

Sets of ROM entries are defined to provide configuration information during node initialisation. Information provided by these ROM entries includes specifying node capabilities as well as identifying the software drivers and diagnostic software for a device. Two ROM formats are defined, namely the minimal and general formats. The minimal format only contains a vendor identifier, while the general format contains a vendor identifier, a bus information block, and a root directory containing information entries and/or pointers to other directories such as the ones that Yamaha Corporation introduced for all mLAN devices [Yamaha Corporation, 2003b].

3.1.1.6 Message Broadcast

The CSR architecture defines an optional broadcast mechanism to allow sending of messages to multiple nodes on a bus. All broadcast messages are addressed to a node ID of 63. However, when a node receives a broadcast message, no acknowledgement is sent back to the initiator, since it leads to bandwidth contention on the bus, as all nodes receiving the broadcast message attempt to send acknowledgements at the same time.

3.1.2 IEEE 1394 Protocol Stack

A number of Ethernet based audio network technologies discussed in section 2.1.1 on page 9 build upon the Open System Interconnection (OSI) reference model [McClain, 1991]. This model forms the basis of data network protocols and other standards that enable multivendor equipment interoperability. IEEE 1394 defines its own protocol stack comprising four layers, namely bus management, transaction, link, and physical layers [Anderson, 1999]. Figure 3.5 shows the relationship between these layers.

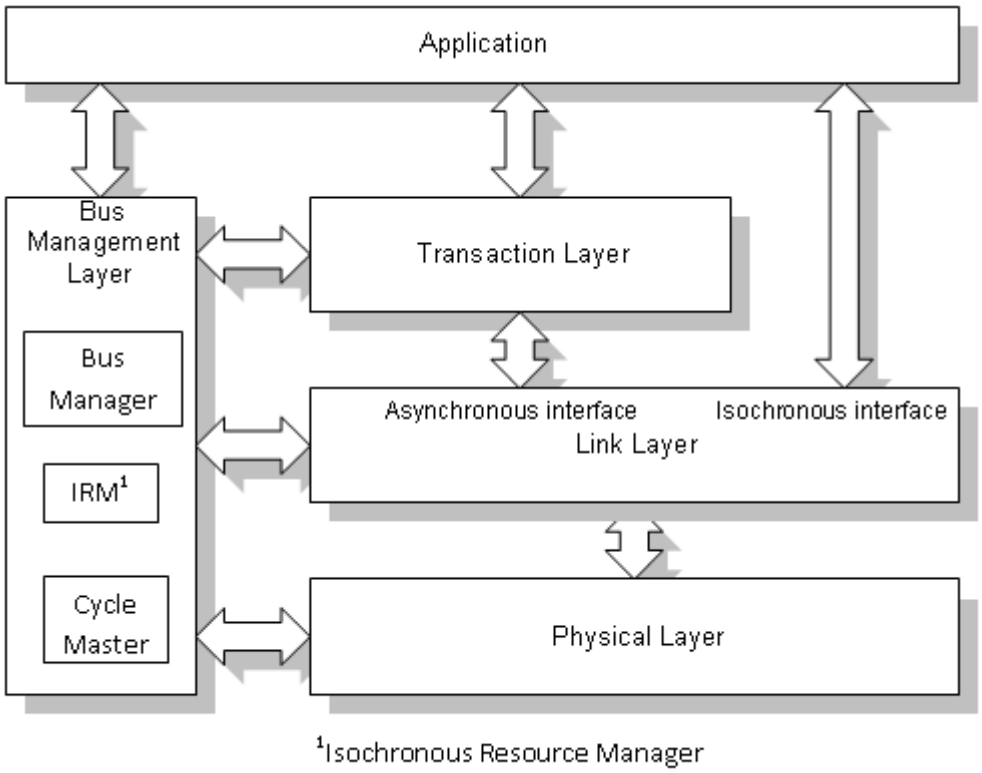


Figure 3.5: IEEE 1394 Protocol Layers

3.1.2.1 Physical Layer

This layer provides the actual interface to the serial bus, and is responsible for receiving bits that make up packets from the serial bus, as well as sending bits that make up packets onto the bus. The physical layer, also known as the PHY, is implemented in chip form and is commonly associated with two or three FireWire ports. An important role of the physical layer is to participate in the *bus configuration process*. This bus configuration process is initiated by power up of a device on the bus, or addition of a new device to the bus, and removal of a device from the bus. There are three main procedures performed during this configuration process, as shown in Figure 3.6. In particular, *bus initialisation (bus reset)* is carried out first, followed by *tree identification,*

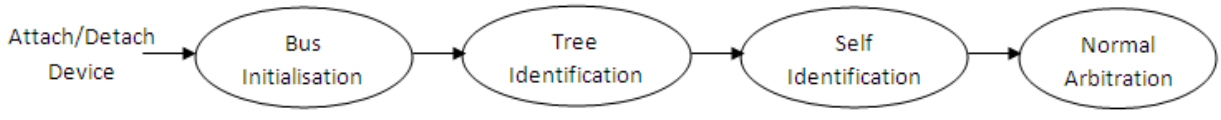


Figure 3.6: Bus Configuration Process

which is then followed by *self identification*.

During bus initialisation, all nodes are forced to return to their initial states in preparation for the tree identification process. Any previously established topology will be cleared during this process. Upon completion of bus initialisation, the tree identification process commences. This process results in one node being identified as the *root node*. When tree identification is complete, the root node coordinates the self identification process. During self identification, each node selects a unique node ID (commonly referred to as a *physical ID*) and broadcasts a *self-ID packet* to announce its identity, in addition to specifying parameters required for bus management. On completion of the self identification process, and thus the bus configuration process, nodes can start performing transactions across the bus. The execution of the bus configuration process may have interrupted other transactions, therefore bus arbitration occurs before any data is transferred in order to determine which node has access to the bus.

3.1.2.2 Link Layer

The link layer is also implemented in chip form. For asynchronous transactions, it provides the interface between the transaction layer and the physical layer, as shown in Figure 3.5. Section 3.1.1.3 mentioned that asynchronous transfers are acknowledged. This creates two roles between nodes involved in such transfers, namely a *requester* and a *responder*. A requester is a node that initiates an asynchronous transaction, while a responder is the node targeted by the requester. The responder is so called because it is required to send an acknowledgement packet back to the requester as notification of success or failure of a transfer. The requester's link layer converts requests from the transaction layer to asynchronous packets, which are forwarded to the PHY for transmission over the IEEE 1394 cable. Requests received by the link layer from the transaction layer contain information such as destination address, transaction type, data length, the data to be transmitted, speed of transmission, retry code, and the transaction label. When asynchronous packets are received by responder's PHY, they are forwarded to the link layer for decoding, and if they are destined for the node they are passed on to the transaction layer.

For isochronous transactions, the link layer provides the interface between the host application (usually an isochronous software driver) and the physical layer, as shown in Figure 3.5. During isochronous transmission, the link layer receives information such as channel number, data length, data to be transferred, and the speed of data transfer from the application. This information is used by the link layer to create isochronous packets that are forwarded to the PHY for transmission over the IEEE 1394 cable. Conversely, when the link layer receives an isochronous

packet from the PHY, it decodes the channel number. If the packet is destined for the node, it is forwarded to a software driver within the host application.

3.1.2.3 Transaction Layer

The transaction layer is a software layer exclusively for handling asynchronous transactions. This layer receives data transfer requests from applications and converts these to one or more transaction requests. Transaction requests are in the form of the read, write, and lock transactions described in section 2.1.1.5 on page 18. On reception, the transaction layer notifies the application of the receipt of the packet.

3.1.2.4 Bus Management Layer

The bus management layer handles bus configuration and management functions for each node. There are three main roles that provide support for a completely managed bus, namely cycle master, isochronous resource manager, and bus manager. Section 3.1.1.4 described some of the services provided by these three roles. In addition, other bus management services that exist include publishing topology and speed maps, enabling the cycle master, and power management control. A topology map gives bus generation information, a summary of the nodes on a bus, and their IDs. Speed map information is used to get the maximum allowable transaction speed between nodes. Power management control is necessary when there are nodes that require power from the bus. The bus manager node may also update the BANDWIDTH_AVAILABLE register of the isochronous resource manager node in order to reserve the 20% bus bandwidth for asynchronous transfers.

3.2 Audio and Music Data Transmission Protocol

As mentioned in section 2.1.1.5 on page 18, audio and MIDI data transmission on an mLAN network complies with the IEC 61883-6 specification [IEC, 2005]. This specification is based on a document initiated by Yamaha Corporation but later adopted by the 1394 Trade Association (1394 TA). This 1394 TA document is known as the “Audio and Music Data Transmission Protocol” [1394 Trade Association, 2002]. The IEC 61883-6 specification is the sixth part in a series of IEC standards (IEC 61883) that relate to the transmission of audio, video, and multimedia data over IEEE 1394 .

3.2.1 The Common Isochronous Packet (CIP) Format

The IEC 61883-1 specification [IEC, 2003] is the first part of the IEC 61883 standards. Among other things, it defines the general packet format for audiovisual data over IEEE 1394. This general packet format is known as the *Common Isochronous Packet (CIP)* format. The IEC 61883-6 makes use of this CIP format for transmission of audio and MIDI data. Figure 3.7 shows how the CIP builds upon the standard IEEE 1394 isochronous packet that has been shown in Figure 3.4. The CIP header fields that are of importance to us include S*ID, DBS, DBC, FMT,*

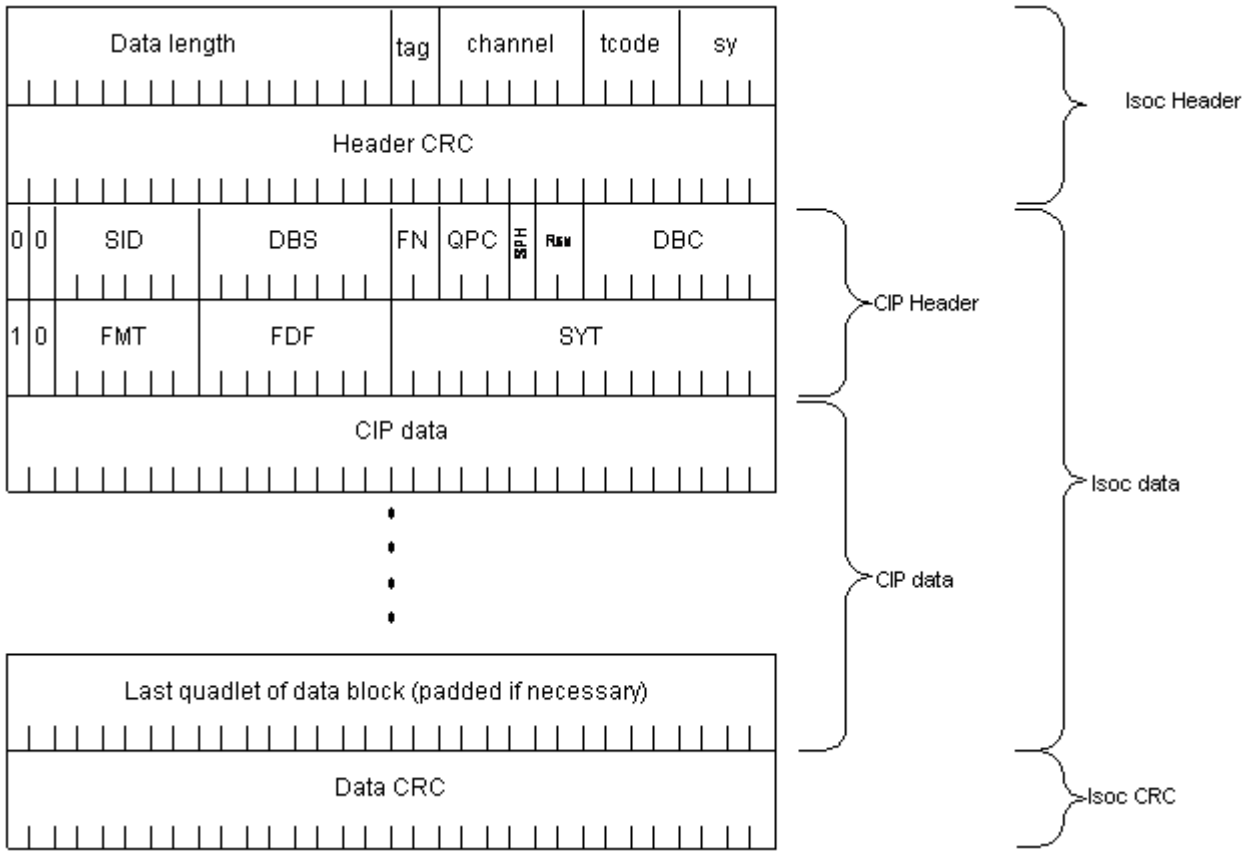


Figure 3.7: The Common Isochronous Packet Structure

FDF, and *SYT.* A description of these fields follows.

SID: This is the source identifier which represents the 6-bit physical ID of the node that sent the packet.

DBS: This is the data block size. Audiovisual data in the CIP data section is clustered into one or more data blocks.

DBC: A running count of the transmitted data blocks.

FMT: Identifies the type of data being transmitted within the CIP data section.

FDF: This is the format dependent field used to identify sub-formats such as the *event type* (described in 3.2.2) and to convey other information such as sampling frequency.

SYT: A field indicating the time at which a particular data block (*event*) within the packet should be presented at the receiver.

The most important concept in the IEC 61883-6 specification, as identified by Fujimori and Foss in "A New Connection Management Architecture for the Next Generation of mLAN" [Fujimori and Foss, 2003], is that of a *sequence*. An isochronous packet comprises a number of *packet clusters* known as data blocks. Each data block contains audio samples and MIDI messages that occur at a particular point in time. This simultaneous availability of audio samples and MIDI messages on the transmitter is called an *event*. Audio samples and MIDI messages within each data block are arranged in order, where each position within the data block is known as a sequence. Thus, each sequence corresponds to a particular position within an isochronous packet cluster. Figure 3.8 shows an example of an isochronous packet with six data blocks.

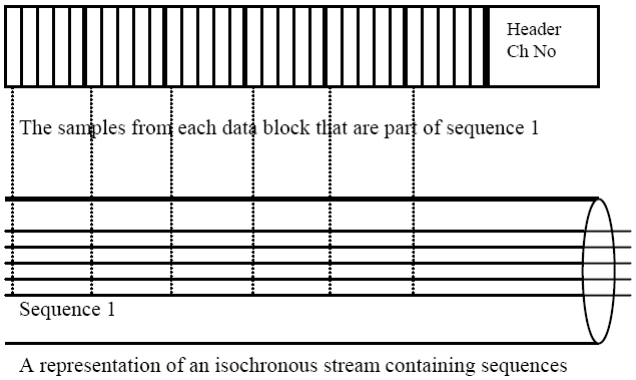


Figure 3.8: An Isochronous Stream with Sequences

Six data blocks is the typical number at 48 kHz, since isochronous packets are transmitted at 8 kHz (every 125μs). Each data block in the isochronous packet has a number of quadlets (32-bit elements). These quadlets are the AM824 (8-bit label and 24-bit Audio/Music data) formatted audio samples and/or MIDI messages, where the position of a quadlet corresponds to the sequence that it belongs to. Figure 3.8 shows five quadlets per data block.

Another important aspect to note is that up to eight MIDI data streams may be carried within a single sequence [IEC, 2005]. This is achieved by multiplexing the MIDI messages from the MIDI data streams onto a single sequence (position) within the successive data blocks. Figure 3.9 illustrates MIDI multiplexing, where the fourth data block position has been assigned to a MIDI sequence. Successive quadlets at this sequence position will contain MIDI messages

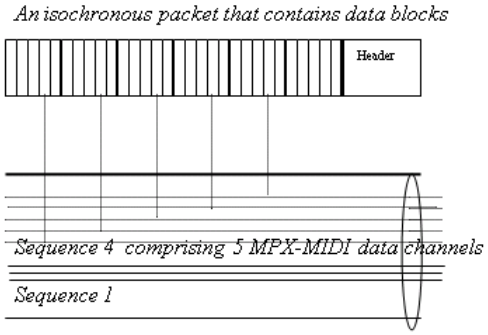


Figure 3.9: Multiplexing MIDI Data Streams

of successive data streams. The diagram shows messages from five MIDI data streams within a single sequence. Since up to eight MIDI data streams may be multiplexed within a single sequence, every eighth data block will have, at position four, the next message for the first MIDI data stream. Multiplexing of MIDI is possible, since the rate of MIDI transmission is lower than that required for audio.

Nodes may transmit CIP packets containing data or empty packets. There are two main packet transmission methods, namely *blocking* and *non-blocking* transmission [1394 Trade Association, 2002]. In non-blocking transmission, a non-empty CIP packet is transmitted when one or more events arrive in a nominal isochronous cycle. Blocking transmission, on the other hand, waits until a fixed number of events have arrived before transmitting the CIP packet.

3.2.2 Audio and MIDI Data Transmission Formats

As mentioned in section 3.2.1 above, audio and music data is transmitted in a format known as AM824. The AM824 is one of three *event types* defined by the IEC 61883-6 specification [IEC, 2005]. The other two event types are *32-bit floating point data* and *24-bit * 4 Audio Pack*, and are described in more detail in the IEC 61883-6 specification. We focus on the AM824 event type, since it is commonly used within mLAN audio networks. The generic format of AM824 data is shown in Figure 3.10. Receivers capable of receiving AM824 data are required to check

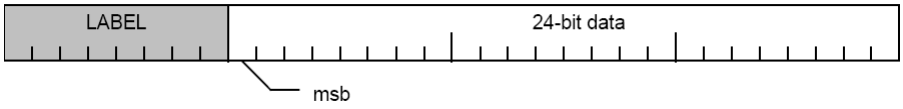


Figure 3.10: Generic AM824 Format [IEC, 2005]

the label for each AM824 sequence being received.

Different label values have different meanings based on the type of data transmitted. For example, while audio and MIDI data may both be AM824 formatted, their labels are necessarily different in order to distinguish the two. Table 3.5 lists the various AM824 label values and gives descriptions of the data *subformats* that they represent. The values in the table are indicated as

Value	**Description**
00_{16}- $3F_{16}$	IEC 60958 Conformant
40_{16}- $4F_{16}$	Multi-bit Linear Audio
50_{16} - 57_{16}	One bit audio (plain)
58_{16} - $5F_{16}$	One bit audio (encoded)
60_{16} - 67_{16}	High precision multi-bit linear audio
70_{16} - $7F_{16}$	Reserved
80_{16} - 83_{16}	MIDI Conformant
84_{16} - 87_{16}	Reserved
88_{16} - $8B_{16}$	SMPTE Time Code Conformant
$8C_{16}$ - $8F_{16}$	Sample count
90_{16} - BF_{16}	Reserved
$C0_{16}$ - EF_{16}	Ancillary data
$F0_{16}$ - FF_{16}	Reserved

Table 3.5: AM824 LABEL Definition

ranges between two hexadecimal numbers. This allows the different subformats to be further qualified and convey meaningful information about the data to the receiver, such as the number of valid bits within a 24-bit data segment.

We have shown how audio samples and MIDI data are packaged into packets for transmission on the serial bus. In addition to this, the IEC 61883-6 specification defines a mechanism to ensure that the audio samples have a similar time relationship to each other, at the receiver side, as they did when they were generated at the transmitter side. This mechanism takes into account the delay in packet transmission from a source device to a destination device, and also the fact that audio sampling clock frequencies on the two devices, while nominally the same, differ slightly. The next section describes this mechanism.

3.2.3 Transmission of Timing Information

The IEC 61883-6 specification defines an implementation-dependent mechanism for the synchronisation of a slave device's clock to a master device's clock [IEC, 2005]. This mechanism mainly depends on the *cycle master/cycle slave* relationship described in section 3.1.1.4 as well as the SYT field of the CIP header shown in Figure 3.7.

When an mLAN device transmits isochronous packets that contain audio samples, it time stamps these packets with a value that is read from its CYCLE_TIME register (see section 3.1.1.4), with a delay offset (TRANSFER_DELAY) added to it. This value (CYCLE_TIME + TRANSFER_DELAY), also known as the presentation time, is held within the SYT field of the CIP header. The default TRANSFER_DELAY value is 354.17 μs + 125 μs. This default value takes into account the maximum latency time of CIP transmission through an arbitrated bus reset. A receiving device extracts this time stamp and uses it to recreate the sampling frequency of the transmitted audio samples. Phase locked loop (PLL) techniques are usually used for this sampling frequency recreation. However, a discussion of PLL techniques is beyond the scope of this thesis.

3.3 mLAN (music Local Area Network) Technology

We have described the fundamental standards on which mLAN is based, as specified by the IEEE 1394 and IEC 61883-6 specifications. Section 2.1.1.5 on page 18 introduced the concept of the Plural Node Architecture. Second generation mLAN devices are based on the Plural Node Architecture and these were the focus of our investigation.

This section gives an overview of the first and second generations of mLAN. We focus more on the second generation of mLAN. In particular, we give a detailed overview of the Plural Node Architecture. In the context of the Plural Node Architecture, the concept of the Open Generic Transporter (OGT) is introduced. The OGT concept sets the context for the next chapter, Chapter 4, which fully describes the need for our investigation.

3.3.1 First Generation mLAN (mLAN Version 1)

This generation of mLAN audio networks is based on the IEEE 1394 Trade Association's AV/C (Audio Video Control) protocol [1394 Trade Association, 2001b]. In particular, this mLAN

implementation uses a set of vendor dependant commands that are part of the AV/C protocol suite. A set of data structures known as descriptors and info blocks reside in the devices and are implemented by the processor and ROM [1394 Trade Association, 2001a], as shown in Figure 3.11. These descriptors and info blocks hold information about plugs such as channel number,

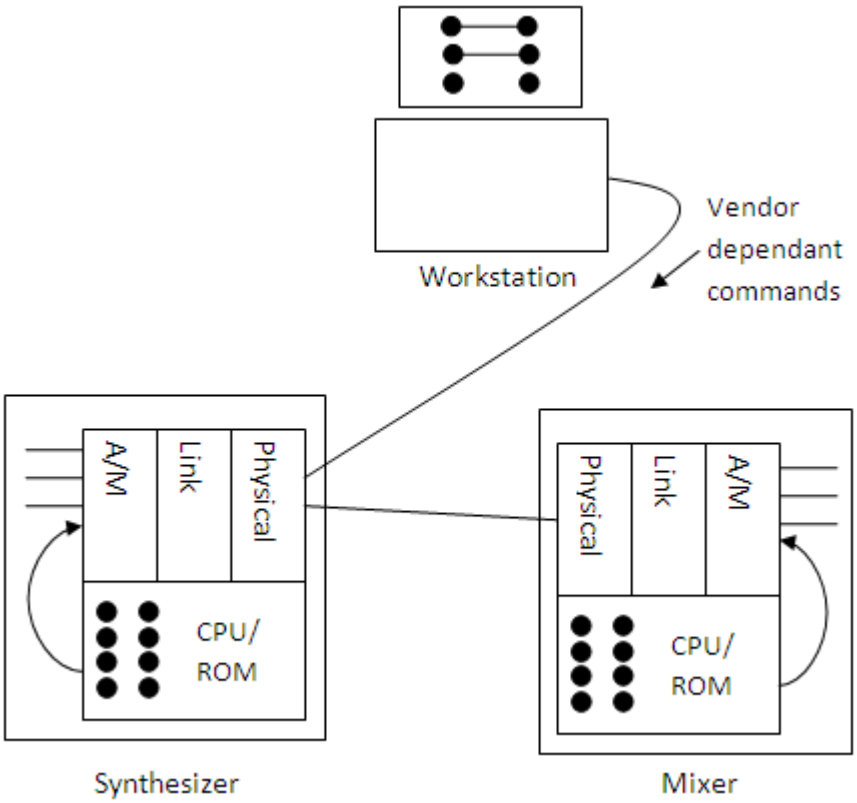


Figure 3.11: mLAN Version 1 Architecture [Fujimori and Foss, 2003]

sequence number to receive, type (MIDI or audio), and the connections between plugs. There are also word clock plugs that enable sampling clock synchronisation relationships. With reference to Figure 3.11, connection management requests from the workstation's graphical user application are fulfilled by sending vendor dependant commands to devices on the IEEE 1394 bus.

The main advantage of this architecture is that mLAN plug abstractions are implemented within the devices. Hence, the state of the plugs can be stored in non-volatile memory within the devices [Fujimori and Foss, 2003]. This allows devices, without any external intervention, to restore their previous state after a power cycle. In spite of such an advantage, a number of problems with this architecture led to the creation of the second generation of mLAN. These problems include:

- The implementation of plug abstractions within the device requires a lot of memory, and hence, raises the cost of providing mLAN compatibility.
- Any changes in implementation, such as bug fixes and upgrades, require firmware upgrades in all mLAN compatible devices.
- Non-mLAN chip vendors wishing to provide mLAN compatibility within their chipsets require considerable effort and mLAN expertise to provide the necessary mLAN plug abstractions.
- Not all applications require the level of complexity provided by mLAN plug abstractions, especially in fixed installations that may only need an initial setup using isochronous channel number and sequence selections.

3.3.2 Second Generation mLAN (mLAN Version 2) and the Plural Node Architecture

This is the current generation of mLAN. It is based on the Plural Node Architecture, and was created to resolve the problems identified in section 3.3.1 regarding the first generation of mLAN. As described in section 2.1.1.5 on page 18, and also shown in Figure 3.12, the Plural Node Architecture splits connection management of IEEE 1394 audio devices between an Enabler node and a Transporter node. As a result of this split, the Plural Node Architecture is also known as the Enabler/Transporter Architecture. This approach allows for flexible upgrades, and the capability for hardware vendors to easily make their hardware mLAN compatible.

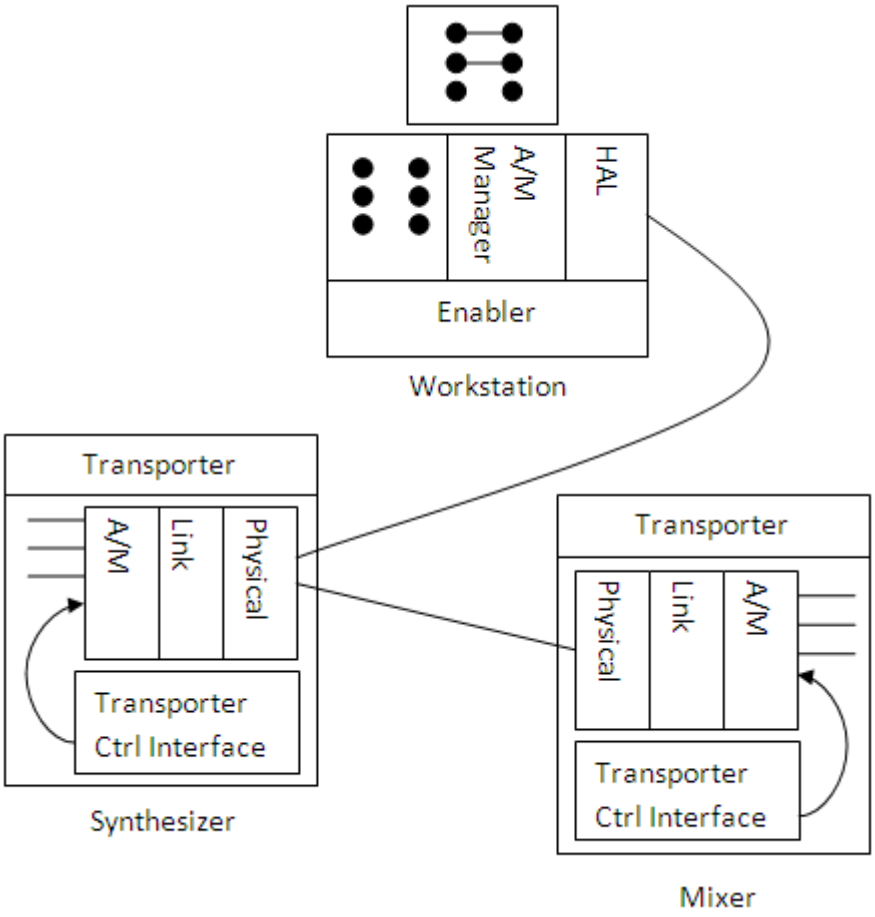


Figure 3.12: mLAN Version 2 Architecture [Fujimori and Foss, 2003]

3.3.2.1 The Enabler

As shown in Figure 3.12, the Enabler comprises three layers, namely the mLAN Plug Abstraction, A/M Manager, and Hardware Abstraction layers. Figure 3.13 shows a representation of these layers. A detailed discussion of an Enabler can be found in a PhD thesis entitled "High Speed End-To-End Connection Management in a Bridged IEEE 1394 Network of Professional Audio Devices" [Okai-Tettey, 2005]. This Enabler, a product of a Yamaha Corporation initiative, was created by Okai-Tettey [2005], and was used for our investigation. In the remainder of this thesis, we refer to this Enabler as the Redesigned Enabler. In cases where reference is made to Yamaha Corporation's original Enabler implementation, we use the term Basic Enabler to make the distinction. A brief description of each of the Enabler layers follows.

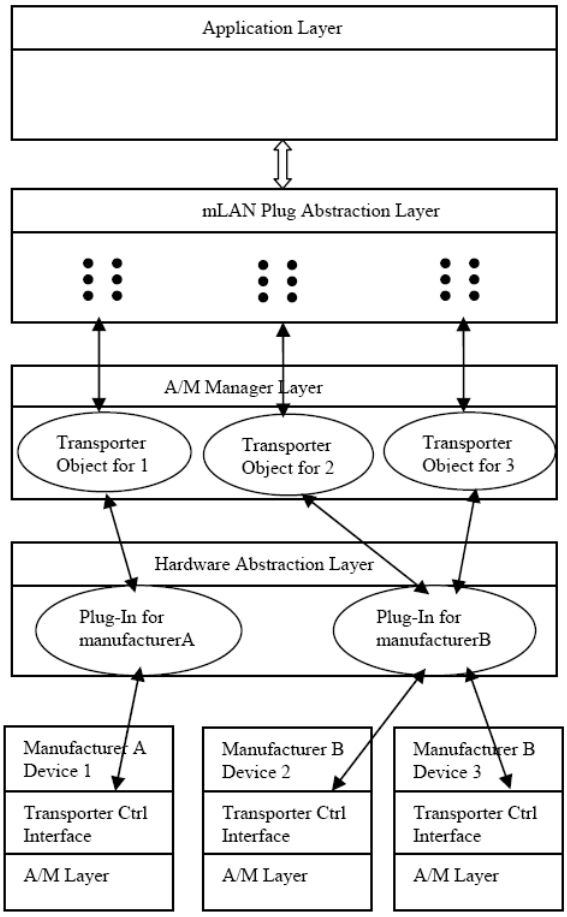


Figure 3.13: The Enabler's Layers and Interfaces

mLAN Plug Abstraction Layer

This is the top layer which implements a number of mLAN plugs. These plugs can be viewed as *terminators* for the audio and music data sequences that are transmitted and received by Transporters. In particular, this layer implements input and output mLAN plug abstractions for all possible sequence end-points on all Transporters under the Enabler's control. An API (Application Programming Interface) that may be used by connection management applications is provided by this layer.

A/M Manager Layer

Located below the mLAN Plug Abstraction layer, this layer is responsible for reading audio and music data transmission and reception parameters from the associated Transporters, and updating

these parameters in response to requests from the mLAN plug abstraction layer. Each Transporter under the control of the Enabler has a Transporter object that keeps its state information and handles requests from both the mLAN plug abstraction layer and the actual Transporter.

Hardware Abstraction Layer

This layer provides the Enabler with a uniform interface to different Transporter implementations [Yamaha Corporation, 2002c]. Non-mLAN chip manufacturers acquire mLAN compliance via this layer, in conjunction with the Transporter Control Interface. The role of the Transporter Control Interface is described in section 3.3.2.2. There is variation in the way in which parameters are set up for A/M transmission and reception across manufacturers. Consequently, each manufacturer provides a software plug-in that translates A/M configuration requests from the Enabler's A/M Manager layer into proprietary requests that the Transporter Control Interface of that particular manufacturer will understand. The Enabler discovers the various device node types , and loads device-specific plug-ins that can communicate, at a low level, with their respective Transporters. Currently, Microsoft's Component Object Model (COM) is used in the Windows implementation of the Enabler as the mechanism for plug-in loading [Yamaha Corporation, 2004b]. Linux and Macintosh implementations follow the same design as used for Windows but make use of a shared library mechanism instead of COM. Yamaha Corporation has defined a Transporter Hardware Abstraction Layer (HAL) Application Programming Interface(API) that manufacturers can use to create software plug-ins for their devices. The current Transporter HAL API was initially designed to access the capabilities of Yamaha Corporation's chips (ASICs). This Transporter HAL API is fully described in Appendix A. In addition to knowing the capabilities exposed by the Transporter HAL API, manufacturers also need to know the mechanism used by the Enabler to load their plug-ins. This plug-in loading mechanism is described in Chapter 4.

3.3.2.2 The Transporter

The Transporter, typically residing in an IEEE 1394 audio device, is dedicated to isochronous data transmission and reception [Yamaha Corporation, 2002a]. Transporters may operate in one of two modes, namely "B Mode" and "B-Pro Mode"[Yamaha Corporation, 2002c]. In particular, a Transporter that always requires an Enabler for A/M data transmission is called a "B Mode" Transporter. On the other hand, a Transporter that is capable of Connectionless Isochronous

Transmission (CIT) is called a "B-Pro Mode" Transporter. In CIT, a Transporter broadcasts or receives A/M data on isochronous channels that are restored from the device's non-volatile memory.

Figure 3.14 shows the components of a Transporter node. In its simplest form, a Transporter

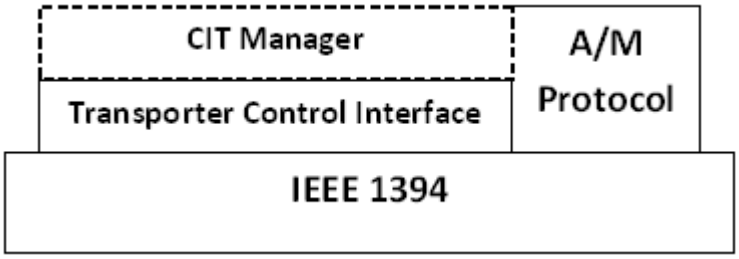


Figure 3.14: Components of a Transporter Node [Yamaha Corporation, 2002c]

consists of an A/M Protocol Layer, and a Transporter Control Interface for modifying the A/M Protocol Layer parameters via the serial bus. The Transporter Control Interface provides an implementation of the node's *private space*. Section 3.1.1.2 has described the *private space* as the section of a node's address space that is reserved for the node's local use. The *private space* also points to other configuration spaces that are implemented by a Transporter. The Enabler accesses these configuration spaces via the Transporter Control Interface. A Transporter depends on an Enabler to set its A/M Protocol Layer parameters for data transmission and reception. Transporters capable of CIT implement an additional component, namely the CIT Manager. When a device is powered up, the CIT Manager loads the A/M Protocol Layer of the Transporter with an entire set of A/M parameters from the Transporter's non-volatile memory known as the Boot Parameter Memory. An Enabler is responsible for storing data in the Boot Parameter memory.

A number of chips (ASICs) that are responsible for the encapsulation (at the transmitter side) and subsequent extraction (at the receiver side) of audio and MIDI data in accordance with the A/M Data Transmission Protocol (IEC 61883-6 specification) have been created. Yamaha Corporation, for example, have created the "mLAN-NC1" [Yamaha Corporation, 2001] and "mLAN-PH2" [Yamaha Corporation, 2003a] chips. Each of the chips has a transmission FIFO that receives audio or MIDI data from its input pins, packages the data into isochronous packets and sends out the packets onto the IEEE 1394 bus. For data reception, there are FIFO buffers that receive audio samples from the audio sequences and MIDI messages from the MIDI sequences. Selection of an isochronous stream and sequence within that stream is done via a pair of registers

for each buffer as shown in Figure 3.15. The channel register (shown as "Chan" in the dia-

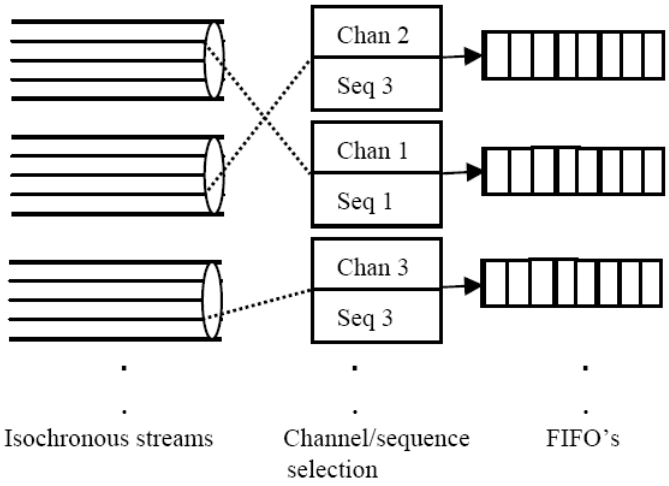


Figure 3.15: mLAN Sequence Selection [Fujimori and Foss, 2003]

gram) specifies the isochronous channel number of the isochronous packets that are received by the FIFO buffer, while the sequence number register (shown as "Seq" in the diagram) indicates the position of the sequence that is extracted by the FIFO buffer from each isochronous packet cluster of the received isochronous packets.

3.3.3 The Open Generic Transporter Guideline Document

In the context of the Plural Node Architecture, the Audio Engineering Society's SC-02-12-G Task Group has produced an Open Generic Transporter (OGT) guideline document in order to ease the task of manufacturers [Audio Engineering Society - Standards Committee, 2005; Kounosu, Fujimori, Laubscher, and Foss, 2005]. If manufacturers design Transporter Control Interfaces conforming to the Open Generic Transporter guideline document, there will be no need to create the manufacturers' own HAL plug-ins for the Enabler, since they can make use of a common, open, generic Transporter HAL plug-in.

The OGT guideline document indicates a number of functional blocks that comprise a generic Transporter, and can be controlled via a set of registers [Kounosu et al., 2005]. This set of registers and their locations comprise the "open" interface. The Enabler makes use of this interface to control a Transporter. Manufacturers are not required to utilise the OGT guideline and the option to create a Transporter and HAL plug-in with a closed Transporter-HAL interface remains available.

3.3.3.1 Generic Transporter Architecture

The generic Transporter is modelled by a Generic Transporter Block as shown in Figure 3.16.

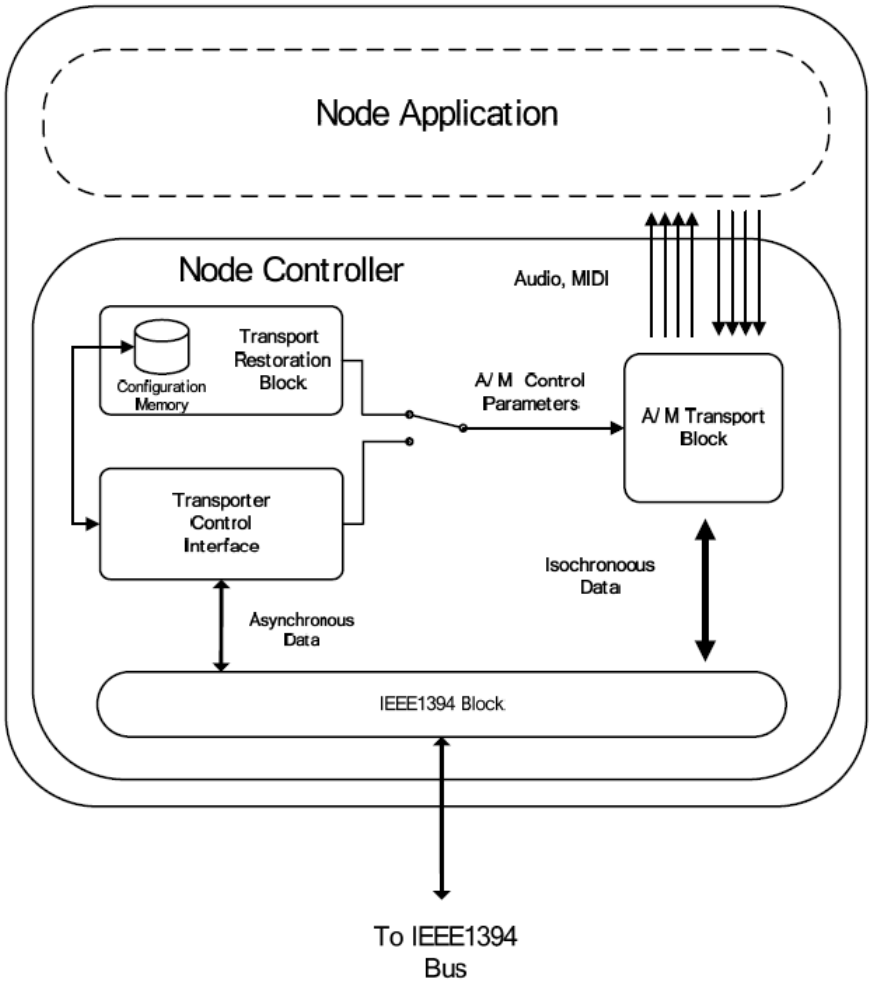


Figure 3.16: Generic Transporter Overview [Audio Engineering Society - Standards Committee, 2005]

Two sub-blocks exist within the Generic Transporter Block, namely the node application and the node controller blocks. The node application represents the hosting device and its range of legacy audio and MIDI plugs. The node controller represents the IEEE 1394 node that is hosted by the device. The Enabler sends control commands to the Transporter Control Interface via a number of generic registers. These commands are then forwarded to the A/M Transport block to perform actual encapsulation and extraction in terms of the underlying hardware/software supplied by the Transporter's vendor. The Transport Restoration block allows the Transporter to

resume transmissions in the absence of an Enabler.

3.3.3.2 Generic Architecture of the A/M Transport Block

Figure 3.17 shows the generic architecture for the A/M Transport block. In the diagram, the

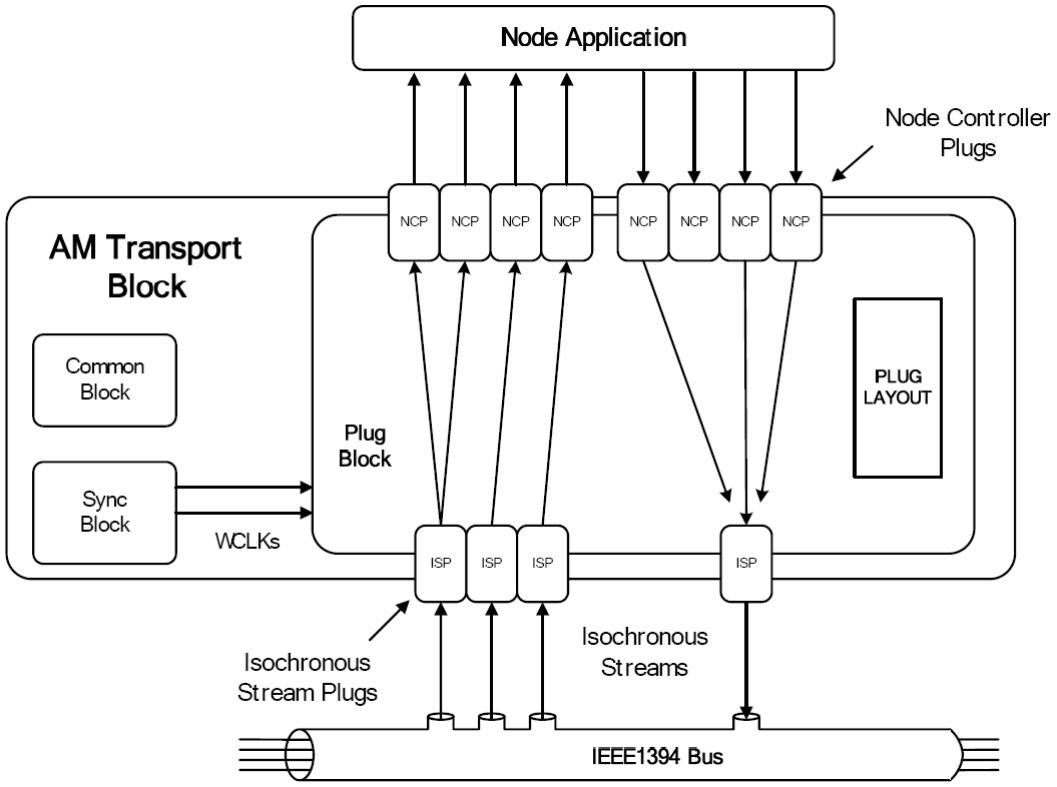


Figure 3.17: A/M Transport Block [Audio Engineering Society - Standards Committee, 2005]

Sync block handles the synchronisation functionality, while the Plug block handles connection management functionality. A detailed description of how synchronisation is handled may be found in the IEC 61883-6 specification [IEC, 2005]. An important point to be noted at this point is the introduction of the ISP-NCP model within the A/M Transport block. An Isochronous Stream Plug (ISP) represents an input or output for a single isochronous stream to or from the IEEE 1394 bus [Audio Engineering Society - Standards Committee, 2005]. On the other hand, a Node Controller Plug (NCP) represents an input or output for a single monaural channel of audio or a single cable of MIDI to or from the node application. NCPs are thus equivalent to the *terminators* (sequence end-points) described in section 3.3.2.1 under the "mLAN Plug Abstraction" heading. One or more NCPs may be dynamically or statically associated with

each ISP. This results in the static or dynamic clustering of multiple sequence end-points to an isochronous stream.

3.4 Chapter Summary

This chapter has outlined how real-time data transmission, with guaranteed QoS, is handled on an IEEE 1394 high performance serial bus. IEEE 1394 is based on the ISO/IEC 13213 (ANSI/IEEE 1212) specification. IEEE 1394 uses some of, and also extends, the features defined in this specification. Isochronous and asynchronous transfers are the two main forms of data transfer on an IEEE 1394 serial bus. Isochronous transfers are packaged in isochronous packets for the transmission of real-time audio and video data. Asynchronous transfers are packaged in asynchronous packets for the transmission of periodic data that requires guaranteed delivery. We have also described the roles of the four IEEE 1394 protocol layers, namely bus management, transaction, physical, and link layers in transmission of isochronous and asynchronous packets.

Encapsulation and extraction of audio and MIDI data in mLAN networks is done in accordance with the IEC 61883-6 specification. This specification is commonly known as the Audio and Music Data Transmission Protocol. The nature of isochronous packets, audio and MIDI data transmission formats, as well as the transmission of timing information, have been described.

Two generations of mLAN have been described. The second generation of mLAN, on which our investigation was based, has been described in detail. This second generation of mLAN is based on the Plural Node Architecture, also known as the Enabler/Transporter Architecture, which splits connection management between two nodes types, namely an Enabler (residing in a workstation) and a Transporter (residing in an IEEE 1394 audio device). The Enabler interacts with Transporters using a plug-in mechanism. A Transporter Hardware Abstraction Layer (HAL) Application Programming Interface (API) has been defined to access Transporter capabilities of different Transporter implementations in a uniform manner. In the context of the Plural Node Architecture, the concept of an Open Generic Transporter (OGT) has been introduced.

The next chapter, Chapter 4, looks more closely at the two main areas of our investigation, namely the plug-in loading mechanism and the Transporter HAL API that governs the Enabler and Transporter interaction. The chapter aims to highlight shortcomings of the existing plug-in mechanism and Transporter HAL API.

Chapter 4

Current HAL Design and Implementation

Chapter 3 has given a detailed description of mLAN technology and its underlying standards. There are two generations of mLAN, namely the first and second generations. Our investigation focused on the second generation of mLAN, where connection management of IEEE 1394 audio devices is split between two node types, namely an Enabler (residing in a workstation) and a Transporter (residing in an IEEE 1394 audio device). The Enabler has a Hardware Abstraction Layer (HAL) that makes it possible to achieve compatibility with different Transporter implementations. The HAL is implemented using a plug-in mechanism. A Transporter HAL Application Programming Interface (API) exists. This Transporter HAL API exposes capabilities of Transporters to the Enabler in a uniform manner. Manufacturers provide device-specific implementations of the Transporter HAL API for their hardware via device-specific plug-in implementations. The Enabler loads these device-specific plug-ins before interacting with devices. The concept of the Open Generic Transporter was introduced to ease the task of manufacturers by providing an open-standards-based implementation of a Transporter, which eliminates the need for device-specific plug-ins from each manufacturer.

This chapter describes evaluations that were done for two main components of the current HAL design and implementation, namely the plug-in mechanism and the Transporter HAL API. The introduction of the OGT concept has revealed inadequacies within the current Transporter HAL API. Shortcomings that were identified from our evaluations are highlighted in this chapter.

4.1 Current Redesigned Enabler Design Overview

Figure 4.1 shows an object model of the Basic Enabler. The diagram models an IEEE 1394 bus

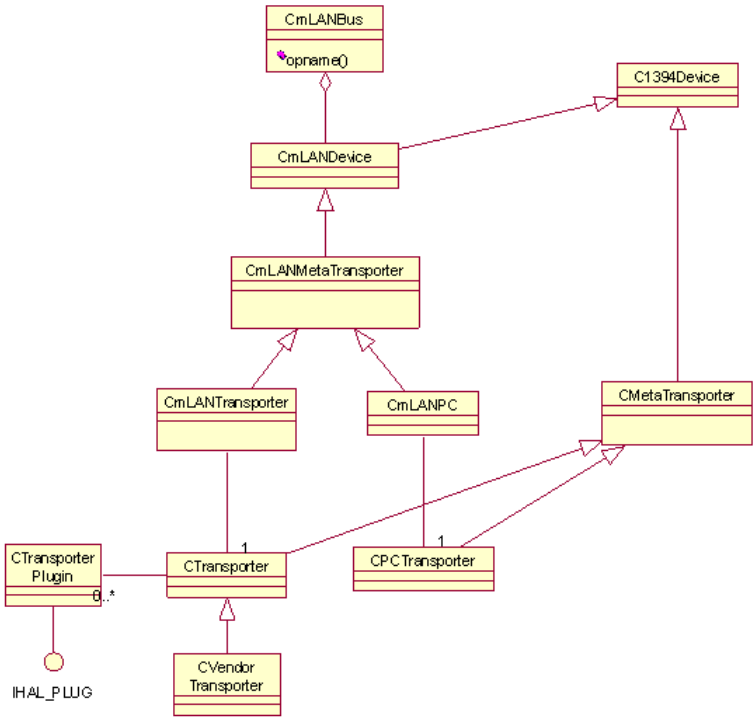


Figure 4.1: Basic Enabler Object Model [Yamaha Corporation, 2004b]

as the *CmLANBus* class. The bus has a number of mLAN devices (*CmLANDevice* class) which host Transporter modules (*CmLANMetaTransporter* class). Transporters are a further classified as either standalone device Transporters or PC Transporters[1]. Standalone device Transporters are modelled by the *CmLANTransporter* class while PC Transporters are modelled by the *CmLANPC* class. In the object model, there are six main components that represent the Hardware Abstraction Layer (HAL), namely *CTransporterPlugin, IHAL_PLUG, CVendorTransporter, CTransporter, CPCTransporter*, and *CMetaTransporter*. The *C1394Device* class defines functionality required to handle asynchronous transactions such as read, write and lock operations on the IEEE 1394 bus. Recall, from section 3.3.2.1 on page 44, the use of the term Basic Enabler to refer to the Yamaha Corporation's original Enabler implementation and Redesigned Enabler to refer to the one created by Okai-Tettey [2005]. The Basic Enabler was the starting point for other Enabler implementations under Windows, Linux, and Macintosh platforms [Yamaha Corporation, 2004b]. A detailed discussion of the Basic Enabler is beyond the scope of this thesis. However, reference is made, for comparison purposes, to its plug-in mechanism which was adopted by the Redesigned Enabler with some modification.

[1]PC Transporters are implemented on a PC workstation as opposed to a standalone device.

The model above was redesigned to produce the Redesigned Enabler. Figure 4.2 shows the object model for the Redesigned Enabler. This model more closely describes the components of

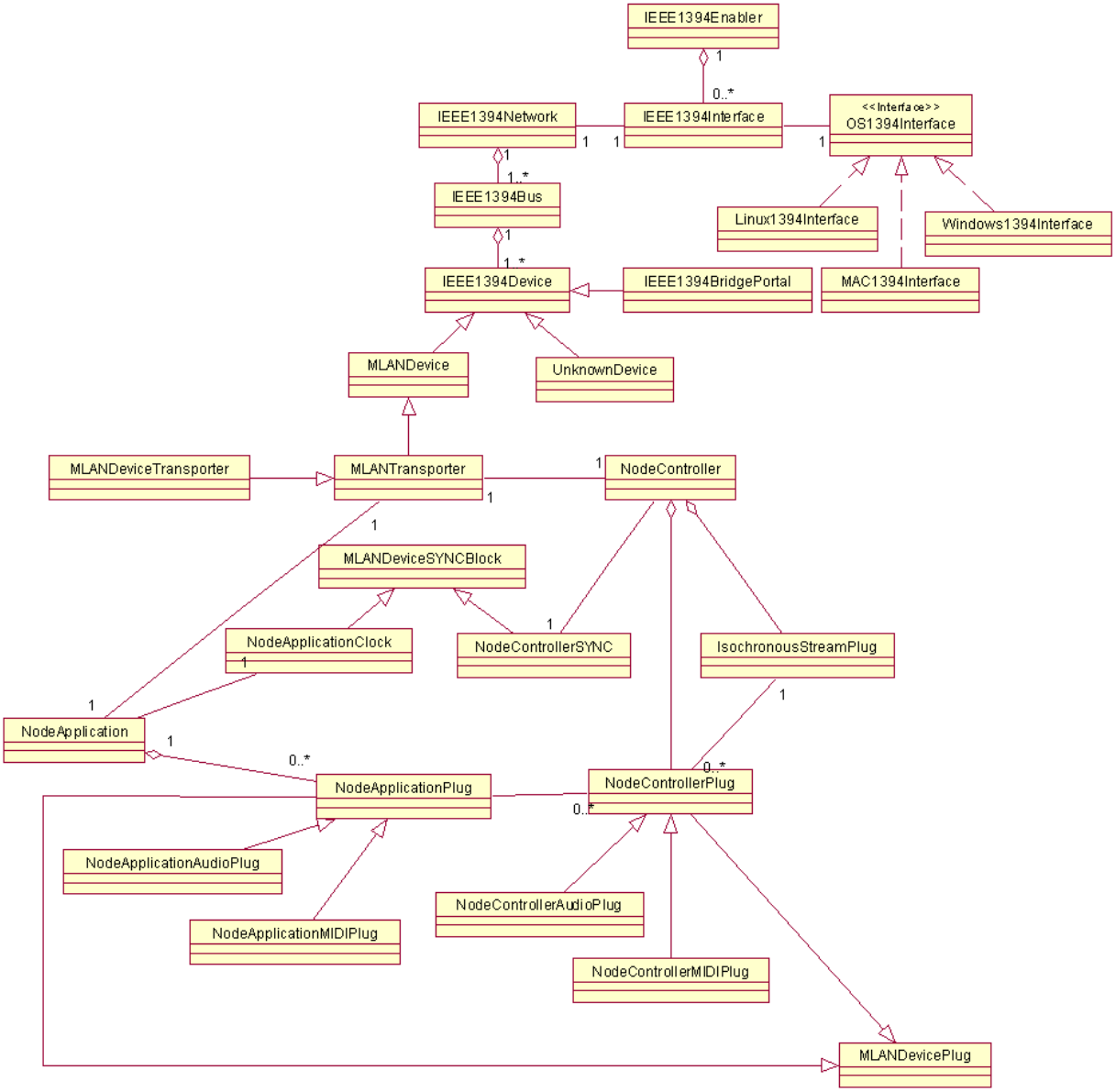


Figure 4.2: Redesigned Enabler Object Model [Okai-Tettey, 2005]

an IEEE 1394 bus. Specifically, the Enabler (*IEEE1394Enabler* class) resides in a workstation that may contain one or more IEEE 1394 interface cards (*IEEE1394Interface* class). Each interface card is connected to an IEEE 1394 network (*IEEE1394Network* class) which comprises a number of buses (*IEEE1394Bus* class). Each bus contains a number of IEEE 1394 devices

(*IEEE1394Device* class). IEEE 1394 devices may be bridge portals (*IEEE1394BridgePortal* class), mLAN devices (*MLANDevice* class), or any device that implements the IEEE 1394 protocol (*UnknownDevice* class). Bridge portals implement the 1394 bridge specification, "IEEE Standard for High Performance Serial Bus Bridges" [IEEE, 2005]. A discussion of bridge portals is beyond the scope of this thesis. Okai-Tettey [2005] discusses bridge portals in more detail. We are interested in mLAN devices. Each mLAN device hosts a Transporter module (*MLANTransporter* class) which contains a node application block (*NodeApplication* class) and a node controller block (*NodeController* class). The roles of these two blocks have been described in section 3.3.3.1 on page 49. We explore this model further, since it formed the foundation of our investigation.

