HIGH SPEED END-TO-END CONNECTION MANAGEMENT IN A BRIDGED IEEE 1394 NETWORK OF PROFESSIONAL AUDIO DEVICES

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

DOCTOR OF PHILOSOPHY

of

RHODES UNIVERSITY

by

HAROLD A. OKAI-TETTEY

December 2005

Abstract

A number of companies have developed a variety of network approaches to the transfer of audio and MIDI data. By doing this, they have addressed the configuration complications that were present when using direct patching for analogue audio, digital audio, word clock, and control connections. Along with their approaches, controlling software, usually running on a PC, is used to set up and manage audio routings from the outputs to the inputs of devices.

However one of the advantages of direct patching is the conceptual simplicity it provides for a user in connecting plugs of devices, the ability to connect from the host plug of one device to the host plug of another. The connection management or routing applications of the current audio networks do not allow for such a capability, and instead employ what is referred to as a two-step approach to connection management. This two-step approach requires that devices be first configured at the transport layer of the network for input and output routings, after which the transmit and receive plugs of devices are manually configured to transmit or receive data.

From a user's point of view, it is desirable for the connection management or audio routing applications of the current audio networks to be able to establish routings directly between the host plugs of devices, and not the audio channels exposed by a network's transport, as is currently the case.

The main goal of this work has been to retain the conceptual simplicity of point-to-point connection management within digital audio networks, while gaining all the benefits that digital audio networking can offer.

Acknowledgements

My gratitude and thanks goes to the following people, without whose support, this work would never have been accomplished:

- First and foremost, I would like to thank **Richard Foss** my supervisor, without whose hard work, this thesis would not have been possible. His supervision has challenged me to learn and do more. I am most grateful for the long hours he has put into this work and for teaching me by example to be intellectually balanced.
- The mLAN team at **Yamaha Corporation, Japan** – Jun-ichi Fujimori, Ken Kounosu, Hirotaka Kuribayashi, Sean (Seiichi) Hashimoto, and Takashi Furukawa, for making my work possible through their generous funding, access to equipment and an unlimited access to their invaluable information.
- The **Distributed Multimedia Centre of Excellence at Rhodes University**, which has financial support from Telkom SA, Business Connexion, Comverse, Verso Technologies, THRIP, and the National Research Foundation.
- Shoichi Matsumoto and Masa Watanabe of **Otari Inc., Japan**, for loaning us two ND-20B devices for use in this research.
- Thank you to the **Rhodes University Computer Science Department** for the opportunity to read for this degree, from which I have immensely benefited.
- I am grateful to the staff in **Hamilton Building** who have assisted me and made available the excellent resources that have made writing and compiling this work possible.
- I would like to thank my **parents** for their continuous support and for always encouraging me to be excellent.
- I would like to thank my **brother and sister** for being with me throughout my academic and non-academic years and for helping me keep my sanity.
- Thank you to all my **friends** who have been encouraging.

Table of Contents

List of Figures

List of Tables

Chapter 1

1. Introduction

With the increasing number of audio processing devices that are manufactured by different vendors today, there is a growing need to provide a common music transport network that is capable of interconnecting these devices and that also meets the demands of today's professional music industry. In music studios, broadcasting stations, and sound installations, setting up and routing audio between the devices that form part of these environments requires considerable effort, and in most cases is an expensive process.

1.1 Routing Audio and MIDI Data

Various techniques are employed by system designers in finding effective ways of routing audio/MIDI data between devices. Often, the solutions employed are tailored towards their specific needs. For studio environments, physical point-to-point cable connections from the output connector of a device to the input connector of another are usually used. In environments that require wider audio distribution, long runs of analogue audio cables from the connectors of sound producing devices are connected to a central mixing unit, from where the audio is routed. Alternatively, a number of networking solutions that provide easier set up and data routing techniques are employed.