4.2 Current Redesigned Enabler HAL Design Overview

Figure 4.3 shows the object model of the HAL for the Redesigned Enabler. This HAL design

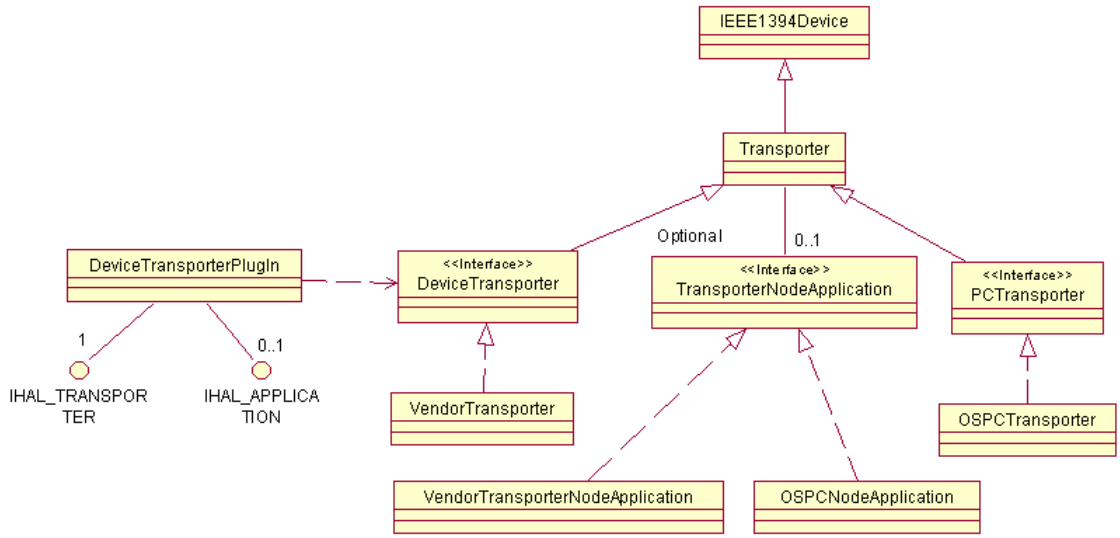


Figure 4.3: Redesigned Enabler Hardware Abstraction Layer Object Model [Okai-Tettey, 2005]

has been derived from the Basic Enabler's HAL design. In the model, two interfaces have been defined to access the node application and node controller capabilities of a Transporter, namely *IHAL_APPLICATION* and *IHAL_TRANSPORTER* respectively. The purposes of the remaining classes shown in the object model are as follows:

- The *IEEE1394Device* class defines general functionality for an IEEE 1394 device including asynchronous transaction functionality such as read, write, and lock operations.
- The *Transporter* abstract class defines methods that are necessary to control IEC 61883-6 transmission and reception for any Transporter. This functionality is used by both standalone device Transporters and PC Transporters.
- The *DeviceTransporter* abstract class defines node controller methods that are specific to standalone device Transporters, although in a hardware independent manner.
- The *PCTransporter* abstract class defines node controller methods that are specific to PC Transporters, although in a platform independent manner.
- The *VendorTransporter* class provides a device-specific implementation of the node controller component of a standalone device Transporter.
- The *OSPCTransporter* class provides a platform-specific implementation for the node controller component of a PC Transporter.
- The *TransporterNodeApplication* abstract class defines methods to control the node application component of a Transporter. These methods are implemented by the *VendorTransporterNodeApplication* and *OSPCNodeApplication* classes for device-specific Transporters and platform-specific PC Transporters, respectively.

We refer to the collection of methods that are defined to access the capabilities of Transporters as the Transporter HAL API. The Transporter HAL API comprises methods defined by the following abstract classes: *Transporter*, *DeviceTransporter*, *TransporterNodeApplication*, and *PCTransporter*, as well as the methods that are defined by the *IHAL_TRANSPORTER* and *IHAL_APPLICATION* interfaces, as shown in Figure 4.3.

Table 4.1 shows equivalent components of the HALs for both the Basic and Redesigned Enablers. Some features of the Redesigned Enabler are not available in the Basic Enabler and are marked *N/A* in the table. A closer look at the table reveals that the only components that differ between the Basic and Redesigned Enabler HALs are the node application specific components, which the latter implements. Recall, from section 3.3.3.1 on page 49, that the node application represents the hosting device and its range of legacy audio and MIDI plugs, as well as word clock sources. In addition, there is no equivalent *OSPCTransporter* class in the Basic Enabler, due to design refinements made in the Redesigned Enabler.

Redesigned Enabler HAL Component	Equivalent Basic Enabler HAL Component
DeviceTransporterPlugin	CTransporterPlugin
IHAL_TRANSPORTER	IHAL_PLUG
IHAL_APPLICATION	N/A
TransporterNodeApplication	N/A
VendorTransporterNodeApplication	N/A
OSPCNodeApplication	N/A
DeviceTransporter	CTransporter
VendorTransporter	CVendorTransporter
PCTransporter	CPCTransporter
OSPCTransporter	*N/A (Implemented within CPCTransporter)*
Transporter	CMetaTransporter
IEEE1394Device	C1394Device

Table 4.1: Comparison of Redesigned and Basic Enabler HALs

With reference to Figure 4.3, we now describe the roles of each of the components of the HAL. The *Transporter* abstract class encapsulates the functionality of a typical Transporter device [Okai-Tettey, 2005]. In particular, it declares methods to access and modify the A/M Protocol Layer (described in section 3.3.2.2 on page 46) of a Transporter. These include methods to set the transmission or reception isochronous channel number, sequence number, subsequence number, isochronous transmission speed, sample rate, and event type of the isochronous stream data. In addition, there are methods defined to modify the SYT synchronisation channel number.

Section 2.1.1.5 on page 18 introduced the concept of drivers that allow a PC to act as an mLAN device. Consequently, this leads to two types of Transporters which are modelled by the classes *DeviceTransporter* and *PCTransporter*. Device Transporters are implemented as standalone devices, while PC Transporters are implemented on a PC workstation with an IEC 61883-6 capable driver. These classes define extra methods that are more specific than the common Transporter methods. The *DeviceTransporter* and *PCTransporter* classes define standard methods that are required to be implemented by the hardware abstraction layers of the various vendor or PC Transporters, namely *VendorTransporter* and *OSPCTransporter* class implementations. For example, device Transporters that are based on different hardware architectures have different hardware-specific implementations for the *VendorTransporter* class. Similarly, there are different *OSPCTransporter* class implementations for different operating systems such as Macintosh, Windows, and Linux.

The Transporter HAL API also defines an optional node application component. This functionality is encapsulated in the *TransporterNodeApplication* abstract class. Methods of this base class

allow direct access to the capabilities and resources implemented by a Transporter's host device, such as manipulating the legacy hardware plugs of a device or modifying volume for a device's plugs. These methods are intended to be implemented by vendor- or platform-specific implementations which are modelled by *VendorTransporterNodeApplication* and *OSPCNodeApplication* classes respectively.

Lastly, the *DeviceTransporterPlugin* class represents the Enabler's Transporter HAL plug-in class. A mandatory *query interface* mechanism is implemented by this class to determine the presence of the interfaces *IHAL_APPLICATION* and *IHAL_TRANSPORTER*. *IHAL_TRANSPORTER*, named *IHAL_PLUG* in the Basic Enabler, is a mandatory interface that acknowledges the implementation of the node controller component HAL API within a HAL plug-in. This node controller component provides access to the A/M Protocol Layer of a Transporter. On the other hand, the presence of the *IHAL_APPLICATION* interface acknowledges the implementation of the node application component HAL API within a HAL plug-in. This node application component provides access to the host implementation of a Transporter device. When vendors create Transporter HAL plug-ins for the Redesigned Enabler, they are required to provide implementations for *IHAL_TRANSPORTER* methods and others, optionally, for *IHAL_APPLICATION*, as part of the implementation for the *DeviceTransporterPlugin* class. The methods defined by the *IHAL_TRANSPORTER* interface are:

- CreateTransporter
- DisposeTransporter
- GetHALVendorID
- GetHALModelID

The methods defined by the *IHAL_APPLICATION* interface are:

- CreateNodeApplication
- DisposeNodeApplication

The next section describes the current plug-in loading mechanism that is recommended by Yamaha Corporation and has been adopted for the Redesigned Enabler with some modification.

4.3 Current Transporter Plug-In Mechanism

Yamaha Corporation has defined a plug-in mechanism that can be used by software developers wishing to create plug-ins for mLAN devices [Yamaha Corporation, 2004b]. It is important to understand this mechanism in order to successfully create plug-ins that are compliant with a corresponding Enabler. There may be a variety of Enabler implementations and each may define its own plug-in loading mechanism. This implies that plug-ins written for a particular Enabler implementation are not necessarily loadable using another Enabler implementation.

Figure 4.4 shows a sequence diagram demonstrating part of the process of device enumeration within the Redesigned Enabler. This process is initiated when a new device is identi-

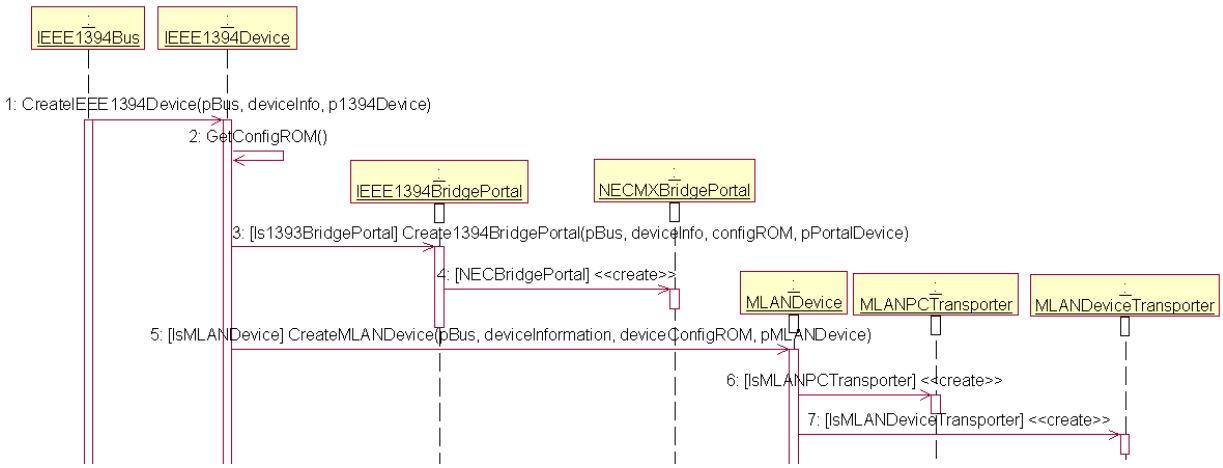


Figure 4.4: Redesigned Enabler Device Enumeration Sequence Diagram

fied on a given bus. This results in the series of events shown in the figure. In particular, a static *CreateIEEE1394Device* member function of the *IEEE1394Device* class is called by the *IEEE1394Bus* instance. This *CreateIEEE1394Device* static member function then gets the config ROM information from the device. From the config ROM information, it determines whether the device is a bridge portal or an mLAN device. Bridge portals are used in a multi-bus environment [Okai-Tettey, 2005]. A discussion of bridging is beyond the scope of this thesis. For our purposes, we are interested in mLAN Devices within a single-bus environment. When the *CreateIEEE1394Device* member function ascertains that the identified device is an mLAN device, a static *CreateMLANDevice* member function of the *MLANDevice* class is called. This *CreateMLANDevice* static member function determines if the device is a PC Transporter or a device Transporter. If the mLAN device is a device Transporter, a new instance of the *MLANDeviceTransporter* class is created, otherwise a new instance of the *MLANPCTransporter* class

is created.

At the time of this writing, the PC Transporter was not implemented in terms of a plug-in since it resides in the same PC that runs the Enabler. We explore the creation of an *MLANDevice-Transporter* instance further. Figure 4.5 shows a sequence diagram for the creation of an *MLAN-DeviceTransporter* instance. This follows the plug-in loading mechanism suggested by Yamaha

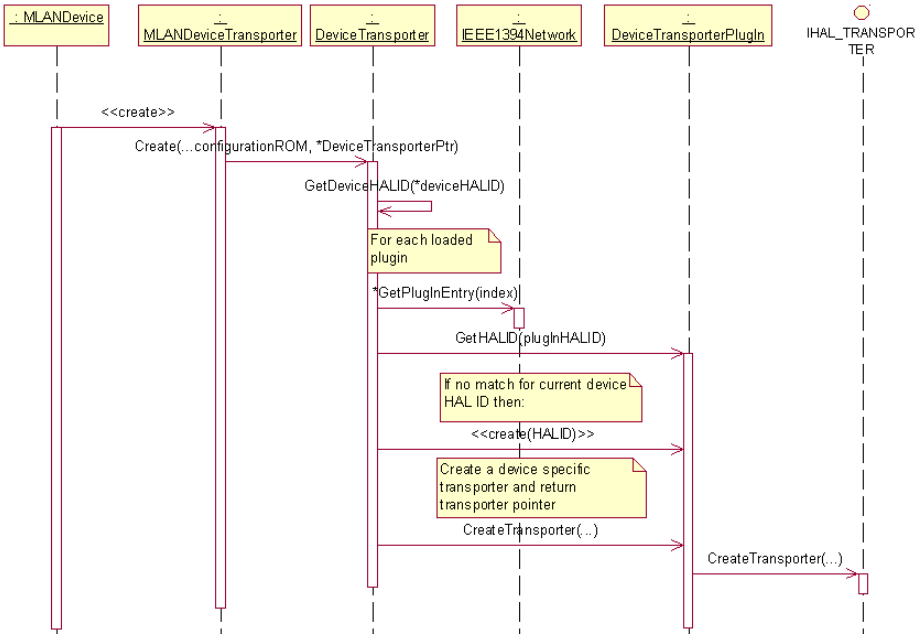


Figure 4.5: *CreateMLANDeviceTransporter* Sequence Diagram

Corporation. In particular, a static *Create* member function of the *DeviceTransporter* class is called. In this member function, the HAL ID of the device is retrieved and a search is done to check if a plug-in with a matching HAL ID is already loaded. If no plug-in is already loaded, an attempt is made to load the plug-in, in a platform-specific manner, through creation of a new instance of the *DeviceTransporterPlugin* class. Bear in mind that the HAL ID comprises the HAL Vendor ID and the HAL Model ID, which are defined within the config ROM of all mLAN devices [Yamaha Corporation, 2003b].

Once the plug-in is loaded, a device-specific *Transporter object* is created via the plug-in's *CreateTransporter* member function. Transporter creation within a plug-in is fulfilled in terms of a call to the *IHAL_TRANSPORTER* interface's *CreateTransporter* function, where the implementation is plug-in-specific as mentioned in section 4.2. A pointer to the device-specific *Transporter object* is passed back to the plug-in instance, which also passes the pointer to the *DeviceTransporter* class, and eventually back to the *MLANDeviceTransporter* object. The

MLANDeviceTransporter object makes use of this pointer, for the duration of its existence, to directly access the vendor-specific implementation of the *DeviceTransporter* class, which includes the vendor-specific A/M Protocol Layer implementation defined by the *Transporter* class. When the device is removed from the bus, the device-specific *Transporter object* is then deleted via the *DeviceTransporterPlugin* class' *DisposeTransporter* member function. Similar to the *CreateTransporter* member function, this member function is fulfilled in terms of a call to the *IHAL_TRANSPORTER* interface's *DisposeTransporter* method where the implementation is plug-in specific.

In the event of updates to the Transporter HAL API, the Enabler's direct use of a device-specific *Transporter object* pointer to access the vendor-specific capabilities of Transporters is error prone. An example of one such update is the addition of new methods to the *DeviceTransporter* class. Bear in mind that methods of the *DeviceTransporter* class are part of the Transporter HAL API that is used by the Enabler to access the vendor-specific capabilities of Transporters, as described in the introduction to section 4.2. In such a cases, direct use of the *Transporter object* pointer works well if the Enabler and the Transporter plug-in share exactly the same *function prototypes* for the *DeviceTransporter* class, and hence the Transporter HAL API. However, if updates are only made to the Enabler's *DeviceTransporter* class without making similar updates to the *DeviceTransporter* class for all Transporter plug-ins, problems may occur when the Enabler loads plug-ins that are based on earlier versions of the *DeviceTransporter* class. These problems, most likely runtime exceptions, arise when the Enabler attempts to access new methods of the *DeviceTransporter* class, while the plug-ins do not provide the required implementation. This implies that, at the Transporter HAL API level, there is some form of binary dependence between the Enabler and its Transporter plug-ins. Such binary dependence complicates the possibility of maintaining different versions of the Transporter HAL API.

The above description of the Redesigned Enabler plug-in mechanism is also true for the Basic Enabler with regards to creation and disposal of the device-specific *Transporter object*. However, the Basic Enabler's *IHAL_PLUG* interface, the equivalent of *IHAL_TRANSPORTER* interface in the Redesigned Enabler, defines an additional *CallTransporter* function. Rather than making use of the device-specific *Transporter object* pointer to directly access the capabilities of Transporters, all Transporter HAL API methods are accessed via this intermediary *CallTransporter* function.

Figure 4.6 shows an example of an object model for a generic plug-in for the Basic Enabler. Based on this model, the sequence diagram shown in Figure 4.7 shows an example of how a *CTransporter* class method, namely *GetSYTSyncChannel*, is accessed via the intermediary *Call-*

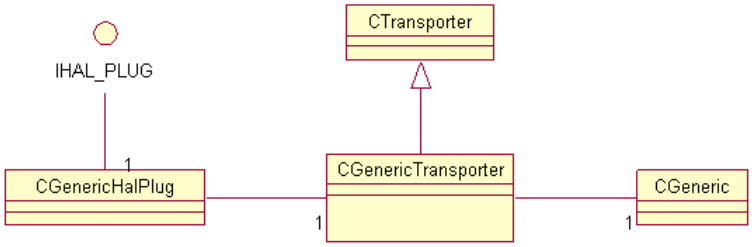


Figure 4.6: Generic Plug-In Object Model for the Basic Enabler [Yamaha Corporation, 2004b]

Transporter function of the *IHAL_PLUG* interface. As shown in Table 4.1, the *CTransporter*

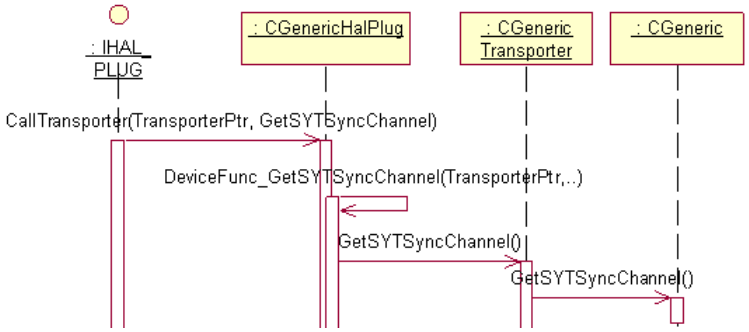


Figure 4.7: *CallTransporter* Sequence Diagram for the Basic Enabler [Yamaha Corporation, 2004b]

class of the Basic Enabler HAL is equivalent to the *DeviceTransporter* class of the Redesigned Enabler HAL. The sequence diagram in Figure 4.7 shows that the *CallTransporter* function takes two arguments, namely a device-specific *Transporter object* pointer (*TransporterPtr*) and a selector ID (*GetSYTSyncChannel*). The selector ID is a unique numerical identifier that corresponds to a particular method of the Transporter HAL API. The *CallTransporter* function of the *CGenericHALPlug* class resolves the selector ID and chooses the right method to call, namely *DeviceFunc_GetSYTSyncChannel*. The *DeviceFunc_GetSYTSyncChannel* method then makes use of the *Transporter object* pointer to call its corresponding *GetSYTSyncChannel* method. An important advantage of the intermediary *CallTransporter* function is that changes in the Transporter HAL API are detected when the selector ID is resolved. If the ID cannot be resolved by a plug-in, the most likely reason is that the method is not supported by the version of the Transporter HAL API in use and a meaningful error return value is sent back to the Enabler before any runtime exceptions occur. This advantage, however, comes at the price of additional overhead incurred by an additional *CallTransporter* function call before accessing any Transporter HAL API method. In addition, each new method of the Transporter HAL API requires a new selector

ID to be defined. This selector ID is shared by the Enabler and all Transporter plug-ins, since it acts as a unique identifier for each of the Transporter HAL API methods. When creating Transporter plug-ins, care must be taken to ensure that these selector IDs match the methods that they represent. Consequently, the use of selector IDs to enable Transporter HAL API versioning via an intermediary *CallTransporter* function call becomes inherently error prone.

In the above overview of the HAL plug-in mechanism there has been an indication that plug-ins are loaded in a platform-specific manner. We now discuss how plug-in loading is handled on three popular workstation platforms, namely Windows, Linux, and Macintosh. These platform-specific plug-in loading techniques flesh out the sequence of events that occur upon creation of the *DeviceTransporterPlugin* object, as shown in the sequence diagram in Figure 4.5.

4.3.1 Windows Transporter Plug-In Loading Mechanism

As mentioned in section 3.3.2.1 on page 44 under the "Hardware Abstraction Layer" heading, plug-in loading on Windows makes use of Microsoft's COM mechanism. This approach to Transporter plug-in loading for the Enabler is described in more detail in the "mLAN Transporter Plug-In Mechanism" [Yamaha Corporation, 2004b] document. We now describe how the Redesigned Enabler makes use of this COM mechanism. Figure 4.8 shows a sequence diagram which describes the creation of a *DeviceTransporterPlugin* object on Windows. We focus on the constructor of the *DeviceTransporterPlugin* class, since it loads a plug-in with a given HAL ID[2].

The starting point of the constructor is the loading of COM libraries by calling the *CoInitialize* function [Yamaha Corporation, 2004b]. This is followed by an iterative search of the Windows registry for a Transporter plug-in (known as a component in Windows) with a matching HAL ID. The registry acts like a database where applications and components store information about themselves, such as their location on the hard drive. The registry comprises entries, known as keys, that are arranged hierarchically. Values may be associated with these keys. All Transporter plug-in components would have previously loaded their information into the registry. An example of keys leading to a Transporter plug-in key is:

```
HKEY_LOCAL_MACHINE\SOFTWARE\YAMAHA\HAL_DEMO\HAL_NCP0405
```

The key above is just one out of a number of Transporter plug-in (also known as HAL plug-in) keys for various Transporters that the Redesigned Enabler interacts with, as shown in the snapshot in Figure 4.9.

[2]HAL ID is a combination of HAL Vendor ID and HAL Model ID.

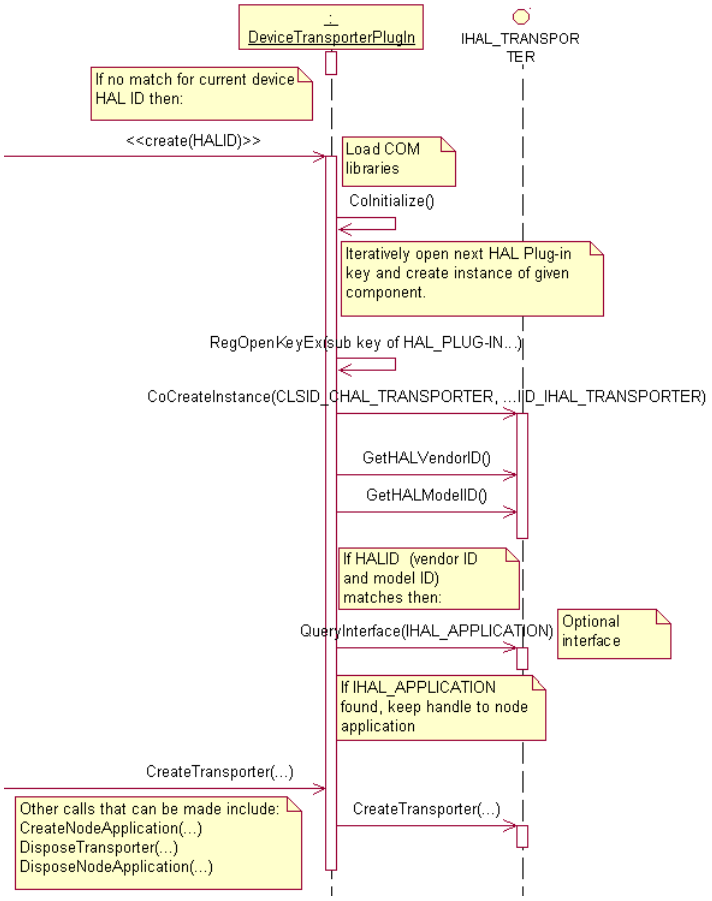


Figure 4.8: Plug-In Loading Mechanism for Windows

When searching the registry, a handle to the *HAL_DEMO* registry key is obtained. For each HAL plug-in that exists under the *HAL_DEMO* key, as shown in Figure 4.9, an instance of the component's *IHAL_TRANSPORTER* interface is created using a *CoCreateInstance* COM function. This COM function requires information about the plug-in such as the class identifier (CLSID[3]) and interface identifier (IID[4]), shown as *CLSID_CHAL_TRANSPORTER* and *IID_IHAL_TRANSPORTER* respectively, in Figure 4.8. This information, passed as function arguments, is stored in the individual HAL plug-in keys such as the *HAL_NCP0405* key (the selected key in Figure 4.9). The *CoCreateInstance* function creates a single uninitialised object of the class associated with a specified CLSID [Microsoft Developer Network, 2007a]. Upon successful creation of the component's instance, the HAL ID is queried using the *GetHALVendorID* and *GetHALModelID* functions of the *IHAL_TRANSPORTER* interface. In some implementa-

[3]Uniquely identifies a COM component.
[4]Uniquely identifies an interface that may be implemented by a COM component.

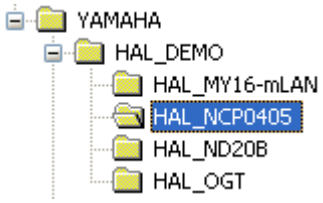


Figure 4.9: Snapshot of HAL Plug-In Registry Contents

tions, *GetHALVendorID* and *GetHALModelID* functions are grouped into one function that is named *GetHALDescription*. If the HAL ID of the plug-in matches that of the device's config ROM, one can be certain that the plug-in has been successfully loaded and the newly created *DeviceTransporterPlugin* object is stored in an array of *DeviceTransporterPlugin* objects. The existence of the optional *IHAL_APPLICATION* interface is queried and, if available, a pointer to the interface is also stored. Subsequent calls to the *DeviceTransporterPlugin* object are then fulfilled via functions defined in either the *IHAL_TRANSPORTER* or *IHAL_APPLICATION* interfaces of the plug-in. These include the *CreateTransporter* and *DisposeTransporter* functions of the *IHAL_TRANSPORTER* interface, and, if the present, the *CreateNodeApplication* and *DisposeNodeApplication* functions of the *IHAL_APPLICATION* interface.

We have described the plug-in loading mechanism on the Windows platform in detail above. Figure 4.10 shows a high level conceptual diagram that summarises this plug-in loading mechanism. In particular, before a plug-in is loaded, a HAL ID is retrieved from a device's config ROM in order to identify the plug-in to load. COM libraries are then loaded and an iterative search through all plug-in keys in the registry is done. For each plug-in key, an instance of the HAL plug-in is created and the HAL ID is compared, until the one with a matching HAL ID is found. Once a matching HAL ID is found, the plug-in is loaded and the node application implementation is queried. Thereafter, HAL plug-in functions such as *CreateTransporter* can be accessed.

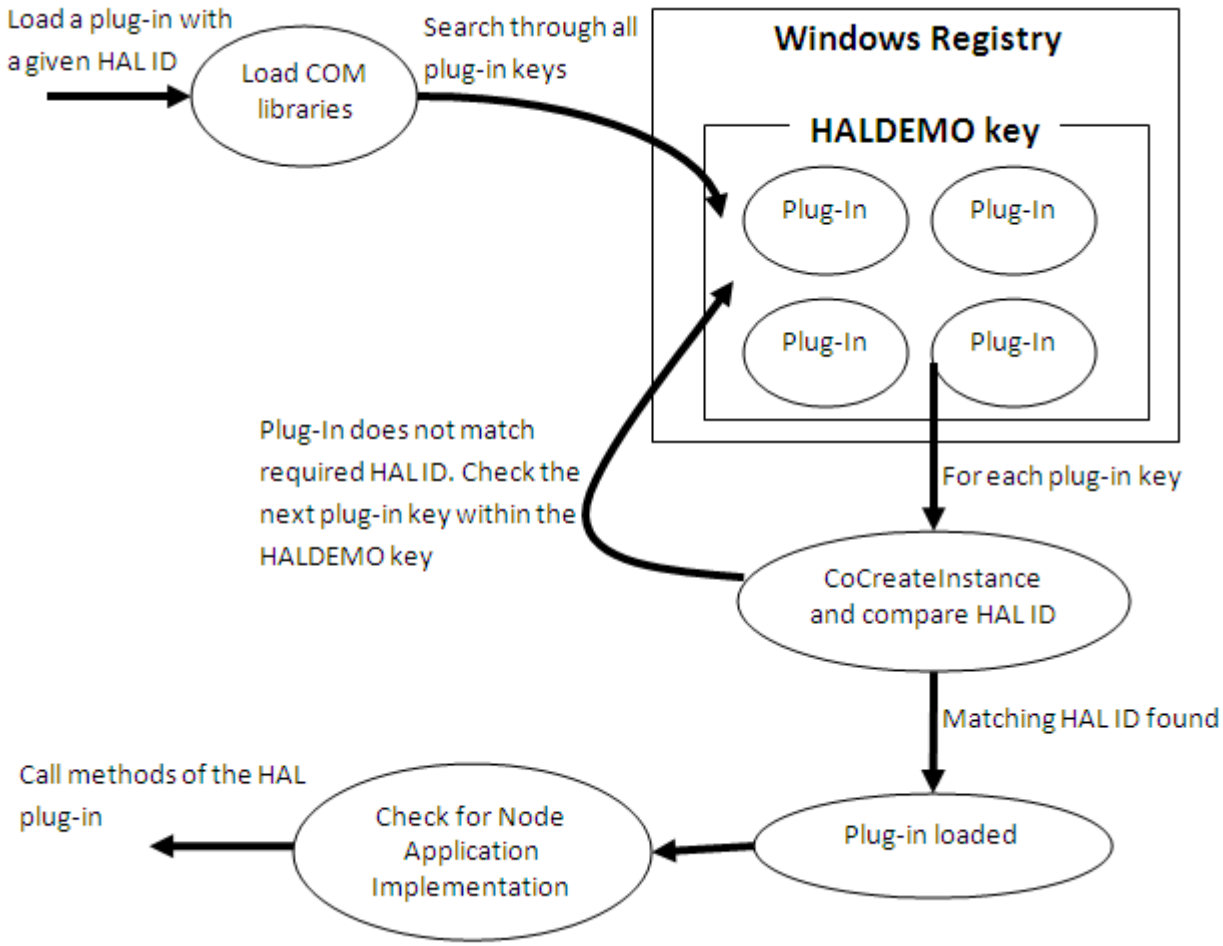


Figure 4.10: Conceptual Diagram for the Plug-In Loading Mechanism on Windows

4.3.2 Linux Transporter Plug-In Loading Mechanism

Within Linux, there is neither a concept of a registry nor COM, hence the *DeviceTransporterPlugin* class implementation takes a different form from that of Windows. At the time of this writing, the "mLAN Transporter Plug-In Mechanism" [Yamaha Corporation, 2004b] document did not specify mechanisms for plug-in loading on platforms other than Windows. However, source code was available for the Linux implementation of the Redesigned Enabler. We describe the plug-in loading mechanism on the Linux platform based on our understanding of the source code.

As highlighted by Okai-Tettey [2005], plug-in loading on Linux follows a shared library approach. Shared libraries (also known as shared objects or dynamically linked libraries) are groupings of object files [Mitchell, Oldham, and Samuel, 2001]. All objects that compose a

shared library are combined into a single file. This single file uses the *.so* (standing for shared object) file extension, and the file name always begins with *lib*. Code within shared libraries may be loaded and unloaded dynamically during runtime, which is what the Enabler requires of Transporter plug-ins.

In the context of the Enabler/Transporter Architecture, all Linux-based Transporter plug-in implementations are in the form of shared libraries that expose the functions *GetHALDescription, CreateTransporter*, *DisposeTransporter*, *CreateNodeApplication*, and *DisposeNodeApplication* [Okai-Tettey, 2005]. Each of these functions is represented by a pointer within the Enabler's *DeviceTransporterPlugin* class. These functions are exposed without any function name mangling. Function name mangling is a form of compiler encoding that adds additional information about the function to its name and results in a "funny-looking" name [Mitchell et al., 2001]. C compilers do not mangle names. However, C++ functions require that mangling be suppressed by defining Transporter plug-in functions with the *extern "C"* declaration such as:

```
extern "C" void GetHALDescription(...)
```

All Transporter plug-ins for the Linux implementation of the Redesigned Enabler are stored in a common location on the hard drive, namely */usr/local/lib/HALs*. As was done for Windows, the constructor of the Linux Enabler's *DeviceTransporterPlugin* class is explored in more detail in order to analyse the plug-in loading mechanism on Linux. Figure 4.11 shows a sequence diagram which gives an outline of the implementation of this shared library approach.

The starting point in the constructor of the *DeviceTransporterPlugin* class under Linux is locating the shared libraries of all Transporter plug-ins. This is done by calling a function of the form:

```
system("find /usr/local/lib/HALs/lib*.so.?
-fprint PlugInFileDump");
```

This function results in the creation of a text file named *PlugInFileDump* which contains the full path names, each on a separate line, of each unique Transporter plug-in shared library within the location */usr/local/lib/HALs*. An example of one such path name is */usr/local/lib/HALs/libMY16-_mLAN.so.1*, for the *MY16_mLAN*[5] Transporter plug-in shared library.

An iterative search for the plug-in with a matching HAL ID is done. Using each of the path names in the *PlugInFileDump* text file, each of the corresponding shared libraries is dynamically loaded by using the *dlopen* function. The *dlopen* function returns a handle/pointer to the loaded shared library. This pointer to the shared library, in conjunction with the textual name of a function (such as "GetHALDescription"), can be passed to the *dlsym* function in order to

[5] A name referring to Yamaha's 01V Mixing desk Transporter.

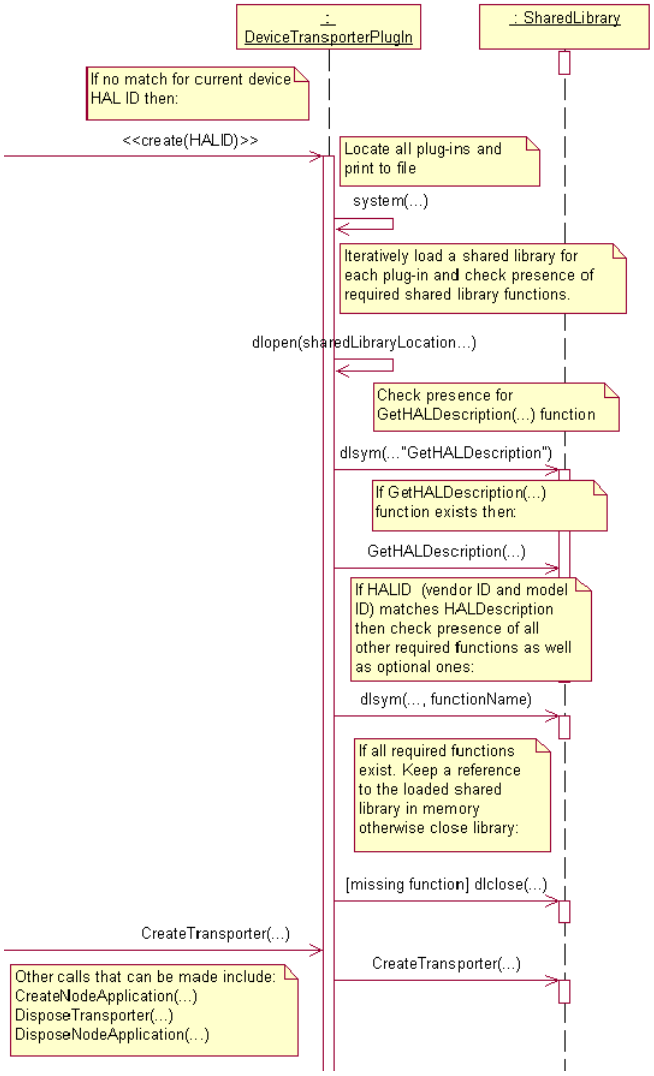


Figure 4.11: Plug-In Loading Mechanism for Linux

get the address of the named function within the loaded shared library. This is the reason why function names, as mentioned above, should be free of any name mangling. The next step, after loading the library within each iteration, is to get the address of the *GetHALDescription* function by calling the *dlsym* function. Upon getting the address of the *GetHALDescription* function, the function is called in order retrieve the HAL ID of the plug-in represented by the shared library. If the HAL ID matches that of the device's config ROM, the addresses of the mandatory functions, *CreateTransporter* and *DisposeTransporter*, are retrieved by calling the *dlsym* function appropriately. Assuming that all the mandatory functions are present, the addresses of the optional functions, *CreateNodeApplication* and *DisposeNodeApplication*, are also retrieved calling the

dlsym function again. All function addresses that are retrieved are stored in pointers defined for each function within the *DeviceTransporterPlugin* class of the Enabler. It is these pointers to function addresses that are used to fulfil subsequent function calls, such as *CreateTransporter*, that are made via a *DeviceTransporterPlugin* object. If the address of any mandatory function cannot be retrieved, or the HAL ID of the plug-in does not match the device's, the shared library is unloaded by calling the *dlclose* function.

Figure 4.12 shows a high level conceptual diagram to summarise the Linux plug-in loading mechanism process described above. In particular, the HAL ID for the plug-in to be loaded

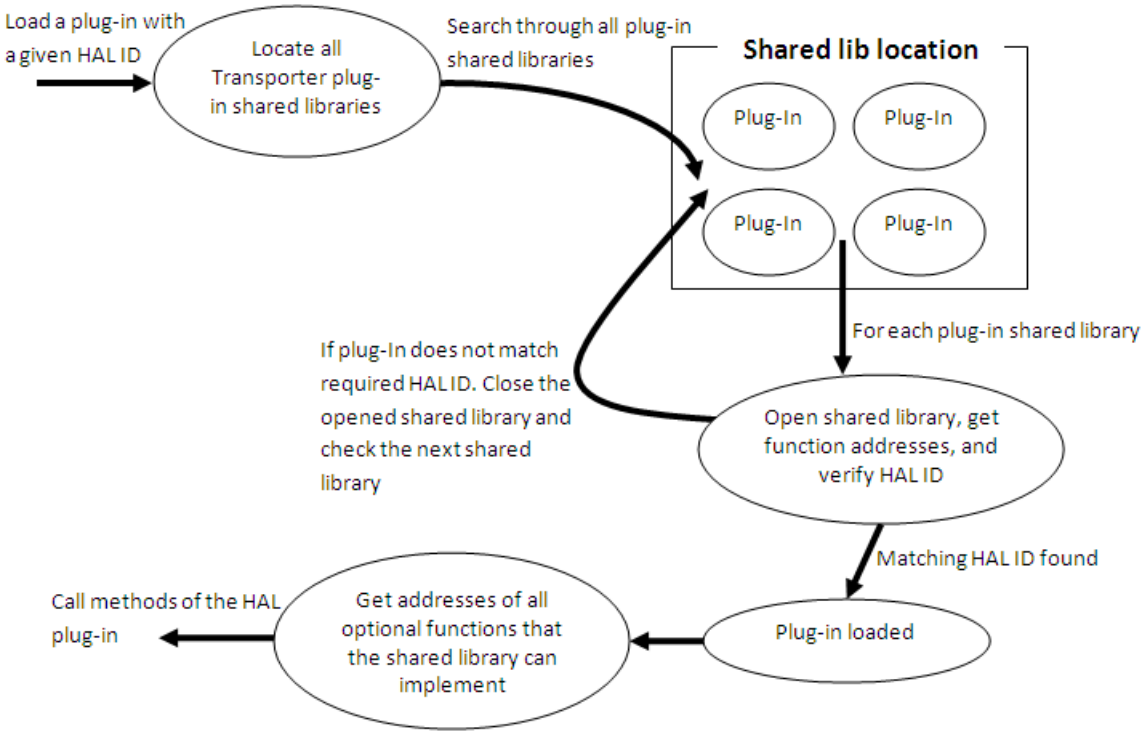


Figure 4.12: Conceptual Diagram of the Plug-In Loading Mechanism on Linux

is supplied. Shared libraries for all Transporter plug-ins on the Enabler are located, followed by a search through all the plug-ins in the shared library location. For each shared library, until the one with the matching HAL ID is found, the shared library is opened and addresses for the mandatory functions are retrieved before the HAL ID is verified. When a shared library with a matching HAL ID is found, one can be certain that the plug-in has been successfully loaded and the addresses for the optional HAL plug-in functions such as those for the node application are retrieved. Thereafter, any of the HAL plug-in functions can be accessed.

4.3.3 Macintosh Transporter Plug-In Loading Mechanism

At the time of this writing, there was no Macintosh implementation for the Redesigned Enabler that made use of the existing plug-in mechanism. However, Okai-Tettey [2005] mentions that a shared library approach similar to that of Linux could be used. Macintosh is a UNIX-based operating system which makes it possible to use the dynamic loader compatibility (DLC) functions mentioned above, namely *dlopen*, *dlsym*, and *dlclose* [Apple Computer, Inc., 2006b,d].

4.3.4 Shortcomings of the Current Transporter Plug-In Mechanism

There is a shortcoming resulting from the Redesigned Enabler's direct use of a device-specific *Transporter object* pointer to access the capabilities of a Transporter. This shortcoming relates to versioning. In particular, problems arise upon updates to the Transporter HAL API. Updating the Transporter HAL API may take the form of, for example, the introduction of additional methods to the *Transporter* or *DeviceTransporter* classes shown in Figure 4.3. In such cases, backwards compatibility problems arise when the Enabler interacts with Transporter plug-ins, which are implemented based on earlier versions of the Transporter HAL API, and do not provide implementations for all the methods that are defined in later versions of the Transporter HAL API. This implies that a certain level of binary dependence exists between an Enabler and its associated Transporter plug-ins.

In the context of the above shortcoming, it became clearer why Yamaha Corporation has an intermediary *CallTransporter* function in their *IHAL_PLUG* plug-in interface. Any request to access the capabilities of a Transporter that are exposed by the Transporter HAL API goes via the plug-in's *CallTransporter* function, where it is resolved and the corresponding function of the device-specific *Transporter object* is called. Any differences between the Transporter HAL API versions on which a plug-in and the Enabler are based are resolved within the *CallTransporter* function. Consequently, plug-ins based on earlier versions of the Transporter HAL API remain fully functional, except that functionality defined by newer versions of the Transporter HAL API is not available. The Enabler is notified of such differences if the *CallTransporter* function fails to resolve any request from the Enabler. It is then possible for an appropriate error message to be reported back to the Enabler without affecting its operation.

While the use of an intermediary *CallTransporter* function allows for Transporter HAL API versioning, it adds to the overhead incurred when making calls to any Transporter HAL API functions. Bearing in mind that many Transporter HAL API functions may be called before au-

dio/MIDI connections are made between Transporters, such an overhead is undesirable, although it is functional. Box [1998] refers to this approach as a *handle class* approach. In addition to the cost of making two function calls, first to *CallTransporter* and then to the actual function being called, Box [1998] suggests that for APIs with hundreds or thousands of methods, writing forwarding routines for each API method becomes tedious and error prone. In the case of the *CallTransporter* forwarding routine, each Transporter HAL API function is associated with a selector ID that uniquely identifies the function. When the Enabler calls a function, it provides the *CallTransporter* function with the selector ID of the function to call and a pointer to the *Transporter object* for which the function is to be called. Upon receiving such a request, the *CallTransporter* function calls the appropriate function based on its interpretation of the selector ID. Each selector ID should map to exactly one Transporter HAL API function. The process of interpreting the selector ID and mapping it to the correct function requires great care by ensuring that definitions of selector IDs within the Enabler and Transporter plug-in are exactly the same. Consequently, this makes the process of Transporter plug-in writing error prone.

It became apparent that the plug-in loading mechanism described above needed further investigation in order to come up with a plug-in mechanism solution that:

- Allows for better Transporter HAL API versioning.
- Reduces the level of binary dependence between the Enabler and Transporter plug-ins.
- Eliminates the need for an intermediary function call before accessing any method of Transporter HAL API.

Bearing in mind these primary needs, a successful solution was one that allowed for ease of portability of the plug-in loading mechanism across different platforms. As shown above in sections 4.3.1 and 4.3.2, the only differences in plug-in loading implementations are conveniently isolated within the *DeviceTransporterPlugin* class. Such an object oriented design eases portability of the Enabler's plug-in implementation, requiring minimal changes. We aimed to retain such flexibility, although with an improved underlying plug-in loading mechanism.

4.4 Current Transporter HAL API

Up to this point, we have been discussing the plug-in loading mechanism. As mentioned in the introduction to this chapter, the plug-in loading mechanism is the first of the two components

of the HAL that we investigated. The second component, the interaction between the Enabler and each Transporter, is accomplished via a pointer to a *Transporter object.* The *Transporter object* is created via the *CreateTransporter* function of the *IHAL_TRANSPORTER* interface. The interaction between the Enabler and the Transporter is carried out within the confines of the collection of methods that compose the Transporter HAL API. The current Transporter HAL API has been created to fully utilise the capabilities of Yamaha Corporation's hardware architecture and node design, although plug-ins may be written to achieve interoperability with other vendor-specific Transporter implementations. In this section, we begin by revisiting Yamaha Corporation's Transporter design, on which the current Transporter HAL API is based. We end the section by highlighting shortcomings of the current Transporter HAL API in the context of the OGT guideline document.

4.4.1 Yamaha Corporation's Transporter Design

4.4.1.1 Transporter Hardware Architecture

Section 3.3.2.2 on page 46 introduced the two main chips (ASICs) that Yamaha Corporation has produced, namely the mLAN-NC1 and mLAN-PH2 chips. The mLAN-NC1 is capable of simultaneously transmitting and receiving eight sequences of audio data, two sequences of non-audio data, and four sequences of MIDI data streams, or transmitting and receiving one sequence of non-audio data and eight sequences of MIDI data streams [Yamaha Corporation, 2001]. On the other hand, the mLAN-PH2 chip is capable of simultaneously transmitting and receiving 32 audio data sequences [Yamaha Corporation, 2003a]. Up to four mLAN-PH2 chips can be cascaded together, in addition to an mLAN-NC1 chip, giving a maximum full-duplex audio channel count of 128 at 48 kHz, and eight MIDI data streams.

The most common chip arrangement that is currently in Yamaha Corporation's products comprises one mLAN-NC1 and one mLAN-PH2 chip. An evaluation board known as the MAP4 Evaluation board [Yamaha Corporaton, 2003] implements this arrangement, although without the host specific hardware components that are commonly found on audio mixing desks and synthesisers. We used this board as a reference to an existing Yamaha-specific hardware architecture that is compatible with the current Transporter HAL API. The chip arrangement within the MAP4 Evaluation board is based on the Node Controller Package 5 (NCP05) design [Yamaha Corporation, 2002b]. To mention in passing, the NCP05 design makes use of one mLAN-NC1 and two, instead of one, mLAN-PH2 chips. However, for the remainder of this thesis, we focus

on the arrangement with a single mLAN-PH2 chip. We now discuss the most important features of the mLAN-NC1 and mLAN-PH2 chips.

Recall from section 3.1.2.2 on page 35 that the link layer controller is important for isochronous and asynchronous transfers. One of the main reasons why the mLAN-PH2 is used in conjunction with the mLAN-NC1 chip is that the mLAN-PH2 chip does not have a link layer controller [Yamaha Corporation, 2001, 2003a]. For isochronous and asynchronous transactions, the mLAN-PH2 chip relies on the link layer controller which the mLAN-NC1 chip implements. In NCP05-based designs, such as the MAP4, the mLAN-PH2 handles the transmission and reception of audio data only while the mLAN-NC1 handles the transmission and reception of MIDI data only. It should be borne in mind that the mLAN-NC1 chip can also transmit and receive audio data, as mentioned in the first paragraph of this section. However, this audio capability is not used in the presence of the mLAN-PH2 chip.

4.4.1.2 Transporter Node Design

Section 3.3.2.2 on page 46 introduced the role of the Transporter Control Interface of a Transporter, namely to allow access and modification of A/M protocol parameters in a hardware specific manner. The Transporter Control Interface is implemented in the IEEE 1394 node's *private space*. In the context of Yamaha Corporation's NCP05-based hardware, a Transporter node design exists. It is this node design that determines the nature of the Transporter Control Interface for NCP05-based hardware. We have described the 1394 node address space as shown in Figure 3.2 on page 28. Table 4.2 shows a typical serial bus address map for NCP05-based IEEE 1394 audio devices. We focus on the node's *private space* which describes the Transporter Control Interface. The absolute address of the top of the *private space* is 0xFFFFE0000000. A PRIVATE_SPACE_MAP table is implemented at the top of the *private space*. This PRIVATE_SPACE_MAP table specifies the offsets and sizes of four main register blocks of the Transporter Control Interface. These four registers blocks are Boot Parameter Space, Core Space, mLAN Space, and Node Application Space. Table 4.3 shows the PRIVATE_SPACE_MAP for NCP05-based hardware, where the register offsets are relative to the top of the *private space* (0xFFFFE0000000).

Register Address	Description
0000 0000 0000 FFFF DFFF FFFF	Initial Memory Space
FFFF E000 0000 FFFF E000 F000 FFFF E000 F7FF FFFF E100 0000 FFFF E9FF FFFF FFFF EC00 C000 FFFF EC00 C107 FFFF EE00 0000 FFFF EEFF FFFF FFFF EFFF FFFF	PRIVATE_SPACE_MAP Boot Parameter Space Core Space mLAN Space Node App Space
FFFF F000 0000 FFFF F000 03FF FFFF F000 0400 FFFF F000 07FF	Register Space (CSR) CSR ROM
FFFF F000 0800 FFFF FFFF FFFF	Initial Units Space

Table 4.2: NC1-Transporter Address Map: Serial Bus Addressing

Register Name	Offset	Length	Description
BOOT_PAR_SPACE_OFFSET	+000	32 bits	Boot parameter area offset
BOOT_PAR_SPACE_SIZE	+004	32 bits	Boot parameter area size
CORE_SPACE_OFFSET	+008	32 bits	Core area offset
CORE_SPACE_SIZE	+00C	32 bits	Core area size
MLAN_SPACE_OFFSET	+010	32 bits	mLAN area offset
MLAN_SPACE_SIZE	+014	32 bits	mLAN area size
NODE_APP_SPACE_OFFSET	+018	32 bits	Node application area offset
NODE_APP_SPACE_SIZE	+01C	32 bits	Node application area size

Table 4.3: NC1-Transporter Address Map: PRIVATE_SPACE_MAP

We now give an overview of the main roles of the four main register blocks of the Transporter Control Interface for NCP05-based hardware. The Boot Parameter Space, described in section 3.3.2.2 on page 46, is the Transporter's non-volatile memory from which a Transporter's A/M protocol parameters are restored after a power cycle. The Core Space specifies addresses that are used to modify A/M protocol parameters for the chips that are used by a Transporter, namely the mLAN-NC1 and mLAN-PH2 chips. A complete listing of the Core Space registers for the mLAN-NC1 [Yamaha Corporation, 2001] and mLAN-PH2 [Yamaha Corporation, 2003a] chips is given in the relevant specifications for the chips. The mLAN Space defines registers that are used to keep track of the isochronous resources (channel number and bandwidth) that are in use for data transmission by both chips. Table 4.4 shows a summary of the mLAN Space registers, where the offsets are relative to the top of the mLAN Space. The current Transporter HAL API is associated with the Core and mLAN Spaces. The Node Application Space provides the host system with inbound/outbound asynchronous packet bridging services. In the remainder of this subsection, we show how some of the important transmission and reception A/M parameters relate to the mLAN Space and Core Space registers.

For MIDI transmission, a transmission FIFO[6] buffer exists within the mLAN-NC1 chip. This FIFO buffer receives MIDI messages from the chip's pins and multiplexes them onto a single sequence within isochronous packets for transmission on the IEEE 1394 bus. MIDI multiplexing has been described section 3.2.1 on page 37. Prior to transmission, an isochronous channel number and bandwidth are allocated for the chip to transmit its isochronous stream (comprising a single sequence with multiplexed MIDI messages). The channel number is stored in a Core Space register associated with the mLAN-NC1 chip. Similarly, for audio transmission, a transmission FIFO buffer exists within the mLAN-PH2. This transmission FIFO receives audio samples from the chip's pin and packages the audio in a number of sequences. An isochronous stream comprising up to 32 audio sequences is transmitted onto the IEEE 1394 bus. As with the mLAN-NC1 chip, an isochronous channel number and bandwidth are allocated prior to data transmission. The channel number and the number of sequences per packet cluster are each written to a Core Space registers that are associated with the mLAN-PH2 chip.

[6]First-In-First-Out

Register Name	Offset	Length	Description
ENABLER_ADDRESS	+000	64 bits	The node ID for the Enabler and its 48-bit interrupt register address.
TRANSPORTER_MODE	+008	16 bits	Specifies whether or not a transporter should recall its previous state upon a power cycle.
-reserved-	+00A	16 bits	Not in use.
BANDWIDTH_REQUEST_1	+00C	16 bits	Holds a number of bandwidth allocation units to be requested from the IRM node after bus reset.
CHANNEL_REQUEST_1	+00E	16 bits	Holds an isochronous channel number to be requested from the IRM node after bus reset.
BANDWIDTH_REQUEST_2	+010	16 bits	Holds an additional number of bandwidth allocation units to be requested from the IRM node after bus reset.
CHANNEL_REQUEST_2	+012	16 bits	Holds an additional isochronous channel number to be requested from the IRM node after bus reset.
-reserved-			Not in use.
SERIALPORT_DATA_ADDR	+020	64 bits	An address to which data received on serial port 2 will be relayed to.
SERIALPORT_DATA_OUT	+028	32 bits	Data written to this register will be output on serial port 2.
-reserved-			Not in use.
IDENTIFY	+100	16 bits	Non zero writes to this register will cause the port LEDs of the hardware to blink in order to allow for device identification.
POSTED_WRITE_PARTITION	+104	32 bits	Used by the Node Application Space.

Table 4.4: NC1-Transporter Address Map: mLAN Addressing

Isochronous channel numbers and bandwidth allocated for transmission, using both chips of an NCP05-based Transporter, are written to mLAN Space registers for re-allocation in the event of a bus reset. For example, the channel number and bandwidth for the mLAN-NC1 chip are written to the *CHANNEL_REQUEST_1* and *BANDWIDTH_REQUEST_1* registers of the mLAN Space (shown in Table 4.4) respectively. The additional channel number and bandwidth to be allocated for the mLAN-PH2 chip are written to the *CHANNEL_REQUEST_2* and *BANDWIDTH_REQUEST_2* registers of the mLAN Space (shown in Table 4.4) respectively. If any resources cannot be re-allocated, isochronous transmission stops.

For audio reception, FIFO buffers exist within the mLAN-PH2 chip to receive audio samples from the audio sequences. For each FIFO buffer, there are associated Core Space registers that enable isochronous channel and sequence number selection, as shown in Figure 3.15 on page 48. The channel number uniquely identifies the isochronous stream to receive, while the sequence number specifies the offset of the sequence to pick up within each isochronous packet cluster of the isochronous stream. Similarly, for MIDI data reception, FIFO buffers exist that extract MIDI messages from the MIDI sequences of an isochronous stream. In addition to the channel and sequence number Core Space registers associated with the mLAN-NC1 chip, there are additional Core Space registers for the mLAN-NC1 that allow for selection of a subsequence number within a particular sequence. Subsequence numbers are only relevant for MIDI, since up to eight MIDI messages can be multiplexed on each sequence. A subsequence number is equivalent to a multiplex index. A detailed description of MIDI multiplexing has been given in section 3.2.1.

Recall from section 3.2.3 on page 41 that synchronisation is required between the audio transmitting and receiving devices. Core Space registers associated with the mLAN-PH2 chip are available to specify whether the source of synchronisation word clock is internal within the Transporter or is received via the SYT field of incoming isochronous packets. Although similar Core Space registers associations are possible for the mLAN-NC1 chip, these registers are ignored, since the mLAN-NC1 chip is solely for MIDI transmission and reception. MIDI transmission and reception does not require such word clock synchronisation.

An overview of Yamaha Corporation's generic hardware capabilities has been given in this subsection. The current Transporter HAL API fully utilises all capabilities of Yamaha Corporation's hardware. The next subsection highlights the shortcomings of the current Transporter HAL API in utilising the capabilities of the Open Generic Transporter.

4.4.2 Shortcomings of the Current Transporter HAL API in the Context of the OGT Guideline Document

With reference to Figure 4.3 on page 55, the Transporter HAL API components that govern interaction between an Enabler and Transporters include the *Transporter*, *DeviceTransporter*, *PCTransporter*, and the optional *TransporterNodeApplication* classes. Vendor-specific Transporter implementations technically inherit and provide implementations for these classes. We do not explore the optional *TransporterNodeApplication* class implementations in detail, since the main focus of our investigation was on the mandatory node controller components of the Transporter HAL API. At the time of this writing, the OGT guideline document did not specify any guidelines for the optional node application component of a Transporter. However, Chapter 5 describes how our proposed plug-in loading mechanism affects the manner in which the capabilities of a Transporter's node application component are accessed. A full description of the *Transporter*, *DeviceTransporter*, and *PCTransporter* classes is given in Appendix A on page 206. Although we describe these classes in terms of the Redesigned Enabler, it should be borne in mind that they are based on the Basic Enabler specification which has been documented by Yamaha Corporation in the "mLAN Enabler Software Specification" [Yamaha Corporation, 2004a].

We now look at the shortcomings of the current Transporter HAL API capabilities, provided by the abstract *Transporter, DeviceTransporter*, and *PCTransporter* classes, in the context of the OGT guideline document. Reference is made to some of the functions described in Appendix A.

4.4.2.1 Isochronous Stream and Sequence End-Point Associations

In the context of the current Transporter HAL API, input isochronous streams correspond to isochronous streams that are received by a Transporter from the IEEE 1394 bus, while output isochronous streams refer to those that are sent onto the bus for transmission. The maximum number of isochronous channels/streams that can be received or transmitted by a device is queried via the *GetMaxIsochChannels* function of the *Transporter* class and shown below.

$$GetMaxIsochChannels(isInput, *pNumIsochChannels) \tag{4.1}$$

An *isInput* argument is used by the above function and also by the majority of Transporter HAL API functions. This argument is a boolean value that specifies the direction of the streams, namely *input* or *output*. In particular, *isInput* is true for inputs (reception) and false for outputs (transmission). It is important to understand this *input/output* concept since it is fundamental

to Transporter control. Each isochronous stream is referred to by an *isochID*. An isochID is a number within the range defined by the relation $0 \leq isochID \leq *pNumIsochChannels - 1$, where ***pNumIsochChannels*** is the maximum number of isochronous channels of a given direction returned by the *GetMaxIsochChannels* function (function (4.1)).

The current Transporter HAL API views each of the isochronous streams, described above, as having static associations with a number of sequences of different types. Two types of sequences are defined, namely *audio* and *MIDI* sequences. The maximum number of sequences of a particular type, for each isochronous stream, is queried via the following *Transporter* class function:

$$GetMaxSequences(isInput,\ isochID,\ seqType,\ *pMaxSequences) \tag{4.2}$$

Each of these sequences is identified by a sequence ID, *seqID*, and is in turn associated with a number of subsequences. The *seqID* is a number within the range defined by the relation $0 \leq seqID \leq *pMaxSequences - 1$, where ***pMaxSequences*** is the maximum number of sequences of a particular type, for a given isochronous stream (identified by *isochID* and *isInput*). The value for ***pMaxSequences*** is returned by the *GetMaxSequences* function (function (4.2)) as shown above. For each of the sequences, the maximum number of subsequences is queried via another Transporter class function, namely:

$$GetMaxSubsequences(isInput,\ isochID,\ seqType,\ seqID,\ *pNumSubs) \tag{4.3}$$

Each of the subsequences is identified by a subsequence ID, *subID*. The *subID* is a number within the range defined by the relation $0 \leq subID \leq *pNumSubs - 1$, where ***pNumSubs*** is the maximum number of subsequences of a given sequence (identified by a combination of *isInput, isochID, seqType* and *seqID* parameters). The value for ***pNumSubs*** is returned by the *GetMaxSubsequences* function (function (4.3)) as shown above. Subsequences are important when transmitting data that can be multiplexed, such as MIDI data. However, for data that cannot be multiplexed such as audio, the maximum number of subsequences is one, hence the only valid *subID* is zero. From a higher level, the Enabler views the combination of *isInput, isochID, seqType, seqID* and *subID* as uniquely identifying each of the sequence or subsequence end-points of a Transporter. These end-points represent the high level plug abstractions of the "mLAN Plug Abstraction Layer" described in section 3.3.2.1 on page 44.

We can now look at how the above *Transporter* class functionality is implemented in terms of the MAP4 hardware design. With regards to transmission, each of the transmission FIFO buffers on the mLAN-NC1 and mLAN-PH2 chips described in section 4.4.1 above, for example,

corresponds to a single output isochronous stream. The output stream corresponding to the mLAN-PH2 chip's FIFO buffer is associated with a maximum of 32 audio sequences, each with exactly one subsequence. The output stream corresponding to the mLAN-NC1 chip's FIFO buffer is associated with one MIDI sequence that has eight multiplexed subsequences. This relationship for the transmission FIFO buffers of the two chips is shown in Figure 4.13. It should

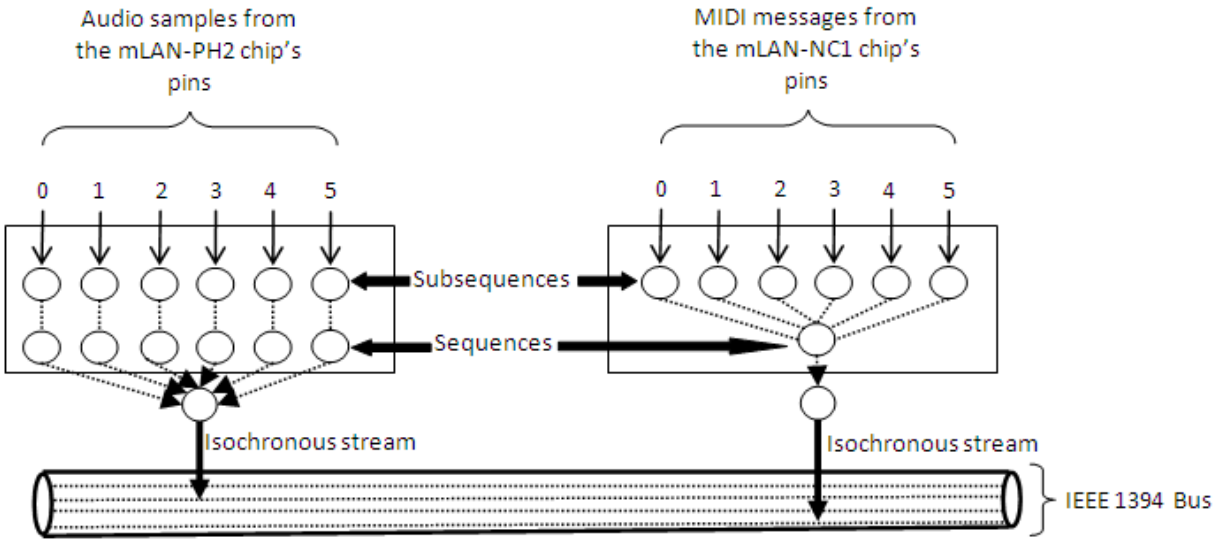


Figure 4.13: MAP4 Transmission FIFO Buffer Model for Current Transporter HAL API

be borne in mind that the diagram serves to demonstrate the concept only and does not show all the sequences. In particular, only six of the possible 32 audio sequences are shown for the mLAN-PH2 chip. Similarly, only six of the possible eight MIDI subsequences are shown for the mLAN-NC1 chip.

On the other hand, for reception, each of the reception FIFO buffers on either of the chips corresponds to an input isochronous stream/channel that is associated with one sequence, which has exactly one subsequence. This implies that each reception sequence or subsequence end-point can receive a sequence or subsequence from a unique isochronous stream, although more than one end-point can receive from the same isochronous stream. Thus, the number of end-points is equivalent to the maximum number of possible unique transmitters that a device can receive from. This relationship for the reception FIFO buffer of the MAP4 is shown in Figure 4.14. Note that the diagram serves to demonstrate the concept only and does not represent all reception FIFO buffers that are present on the mLAN-NC1 and mLAN-PH2 chips of the MAP4. For MIDI reception, there are additional Core Space registers that are not shown for each FIFO buffer. These registers allow for modification of the subsequence number, in addition to the sequence

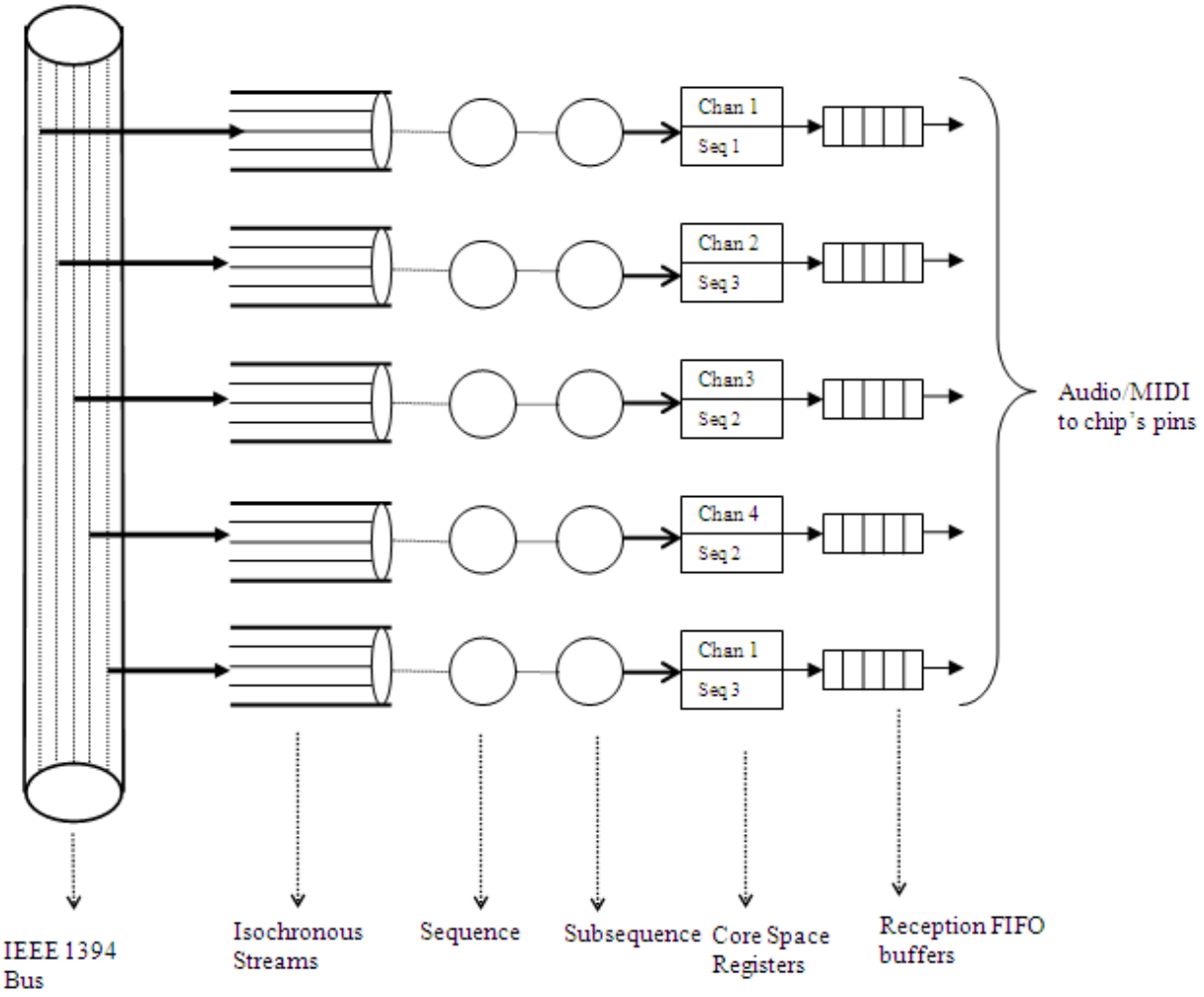


Figure 4.14: MAP4 Reception FIFO Buffer Model for Current Transporter HAL API

number. The subsequence number is only relevant to MIDI, since up to eight MIDI messages can be multiplexed on a single sequence.

On the Windows platform, the *PCTransporter* class is implemented in terms of a Windows mLAN streaming driver. It should be borne in mind that the mLAN streaming driver is tightly coupled with the *Transporter* class interface. This streaming driver exposes four input isochronous streams and one output isochronous stream. The single output isochronous stream is associated with up to 32 audio sequences with exactly one subsequence each, and up to two MIDI sequences with eight subsequences each. Two of the input isochronous streams are each associated with one MIDI sequence with eight subsequences. The remaining two input isochronous streams are each associated with 24 audio sequences which have exactly one subsequence. Additionally, one

of the two remaining input isochronous streams is also associated with a MIDI sequence with one subsequence.

As shown by the above descriptions for the MAP4 and the Windows implementation of the *PC-Transporter* class, it is evident that the current *Transporter* class functionality assumes a static association between isochronous streams and a number of sequence or subsequence end-points that terminate the streams. In particular, as shown in Figures 4.13 and 4.14, each isochronous stream is associated with a fixed number of sequences. Each sequence is in turn associated with a fixed number of subsequences, hence the static associations. These static constraints have influenced the choice of Transporter HAL API member functions, namely the *GetMaxIsochChannels*, *GetMaxSequences*, and *GetMaxSubsequences* functions (functions (4.1), (4.2), and (4.3)) which are described in this subsection starting from page 78.

In an effort to preserve bandwidth used by output isochronous streams, the number of sequences that are actually transmitted may be reduced or increased as needed via the following *Transporter* class function:

$$SetNumSequences(isInput,\ isochID,\ seqType,\ numSequences) \tag{4.4}$$

The implications of the static associations imposed by the *Transporter* class were further investigated in the context of the OGT plug-in implementation against the current Transporter HAL API. An important concept of the OGT that has been described in section 3.3.3.2 on page 50 is the possibility of static or dynamic clustering of multiple sequence end-points to an isochronous stream. This is made possible by the static or dynamic ISP to NCP associations.

The OGT's static ISP to NCP associations fit well with the interface defined by the *Transporter* class. However, dynamic associations pose certain problems, particularly on the input side of a Transporter. Unlike the MAP4 design, the OGT allows for implementations where the number of possible input isochronous streams (ISPs) is less than the number of input sequence end-points (NCPs). This implies that the number of input ISPs, as opposed to the number of sequence end-points in the case of the MAP4, determines the maximum possible number of transmitters that an OGT-based device can receive from. The advantage of dynamic association in this case is that if, for example, a connection is made to a particular NCP (sequence end-point), an available ISP is dynamically associated with the NCP. An ISP is available if it is not receiving any isochronous streams, or if it is neither associated with any active NCPs nor receiving SYT timing information. The ISP is set to receive isochronous streams with a particular channel number, while the NCP is set to receive a particular sequence of the isochronous stream. When all ISPs are in use, all

connections from other transmitted isochronous streams will fail unless one of the ISPs in use has a channel number that already matches that of the source isochronous stream for the new connections.

As shown above, the OGT relies on dynamic association between input ISPs and input NCPs for optimal functionality. On the other hand, the *Transporter* class interface imposes a static association constraint. The current OGT HAL plug-in implementation deals with this constraint by introducing a level of complexity to the implementation. In particular, such complexities have led to the creation of a *software memory mapping structure,* which represents each NCP as a sequence end-point that is statically associated with a *virtual input isochronous stream* as shown in Figure 4.15. A virtual isochronous stream emulates the role of a reception FIFO buffer

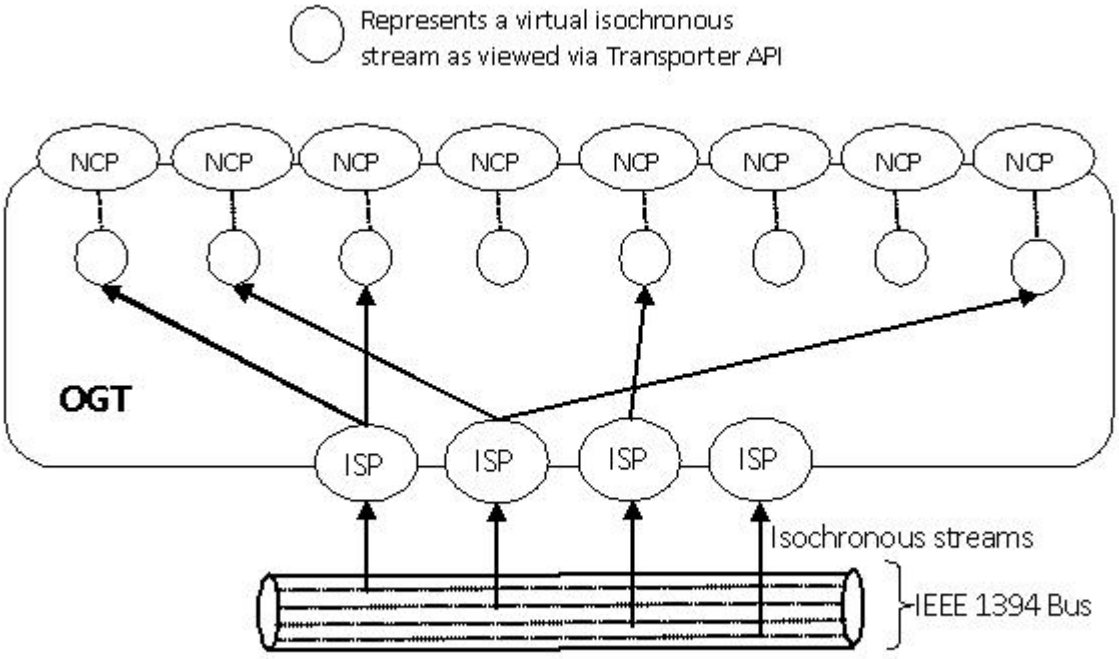


Figure 4.15: OGT Plug-In Mapping Structure Implementation for the Current Transporter HAL API

in terms of a design such as that for NCP05-based hardware architectures. When the *GetMaxIsochChannels* function (function (4.1) on page 78) is called to query the maximum number of input isochronous streams that an OGT-based Transporter can receive, the number returned is equivalent to the number of *virtual input isochronous streams*. Consequently, the *GetMaxSequences* and *GetMaxSubsequences* functions (functions (4.2) and (4.3) on page 79) both return

the value one as the maximum number of sequences and subsequences respectively, for either audio or MIDI sequence types. The Enabler, via the vendor-specific *Transporter* class implementation, configures OGT-based Transporters in terms of the static associations between the NCPs and the *virtual isochronous streams*. Within the OGT plug-in, the API functions are fulfilled in terms of dynamic associations between ISPs and NCPs.

Amongst other information, each of the *virtual isochronous streams* keeps track of the identifier (ID) of its associated NCP in terms of the OGT register set, and also the ID of the associated ISP in terms of the OGT register set. NCPs are dynamically associated with available ISPs when the channel number is set for each *virtual isochronous stream* via the *SetIsochChannel* function of the *Transporter* class as shown below:

$$SetIsochChannel(isInput,\ isochID,\ isochChannel) \tag{4.5}$$

Figure 4.16 shows a four-step process that summarises the process of associating NCPs with ISPs as described above. The first step is the instruction to set a particular channel number to a given

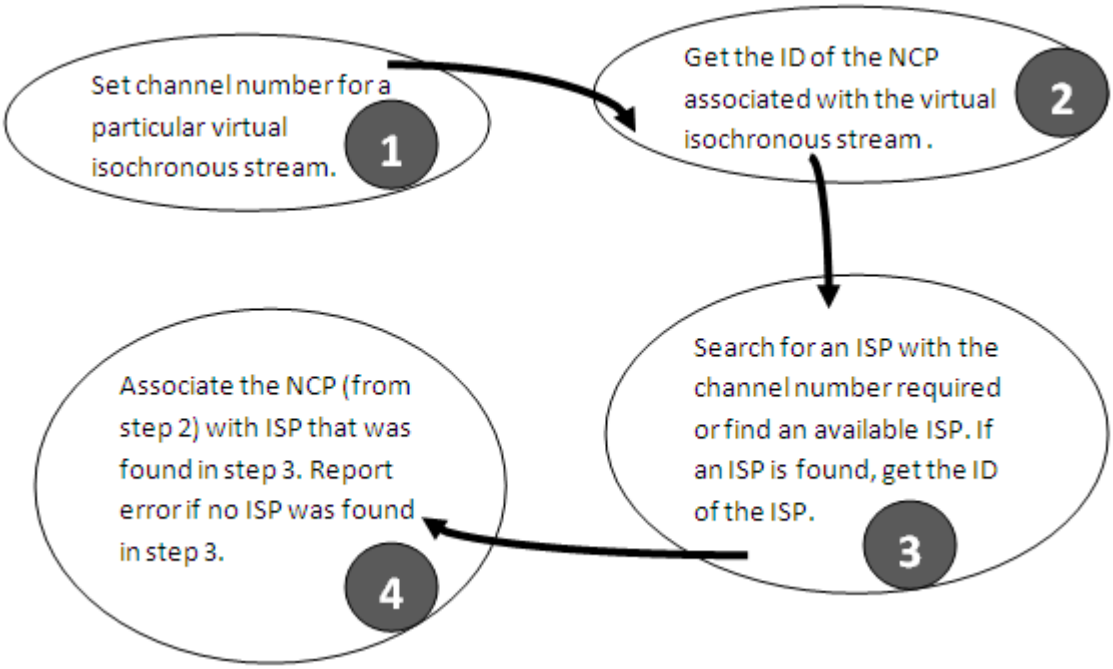


Figure 4.16: Current Transporter HAL API: ISP-to-NCP Association Process

virtual isochronous stream by calling the *SetIsochChannel* function. Step two of the process relates to the retrieval of the ID of the NCP that is associated with the *virtual isochronous stream*. Step three of the process comprises a search for an available ISP, and if an ISP is found, its ID

is retrieved. The last step results in an association between the NCP and the ISP. In terms of the OGT register set, there is an NCP register, the *NCP_ISP_ID* register, that keeps track of the ID of the NCP's associated ISP. It is the dynamic modification of this register that results in dynamic NCP-to-ISP associations. The level of complexity introduced by the *virtual isochronous stream* concept can be removed by having a Transporter HAL API that inherently allows for dynamic isochronous stream (ISP) to sequence end-point (NCP) association.

4.4.2.2 Starting and Stopping of Transmission or Reception

The *Transporter* class defines functions to collectively start and stop isochronous transmission or reception for all isochronous streams of a given direction. These functions are:

$$Start(isInput) \tag{4.6}$$

$$Stop(isInput) \tag{4.7}$$

Recall that output isochronous streams relate to the transmission side of a device. Collective control such as the above is a hardware specific constraint that does not always hold for all Transporters. In particular, the firmware for Yamaha Corporation's NCP05-based designs imposes the restriction that all chips commence or stop transmission/reception at the same time. However, in the context of the OGT, isochronous streams (ISPs) may be controlled individually unless otherwise specified. For output isochronous streams, collective control may result in wastage of isochronous resources. For example, there may be more than one output ISP sending packets onto the IEEE 1394 bus from an OGT-based device. With the current *Start* function of the *Transporter* class (function (4.6)), all output ISPs have to be started simultaneously. As mentioned in section 3.1.1.4 on page 31, before isochronous transmission starts, each transmitter (output ISP in terms of OGT) is allocated the necessary isochronous resources, namely an isochronous channel number and bandwidth. Supposing that only one of the output ISPs is required for transmission, all other output ISPs are wastefully allocated resources. This isochronous resource wastage reduces the total available isochronous resources on the IEEE 1394 network for each OGT device that is transmitting, and hence reduces the maximum possible number of simultaneously transmitting devices on the network. A need for a Transporter HAL API that allows for individual control over ISPs, where possible, became apparent. It should be borne in mind that no isochronous resource allocation is done for reception, and hence collective control over input ISPs does not result in resource wastage.

4.4.2.3 Word Clock Source Selection

With reference to the Redesigned Enabler object model shown in Figure 4.2 on page 54, when the Enabler enumerates the sequence end-points on a Transporter, it also creates an *MLANDeviceSyncBlock* object. In the current implementation, this object is associated with the first isochronous stream that has at least one audio sequence end-point. Synchronisation between a master device's clock and a slave device's clock is then handled via the corresponding *MLANDeviceSyncBlock* objects for each Transporter on the network. Section 3.2.3 on page 41 mentioned how synchronisation of a slave device's clock to a master device's clock is done. In particular, timing information is transmitted from the master device, via the SYT field of the CIP header, to the slave device. The *Transporter* class interface allows for this control via the following functions:

$$GetSYTSynchChannel(*pIsochChannel) \tag{4.8}$$

$$SetSYTSynchChannel(isochChannel) \tag{4.9}$$

The *GetSYTSyncChannel* function gets the isochronous channel number of the isochronous streams on which a device receives SYT timing information for synchronisation. If the device is not using SYTs for synchronisation, an invalid channel is returned. Conversely, *SetSYTSyncChannel* specifies the isochronous channel number for the packets that a device receives SYT timing information from. When an invalid channel number is specified, the device does not use SYT timing information for synchronisation. In such cases, the device is said to be operating using its local word clock source. The local word clock source is usually a device's built-in/internal clock (crystal oscillator).

From the above, the *Transporter* class interface assumes that a device can operate in two modes, namely using internal or SYT word clock sources. The former represents a word clock master device while the latter represents a word clock slave. The OGT defines the possibility of having additional local word clock sources. We refer to these additional local word clock sources as *external word clock sources*. Figure 4.17 shows an example of five possible word clock sources that may be present on an OGT-based Transporter. External word clock sources are so called because they do not necessarily originate from a built-in device clock but may originate from a source, external to the device, such as an external voltage controlled crystal oscillator (VCXO) or by reading the sample rate information of incoming ADAT or AES/EBU digital audio streams. While the OGT allows for selection of such external word clock sources within a device, the current *Transporter* class interface assumes that devices not acting as word clock slaves can only

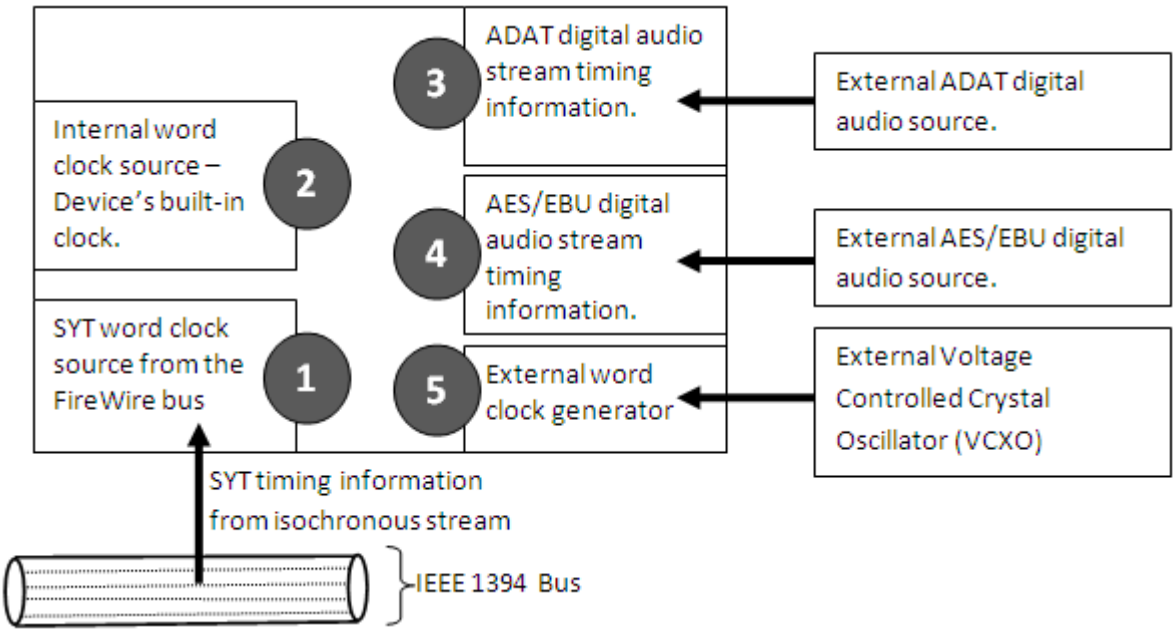


Figure 4.17: Example of Word Clock Sources on an Open Generic Transporter

use their built-in word clock sources. There is no mechanism available to select the external word clock sources. With reference to Figure 4.17, the current Transporter HAL API allows for selection of word clock sources labelled 1 and 2. All other word clock sources labelled 3, 4, and 5 cannot be selected. This constraint leads to the underutilisation of OGT capabilities and indicated another inadequacy of the current Transporter HAL API. In particular, we identified a need to allow for more flexible word clock source selection within the Transporter HAL API, which takes into account external word clock sources.

4.4.2.4 Multiple Word Clock Outputs

One of the roles of the the *MLANDeviceSyncBlock* object (mentioned in section 4.4.2.3) in setting up the word clock master-slave relationships is to ensure that the sampling frequencies of the master and slave device clocks are the same. The current *Transporter* class interface allows sampling frequencies to be set for each isochronous stream via the function:

$$SetSFC(isInput, isochID, sfc) \tag{4.10}$$

However, Transporter plug-ins are currently implemented in such a way that changes made to the sampling frequency (SFC) of the isochronous stream that is associated with the *MLANDe-*

viceSyncBlock object result in similar changes to SFCs for all other isochronous streams on a device. This implies that the current Enabler/Transporter implementation assumes that devices can only transmit or receive audio sequences that are sampled at one common sampling frequency.

While the current implementation demonstrates a single/common SFC assumption, the OGT guideline refines this approach by defining a number of word clock outputs (Sync Output Modules in terms of the OGT register set [Audio Engineering Society - Standards Committee, 2005]). Each word clock output can be associated with one of a number of concurrently usable word clock sources. A word clock source's sampling frequency is set independent of other concurrently usable word clock sources. Each of the ISPs, *input* or *output*, is in turn associated with one of a number of word clock outputs. ISPs are then associated with a number of NCPs. This implies that NCPs associated with a particular ISP operate at the sampling frequency that is determined by the ISP's associated word clock output. The word clock source associated with the word clock output ultimately determines the sampling frequency in use. This means that a Transporter is capable of simultaneously transmitting or receiving audio sequences that are sampled at different sampling frequencies. All sequences within a single isochronous stream share the same sampling frequency. Therefore, in order to concurrently use different sampling frequencies, the audio sequences have to belong to different isochronous streams. However, it should be borne in mind that audio connections cannot be made between source and destination plugs that are using different sampling frequencies.

The *SetSFC* function (function (4.10)) suggests that it may be possible to achieve the same result with the current Transporter HAL API, since SFCs can be set for each isochronous stream individually. However, upon further investigation, it became apparent that the *Transporter* class interface does not provide a mechanism to query the number of different sample rates that can be concurrently used on device, a feature that is inherent within the word clock output concept of the OGT. At the time of this writing there were no OGT firmware implementations that implemented more than one word clock output. However, to cater for this future need, the Transporter HAL API ought to expose such capabilities to the Enabler.

4.4.2.5 Handling Device-Specific Transporter Quirks

There are some quirks that are inherent within the operation of certain Transporters as a result of aspects such as their underlying hardware capability restrictions. An example of a quirk for NCP05-based Transporters is the reduction in the number of audio channels that can be

transmitted or received at high (88.2 kHz and above) sample rates. This is because more data is transmitted at high sample rates, while there are restrictions in the amount of data that the chips can process per unit time. This ultimately reduces the total number of sequences that may be transmitted or received by a device per data block. In dealing with this quirk in particular, the current Transporter HAL API merely reduces the number of plugs that are visible to connection management applications when sample rates are changed from low to high sample rates. The change in the number of plugs that are visible to connection management applications upon sample rate changes leads to a non-obvious design which may result in users being "surprised" by system behaviour [Galitz, 2002].

The OGT guideline document takes into account such quirks by defining a Plug Layout Table implementation which is composed of a number of Plug Layouts. A Plug Layout represents a mutually exclusive Transporter configuration. Each Plug Layout contains a number of NCPs, ISPs, word clock outputs and word clock sources that are concurrently usable. The current Transporter HAL API has no concept of a Plug Layout Table. In the case of the OGT plug-in implementation against the current Transporter HAL API, changes to Plug Layouts are only made upon changes in sample rate since the Transporter HAL API does not explicitly cater for Plug Layouts.

The Plug Layout concept mentioned above was explored further and bandwidth was identified as another possible aspect that can be dealt with by Plug Layouts. In particular, in environments where routing requirements are known in advance, manufacturers can create various Plug Layouts that ensure optimal bandwidth usage. For example, in situations where the maximum number of audio channels required for transmission is known, the manufacturer can create a Plug Layout that exactly uses the required bandwidth. Such a possibility implies that changing Plug Layouts upon sample rate changes only may lead to an inadequacy when dealing with this type of quirk using the current Transporter HAL API. A need for an API that allows for a structured way of dealing with the various quirks of devices became clearer.

4.5 Chapter Summary

This chapter has highlighted shortcomings of the current plug-in loading mechanism. We have identified the need for a plug-in loading mechanism that allows for robust Transporter HAL API versioning while reducing the level of binary dependence between the Enabler and Transporter plug-ins. We have shown how Yamaha Corporation's original mechanism attempted to resolve

this. However, further investigation revealed that, although it is functional, the mechanism is error prone and leads to an increased overhead incurred as a result of the need for an intermediary function to resolve any Transporter HAL API function calls.

In addition to highlighting the shortcomings of the current plug-in mechanism, we have also revealed the shortcomings of the current Transporter HAL API in the context of the OGT guideline document. These shortcomings stem from inadequacies of a Transporter HAL API that was designed prior to the existence of the OGT concept. The shortcomings include:

- Complexities introduced to the OGT plug-in implementation in order to accommodate the dynamic ISP to NCP associations.
- Bandwidth wastage and its consequences on the overall performance of an mLAN network.
- The restrictive word clock source selection which leads to underutilisation of certain OGT capabilities.
- Unclear representation of a future need for Transporters to simultaneously operate at multiple sampling frequencies.
- Lack of a structured way to deal with quirks within different devices.

The next chapter, Chapter 5, describes the design and implementation of our proposed plug-in mechanism. Our plug-in mechanism sought to resolve the shortcomings identified in this chapter relating to the current plug-in mechanism.

Chapter 5

Proposed Transporter Plug-In Mechanism

Chapter 4 has highlighted shortcomings within the current Transporter plug-in mechanism and Transporter HAL API. These two components comprised the two main research areas for our investigation. In this chapter, we describe how the shortcomings of the current plug-in mechanism were resolved.

The Transporter plug-in mechanism was refined to allow for robust Transporter HAL API versioning with minimal binary dependence between the Enabler and Transporter plug-ins. Our proposed plug-in mechanism is based on the fundamentals of COM, and mainly uses pointers to interfaces as opposed to pointers to objects. The use of COM fundamentals allows us to use Microsoft COM in its standard form for the Windows plug-in implementation. However, we also show how we implement these COM fundamentals on two other popular platforms, namely Linux and Macintosh.

5.1 Proposed Redesigned Enabler Design Overview

Figure 5.1 shows our proposed object model for the Redesigned Enabler. By comparing this object model with the current Redesigned Enabler object model shown in Figure 4.2 on page 54, it can be seen that the mandatory *NodeController* class now depends on an *INC_PLUGIN* COM interface. The *INC_PLUGIN* COM interface defines methods that are required to access the node controller capabilities of Transporters. Similarly, the *NodeApplication* class now depends on an *INA_PLUGIN* COM interface. The *INA_PLUGIN* COM interface defines methods that are required to access the node application capabilities of Transporters.

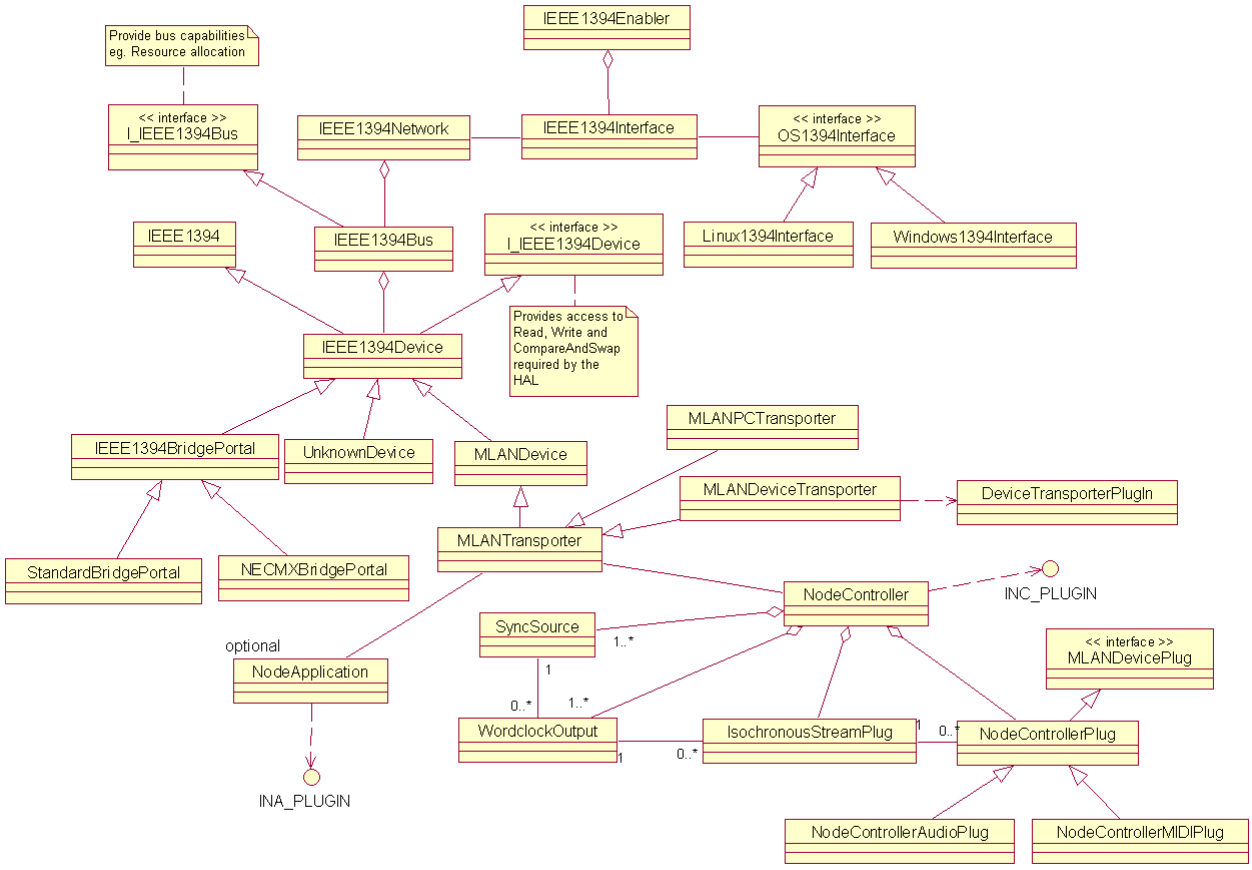


Figure 5.1: Proposed Redesigned Enabler Object Model

Recall from section 4.4.2 that our investigation focused on the mandatory node controller component of Transporters. We did not explore the node application component, since, at the time of this writing, the OGT guideline document did not specify any node application capabilities that could be utilised by OGT-based Transporters. For this reason, the *NodeApplication* class in Figure 5.1 is not expanded in terms of its associated components, except for showing its dependence on the *INA_PLUGIN* COM interface. However, the *NodeController* class has been expanded to show its associated components in addition to its dependence on the *INC_PLUGIN* COM interface.

The next section describes our proposed Transporter HAL model in the context of our proposed Redesigned Enabler design as shown in Figure 5.1.

5.2 Proposed Enabler HAL Design Overview

Recall from section 4.3.4 that we aimed to create a plug-in mechanism that:

- Allows for better Transporter HAL API versioning.
- Reduces the level of binary dependence between the Enabler and Transporter plug-ins.
- Eliminates the need for an intermediate function call for each Transporter HAL API function call.
- Is portable across different operating systems.

Figure 5.2 illustrates our proposed HAL object model. We highlight the conceptual simplic-

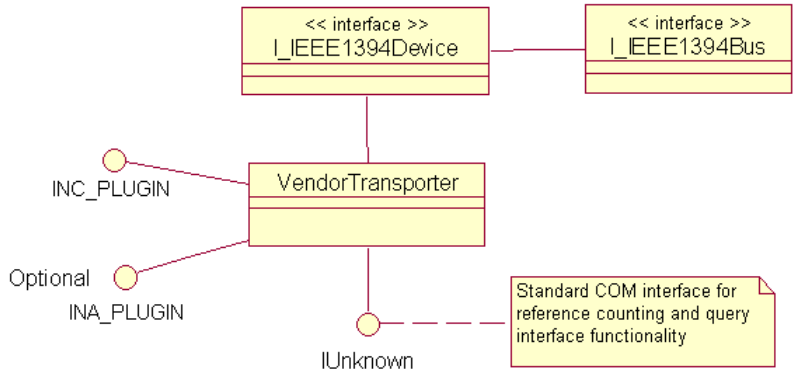


Figure 5.2: Proposed Redesigned Enabler Hardware Abstraction Layer Object Model

ity of our proposed design by comparing it with the current Redesigned Enabler HAL design as shown in Figure 4.3 on page 55. Our design is based on the understanding of COM as an object-oriented, interface-based programming architecture [Troelsen, 2000]. In particular, we have defined two fundamental COM interfaces, namely *INC_PLUGIN* and *INA_PLUGIN*. The *INC_PLUGIN* COM interface defines methods that access the mandatory node controller capabilities of Transporters, while the *INA_PLUGIN* COM interface defines methods that access the optional node application capabilities of Transporters. The *VendorTransporter* class is a COM component that provides device-specific implementations for *INC_PLUGIN* COM interface methods and, optionally, for *INA_PLUGIN* COM interface methods.

In building dynamically composable systems, such as an Enabler and its associated Transporter plug-ins, interfaces are treated as immutable binary and semantic contracts that never change

[Box, 1998]. Once defined, any additional interface methods are defined in their own new COM interfaces which are also implemented by the *VendorTransporter* class. This allows for Transporter HAL API versioning using techniques that are inherent within COM. In particular, there is a mechanism that allows for querying of COM interfaces that are implemented by the *VendorTransporter* class in addition to the mandatory *INC_PLUGIN* COM interface. For example, if additional Transporter HAL API methods are required to access new node controller capabilities of Transporters, a new COM interface that defines methods to access the additional node controller capabilities is created. Figure 5.3 shows how a new node controller COM interface, *INC_PLUGIN_v2*, is added to the original model shown in Figure 5.2. It is possible to

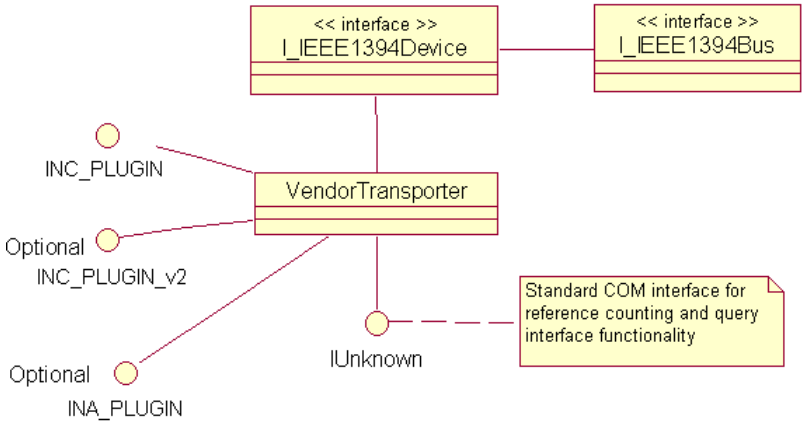


Figure 5.3: Proposed Transporter HAL API Versioning

query the existence of the *INC_PLUGIN_v2* COM interface for any Transporter plug-in via its mandatory *INC_PLUGIN* COM interface. Note that the manner in which the existence of the *INC_PLUGIN_v2* COM interface is queried is also similar to the manner in which the existence of the optional *INA_PLUGIN* COM interface is queried.

The *IUnknown* COM interface is the parent of all COM interfaces [Armstrong and Patton, 2000; Box, 1998; Troelsen, 2000]. It is the only COM interface that does not derive from another COM interface. All legal COM interfaces must derive from *IUnknown* directly, or indirectly from one other COM interface which derives from *IUnknown* either directly or indirectly. The *IUnknown* interface defines a *QueryInterface* method to query the existence of COM interfaces that the *VendorTransporter* class may implement. In addition, two other methods are defined to manage the lifetime of a *VendorTransporter* COM component, namely *AddRef* and *Release*. The C++ definition of *IUnknown* is shown below as adapted from Box [1998]:

```
interface IUnknown {
```

```
    virtual HRESULT STDMETHODCALLTYPE
         QueryInterface(REFIID riid, void **ppv) = 0;
    virtual ULONG   STDMETHODCALLTYPE AddRef(void) = 0;
    virtual ULONG   STDMETHODCALLTYPE Release(void) = 0;
};
```

The *STDMETHODCALLTYPE* macro is required to produce COM-compliant stack frames for the target platform. For example, with the Microsoft C++ compiler, the macro expands to *__stdcall* when targeting Win32 platforms. *HRESULTs* are 32-bit integers that provide information to the caller's runtime environment about the type of errors that may have occurred when a COM method is called. Virtually all COM methods return an *HRESULT*. The *QueryInterface* method is used for runtime type discovery which enables identification, using interface GUIDs (*REFIID*), of interfaces that a COM component implements. The *AddRef* method is used to notify a COM component (*VendorTransporter* in terms of Figure 5.2) that an interface pointer has been duplicated. Conversely, the *Release* method notifies a COM component that an interface pointer has been destroyed and any resources that the object held on behalf of the client[1] can be released. The use of *AddRef* and *Release* methods by clients is known as reference counting. Reference counting is essential for effective COM usage, and rules are provided as a guideline for when *AddRef* or *Release* ought to be called [Box, 1998; Troelsen, 2000]. Since our proposed design attempts to handle plug-in loading in a pure COM manner, one of the implications of the design is that the Enabler is required to handle reference counting explicitly. Such reference counting has been implemented in accordance with the various guidelines available.

As shown in Figure 5.2, the *VendorTransporter* class is associated with an *I_IEEE1394Device* interface. We defined this interface in an attempt to remove binary dependence between the Enabler and Transporter plug-ins. The *I_IEEE1394Device* interface defines methods that enable Transporter plug-ins to carry out asynchronous transactions. Transporter plug-ins use asynchronous transactions to send control commands to the actual Transporters on a network, in order to fulfil connection management requests from an Enabler. In addition, functionality to get a handle to an *I_IEEE1394Bus* pointer is available. The definition of the *I_IEEE1394Device* interface is shown below:

```
class I_IEEE1394Device{
public:
```

[1]The Enabler is the client in this context, since it is responsible for loading Transporter plug-ins.

```
    //Asynchronous transaction functionality
    virtual OSError  ReadQuadlet(...) const = 0;
    virtual OSError  ReadBlock(...) const = 0;
    virtual OSError  WriteQuadlet(...) const = 0;
    virtual OSError  WriteBlock(...) const =0 ;
    virtual OSError  CompareAndSwap(...) const = 0;
    //Return the bus interface pointer
    virtual I_IEEE1394BusPtr GetIEEE1394BusInterfacePtr()
       = 0;

};
```

As shown above, the *I_IEEE1394Device* interface is associated with an *I_IEEE1394Bus* interface. The *I_IEEE1394Bus* interface defines methods that Transporter plug-ins can use to query, allocate, and de-allocate isochronous resources (bandwidth and isochronous channel numbers) on the IEEE 1394 bus. The definition of the *I_IEEE1394Bus* interface is shown below:

```
class I_IEEE1394Bus {
public:


    virtual OSError GetChannelsAvailable(...) const = 0;
    virtual OSError AllocateChannel(...) = 0;
    virtual OSError AllocateChannel(...) = 0;
    virtual OSError ReleaseChannel(...) = 0;
    virtual OSError GetBandwidthAvailable(...) const = 0;
    virtual OSError AllocateBandwidth(...) = 0;
    virtual OSError ReleaseBandwidth(...) = 0;
    virtual void    GetBusGeneration(...) const = 0;

};
```

Although the use of virtual functions in the *I_IEEE1394Device* and *I_IEEE1394Bus* interfaces restricts the implementation of the Enabler and Transporter plug-ins to C++, we have reduced the binary dependence between the Enabler and its associated Transporter plug-ins. Figure 4.3 on page 55 shows how, in the current Enabler implementation, the *VendorTransporter* interface is ultimately dependent on the *Transporter* and *IEEE1394Device* classes. This creates a binary

dependence between Transporter plug-ins and the Enabler, since Transporter plug-ins are dependent on an Enabler library which contains the implementation of the functions that are now defined by our *I_IEEE1394Device* and *I_IEEE1394Bus* interfaces above. The dependence on an Enabler library implies that the implementation of Transporter plug-ins is also restricted to the programming language in which the Enabler is written, namely C++. Thus, while our proposed mechanism maintains this C++ restriction, it decouples the implementation between the Enabler and its associated Transporter plug-ins. An Enabler library is no longer required for Transporter plug-ins to be created. In order to create Transporter plug-ins that conform to our proposed HAL design, all that is required are the *I_IEEE1394Device* and *I_IEEE1394Bus* interface definitions, and the custom type definitions such as *OSError*. We use virtual functions for our interface definitions, since all C++ compilers on a given platform implement the virtual function call mechanism equivalently [Box, 1998].

Up to this point, we have shown how our proposed HAL design eliminates the need for an intermediary function call that resolves each Transporter HAL API function call. In particular, we have accomplished this by creating a COM interface which defines methods that access the node controller capabilities of Transporters and another optional COM interface to access the node application capabilities of Transporters. Methods defined by these COM interfaces are directly implemented by a *VendorTransporter* COM component. We have shown how Transporter HAL API versioning is achieved through the use of the *QueryInterface* capability which is inherent within COM. Interfaces have also been used to attempt to reduce the level of binary dependence between the Enabler and Transporter plug-ins.

The next section focuses on how Transporter plug-ins that comply with our proposed HAL design are loaded by our proposed version of the Redesigned Enabler. The section also describes how our design has been implemented on three popular operating systems, namely Windows, Linux, and Macintosh.

5.3 Proposed Plug-In Loading Mechanism

Section 4.3 on page 59 has described the current plug-in loading mechanism. Our proposed plug-in loading mechanism is similar to the current mechanism up to the point of creation of a new *MLANDeviceTransporter* or *MLANPCTransporter* instance, as shown in Figure 4.4 on page 59. In particular, new mLAN devices are identified on the IEEE 1394 bus and an object of type *MLANDeviceTransporter* is created for each mLAN device found. We modified the process that

leads to the creation of an instance of the *MLANDeviceTransporter* class (shown in Figure 4.5 on page 60) to follow our proposed HAL design.

Figure 5.4 shows a sequence diagram that describes our proposed *MLANDeviceTransporter* instance creation process. The first difference to note is the absence of the *DeviceTransporter* class

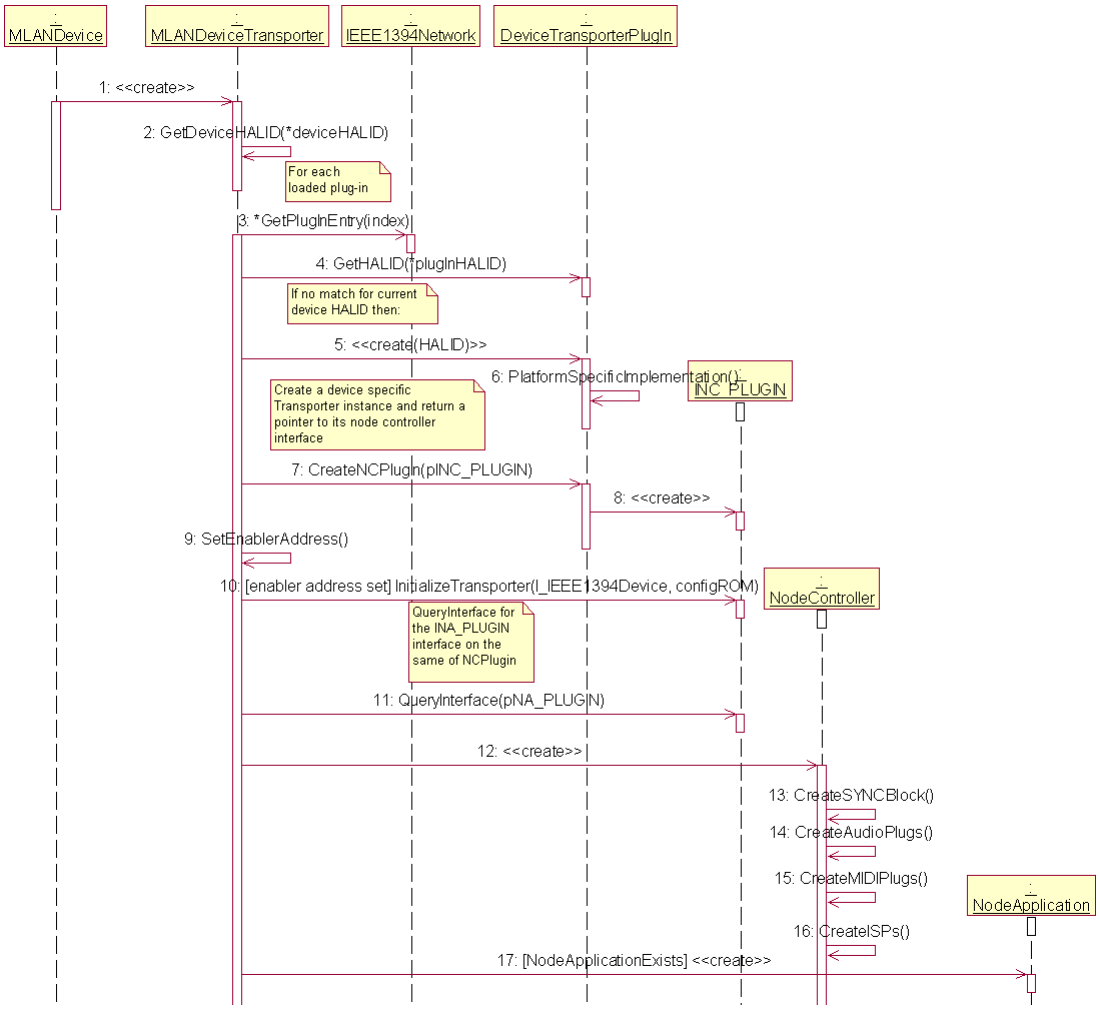


Figure 5.4: Proposed *CreateMLANDeviceTransporter* Sequence Diagram

which has been shown in Figure 4.5 on page 60. In the constructor of *MLANDeviceTransporter*, we now begin by retrieving the HAL ID of the mLAN device from its config ROM information. A search is then done through all the loaded Transporter plug-ins to check if a plug-in with a matching HAL ID has already been loaded. If the plug-in is not already loaded, an attempt is made to load the plug-in in a platform-specific manner by creating a new instance of the *DeviceTransporterPlugin* class, as shown by the *"<<create(HALID>>"* sequence in Figure 5.4. We

propose a new mechanism for plug-in loading on Windows, Linux, and Macintosh platforms. Sections 5.3.1, 5.3.2, and 5.3.3 describe the specifics of these three platforms in detail.

Once the plug-in is loaded, a pointer to the *DeviceTransporterPlugin* object is stored in an array of pointers to similar objects, where each object represents a loaded plug-in. Assuming that a plug-in is loaded, an uninitialised instance of the *VendorTransporter* COM component (shown in Figure 5.2) is created by calling the *CreateNCPlugin* function of the *DeviceTransporterPlugin* class, as shown by the "*CreateNCPlugin(pINC_PLUGIN)*" sequence in Figure 5.4. The *CreateNCPlugin* function is implemented in a platform-specific manner and returns a pointer to the *VendorTransporter* COM component. We have indicated that the *INC_PLUGIN* COM interface is mandatory for all Transporter plug-ins, hence the pointer returned by *CreateNCPlugin* is of type *INC_PLUGIN*. The existence of optional COM interfaces that may be implemented by the *VendorTransporter* COM component is queried via the *INC_PLUGIN* COM interface pointer by calling the *QueryInterface* function.

After the uninitialised *VendorTransporter* COM component is created, the Enabler address[2] is set via the *MLANDeviceTransporter* object, as shown by the "*SetEnablerAddress()*" sequence in Figure 5.4. This results in the unique node ID of the workstation (on which the Enabler is running) being written to a register of the Transporter. Note that there may be more than one Enabler on an IEEE 1394 bus. However, each Transporter is controlled by one Enabler. If setting the Enabler address succeeds, it implies that the specified Transporter is now under the control of the Enabler. Upon successfully setting the Enabler address, the newly created *VendorTransporter* COM component is initialised by calling the *InitializeTransporter* function of the *INC_PLUGIN* COM interface. It is during this initialisation that a pointer to the *I_IEEE1394Device* interface (mentioned in section 5.2) is forwarded to the *VendorTransporter* COM component for its subsequent use.

After initialisation, a new instance of the *NodeController* class (shown in Figure 5.1) is created, as shown by the "< <*create*> >" sequence that is labelled 12 in Figure 5.4. This *NodeController* object is responsible for fulfilling connection management requests, and hence it is dependent on the *INC_PLUGIN* COM interface as shown in Figure 5.1. We defined methods of the *INC_PLUGIN* COM interface in a manner that resolves shortcomings identified in section 4.4.2. The *INC_PLUGIN* COM interface now forms the foundation of the Transporter HAL API and its important functions are described in Chapter 6. If the *VendorTransporter* COM component implements the *INA_PLUGIN* COM interface, a new instance of the *NodeApplication* class

[2]The Enabler is aware of Enabler address offset for each type of Transporter.

(shown in Figure 5.1) is created, as shown by the "*[NodeApplicationExists] <<create>>*" sequence in 5.4. However, further examination of the *NodeApplication* class is beyond the scope of this thesis.

In the context of the above, we now describe the platform-specific implementations of our proposed plug-in loading mechanism. The first implementation was done on the Windows platform since it inherently supports the development and use of COM components. We also describe how we enabled portability of the implementation to Linux and Macintosh platforms.

5.3.1 Proposed Windows Transporter Plug-In Loading Mechanism

Section 4.3.1 has described how plug-in loading is currently done on Windows. In particular, it is done in the constructor of the *DeviceTransporterPlugin* class (shown in Figure 5.1). A conceptual summary of this mechanism has been shown in Figure 4.10 on page 66. We propose a refined plug-in loading mechanism that complies with our proposed HAL design. Figure 5.5 shows a sequence diagram for our proposed Transporter plug-in loading mechanism and creation of a *VendorTransporter* COM component under Windows. These two aspects have platform-specific implementations and have not been shown in detail in Figure 5.4.