1.1.1 Physical Point-to-Point Cabling

With the vast number of analogue and digital connectors capable of being supported by an audio device, an equivalent number of cables are required in order to establish routings between devices. These connector types, and their corresponding cables, include MIDI, RCA, phone jacks, BNC, AES/EBU, SPDIF, ADAT, TDIF and MADI. Using this wide variety of cables ultimately leads to cable-cluttering within an environment, and in turn complicates the management of plug routings. In other words, keeping track of 'which cable is connected to what' becomes a daunting and frustrating task. In most cases, the users of such environments resort to labelling the end-points of a cable in order to identify the source of a particular input or the input that corresponds to a particular source. In addition to the clutter, there are also limitations and disadvantages in using MIDI cables, as well as the different analogue and digital audio cables.

MIDI has a slow data transmission rate of 31.25 kbaud, which results in transmission latencies. It also has a limited channel count of 16 [MMA, 2001] A further discussion related to the transmission speed of MIDI is provided by [Lehrman and Tully, 1993]. For unbalanced audio cables, noise may be introduced into the audio signal through interference from power cords, fluorescent lights, or from other devices, although balanced cables and digital transfer of audio can be used to combat these interferences. In setups where long analogue cable runs are used, amplifiers may be used to boost signal strength of the audio and this requires an extra cost. It is also not economically efficient to acquire all the different cables required by a device for interconnecting with other devices.

1.1.2 Networked Audio Alternatives

A number of companies have developed a variety of network approaches to the transfer of audio and MIDI data. Along with their approaches, controlling software, usually running on a PC, is used to set up and manage audio routings from the outputs to the inputs of devices. These networks, which are inherently digital, deal with the problems encountered with using physical point-to-point cabling, and also offer improved performance in the quality of the audio data transmitted over the network.

In the emerging world of networked audio technology, a number of criteria are used to judge the performance of a particular audio network over another. The main criteria include:

- The amount of latency.
- Capability of providing a common clock signal.
- Whether the system is based on a proprietary specification or an open standard.
- Whether it incorporates a means for routing audio and MIDI data.
- The amount of user effort required to set up and to add or remove devices.

Chapter 2 describes these criteria and uses them to appraise a number of widely used networked audio technologies. From this chapter, it is discovered that most of these networks use Ethernet as their transport medium. Also in chapter 2, three other criteria that should be considered, in addition to those listed above, are introduced. These criteria form the basis on which this study is based.

From a review conducted on the networked audio technologies, it is found out that mLAN[1], an emerging audio networking standard, fulfils the listed criteria. An mLAN network, unlike most other networked audio solutions, uses the IEEE 1394 high performance serial bus [IEEE, 1995] as its physical transport medium. The mLAN concept was first envisioned and created by Yamaha Corporation, and is now gaining increasing popularity amongst a number of other device manufacturers.

1.1.2.1 The Problem Statement

Over the years, a number of hardware as well as software protocols have been developed to enable audio and control data to be transmitted over IEEE 1394. In order for one to set up routings between inputs and outputs of devices on an mLAN network, a connection management service, referred to as the *Enabler*, was developed on workstations to expose to user-level applications the plug connection behaviour of the devices on the network.

[1] mLAN is an acronym for music Local Area Network

There have been two versions of mLAN so far – *mLAN Version 1* and *mLAN Version 2*. With the change in device architecture from *mLAN Version 1* to *mLAN Version 2*, the Enabler also had to go through significant changes in its design and implementation. With *mLAN Version 2*, the Enabler was required to have full control of and model appropriately the basic audio/MIDI transport capabilities of a device. Industry use of this Enabler in setting up audio/MIDI routings required that across-bus plug connections (using IEEE 1394 bridges), and proper bandwidth management be handled by the Enabler. The initial design and implementation of the *mLAN Version 2* Enabler does not meet the performance standards of industry. This mainly has to do with:

- The speed at which the Enabler establishes audio/MIDI routings.
- The stability of the Enabler in handling multiple/simultaneous user actions, and recovering from network configuration changes.
- Inadequate facilities for the control of network bandwidth. This control is necessary to allow the optimal number of data transmissions.
- Incomplete word clock control.
- Problems associated with providing IEEE 1394 bridging support, thus enabling audio/MIDI routings to be set up between devices across different buses.
- Providing interoperability between *mLAN Version 1* and *mLAN Version 2* devices.