As described in section 4.3.1, and also shown by the "*CoInitialize*" sequence in Figure 5.5, the starting point of the *DeviceTransporterPlugin* constructor is the loading of the COM libraries by calling the *CoInitialize* function. This is followed by an iterative search of the registry for the plug-in with a matching HAL ID, as described in section 4.3.1. Up to this point, everything is done in exactly the same manner as before. We now highlight and provide supporting motivations for the changes we propose for plug-in loading.

For each HAL plug-in that exists within the registry, we are no longer creating a single instance of the COM component using the *CoCreateInstance* function. We acknowledge the possibility of having more than one mLAN device using the same plug-in. The *CoCreateInstance* function creates a single uninitialised object of the class that is associated with a specified class identifier (CLSID). On the contrary, we aim to create multiple objects based on a single CLSID. The Microsoft Developer Network documentation provides guidelines on how this can be achieved [Microsoft Developer Network, 2007b]. In particular, the guidelines recommend the use of the *CoGetClassObject* function, which we are now making use of. The *CoGetClassObject* function provides a pointer to an interface on a class object associated with a specified CLSID. An excerpt of the guidelines is quoted below to indicate our motivation for following this route. Note that some text has been omitted and this is indicated by the ellipsis.

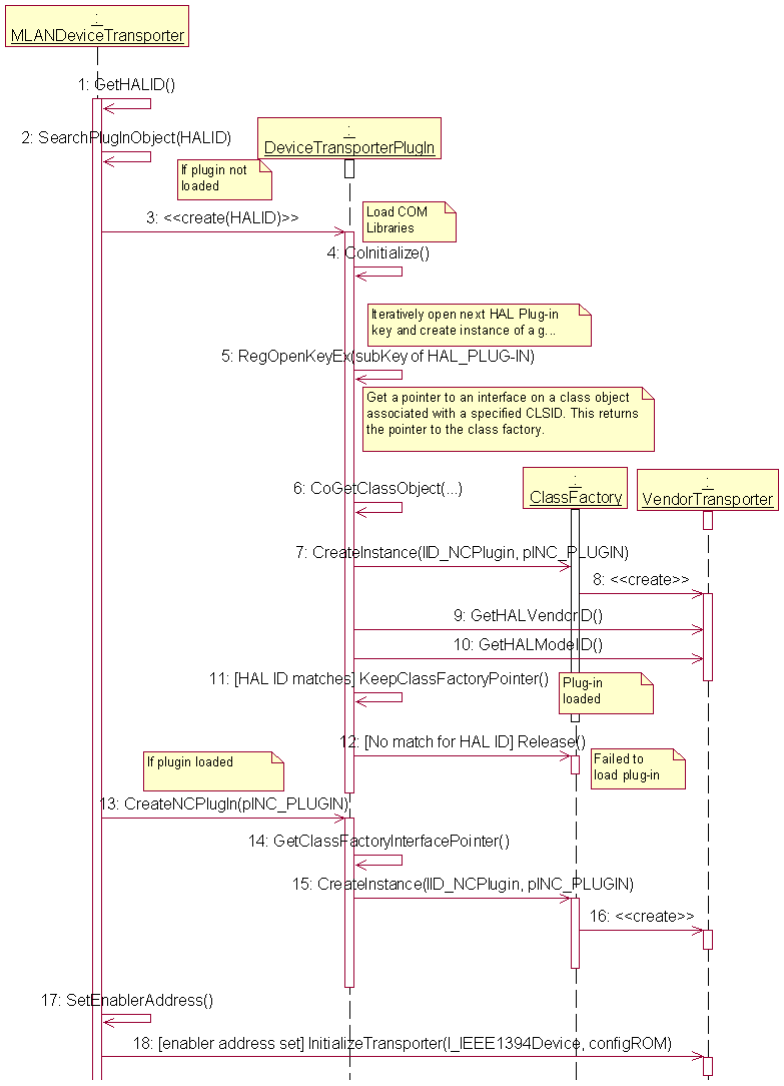


Figure 5.5: Proposed Plug-In Loading Mechanism on Windows

"Call CoGetClassObject directly when you want to create multiple objects through a class object for which there is a CLSID in the system registry. ... Most class objects implement the IClassFactory interface. You would then call IClassFactory::CreateInstance to create an uninitialised object. It is not always necessary to go through this process. To create a single object, call instead the CoCreateInstanceEx function, which allows you to create an instance on a remote machine. This replaces the CoCreateInstance function, which can still be used to create an instance on a local machine. Both functions encapsulate connecting to the class object, creating the instance, and releasing the class object." [Microsoft Developer Network, 2007b]

As shown by the "*CoGetClassObject(...)*" sequence in Figure 5.5, a handle to the class object is retrieved in each iteration of the search by calling the *CoGetClassObject* function with the CLSID of the each HAL plug-in within the registry. This class object implements the *IClassFactory* interface. The *CreateInstance* function is the most important function of the *IClassFactory* interface that we make use of. The *CreateInstance* function is called in order to create an uninitialised instance of the *VendorTransporter* COM component, as shown by the "*CreateInstance(IID_NCPlugin, pINC_Plugin)*" sequence that is labelled 7 in Figure 5.5. This instance is referenced via a pointer to the mandatory *INC_PLUGIN* COM interface. The HAL Vendor ID and HAL Model ID are then queried via the *GetHALVendorID* and *GetHALModelID* functions of the *INC_PLUGIN* COM interface, respectively. Note that a combination of the HAL Vendor ID and HAL Model ID represents a HAL ID. If the HAL ID of the plug-in matches the one intended to be loaded, one can be certain that the plug-in has been successfully loaded. For a successfully loaded plug-in, the pointer to the class object that would have been retrieved by calling the *CoGetClassObject* function is kept in memory, otherwise the pointer to the class object is released before the next iteration of the search, as shown by the "*[No match for HAL ID] Release()*" sequence that is labelled 12 in Figure 5.5.

After a plug-in has been loaded, creation of *VendorTransporter* COM component instances for devices that use the plug-in is done by calling the *CreateInstance* function of *IClassFactory* interface via the pointer to the class object. This process is shown by the sequences labelled 13 through to 16 in Figure 5.5. Only when the plug-in is unloaded will the pointer to the class object be released. A plug-in is unloaded when the last device that makes use of the plug-in is removed from the IEEE 1394 bus. In the current plug-in loading mechanism, a COM interface function such as *DisposeTransporter* has been defined to explicitly free up memory being used by an instance of a Transporter. We have removed this explicit memory management and replaced it with COM reference counting (*AddRef* and *Release* functions of a COM interface), as described in section 5.2.

The *CoCreateInstance* function is suited to creating a single object, while the nature of connection management environments requires that multiple objects be created for the same plug-in. The *CoCreateInstance* function encapsulates functions that initially retrieve a class object, then create an instance of a COM component via the class object, and then finally release the class object. Using *CoCreateInstance* for each device on a network brings with it the unnecessary overhead of getting a class object and releasing it each time an instance for a device is created. Our proposed approach removes this overhead for each instance of a COM component created. In particular, we get the class object when the first device that uses a plug-in is added to the

network, and only release the class object when the last device that uses the same plug-in is removed from the network. The reduction in overhead may be insignificant. However, it results in an implementation that truly models COM as a plug-in mechanism.

The next two sections describe how the COM implementation we have just described was ported to Linux and Macintosh platforms. Bear in mind that these two platforms do not allow software developers to write COM components in the manner in which Microsoft have provided for the Windows platform. We show how we use the capabilities of each of these platforms, Linux and Macintosh, to port our Windows COM implementation.

5.3.2 Proposed Linux Transporter Plug-In Loading Mechanism

We continue to use a shared library approach for plug-in loading on Linux, as described for the current implementation of the plug-in loading mechanism in section 4.3.2, although with some design modifications. A conceptual summary of the current plug-in mechanism on Linux has been shown in Figure 4.12 on page 69. We propose that shared libraries representing Transporter plug-ins expose three global functions that are declared as follows:

```
extern "C" INC_PLUGIN* CreateInstance()
extern "C" void GetHALVendorID(LONG* pHALVendorID)
extern "C" void GetHALModelID(LONG* pHALModelID)
```

Figure 5.6 shows a sequence diagram illustrating our proposed "COM" implementation of the plug-in loading mechanism on Linux. Plug-in loading occurs in the constructor of the *Device-TransporterPlugin* class (shown in Figure 5.1). The starting point of this constructor is locating the shared libraries of all Transporter plug-ins that interact with the Enabler by calling the *system* function, as shown by the "*system(...)*" sequence in Figure 5.6, and also described in detail in section 4.3.2. This is followed by an iterative search for a plug-in with a matching HAL ID, as shown by the sequences labelled 5 through to 11 in Figure 5.6. In particular, each of the shared libraries is loaded by calling the *dlopen* function, while the *dlsym* function is used to get the addresses of the *GetHALVendorID* and *GetHALModelID* functions of the loaded shared library. If either of the addresses cannot be retrieved, the shared library is unloaded by calling the *dlclose* function. Upon retrieval of the addresses for the *GetHALVendorID* and *GetHALModelID* functions, the HAL Vendor ID and HAL Model ID of the plug-in are obtained by calling the corresponding functions. If the HAL ID of the plug-in matches the one that is required, the

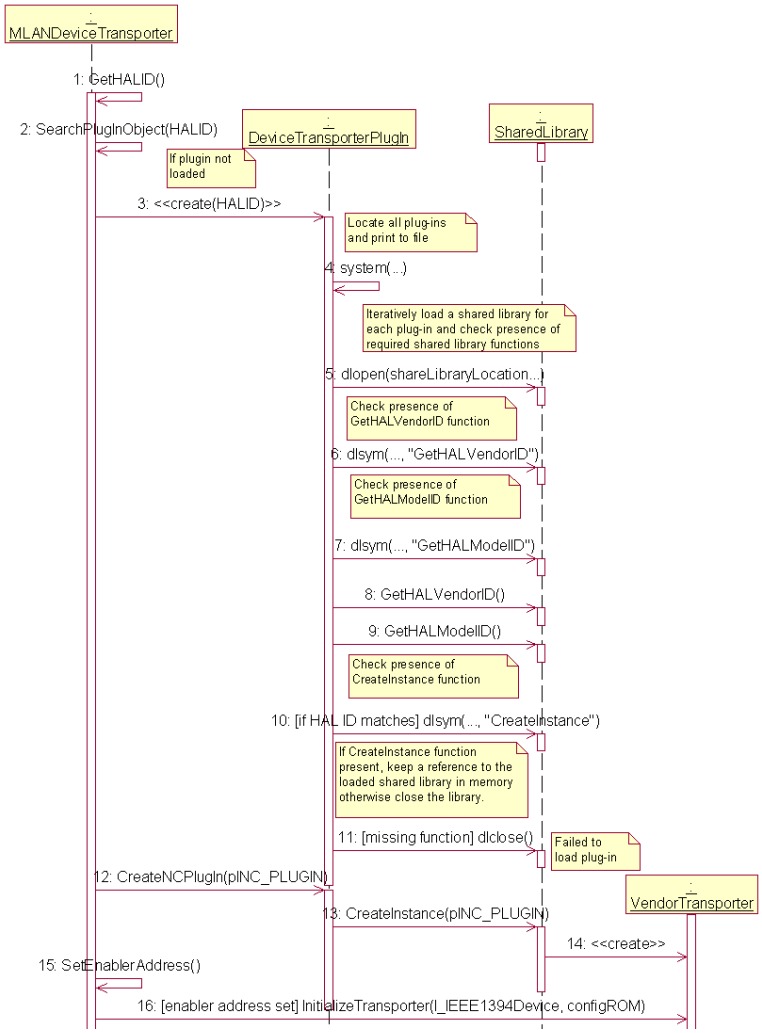


Figure 5.6: Proposed Plug-In Loading Mechanism on Linux

shared library is further queried for the address of the *CreateInstance* function. Upon retrieval of the address of the *CreateInstance* function, one can be certain that the plug-in has been successfully loaded and a handle to the shared library is kept in memory together with a pointer to the *CreateInstance* function. After a plug-in has been loaded, instances of the *VendorTransporter* "COM" components that correspond to the plug-in are created via the *CreateNCPlugin* function of the *DeviceTransporterPlugin* class. The *CreateNCPlugin* function is fulfilled in terms of the *CreateInstance* function of the shared library.

Unlike Windows, there is no conventional COM implementation for Linux. Guidelines described by Box [1998] were followed to provide a COM-like implementation for plug-ins created on Linux. In particular, a custom *IUnknown* "COM" interface was defined as shown below:

```
class IUnknown {
public:
    virtual void   AddRef() = 0;
    virtual UInt32 Release() = 0;
    virtual void*  QueryInterface(const char*
         interfaceTypeName) = 0;
};
```

Section 5.2 has already described the role of the *IUnknown* COM interface in reference counting and querying the existence of interfaces that may be implemented by a COM component. Our *INC_PLUGIN* "COM" interface derives from our custom *IUnknown* “COM” interface. To illustrate the concept, an *INC_PLUGIN* “COM” interface definition with only one member function is shown is shown below:

```
class INC_PLUGIN : public IUnknown {
public:
    virtual OSError Initialize(...) = 0;
           :
           :
    //More member functions omitted for brevity.
};
```

A common aspect to highlight in the above "COM" interface definitions is that they are abstract classes with pure virtual member functions. The onus is on the plug-in writer to provide implementations for each of the interface functions. Third party plug-in writers are typically presented with these "COM" interfaces. We show an example class definition for a *VendorTransporter* “COM” component (named *GenericVendorTransporter*) below:

```
class GenericVendorTransporter : public INC_PLUGIN,
                                 public INA_PLUGIN {
public:
    //IUnknown Functions
    void AddRef();
```

```
        UInt32 Release();
        void* QueryInterface(const char* interfaceTypeName);
        //INC_PLUGIN Functions
        ...
        //INA_PLUGIN Functions
        ...
    };
```

Implementations for the methods of our custom *IUnknown* "COM" interface have to be provided in the *GenericVendorTransporter* "COM" component's implementation. The *AddRef* and *Release* functions manage the *GenericVendorTransporter* "COM" component's lifetime in a manner that is recommended for all COM components. The *QueryInterface* function checks if the *GenericVendorTransporter* "COM" component implements a particular optional interface. In the *GenericVendorTransporter* example shown above, the *QueryInterface* function queries the existence of an implementation for the *INA_PLUGIN* "COM" interface and, if present, it returns a pointer to the interface. For completeness, we describe the implementations of the *CreateInstance*, *AddRef*, *Release*, and *QueryInterface* functions in the context of the *GenericVendorTransporter* example shown above.

The *CreateInstance* function is a global function of a shared library that represents a Transporter plug-in. It is the entry point to any interaction between the Enabler and Transporters. The definition of an example of the *CreateInstance* function's implementation is shown below:

```
    extern "C" INC_PLUGIN *CreateInstance() {


        INC_PLUGIN* pNCPluginPtr =  new GenericVendorTransporter();
        if(pNCPluginPtr)
           pNCPluginPtr->AddRef()
        return pNCPluginPtr;
    }
```

As shown above, the *CreateInstance* function creates an instance of the *GenericVendorTransporter* "COM" component. Each newly created instance is referred to via its *INC_PLUGIN* "COM" interface. The *INC_PLUGIN* "COM" interface is a mandatory interface that all Transporter plug-ins should implement. It is important to note that each *GenericVendorTransporter*

object has a variable, *m_cPtrs*, that keeps track of the number of pointers to the object. Therefore, the constructor of the *GenericVendorTransporter* class ensures that *m_cPtrs* is initialised to a value of zero, in addition to any other required initialisation. After a *GenericVendorTransporter* instance has been successfully created, the *AddRef* function, defined by our custom *IUnknown* interface, is called. The *AddRef* function simply increments the count that is kept by the *m_cPtrs* variable as shown below:

```
void GenericVendorTransporter::AddRef() {
    ++m_cPtrs;
}
```

Conversely, the *Release* function, also defined by our custom *IUnknown* interface, decrements the *m_cPtrs* variable. However, a special condition is taken into account. This condition is that when the value of *m_cPtrs* reaches the initial value of zero, it implies that *GenericVendorTransporter* object is no longer referenced by the Enabler, and hence the object may be deleted from memory. An implementation of the *Release* function is shown below.

```
UInt32 GenericVendorTransporter::Release() {
    if(--m_cPtrs == 0)
       delete this;
    return m_cPtrs;
}
```

In order to allow the Enabler to query whether a Transporter plug-in implements certain optional "COM" interfaces, the *QueryInterface* function of our *IUnknown* "COM" interface is used. An example of a *QueryInterface* implementation for our proposed plug-in loading mechanism is shown below:

```
void* GenericVendorTransporter::QueryInterface(
                        const char* interfaceTypeName){

    void * pvResult = 0;
    if(strcmp(interfaceTypeName, "INA_PLUGIN") == 0)
```

```
            pvResult = static_cast<INA_PLUGIN*>(this);
        else

            return 0; //Request for unsupported interface
        ((INC_PLUGIN*)pvResult->AddRef();
        return pvResult;
    }
```

As shown above, the textual name of the "COM" interface that is queried is passed as an argument, namely "*const char* interfaceTypeName*". If the interface is supported, the *AddRef* function is called and a *"void*"* pointer to the queried interface is returned.

Section 5.2 has already mentioned that a set of rules exist to govern the calling of *AddRef* and *Release* functions of the *IUnknown* COM interface. Our *AddRef* and *Release* functions are implemented in a manner that is recommended for COM reference counting. Note that our implementations of *AddRef*, *Release* and *QueryInterface* shown above were adapted from Box [1998] without any semantic modification.

Some of the common scenarios of when *AddRef* is called include [Box, 1998]:

1. When the Enabler writes a non-null interface pointer to a local variable.
2. When a Transporter plug-in function writes a non-null interface pointer to one of its parameters, where this parameter has been passed by reference.
3. When a Transporter plug-in returns a non-null interface pointer as a physical result of a function. An example of this is clearly illustrated by the *CreateInstance* and *QueryInterface* functions described above.
4. When the Enabler writes a non-null interface pointer to a data member of an object.

Similarly, some of the common scenarios of when *Release* is called include [Box, 1998]:

1. Prior to overwriting a non-null local variable or data member that contains an interface pointer within the Enabler.
2. Prior to leaving the scope of a non-null local variable that contains an interface pointer within the Enabler.

3. Prior to overwriting a non-null data member of an object that contains an interface pointer within the Enabler.
4. Prior to leaving the destructor of an object that has a non-null interface pointer as a data member within the Enabler.

The functions shown above are all that are required for the C++ "COM" implementation of our proposed plug-in loading mechanism on Linux. While this implementation is language dependent, C++ in our case, we benefit from the advantages of proper COM versioning and binary independence between the Enabler and Transporter plug-in implementations. The next section describes how our proposed plug-in mechanism was implemented on the Macintosh platform.

5.3.3 Proposed Macintosh Transporter Plug-In Loading Mechanism

Section 4.3.3 mentioned that while there is no Macintosh implementation for the current version of the Redesigned Enabler, Okai-Tettey [2005] suggests that a shared library approach, similar to the one used on Linux, could be used for plug-in loading on the Macintosh platform. The previous section, section 5.3.2, has described how the shared library approach on Linux was modified to operate in terms of our proposed plug-in loading mechanism, which is more representative of COM fundamentals. Our modified version of the Redesigned Enabler implementation was ported to the Macintosh platform. In this section, we describe an alternative that facilitates porting of Windows COM components directly to Mac OS X. We also give motivations for discarding this alternative despite its potential. A shared library approach, similar to that on Linux, was used for plug-loading on the Macintosh platform.

5.3.3.1 An Alternative to the Shared Library Approach: COM on Mac OS X

Before adopting a shared library approach for plug-in loading on the Macintosh platform, an alternative implementation, documented in “Component Object model (COM) Development on Mac OS X” [Hunt, 2004], was considered. In this implementation, Hunt [2004] suggests that it is possible to write COM components that have the potential to run on both Windows and Mac OS X with no code changes.

Hunt [2004] indicates that Mac OS X has a COM architecture. However, this COM architecture exists only to support Apple Corporation’s *Core Foundation Plugins* architecture. The *Core*

Foundation is a framework that provides fundamental software services useful to application services, application environments, and to applications themselves [Apple Computer, Inc., 2006a]. The *Core Foundations Plugins* architecture is a part of the *Core Foundation* framework that provides plug-in support for applications. In spite of this, Hunt [2004] has created a software development kit that allows development of COM components, using Apple Corporation's COM architecture, as is done on Windows. With this development kit, COM components written on Windows can potentially run under Mac OS X with no code changes, provided that certain constraints are adhered to.

Since COM under Mac OS X mainly exists to support the *Core Foundation Plugins* architecture, it can only be implemented using dynamic link libraries (DLL). In the context of the Windows Enabler/Transporter implementation, COM is implemented in terms of DLLs hence this constraint does not affect its implementation on Mac OS X. One significant difference between COM under Mac OS X and COM under Windows is the absence of a registry of COM components on Macintosh. On Macintosh, DLLs are manually loaded into memory. Recall from sections 4.3.1 and 5.3.1 that our Windows plug-in loading mechanism relies on the presence of a registry. Hunt [2004] suggests that the absence of a registry of COM components can be dealt with in one of two ways: either by fabricating a registry, or manually loading DLLs on both Windows and Macintosh, in order to allow for portability of code without any code changes.

5.3.3.2 Motivations for Remaining with the Shared Library Approach

The COM implementation for Mac OS X that has been described in section 5.3.3.1 above shows great potential with regards to easing the portability of code from Windows to Mac OS X. However, a decision was made to discard this alternative and rather implement plug-in loading on the Macintosh platform using shared libraries in a manner similar to the Linux plug-in loading mechanism described in section 5.3.2. A number of considerations influenced our decision, namely:

- Should anything go wrong with plug-in implementations, there is no liability. The implementation for COM on Mac OS X described above is based on a single individual's (Hunt [2004]) code, hence Apple Corporation is not obliged offer developer support.
- Any third party Transporter plug-in writers have to first understand Hunt's implementation. This may potentially complicate and/or lengthen the task of writing Transporter plug-ins.

- Fabricating a registry for COM components or manually loading DLLs might result in error prone implementations.
- Hunt [2004] describes the mechanism in the context of "Mac OS X". There is no documentation to show that the implementation works for other versions of the Macintosh operating system.
- The shared library approach described in section 5.3.2 is simple to implement and is based on standard shared library mechanisms.

5.3.3.3 The Shared Library Approach on Macintosh

Our proposed Redesigned Enabler implementation on Macintosh uses exactly the same "COM" mechanism as on Linux, as described in section 5.3.2. However, the mechanism for IEEE 1394 messaging on Macintosh workstations differs from that on Windows or Linux workstations. Therefore, platform-specific code regarding the Enabler's interaction with IEEE 1394 devices had to be written. In writing the platform-specific code, we followed guidelines that are documented in the "FireWire Device Interface Guide" [Apple Computer, Inc., 2006c], since, at the time of implementation, the Redesigned Enabler had not been implemented for the Macintosh platform, as mentioned in section 4.3.3 on page 70. We now give an indication of the modifications that were made to our proposed design of the Enabler in order to make it compatible with the IEEE 1394 messaging mechanism on Macintosh.

The object model shown in Figure 5.1 on page 92 accurately models our proposed design for an Enabler that runs on a Windows or Linux workstation. Of interest to us, the object model shows a class, named *IEEE1394Enabler*, which models the Enabler. The *IEEE1394Enabler* class contains an aggregation of objects of the *IEEE1394Interface* class. The *IEEE1394Interface* class models each IEEE 1394 interface card on the PC that runs the Enabler and is associated with the *OS1394Interface* class. The *OS1394Interface* class abstracts the platform-specific implementation that is used to interact with IEEE 1394 interface cards on the PC. The *Linux1394Interface* and *Windows1394Interface* classes represent platform-specific implementations for the *OS1394Interface* class on Linux and Windows, respectively. The *Linux1394Interface* class functionality is fulfilled through the Linux kernel's *raw1394* module which allows for read, write, and lock transactions [Linux 1394, 2006], while the *Windows1394Interface* class functionality is fulfilled through an IEEE 1394 bus driver which also enables read, write, and lock transactions [Okai-Tettey, 2005]. Note the absence of a platform-specific implementation for Macintosh in this model. This is

because IEEE 1394 messaging on Macintosh is done in a manner that is not compatible with the design that is shown in Figure 5.1.

Our proposed shared library mechanism was implemented on the Mac OS X version of the Macintosh operating system. In this version, IEEE 1394 messaging is done via in-kernel objects that belong to the IOFireWire family [Apple Computer, Inc., 2006c]. Figure 5.7 shows a stack of these objects. For each IEEE 1394 interface on the Macintosh, the IOFireWire family publishes

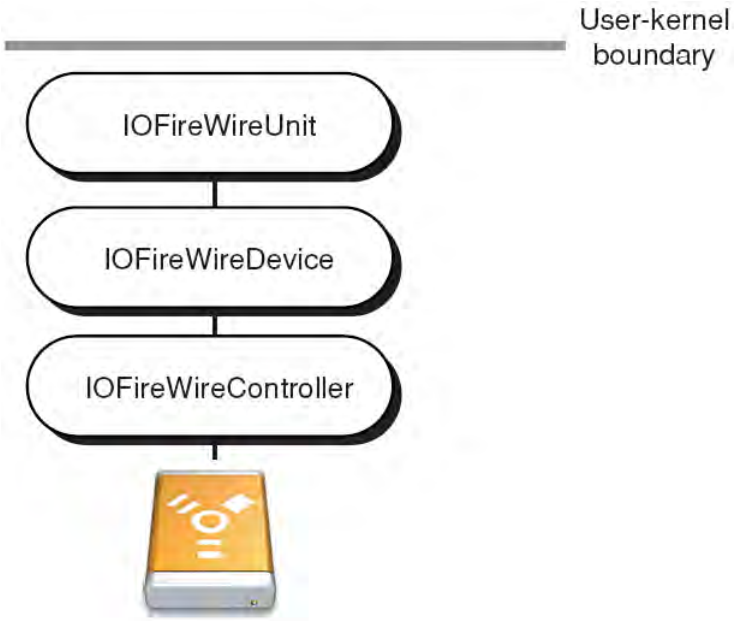


Figure 5.7: IEEE 1394 Device Stack for Mac OS [Apple Computer, Inc., 2006c]

an IOFireWireController object in the I/O Registry. The IOFireWireController object provides bus management services. For each device with a valid bus information block within its config ROM, the IOFireWire family publishes an IOFireWireDevice object in the I/O Registry. The IOFireWireDevice object keeps track of the device's node ID and certain information of its config ROM. For each unit directory found in the device's config ROM, the IOFireWire family publishes an IOFireWireUnit object in the I/O Registry. The IOFireWireUnit object contains information stored within the corresponding unit directory.

The IOFireWire family provides an IOFireWireLib device interface, which is used by the Enabler to access the I/O registry and to perform standard FireWire transactions. Based on the capabilities of this device interface, the Enabler object model was modified to the one shown in Figure 5.8. The important difference to note in this version of the Enabler's object model is that there is an *IEEE1394Enabler* class that is directly associated with an *IEEE1394Network* class. The

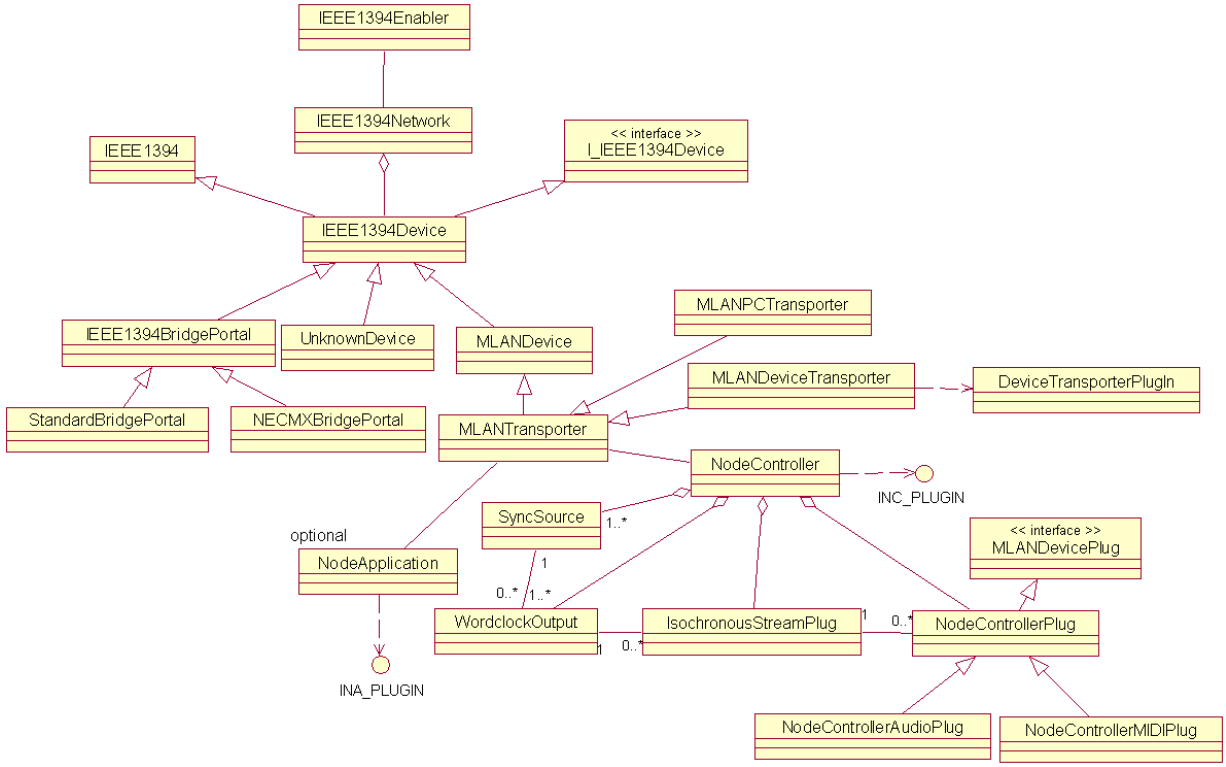


Figure 5.8: Proposed Redesigned Enabler Object Model for the Macintosh Platform

IEEE1394Network class contains an aggregation of objects of the *IEEE1394Device* class. In this model, we have removed the classes that model the FireWire interfaces on the Macintosh, since the *IEEE1394Network* class has direct access to the I/O Registry via the IOFireWireLib device interface. The IOFireWireLib device interface contains functionality that includes identifying new devices as they are added to the network, identifying devices as they are removed from the network, and functionality to carry out read, write, and lock transaction on a given IEEE 1394 device. As a result of direct access to the I/O Registry which only returns available devices, and our focus on a single bus environment, we have also removed the *IEEE1394Bus* class from the model. However, the rest of the model remains the same.

5.4 Chapter Summary

In this chapter, we have given a description of our proposal for a Transporter plug-in mechanism that attempts to fully utilise the capabilities of COM as a plug-in mechanism. These capabilities enable robust Transporter HAL API versioning which eliminates the need for an intermediary

function call that resolves each Transporter HAL API function call. In addition, we have also described how our proposed mechanism attempts to reduce the level of binary dependence between the Enabler and Transporter plug-ins. The chapter ended with a description of platform-specific implementations of our proposed Transporter plug-in mechanism, namely on Windows, Linux, and Macintosh operating systems. We have also shown the important changes to the Enabler model that were influenced by our Macintosh implementation.

Chapter 4 has described the shortcomings within two main research components of our investigation, namely the Transporter plug-in mechanism and the Transporter HAL API. We have shown how the shortcomings within the current Transporter plug-in mechanism were resolved. The next chapter, Chapter 6, focuses on how the shortcomings within the current Transporter HAL API were resolved. In particular, the chapter describes important aspects of our proposal for a new Transporter HAL API.

Chapter 6

Proposed Transporter HAL API

The previous chapter has described our proposed Transporter plug-in mechanism. In this chapter, we describe how the shortcomings raised in Chapter 4 with regards to the Transporter HAL API were resolved. The Transporter HAL API, the second main component of our investigation, defines an interface that governs the interaction between the Enabler and Transporters from various vendors, in a hardware independent manner.

We describe our proposal for a new Transporter HAL API that fully utilises all capabilities of the Open Generic Transporter (OGT). We acknowledge that the OGT provides a high level abstraction of the functions of the A/M Data Transmission Protocol. Consequently, our proposed Transporter HAL API is biased towards the OGT. In support of our proposed Transporter HAL API we describe the important concepts of three HAL implementations that demonstrate interoperability between OGT-based and non-OGT-based Transporters. We also give an overview of a test application that was created to manually test each of the Transporter HAL API functions.

6.1 Proposed Transporter HAL API Implementations

Our proposed Transporter HAL API defines methods that are necessary to access the node controller capabilities of Transporters, including the additional node controller capabilities that have been revealed by the OGT guideline document. These methods are defined in the *INC_PLUGIN* COM interface that has been mentioned in section 5.2 on page 93. At the time of this writing, the OGT document did not provide guidelines for node application capabilities. Hence, our proposed Transporter HAL API does not currently provide methods to access the node application

capabilities of Transporters, although section 5.2 on page 93 has mentioned that such capabilities may be accessed via methods defined in an *INA_PLUGIN* COM interface of the Transporter HAL API.

We acknowledge that the OGT concept provides a high level abstraction that completely encapsulates the underlying workings of the A/M Data Transmission Protocol. Consequently, our proposed Transporter HAL API is biased towards the OGT guidelines. In order to resolve the shortcomings raised in section 4.4.2 on page 78, our proposed Transporter HAL API design is based on the ISP-NCP concept, provides explicit Plug Layout switching capabilities to handle device-specific quirks, supports multiple word clock outputs, and allows for word clock source selection for each word clock output [Chigwamba and Foss, 2007].

Three proof-of-concept Transporter HAL implementations were created based on our proposed Transporter HAL API, namely:

1. An OGT plug-in implementation for Transporters that comply with the guidelines laid out in the OGT document. This plug-in was intended to demonstrate full utilisation of the OGT capabilities that are not reflected by the current Transporter HAL API.
2. A plug-in implementation for the MAP4 Evaluation board (described in section 4.4.1 on page 72). This plug-in implementation demonstrated backwards compatibility of our proposed Transporter HAL API when implemented in terms of an existing hardware architecture that is fully compatible with the current Transporter HAL API.
3. A PC Transporter implementation that demonstrated how backwards compatibility can be achieved by fulfilling our proposed Transporter HAL API function calls in terms of the current Transporter HAL API function calls. The need for such an implementation resulted from the tight coupling of the mLAN Streaming Driver to the current Transporter HAL API. Therefore, a mechanism was implemented to map our proposed Transporter HAL API functionality to the current Transporter HAL API functionality.

In this section we describe important concepts of our proposed *INC_PLUGIN* COM interface in the context of the three proof-of-concept HAL implementations mentioned above. This COM interface defines the mandatory node controller functions that are required to control any Transporter in a vendor independent manner. A comprehensive description of the *INC_PLUGIN* COM interface can be found in Appendix B on page 217.

6.1.1 Open Generic Transporter Plug-In Design and Implementation

Figure 6.1 shows an object model that accurately models the node architecture of the node controller block of the generic Transporter as described in the OGT guideline document [Audio Engineering Society - Standards Committee, 2005]. Recall from section 3.3.3.1 on page 49 that the

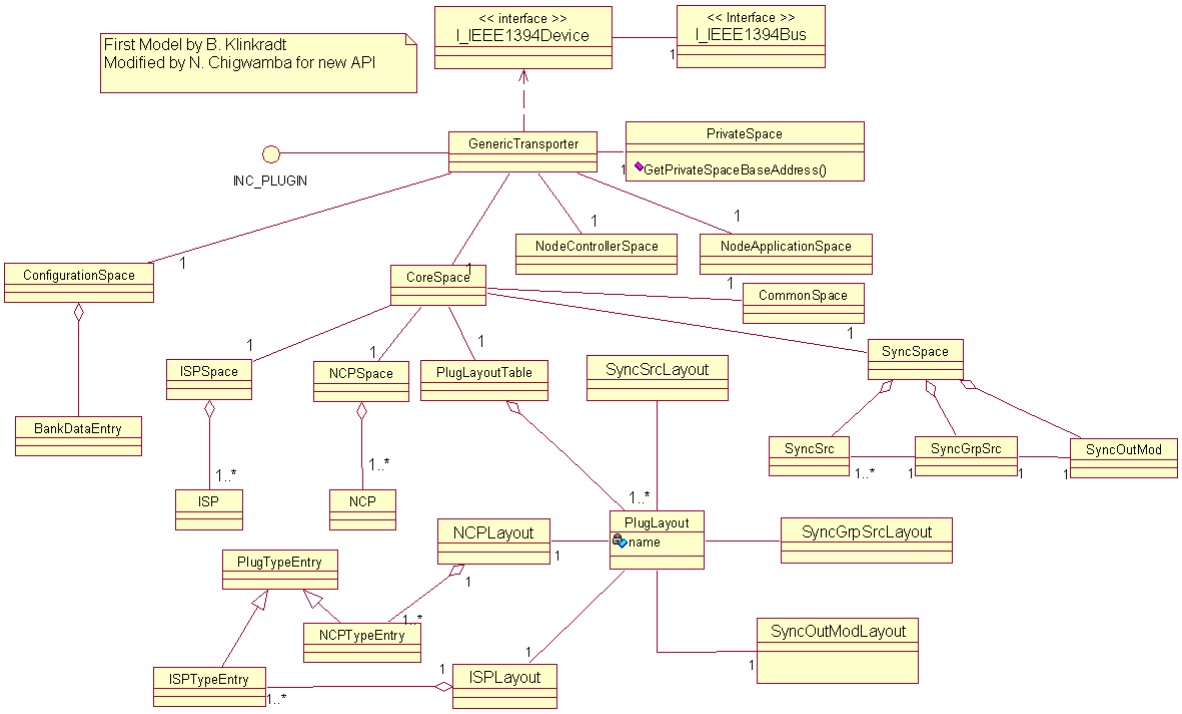


Figure 6.1: Open Generic Transporter HAL Plug-In Object Model

node controller represents the IEEE 1394 node that is hosted by a device. As shown in the model, the *INC_PLUGIN*, *GenericTransporter*, *I_IEEE1394Device*, and *I_IEEE1394Bus* components represent our proposed HAL design, as described in section 5.2 on page 93. The node controller block of the generic Transporter is mapped into four main regions, namely the Configuration Parameter Space, Core Space, Node Controller Space, and Node Application Space. Figure 6.1 shows these regions as the classes *ConfigurationSpace*, *CoreSpace*, *NodeControllerSpace*, and *NodeApplicationSpace*, respectively. The *PrivateSpace* class models a Transporter Control Interface register map that specifies address offsets for the control registers of the four main regions. We now describe the functions of three of the four regions mentioned above and how these are accessed via our proposed Transporter HAL API. Recall from section 4.4.2 on page 78 that at the time of this writing the OGT document did not specify any guidelines for the node application block of a generic Transporter, and thus we do not describe the functions of the Node Application

Space.

6.1.1.1 Configuration Parameter Space

The Configuration Parameter Space is a storage area for the non-volatile parameters of a Transporter [Audio Engineering Society - Standards Committee, 2005]. The space comprises a Configuration Bank area to store device configurations, which are restored when a device is turned on. The Configuration Bank area is represented by an aggregation of *BankDataEntry* class objects in Figure 6.1. Our proposed Transporter HAL API does not expose any components of the Configuration Parameter Space. However, there are certain registers that store A/M Data Transmission Protocol parameters for a Transporter. These registers are accessed via the Transporter HAL API and are also associated with registers within the Configuration Parameter Space. Consequently, modification of such A/M protocol parameter registers results in updates of the associated Configuration Parameter Space registers.

6.1.1.2 Node Controller Space

Registers that relate to the overall operation of a Transporter are contained within the Node Controller Space [Audio Engineering Society - Standards Committee, 2005]. These registers include the Enabler address, Transporter mode[1], an identifier for the Plug Layout that is currently in use, and name entries such as the Transporter nickname and firmware version name. We highlight a new concept that we introduced to the Transporter HAL API, namely the Plug Layout concept. Plug Layouts are mutually exclusive Transporter configurations that indicate different Transporter capabilities. As mentioned in section 4.4.2.5 on page 88, we identified bandwidth, in addition to audio sampling frequencies, as another aspect that could be modelled by Plug Layouts. This presented a further motivation for the need to change Plug Layouts explicitly via the Transporter HAL API. The current Transporter HAL API does not take Plug Layouts into account and therefore, it is impossible for the Enabler to explicitly switch Plug Layouts. In order to allow the Enabler to have control over Plug Layout switching, our proposed Transporter HAL API contains the following functions in the *INC_PLUGIN* COM interface definition:

$$GetCurrentPlugLayout(*plugLayoutID) \tag{6.1}$$

$$SetCurrentPlugLayout(plugLayoutID) \tag{6.2}$$

[1] Specifies the operation mode of the Transporter Restoration block described in section 3.3.3.1 on page 49

In the context of the generic Transporter, the *GetCurrentPlugLayout* function (function (6.1)) returns the identifier of the Plug Layout that is currently in use, while the *SetCurrentPlugLayout* function (function (6.2)) specifies the identifier of the Plug Layout to be used.

6.1.1.3 Core Space

Transporter control registers for the A/M Transport Block of the generic Transporter are implemented by the Core Space [Audio Engineering Society - Standards Committee, 2005]. These registers are further grouped into five sub-spaces, namely the Plug Layout Table, Isochronous Stream Plug Space (ISP Space), Node Controller Plug Space (NCP Space), Sync Space, and Common Space. Figure 6.1 models these sub-spaces as *PlugLayoutTable, ISPSpace*, *NCPSpace*, *SyncSpace*, and *CommonSpace* classes, respectively.

PLUG LAYOUT TABLE

As mentioned in section 6.1.1.2 above, our proposed Transporter HAL API now allows for explicit Plug Layout switching. The OGT guideline document defines a Plug Layout Table as a read only area which provides the Enabler with information about the types of plugs and word clocks that a device implements, and the number of plugs of each type that are supported by the device [Audio Engineering Society - Standards Committee, 2005]. The object model in Figure 6.1 shows the Plug Layout Table design clearly. In particular, the Plug Layout Table comprises a number of Plug Layouts. Each Plug Layout contains an entry for the Plug Layout name, ISP Layout, NCP Layout, Sync Source Layout, Sync Group Source Layout, and a Sync Output Module Layout. In the object model these components, except for the Plug Layout name which is an attribute of the *PlugLayout* class, are shown as *ISPLayout*, *NCPLayout*, *SyncSrcLayout*, *SyncGrpSrcLayout*, and *SyncOutModLayout* classes, respectively. A brief description of each of these follows.

Recall from section 3.3.3.2 on page 50 that the generic Transporter introduces the concept of Isochronous Stream Plugs (ISPs) and Node Controller Plugs (NCPs), where an ISP can be statically or dynamically associated with number of NCPs. An ISP represents an input or output for a single isochronous stream to or from the IEEE 1394 bus. An NCP represents an input or output for a single monaural channel of audio or a single cable of MIDI to or from the node application block, and corresponds to a terminator of a sequence (sequence end-point). The ISP Layout declares all ISP types (shown as the *ISPTypeEntry* class in Figure 6.1) available in a given Plug Layout. Plugs with different capabilities or restrictions are declared as different plug types. The NCP Layout declares all NCP types (shown as the *NCPTypeEntry* class in Figure 6.1) available

in a given Plug Layout. The Sync Source Layout, Sync Group Source Layout, and Sync Output Module Layout represent the Sync Block of the A/M Transport Block of the generic Transporter. The Sync Block supplies word clocks "necessary for driving ISPs and NCPs" [Audio Engineering Society - Standards Committee, 2005].

ISP AND NCP SPACES

The ISP Space of the generic Transporter implements the control registers for ISPs, while NCP Space implements registers for NCPs. As mentioned in section 4.4.2.1 on page 78 the current Transporter HAL API does not take into account the possibility of dynamic associations between ISPs and NCPs, hence *virtual isochronous streams* were introduced as shown in Figure 4.15 on page 83. In order to resolve this, our proposed Transporter HAL API was defined in terms of ISPs and NCPs. While this design decision was influenced by the OGT document, it is acknowledged that the document provides a hardware independent high level abstraction of the A/M Data Transmission Protocol. Sections 6.1.2 and 6.1.3 describes two proof-of-concept implementations that were created to demonstrate backwards compatibility of our proposed Transporter HAL API. Functions have been defined in the *INC_PLUGIN* COM interface to query the number of ISPs/NCPs and to retrieve lists of identifiers[2] of ISPs/NCPs as shown below:

$$GetNumISPs(plugLayoutID,\ isInput,\ *pNumISPs) \quad (6.3)$$

$$GetISPs(plugLayoutID,\ isInput,\ listSize,\ *numISPsFound,\ *ispIDList) \quad (6.4)$$

$$GetNumNCPs(plugLayoutID,\ isInput,\ ncpType,\ *pNumNCPs) \quad (6.5)$$

$$GetNCPs(plugLayoutID,\ isInput,\ ncpType,\ listSize,\ *numNCPs,\ *ncpIDList) \quad (6.6)$$

With regards to ISPs, the Enabler calls the *GetNumISPs* function (function (6.3)) to retrieve the total number of ISPs, **pNumISPs*, of a direction specified by *isInput* in a Plug Layout that is identified by the *plugLayoutID* identifier. Under normal circumstances, *plugLayoutID* refers to the ID of the Plug Layout configuration that a device is using at a given time. Armed with the knowledge of the number of ISPs (**pNumISPs*) to expect in a given Plug Layout, the Enabler calls the *GetISPs* function (function (6.4)) to retrieve a list of unique ISP identifiers (ISP IDs), **ispIDList*, for all ISPs of a direction specified by *isInput* in a Plug Layout that is identified by *plugLayoutID*. Each of the ISP IDs retrieved is then used by the Enabler to build its object model of the Transporter. In particular, each ISP ID results in the creation of an instance of the *IsochronousStreamPlug* class, as shown in the object model in Figure 5.1 on page 92.

[2]Unique indexes relative to the OGT register set

The retrieval of NCP IDs is similar to that for ISP IDs. However, in addition to direction, NCPs are further distinguished by the type of sequences they terminate, namely audio or MIDI. In particular, the Enabler calls the *GetNumNCPs* function (function (6.5)) to retrieve the total number of NCPs, **pNumNCPs,* of a given type and direction. Based on the number of NCPs (**pNumNCPs*) retrieved, the Enabler calls the *GetNCPs* function (function (6.6)) to retrieve a list of unique NCP identifiers (NCP IDs), **ncpIDList.* These NCP IDs are used to build the Enabler's object model. In particular, for each retrieved NCP ID, the Enabler creates an instance of the *NodeControllerPlug* class, as shown in the object model in Figure 5.1 on page 92. For NCP IDs that correspond to MIDI NCPs, the instance created is of the *NodeControllerMIDIPlug* class, while for NCP IDs that correspond to audio NCPs, the instance created is of the *NodeControllerAudioPlug* class. The *NodeControllerAudioPlug* and *NodeControllerMIDIPlug* classes both derive from the *NodeControllerPlug* class as shown in Figure 5.1 on page 92.

Making Dynamic ISP-to-NCP Associations

The ISP-NCP model was incorporated in the Transporter HAL API in order remove the *software mapping structure* that led to the concept of *virtual isochronous streams*, as shown in Figure 4.15 on page 83. The *GetNumISPs*, *GetISPs*, *GetNumNCPs*, and *GetNCPs* functions (functions (6.3), (6.4), (6.5), and (6.6) on page 120) were introduced to query the number of ISPs/NCPs and their corresponding unique identifiers.

With reference to Figure 3.17 on page 50, ISPs interface directly with the IEEE 1394 bus and receive or transmit isochronous streams that comprise audio or MIDI sequences. Recall the distinction between *input* and *output* in the context of isochronous streams, as described in section 4.4.2.1 on page 78. *Input* isochronous streams are received by the Transporter from the IEEE 1394 bus via *input ISPs*. On the other hand, *output* isochronous streams are transmitted onto the IEEE 1394 bus from the Transporter via *output ISPs*. *Input ISPs* may be statically or dynamically associated with one or more *input NCPs*. Each *input NCP* is configured to receive an audio sequence or MIDI subsequence of a received isochronous stream, which can be played out via the Transporter's hardware plugs. Conversely, *output NCPs* receive audio samples and MIDI messages from a host Transporter's hardware plugs. One or more *output NCPs* may be dynamically or statically associated with an *output ISP* for transmission of the data onto the IEEE 1394 bus. Each of the associated NCPs is configured to pack audio samples or MIDI messages at a particular position within each isochronous packet that gets sent onto the bus. This position corresponds to the sequence or subsequence number.

We focused on alleviating the problems resulting from the possibility of dynamic associations

between ISPs and NCPs. In doing so, our proposed Transporter HAL API defines the following functions in the *INC_PLUGIN* COM interface:

$$GetAttachedISP(ncpID, *ispID, ...) \tag{6.7}$$

$$SetAttachedISP(ncpID, ispID) \tag{6.8}$$

$$Attach(ncpID) \tag{6.9}$$

$$Detach(ncpID) \tag{6.10}$$

The *GetAttachedISP* function (function (6.7)) retrieves the ID, **ispID*, of the ISP that an NCP, identified by *ncpID*, is associated with. Some function arguments have been omitted for conceptual clarity and these are indicated by the ellipsis. Conversely, the *SetAttachedISP* function (function (6.8)) is used to dynamically associate an NCP, identified by *ncpID*, with an ISP that is identified by *ispID*. The *Attach* function (function (6.9)) is used to commence data transfer between an NCP and its associated ISP, while the *Detach* function (function (6.10)) stops the data transfer between an NCP and its associated ISP.

A consequence of incorporating the ISP-NCP model into the Transporter HAL API is the elimination of the additional complexity introduced by the concept of *virtual isochronous streams* shown in Figure 4.15 on page 83. Figure 6.2 shows how our proposed Transporter HAL API has removed this additional complexity in the OGT HAL plug-in implementation.

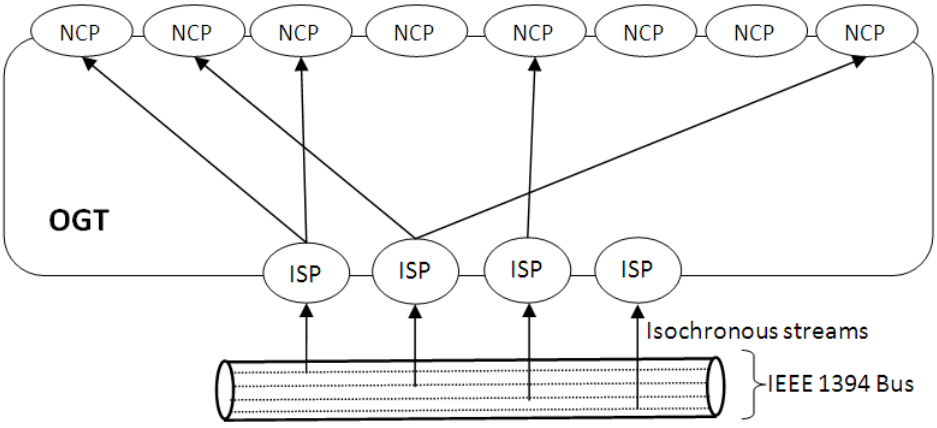


Figure 6.2: Proposed OGT Plug-In Implementation Without Mapping Structure for Inputs

This diagram shows *input ISPs* and *input NCPs*, where the *input ISPs* receive unique isochronous streams from the IEEE 1394 bus. These *input ISPs* are associated with *input NCPs*, where each NCP extracts audio/MIDI data at a particular position (sequence/subsequence) within each

isochronous packet cluster of a received isochronous stream. The audio/MIDI extracted by the NCPs is played out on the Transporter's host device plugs, although not shown in the diagram.

Figure 6.2 above shows the case for *input ISP* and *input NCP* association. The case for *output ISP* and *output NCP* associations is shown in Figure 6.3. The *output NCPs* shown in the diagram

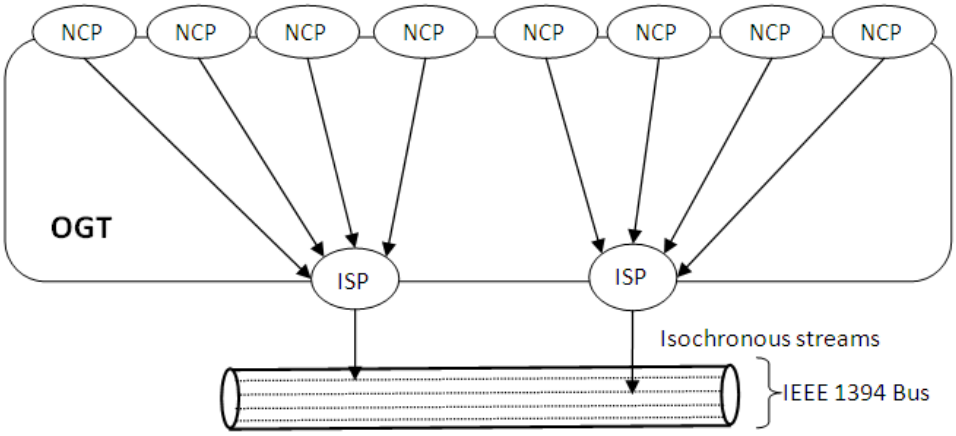


Figure 6.3: Proposed OGT Plug-In Implementation without Mapping Structure for Outputs

typically receive audio channels or MIDI messages from the Transporter's host device plugs. These audio channels or MIDI messages are packed at particular positions within the isochronous packet clusters of the isochronous streams that are transmitted by the *output ISPs* onto the IEEE 1394 bus. It is common for Transporters to have fewer *output ISPs* than *input ISPs* as shown by contrasting Figures 6.2 and 6.3. This due to the fact that *input ISPs* receive isochronous streams that are simultaneously transmitted by more than one other transmitter, while *output ISPs* only transmit isochronous streams from the host Transporter. Each ISP may only receive or transmit one isochronous stream. Therefore, in order to concurrently receive sequences from multiple isochronous streams, a Transporter requires more *input ISPs*.

Starting and Stopping of Transmission or Reception

In addition to eliminating complexities due to the *software mapping structure* and *virtual isochronous streams*, as described above, our proposed Transporter HAL API allows for individual control over ISPs. This individual control has resulted in, for example, functions previously defined to collectively start and stop transmission or reception being modified to allow for individual control over ISPs, where possible. The *Start* and *Stop* functions (functions (4.6) and (4.7) on page 85) described in section 4.4.2.2 have been modified to allow for individual control over ISPs, as

shown below:

$$Start(ispID) \tag{6.11}$$

$$Stop(ispID) \tag{6.12}$$

In particular, our *Start* function (function (6.11)) commences transmission or reception of isochronous streams with a given channel number for an ISP that is identified by *ispID*. On the other hand, our *Stop* function (function (6.12)) stops transmission or reception of isochronous streams with a given channel number for an ISP that is identified by *ispID*. Section 4.4.2.2 on page 85 described how collective control over *output ISPs* may result in bandwidth wastage. We have eliminated this, where possible, by allowing individual control over independent ISPs. This means that it is possible to start a single *output ISP* from a group of *output ISPs*, and hence only the required isochronous resources are allocated. Although there are no bandwidth problems resulting from collective control over input ISPs, our proposed Transporter HAL API model also means that *input ISPs* can now be controlled individually, where possible.

SYNC SPACE

The object model in Figure 6.1 on page 117 shows the Sync Space modelled as the *SyncSpace* class. This *SyncSpace* class is associated with other classes that model the components of the Sync Block of a generic Transporter, namely the *SyncSrc*, *SyncGrpSrc*, and *SyncOutMod* classes. The Sync Space of the generic Transporter contains control registers for the Sync Block. In particular, the Sync Block comprises a pool of *Sync Sources*, *Sync Group Sources* and *Sync Output Modules*, as shown in Figure 6.4 [Audio Engineering Society - Standards Committee, 2005]. These supply word clocks to the Plug Block. Recall from Figure 3.17 on page 50 that the Plug Block mainly consists of ISPs and NCPs. The word clocks supplied to the Plug Block control the rate of analog-to-digital and digital-to-analog conversion.

A Sync Source represents any functional unit that is capable of producing or supplying word clocks to the Plug Block. The complete collection of all Sync Sources makes up a device's pool of word clock sources. Three main types of word clock sources are:

- SYT regeneration circuits. This type word clock source receives SYT timing information from the IEEE 1394 bus and regenerates the word clocks using techniques such as the Phase Locked Loop (PLL).
- External word clock inputs. This type of word clock source originates from a source, other than the IEEE 1394 bus, that is external to a device, such as the word clock information

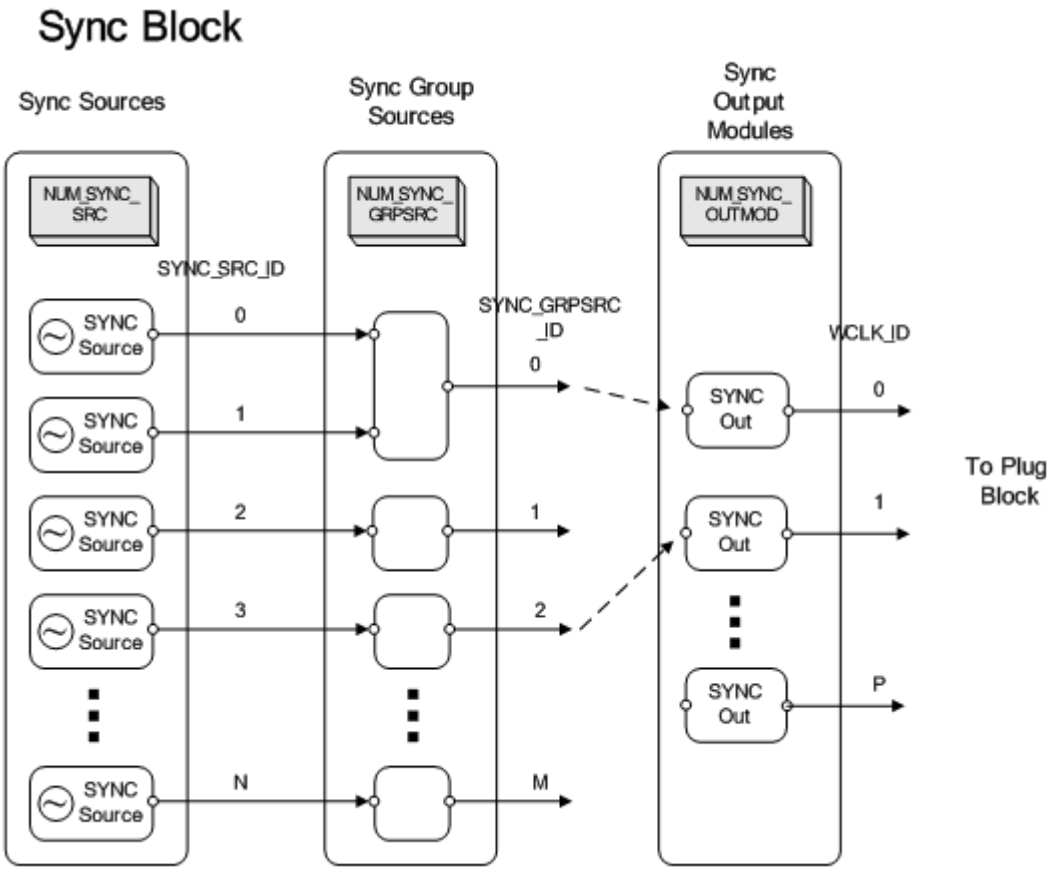


Figure 6.4: Open Generic Transporter Sync Block [Audio Engineering Society - Standards Committee, 2005]

from an ADAT digital audio signal. Recall from section 4.4.2.3 on page 86 that the current Transporter HAL API does not allow for selection of these external word clock sources.

- Internal clock generation circuits such as a device's built-in clock.

Sync Group Sources represent a device's concurrently usable word clocks, as shown in Figure 6.4. In particular, they model a group of Sync Sources where only one source may be active at a time. Sync Output Modules represent the word clock inputs to the Plug Block. In the context of the Sync Block, the Sync Output Modules generate word clock outputs. In the context of the Plug Block, they are word clock inputs. The number of Sync Output Modules is equal to the maximum number of concurrent word clock domains of the Plug Block, which is typically equal to the number of word clock inputs of the packet handling hardware. Recall from section 4.4.2.4 that at the time of this writing there were no firmware implementations for the Open Generic

Transporter that implemented more than one concurrently usable work clock. Notwithstanding, we designed the Transporter HAL API in a manner that takes into account the future need for more than one concurrently usable word clock.

Incorporating Multiple Word Clock Outputs and External Word Clock Source Selection into the Transporter HAL API

As mentioned above, our proposed Transporter HAL API is designed in a manner that takes into account the future need for more than one word clock output (concurrently usable word clock). In particular, the *INC_PLUGIN* COM interface defines the following function:

$$GetNumWordclockOutputs(pluglayoutID, *numWCLKOutputs) \tag{6.13}$$

The *GetNumWordclockOutputs* function (function (6.13)) returns the number of word clock outputs, ***numWCLKOutputs***, present in a given Plug Layout (identified by *plugLayoutID*). Each of the work clock outputs present can be referenced via a unique identifier, which we refer to as *wclkID*. Note that the *wclkID* is unique within the context of a specified Plug Layout and the valid values are represented by the relation $0 \leq wclkID \leq *numWCLKOutputs$, where ***numWCLKOutputs** is the value returned by the *GetNumWordclockOutputs* function.

Each of the word clock outputs mentioned above ultimately receives its word clock from a word clock source within a device's pool of word clock sources. The current Transporter HAL API only takes into account the capability of selecting internal and SYT word clock sources via the *GetSYTSyncChannel* and *SetSYTSyncChannel* functions (function (4.8) and (4.9) on page 86) described in section 4.4.2.3. We have shown how the OGT guideline document defines external word clock inputs as selectable word clock sources. This capability is lacking in the current Transporter HAL API. We resolved this by introducing the following functions to the *INC_PLUGIN* COM interface:

$$GetNumClockSources(pluglayoutID, *numClockSources) \tag{6.14}$$

$$GetCurrentClockSource(pluglayoutID, wclkID, *clockID) \tag{6.15}$$

$$SetCurrentClockSource(pluglayoutID, wclkID, clockID) \tag{6.16}$$

The *GetNumClockSources* function (function (6.14)) retrieves the number of word clock sources, ***numClockSources**, present in a Plug Layout identified by *plugLayoutID*. Each of the word clock sources is referenced via a unique identifier, which we refer to as *clockID*. Similar to the *wclkID*, the *clockID* is unique within the context of a given Plug Layout. The range of valid values for

the *clockID* is defined by the relation $0 \leq clockID \leq *numClockSources$, where ***numClockSources*** is the value returned by the *GetNumClockSources* function. The manner in which we identify word clock sources, as shown above, does not restrict the Enabler to any particular types of word clock sources, hence external word clock sources are also incorporated.

For a Plug Layout identified by *pluglayoutID*, the *GetCurrentClockSource* function (function (6.15)) is used to query the ID, ***clockID*, of word clock source that is supplying word clocks to a given word clock output that is identified by *wclkID*. Conversely, the *SetCurrentClockSource* function (function (6.16)) is used to specify the ID of the word clock source that should supply word clock to a word clock output that is identified by *wclkID*.

Figure 6.4 shows how word clock outputs feed into the Plug Block. For completeness, we mention that the following functions have been defined in the *INC_PLUGIN* COM interface in order to bridge between the Sync Block and Plug Block:

$$GetISPWCLKID(ispID, *wclkID, ...) \tag{6.17}$$

$$SetISPWCLKID(ispID, wclkID) \tag{6.18}$$

Within the Plug Block of the Open Generic Transporter, each ISP has a *wclkID* attribute that identifies the word clock output from which it is receiving word clocks. The *GetISPWCLKID* function (function (6.17)) is used to query the *wclkID* of the word clock output that supplies word clocks to an ISP that is identified by *ispID*. This function has additional attributes that have been omitted here for conceptual simplicity. Conversely, the *SetISPWCLKID* function (function (6.18)) specifies the *wclkID* of the word clock output that an ISP, identified by *ispID*, should receive word clocks from.

COMMON SPACE

The Common Space of the Open Generic Transporter implements control registers that apply globally to the A/M Transport Block. Such registers include those to query and specify the transmission modes, *blocking* and *non-blocking* as described in section 3.2.1 on page 37, that a Transporter can operate in. Such functionality is catered for in the current Transporter HAL API. Our proposed Transporter HAL API continues to implement such functionality, and hence we do not explore this functionality further.

6.1.2 MAP4 Transporter Plug-In Design and Implementation

The previous section focused on the various aspects of the Open Generic Transporter and how they influenced the design of our proposed Transporter HAL API. In acknowledgement of the hardware independent high level abstraction offered by the OGT guidelines, it became apparent that our proposed Transporter HAL API was biased towards the OGT. In this section, we show how this bias towards the OGT does not preclude us from writing Transporter plug-ins for Transporters that are based on other hardware architectures.

We describe the Transporter plug-in design and implementation for Yamaha Corporation's MAP4 Evaluation board. This plug-in was created against our proposed Transporter HAL API in order to demonstrate backwards compatibility with existing hardware architectures [Chigwamba and Foss, 2007]. Our implementation is based on an understanding of the hardware capabilities of the MAP4 Evaluation board. It should be borne in mind that the implementation we describe here is merely one way of achieving backwards compatibility. Section 6.1.3 describes another way of achieving backwards compatibility by mapping methods of our proposed Transporter HAL API to methods of the current Transporter HAL API. As mentioned by Chigwamba and Foss [2007] in "Enhancing End-User Capabilities in High Speed Audio Networks" any Tranporter previously controllable by the Enabler requires modifications to its HAL plug-in before it can successfully use our proposed Transporter HAL API.

Figure 6.5 shows the object model for our proposed plug-in implementation for the MAP4 Transporter. This object model resulted from the redesign the object model for the current MAP4 Transporter plug-in which is based on the current Transporter HAL API. Similar to the OGT plug-in implementation, described in section 6.1.1, the *INC_PLUGIN*, *MAP4Transporter*, *I_IEEE1394Device* and *I_IEEE1394Bus* components shown in the object model represent our proposed HAL design, which all our HAL plug-ins comply with. This HAL design was described in section 5.2 on page 93. In the object model, the classes *PrivateSpace, CoreSpace, MLANSpace*, and *BootParameterSpace* represent the node architecture for the MAP4 Transporter. The roles of these spaces were described in section 4.4.1 on page 72. The same section highlights that the MAP4 Evaluation board houses one mLAN-NC1 chip and one mLAN-PH2 chip. These chips are modelled by the *NC1* and *PH2* classes in the object model. There are some classes with names containing the "*Dump*" substring, namely *NC1DumpNC1*, *NC1DumpPH2*, *NC1DumpMaster*, *NC1DumpEnd*, and *NC1DumpMLAN*. These classes are responsible for storing device configurations to the non-volatile memory of the Transporter, and are restored when the device is turned on. An important addition to this model is the introduction of the *NCP*, *ISP,*

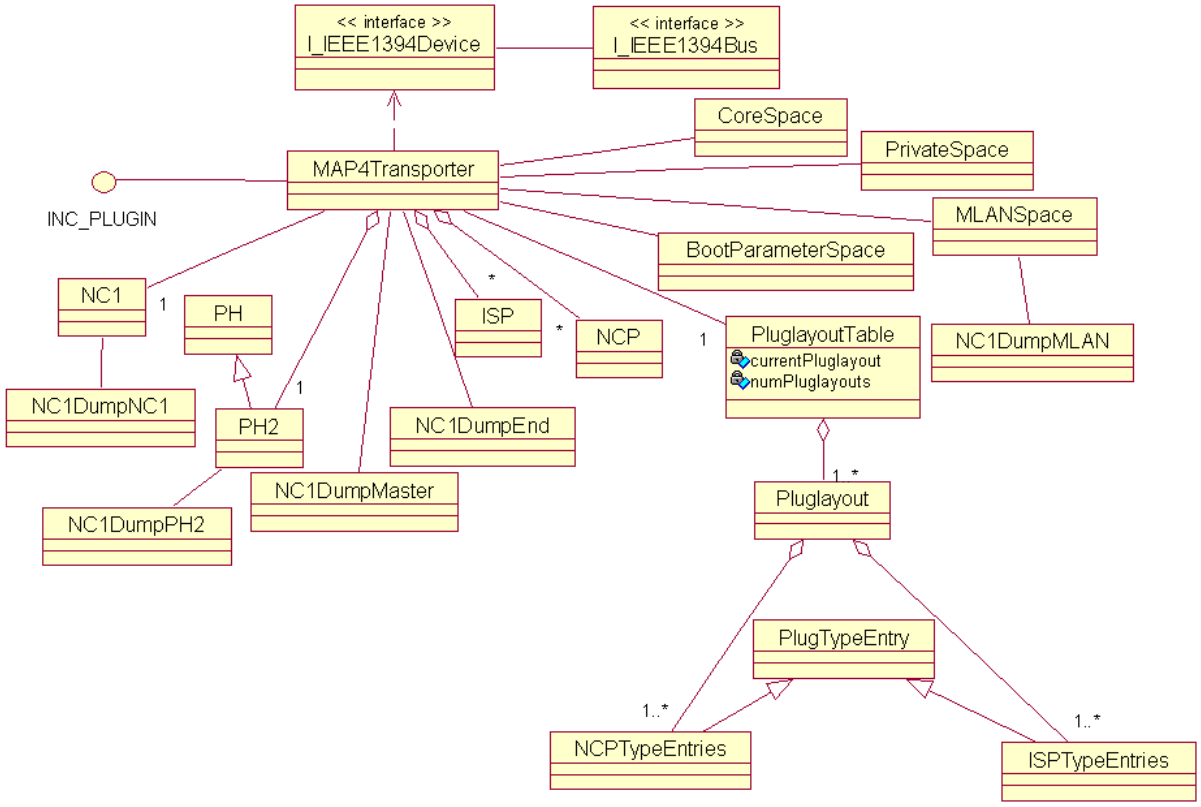


Figure 6.5: MAP4 Transporter HAL Plug-In Object Model

PlugLayoutTable, *PlugLayout*, *PlugTypeEntry*, *NCPTypeEntries*, and *ISPTypeEntries* classes. These were introduced in order to handle functions of our proposed Transporter HAL API.

We now focus on the most important aspects of our MAP4 Transporter plug-in implementation in the context of our proposed Transporter HAL API. In particular, we focus on how Plug Layouts were defined, how ISPs and NCPs were modelled, and synchronisation with regards to word clock outputs and word clock source selection [Chigwamba and Foss, 2007].

6.1.2.1 Defining Plug Layouts

The Open Generic Transporter introduced the concept of Plug Layouts. Recall from section 4.4.2.5 on page 88 that Plug Layouts represent mutually exclusive Transporter configurations that a device may have. Each of these configurations defines a number of plugs (NCPs and ISPs) and word clocks that are usable concurrently [Audio Engineering Society - Standards Committee, 2005]. The number of usable plugs in a given Plug Layout is fixed. The OGT guideline document stipulates that, should the number of usable plugs differ depending on sampling frequency

or data format, a device would have to implement separate "modes", where each mode has a separate Plug Layout.

The number of audio channels that can be transmitted or received by the MAP4 Transporter at sampling frequencies greater than 48 kHz is half the number that can be transmitted at 48 kHz and lower sampling frequencies [Yamaha Corporation, 2002b]. This is due to the fact that at high sampling frequencies (above 48 kHz), more data that is transmitted or received, while there are restrictions in the data processing capabilities of the MAP4 Transporter's chips at high sample rates. This presented the need to expose two Plug Layouts, one to represent audio channels that are usable at 48 kHz and lower sampling frequencies, and another for sampling frequencies greater than 48 kHz. It should be borne in mind that, at any point, the word clocks available and the number of MIDI channels that can be transmitted is constant regardless of the sampling frequency. For this reason, our model of the Plug Layout Table in Figure 6.5 only defines Plug Layouts that contain plug type entries for ISPs and NCPs. We do not include word clock related components in the model of the Plug Layout. This is due to the fact that, in terms of the MAP4 Transporter hardware capabilities, the number of word clock sources and word clock outputs remains the same across all Plug Layouts. Therefore, there was no need to model word clock components in each Plug Layout. Sections 6.1.2.2 and 6.1.2.3 will describe how the ISP-NCP model was incorporated into the MAP4 Transporter plug-in, and how word clock outputs and word clock source selection were handled, respectively.

The MAP4 Transporter was not originally designed with the concept of Plug Layouts. However, our proposed Transporter HAL API takes into account the existence of Plug Layouts and we have defined Plug Layouts for our plug-in implementation. One of the important implications of our decision to implement Plug Layouts led to a need to keep track of the Plug Layout that a device is using. In fulfilling this need, we resorted to using sampling frequency information to determine the Plug Layout that is in use. For example, we use the Plug Layout ID of zero to refer to the Plug Layout representing sampling frequencies of 48 kHz and below, while the Plug Layout with an ID of one represents the Plug Layout with sampling frequencies greater than 48 kHz. Bearing this in mind, our Plug Layout Table structure is initialised accordingly upon creation of the *MAP4Transporter* COM component.

In order to determine which Plug Layout configuration a Transporter is operating in, the *GetCurrentPlugLayout* function (function (6.1)) shown on page 118 is called. The sampling frequency is queried within the Transporter plug-in and a corresponding Plug Layout ID is returned. Similarly, to change the Plug Layout that a device is using, the *SetCurrentPlugLayout* function (function (6.2)) shown on page 118 is called. This results in a change in a sampling frequency register. For

instance, if a Transporter is using Plug Layout number zero, changing to Plug Layout number one results in setting the sampling frequency to 88.2 kHz. The choice of which sampling frequency to set is arbitrary, since users will configure this after the Plug Layout has been switched. This handling of Plug Layout selection on the MAP4 Transporter differs from the OGT-based Transporter, where a designated Plug Layout register is queried or set. However, Plug Layouts make the configuration more obvious to users, and eliminate the element of "surprise" when the number of plugs are reduced. This is due to the fact that the number of plugs in each Plug Layout is fixed. The number of plugs only changes when an explicit request is made, via the *INC_PLUGIN* COM interface, to change the Plug Layout.

6.1.2.2 Model and Operation of ISPs and NCPs

Transmitting NCPs and ISPs

Recall from section 4.4.1 on page 72 that the MAP4 Evaluation board houses the mLAN-NC1 and mLAN-PH2 chips. Figure 6.6 shows that each of these chips has one transmission FIFO buffer that receives audio samples and/or MIDI messages from its pins, packages these into sequences of isochronous packets and, assuming isochronous resources have been previously allocated, sends these packets onto the IEEE 1394 bus. For example, the diagram shows a chip

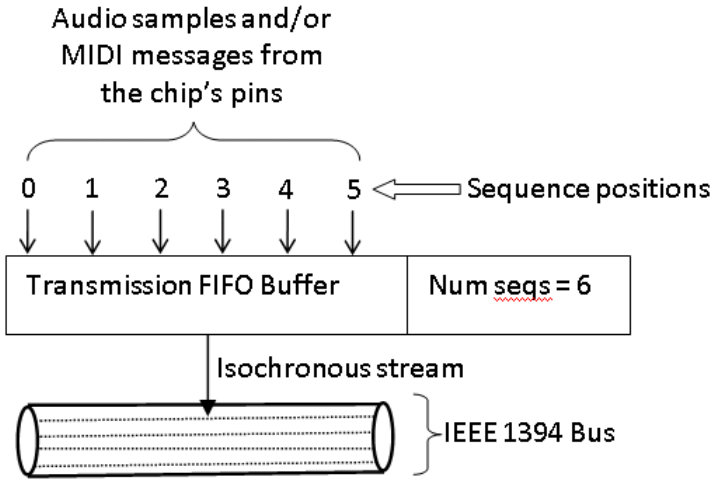


Figure 6.6: Transmission FIFO Buffer on MAP4 Transporter Chips (ASICs)

that is configured to transmit six audio sequences, where various inputs from the chip's pins correspond to various sequence positions within the isochronous packets that get sent onto the IEEE 1394 bus. Note that the number of sequences that are transmitted onto the bus is proportional to

the amount of bandwidth used, and is controllable via a register. The current Transporter HAL API provides functions to query and modify this register, namely the *GetNumSequences* and *SetNumSequences* functions, respectively. For example, reducing the number of sequences shown in Figure 6.6 to three results in a representation shown in Figure 6.7. In particular the number of

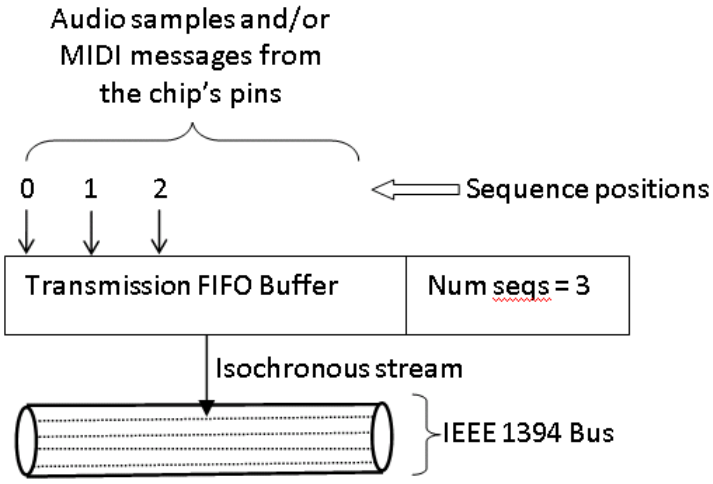


Figure 6.7: Transmission FIFO Buffer on MAP4 Transporter Chips (ASICs) - Reduced Sequences

sequence positions that are packaged in the isochronous packets is also reduced accordingly and the last three highest numbered sequence positions in Figure 6.6 will no longer be transmitted. It should be borne in mind that the ordering of sequence positions is fixed and increasing or reducing the number sequences transmitted activates or deactivates some sequence positions for data transmission onto the IEEE 1394 bus.

In the context of the above, a mechanism had to be provided to ensure that transmission from the MAP4 Transporter was handled in terms of NCPs and ISPs, and in a manner that allows for manipulation of the "number of sequences" register. We created a model where each transmission FIFO buffer on a chip represents a uniquely identifiable transmission ISP that is statically associated with a number of transmission NCPs, as shown in Figure 6.8. The number of NCPs corresponds to the maximum number of sequences that may be transmitted by a given chip. Attributes of the transmission ISP are directly implemented in the form of registers of the FIFO buffer. However, there are no registers that control the NCPs that we introduced to the design. Therefore, we implemented transmitting NCPs as *virtual in-memory* objects within the plug-in. However, the importance of transmission NCPs is that they collectively determine the number of sequences that are being transmitted by a chip's FIFO buffer.

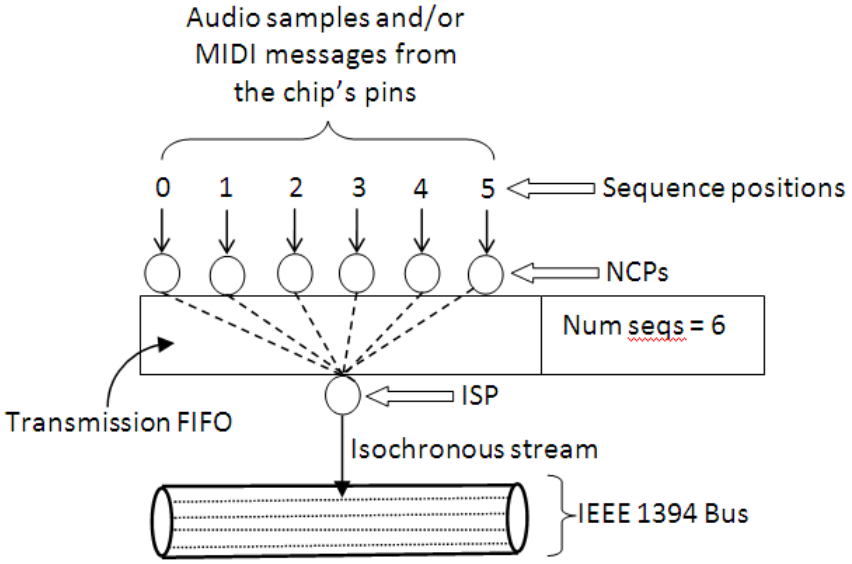


Figure 6.8: Transmission FIFO Buffer ISP-NCP model

The purpose of our *Attach* function (function (6.9)) has been described on page 122. In particular, it commences data transfer between a specified NCP and its associated ISP. Conversely, the *Detach* function (function (6.10)) stops the data transfer. In the context of the MAP4 Transporter transmission, we used the *Attach* and *Detach* functions to control the number of sequences transmitted. Figure 6.9 shows an example of how this was handled. In this example, NCPs that are shaded represent NCPs for which the *Attach* function has been called. Assume the following sequence of events takes place in the following order:

1. All NCPs start off in their *Attached* state hence all six sequences are transmitted onto the bus.
2. *Detaching* NCPs for sequences 3 and 4 has no effect on the overall number of sequences being transmitted, since the sequence at position 5 is still actively transmitting and the sequence positions are fixed.
3. The NCP for sequence 5 is *detached*.
4. The number of sequences transmitted is reduced to three since the other NCPs for sequences 3 and 4 were previously *detached*. Only when the NCP for sequence 2 is *detached* will the number of sequences reduce further. *Attaching* additional NCPs will also result in an increase in the number of sequences that are transmitted.

The manner in which the number of sequences is modified upon calling the *Attach* and *Detach* functions led to dynamic bandwidth allocation and de-allocation respectively.

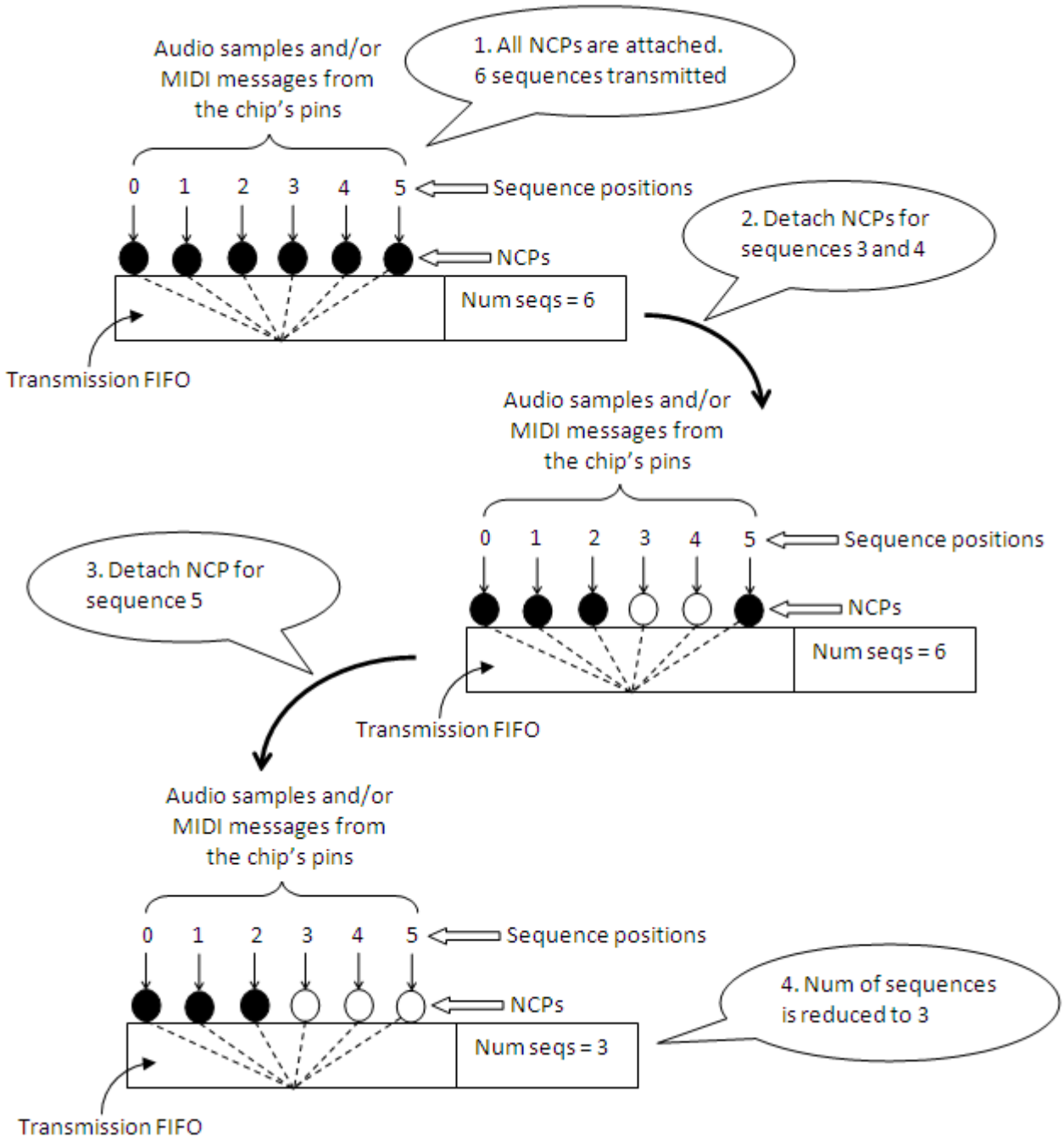


Figure 6.9: Controlling the Number of Sequences Transmitted by the MAP4 Transporter

Receiving NCPs and ISPs

As mentioned in section 4.4.1 on page 72 reception FIFO buffers exist on the chips of the MAP4 Transporter. These FIFO buffers have channel and sequence selection registers as shown in Figure 3.15 on page 48. In implementing the ISP-NCP model for reception on the MAP4 Transporter, we modelled receiving ISPs as *virtual in-memory* objects, where the number of ISPs is equivalent to the maximum number of unique isochronous streams that the Transporter can receive [Chigwamba and Foss, 2007]. In the case of the MAP4 Transporter, this is equivalent to the number of reception FIFO buffers. It should be borne in mind that receiving ISPs do not map to any registers although they keep track of the number of unique isochronous streams that a Transporter can receive. Each of the FIFO buffers shown in Figure 3.15 on page 48 now represents an NCP as shown in Figure 6.10.

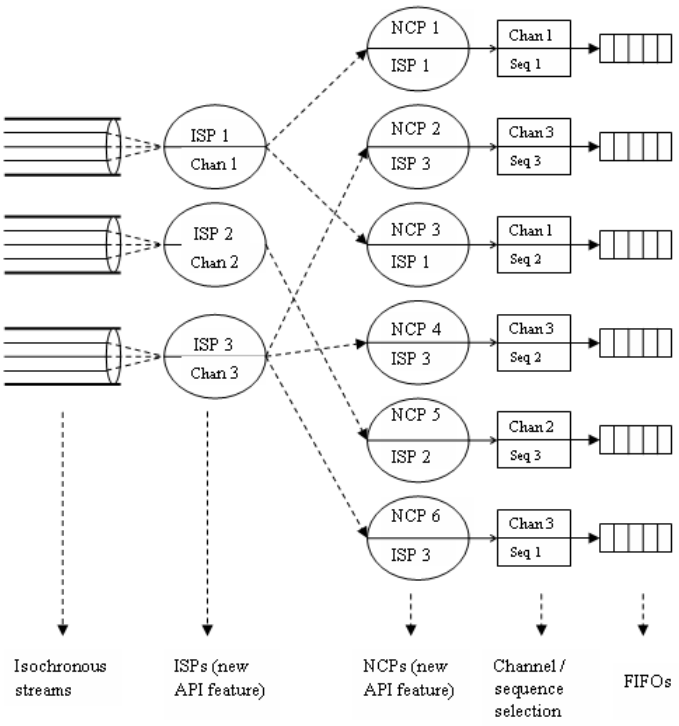


Figure 6.10: Modified mLAN Sequence Selection with NCPs and ISPs [Chigwamba and Foss, 2007]

By definition, an isochronous stream is composed of isochronous packets with the same isochronous channel number. This implies that the isochronous channel number is an attribute of an ISP. To that effect, our proposed Transporter HAL API allows channel numbers to be modified for ISPs

using the function:

$$SetIsochChannel(ispID, isochChannelNum) \tag{6.19}$$

In the context of the *SetIsochChannel* function, it is important to note that the channel number for the MAP4 Transporter is only modifiable via a FIFO buffer register. However, as mentioned above, we modelled reception FIFO buffers as NCPs. A mechanism had to be provided to ensure that our proposed Transporter HAL API was implementable in terms of the hardware of the MAP4 Transporter. In order to enable this, we ensured that the channel register of a reception FIFO buffer is only modified, if necessary, when an ISP is associated with an NCP by calling the *SetAttachedISP* function (function (6.8)) of our proposed Transporter HAL API, as described on page 122. When initialising an instance of the MAP4 Transporter, all the reception FIFO buffer channel registers are queried, and for each unique channel number found, a uniquely identifiable ISP is initialised with the given channel number. Any FIFO buffers that make use of a channel number that has an ISP that is already initialised will have their matching NCPs associated with the ISP in question.

Upon further investigation of the reception FIFO buffer operation for the MAP4 Transporter, we discovered that each of the reception FIFO buffers can be individually enabled or disabled to receive data from a given isochronous stream on the IEEE 1394 bus. This capability allowed us to provide an implementation for the *Attach* and *Detach* functions (functions (6.9) and (6.10)) which have been described on page 122. When the *Attach* function is called for a given NCP, its associated FIFO buffer commences reception of sequences with a particular sequence number from its associated isochronous stream. Conversely, when the *Detach* function is called, reception is stopped.

6.1.2.3 Word Clock Outputs and Word Clock Source Selection

Section 6.1.2.1 has already mentioned that there was no need to model the word clock component of the MAP4 Transporter in each of the two Plug Layouts that are available. This is because the number of word clocks is fixed across all the Plug Layouts that we have defined. In particular, we modelled the MAP4 Transporter as having one word clock output. Consequently, the *GetNumWordclockOutputs* function (function (6.13)) described on page 126 always returns a result of one. Given that there is only one word clock output, all the ISPs have a fixed association with this word clock output and hence, the *SetISPWCLKID* function (function (6.18)) that has been described on page 127 for the Open Generic Transporter is never called for the MAP4 Trans-

porter. Section 6.1.2.4 below describes how we handled constraints such as these, where certain attributes of a Transporter cannot be modified.

Within the single word clock output, we defined two possible word clock sources that may be selected via our proposed Transporter HAL API functions. These two types of word clock sources are an SYT word clock source and an internal word clock source as was described in section 4.4.2.3 on page 86, and are also selectable using the current Transporter HAL API. For the MAP4 Transporter, our *GetNumClockSources* function (function (6.14)) described on page 126 always returns a value of two, indicating the two selectable word clock sources on the MAP4 Transporter. Querying and specifying the word clock source that is in use may be done by calling the *GetCurrentClockSource* and *SetCurrentClockSource* functions of our proposed Transporter HAL API, respectively. The *GetCurrentClockSource* and *SetCurrentClockSource* functions (functions (6.15) and (6.16)) have been described on page 126 for the Open Generic Transporter. However, for the MAP4 Transporter, these functions are implemented via the current Transporter HAL API functions, namely the *GetSYTSynchChannel* and *SetSYTSynchChannel* functions (functions (4.8) and (4.9)) that have been described on page 86. For example, selecting an internal word clock source by calling our *SetCurrentClockSource* function results in a call to the current Transporter HAL API *SetSYTSynchChannel* function while passing an invalid SYT sync channel number, as is done for the current Transporter HAL API.

6.1.2.4 Specifying Constraints for Calling Some Proposed Transporter HAL API Functions

Throughout the implementation of the plug-in for the MAP4 Transporter, a number of special cases were identified. Examples of two such special cases are:

1. **Linked ISPs**
   Recall from section 4.4.1 on page 72 that the MAP4 Transporter uses the mLAN-NC1 and mLAN-PH2 chips. In addition, it has also been mentioned in section 4.4.2.2 on page 85 that there are some firmware restrictions within this hardware architecture that require transmission on all chips to be started or stopped simultaneously. In section 6.1.2.2, we have shown how each transmission FIFO buffer on each chip was modelled as an ISP. This implied that we needed a mechanism to ensure that all transmission ISPs on the MAP4 were started at the same time.

2. **Fixed ISP Attributes**
   In section 6.1.2.3, we mentioned that some attributes of the MAP4 Transporter are required to be fixed. This means that certain functions such as *SetISPWCLKID* should never be called in the context of the MAP4 Transporter. This indicated a need to convey information about fixed attributes to the Enabler in order to prevent any attempts to modify fixed attributes.

In the context of constraints such as the above, the Open Generic Transporter guidelines were further investigated in order ascertain how a hardware independent, high-level, abstract description such as the OGT guideline document caters for such situations. Fortunately, the OGT guideline caters for this by introducing the concepts of *fixed attributes*, *unique attributes* and *linked plugs* [Audio Engineering Society - Standards Committee, 2005]. Ideally, all attributes of a Transporter should be modifiable by an Enabler but certain constraints in devices do not allow this. Further analysis of these concepts led to a design decision being made to incorporate these concepts within some of our proposed Transporter HAL API methods. The remainder of this section describes how we have incorporated these concepts in the context of one of several of our proposed Transporter HAL API methods that require such constraints.

ATTRIBUTE CONSTRAINTS

The *GetISPWCLKID* function (function (6.17)) described for an OGT-based Transporter plug-in on page 127 had an ellipsis, indicating that certain arguments were omitted for conceptual simplicity. The purpose of the function is to query the identifier for a word clock output that feeds word clocks into a given ISP. The full declaration of the function is:

$$GetISPWCLKID(ispID, *wclkID, *pFixed, *pLinked, *pUnique, *pDependency, *pGroup) \tag{6.20}$$

In the context of the *GetISPWCLKID* function above, we focus on the numeric values that are returned by the **fixed*, **pLinked*, **pUnique*, **pDependency*, and **pGroup* arguments. This collection of arguments conveys information about *fixed attributes*, *unique attributes*, and *linked plugs* to the Enabler. We describe each of these concepts in the context of the abovementioned arguments as adapted from "*Draft AES Standard for Audio over IEEE 1394 - Specification of Open Generic Transporter*" [Audio Engineering Society - Standards Committee, 2005].

Fixed Attributes

These are parameters/attributes that a device cannot modify. The plug-in reads this information from the Transporter itself, in the case of an OGT-based Transporter, or the plug-in implements this information in its software, as in the case of our MAP4 Transporter plug-in. In particular, our proposed Transporter HAL API conveys this information to the Enabler via the the **pFixed* argument. At present there are two possible values that are returned by the **pFixed* argument, namely zero and one, where

- A value of zero implies that the attribute is modifiable.
- A value of one implies that the attribute is not modifiable, hence fixed.

Linked Plugs

These are plugs whose attribute values are dictated by attributes of other plugs. For example, in the context of starting all transmission ISPs of a MAP4 Transporter, starting of one transmission ISP results in starting all other transmission ISPs. We convey this information to the Enabler via the **pLinked* argument of some of our proposed Transporter HAL API methods. Again, there are two possible values for the **pLinked* argument, namely zero and one, where

- A value of zero implies that the given attribute is not linked with attributes of other plugs.
- A value of one implies that the attribute is part of a linked group of attributes. In this case, further information is conveyed by the **pFixed* argument. In particular, if the linked attribute is modifiable (**pFixed* has a value of zero, while **pLinked* is also one), it is called the "key" attribute of the group. When the value of a key attribute is modified, all other attributes in the group are modified. On the contrary, if the linked attribute is fixed (**pFixed* has a value of one, while **pLinked* is one) the attribute is known as a dependent attribute, and it can only be modified by modifying the key attribute.

Linked attributes have other additional information that is conveyed to the Enabler via the **pDependency* and **pGroup* arguments. The **pDependency* argument specifies whether the values of dependent attributes are identical to the key attribute, or the dependent attributes take on consecutive values starting at the value of the key attribute. The **pGroup* argument assists the Enabler to identify the number of plugs in a linked group or the position of a certain plug within the linked group.

Unique Attributes

These are attributes where no two plugs of the same group may take the same value [Audio Engineering Society - Standards Committee, 2005]. In the context of our proposed Transporter HAL API, this information is conveyed to the Enabler via the **pUnique* argument of certain methods. Similar to the **pLinked* and **pFixed* arguments, there are two possible values for the **pUnique* argument, namely zero and one, where

- A value of zero indicates that an attribute may take the same value as other attributes in the group.
- A value of one indicates that an attribute value must be different from any other in the same group.

6.1.3 PC Transporter HAL Design and Implementation

We have shown how backwards compatibility for our proposed Transporter HAL API is achieved by implementing our API functions calls in terms of the underlying hardware architecture capabilities in a similar manner to how the ISP-NCP model was implemented for the MAP4 Transporter plug-in. The current PC Transporter implementation is fulfilled by an mLAN Streaming driver that is tightly coupled to the current Transporter HAL API. Consequently, a need arose to provide a way to implement our proposed Transporter HAL API methods in terms of the current Transporter HAL API methods. In this section, we describe how this was done, and focus on how the ISP-NCP model was mapped to the current Transporter HAL API.

Figure 6.11 shows an object model for our proposed PC Transporter HAL implementation. It should be noted that much of the implementation and model remained unchanged from the current Transporter HAL API implementation. As shown in the object model, there is a *Transporter* class which declares methods of the current Tranporter HAL API. The *INC_PLUGIN* COM interface defines methods of our proposed Transporter HAL API. The *WinPCTransporter* class implements methods defined in the *Transporter* class that enable IEC 61883-6 transmission. In the context of the PC Transporter, IEC 61883-6 transmission and reception is done by a platform-specific mLAN streaming driver. Our investigation only focused on the Windows mLAN streaming driver. The *WinPCTransporter* class communicates with the mLAN streaming driver's *mLANSoftPH* process via an instance of the *WindowsMLANVDeviceInterface* class. In addition, the *WinPCTransporter* class implements methods defined by our *INC_PLUGIN* COM

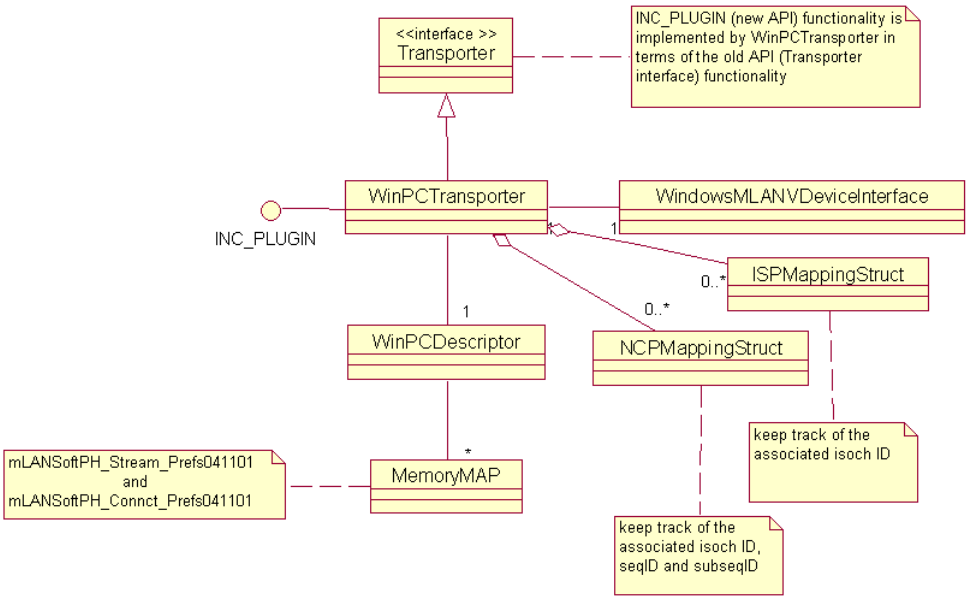


Figure 6.11: Proposed PC Transporter Implementation

interface. These methods are fulfilled in terms of the methods specified by the *Transporter* class. It was necessary to maintain this dependence since the *mLANSoftPH* process of the mLAN streaming driver operates in a manner that is tightly coupled to the current Transporter HAL API. The *WinPCDescriptor* and *MemoryMAP* classes model the state of the various components of the mLAN streaming driver. In order to map the methods of our proposed Transporter HAL API to those of the current Transporter HAL API, a mechanism to keep these mappings in memory was required. Consequently, the *NCPMappingStruct* and *ISPMappingStruct* object aggregations were introduced for this mapping. These are described in section 6.1.3.1.

We have given a motivation for the need to keep a dependence between our proposed Transporter HAL API and the current Transporter HAL API. The main component of the implementation that was affected by this was the ISP-NCP model. Before we describe this in more detail, it is important to mention a few aspects of the PC Transporter that make it different from a standalone device Transporter. As shown by the object model in Figure 6.11, we do not model a Plug Layout Table for the PC Transporter. This is due to the fact that, at the time of implementation, the current PC Transporter implementation did not change the number of plugs available for streaming upon sample rate changes. However, the number of plugs available for streaming is changed explicitly by calling the *SetNumSequences* function (function (4.4)) which was described on page 82. Another difference to note about the PC Transporter is that its implementation is part of the Enabler's implementation. Therefore, it is not implemented in a standalone plug-in (DLL) like

the OGT and MAP4 Transporter plug-ins.

6.1.3.1 Mapping the ISP-NCP Model of the Proposed Transporter HAL API to the Current PC Transporter HAL API

Recall from section 4.4.2.1 on page 78 that the PC Transporter implementation assumes that there is a fixed number of isochronous streams for either transmission or reception. Each of these isochronous streams, identified by an *isochID*, has a fixed number of audio and/or MIDI sequences. Each sequence, identified by *seqID*, is terminated by a fixed number of subsequences. Each of the subsequences, identified by *subID*, represents a sequence end-point, which is viewed as an abstract plug by the Enabler.

Figure 6.12 shows an example of the static/fixed nature of sequences and subsequences of an isochronous stream. While we only show two input isochronous streams in the diagram, it should be borne in mind that output isochronous streams are modelled in a similar manner. In practise, there are usually more than two input isochronous streams and therefore this diagram only describes the concept.

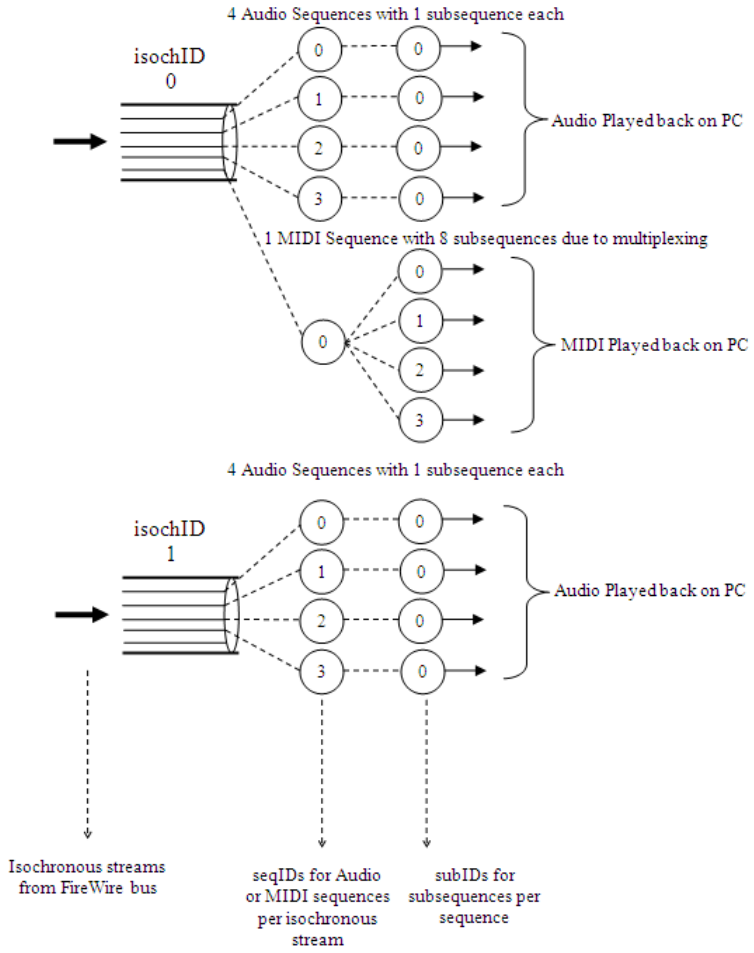


Figure 6.12: Static Representation of Input Isochronous Streams for PC Transporter

In the context of the above, we derived a mechanism that maps ISPs to the isochronous streams shown in Figure 6.12. From each ISP it then becomes possible to identify the *isochID* and *direction* of streams. The *isochID* and *direction* are required by some methods of the current Transporter HAL API. In addition, our mechanism also maps NCPs to each of the subsequences. From each NCP it then becomes possible to identify the *subID*, *seqID*, type of sequence, and the *isochID* of the subsequence associated an NCP. Effectively, this means that NCP-to-ISP associations have to be fixed, and again the constraints described in section 6.1.2.4 were proven to be useful. Figure 6.13 shows this mapping relationship more clearly. We have greyed out the intermediaries between NCPs and ISPs since these are described in Figure 6.12. Instead, we emphasise the static NCP to ISP mappings in the context of the required model.

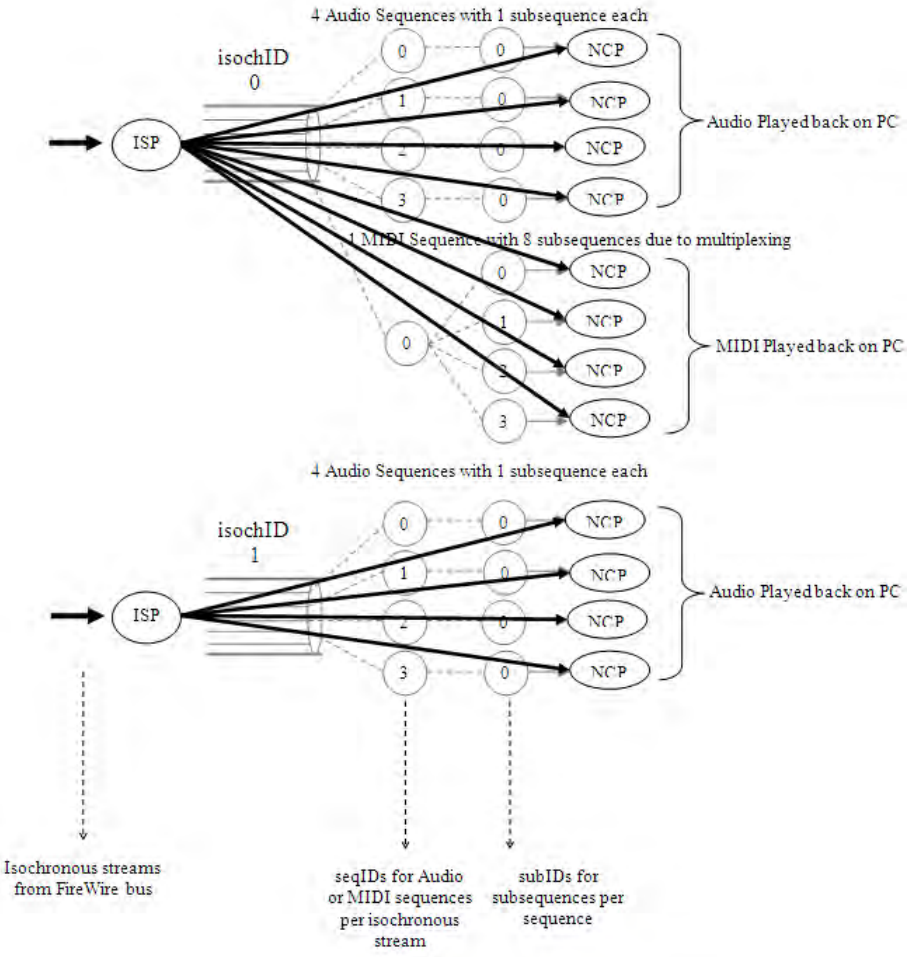


Figure 6.13: Static Representation of Input Isochronous Streams for PC Transporter with NCPs and ISPs

6.1.4 Insights Gained from HAL Implementations

We have described how our proposed Transporter HAL API, although biased towards the OGT, can be used to create HAL plug-ins for devices that are based on any hardware architecture. Section 6.1.2 described how this can be achieved by implementing our proposed Transporter HAL API function calls in terms of the underlying MAP4 Transporter hardware architecture. In addition, section 6.1.3 described another way of achieving backward compatibility by making use of the current Transporter HAL API methods. We have effectively explored two routes to achieve backwards compatibility.

Certain insights were gained from these two implementations. In particular, it was clearer that the approach described in section 6.1.3 resulted in a shorter development time, as a result of significant reuse of existing code. In addition to the reduced development time, we can also safely assume a reduced level of implementation errors as a result of reuse of tried and tested code. However, the approach described in section 6.1.2 allows for greater implementation flexibility and presumably enhances the HAL plug-in's runtime speed, since function calls of our proposed Transporter HAL API are implemented directly in terms of the underlying hardware capabilities as opposed to via an intermediary function, namely a function of the current Transporter HAL API.

6.2 Transporter Plug-In Test Application

As the plug-in implementations for the Open Generic Transporter and the MAP4 Transporter evolved, a graphical test application was created to test the plug-in mechanism and each function of our proposed Transporter HAL API. For testing purposes, DICE II EVM evaluation boards and MAP4 Evaluation boards were used. The DICE II EVM board hosts the "DICE II chip" [TC Applied Technologies] which has a firmware implementation that is based on the OGT guideline document. The MAP4 Evaluation board hosts Yamaha Corporation's mLAN-NC1 and mLAN-PH2 chips (ASICs) as mentioned in section 4.4.1 on page 72. These evaluation boards made it possible to test the feasibility of our proposed Transporter HAL API in achieving interoperability between OGT-based and non-OGT-based Transporters.

Our test application makes use of a stripped down version of the Enabler, which we refer to as the *test Enabler*. This *test Enabler* is responsible for loading plug-ins using our proposed plug-in mechanism, which was described in Chapter 5. The *test Enabler* does not handle any

high level connection management, but merely allows for low level modification of Transporter parameters by directly accessing methods of the *INC_PLUGIN* COM interface of our proposed Transporter HAL API. However, connection management is still possible using the *test Enabler*, although at a low level, by manually modifying each of the Transporter parameters required to accomplish a desired outcome, such as making a connection between two plugs. A complete listing of the *INC_PLUGIN* COM interface can be found in Appendix B. In this section we give an overview of our test application. This test application was used for connection management until a decision was made to handle high level connection management and plug-in loading using our final, complete version of the Enabler.

6.2.1 Transporter Enumeration

Our test application has capabilities to query the *test Enabler* for a list of Transporters that have been successfully enumerated. This is a way to test the functionality of our plug-in loading mechanism. All successfully loaded Transporters are displayed in a list showing their globally unique identifiers (GUIDs) and "nicknames" as shown in Figure 6.14. In particular, the diagram shows three Transporters that have been successfully enumerated, namely "MAP4", "OGT - IOne Dest", and "OGT - IOne Src". Our MAP4 Transporter plug-in is loaded for the "MAP4" Transporter, while our OGT plug-in is loaded for both the "OGT - IOne Dest" and "OGT - IOne Src" Transporters.

Figure 6.14: Test Application: Device Transporter List

6.2.2 Accessing Transporter Attributes

Selecting a Transporter from the list shown in Figure 6.14 results in display of the common Transporter attributes as shown in Figure 6.15. Briefly, these attributes include:

- An indication of the ID for the Plug Layout that is currently in use.
- A list of all the Plug Layouts that are implemented by the Transporter.
- Lists of IDs for ISPs, NCPs, and concurrently usable Word Clock Outputs (shown as Word Clock Domains) in the current Plug Layout.
- General attributes of the Transporter such as nickname and firmware version name.
- Buttons such as “Identify”, “Clear Identify”, and “Reboot”, which are associated with methods of the *INC_PLUGIN* COM interface of our proposed Transporter HAL API. The “Identify” button is used to make the light emitting diodes (LEDs) on a Transporter blink for user identification purposes, while the “Clear Identify” button stops the blinking. The “Reboot” button causes a software reset on a Transporter.

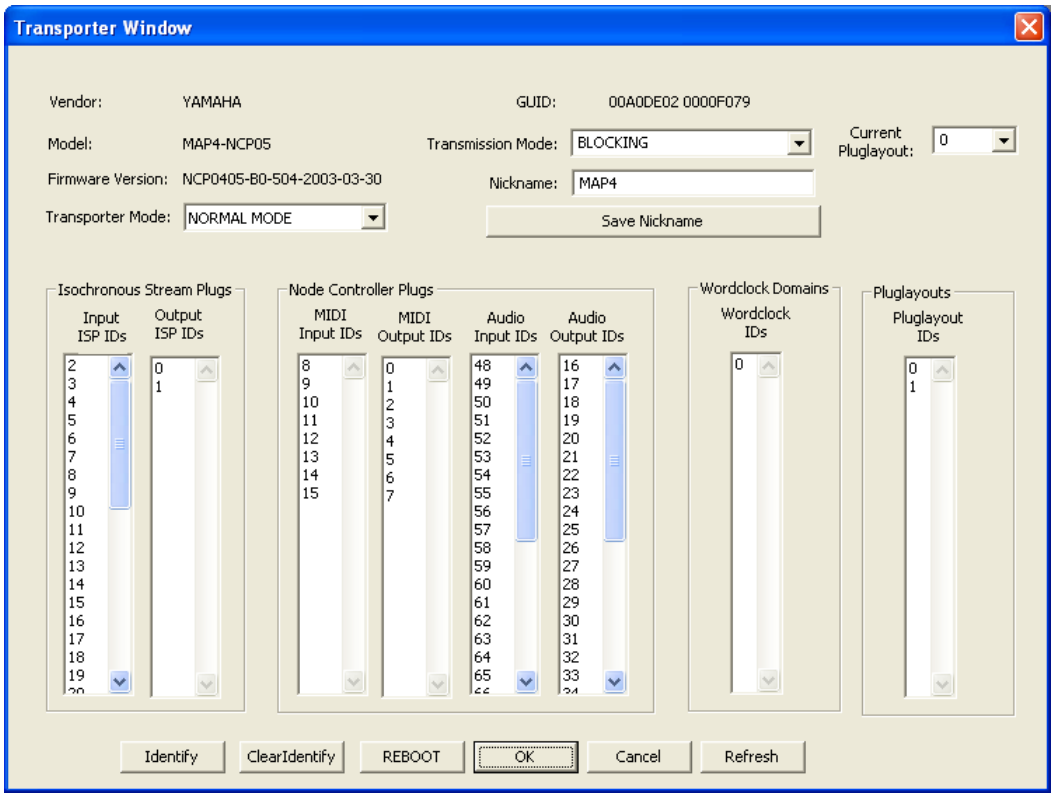


Figure 6.15: Test Application: Main Transporter Window

In the context of our proposed Transporter HAL API, we now focus on how the test application allows for viewing and modification of the attributes of three important components of a Transporter. These components relate to isochronous stream plugs (ISPs), node controller plugs (NCPs), and word clocks.