In addition to these industry-driven requirements, it was also observed that the connection management control services provided by the current networking audio systems (including the Enabler of mLAN) lack the capability of routing audio/MIDI data all the way from an input connector on a device, or from an internal data bus line, to the transport layer for transmission over the network. They also lack the capability of routing particular audio/MIDI channels carried by the network all the way to one or more output connectors on a device, or data bus lines implemented within the device. Figure 1-1 below depicts the scope of connection management provided for a typical networked audio device.

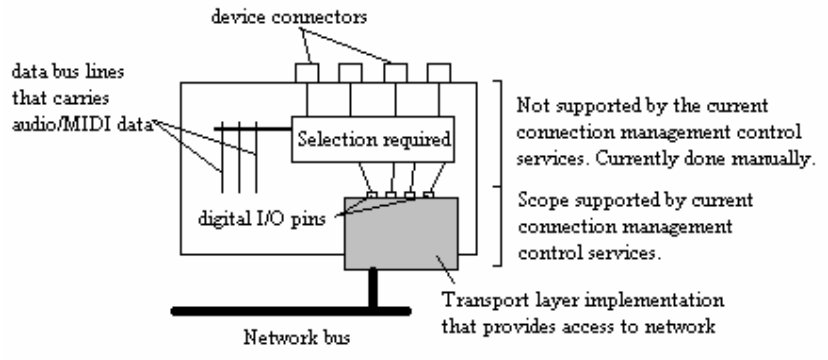


Figure 1-1: Scope of connection management supported by a typical networked audio device

The current connection management control services offered by today's networked audio technologies only go as far as providing audio/MIDI data routings between devices at the network's transport layer, and then usually require a user to manually configure a device to select particular audio/MIDI channels to/from the device's transport implementation. This operation is illustrated using Figure 1-2.

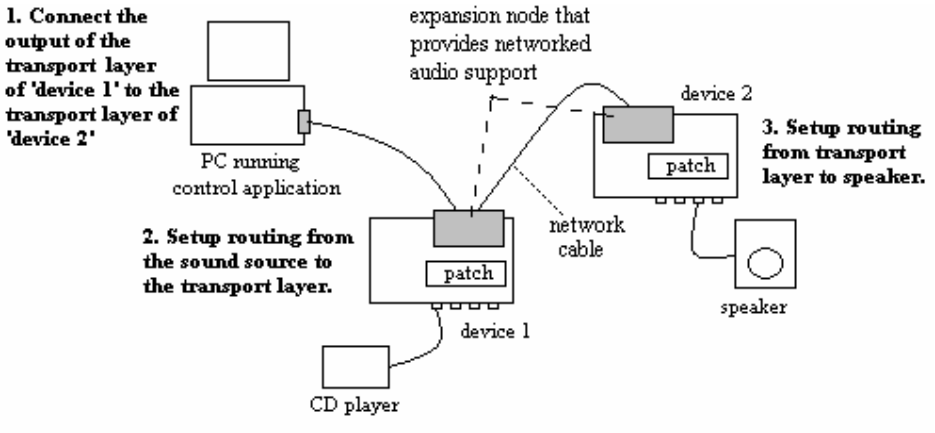


Figure 1-2: Connection management provided by current networked audio systems

From the setup shown, three operations are required to be performed by a user in order to get the audio produced by the CD player attached to 'device 1' to the speaker attached to 'device 2'. The devices labelled 'device 1' and 'device 2' are simple break-out boxes with each implementing an expansion node that is capable of transmitting and receiving audio and music data from the audio network. A detailed description of the interaction between the expansion node and its host device is given in section 6.2.1. First, from the control application running on the PC, the user connects the output of the transport layer of 'device 1' to the transport layer of 'device

2’. Following this, the user then sets up a routing from the input connector of ‘device 1’ on which the CD player is attached, to an appropriate output transport channel implemented by the audio network. Similarly on the receiving device – ‘device 2’, the user also selects from its transport layer, the relevant audio channel that is to be received by the speaker attached to the device.

Real world practical examples of these operations include the extra set up required in configuring the Yamaha O1X digital mixing console for transmission/reception of audio data, after the ‘transport-layer’ routings have been set up using the mLAN Graphic Patchbay application [Yamaha Corp., 2005a]. Also, a similar operation is performed for CobraNet enabled devices, after ‘transport-layer’ routings have been set up using the CobraNet Manager software [D&R Electronica, 2005].

Although an audio network demonstrates superior qualities over physical point-to-point cabling, the basic concept of connection management provided by point-to-point cabling is far more intuitive than what the current networked audio solutions provide. Audio networks were primarily created, amongst other things, to reduce cable clutter and provide an efficient way of routing audio/MIDI data. Current audio networking technologies fulfil the task of reducing cable clutter, but fail at providing intuitive techniques for routing audio/MIDI data – there’s the added task of performing internal routings on a device.

For an audio network to provide an intuitive data routing implementation, in other words to allow for true end-to-end connectivity, it should be able to establish point-to-point connectivity between ‘user-recognizable’ plugs and/or data bus lines of various devices, while maintaining its integrity of acting as an audio network.

A number of mLAN hardware chips that have been developed above the basic features of IEEE 1394, are capable of enabling the extraction of a channel of audio/MIDI data from an isochronous stream. These ‘extracted’ channels can then be remotely configured, with the necessary hardware, to be routed to ‘user-recognizable’ plugs of a device. The mLAN-enabled Otari ND-20B device [Otari Inc., 2005] is an example of a professional audio unit that allows for such an implementation. This research is focused on providing a specification and a corresponding implementation

for a true end-to-end connection management solution for mLAN-enabled IEEE 1394 audio devices. In achieving this goal, it is envisioned that this would also provide enhanced techniques for efficiently managing isochronous bandwidth usage of devices on a network, since isochronous resources should only be allocated to 'user-recognizable' plugs that are required for streaming. However, this form of bandwidth management is only effective if device manufacturers implement flexible routing capabilities between the 'user-recognizable' plugs of a device and its network transport implementation.

This research also aims at providing an industrial quality solution to connection management, with particular focus on providing support for IEEE 1394 bridging that:

- Out performs existing mLAN connection management Enablers developed.
- Provides backward compatibility with existing mLAN devices that do not allow for true end-to-end connectivity.
- Realizes and implements the fundamental concept of connection management, thereby allowing for true end-to-end connectivity.
- Provides user-level bandwidth management techniques for a device that implements flexible routings between its 'user-recognizable' plugs and its mLAN transport layer.
- Provides a complete connection management solution for the audio industry.

The research starts off by thoroughly examining the architectures of the *mLAN Version 1* and *mLAN Version 2* devices, as well as their corresponding Enablers. This forms the basis from which various innovative concepts are derived for the implementation of a true end-to-end solution. The Linux platform is used for development, mainly because of its open source IEEE 1394 drivers, which facilitated mLAN Enabler support of IEEE 1394 bridges. However, the design architecture of the 'new' or 'redesigned' Enabler – as it is referred to – is such that it can be ported to other operating systems. The layout of this research as described by this thesis is given below:

IEEE 1394 and Related Technologies

Chapter 3 gives an overview of the IEEE 1394 architecture that facilitates the transfer of audio and MIDI data, and permits a controller application to enumerate, reconstruct and represent the topology of an IEEE 1394 network. A brief outline of the IEEE standard bridge specification – "IEEE Standard for High Performance Serial Bus Bridges" [IEEE, 2005], and the proprietary NEC 1394 bridge implementation, which permit across bus forwarding of data are also described. At the end of this chapter, the mLAN architecture is reviewed, describing its audio and music data transmission protocol, the connection management techniques, as well as the initial hardware chips developed to provide a complete mLAN networking solution.

The *mLAN Version 1* and *mLAN Version 2* Architectures

Since the birth of mLAN in 1993, the mLAN device implementation architecture provided to vendors has changed from what is now referred to as the *mLAN Version 1* architecture, to the *mLAN Version 2* architecture. The change to *mLAN Version 2* was necessary in order to reduce the cost and mLAN expertise required for vendors to provide mLAN support within their products. The *mLAN Version 2* architecture also has a number of advantages over *mLAN Version 1*, which makes it versatile in its operation. Along with these architectural changes, Yamaha also developed workstation-based Windows Enablers that interacted with devices of the respective architectures. Similar Enablers were also developed for the Linux platform as part of this research.