6.2.2.1 Isochronous Stream Plug (ISP) Attributes

For each of the ISP IDs shown in Figure 6.15 it is possible to view and modify attributes of the associated ISP as shown by the window in Figure 6.16. We now describe the important attributes of an ISP that are accessible via this window. Each of the attributes shown in the diagram has a corresponding Transporter HAL API method that accesses or modifies an associated Transporter parameter. The need for parameters that convey constraint information to the Enabler has been discussed in section 6.1.2.4. These constraints are labelled *Fixed*, *Dep*, *Linked*, *Unique*, and *Group* in the diagram.

Recall from section 3.3.3.2 on page 50 that an ISP represents an input or output for a single

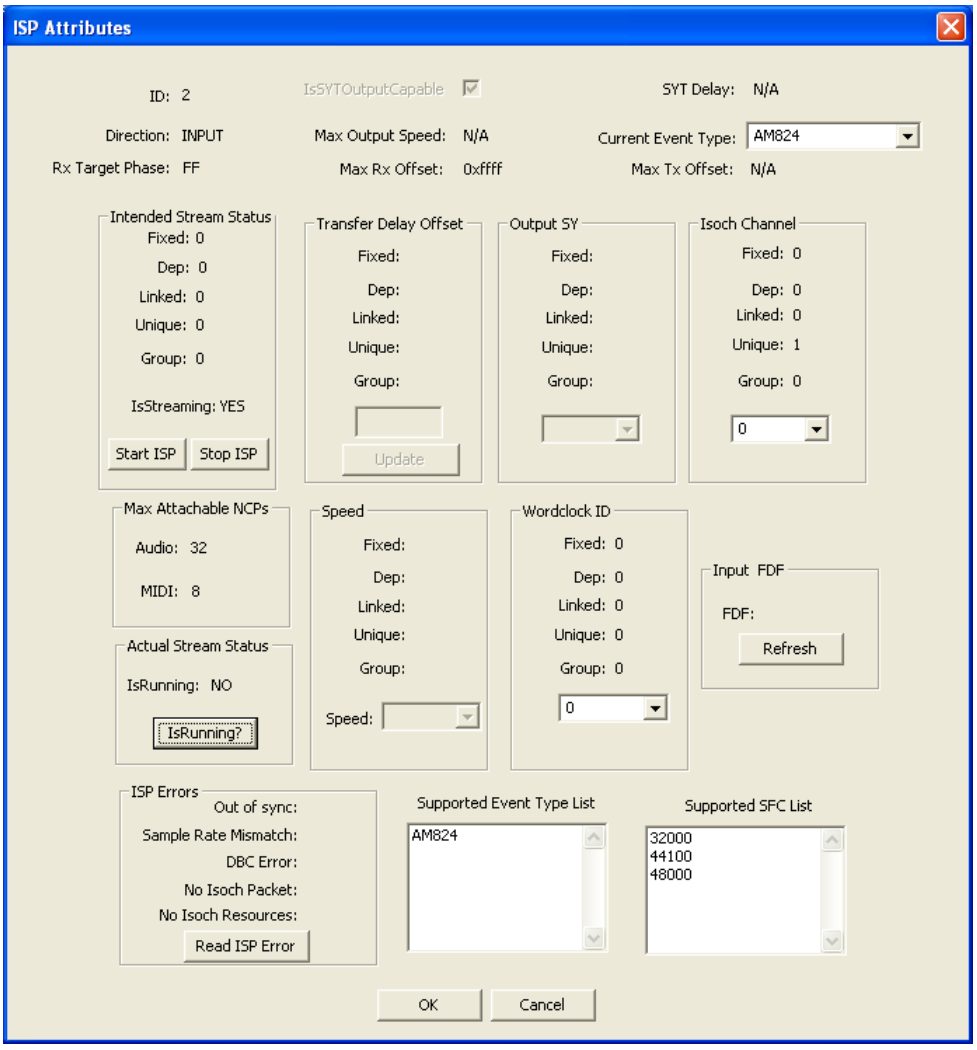


Figure 6.16: Test Application: ISP Attributes Window

isochronous stream to or from the IEEE 1394 bus, where an isochronous stream is composed of isochronous packets bearing the same channel number in their packet headers. For each ISP, there is a read-only "Direction" attribute (shown in Figure 6.16) that specifies the stream direction of the ISP. The stream direction of an ISP is either *input* or *output*, where *input ISPs* receive isochronous streams from the IEEE 1394 bus, while output ISPs send isochronous streams to the IEEE 1394 bus. This *input/output* concept has also been described in section 4.4.2.1 on page 78. The "ID" attribute specifies a unique identifier for a given ISP.

Figure 3.4 on page 30 shows the structure of the isochronous packet header. The "Isoch Channel" attribute shown in Figure 6.16 specifies the channel number of the isochronous packets that an ISP sends to or receives from the IEEE 1394 bus. This channel number corresponds to the *chan-*

nel field of the isochronous packet header. The *sy* field is the only other field of the isochronous packet header that is accessed via an attribute of an ISP. Recall from section 3.1.1.3 on page 28 that the *sy* field represents an application-specific synchronisation code. This *sy* field is accessed via the "Output SY" attribute shown in Figure 6.16.

Figure 3.7 on page 37 shows the fields within the common isochronous packet (CIP) header. The *FDF* field is the only field of the CIP header that is accessible via the window shown in Figure 6.16, through the read-only "Input FDF" attribute. This attribute is read-only since information such as the event type and sampling frequency is derived from the FDF field as described in section 3.2.1 on page 38. Of these two, only the event type is modifiable via the "Current Event Type" attribute shown in Figure 6.16. The diagram also displays a "Supported Event Type List" and a "Supported SFC List" which show all event types and sampling frequencies supported by an ISP respectively. The sampling frequency is not modified per ISP. However, there exists a "Word Clock ID" attribute that specifiers the identifier of the word clock output that drives an ISP. An ISP's sampling frequency is modifiable via its associated word clock output. This is described in section 6.2.2.3.

"Rx Target Phase", "Max Rx Offset", "Max Tx Offset", and "SYT Delay" are read-only attributes that are required by the Enabler when performing SYT phase adjustment. SYT phase adjustment is performed for devices that cannot receive A/M streams of any arbitrary SYT phase [Audio Engineering Society - Standards Committee, 2005]. It ensures that the phase of incoming A/M streams is adjusted in a manner that guarantees proper reception. The Enabler makes use of algorithms to determine extent of phase adjustment. For simplicity, our *test Enabler* does not handle SYT phase adjustment. However, the final, complete version of the Enabler handles SYT phase adjustment using algorithms that are used in the current implementation of the Enabler.

Recall from section 3.3.3.2 on page 50 that one or more NCPs may be dynamically or statically attached to ISPs, resulting in the clustering of multiple sequence end-points to an isochronous stream. The "Max Attachable NCPs" attribute shows the maximum number of NCPs of a particular type that can be associated with the given ISP. Our investigation focused on audio and MIDI NCP types only.

After the modifiable parameters of an ISP have been configured, transmission or reception of isochronous packets with a specified channel number in their header is commenced by clicking on the "Start ISP" button shown in Figure 6.16. This button is associated with the *Start* function (function (6.11) on page 124) of the *INC_PLUGIN* COM interface of our proposed Transporter HAL API. Conversely, transmission or reception of the isochronous packets is stopped by click-

ing on the "Stop ISP" button. This button is associated with the *Stop* function (function (6.12) on page 124) of the *INC_PLUGIN* COM interface of our proposed Transporter HAL API.

Various errors that may prevent an ISP from successfully transmitting or receiving isochronous packets may occur . These errors are indicated by the flags of the "ISP Errors" attribute. All the errors shown except for the "No Isoch Resources" error relate to *input ISPs*. *Input ISP* errors mainly occur when there are no isochronous packets with the expected channel number on the IEEE 1394 bus, when there is a mismatch between the ISP's expected sampling frequency and the sampling frequency that was used to create the isochronous packets, or when the isochronous packets are corrupt. The "No Isoch Resources" error is flagged when an *output ISP* fails to transmit isochronous packets as a result of failure to allocate isochronous resources, namely when the channel that is specified by the "Isoch Channel" attribute is already in use or when there is insufficient bandwidth available for isochronous packets to be transmitted.

6.2.2.2 Node Controller Plug (NCP) Attributes

Attributes of each NCP that is associated with each NCP ID shown in Figure 6.15 can be viewed or modified using the window shown in Figure 6.17. We now describe the important attributes of

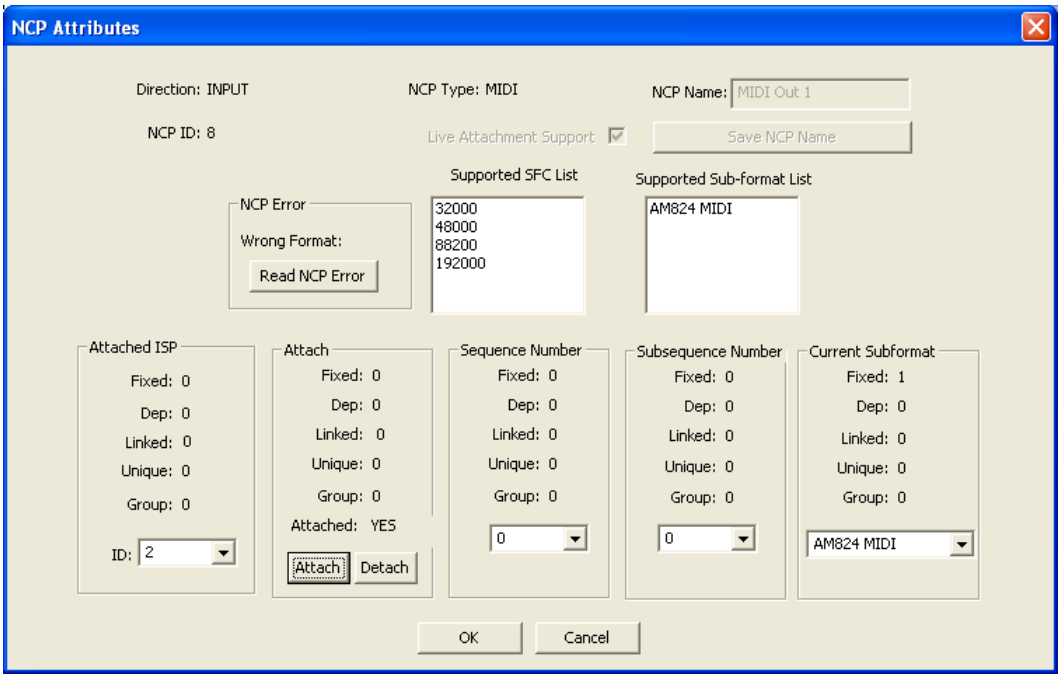


Figure 6.17: Test Application: NCP Attributes Window

an NCP that are accessible via this window. Again, each of the attributes shown in the window

has a corresponding method within the *INC_PLUGIN* COM interface of our proposed Transporter HAL API that accesses and/or modifies an associated Transporter parameter. Also note the presence of constraint information shown by the labels *Fixed*, *Dep*, *Linked*, *Unique*, and *Group*, as was described for ISPs in section 6.2.2.1 above.

Recall from section 3.3.3.2 on page 50 that an NCP represents an input or output for a single monaural channel of audio or a single cable of MIDI to or from the node application. In this regard, NCPs are equivalent to sequence end-points for the sequences that are packed in the isochronous packets. The "Direction" attribute for an NCP (shown Figure 6.17) has the same meaning as the "Direction" attribute for ISP attributes (described in section 6.2.2.1 above), where *input* represents data being received from the IEEE 1394 bus, while *output* represents data being sent to the IEEE 1394 bus. In particular, each *input NCP* unpacks audio/MIDI data from an associated *input ISP* and sends it out to the node application, while each *output NCP* receives data from the node application and packs the data to an isochronous stream before it is transmitted onto the IEEE 1394 bus via an *output ISP*. The "NCP ID", "NCP Name", and "NCP Type" attributes specify the unique identifier, the user-friendly textual name, and the data type (audio or MIDI) for a given NCP, respectively.

As mentioned in section 3.3.3.2 on page 50, each NCP is associated with an ISP, either dynamically or statically. The "Attached ISP" attribute (shown in Figure 6.17) specifies the ID of the ISP that is associated with the NCP. The ISP sends or receives isochronous packets with a particular channel number. Recall from section 3.2.1 on page 37 that each isochronous packet contains a number of packet clusters known as data block*s*. Each data block in the isochronous packet has a number of quadlets (32-bit elements). These quadlets are the AM824 formatted audio samples and/or MIDI messages, where the position of a quadlet corresponds to the sequence that it belongs to. This sequence position is known as the sequence number. The "Sequence Number" attribute (shown in Figure 6.17) is responsible for determining this sequence number, indicating the position where audio/MIDI data should be packed to (for *output NCPs*) or unpacked from (for *input NCPs*) within each isochronous packet that is transmitted or received. Section 3.2.1 on page 37 also described how data such as MIDI can be multiplexed onto a single sequence. The multiplex index is known as the subsequence number. This subsequence number is determined by the "Subsequence Number" attribute (shown in Figure 6.17).

Table 3.5 on page 40 shows the subformats the exist for the AM824 formatted data. The "Supported Sub-format List" shows all the subformats that are supported by an NCP. Since NCPs are equivalent to sequence end-points, the subformat type for the sequence that each NCP terminates is determined by the "Current Subformat" attribute (shown in Figure 6.17). In the case of *out-*

put NCPs, this attribute specifies the "actual" subformat of the transmitted sequence. However, for *input NCPs*, the attribute specifies the "expected" subformat for the received sequence. In situations, where the subformat of the sequence received by an *input NCP* is different from the expected subformat, the "Wrong Format" error is flagged by the "NCP Error" attribute. This is the only possible error that can be flagged for NCPs, since *output NCPs* do not flag any errors. Although the sampling frequency is not modifiable per NCP, a list of supported sampling frequencies is shown in the "Supported SFC List".

When all relevant attributes of an NCP have been configured, transmission or reception of audio/MIDI data to or from the associated ISP is commenced by clicking on the "Attach" button shown in Figure 6.17. This button is associated with the *Attach* function (function (6.9) on page 122) of the *INC_PLUGIN* COM interface of our proposed Transporter HAL API. Conversely, transmission or reception of audio/MIDI data is stopped by clicking on the "Detach" button, which is associated with the *Detach* function (function (6.10) on page 122) of the *INC_PLUGIN* COM interface of our proposed Transporter HAL API.

6.2.2.3 Word Clock Attributes

The last main component of our proposed Transporter HAL API that was tested is word clock outputs and word clock source selection functionality. For each of the word clock IDs shown in Figure 6.15, word clock source selection is done via the window shown in Figure 6.18. The window is split into two main sections, namely a "Current Wordclock Domain" section and an "Available Clock Sources" section. Once again, each of the attributes shown in the diagram has a corresponding method within the *INC_PLUGIN* COM interface of our proposed Transporter HAL API that accesses and/or modifies an associated Transporter parameter.

The "Current Wordclock Domain" section shows attributes of a word clock output, which is identified by the "Wordclock ID" attribute on the top of the window. These attributes include the identifier ("Current Clock" attribute) for the word clock source that is used to generate word clocks, the sample period ("Sample Period" attribute) of the generated word clocks, and an indication of any error conditions (error flags of the "Wordclock Errors" attribute) that are present. Word clock source selection for the word clock output is possible by changing the value of the "Current Clock" attribute, via a drop-down list. The drop-down list contains the identifiers of all the available word clock sources that can be selected. Possible word clock output errors are "Wordclock Loss" and "Sample Rate Mismatch". A "Wordclock Loss" error occurs when the generated word clock output is unstable, while a "Sample Rate Mismatch" error occurs when

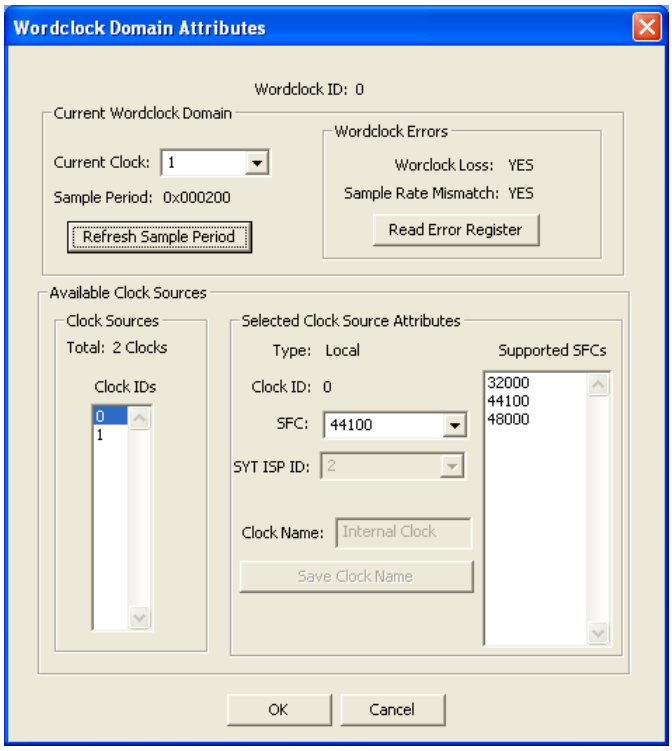


Figure 6.18: Test Application: Word Clock Attributes Window

there is a mismatch between the word clock source's sampling frequency attribute and the actual sampling frequency of receive isochronous streams [Audio Engineering Society - Standards Committee, 2005].

The "Available Clock Sources" section is further divided into two subsections, namely the "Clock Sources" subsection and the "Selected Clock Source Attributes" subsection, as shown in Figure 6.18. The "Clock Sources" subsection shows a list of unique identifiers ("Clock IDs" list) for available word clock sources. When an identifier for an available word clock source is selected in the "Clock IDs" list, the attributes of the selected word clock source are shown in the "Selected Clock Source Attributes" subsection. The "Type" attribute indicates the mode of the word clock source, namely local or SYT. Figure 4.17 on page 87 clearly illustrates the distinction between local and SYT word clock modes relative to the IEEE 1394 bus. A word clock source that is in local mode receives its timing information from either the device's internal clock or an external word clock generator that is attached to the device, while a word clock source in SYT mode receives timing information from the IEEE 1394 bus. For word clock sources in SYT mode, it is possible to specify the ID ("SYT ISP ID" attribute) of the *input ISP* that receives isochronous packets from which the SYT timing information is read. Figure 3.7 on page 37 shows that the

SYT field is part of the CIP header of isochronous packets that are received by the ISP. Sampling frequencies supported by a word clock source are shown in the "Supported SFCs" list, while the sampling frequency that is used by the word clock source is shown, and can also be modified, via the "SFC" attribute. A user-friendly textual name of a word clock source is shown by the "Clock Name" attribute.

6.3 Chapter Summary

In this chapter we have described important concepts of our proposed Transporter HAL API at a low level. The description of our proposed Transporter HAL API focused on three compatible proof-of-concept HAL implementations, namely an Open Generic Transporter (OGT) plug-in implementation, a MAP4 Transporter plug-in implementation, and a PC Transporter HAL implementation. The OGT plug-in implementation demonstrates how our API completely utilises the OGT capabilities. The MAP4 Transporter plug-in and PC Transporter HAL implementations show two different approaches to achieve backwards compatibility using our proposed Transporter HAL API. Transporters that are controllable using the current Transporter HAL API require changes to their HAL implementations in order to be compatible with our proposed Transporter HAL API. We have recommended an approach to make these changes, while minimising the risk of implementation errors and reducing development time. This is done by fulfilling our proposed Transporter HAL API methods in terms of the current Transporter HAL API methods, where possible.

The chapter has also given an overview of a test application that was developed to test each of our proposed Transporter HAL API member functions for the MAP4 Transporter and OGT plug-in implementations. This test application made it possible to verify correct operation of our proposed Transporter HAL API before any changes were integrated with the final, complete version of the Enabler, in order to fulfil high level connection management tasks.

We used the test application that has been described in this chapter to verify correct functionality of our proposed API with OGT-based and non-OGT-based Transporters. The Enabler was enhanced to fully utilise our proposed Transporter HAL API. The next chapter, Chapter 7, focuses on the innovations that were realised as a result of enhancing the Enabler to fully utilise our proposed Transporter HAL API.

Chapter 7

Innovations From Proposed Transporter HAL API

The previous chapter has described the important concepts of our proposed Transporter HAL API. This Transporter HAL API completely utilises the capabilities of existing Transporters and also the additional capabilities of the Open Generic Transporter that are lacking in the current Transporter HAL API. The Enabler implementation was modified to fully utilise our proposed Transporter HAL API. Consequently, connection management applications that make use of the Enabler were enhanced in order to make use of the enhanced capabilities that are inherent within our proposed Transporter HAL API.

In this chapter we begin by revisiting the main investigation areas of our research. We introduce an additional investigation area that has not been discussed thus far. This investigation area relates to a connection management server application which utilises the Enabler, and the client applications that interact with the connection management server. We then provide a more detailed overview of the connection management client and server applications. The server serves up capabilities of our modified Enabler to client applications. We discuss the extent to which client applications have benefited from our proposed Transporter HAL API. Some of the more subtle optimisations that were realised at the Enabler level will be discussed. We conclude the chapter by recommending areas that require future investigation.

7.1 Overview of Investigation Areas

Figure 7.1 shows an overview of the investigation areas that are relevant to this thesis.

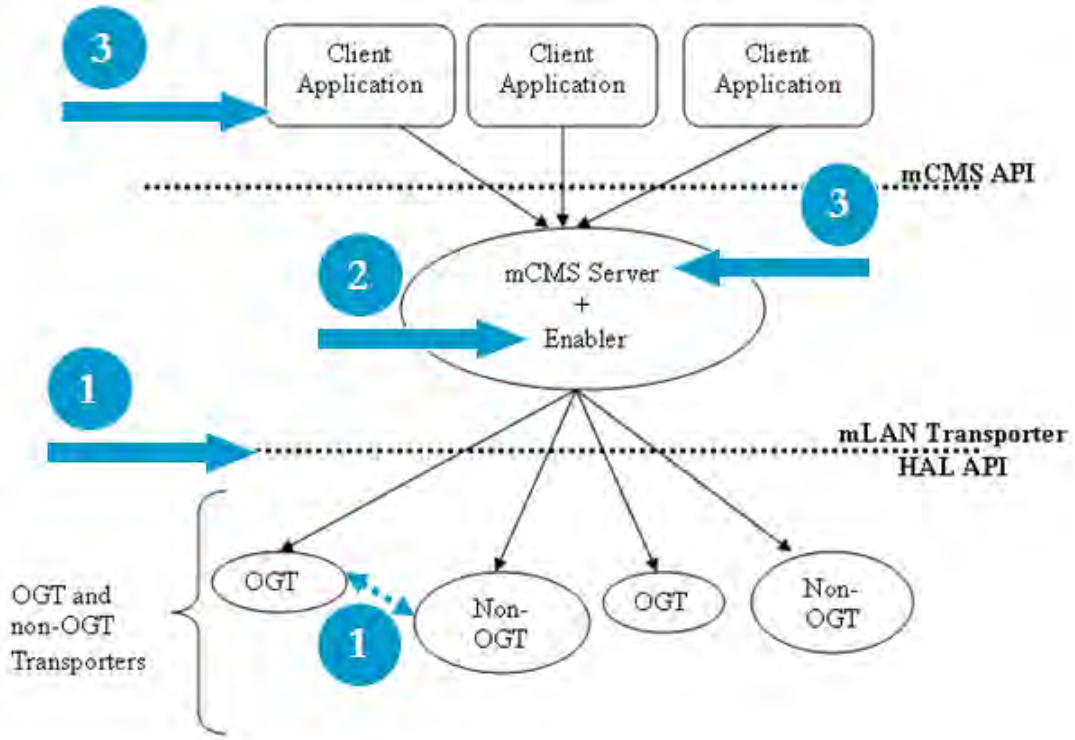


Figure 7.1: Overview of Investigation Areas

Our discussions thus far have been focusing on the areas labelled 1 and 2. Investigation area number 1 resulted in our proposed Transporter HAL API and its associated HAL implementations for both OGT-based and non-OGT-based Transporters, as described in Chapter 6. Investigation area number 2 resulted in our proposed plug-in mechanism and modifications made to the Enabler in order to fully utilise our proposed Transporter HAL API. Our proposed plug-in mechanism has been described in Chapter 5. The enhanced capabilities within our proposed Transporter HAL API have also resulted in enhanced Enabler capabilities.

Up to this point, our discussions have been limited to the Enabler and its interaction with Transporters. We now introduce a third area of investigation which is labelled 3 in Figure 7.1. Recall from section 3.3.2.1 on page 44 that the Enabler has an "mLAN Plug Abstraction Layer". This layer exposes an API which is used by connection management applications to fulfil connection management requests. Our third investigation area focuses on an example of a connection man-

agement application that uses the Enabler. This connection management application is based on a client-server architecture, where the server fulfils connection management requests from graphical client applications by calling on the Enabler.

As mentioned in the first paragraph of this section, our proposed Transporter HAL API has resulted in enhanced Enabler capabilities. Consequently, these enhancements to Enabler capabilities have resulted in an enhanced API between the connection management server and the Enabler. We now discuss the connection management client-server architecture in more detail. We focus more on the enhanced end-user graphical client application capabilities that have been made possible by our proposed Transporter HAL API. Some capabilities of our proposed Transporter HAL API have resulted in non-graphical enhancements within the Enabler. These subtle innovations are discussed separately.

7.2 mLAN Connection Management Server

The mLAN Connection Management Server (mCMS) is a server application that runs on the same workstation as the Enabler [Fujimori, Foss, Klinkradt, and Bangay, 2003]. It accepts mLAN device word clock synchronisation, and audio or MIDI routing requests from client applications. These requests are fulfilled via the Enabler. Client applications are the various graphical user applications used by end-users to effect connection management in networked audio environments such as broadcast and recording studios, stadiums, houses of worship, “hotels, and convention centres” [Foss, Fujimori, Chigwamba, Klinkradt, and Okai-Tettey, 2006]. Client applications run on a variety of devices ranging from PDAs and Laptops, to PC workstations. The server also reports to client applications when the status of devices on the IEEE 1394 network changes. Communication between each client and the server is in the form of a documented XML protocol across a TCP/IP connection [Networked Audio Solutions, 2006]. Figure 7.2 shows an example of an mLAN client-server configuration as we have described it here.

An investigation into the Transporter HAL API and resolving issues identified in section 4.4.2 on page 78 inevitably leads to an enhanced Enabler interface to connection management applications such as the mCMS. As a secondary objective, our investigation aimed to demonstrate the impact that a change in Transporter HAL API has on the capabilities of end-user graphical client applications. The mLAN connection management server implementation was modified accordingly in order to completely serve all the capabilities of the enhanced Enabler to client applications. Consequently, the XML protocol which is used for client-server communication

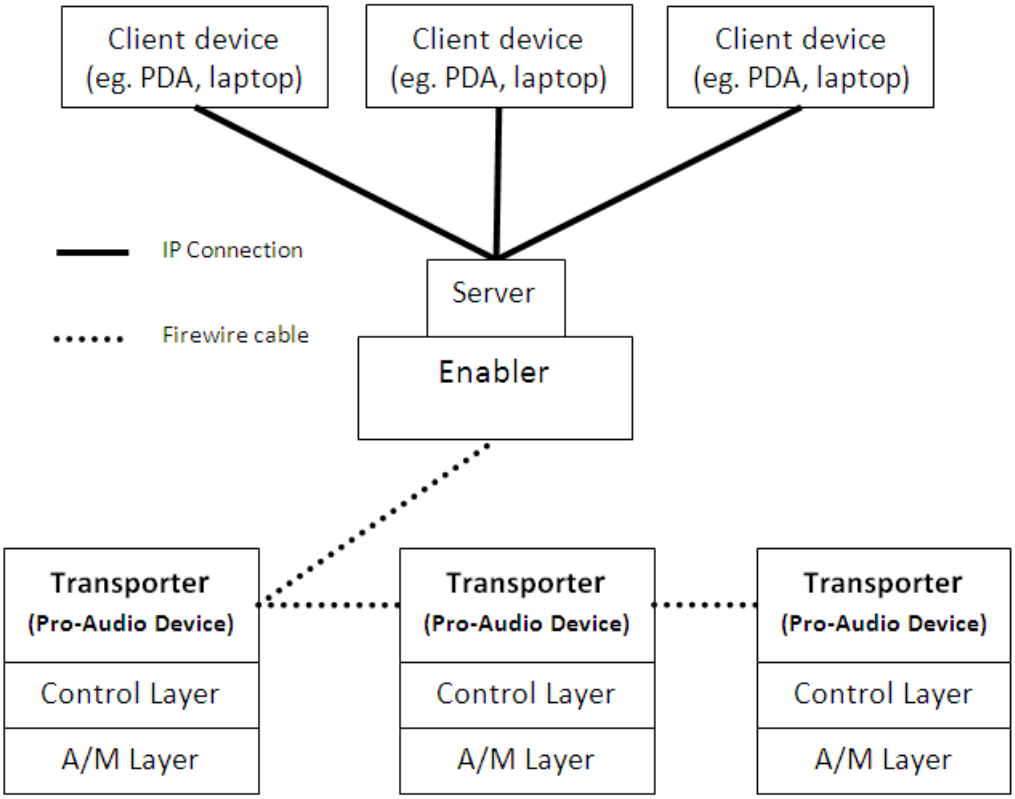


Figure 7.2: mLAN Client-Server Configuration [Fujimori et al., 2003]

was modified. We now focus on an example of a client application that was modified to demonstrate enhanced graphical client application capabilities as documented in "Enhancing End-User Capabilities in High Speed Audio Networks" [Chigwamba and Foss, 2007]. Reference is also made to important sections of the XML protocol that were modified or introduced.

7.2.1 Enhanced Graphical Client Application Capabilities

7.2.1.1 Making and Breaking Connections

Figure 7.3 shows a screenshot of the main connection management window for the list based client application that was modified to demonstrate the benefits of our proposed Transporter HAL API. The diagram shows two tree lists, one for source plugs and another for destination plugs. Each of these lists has a node that represents a bus with a given ID. In our case, the bus with an ID of 3FF represents the local bus. Recall from section 3.1.1.2 on page 27 that there may be 1024 buses on an IEEE1394 network. Each bus can have up to 64 nodes. Also recall

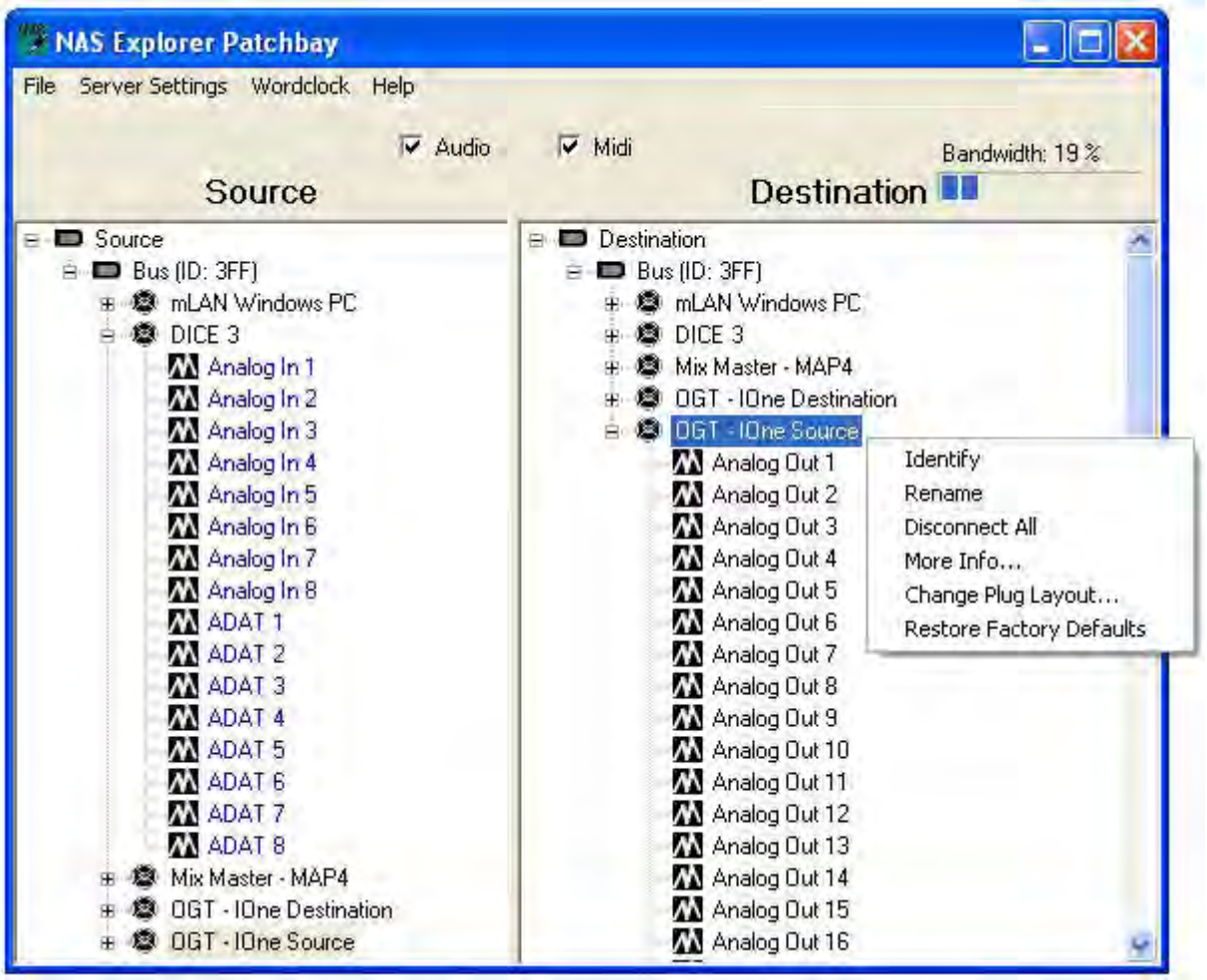


Figure 7.3: Main Connection Management Patch Bay Window [Chigwamba and Foss, 2007]

from section 4.3 on page 59 that bridge portals are used in multi-bus environments. However, we have mentioned that our investigation was carried out in the context of a single-bus environment. Hence, only the local bus is shown in Figure 7.3.

For each of the bus nodes within Figure 7.3, there is a list of "nicknames" for Transporters that are present on the bus. These Transporters include "mLAN Windows PC", "DICE 3", "Mix Master MAP4", "OGT - IOne Destination", and "OGT - IOne Source". For each of these Transporters, a list of the corresponding source or destination audio/MIDI plugs is shown. At a low level, these plugs correspond to the NCPs which have been incorporated in our proposed Transporter HAL API. Each of these NCPs is uniquely identified by an NCP ID. At a high level we simply refer to this NCP ID as the plug ID, since we are guaranteed that it is unique for a given Transporter.

In order to make a connection and ensure that audio/MIDI is routed from a source plug to a

given destination plug, the client application sends an XML document which is similar to the one shown in Figure 7.4 to the server. For both the source and destination plugs, the globally

```
<?xml version="1.0" encoding="utf-16" ?>
- <mLANServerCommand version="1.0">
  - <object name="patch">
    - <method name="connect">
        <parameter name="sourceGUID" value="0013f00400400011" />
        <parameter name="sourcePlugType" value="audio" />
        <parameter name="sourcePlugID" value="1" />
        <parameter name="destinationGUID" value="0013f00400000014" />
        <parameter name="destinationPlugType" value="audio" />
        <parameter name="destinationPlugID" value="33" />
      </method>
    </object>
  </mLANServerCommand>
```

Figure 7.4: Connect Request XML Document

unique ID (GUID) of the Transporter on which the plug resides and the plug type (audio or MIDI) are sent to the server as is done in the current implementation. Unlike in the current implementation, where plug names are sent in addition to this information, we are now sending the plug ID for each of the plugs. Our model makes it possible for plug names such as "Analog In 1" to be changed if the Transporter's firmware implementation permits. In such cases, the use of a plug name as part of a unique identifier for a plug is flawed, since it will be possible for more than one plug to share the same plug name. Our plug ID approach guards against such an undesirable outcome.

To break any audio/MIDI stream connections between a source and destination plug, the client application sends an XML document which is similar to the one shown in Figure 7.5 to the server. Only the destination plug information is required in this XML document, since a destination plug can only be connected to one source plug. Using the destination plug's ID, the server is able to identify the connected source plug via the Enabler. The information required for the destination plug includes the Transporter's GUID, plug type (audio or MIDI), and the plug ID. Once again, note our use of the plug ID as opposed to the plug name.

Figure 7.3 shows a context menu that is superimposed above the destination plug list. We focus on the "More Info..." and "Change Plug Layout" menu items, since they demonstrate some aspects that derived from our proposed Transporter HAL API. The "More Info..." menu item leads to the display of a node's information, as described in section 7.2.1.2. The "Change Plug

```
<?xml version="1.0" encoding="utf-16" ?>
- <mLANServerCommand version="1.0">
  - <object name="patch">
    - <method name="disconnect">
        <parameter name="destinationGUID" value="0013f00400000014" />
        <parameter name="destinationPlugType" value="audio" />
        <parameter name="destinationPlugID" value="32" />
      </method>
    </object>
</mLANServerCommand>
```

Figure 7.5: Disconnect Request XML Document

Layout" menu item allows a user to change the current Plug Layout for a device, as described in section 7.2.1.3.

7.2.1.2 Conveying More Information to Users

Selecting the "More Info..." context menu item of the context menu shown in Figure 7.3 results in the display of the node information for the Transporter which corresponds to the node that is selected in either the source or destination plugs list of Figure 7.3. Figure 7.6 shows the node information for the "OGT - IOne Source" Transporter, which is selected in the destination plugs list. The information shown for this Transporter includes the nickname, GUID, number of

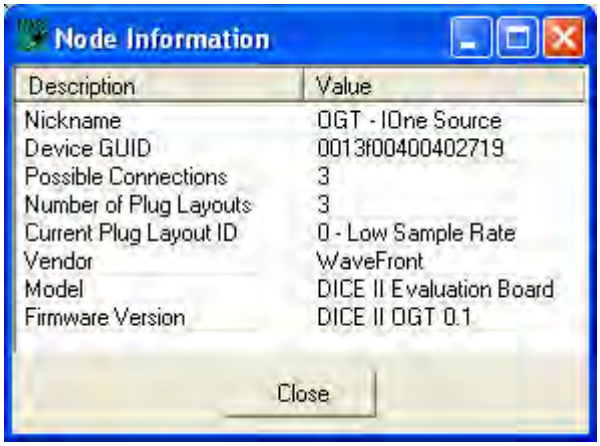


Figure 7.6: Client Application: Node Information Window

possible connections, number of Plug Layouts implemented, ID of the current Plug Layout, name of the vendor, model name, and firmware version name. Three of these elements are a result of

our proposed Transporter HAL API, namely the number of possible connections, number of Plug Layouts implemented, and current Plug Layout ID.

Possible Connections

The "Possible Connections" entry in Figure 7.6 gives information about the number of MIDI/audio stream connections that can be made from other additional unique isochronous streams. We describe this concept using a simplified example, as shown in Figure 7.7. In the diagram, there are

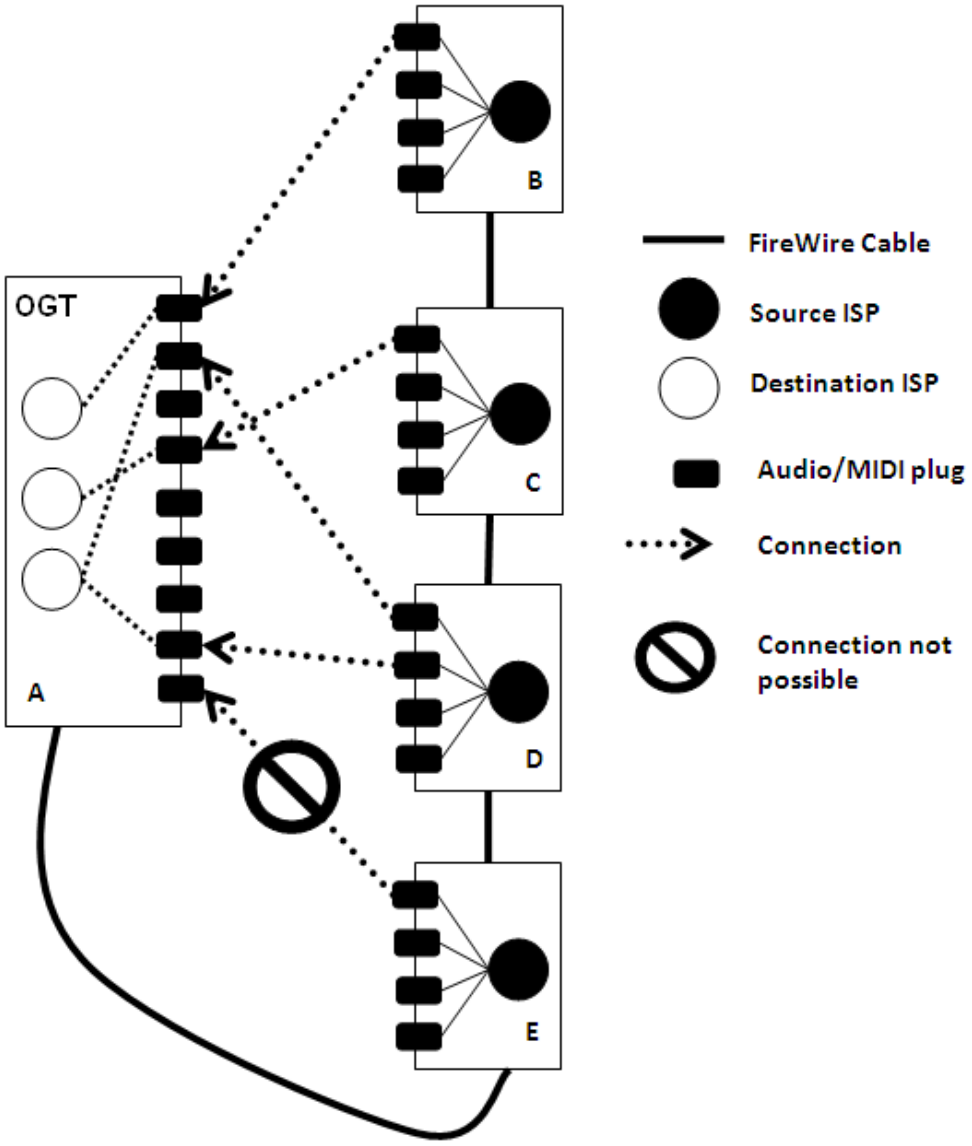


Figure 7.7: Possible Connections Concept

five Transporters labelled A, B, C, D and E. These Transporters are daisy-chained using IEEE 1394 cables. With the exception of Transporter A, which is receiving data from the IEEE 1394 bus, all other Transporters are transmitting data onto the IEEE 1394 bus.

Recall from section 3.1.1.4 on page 31 that each transmitter uses a unique isochronous channel number that is allocated from the isochronous resource manager node. It follows that isochronous packets that are transmitted by each Transporter shown in Figure 7.7 make use of unique isochronous channel numbers. By definition, isochronous stream plugs (ISPs) receive or transmit isochronous packets with a specific channel number. In order to stream audio/MIDI successfully, each source (transmitting) audio/MIDI plug is associated with a source ISP, while each destination (receiving) audio/MIDI plug is associated with a destination ISP. When a connection is made between a source audio/MIDI plug and a destination audio/MIDI plug, the associated destination ISP receives the isochronous packets that are transmitted by the source ISP. It should be borne in mind that each destination ISP can only receive packets from one source ISP. However, isochronous packets that are transmitted by a single source ISP can be received by more than one destination ISP. Therefore, the stream connection between B and A makes use of one destination ISP of Transporter A. The stream connection from C to A makes use of another destination ISP of Transporter A. The two stream connections from D to A make use of the remaining destination ISP of Transporter A. Now, all the destination ISPs are in use, which implies that no more connections except from Transporters B, C and D are possible. For this reason, the connection from E to A is not possible, since there are no more available destination ISPs. Assuming that no connections were made to Transporter A, the value of "Possible Connections" would be 3. For each connection made to Transporter A, this value is decremented by 1. Note that the value is not decremented if there is destination ISP on Transporter A that is already receiving from the relevant source ISP, as shown for the two connections from D to A in Figure 7.7. In particular, the when the first of the two connections from D to A is made the value of "Possible Connections" is decremented by 1. However, when the second connection is made, the value of "Possible Connections" is not decremented since the connection shares the same destination and source ISPs as the first connection.

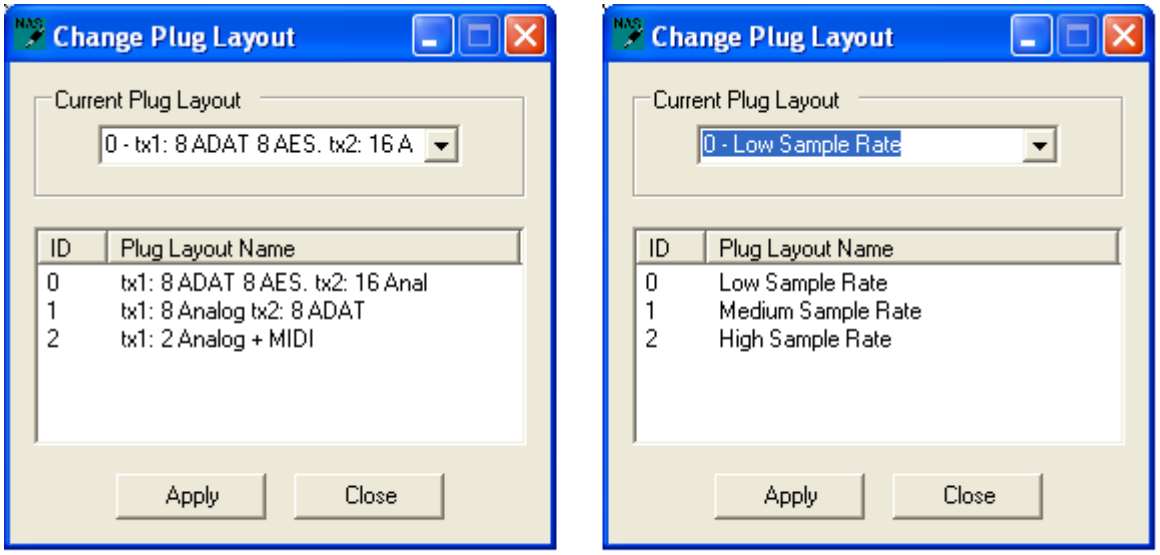
At a low level, the "Possible Connections" element shown in Figure 7.6 shows the number of available *input ISPs* on a given Transporter, and hence the number of connections that are possible from other unique isochronous streams that are transmitted by other Transporters. Such proactive notification has been made possible by our proposed Transporter HAL API, since it is defined in terms of node controller plugs (NCPs) and isochronous stream plugs (ISPs). It should be borne in mind that only OGT-based Transporters are affected by such a constraint in

cases where the number of NCPs is greater than the number of ISPs. Our approach to proactive user notification results in fewer surprises when connection requests between plugs of different Transporters fail. However, some level of prior user education about this constraint is necessary.

Plug Layout Information

Recall from section 6.1 on page 115 that our proposed Transporter HAL API explicitly takes into account the concept of Plug Layouts and Plug Layout switching. Consequently, it is possible to query the number of Plug Layouts on a Transporter and also to retrieve the ID of the Plug Layout that a Transporter is using. This information is shown by the "Number of Plug Layouts" and "Current Plug Layout ID" entries in Figure 7.6. We focus on how Plug Layouts are actually switched in section 7.2.1.3.

7.2.1.3 Changing Plug Layouts



(a) Plug Layout switching based on bandwidth optimisation.
(b) Plug Layout switching based on sample rate capabilities.

Figure 7.8: Client Application: Plug Layout Switching Windows

We have mentioned that our proposed Transporter HAL API takes into account the concept of Plug Layouts and Plug Layout switching. Figure 7.8 shows two examples of how Plug Layouts can be changed from a client application. Figure 7.8 (a) demonstrates how Plug Layouts can

be used for bandwidth optimisation as mentioned in section 4.4.2.5 on page 88. In particular, diagram (a) shows that there are three possible configurations for a device. The first configuration makes use of two transmitters, where one transmitter is capable of transmitting eight ADAT and eight AES digital audio channels, while the other transmitter is capable of transmitting 16 analog audio channels. The second configuration also makes use of two transmitters, where one transmitter is capable of transmitting eight analog audio channels, while the other transmitter is capable of transmitting eight ADAT digital audio channels. The third configuration makes use of one transmitter that is capable of transmitting two analog audio channels and one MIDI stream. Such Plug Layout switching is desirable in environments where audio/MIDI routing requirements are known in advance.

Figure 7.8 (b) shows how Plug Layouts can be used to display the number of audio plugs that are available based on the sample rate that is used for either analog-to-digital or digital-to-analog conversion for the audio channels, namely low (48 kHz and lower), medium (88.2 kHz and 96 kHz), and high (176.4 kHz and higher) sample rates. In some hardware architectures, fewer audio channels can be routed at higher sample rates than at lower sample rates. To mention in passing, Plug Layouts for the current NCP05-based Transporters are based on sample rate capabilities. In particular, the number of audio channels are that can be transmitted/received at sample rates above 48 kHz is half the number that can be transmitted/received at 48 kHz and lower [Yamaha Corporation, 2002b].

When a Plug Layout is changed, the client application sends an XML document similar to the one shown in Figure 7.9 to the server. This XML document indicates the GUID of the Transporter

```
<?xml version="1.0" encoding="utf-16" ?>
- <mLANServerCommand version="1.0">
  - <object name="patch">
    - <method name="setCurrentPlugLayout">
        <parameter name="GUID" value="0013f00400000014" />
        <parameter name="plugLayoutID" value="1" />
      </method>
    </object>
  </mLANServerCommand>
```

Figure 7.9: Change Plug Layout Request XML Document

on which the Plug Layout is to be switched. In addition, the ID of the Plug Layout to switch to is also provided. Note that the Plug Layout functionality is not implemented in the current Transporter HAL API, hence, this XML document is new to the client-server XML protocol.

While we have discussed Plug Layouts based on bandwidth optimisation and sample rate capabilities, it should be borne in mind that the nature of the Plug Layout concept makes it adaptable to any other device quirks. Plug Layout names can be made more meaningful and thus describe the capabilities of a Plug Layout to users.

7.2.1.4 Multiple Word Clock Outputs

Recall from section 4.4.2.4 on page 87 that the OGT guideline document defines the capability of having more than one concurrently usable word clock output, although current OGT-based Transporter firmware implementations only make use of one word clock output. Notwithstanding, our proposed Transporter HAL API attempts to take this feature into account. As a result, the *GetNumWordClockOutputs* function (function (6.13) on page 126), and the *GetISPWCLKID* and *SetISPWCLKID* functions (functions (6.17) and (6.18) on page 127) have been defined in our proposed Transporter HAL API in order to incorporate the capability of multiple word clock outputs.

The client application was modified to reflect this capability to end-users. Figure 7.10 shows a word clock setup window that incorporates the capability to concurrently use multiple word clock outputs.

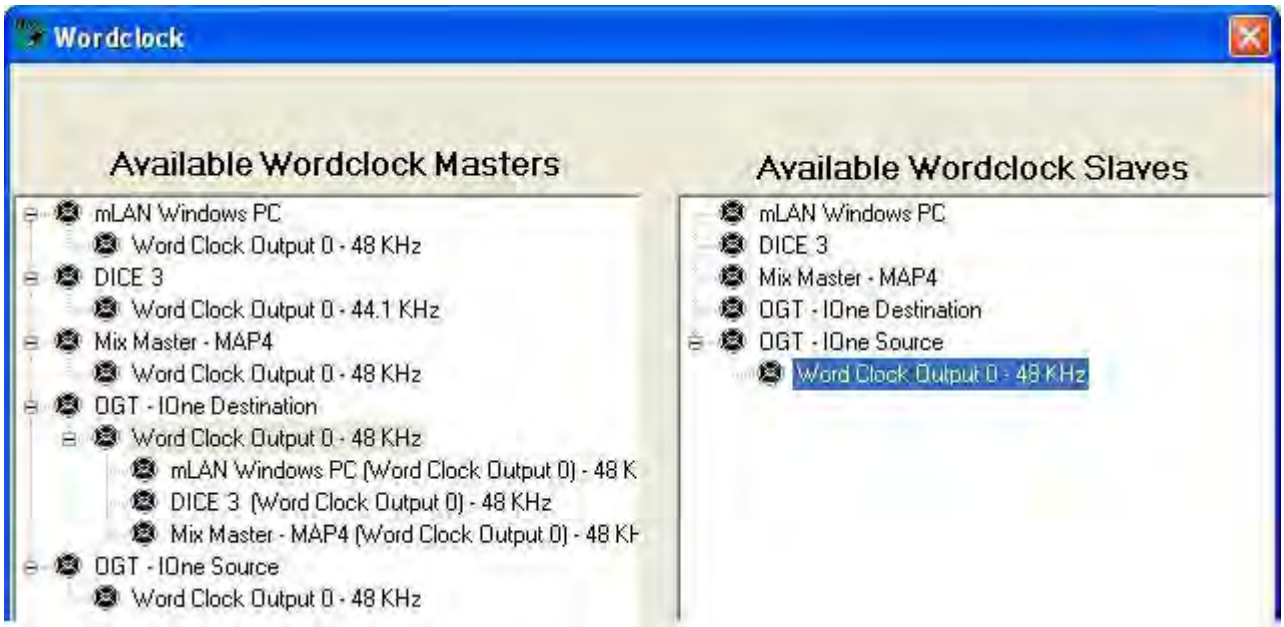


Figure 7.10: Word Clock Setup Window [Chigwamba and Foss, 2007]

Work clock slaves can be synchronised to a word clock master via this window. The window shows two tree lists, one for available word clock masters and another for available word clock slaves. The first level of nodes on both lists represent the Transporters that are present on the IEEE 1394 bus. The second level of nodes on both lists represent the word clock outputs that are available on each Transporter. The sampling frequency of the word clock output's word clock source is also shown at this level. It is at this level that more than one word clock outputs for a Transporter are shown. On the "Available Wordclock Masters" list, there is an additional third level of nodes that shows word clock outputs that are slaves of a particular master word clock output on the Transporter. We explore the levels of this hierarchy for the "OGT - IOne Destination" Transporter node of the "Available Wordclock Masters" list in Figure 7.10, as shown below:

- OGT - IOne Destination.....(first level node)
  - Word Clock Output 0 - 48 kHz.....(second level node)
    - mLAN Windows PC (Word Clock Output 0) - 48 kHz...(third level node)
    - DICE 3 (Word Clock Output 0) - 48 kHz.....(third level node)
    - Mix Master - MAP4 (Word Clock Output 0) - 48 kHz....(third level node)
  - Word Clock Output 1.....(second level node)
  - Word Clock Output 2.....(second level node)

We have added second level nodes above to show how multiple word clock output could be incorporated in future, namely "Word Clock Output 1" and "Word Clock Output 2". We reiterate that at the time of this writing, there were no OGT-based Transporter firmware implementations implemented multiple word clock outputs, hence this functionality could not be tested. In this regard, section 7.4 highlights some areas that can be further investigated.

7.2.1.5 Word Clock Source Selection

For each word clock output in the "Available Wordclock Masters" list of Figure 7.10, it is possible to select the word clock source that generates word clocks for the word clock output. Section 4.4.2.3 on page 86 has already described the inadequacy of the current Transporter HAL API capabilities with regards to the lack of external word clock source selection. As a result of our proposed Transporter HAL API, word clock source selection now includes the capability to select external word clock sources in addition to the device's internal word clock source (built-in clock).

Our proposed word clock source selection mechanism is done via a window such as the one shown in Figure 7.11. The window shows the currently selected word clock source. This selected

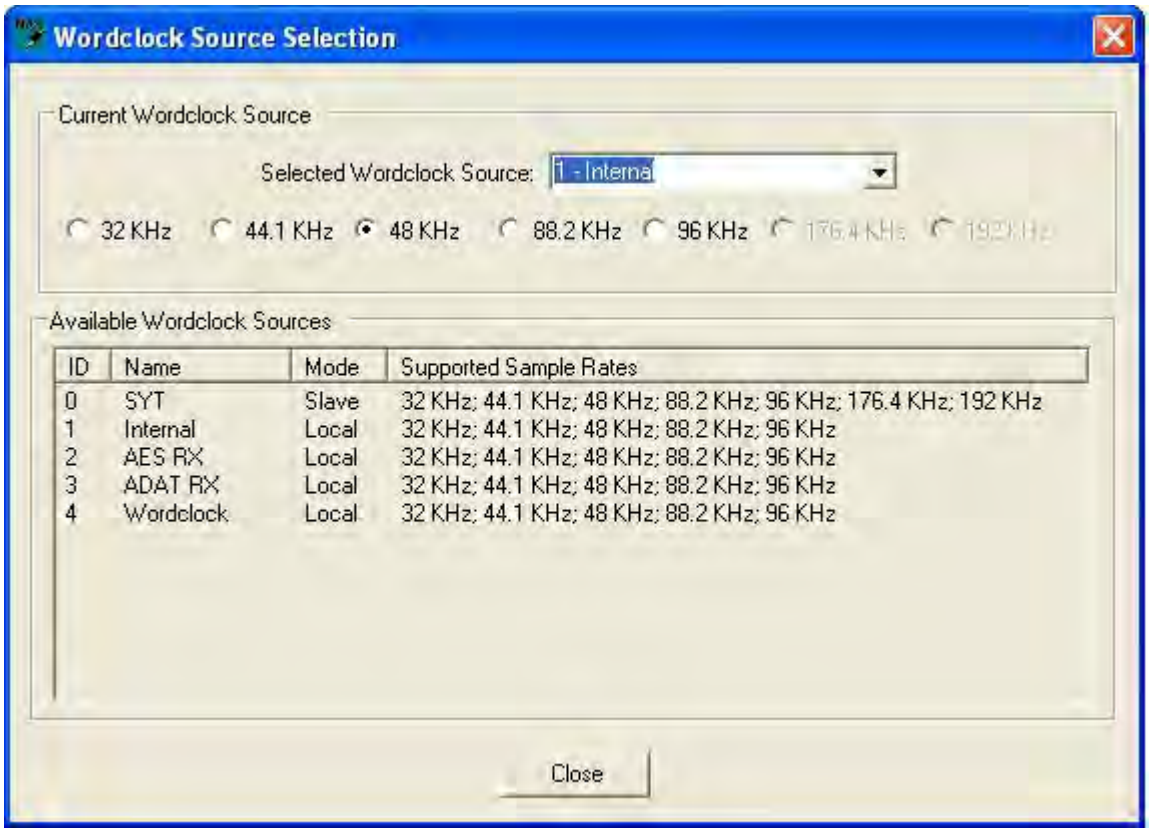


Figure 7.11: Word Clock Source Selection Window [Chigwamba and Foss, 2007]

word clock source can be changed to any "local" word clock sources in the list of available word clock sources. In the diagram, the local word clock sources that may be selected include "Internal", "AES RX", "ADAT RX" and "Wordclock". The "slave" word clock source may not be selected since it is selected by default whenever a slave word clock output is synchronised to a master word clock output. In addition to selecting a word clock source, it is also possible to specify the sampling frequency for the selected word clock source from its range of supported sampling frequencies. Note that the word clock source selection capability, as described here, is not available in existing client applications as a result of an inadequacies within the current Transporter HAL API. Our proposed Transporter HAL API has made this word clock source selection possible.

After master-to-slave synchronisation relationships have been setup as described in section 7.2.1.4, and word clock source selection has been done in the manner described above, a synchronisation

setup request XML document similar to the one shown in Figure 7.12 is sent to the server. This

```
<?xml version="1.0" encoding="utf-16" ?>
- <mLANServerCommand version="1.0">
  - <object name="samplerate">
    - <method name="syncsetup">
        <parameter name="masterGUID" value="0013f00400400011" />
        <parameter name="masterWordClockOutputID" value="0" />
        <parameter name="masterCurrentSyncSourceID" value="1" />
        <parameter name="masterSampleRate" value="0000BB80" />
        <parameter name="slave" value="NODE_GUID='0013f00400000014',WORD_CLOCK_ID='0'" />
      </method>
    </object>
  </mLANServerCommand>
```

Figure 7.12: Synchronisation Setup Request XML Document

XML document specifies the GUID of the Transporter that hosts the master word clock output, the ID of the master word clock output, the selected word clock source, the sampling frequency (in hex format), and a list of one or more slave word clock outputs. For each slave word clock output, the GUID of the Transporter on which the slave word clock output resides, and the ID of the word clock output are required, as shown in Figure 7.12.

7.2.1.6 Updating the FireWire Network Configuration

We have shown how a client applications graphically display the additional capabilities that have been derived from our proposed Transporter HAL API. We now show how the mLAN connection management server conveys information to client applications. In particular, we use an example of a configuration XML document that the server sends to connected client applications when any configurations on the IEEE 1394 network change, such as when devices are added to or removed from the network, when connections between plugs are made or broken, and when synchronisation is set up. This allows all client applications to update their displays in order to truly represent the IEEE 1394 network upon configuration changes.

Figure 7.13 shows an example of a configuration XML document. The document models the IEEE 1394 network. As shown in the diagram, there is an *IEEE1394Network* element that comprises a number of *IEEE1394Bus* elements. As mentioned in section 7.2.1.1, we focus on a single bus environment. Hence, the XML document only shows the local bus. The *IEEE1394Bus* element comprises up to 63 *IEEE1394Device* elements, since 63 is the maximum number of devices that can be present on an IEEE 1394 bus. Each *IEEE1394Device* element that corresponds to a device with a Transporter node implementation is composed of an *mLANDevice* element.

```
- <IEEE1394Network>
  - <IEEE1394Bus bandwidthAvailable="2756" busName="3FF">
    - <IEEE1394Device GUID="0013f00400400011" firmware="DICE II OGT 0.1" model="DICE II Evaluation
        Board" nickname="IOne Connects-left" nicknameIsWriteable="yes"
        numPossibleDeviceConnections="4" vendor="WaveFront">
      - <mLANDevice>
        - <mLANDevicePlugs>
            <plug direction="out" id="0" isDangling="yes" nameIsWriteable="no" plugName="AES1 L"
              plugType="audio" />
            <plug direction="out" id="1" isDangling="yes" nameIsWriteable="no" plugName="AES1 R"
              plugType="audio" />
            <plug direction="out" id="2" isDangling="yes" nameIsWriteable="no" plugName="MIDI In"
              plugType="midi" />
            <plug direction="in" id="3" isDangling="no" nameIsWriteable="no" plugName="AES1 L"
              plugType="audio" />
            <plug direction="in" id="4" isDangling="no" nameIsWriteable="no" plugName="AES1 R"
              plugType="audio" />
            <plug direction="in" id="5" isDangling="no" nameIsWriteable="no" plugName="MIDI Out"
              plugType="midi" />
          </mLANDevicePlugs>
        - <mLANDevicePlugLayouts currentPlugLayoutID="0" numPlugLayouts="3">
            <plugLayout id="0" nameIsWriteable="no" plugLayoutName="Low Sample Rate" />
            <plugLayout id="1" nameIsWriteable="no" plugLayoutName="Medium Sample Rate" />
            <plugLayout id="2" nameIsWriteable="no" plugLayoutName="High Sample Rate" />
          </mLANDevicePlugLayouts>
        - <mLANDeviceSyncSources numSyncSources="4">
            <syncSource currentSampleRate="0000bb80" id="0" nameIsWriteable="no"
              supportedSampleRates="00007d00|0000ac44|0000bb80|00015888|00017700|0002b110|"
              syncMode="1" syncSourceName="SYT" />
            <syncSource currentSampleRate="0000bb80" id="1" nameIsWriteable="no"
              supportedSampleRates="00007d00|0000ac44|0000bb80|00015888|00017700|0002b110|"
              syncMode="3" syncSourceName="Internal" />
            <syncSource currentSampleRate="0000bb80" id="2" nameIsWriteable="no"
              supportedSampleRates="00007d00|0000ac44|0000bb80|00015888|00017700|0002b110|"
              syncMode="3" syncSourceName="AES RX" />
            <syncSource currentSampleRate="0000bb80" id="3" nameIsWriteable="no"
              supportedSampleRates="00007d00|0000ac44|0000bb80|00015888|00017700|"
              syncMode="3" syncSourceName="ADAT RX" />
          </mLANDeviceSyncSources>
        - <mLANDeviceWordClockOutputs numWordClockOutputs="1">
            <wordClockOutput currentSyncSourceID="1" id="0" masterGUID="none"
              masterWordClockOutputID="none" />
          </mLANDeviceWordClockOutputs>
        </mLANDevice>
      </IEEE1394Device>
    + <IEEE1394Device GUID="0013f00400000014" firmware="X1 OGT 070222" model="Firewire Digital
        Audio Snake" nickname="IOne Connects-right" nicknameIsWriteable="yes"
        numPossibleDeviceConnections="3" vendor="I/One Connects">
    </IEEE1394Bus>
  </IEEE1394Network>
```

Figure 7.13: FireWire Network Configuration XML Document

The *mLANDevice* element comprises the four main elements that model a Transporter, namely the *mLANDevicePlugs*, *mLANDevicePlugLayouts*, *mLANDeviceSyncSources*, and *mLANDevice-*

WordClockOutputs elements. We now describe the various elements of the configuration document shown in Figure 7.13 in more detail. We only highlight the important modifications that were made to the configuration document.

With reference to Figure 7.13, the *IEEE1394Device* element shows that we have added information to show the number of possible connections, as described in section 7.2.1.2. The *numPossibleDeviceConnections* attribute of the *IEEE1394Device* element shows this number of possible connections.

The *mLANDevicePlugs* element is composed of a number of *plug* elements. These *plug* elements represent the high level plug abstractions that are exposed by the Enabler to connection management applications such as the server. The direction of the plugs is provided via the *direction* attribute, namely "in" for destination plugs and "out" for source plugs. For each plug, the ID, plug type, plug name, and an indication of whether the name is modifiable are provided via suitably named attributes. The *isDangling* attribute is a yes/no attribute that was introduced to shows whether a previously connected source or destination plug has lost its connection. Connections are lost when two plugs of two different Transporters are connected and one of the two Transporters is subsequently removed from the IEEE 1394 network. The plug on the Transporter that remains on the network will still be configured to send or receive data to/from a plug that no longer exists, hence the name *dangling plug*. We also introduced the *id* and *nameIsWriteable* attributes. Section 7.2.1.1 has already alluded to the importance of the plug ID attribute in cases where the plug name is modifiable. A plug name is modifiable when the value of the *nameIsWriteable* attribute is "yes". At the time of this writing, there were no Transporter firmware implementations that allowed plug names to be modified, but when the need arises, our proposed Transporter HAL API will already be compatible.

Section 7.2.1.2 introduced the concept of Plug Layouts within the client application. The manner in which Plug Layouts of a Transporter are switched has been described in section 7.2.1.3. The *mLANDevicePlugLayouts* element of Figure 7.13 shows information about the Plug Layouts that are implemented by a Transporter. This element has two attributes, namely *curentPlugLayoutID* and *numPlugLayout*s. The *numPlugLayout*s attribute shows the total number of Plug Layouts on a Transporter, while the *currentPlugLayoutID* attribute shows the ID of the Plug Layout that is currently active on a Transporter. The Plug Layout ID is a number between zero and the total number of Plug Layouts on a Transporter. For each Plug Layout on the corresponding Transporter, there is a *plugLayout* element within the *mLANDevicePlugLayouts* element. Each *plugLayout* element has attributes that show the Plug Layout's ID, its name, and an indication of whether the name is modifiable. The capability to access Plug Layouts from client applications is

an innovation that has been derived from our proposed Transporter HAL API, hence this section of the XML document is new to the XML protocol.

The process of word clock source selection has been described in section 7.2.1.5. The *mLANDeviceSyncSources* element shown in Figure 7.13 gives information about all the word clock sources that are implemented by a Transporter. The element has a *numSyncSources* attribute which shows the total number word clock sources that are implemented by the Transporter. For each word clock source that is implemented by the Transporter, there is a *syncSource* element within the *mLANDeviceSyncSources* element. Each *syncSource* element shows information about the associated word clock source such as its ID, the sample rate that is in use, a list of supported sample rates, the mode[1], the name of the word clock source, and an indication of whether the name is modifiable or not. Although word clock synchronisation is currently using client applications that utilise the capabilities of the current Transporter HAL API, it does not take into account the selection of external word clock sources. Our XML design in this regard has been influenced by our proposed Transporter HAL API, and hence takes into account the capability of external word clock source selection from client applications.

Section 7.2.1.4 has described how multiple word clock outputs are modelled by our modified client application. The *mLANDeviceWordClockOutputs* element of Figure 7.13 shows how multiple word clock outputs are modelled in the configuration XML document that is sent to client applications from the server. The element has a *numWordClockOutputs* attribute that shows the total number of word clock outputs available on the associated Transporter. For each word clock output, a *wordClockOutput* element exists within the *mLANDeviceWordClockOutputs* element. Each *wordClockOutput* element contains attributes that show information about the associated word clock output, namely its ID, and the word clock source that it is currently associated with. In cases where the word clock output is a slave of another word clock output as described in section 7.2.1.4, the ID of the master word clock output and GUID of the Transporter hosting the master word clock output are given as attributes, otherwise the “none” keyword is used in both cases.

[1] A value of 01_2 represents slave mode, while a value 11_2 represents local mode

7.3 More Subtle Non-Visual Innovations

7.3.1 Resource Optimisation

Up to this point we have been discussing the visual client application enhancements that resulted from our proposed Transporter HAL API. We now focus on some non-visual innovations that have been derived from our proposed Transporter HAL API. These innovations relate to isochronous resource optimisation. We use Figure 7.14 to demonstrate this resource optimisation. As shown in the diagram, assume there are two Transporters on an IEEE 1394 bus, labelled A and B. Transporter A is transmitting data while Transporter B is acting as receiver. For Transporter A, we have only shown the source (*output*) ISPs and NCPs, although the Transporter can have destination (*input*) ISPs and NCPs as well. Conversely, for Transporter B, we have only shown destination ISPs and NCPs, although source ISPs and NCPs may be present. Also shown in the diagram are the word clock sources associated with the various ISPs, namely a local word clock source for Transporter A and a slave word clock source for Transporter B. While the choice of word clock source can be done in any order, we use this arrangement to demonstrate a case for which we identified the possibility for resource optimisation.

Typically, word clock synchronisation is one of the first things that is done before audio data connections are made between plugs of Transporters. In the case of Figure 7.14, assume that Transporter A is the word clock master of Transporter B. Recall from section 3.2.3 that timing information is transmitted via the *SYT* field of an isochronous packet. As shown in the top half of Figure 7.14, timing information from Transporter A is sent to Transporter B via Transporter A's ISP number 0. This timing information is received by Transporter B's ISP number 0. At this point, there is a flow of isochronous streams from Transporter A to Transporter B via the corresponding ISPs. In order for this to occur, bandwidth is allocated for any NCPs of Transporter A that are associated with its ISP number 0.

After synchronisation has been setup, audio/MIDI connections are typically made. In cases where a transmitter is also a word clock master, we identified the need for resource optimisation. Such resource optimisation is necessary for Transporters with more than one source ISP, as in the case of Transporter A. We give an example of a connection between NCP number 3 of Transporter A and NCP number 0 of Transporter B. Note that NCP number 3 of Transporter A has a fixed association with ISP number 1, and thus ISP number 1 has to transmit to the receiver. However, SYT timing information is transmitted via ISP number 0 of Transporter A. This implies that both ISPs on the transmitter are streaming to the same receiver. On the receiver side, two

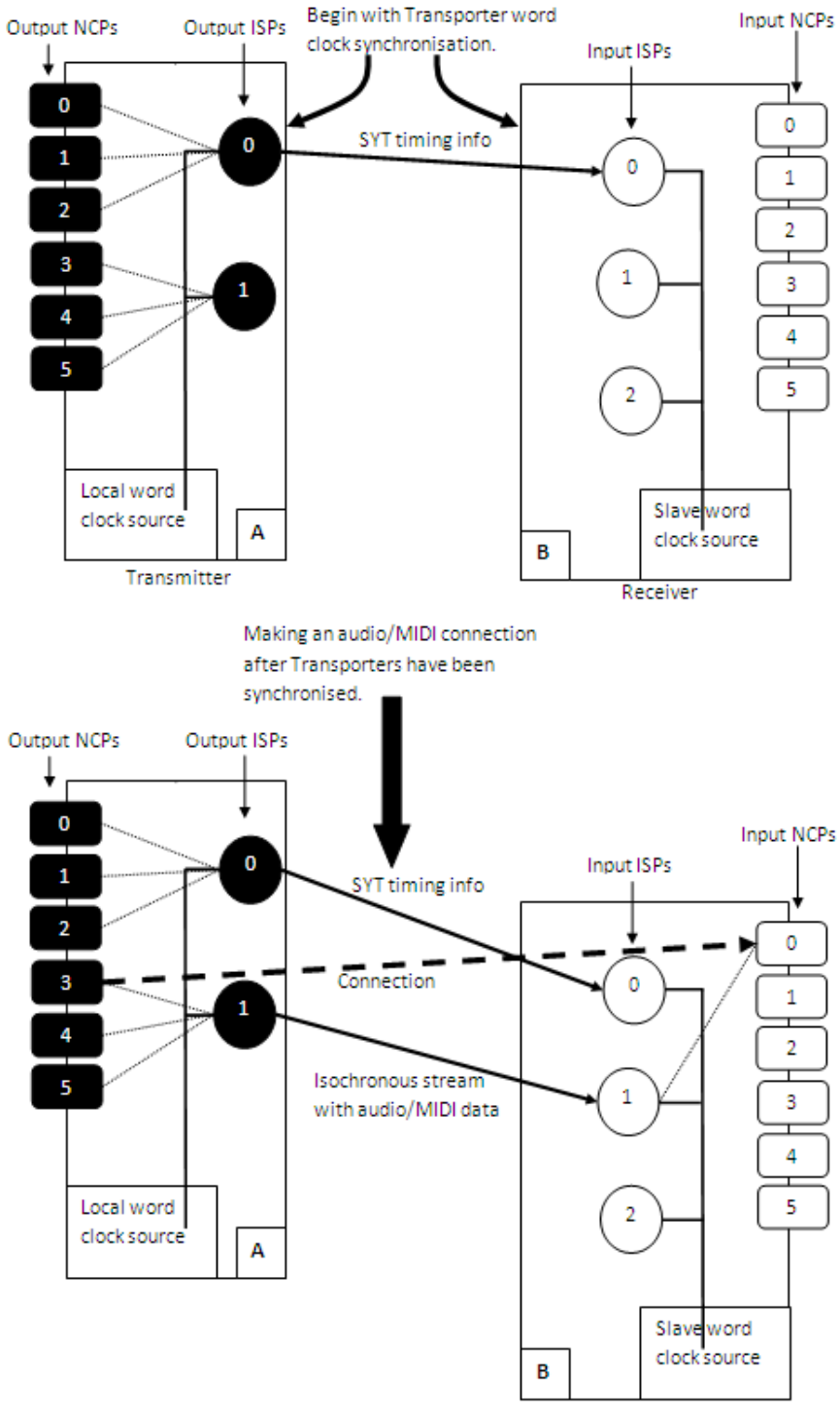


Figure 7.14: Making Connections without Resource Optimisation

ISPs are used, one to receive SYT timing information and another to receive audio/MIDI data. On the transmitter side, isochronous resources are allocated for two ISPs and their associated NCPs. This situation is shown in the bottom half of Figure 7.14.

We have derived an algorithm that optimises resource utilisation by ensuring that, where possible, SYT timing information and audio/MIDI data are transmitted via the same ISP, as shown in Figure 7.15. Our algorithm makes it possible to avoid a situation such as the one shown in

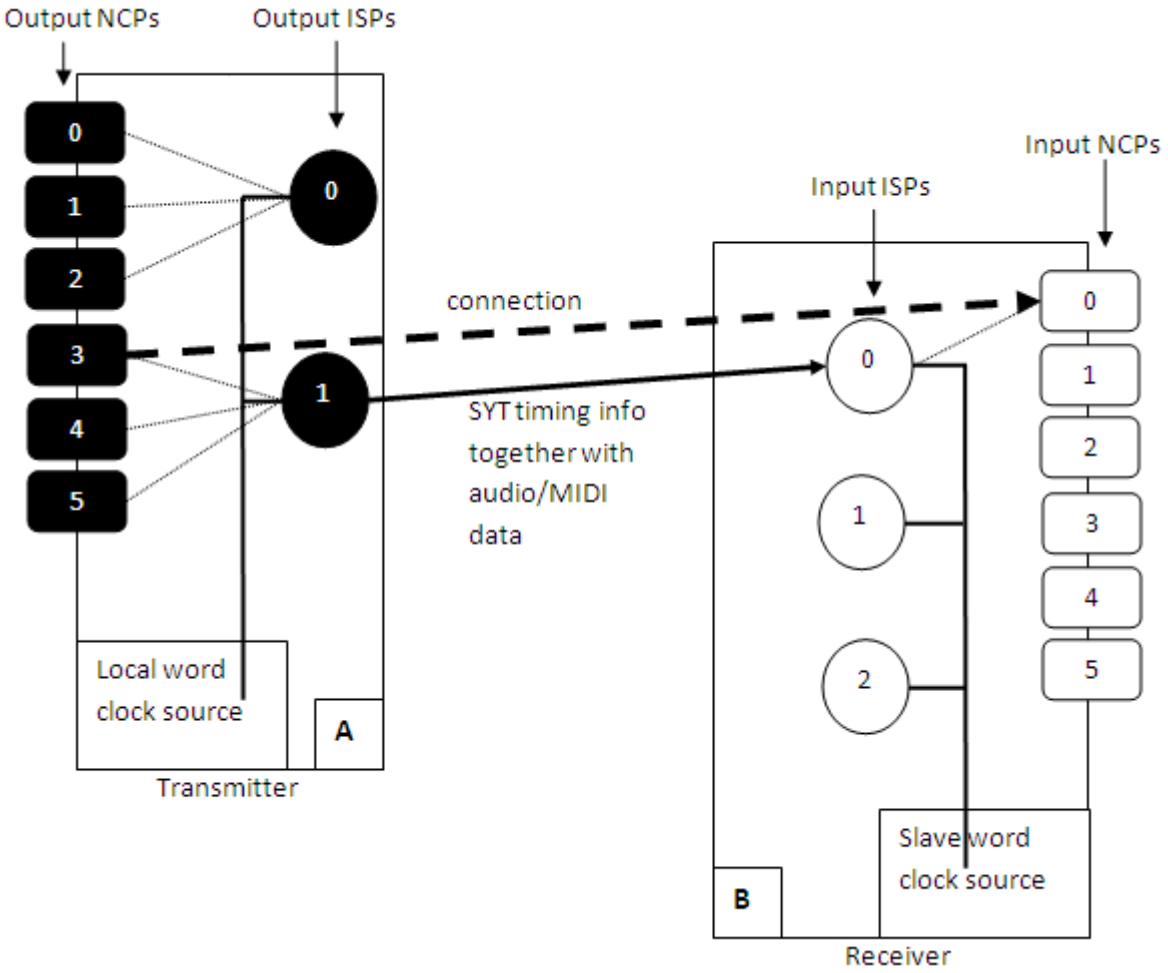


Figure 7.15: Making Connections with Resource Optimisation

the bottom half of Figure 7.14, where two separate isochronous streams are routed to the same receiver. A disadvantage resulting from this situation is that if the receiver has limited receiving capabilities as a result of few destination ISPs, the number of other devices that can potentially stream to the receiver concurrently is reduced. In the context of our example, if we include our optimisation as shown in Figure 7.15, it is possible for two other Transporters to make au-

dio/MIDI data connections to Transporter B, since input ISP numbers 1 and 2 are both available. In contrast, if we do not apply our optimisation techniques, as shown in the bottom half of Figure 7.14, only one other Transporter can make audio/MIDI data connections to Transporter B, since only input ISP number 2 is available. We attribute our resource optimisation mechanism to the capabilities introduced by defining our proposed Transporter HAL API in terms of node controller plugs (NCPs) and isochronous stream plugs (ISPs), where either may be independently controllable, unless stated otherwise.

7.4 Future Work

Section 6.1 on page 115 has mentioned that at the time of this writing, the OGT document did not provide guidelines for node application capabilities. Consequently, our proposed Transporter HAL API currently provides functionality to access the mandatory node controller capabilities of Transporters via the *INC_PLUGIN* COM interface, as described in section 5.2 on page 93. This implies that connection management applications that use our proposed Transporter HAL API, such as the client-server example described in this chapter, can only access the node controller capabilities of Transporters. While it is possible to implement functionality to access the node application capabilities of existing Transporter, such a process is best done after guidelines relating to the design of the node application component of the Open Generic Transporter have been produced. Three further research areas have been identified in this regard, namely:

1. The OGT guidelines that are documented in "Draft AES Standard for Audio over IEEE 1394 - Specification of Open Generic Transporter" [Audio Engineering Society - Standards Committee, 2005] need to be augmented in order to include guidelines relating to the node application component of the Open Generic Transporter.
2. After the OGT guideline document has been augmented, the node application capabilities for both non-OGT-based and OGT-based Transporters need to be evaluated. The evaluation should be aimed at identifying methods that are required to access all node application capabilities for both non-OGT-based and OGT-based Transporters. The methods identified would then become part of the *INA_PLUGIN* COM interface our proposed Transporter HAL API. This *INA_PLUGIN* COM interface accesses the node application capabilities of Transporters, as described in section 5.2 on page 93.
3. After the *INA_PLUGIN* COM interface has been defined, the Enabler should be modified to utilise the node application capabilities of Transporters. Consequently, connection man-

agement applications would require modifications in order to access any additional node application capabilities of Transporters.

Recall from section 7.2.1.4 that while the OGT document defines the capability to use multiple word clock outputs concurrently, there are currently no OGT-based Transporter firmware implementations that implement more than one word clock output. While our proposed Transporter HAL API has been designed with this capability in mind, it would have been better to have an OGT-based Transporter firmware implementation to test this functionality. Three further research areas have been identified in this regard, namely:

1. An OGT-based Transporter firmware implementation that implements more than one word clock output is required.
2. Significant effort is required with regards to eliciting user requirements for the multiple word clock output functionality. Such requirements elicitation will help in gathering of information about how users would expect to interact with an application that has this capability.
3. Once the two problem areas above have been rectified, the implications of the multiple word clock output capability ought to be further investigated. As mentioned in section 4.4.2.4 on page 87, multiple word clock outputs allow Transporters to concurrently transmit or receive audio channels that are sampled at different sampling frequencies. On the other hand, section 7.2.1.3 has mentioned that Plug Layouts may be classified based on the audio plug capabilities at different sampling frequencies, such as low (48 kHz and lower), medium (88.2 kHz and 96 kHz), and high (176.4 kHz and higher) sampling frequencies. With the introduction of multiple word clock outputs, a Transporter is able to use any combination of low, medium, and high sampling frequencies, concurrently. For example, assume that at 48 kHz, a Transporter has 32 usable audio plugs, while the number of usable audio plugs is reduced to 16 at 96 kHz. Also assume that the same Transporter allows for concurrent use of multiple sampling frequencies. Typically there would be a Plug Layout that shows plugs that are usable at 48 kHz, and another that shows plugs that are usable at 96 kHz. However, such classification of Plug Layouts implies that the capability to use multiple sampling frequencies concurrently is never used, since the number of usable audio plugs is subject to the sampling frequency in use. Evidently, the extent to which the use of multiple word clock outputs affects the classification of Plug Layouts by sampling frequency needs to be investigated.

7.5 Chapter Summary

In this chapter we have shown how the capabilities of our proposed Transporter HAL API have resulted in enhancements from Transporter control all the way up to graphical user application capabilities. To demonstrate this, an example of a client-server connection management application that was modified to fully utilise our enhanced Enabler has been described. The enhanced Enabler fully utilises our proposed Transporter HAL API. Important changes that have been made to the client-server XML communication protocol were also discussed. A description of a resource-optimisation algorithm that has been derived from our proposed Transporter HAL API has also been given. The chapter concluded by highlighting further research areas.

The next chapter concludes this thesis. It aims to revisit the context of our investigation, our motivations for the investigation, and reiterate the important findings of our investigation.

Chapter 8

Conclusion

8.1 Current State of Digital Audio Networking

This thesis has indicated that there is readiness for audio networking in audio related environments. In particular, there is a move towards digital audio routing. A number of Ethernet based digital audio network technologies have been identified. Four Ethernet based technologies were discussed in detail, namely CobraNet™ by Cirrus Logic [Cirrus Logic, 2007], EtherSound by Digigram [Digigram, 2006], Livewire™ by Telos Systems (Axia Audio) [Telos Systems/Axia Audio, 2007], and SuperMAC (an implementation of AES50-2005 [Audio Engineering Society, 2005a]) by Sony, Oxford [Oxford Technologies, 2007a]. In addition, an IEEE 1394 (FireWire) based digital audio network technology, mLAN by Yamaha Corporation [Yamaha Corporation, 2007], was described in detail.

Ethernet infrastructure is used widely for data communications, and hence its main advantage relates to the re-use of existing infrastructure at no additional cost. Since Ethernet was originally developed for best effort packet delivery which is suitable for data communications, various vendors have introduced a number of proprietary additions to the technology in order to guarantee the real-time data transmission QoS required in digital audio networks. Despite the cost savings from infrastructure re-use, the majority of Ethernet based digital audio network technologies that exist are proprietary, and hence not interoperable across vendors.

In contrast, IEEE 1394 is based on an open standard which was originally developed for real-time multimedia data transfer, including audio and video [IEEE Std. 1394, 1995; IEEE Std. 1394a, 2000; IEEE Std. 1394b, 2002]. In addition, mLAN makes use of IEEE 1394 as its underlying

network technology and is also based on open standards, namely the Plural Node Architecture [1394 Trade Association, 2004], and the Audio and Music Data Transmission Protocol [1394 Trade Association, 2002; IEC, 2005]. IEEE 1394 requires a specialised network infrastructure that is not as widely used as Ethernet, hence its initial setup costs may be higher. However, results from a survey conducted by the AES TC-NAS show that there is no preference for existing data communications infrastructure over the use of specialised infrastructure for digital audio networking [Gross, 2006]. This thesis has described an investigation of the hardware abstraction layer of second generation mLAN devices. Motivations for further investigation of mLAN that were discussed include its dependence on open standards and the guaranteed real-time data transmission QoS derived from the use of IEEE 1394 in its standard form. Open standards foster interoperability across vendors.

8.2 Second Generation mLAN

The Plural Node Architecture, also known as the Enabler/Transporter Architecture, forms the foundation of second generation mLAN. Audio/MIDI connection management is split between two nodes, namely an Enabler and a Transporter. The Transporter typically resides in an mLAN device and its sole responsibility is the transmission and reception of audio or MIDI over IEEE 1394. The Enabler typically resides in a workstation and exposes high level plugs abstractions to connection management applications. The Enabler configures audio/MIDI routing between Transporters in response to requests from high level connection management applications. Plug-ins are written for the Enabler in order for it to communicate with Transporters in a vendor-specific manner via the Transporters' Transporter Control Interfaces. The plug-ins are written against an mLAN Transporter hardware abstraction layer (HAL) application programming interface (API).

A move towards defining an open, standards-based implementation for the mLAN Transporter has resulted in the introduction of the Open Generic Transporter (OGT) concept. An OGT guideline document has been created by the Audio Engineering Society's SC-02-12-G Task Group [Audio Engineering Society - Standards Committee, 2005]. The document describes an open design for the Transporter Control Interface of a generic Transporter. Therefore, if manufacturers define the Transporter Control Interface of their Transporters in accordance with the OGT guideline document, there will be no need for the manufacturers to create their own vendor-specific HAL plug-ins. Instead, the Enabler will make use of a common, open, generic Transporter HAL

plug-in to communicate with their Transporters.

The mLAN Transporter HAL API mentioned above allows for complete access to the capabilities of Transporters. The investigation documented in this thesis was primarily triggered by the introduction of the OGT guideline document. The OGT guideline document defines all capabilities that may be provided by any Transporter, although in a hardware architecture independent manner. Some capabilities defined by the OGT guideline document are currently not implemented by existing Transporters. This results in inadequacies within the current mLAN Transporter HAL API which has been designed to fully utilise the capabilities of existing Transporters.

Initially, the existing HAL for mLAN Transporters was evaluated. This evaluation identified shortcomings within the current Transporter plug-in mechanism that is used by the Enabler. It also identified shortcomings within the current mLAN Transporter HAL API that is used to create HAL plug-ins which are loaded by the Enabler. To resolve these shortcomings, a new plug-in mechanism and a new Transporter HAL API were proposed. The innovations resulting from our proposed Transporter HAL API were described with the aid of an example of a client-server connection management application and the Enabler. The connection management application fulfils high level connection management requests by calling on the Enabler. A brief overview of important findings that have been documented in this thesis is given below.

8.3 mLAN Transporter Plug-In Mechanism

8.3.1 Current Plug-In Mechanism

The current Transporter plug-in mechanism has been described in the context of its implementation within the Enabler. Two such implementations for different operating systems were described in detail, one for Microsoft Windows and the other for Linux. On Microsoft Windows, the component object model (COM) is used as the plug-in mechanism, while shared libraries are used on Linux. Although not described in detail, Okai-Tettey [2005] suggests that the Macintosh operating system could make use of a shared library approach that is similar to that on Linux, since it is a UNIX-based operating system. Three main shortcomings were identified within the current plug-in mechanism. These shortcomings relate to binary dependence, Transporter HAL API versioning, and the use of forwarding routines to implement Transporter HAL API versioning, as described below.

8.3.1.1 Binary Dependence

When a Transporter is enumerated by the Enabler, its corresponding plug-in is loaded and the plug-in creates a *Transporter object* which the Enabler uses for subsequent interaction with the Transporter. The Enabler is required to keep a pointer to this object in its memory in order to access the parameters of the Transporter. This implies that the definition of the *Transporter* class and its associated classes ought to be shared between the Enabler and the plug-in implementation. Furthermore, important Enabler classes that are associated with the *Transporter* class expose functionality to handle asynchronous transactions and isochronous resource allocation. The above description implies that a certain level of binary dependence exists between the Enabler and HAL plug-in implementations. Thus one has to ensure that the *Transporter* class definition and also those of its associated classes, within both the Enabler and the HAL plug-in, are identical. Such binary dependence is unnecessarily restrictive and can be avoided.

8.3.1.2 Transporter HAL API Versioning

Coupled with the binary dependence shortcoming described above, another shortcoming that was identified was that of Transporter HAL API versioning. In particular, with the current plug-in mechanism, problems may potentially occur upon updates of the Transporter HAL API. The Transporter HAL API defines methods that are required to access parameters of Transporters. For example, methods defined by the *Transporter* class mentioned above are part of the Transporter HAL API. Assuming that there are new function definitions added to the *Transporter* class, backwards compatibility with previous Transporter HAL plug-ins may be problematic. Transporter HAL plug-ins based on earlier versions of the *Transporter* class will most likely produce runtime errors when attempts to access the newly defined member functions are made, since they will be absent. It became apparent that there was a need to query the version of the Transporter HAL API implemented by a HAL plug-in.

The Enabler implementation that formed the basis of our investigation demonstrated this Transporter HAL API versioning shortcoming. Further investigation resulted in a deeper understanding of a mechanism that was originally implemented by Yamaha Corporation in their original Enabler. This mechanism aimed to implement Transporter HAL API versioning through the use of forwarding routines for each Transporter HAL API method, as described in section 8.3.1.3 below. However, the mechanism has its inherent shortcomings.

8.3.1.3 Use of Forwarding Routines to Implement Transporter HAL API Versioning

In an attempt to implement Transporter HAL API versioning, Yamaha Corporation originally designed their Transporter HAL API to have one *CallTransporter* function to access all parameters of a Transporter via its HAL plug-in [Yamaha Corporation, 2004b]. When parameters of a Transporter are accessed via its plug-in, the *CallTransporter* function is called with arguments representing a unique identifier (selector ID) for the function to be called and a handle to the *Transporter object* on which the function is to be called. Each function of the Transporter HAL API has a unique selector ID that is used to identify it. The *CallTransporter* function implementation resides in the HAL plug-in. In this implementation, the selector ID for the function to be called is resolved and, based on the interpretation of the selector ID, an additional call to the actual function is made via the handle to the Transporter's object. Therefore, the *CallTransporter* function serves as a forwarding routine that only forwards function calls to the Transporter object when the selector ID is resolved. Any functions that are not implemented by the HAL plug-in would be known when the selector ID of the function is resolved, and hence there is no possibility for any runtime errors.

While the above mechanism is functional, it was discovered that for Transporter HAL APIs with hundreds of methods, writing forwarding routines for each API method potentially becomes tedious and error prone [Box, 1998]. In particular, for each function that is added to the Transporter HAL API, a new selector ID is also introduced. Care should be taken to ensure that the new selector ID is unique, relative to existing selector IDs. Within a Transporter plug-in, the process of interpreting a selector ID and mapping it to the correct function also requires great care, since the definitions of selector IDs within the Enabler and Transporter plug-ins are required to be exactly the same.

8.3.2 Proposed Plug-In Mechanism

A new plug-in mechanism was proposed to resolved the shortcomings that are summarised in section 8.3.1. The new plug-in design makes use of fundamental concepts of COM. A number of features inherent in COM were found to resolve the shortcomings of the the current plug-in mechanism. Of the three popular operating systems, Microsoft Windows, Linux, and Macintosh, only Microsoft Windows has support for COM in its standard form. Nevertheless, the fundamental concepts of COM made it possible for "COM-like" implementations to be created for Linux and Mac OS X.

On Linux, shared libraries continue to be used for the implementation of our proposed plug-in mechanism. However, the shared library implementation has been tailored to behave like COM. In order to achieve this, guidelines described by Box [1998] were used. As suggested by Okai-Tettey [2005], a shared library implementation for our proposed plug-in mechanism on the Macintosh operating system has been created. However, an alternative implementation for the plug-in mechanism on Macintosh was first considered before the shared library implementation was created. The important concepts of this alternative implementation are highlighted below, together with considerations that make it less favourable.

A mechanism for implementing COM on Mac OS X that has been created by Hunt [2004] was first explored before a decision was made to use a shared library approach for Macintosh. The mechanism for implementing COM on Mac OS X has the potential benefit of making it possible for Microsoft COM plug-in implementations to run on Mac OS X without changing the source code. Despite such a benefit, a number of counter considerations make this alternative less favourable. Some of the considerations include the lack of documentation to show how the mechanism is used on versions of Macintosh other than Mac OS X, the level of complexity introduced for third party plug-in development, and reliance on a single individual (Hunt [2004]) for debugging purposes.

A brief summary of the important aspects of our proposed plug-in mechanism that resolve shortcomings of the current HAL plug-in mechanism is given below.

8.3.2.1 Reducing Binary Dependence

In order to reduce binary dependence between the Enabler and HAL plugs-ins, our proposed HAL plug-in mechanism makes use of pointers to interfaces, as opposed to objects. The advantage of making use of interfaces is that the underlying implementation details are abstracted from plug-in implementations at the time of development. Interfaces were defined for Enabler functionality that are required by HAL plug-ins, such as handling asynchronous transactions and isochronous resource allocation for a Transporter.

8.3.2.2 COM Interface Versioning Capabilities

A COM interface that defines all the mandatory capabilities of the Transporter HAL API was defined. Implementing the Transporter HAL API directly as a COM interface eliminates the need for forwarding routines, hence the COM interface function calls directly access the parameters

of a Transporter. A guiding principle that was complied with was that COM interfaces ought to be treated as immutable binary and semantic contracts that must never change [Box, 1998]. Assuming that new capabilities are required in the Transporter HAL API, other suitably named COM interfaces with the required function definitions may be created. This implies that it is possible for the Transporter HAL API to be composed of more than one COM interface.

A feature of COM that facilitates Transporter HAL API versioning through the use of multiple COM interfaces is the ability to query the existence of one or more COM interfaces via another COM interface. In particular, we have defined a mandatory COM interface called *INC_PLUGIN* which all HAL plug-ins should implement. The *INC_PLUGIN* COM interface defines methods that access the node controller capabilities of a Transporter. Any additional or optional COM interfaces are queried via the *INC_PLUGIN* COM interface. In order to query the existence of COM interfaces, the *QueryInterface* function of the *IUnknown* COM interface is called. The *IUnknown* COM interface is the parent of all legal COM interfaces. Any COM interface directly or indirectly provides an implementation for the *IUnknown* COM interface functionality such as the *QueryInterface* function. Other functions defined by the *IUnknown* COM interface assist with managing the lifetime of an instance of a COM component.

The *QueryInterface* function is called with a unique identifier or name of the COM interface being queried. If the COM interface, *INC_PLUGIN* for example, does not provide an implementation for the COM interface being queried, an error is reported, otherwise a pointer to the queried COM interface is returned to the caller. This query interface mechanism allows for greater control over Transporter HAL API versioning. COM interfaces are first queried to ascertain their existence within a given plug-in before functions are called. This prevents the occurrence of runtime errors that are due to the absence of expected functions. Consequently, this mechanism reinforced the suitability of COM, in its standard form, as a plug-in mechanism that resolved the shortcomings described in section 8.3.1.

8.4 mLAN Transporter HAL API

As already mentioned, the introduction of the Open Generic Transporter concept was the primary element that triggered this investigation. In order for the OGT guideline document to be hardware architecture independent, the components of a stream of isochronous packets on the IEEE 1394 bus are defined using high level abstractions. These abstractions relate to isochronous stream plugs (ISPs) and node controller plugs (NCPs). An ISP represents an input or output for a single

isochronous stream to or from the IEEE 1394 bus, while an NCP represents an input or output for a single monaural channel of audio or a single cable of MIDI to or from the host device [Audio Engineering Society - Standards Committee, 2005]. An ISP can be dynamically or statically associated with one or more NCPs.

The manner in which word clock synchronisation is defined by the OGT guideline document abstracts underlying hardware-architecture-specific features, although sufficiently scalable to incorporate a future need, such as the ability for a Transporter to make use of multiple sampling frequencies simultaneously. In addition, the synchronisation model defines a mechanism that does not impose restrictions on the number and types of word clock sources that a Transporter can use to generate word clock outputs.

The OGT guideline document also provides a structured mechanism to deal with any device-specific quirks. This mechanism involves the use Plug Layouts to model the different modes that a Transporter can operate in. This implies that Plug Layouts are mutually exclusive configurations of a Transporter, where each Plug Layout has a fixed number of NCPs, ISPs, and word clock sources that are concurrently usable.

8.4.1 Current Transporter HAL API

The current Transporter HAL API was defined before the OGT concept was introduced. Consequently, this version of the Transporter HAL API completely utilises the capabilities of Transporters that were created before the OGT. When the OGT guideline document was produced, a HAL plug-in for OGT-based Transporters was created against the current Transporter HAL API. This implementation of the OGT HAL plug-in was evaluated to determine the extent to which capabilities of the OGT were utilised. An overview of the shortcomings that were identified is given below.

8.4.1.1 Implementation Complexities

As a result of the static nature of the current Transporter HAL API, complexities have been introduced to the OGT plug-in implementation in order to accommodate dynamic ISP to NCP associations. This is due to the fact that the current Transporter HAL API assumes a static relationship between the components that model a stream of isochronous packets, while the OGT guidelines allows for the possibility of a dynamic relationship between ISPs and NCPs. It was also noted that the current Transporter HAL API is not defined in terms of ISPs and NCPs.

8.4.1.2 Restrictions In Transporter Operation

The current Transporter HAL API was designed with a bias towards fully utilising the capabilities of Yamaha Corporation's hardware, although HAL plug-ins enable cross vendor interoperability. Certain assumptions within the current Transporter HAL API impose restrictions on the operation of OGT-based Transporters. An example of one such restriction is the manner in which ISPs of the OGT cannot be individually controlled. In the context of transmitting ISPs, such collective control leads to wastage of isochronous resources on a network. Isochronous resources available on a network determine the number of Transporters that can transmit data simultaneously, hence such wastage has negative consequences on the overall performance of an mLAN network.

8.4.1.3 Inadequate Word Clock Source Selection Capabilities

In terms of the current Transporter HAL API, only two types of word clock sources may be selected, namely a built-in word clock source and a word clock source that receives synchronisation information from the IEEE 1394 bus. On the contrary, the OGT guideline document does not impose any restriction on the number and types or word clock sources that may be selected for Transporters. For example, OGT-based Transporters may have external word clock sources attached to them such as by reading the sample rate information of incoming ADAT or AES/EBU digital audio streams. The current Transporter HAL API does will not allow such external word clock sources to be selected for any Transporter. This leads to the underutilisation of word clock source selection capabilities of OGT-based Transporters.

8.4.1.4 Lack of Support for the Use Multiple Sampling Frequencies Concurrently

The current Transporter HAL API assumes that all audio plugs of a Transporter are only allowed to operate at one common sampling frequency. However, the OGT guideline document caters for a future need where plugs of a Transporter may operate at different sampling frequencies. For example, one source plug on a Transporter can operate at 48 kHz while another at 96 kHz. Such a future need cannot be realised using the current Transporter HAL API since it assumes that all plugs on a device can only operate at one common sample rate. It should be borne in mind that while plugs on a Transporter may operate at different sampling frequencies, connections between source and destination plugs are only permitted if both plugs share the same sampling frequency.

8.4.1.5 Inadequate Control over Device-Specific Quirks

The is currently no mechanism that explicitly handles device-specific quirks. For example, a common quirk is the reduction in the number of audio channels that can be transmitted at high sampling frequencies as opposed to lower sampling frequencies. In an example of a current plug-in implementation, the number of audio channels that are transmitted by a Transporter is halved for sampling frequencies greater than 48 kHz. For users of connection management applications who do not have prior knowledge regarding this, it results in undesirable "surprises".

It became apparent that sampling frequency changes are the only quirk that can be handled by the current Transporter HAL API. However, a manufacturer may opt to create an OGT-based Transporter that has different configurations (Plug Layouts) based on a variety of bandwidth requirements. With the current Transporter HAL API, it would be impossible for the Enabler to have optimal control over such quirks, since the API has no knowledge of Plug Layouts in an OGT-based Transporter. Consequently, a need for a more structured way of dealing with any device-specific quirks was identified.

8.4.2 Proposed Transporter HAL API

A new Transporter HAL API was proposed to resolve the shortcomings mentioned above. To demonstrate the feasibility of our proposed Transporter HAL API in resolving these shortcomings whilst remaining backwards compatible, three proof-of-concept HAL implementations were created. These HAL implementations comprised two standalone device Transporter plug-ins and an implementation for a Windows PC Transporter. Of the two standalone device Transporter plug-ins, one is for an example of a type of non-OGT-based Transporter, while the other is for OGT-based Transporters. Important concepts of these implementations have been highlighted in this thesis.

This thesis has also described a test application that was created to manually test each of the Transporter HAL API functions implemented for both non-OGT-based Transporters and OGT-based Transporters. The impact that our proposed Transporter HAL API has on a client-server connection management application has also been discussed. In this connection management application, the server makes use of our modified version of the Enabler to fulfil connection management requests from client applications. The client applications allow for graphical user interaction. Some functionality introduced by the proposed Transporter HAL API resulted in visible enhancements to end-user capabilities of client applications [Chigwamba and Foss, 2007],

while others resulted in more subtle enhancements, such as the possibility for resource optimisation techniques. A brief overview of the key aspects of our proposed Transporter HAL API is given below.

8.4.2.1 ISPs and NCPs as a Basis for Transporter HAL API

Implementation complexities and operational restrictions mentioned in sections 8.4.1.1 and 8.4.1.2 respectively, result from the fact that the current Transporter HAL API is not defined in terms of ISPs and NCPs. Consequently, in order to resolve these shortcomings, our proposed Transporter HAL API was defined in terms of ISPs and NCPs.

The concept of ISPs and NCPs was introduced by the OGT guideline document. However, this concept was found to be abstract enough to encapsulate the underlying components of isochronous streams in the context of the Enabler/Transporter architecture. The proof-of-concept implementations that were created, show how this concept can be used to model the functional blocks within non-OGT-based Transporters without compromising Transporter capabilities.

8.4.2.2 Specifying Constraints to Allow Dependencies Between Transporter Parameters

In order to accommodate both static and dynamic behaviours of components of various Transporters such as associations between ISPs and NCPs, various parameters of a Transporter are associated with dependency information. This dependency information specifies whether values of parameters are fixed, meaning that they cannot be changed, or the parameters belong to a group of parameters whose values may be dependent on each other. This information is conveyed via some of the functions defined within our proposed Transporter HAL API. The use of dependency information contributes to the design of a Transporter HAL API that is flexible and is not biased towards a particular hardware architecture's capabilities.

8.4.2.3 Enhanced Word Clock Source Selection

An enhanced word clock source selection mechanism was introduced by the proposed Transporter HAL API. This mechanism was adopted from the OGT guideline document, since it does not impose restrictions in either the number or types of word clock sources that may be accessed by an Enabler. As a result, external word clock sources that are implemented by some OGT-based Transporters can now be selected. The word clock source selection mechanism allows

the number of word clock sources on a Transporter to be queried and each word clock source is accessible using a unique identifier for the word clock source.

While this mechanism was adopted from the the OGT guideline document, our proof-of-concept implementations showed it that can be implemented for existing Transporters.

8.4.2.4 Support for the Use of Multiple Sampling Frequencies Concurrently

In recognition of the future need for a Transporter to operate at multiple sampling frequencies as suggested by the OGT guideline document, our proposed Transporter HAL API attempts to incorporate this capability. However, there are currently no OGT-based Transporter firmware implementations that make use of this capability. Hence, this functionality cannot be tested completely at present. However, absence of such firmware implementations creates potential research areas where future work may be carried out.

8.4.2.5 Use of Plug Layouts to Handle Device-Specific Quirks

The Plug Layout concept that is defined by the OGT guidelines has been incorporated into the proposed Transporter HAL API. Existing sampling frequency device-specific quirks are now implemented in terms of the Plug Layout concept. It has also been shown that the Plug Layout concept provides a structured approach to handling any device-specific quirks.

8.5 Future Work

Throughout this thesis, it has been constantly mentioned that at the time of this writing, the OGT document did not provide any guidelines relating to the node application capabilities of the Open Generic Transporter. Consequently, our proposed Transporter HAL API only provides functionality to access the mandatory node controller capabilities of Transporters. In this regard, further work is required in order to incorporate node application capabilities within our proposed Transporter HAL API. In particular, the OGT document first needs to be augmented with guidelines for the design of the node application component of the Open Generic Transporter. Secondly, some investigation will be required in order to determine the methods that are required to access the node application capabilities of Transporters using our proposed Transporter HAL API. Lastly, the Enabler and its associated connection management applications will require modifications in order to fully utilise any additional node application capabilities.

The absence of an OGT-based Transporter firmware implementation that supports concurrent use of multiple sampling frequencies implies that this functionality cannot be tested at present. Consequently, this creates an investigation area where further research needs to be done. In particular, this functionality is first required at a firmware level. Should the firmware implementation be provided, more investigation is required in order to elicit user requirements. User requirements will mainly guide the graphical user interface design.

We have already mentioned that should the number of usable plugs differ depending on sampling frequency, a Transporter would have to implement separate "modes", where each mode has a separate Plug Layout which only exposes plugs that are visible within a particular sampling frequency range. In addition, one of the capabilities introduced by the OGT concept, which we have also incorporated in the Transporter HAL API, is the ability for a Transporter to use multiple sampling frequencies concurrently. This leads to a need for further investigation in order to find out how to handle situations where two or more sampling frequencies that have different plug capabilities are used concurrently. For example, assume that at 48 kHz, a Transporter has 32 usable plugs, while the number of usable plugs is reduced to 16 at 96 kHz. Also assume that the same Transporter allows for concurrent use of multiple sampling frequencies. If Plug Layouts are classified based on the various plug capabilities that are available at different sampling frequencies, the capability to use multiple sampling frequencies concurrently is never used. Therefore, a question that needs to be answered is: "To what extent is the classification of Plug Layouts, based on plug capabilities at different sampling frequencies, affected by the capability to use multiple sampling frequencies concurrently?"

8.6 Overall Conclusion

One of the favourable motivations for further investigating the Plural Node Architecture was the need to promote an open, standards-based digital audio network technology. The introduction of the Open Generic Transporter reaffirms the desire for an open, standards-based audio network.

Our investigation has shown how the fundamental concepts of COM can be used to create a plug-in mechanism that allows for robust Transporter HAL API versioning. In addition, we have shown that a new Transporter HAL API is required in order to fully utilise capabilities of the Open Generic Transporter. We acknowledge that the OGT guideline document is hardware architecture independent and completely describes audio and music data encapsulation and extraction in accordance with the A/M Data Transmission Protocol. Consequently, our proposed

Transporter HAL API has been largely influenced by the high level abstractions that are defined in the OGT guideline document.

Enhancements to end-user capabilities have also been realised as a result of our proposed Transporter HAL API. However, inadequacies in the capabilities of current firmware implementations for OGT-based Transporters warrant further investigation in some areas.

Glossary

"B Mode" Transporter: A Transporter that always requires an Enabler for A/M data transmission.

"B-Pro Mode" Transporter: A Transporter that is capable of CIT (see CIT).

A/M: Audio and Music.

AES TC-NAS: The Audio Engineering Society's Technical Committee on Networked Audio Systems.

AM824: A data format that comprises an 8-bit label and 24-bits of Audio/Music data.

API: Application Programming Interface.

ASIC: Application Specific Integrated Circuit.

ASIO: Audio Stream Input/Output.

Asynchronous packet: See ansynchronous transfers.

Asynchronous transfers: The class of IEEE 1394 data transfers that require guaranteed data delivery. Such data include device control requests and is packaged in asynchronous packets.

Basic Enabler: Yamaha Corporation's original Enabler implementation (see Enabler).

Bus reset: Occurs whenever the state of any IEEE 1394 node changes (including addition and removal of nodes).

CIP: Common Isochronous Packet. The general packet format for audiovisual data over IEEE 1394.

CIT: Connectionless Isochronous Transmission. In CIT, a Transporter broadcasts or receives A/M data on isochronous channels that are restored from the device's non-volatile memory.

CLSID (CLasS ID): A number that uniquely identifies a COM component. Applications that support Microsoft's COM architecture register their components' CLSIDs in the Windows registry. Windows uses these CLSIDs to identify software components without having to know their name.

COM: Component Object Model.

config ROM: Sets of read-only memory entries are defined to provide configuration information during node initialisation.

CSR: Control and Status Register.

Data block: A cluster within an isochronous packet. See isochronous packet.

DLL: Dynamic Link Library. A DLL is a code module for Microsoft Windows that can be loaded on demand and linked at run time, and then unloaded when the code is no longer needed.

Enabler: A node in the Plural Node Architecture that provides high level abstractions of the audio channel end points on each of the Plural Node audio devices. This node is responsible for making and breaking connections between these end-points.

FireWire: See IEEE 1394.

Function prototype: A declaration of a function, in C or C++, that omits the function body but specifies the function's name, argument types, and return type. A function prototype can be thought of as specifying a function's interface.

GUID: Globally Unique IDentifier. A unique 128-bit number assigned to each IEEE 1394 device.

HAL: Hardware Abstraction Layer.

HAL ID: A unique identifier for a Transporter's HAL plug-in. This identifier is composed of the Transporter's HAL Vendor ID and HAL Model ID. These two IDs are found in the host device's config ROM.

IEC: International Electrotechnical Commission. An organisation that sets international electrical and electronics standards.

IEC 61883-6: The Audio and Music Data Transmission Protocol.

IEEE 1394: A digital network interface technology that allows professional audio and video equipment, PCs, and electronic devices to be interconnected using a single cable. Commonly known as FireWire.

IID: Interface IDentifier. A globally unique identifier (GUID) that is associated with an interface. There are functions that take IIDs as parameters in order to allow the caller to specify the interface pointer to be returned.

Isochronous packet: See isochronous transfers.

Isochronous stream: A collection of isochronous packets with the same channel number.

Isochronous transfers: The class of IEEE 1394 data transfers that require a constant rate of data transmission. Such data include audio and video content, and is packaged in isochronous packets.

ISP: Isochronous Stream Plug. Represents an input or output for a single isochronous stream to or from the IEEE 1394 bus.

mLAN: music Local Area Network.

NCP: Node Controller Plug. Represents an input or output for a single monaural channel of audio or a single cable of MIDI to or from the node application (see node application).

NCP05: Node Controller Package 5. A Transporter design makes use of one mLAN-NC1 chip and two mLAN-PH2 chips.

Node application: The hosting IEEE 1394 device and its range of legacy audio and MIDI plugs.

Node controller: The IEEE 1394 node that is hosted by a device.

OGT: Open Generic Transporter.

OSI: Open Systems Interconnection. An ISO standard for worldwide communications that defines a framework for implementing data networking protocols in seven layers.

Patch bay: A central routing device or software application which facilitates the connection of outputs of audio devices to inputs of other audio devices.

PC Transporter: An implementation of a Transporter (see Transporter) on workstation that is equiped with an IEC 61883-6 capable driver.

PCM: Pulse-code modulation.

Plug Layout: A mutually exclusive Transporter configuration that contains a number of plugs (NCPs and ISPs) and word clock sources that are concurrently usable.

Power cycle: To power off a device and then power it on again.

QoS: Quality of service.

Quadlet: A 32-bit data element.

Redesigned Enabler: A modified version of Yamaha Corporation's original Enabler implementation.

Sequence: A position within an isochronous packet cluster.

SFC: Sampling frequency.

Structured cabling: Cabling infrastructure that consists of a number of standardised smaller elements.

Structured data cable: See structured cabling.

SYT: A field of a CIP packet whose value indicates the time at which a particular data block (event) within the isochronous packet should be presented at the receiver.

SYT phase: Refers to the phase difference between the SYTs of the stream received by an ISP and the ISP's word clock (see SYT).

Transporter: A node in the Plural Node Architecture that is solely responsible for the encapsulation and extraction of audio and MIDI data in a manner that complies with the A/M Data Transmission Protocol.

VCXO: Voltage controlled crystal oscillator.

WDM: Windows Driver Model.

Word clock: A clock signal used to synchronise other devices.

References

1394 Trade Association. *"TA Document 2001024: Audio and Music Data Transmission Protocol 2.1 "*, 2002.

1394 Trade Association. *"TA Document 1999025: AV/C Descriptor Mechanism 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. *"Working Draft: Plural Node Implementation of a Professional A/M Device"*, 2004.

Don Anderson. *"FireWire System Architecture"*. Mindshare Inc., New Jersey, 2nd edition, 1999.

Apple Computer, Inc. *"Core Foundation Framework Reference"*, 2006a.

Apple Computer, Inc. *"Dynamic Library Programming Topics"*, 2006b.

Apple Computer, Inc. *"FireWire Device Interface Guide"*, 2006c.

Apple Computer, Inc. *"Porting UNIX/Linux Applications to MAC OS X"*, 2006d.

Tom Armstrong and Ron Patton. *"ATL Developer's Guide"*. IDG Books Worlwide, Inc., Foster City, California, 2nd edition, 2000.

Audinate. *"Technology Overview"*. 2007. [online] Available at: `http://www.audinate.com/techoverview.shtml`. [Accessed 11 June 2007].

Audio Engineering Society. *"AES3-2003: REVISED AES Standard for Digital Audio - Digital Input-Output Interfacing - Serial Transmission Format for Two-Channel Linearly Represented Digital Audio Data"*, 2003.

Audio Engineering Society. *"AES47-2006: AES Standard for Digital Audio - Digital Input-Output Interfacing - Transmission of Digital Audio over Asynchronous Transfer Mode (ATM) Networks"*, 2006.

Audio Engineering Society. *"AES50-2005: AES Standard for Digital Audio Engineering - High-Resolution Multi-Channel Audio Interconnection (HRMAI)"*, 2005a.

Audio Engineering Society. *"AES-R6-2005: AES Project Report - Guidelines for AES Standard for Digital Audio Engineering - High-Resolution Multi-Channel Audio Interconnection (HRMAI), AES50"*, 2005b.

Audio Engineering Society - Standards Committee. *"Draft AES Standard for Audio over IEEE 1394 - Specification of Open Generic Transporter"*. Audio Engineering Society, 2005.

Aviom, Inc. *"Pro64 Audio Networking"*. 2007. [online] Available at: `http://www.aviom.com/LibraryDocs/Brochures/Pro64OverviewBrochure.pdf`. [Accessed 11 June 2007].

Ray Bayburn. *"Bundle Assignments in CobraNet Systems"*. Cirrus Logic, Inc., 2004. [online] Available at: `http://www.cobranet.info/en/pubs/appNote/CobraNet_BundleAssignments.pdf`. [Accessed 12 June 2007].

Don Box. *"Essential COM"*. Addison Wesley, Massachusetts, 1998.

Nyasha Chigwamba and Richard Foss. *"Enhancing End-User Capabilities in High Speed Audio Networks"*. Presented at the 123rd Audio Engineering Society Convention, New York, 2007.

Cirrus Logic. *"Networked Digital Audio Technology"*. 2007. [online] Available at: `http://www.cobranet.info/en/pubs/proBulletin/CobraNet_Networked_Digital_Audio_Technology.pdf`. [Accessed 19 December 2007].

Cirrus Logic. *"CobraNet Programmer's Reference"*, 2006. [online] Available at: `http://cobranet.info/en/pubs/manual/CobraNet_Programmer_Manual_PM25.pdf`. [Accessed 12 June 2007].

Digigram. *"Overview - An Introduction to the ES-100 Technology"*, 2006. [online] Available at: `http://www.ethersound.com`. [Accessed 6 June 2007].

Digigram. *"Digigram Products - Networked Audio Devices and Technologies"*, 2007. [online] Available at: `http://www.digigram.com/products/`. [Accessed 14 June 2007].

D&R Electronica. *"CobraNet Manager"*, 2007. [online] Available at: `http://www.cobranetmanager.com`. [Accessed 12 June 2007].

Richard Foss, Jun-ichi Fujimori, Nyasha Chigwamba, Bradley Klinkradt, and Harold Okai-Tettey. *"Flexible High Speed Audio Networking for Hotels and Convention Centres"*. Presented at the 120th Audio Engineering Society Convention, Paris, 2006.

Jun-ichi Fujimori and Richard Foss. *"A New Connection Management Architecture for the Next Generation of mLAN"*. Presented at the 114th Audio Engineering Society Convention, Amsterdam, 2003.

Jun-ichi Fujimori, Richard Foss, Bradley Klinkradt, and Shaun Bangay. *"An mLAN Connection Management Server for Web-Based, Multi-User, Audio Device Patching"*. Presented at the 115th Audio Engineering Society Convention, New York, 2003.

Wilbert O. Galitz. *"The Essential Guide to User Interface Design - An Introduction to GUI Design Principles and Techniques"*. John Wiley & Sons, Inc., Canada, 2002.

Gibson Musical Instruments. *"Media-Accelerated Global Information Carrier (MAGIC): Engineering Specification"*. 2003. [online] Available at: `http://www.gibson.com/files/_audio/magic/magic3_0c.pdf`. [Accessed 19 December 2007].

Steve Gray. *"CobraNet Routing Primer"*. Cirrus Logic, Inc., 2004. [online] Available at: `http://www.cobranet.info/en/pubs/appNote/CobraNet_AudioRoutingPrimer.pdf`. [Accessed 12 June 2007].

Kevin Gross. *"Audio Networking: Application and Requirements"*. Journal of the Audio Engineering Society, Volume 54(Number 1/2):62-66, January/February, 2006.

Hear Technologies. *"Hear Back System"*. 2007. [online] Available at: `http://www.heartechnologies.com/hb/hearbacksystem.htm`. [Accessed 11 June 2007].

Terry Holton. *"mLAN - Time to Pay Attention"*. 2003. [online] Available at: `http://www.mlancentral.com/mlan_info/mLAN_resolution_v24.pdf`. [Accessed 25 June 2007].

Chrisopher Hunt. *"Component Object Model (COM) Development on Mac OS X"*. 2004. [online] Available at: `http://www.macdevcenter.com/lpt/a/4774`. [Accessed 2 March 2007].

International Electrotechnical Commission - IEC. *"ISO/IEC 13213: Information Technology - Microprocessor Systems - Control and Status Registers (CSR) Architecture for Microcomputer Buses"*. IEC, 1994.

International Electrotechnical Commission - IEC. *"IEC 61883-1: Consumer Audio/Video Equipment - Digital Interface - Part 1: General "*. IEC, 2003.

International Electrotechnical Commission - IEC. *"IEC 61883-6: Consumer Audio/Video Equipment - Digital Interface - Part 6: Audio and Music Data Transmission Protocol"*. IEC, 2nd edition, 2005.

Institute of Electrical and Electronics Engineers - IEEE. *"IEEE Standard for High Performance Serial Bus Bridges"*. IEEE, 2005.

IEEE P1394c Working Group. *"IEEE p1394c: 1394 with 1000BASE-T PHY Technology"*, 2007. [online] Available at: `http://grouper.ieee.org/groups/1394/c/1394cIntroKevinBrown.pdf`. [Accessed 19 June 2007].

IEEE Std. 1394. *"Standard for a High Performance Serial Bus"*, 1995.

IEEE Std. 1394a. *"Standard for a High Performance Serial Bus - Amendment 1"*, 2000.

IEEE Std. 1394b. *"Standard for a High Performance Serial Bus - Amendment 2"*, 2002.

Intelligent Media Technologies, Inc. *"SmartBuss - High Performance Private Multimedia Network"*. 2007. [online] Available at: `http://www.intelligentmedia.us/pdf/Overview2007.pdf`. [Accessed 11 June 2007].

Ken Kounosu, Jun-ichi Fujimori, Rob Laubscher, and Richard Foss. *"An Open Generic Transporter Specification for the Plural Node Architecture of Professional Audio Devices"*. Presented at the 118th Audio Engineering Society Convention, Barcelona, 2005.

Linux 1394. *"IEEE 1394 for Linux"*. 2006. [online] Available at: `http://www.linux1394.org`. [Accessed 10 February 2007].

Gary R. McClain. *"Open Systems Interconnection Handbook"*. Multiscience Press, Inc., New York, 1991.

Microsoft Developer Network. *"CoCreateInstance Function"*. 2007a. [online] Available at: `http://msdn2.microsoft.com/en-us/library/ms686615.aspx`. [Accessed 31 May 2007].

Microsoft Developer Network. *"CoGetClassObject Function"*. 2007b. [online] Available at: `http://msdn2.microsoft.com/en-us/library/ms684007.aspx`. [Accessed 22 August 2007].

Mark Mitchell, Jeffrey Oldham, and Alex Samuel. *"Advanced Linux Programming"*. New Riders Publishing, Indianapolis, 2001.

National ICT Austria. *"Audinate"*. 2007. [online] Available at: `http://nicta.com.au/business/success_stories/audinate`. [Accessed 11 June 2007].

Networked Audio Solutions. *"mLAN Connection Management Server - Client-Server Communication 0.0.4"*, 2006. Confidential.

Nine Tiles Networks Ltd. *"Audiolink 2"*, 2007. [online] Available at: `http://www.ninetiles.com/Audiolink2.htm`. [Accessed 19 June 2007].

Harold Okai-Tettey. *"High Speed End-To-End Connection Management in a Bridged IEEE 1394 Network of Professional Audio Devices"*. PhD thesis, Rhodes University, Grahamstown, 2005.

Harold Okai-Tettey. *"Enabler Communication Model"*. May 2007. Personal Communication.

Oxford Technologies. *"Digital Audio Interconnection"*. 2007a. [online] Available at: `http://www.sonyoxford.co.uk/pub/supermac/download/xmac_brochure.pdf`. [Accessed 6 June 2007].

Oxford Technologies. *"SuperMAC Technology Specification"*. 2007b. [online] Available at: `http://www.sonyoxford.co.uk/pub/supermac/tech.html`. [Accessed 19 June 2007].

Roland Systems Group. *"Roland Ethernet Audio Communication (REAC) Cakewalk SONAR Beta Test Program"*. 2007. [online] Available at: `http://www.rolandsystemsgroup.com`. [Accessed 6 June 2007].

Francis Rumsey and John Watkinson. *"The Digital Interface Handbook"*. Focal Press, London, 1993.

TC Applied Technologies. *"Specification: DICE (Digital Interface Communications Engine) - User Guide"*.

Telos Systems/Axia Audio. *"Axia Professional Networked Audio"*. 2007. [online] Available at: `http://www.axiaaudio.com/brochures/axia_2-5-07_screen.pdf`. [Accessed 14 June 2007].

Telos Systems/Axia Audio. *"Introduction to Livewire - System Design Reference & Primer"*. 2004. [online] Available at: `http://www.axiaaudio.com/manuals/files/Axia_IntrotoLW_v1.0.pdf`. [Accessed 14 June 2007].

Andrew W. Troelsen. *"Developer's Workshop to COM and ATL 3.0"*. Wordware Publishing, Inc., Plano, Texas, 2000.

John Watkinson. *"The Art of Digital Audio"*. Focal Press, London, 3rd edition, 2001.

Yamaha Corporation. *"mLAN-1.0 B:mLAN Specification"*, 2002a. Confidential.

Yamaha Corporation. *"mLAN Enabler Software Specification"*, 2004a. Confidential.

Yamaha Corporation. *"NCP04/05 Transporter Specifications"*, 2002b. Confidential.

Yamaha Corporation. *"mLAN-1.0 Transporter Specification"*, 2002c. Confidential.

Yamaha Corporation. *"mLAN-NC1 PH1 Block Specification"*, 2001. Confidential.

Yamaha Corporation. *"mLAN-PH2 (YTS440-F) Specification"*, 2003a. Confidential.

Yamaha Corporation. *"Guide for Bandwidth Efficiency"*, 2002d.

Yamaha Corporation. *"mLAN-2.0 Configuration ROM specification"*, 2003b. Confidential.

Yamaha Corporation. *"mLAN Networking"*. 2007. [online] Available at: `http://www.yamahasynth.com/products/mlan/index.html`. [Accessed 6 June 2007].

Yamaha Corporation. *"mLAN Transporter Plug-In Mechanism"*, 2004b. Confidential.

Yamaha Corporaton. *"MAP4 User's Manual"*, 2003. Confidential.

Appendix A

Current Transporter HAL API

Figure A.1 shows the object model for the current Hardware Abstraction Layer (HAL) of the Redesigned Enabler. The Redesigned Enabler is a version of an Enabler that has been created

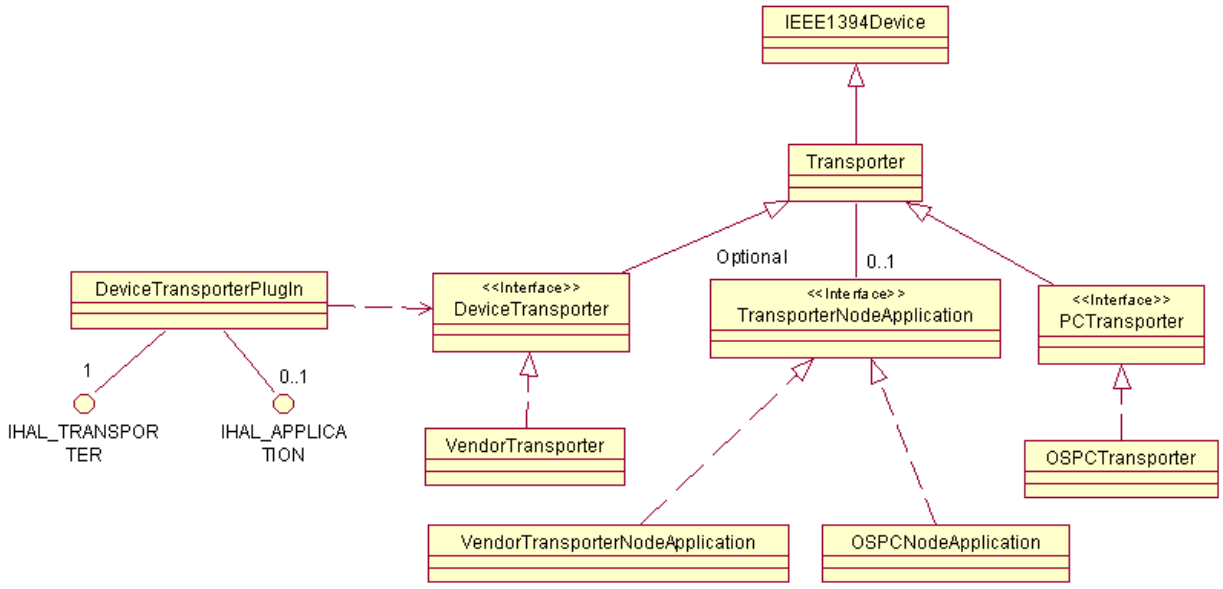


Figure A.1: Current Redesigned Enabler Hardware Abstraction Layer Object Model

by Okai-Tettey [2005]. This Enabler is based on Yamaha Corporation's original implementation which is known as the Basic Enabler. The design of the Basic Enabler is documented in the "mLAN Enabler Software Specification" [Yamaha Corporation, 2004a]. In this appendix, we describe the capabilities of the important classes that compose the current Transporter HAL API of the Redesigned Enabler. With reference to Figure A.1, these classes are the *Transporter*,

DeviceTransporter, and *PCTransporter* class.

A.1 *Transporter* Class Definition

The *Transporter* class is an abstract class which represents IEEE 1394 audio devices (Transporters) that support transmission and reception of isochronous streams in terms of the "A/M Data Transmission Protocol" [1394 Trade Association, 2002; IEC, 2005]. This class defines methods that are required to access capabilities that are common to all Transporters, hence the class name *Transporter*. These methods, together with short descriptions of their purposes, are listed below:

1. *GetNameString(UInt32 strType, UInt8 *strName)*
2. *SetNameString(UInt32 strType, const UInt8 *strName)*
   Different components of a Transporter are associated with textual names. These names include: module vendor name, module model name, node vendor name, node model name, Transporter nickname, and firmware version name. All of these names, with the exception of the Transporter nickname, are read only. The *GetNameString* method is used to retrieve a textual name, depending on the value of *strType*, while the *SetNameString* method modifies the textual name, if possible.

3. *GetMaxSpeed(UInt32 *pSpeed)*
   This method is used to query the maximum transmission speed that a Transporter can transmit at. These speeds range from 100 Mb/s through to 3.2 Gb/s.

4. *GetMaxIsochChannels(bool isInput, UInt32 *pNumIsochChannels)*
   This method returns the maximum number of isochronous channels/streams that can be received or transmitted by the corresponding device, depending on the value of the *isInput* argument. *isInput* is true for input (reception), and false for output (transmission).

5. *IsRunning(bool isInput, bool *pRunning)*
6. *Start(bool isInput)*

7. *Stop(bool isInput)*
   The *isRunning* method returns the status of isochronous transmission or reception. The *Start* method commences all isochronous transmission or reception, while the *Stop* method stops all isochronous transmission or reception for the corresponding device.

8. *GetSYTSynchChannel(UInt32 *pIsochChannel)*

9. *SetSYTSynchChannel(UInt32 isochChannel)*

10. *GetSYTSynchStatus(bool *pSynch)*
    The *GetSYTSyncChannel* method returns the isochronous channel number of the isochronous packets from which the corresponding device receives SYT timing information for word clock regeneration, while the *SetSYTSyncChannel* method specifies this isochronous channel number. The *GetSYTSyncStatus* method indicates whether the device is working as a word clock slave and receiving correct SYT timing information.

11. *GetOutputSpeed(UInt32 isochID, UInt32 *pSpeed)*

12. *SetOutputSpeed(UInt32 isochID, UInt32 speed)*
    The *GetOutputSpeed* method returns the current isochronous transmission speed for an output isochronous stream, while the *SetOutputSpeed* method sets the isochronous transmission speed. The value of the isochronous transmission speed should not exceed the value returned by the *GetMaxSpeed* method in item 3 above.

13. *GetOutputMaxDataBlocks(UInt32 isochID, UInt32 *pBlocks)*

14. *SetOutputMaxDataBlocks(UInt32 isochID, UInt32 blocks)*
    The *GetOutputMaxDataBlocks* method returns the maximum number of output data blocks per packet for the stream identified by *isochID*. This value is used when getting isochronous resources and does not necessarily control the actual number of data blocks per packet. Conversely, the *SetOutputMaxDataBlocks* method sets the maximum number of output data blocks per packet for the stream which is identified by *isochID*. The maximum value for the argument "blocks" is the value of *SYT_INTERVAL* for the current sampling frequency, namely 8 for 32 kHz, 44.1 kHz, and 48 kHz, and, 16 for 88.2 kHz and 96 kHz.

15. *GetOutputOverhead(UInt32 isochID, UInt32 *pOverhead)*

16. *SetOutputOverhead(UInt32 isochID, UInt32 overhead)*
    The overhead value is used when getting isochronous resources. The *GetOutputOverhead* method returns the overhead value, in bandwidth units (BWUs), for an isochronous stream that is identified by *isochID*. Conversely, the *SetOutputOverhead* method sets the overhead value, in BWUs, for an isochronous stream that is identified by *isochID*.

17. *GetOutputTransferDelay(UInt32 isochID, UInt32 *pDelay)*

18. *SetOutputTransferDelay(UInt32 isochID, UInt32 delay)*
    The "Audio and Music Data Transmission Protocol" [1394 Trade Association, 2002; IEC, 2005] gives a description of the Transfer Delay value. The *GetOutputTransferDelay* method returns the transfer delay value, in terms of cycle offsets, for an isochronous stream that is identified by *isochID*, while the *SetOutputTransferDelay* method sets this value.

19. *GetOutputSamplePeriod(UInt32 isochID, UInt32 *pPeriod)*
    The *GetOutputSamplePeriod* method returns the actual sampling period value (in terms of cycle offsets) for an isochronous stream identified by *isochID*. For example, for a sampling frequency of 48000 Hz, the value returned is 512 (= 8000 * 3072 / 48000).

20. *GetSYTDelay(UInt32 isochID, UInt32 *pDelay)*
    For output isochronous streams using SYT sync, SYT Delay refers to the delay between the SYTs of the stream used to regenerate the isochronous stream's word clock and the SYTs of the stream output by the isochronous stream [Audio Engineering Society - Standards Committee, 2005]. The *GetSYTDelay* method gets the SYT Delay value for the isochronous stream that is identified by *isochID*.

21. *AdjustInputPhase(UInt32 isochID, const TransporterPhaseInfo *pInfo)*
    This method does the phase adjustment processing for an input isochronous stream that is identified by *isochID*. The method is assumed to be called when a pair of audio plugs are logically connected for transfer. The actual processing depends on the input device.

22. *GetIsochChannel(bool isInput, UInt32 isochID, UInt32 *pIsochChannel)*

23. *SetIsochChannel(bool isInput, UInt32 isochID, UInt32 isochChannel)*
    The *GetIsochChannel* method returns the current isochronous channel number for an isoch-

ronous stream that is identified by *isochID*, while the *SetIsochChannel* method sets the isochronous channel number.

24. *SetTxIsochChannelAuto(UInt32 isochID, UInt32 *pIsochChannel)*
    This method automatically allocates an available isochronous channel number to a stream that is identified by isochID, and returns the number of isochronous channel that has been allocated.

25. *GetEventTypeList(bool isInput, UInt32 isochID, UInt32 listSize, UInt32 *aEventType, UInt32 *pNumEventTypes)*
26. *GetEventType(bool isInput, UInt32 isochID, UInt32 *pEventType)*
27. *SetEventType(bool isInput, UInt32 isochID, UInt32 eventType)*
    The *GetEventTypeList* method returns a list of event types of the A/M Data Transmission Protocol that are supported by the corresponding isochronous stream. The "Audio and Music Data Transmission Protocol" [1394 Trade Association, 2002; IEC, 2005] describes these event types in detail. The *GetEventType* method returns the current event type of an isochronous stream that is identified by *isochID*, while the *SetEventType* method sets the current event type of the isochronous stream.

28. *GetSFCList(bool isInput, UInt32 isochID, UInt32 listSize, UInt32 *aSFC, UInt32 *pNumSFCs)*
29. *GetSFC(bool isInput, UInt32 isochID, UInt32 *pSFC)*
30. *SetSFC(bool isInput, UInt32 isochID, UInt32 sfc)*
    The *GetSFCList* method returns a list of SFCs that are supported by an isochronous stream that is identified by *isochID*. The *GetSFC* method returns the current SFC for an isochronous stream that is identified by *isochID*, while the *SetSFC* method sets the current SFC.

31. *GetSequenceTypeList(bool isInput, UInt32 isochID, UInt32 listSize, UInt32 *aSeqType, UInt32 *pNumSeqTypes)*
    This method retrieves a list of sequence types that are supported by an isochronous stream that is identified by *isochID*.

32. *GetMaxSequences(bool isInput, UInt32 isochID, UInt32 seqType, UInt32 *pMaxSequences)*
33. *GetMaxSequences(bool isInput, UInt32 isochID, UInt32 seqType, UInt32 sfc, UInt32 *pMaxSequences)*
34. *GetNumSequences(bool isInput, UInt32 isochID, UInt32 seqType, UInt32 *pNumSequences)*
35. *SetNumSequences(bool isInput, UInt32 isochID, UInt32 seqType, UInt32 numSequences)*
36. *GetMaxSequenceNumber(bool isInput, UInt32 isochID, UInt32 seqType, UInt32 *pMaxSeqNumber)*

    The two *GetMaxSequences* methods (items 32 and 33 above) return the maximum number of sequences of a given sequence type that are available for an isochronous stream which is identified by *isochID*. The *GetMaxSequences* method shown in item 33 has an additional *sfc* argument, hence it only returns the maximum number of sequences that are supported at a sampling frequency that is specified by the *sfc* argument. The *GetNumSequences* method returns the current number of sequences of the specified sequence type within an isochronous stream that is identified by *isochID*, while the *SetNumSequences* method sets the number of sequences. The *GetMaxSequenceNumber* method returns the maximum value of the sequence number for a given sequence type within an isochronous stream that is identified by *isochID*.

37. *GetSequenceDataTypeList(bool isInput, UInt32 isochID, UInt32 seqType, UInt32 seqID, UInt32 listSize, UInt32 *aSeqDataType, UInt32 *pNumSeqDataTypes)*
38. *GetSequenceDataType(bool isInput, UInt32 isochID, UInt32 seqType, UInt32 seqID, UInt32 *pDataType)*
39. *SetSequenceDataType(bool isInput, UInt32 isochID, UInt32 seqType, UInt32 seqID, UInt32 dataType)*

    The *GetSequenceDataTypeList* method returns a list of sequence data types that are supported for a specified sequence. The *GetSequenceDataType* method returns the current data type of the specified sequence, while the *SetSequenceDataType* method sets the current data type for the sequence.

40. *GetSequenceNumber(bool isInput, UInt32 isochID, UInt32 seqType, UInt32 seqID, UInt32 *pSeqNumber)*
41. *SetSequenceNumber(bool isInput, UInt32 isochID, UInt32 seqType, UInt32 seqID, UInt32*

seqNumber)
The *GetSequenceNumber* method returns the current sequence number for the specified sequence, while the *SetSequenceNumber* sets the current sequence number for the sequence.

42. *GetMaxSubsequences(bool isInput, UInt32 isochID, UInt32 seqType, UInt32 seqID, UInt32 *pNumSubs)*
43. *GetNumSubsequences(bool isInput, UInt32 isochID, UInt32 seqType, UInt32 seqID, UInt32 *pNumSubs)*
44. *SetNumSubsequences(bool isInput, UInt32 isochID, UInt32 seqType, UInt32 seqID, UInt32 numSubs)*
    The *GetMaxSubsequences* method returns the maximum number of subsequences that are available in a specified sequence. The *GetNumSubsequences* method returns the current number of subsequences in a specified sequence, while the *SetNumSubsequences* method sets the number of subsequences in the specified sequence.

45. *GetSubsequenceNumber(bool isInput, UInt32 isochID, UInt32 seqType, UInt32 seqID, UInt32 subID, UInt32 *pSubNumber)*
46. *SetSubsequenceNumber(bool isInput, UInt32 isochID, UInt32 seqType, UInt32 seqID, UInt32 subID, UInt32 subNumber)*
    Up to eight MIDI messages may be multiplexed in a single sequence. For multiplexed MIDI, the value of MOD(DBC, 8) is returned via the subsequence number. The *GetSubsequenceNumber* method returns the current subsequence number of a specified subsequence, while the *SetSubsequenceNumber* method sets the subsequence number for the specified subsequence.

47. *GetSequenceInfo(bool isInput, UInt32 isochID, UInt32 seqType, UInt32 seqID, UInt32 subID, UInt32 infoType, void *pInfo, UInt32 infoSize)*
48. *SetSequenceInfo(bool isInput, UInt32 isochID, UInt32 seqType, UInt32 seqID, UInt32 subID, UInt32 infoType, const void *pInfo, UInt32 infoSize)*
    The *GetSequenceInfo* method returns information on a specified sequence, while the *SetSequenceInfo* method sets the information on a particular sequence. The value that is returned or set depends on the corresponding device.

A.2 *DeviceTransporter* Class Definition

The *DeviceTransporter* class is a type of *Transporter* class that represents a standalone device Transporter. The class abstracts the functionality that is common to all standalone device Transporters, and defines a common API that abstracts the hardware-dependent portions. As shown in Figure A.1, the *DeviceTransporter* class inherits from the *Transporter* class and hence, it implements all the methods that are defined in the abstract *Transporter* class. The methods that are defined exclusively within the *DeviceTransporter* class, together with descriptions of their purposes,are shown below:

1. *GetPlugIn(void)*
   This method returns a pointer to the plug-in module object that the *DeviceTransporter* object uses. The specification of plug-in module objects is defined for each platform.

2. *SetEnablerAddress(void)*
3. *ClearEnablerAddress(void)*
4. *IsUnderControl(void)*
   There may be more than one Enabler on an mLAN network. However, each Transporter may only be controlled by one Enabler. The *SetEnablerAddress* method sets the address of Enabler that is controlling the Transporter to the ENABLER_ADDRESS register of the Transporter, while the *ClearEnablerAddress* method clears the ENABLER_ADDRESS register of the Transporter. The *IsUnderControl* method checks whether the Transporter is under the computer's control.

5. *InterruptSetNotify(void (*callback)(UInt32 code, void *pParam), void *pParam)*
6. *InterruptClearNotify(void (*callback)(UInt32 code, void *pParam), void *pParam)*
   The *InterruptSetNotify* method sets the callback function that will be called when an interrupt occurs from the corresponding Transporter device. This callback function is cleared by the *InterruptClearNotify* method.

7. *InputPortSetNotify(UInt32 portID, void (*callback)(UInt32 count, const UInt8 *aData, void *pParam), void *pParam)*

8. *InputPortClearNotify(UInt32 portID, void (*callback)(UInt32 count, const UInt8 *aData, void *pParam), void *pParam)*
   The *InputPortSetNotify* method sets the callback function that will be called when data is received by an input port of the corresponding Transporter device. This callback is cleared by the *InputPortClearNotify* method.

9. *OutputPortSend(UInt32 portID, UInt32 *pCount, const UInt8 *aData)*
   This method sends data via an output port of the corresponding Transporter device.

10. *GetTransporterMode(UInt32 *pMode)*

11. *SetTransporterMode(UInt32 mode)*
    The *GetTransporterMode* method returns operating mode of the Transporter device, while the *SetTransporterMode* method specifies the operating mode. The value of operating mode depends on the device.

12. *Identify(UInt16 data)*

13. *GetIdentificationStatus(UInt16 *pData)*
    The *Identify* method controls "Identify" operation of the Transporter device. For example, such control may start/stop the blinking operation of the device's LEDs for device recognition purposes. The value to specify operation depends on the device, but the device should stop the identification process when zero is specified in *data*. The *GetIdentificationStatus* method returns the status of "Identify" operation of the corresponding device.

14. *CallPlugInControlPanel(PlugInCPParamPtr pParam)*
    This method starts the user interface program (Control Panel) which is used to change attributes of the *DeviceTransporter* object.

15. *CallPlugInSpecificFunction(UInt32 funcID, void *pParam, UInt32 paramSize)*
    This method calls a plug-in module specific function.

16. *CaptureWholeParameters(void *pParam, UInt32 *pParamSize)*

17. *RestoreWholeParameters(const void *pParam, UInt32 paramSize)*

The *CaptureWholeParameters* method returns all the parameters of the *DeviceTransporter* object. These parameters may be restored at a later stage by calling the *RestoreWholeParameters* method.

A.3 *PCTransporter* Class

The *PCTransporter* class is a type of *Transporter* class that represents Transporter implementations on workstations that have IEC 61883-6 capable drivers. The *PCTransporter* class abstracts all platform-dependent components of PC Transporter implementations. The *OSPCTransporter* class that is shown in Figure A.1 represents platform-specific PC Transporter implementations. For example, a Windows based PC Transporter implementation is shown in Figure A.2, as described by Okai-Tettey [2007]. The WinPCTransporter component that is shown in the diagram

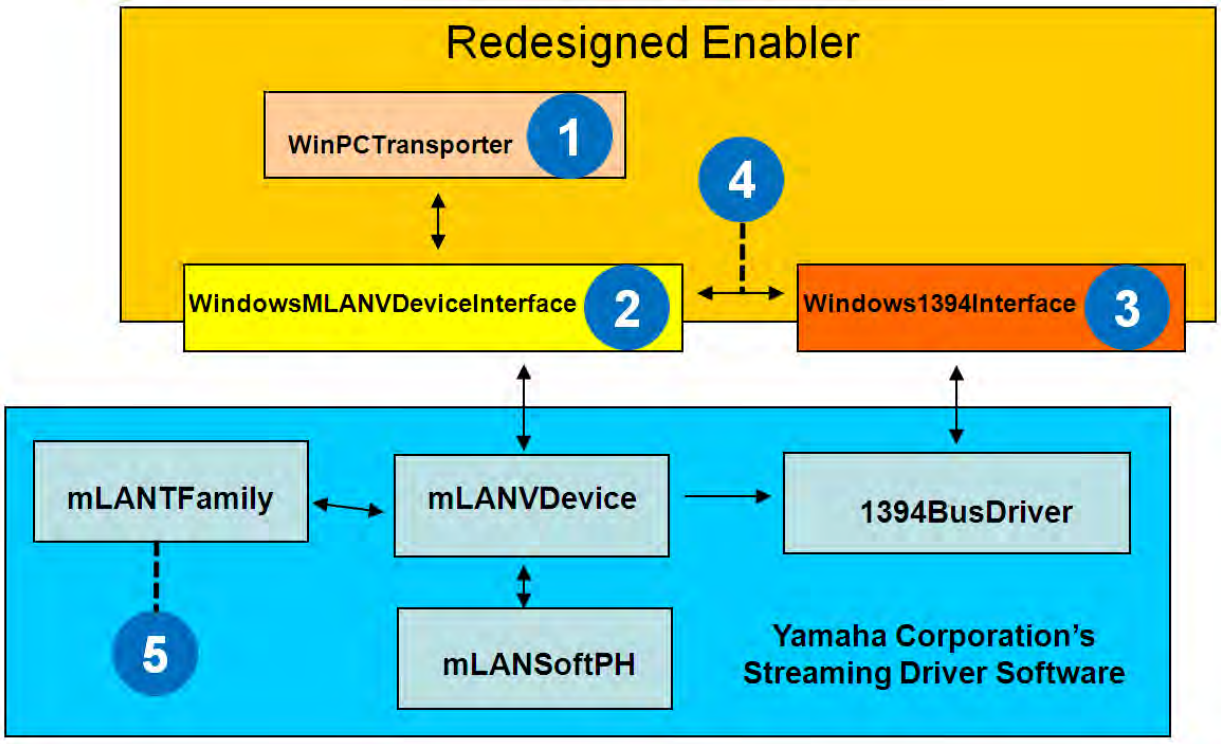


Figure A.2: PC Transporter Communication Model

represents an implementation that is equivalent to the *OSPCTransporter* class in Figure A.1. The Window PC Transporter implementation utilises Yamaha Corporation's streaming driver software, as shown in Figure A.2. Important components of this implementation are labelled 1,

2, 3, 4, and 5. A description of these five components is given below:

1. The *WinPCTransporter* class implements the methods that are defined in the *Transporter* class. These methods enable IEC 61883-6 transmissions. The *WinPCTransporter* class communicates with Yamaha Corporation's *mLANSoftPH* process via a *WindowsMLANVDeviceInterface* object.
2. The *WindowsMLANVDeviceInterface* object is a global object that is created when the Enabler is initialised. It implements an interface to Yamaha Corporation's *mLANVDevice* process, as well as to the *mLANSoftPH* process.
3. The *Windows1394Interface* class implements an interface to Yamaha Corporation's *1394BusDriver* which enables 1394 read, write, and lock transactions, in addition to other capabilities.
4. The *Window1394Interface* class mainly communicates with the *WindowsMLANVDeviceInterface* object to set up bus reset, FCP, and Bridge messaging notifications.
5. The *mLANTFamily* process is not used by the Redesigned Enabler, although it is used by the Basic Enabler.

Appendix B

Proposed Transporter HAL API

Figure B.1 shows an object model of the Hardware Abstraction Layer (HAL) design that has been proposed in this thesis for the Redesigned Enabler. The design is based on the fundamental

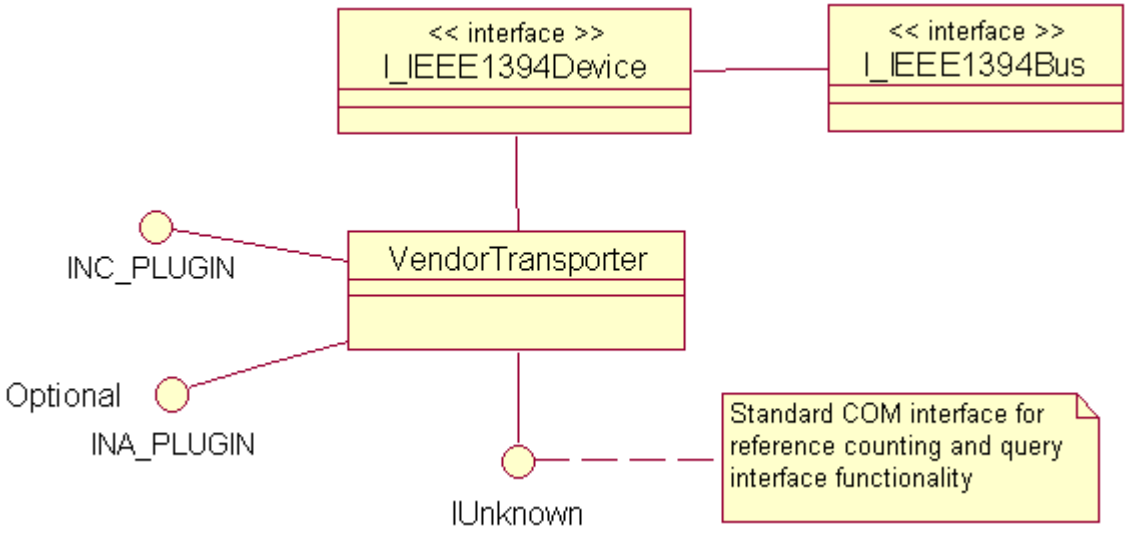


Figure B.1: Proposed Redesigned Enabler Hardware Abstraction Layer Object Model

concepts of the Component Object Model (COM). As shown in the diagram there are two main COM interfaces that comprise the Transporter HAL API, namely the *INC_PLUGIN* interface and the *INA_PLUGIN* interface. The *VendorTransporter* class represents the COM component which implements the methods that are defined by the COM interfaces. For each device that uses a particular plug-in, a corresponding instance of the *VendorTransporter* COM component is created.

The node application component of a Transporter represents the hosting IEEE 1394 device and its range of legacy audio and MIDI plugs, while the node controller component of the Transporter represents the IEEE 1394 node that is hosted by a device. The *INC_PLUGIN* COM interface is a mandatory interface that defines methods to access the node controller capabilities of Transporters. All Transporter plug-ins that comply with our proposed design should provide an implementation of the *INC_PLUGIN* COM interface. The *INA_PLUGIN* COM interface is an optional interface that accesses the node application capabilities of Transporters. This thesis has indicated that there is currently no implementation for the *INA_PLUGIN* COM interface, hence, only the *INC_PLUGIN* COM interface definition is provided in this appendix.

B.1 *INC_PLUGIN* COM Interface Definition

1. *GetHALVendorID(LONG* pHALVendorID)*
2. *GetHALModelID(LONG* pHALModelID)*
   The *GetHALVendorID* and *GetHALModelID* methods return the HAL Vendor ID and the HAL Model ID, respectively, that match those of the devices that the plug-in can control.

3. *Initialize(LPSTR p1394DevInterfacePtr, LPSTR pConfigROM)*
   This method should be called immediately after an instance of the *VendorTransporter* COM component has been created. The method forwards pointers to the important interfaces of the Enabler (*I_IEEE1394Device* and *I_IEEE1394Bus*) that enable the COM instance to perform asynchronous transactions and isochronous resource allocation.

   GENERAL TRANSPORTER METHODS

4. *GetSupportedTransmissionModes(LONG listSize, LONG* pNumModes, LONG *pTransmissionModesList)*
5. *GetTransmissionMode(LONG* transmissionMode)*
6. *SetTransmissionMode(LONG transmissionMode)*
   The "Audio and Music Data Transmission Protocol" [1394 Trade Association, 2002; IEC, 2005] specifies two transmission modes that a Transporter can operate in, namely blocking and non-blocking modes. The *GetSupportedTransmissionModes* method returns a list of

the transmission modes that are supported by a Transporter. The *GetTransmissionMode* method returns the current transmission mode that the Transporter is using, while the *SetTransmissionMode* method sets the current transmission mode.

7. *GetMaxRxOffset(LONG* rxOffset)*
8. *GetMaxTxOffset(LONG* txOffset)*
   The *GetMaxRxOffset* method returns the maximum value of the offset that may be added to the Presentation Time of incoming isochronous streams. The *GetMaxTxOffset* method returns the maximum value of Transfer Delay that is used for the formation of SYTs of outgoing isochronous streams.

9. *GetRxTargetPhase(LONG* rxTargetPhase)*
   This method returns the target phase for devices that are not capable of receiving incoming isochronous streams of arbitrary phase.

10. *Identify(LONG identify)*
11. *GetIdentificationStatus(LONG* identificationStatus)*
    The *Identify* method controls "Identify" operation of the Transporter device. For example, such control may start/stop the blinking operation of the device's LEDs for device recognition purposes. The value to specify the operation depends on the device, but the device should stop the identification process when zero is specified in data. The *GetIdentificationStatus* method returns the status of "Identify" operation of the corresponding device.

12. *GetNickname(LPSTR* pNickname, VARIANT_BOOL* writeable)*
13. *SetNickname(LPSTR pNickname)*
    The *GetNickname* method returns the nickname that is assigned to a device, while the *SetNickname* method assigns a nickname to the device.

14. *GetFirmwareVersionName(LPSTR* pFirmwareVersionName)*
    This method returns a textual name that indicates the firmware version of the Transporter.

15. *Reboot()*

This method initiates a software reset on a device.

16. *GetTransporterMode(LONG* pMode)*
17. *SetTransporterMode(LONG mode)*
    The *GetTransporterMode* method returns operating mode of the Transporter device, while the *SetTransporterMode* method specifies the operating mode. The value of operating mode depends on the device.

PLUG LAYOUT METHODS

18. *GetNumPlugLayouts(LONG* numPluglayouts)*
    This method returns the total number of Plug Layouts that are implemented by the Transporter. The Plug Layouts shall be referred to by using an ID that runs from 0 to (numPluglayouts - 1).

19. *GetCurrentPlugLayout(LONG* plugLayoutID)*
20. *SetCurrentPlugLayout(LONG plugLayoutID)*
    The *GetCurrentPlugLayout* method returns the ID of the Plug Layout that is currently in use, while the *SetCurrentPlugLayout* method sets the Plug Layout to be used.

21. *GetPlugLayoutName(LONG plugLayoutID, LPSTR* pluglayoutName, VARIANT_BOOL* writeable)*
22. *SetPlugLayoutName(LONG plugLayoutID, LPSTR pluglayoutName)*
    The *GetPlugLayoutName* method returns the name that is assigned to the specified Plug Layout, while the *SetPlugLayoutName* method assigns a name to the specified Plug Layout.

ISOCHRONOUS STREAM PLUG (ISP) METHODS

23. *GetNumISPs(LONG pluglayoutID,VARIANT_BOOL isInput, LONG* numISPs)*
24. *GetISPs(LONG pluglayoutID, VARIANT_BOOL isInput,LONG listSize, LONG* numISPsFound, LONG* ispIndexes)*

The *GetNumISPs* method returns the number of ISPs of a particular direction, in a specified Plug Layout. The *GetISPs* method returns an aggregation of the IDs of ISPs of a particular direction within the specified Plug Layout. The IDs returned in this case correspond to unique identifiers of each ISP regardless of the underlying hardware architecture.

25. *Start(LONG ispID)*
26. *Stop(LONG ispID)*
    The *Start* method starts isochronous streaming for the specified ISP, while the *Stop* method stops isochronous streaming for the specified ISP.

27. *GetIntendedStreamingStatus(LONG ispID, VARIANT_BOOL *isStartSet, LONG* fixed, LONG* linked, LONG* unique, LONG* dependency, LONG* group)*
28. *IsRunning(LONG ispID, VARIANT_BOOL* isRunning)*
    The *GetIntendedStreamingStatus* method checks if the specified ISP has been started. This indicates the intended streaming state of the ISP. Note that errors may occur during streaming so the *GetIntendedStreamingStatus* method does not tell the Enabler of the actual streaming state of the ISP. The *IsRunning* method reports the actual streaming state of an ISP.

29. *GetIsochChannel(LONG ispID, LONG* isochChannelNum, LONG* pFixed, LONG* pLinked, LONG* pUnique, LONG* pDependency, LONG* pGroup)*
30. *SetIsochChannel(LONG ispID, LONG isochChannelNum)*
    The *GetIsochChannel* method returns the isochronous channel number for the isochronous stream transmitted or received by a specified ISP, while the *SetIsochChannel* method specifies the isochronous channel number for the isochronous stream to be transmitted or received by the specified ISP.

31. *GetISPSupportedEvtTypeList(LONG ispID,LONG listSize, LONG* pNumEvtTypes, LONG* evtTypeList)*
32. *GetISPCurrentEvtType(LONG ispID, LONG* evtType)*
33. *SetISPCurrentEvtType(LONG ispID, LONG evtType)*
    The *GetISPSupportedEvtTypeList* method returns a list of event types of A/M Data Trans-

mission Protocol that are supported by the specified ISP. The "Audio and Music Data Transmission Protocol" [1394 Trade Association, 2002; IEC, 2005] describes event types in detail. The *GetISPCurrentEvtType* method returns the current event type for a specified ISP, while the *SetISPCurrentEvtType* method sets the current event type for the specified ISP.

34. *GetISPSupportedSFCList(LONG ispID, LONG listSize, LONG* pNumSFCs, LONG* sfcList)*
    This method returns a list of the sampling frequencies that are supported by the specified ISP.

35. *GetMaxAttachableNCPs(LONG ispID, LONG ncpType, LONG* pNumNCPs)*
    This method returns the maximum number of NCPs of the given NCP type that are attachable to the specified ISP. NCP type is either MIDI or audio.

36. *IsSYTOutputCapable(LONG ispID, VARIANT_BOOL* isSYTOutputCapable)*
    This method reports whether the specified ISP is capable of receiving SYT timing information that is required by the device for word clock regeneration.

37. *GetISPWCLKID(LONG ispID, LONG* wclkID, LONG* pFixed, LONG* pLinked, LONG* pUnique, LONG* pDependency, LONG* pGroup)*

38. *SetISPWCLKID(LONG ispID, LONG wclkID)*
    The *GetISPWCLKID* method returns the ID of the word clock output supplying word clocks to a particular ISP, while the *SetISPWCLKID* method sets the ID of the word clock output that should supply word clocks to a particular ISP.

39. *GetSYTDelay(LONG ispID, LONG *sytDelay)*
    For output ISPs using SYT sync, SYT Delay refers to the delay between the SYTs of the stream used to regenerate the isochronous stream's word clock and the SYTs of the stream output by the ISP [Audio Engineering Society - Standards Committee, 2005]. The *GetSYTDelay* method gets the SYT Delay value for the specified ISP.

40. *GetTransferDelayOffset(LONG ispID, LONG *delay, LONG* pFixed, LONG* pLinked, LONG* pUnique, LONG* pDependency, LONG* pGroup)*

41. *SetTransferDelayOffset(LONG ispID, LONG delay)*
    Transfer delay is an Enabler specified fixed offset that may be added to the arrival times of the isochronous packet clusters to be time-stamped. The *GetTransferDelayOffset* method returns the transfer delay that is currently set for the specified ISP, while the *SetTransferDelayOffset* method sets the current transfer delay for the specified ISP. Units are in cycle time offsets. Note that the value specified should be within the range specified by the *GetMaxTxOffset* method, shown in item 8.

42. *GetMaxOutputSpeed(LONG *maxSpeed)*

43. *GetOutputSpeed(LONG ispID, LONG *speed, LONG* pFixed, LONG* pLinked, LONG* pUnique, LONG* pDependency, LONG* pGroup)*

44. *SetOutputSpeed(LONG ispID, LONG speed)*
    The *GetMaxOutputSpeed* method returns the maximum speed at which the device (Transporter) can transmit at. At the time of this writing possible values include S100, S200, S400, S800, S1600, and S3200. The *GetOutputSpeed* method returns the speed that the specified output ISP is transmitting at, while the *SetOutputSpeed* method specifies the speed at which the output ISP should transmit at. The speed specified should not be greater than the maximum speed at which the device is capable of transmitting at.

45. *GetOutputSY(LONG ispID, LONG *outputSY, LONG* pFixed, LONG* pLinked, LONG* pUnique, LONG* pDependency, LONG* pGroup)*

46. *SetOutputSY(LONG ispID, LONG outputSY)*
    The *GetOutputSY* method returns the value that is written to the *SY* field of the isochronous packets being transmitted by the specified ISP, while the *SetOutputSY* method specifies the value to be written to the *SY* field of all the isochronous packets transmitted by the ISP. By default, all transmitters set the *SY* field value to 0_{16}.

47. *GetISPFDF(LONG ispID, LONG *inputFDF)*
    This method returns the value of the *FDF* field of incoming isochronous streams for the specified ISP.

48. *AdjustTargetPhase(LONG ispID, LPSTR phaseInformation)*
    This method does the phase adjustment processing for a specified input ISP. The method is assumed to be called when a pair of audio plugs are logically connected for transfer. The actual processing depends on the input device.

49. *GetMaxNumSequences(LONG ispID, LONG* pMaxNumSeqs)*
50. *GetOutputNumSequences(LONG ispID, LONG* numSeqs)*
51. *GetOutputMaxDataBlocks(LONG ispID, LONG* maxDataBlocks)*
52. *GetOutputOverhead(LONG ispID, LONG* outputOverhead)*
    These four methods are mainly used in a multi-bus environment for bandwidth allocation when isochronous streams are forwarded from one bus to another. However, this thesis has only focused on a single bus environment. The *GetMaxNumSequences* method gets the maximum number of sequences that may be transmitted by a specified ISP. The *GetOutputNumSequences* method returns the actual number of sequences that a specified ISP is transmitting. The *GetOutputMaxDataBlocks* method returns the maximum number of output data blocks per packet for the stream transmitted by the specified ISPs. The *GetOutputOverhead* method returns the overhead value, in bandwidth units (BWUs), for an isochronous stream that is transmitted by the specified ISP.

53. *ReadISPError(LONG ispID, LONG *ispError)*
    This method reads any errors that are reported by the specified ISP. Such errors include sample rate mismatch between the expected and actual sampling frequency of the isochronous packets received, errors in the *DBC* field of received packets, and insufficient isochronous resources on the IEEE 1394 bus.

NODE CONTROLLER PLUG (NCP) METHODS

54. *GetNumNCPs(LONG pluglayoutID, VARIANT_BOOL isInput, LONG ncpType, LONG* pNumNCPs)*
55. *GetNCPs(LONG pluglayoutID, VARIANT_BOOL isInput,LONG ncpType, LONG listSize, LONG* numNCPs ,LONG* ncpList)*
    The *GetNumNCPs* method returns the number of NCPs of a particular direction and type (audio or MIDI) within a specified Plug Layout. The *GetNCPs* method returns a list of

NCP IDs for NCPs of a particular direction and type within the specified Plug Layout.

56. *IsLiveAttachmentSupported(LONG ncpID, VARIANT_BOOL* isLiveAttachmentSupported)*
This method returns true if the specified NCP supports live attachments, and false otherwise. Live attachment refers to the ability to modify an NCP's attachment point while streaming is in progress.

57. *GetAttachedISP(LONG ncpID, LONG* ispID, LONG* pFixed, LONG* pLinked, LONG* pUnique, LONG* pDependency, LONG* pGroup)*

58. *SetAttachedISP(LONG ncpID, LONG ispID)*
The *GetAttachedISP* method returns the ID of the ISP to which the specified NCP is attached, while the *SetAttachedISP* method specifies the ID of the ISP which the specified NCP should attach to.

59. *Attach(LONG ncpID)*

60. *Detach(LONG ncpID)*

61. *IsAttached(LONG ncpID, VARIANT_BOOL* isAttached, LONG* pFixed, LONG* pLinked, LONG* pUnique, LONG* pDependency, LONG* pGroup)*
The *Attach* method activates transfer of data to/from an NCP's attached ISP. If live attachment is supported, this method also makes effective and any pending modifications to the ISP ID, SEQUENCE and SUBSEQUENCE attributes of the specified NCP. The *Detach* method stops the transfer of data between the specified NCP and its attached ISP. The *IsAttached* method returns a status value that indicates whether or not there is active transfer of data between an NCP and its attached ISP.

62. *GetSequenceNumber(LONG ncpID, LONG* sequenceNum, LONG* pFixed, LONG* pLinked, LONG* pUnique, LONG* pDependency, LONG* pGroup)*

63. *SetSequenceNumber(LONG ncpID, LONG sequenceNum)*
The *GetSequenceNumber* method returns the sequence position at which the NCP sends or extracts audio/MIDI data to/from the isochronous stream transmitted or received by the NCP's attached ISP, while the *SetSequenceNumber* method specifies this sequence position.

64. *GetSubsequenceNumber(LONG ncpID, LONG* subsequenceNum, LONG* pFixed, LONG* pLinked, LONG* pUnique, LONG* pDependency, LONG* pGroup)*

65. *SetSubsequenceNumber(LONG ncpID, LONG subsequenceNum)*
    Up to 8 MIDI sequences may be multiplexed in a single sequence position. The *GetSubsequenceNumber* method returns the subsequence position at which the specified MIDI NCP sends/extracts MIDI data to/from the isochronous stream that is transmitted or received by the NCP's attached ISP. The *SetSubsequenceNumber* method specifies this subsequence position for the specified NCP.

66. *GetNCPSupportedSFCList(LONG ncpID, LONG listSize, LONG* pNumSuppSFCs, LONG* pSuppSFCList)*
    This method returns a list of sampling frequencies that are supported by the specified NCP

67. *GetNCPSupportedSubFormatList(LONG ncpID, LONG listSize, LONG* pNumSuppSubFmts, LONG* pSuppSubfmtList)*

68. *GetNCPCurrentSubformat(LONG ncpID, LONG* currentSubFmt, LONG* pFixed, LONG* pLinked, LONG* pUnique, LONG* pDependency, LONG* pGroup)*

69. *SetNCPCurrentSubformat(LONG ncpID, LONG currentSubFmt)*
    The *GetNCPSupportedSubFormatList* method returns a list of subformats supported by the specified NCP. Possible value include 60958, MBLA, One bit audio (plain), One bit audio (encoded), High precision MBLA, MIDI, SMPTE Time Code, Sample Count, Ancillary Data, 32-bit floating point and 24*4 Audio Pack. The *GetNCPCurrentSubformat* method reports current subformat that is used by the specified NCP, while the *SetNCPCurrentSubformat* method sets the current subformat to be used by the specified NCP.

70. *GetNCPName(LONG ncpID, LPSTR* pNCPname, VARIANT_BOOL* writeable)*

71. *SetNCPName(LONG ncpID, LPSTR ncpName)*
    The *GetNCPName* method returns the textual name of the specified NCP, while the *SetNCPName* method specifies the textual name of the specified NCP. This textual name is displayed on connection management applications such as patch bays.

72. *ReadNCPError(LONG ncpID, LONG* ncpError)*
    This method read any errors that are reported by the specified NCP. Currently the only

error that is possible is a mismatch between the expected subformat and actual subformat of the sequence or subsequence that is received by the specified NCP.

WORD CLOCK SOURCES METHODS

73. *GetNumClockSources(LONG pluglayoutID, LONG* numClockSources)*
This method returns the number of word clock sources that are available on the Transporter, in a specified Plug Layout.

74. *IsSYTClock(LONG pluglayoutID, LONG clockID, VARIANT_BOOL* isSYTClock)*
There are currently two types of word clock sources, namely SYT and local. An SYT word clock source uses SYT timing information from incoming isochronous streams to regenerate word clocks, while a local word clock source receives its word clocks from a built-in device clock or from an external source such as the timing information within an ADAT digital audio stream. The *IsSYTClock* method returns whether the specified word clock source (identified by clockID) is an SYT word clock source or a local word clock source.

75. *GetSYTISPID(LONG pluglayoutID, LONG clockID, LONG* ispID)*

76. *SetSYTISPID(LONG pluglayoutID, LONG clockID, LONG ispID)*
The *GetSYTISPID* method returns the ISP ID for the ISP that is receiving isochronous streams from which SYT timing information is read by an SYT type of word clock source. The *SetSYTISPID* method sets the ISP ID for the SYT word clock source with the given ID.

77. *GetSupportedClockSFCList(LONG pluglayoutID, LONG clockID, LONG listSize, LONG* numSFCsSupported, LONG* supportedSFCList)*

78. *GetClockSFC(LONG pluglayoutID, LONG clockID, LONG* sfc)*

79. *SetClockSFC(LONG pluglayoutID, LONG clockID, LONG sfc)*
The *GetSupportedClockSFCList* method returns a list of the sampling frequencies that are supported by the word clock source that is identified by *clockID*, within a particular Plug Layout. At the time of this writing possible values were 32 kHz, 44.1 kHz, 48 kHz, 88.2 kHz, 96 kHz, 176.4 kHz, and 192 kHz. The *GetClockSFC* method returns the expected sampling frequency that is currently in use by the specified word clock source, while the

SetClockSFC method specifies the sampling frequency that is to be used by the specified word clock source.

80. *GetClockName(LONG pluglayoutID, LONG clockID, LPSTR* clockName, VARIANT_BOOL* writeable)*
81. *SetClockName(LONG pluglayoutID, LONG clockID, LPSTR clockName)*
The *GetClockName* method returns the textual name of the specified word clock source in the given Plug Layout, while the *SetClockName* method modifies this textual name.

WORD CLOCK OUTPUTS METHODS

82. *GetNumWordclockOutputs(LONG pluglayoutID, LONG* numWCLKOutputs)*
This method queries a device for the number of concurrently usable word clock outputs in a particular Plug Layout of a Transporter. Each of these word clock output is identified a WCLK ID that ranges from 0 to (numWCLKOutputs - 1).

83. *GetCurrentClockSource(LONG pluglayoutID, LONG wclkID, LONG* clockID)*
84. *SetCurrentClockSource(LONG pluglayoutID, LONG wclkID, LONG clockID)*
The *GetCurrentClockSource* method returns the ID of the word clock source that is generating word clocks for the given word clock output, while the *SetCurrentClockSource* method specifies the ID of the word clock source that should be used to generate word clocks for the given word clock output.

85. *GetWCLKOutputSamplePeriod(LONG pluglayoutID, LONG wclkID, LONG* outputSamplePeriod)*
This method returns the actual sample period of the word clocks that are generated by the specified word clock output.
86. *ReadWCLKError(LONG pluglayoutID, LONG wclkID, LONG* wordclockError)*
This method reads errors that are reported by a specified word clock output. Such errors include mismatches between the expected and actual sampling frequencies, and any instabilities within the generated word clocks.