Chapter 4 describes the *mLAN Version 1* architecture and the Windows and Linux Enabler developed for this architecture. Various performance measurements are applied to *mLAN Version 1* devices, with results that motivate the need for a new architecture – *mLAN Version 2*.

Chapter 5 describes the *mLAN Version 2* architecture, together with the corresponding Windows and Linux Enablers. Similar performance measurements are conducted on *mLAN Version 2* devices, which reveal remarkable speed-up in operation when compared with the results from *mLAN Version 1*.

Limitations of the Current Enabler Specification

Chapter 6, the kernel chapter of this research, reveals the limitations of the current Enabler specification in providing an industry standard connection management solution. The current Enabler specification is referred to as the Basic Enabler specification. Various factors that inhibit speed as well as increase the onset of bugs and errors are examined. Also, an analysis is performed to establish reasons why the design of the Enabler, as described by the Basic Enabler specification, is inadequate to support true end-to-end plug connectivity, and also to implement IEEE 1394 bridging capabilities. The Linux Enabler described in chapter 5 was enhanced to provide bridging support and using this, a number of performance tests and measurements have been conducted to highlight the inefficiencies of the Basic Enabler specification. The test and measurements are described and discussed.

Design and Component Level Innovations

Chapter 7 addresses the problems and limitations of the Basic Enabler specification by suggesting a number of innovative solutions. These innovations are used in designing the object model of the software architecture for a new Enabler that is focused on providing true end-to-end plug connectivity, as well as congruently incorporating support for IEEE 1394 bridges. The object model proposed illustrates a *node application* and *node controller* concept which is also defined by the chapter. The node application of the object model is defined to model that part of a device that contains the ‘user-recognizable’ properties, while the node controller is defined to model the mLAN transport layer of a device. Also from this object model, the word clock capabilities of a device are modelled for both the node application and node controller components. In the latter part of the chapter, a description is given of how the classes defined by the object model interoperate to handle network enumeration, and also how they enable client applications to perform plug connections, disconnections and word clock synchronizations.

A more detailed review of the component level innovations of the new Enabler specification is given in chapter 8. The implementations of three main components are discussed:

- Support for IEEE 1394 bridges that particularly deal with network enumeration and across-bus flow of isochronous streams.
- Enhanced isochronous resource management techniques resulting from modelling the true plugs (node application plugs) of a device.
- Effective control of word clock synchronization.

These components, in addition to the techniques employed in connecting and disconnecting audio/MIDI routings between devices, form the core of any connection management service and must be implemented rigorously to ensure optimal performance and stability. This chapter describes the techniques, and in some cases, algorithms of the main components listed above that are implemented by the new mLAN Enabler.

Evaluation

In chapter 9, the features of the new Enabler are demonstrated and tested. A description is given of a patch bay application that was also developed as part of this research and was used in demonstrating the various features of the enhanced Enabler. Otari's ND-20B devices were used in demonstrating the true end-to-end plug connectivity of the Enabler. The features of the new Enabler that were demonstrated using the patch bay include:

- True end-to-end connectivity.
- Backward compatibility with current mLAN devices that do not allow for true end-to-end connectivity.
- Effective word clock control and monitoring
- User-level representation of isochronous bandwidth usage on a network, and the various optimization techniques that can be used in efficiently managing the bandwidth.

In addition to this, chapter 9 also describes the results of a number of stress and performance tests conducted on the new Enabler. The results of these tests, in comparison with results of similar tests conducted on the previous Enablers, reveal a superior efficiency, stability and performance. However, most importantly, there is a

negligible overhead added to the overall performance of the Enabler in realizing true end-to-end connectivity.

Note that a number of technical terms is used through out this text, the definition of these terms is provided in a glossary at the last page of the thesis.

Chapter 2

2. Current Audio Networks and End-to-End Connection Management

In a survey conducted by the AES Technical Committee on Network Audio Systems (AES TC-NAS) [AES TC-NAS, 2005], a number of important criteria were used for a choice of an audio network. Some of these criteria have greater or lesser importance, and are dependent on a particular application. The criteria used include:

- General – cost, single-sourced vs. multiple-sourced, audio distribution standards
- Audio performance – latency, resolution and bandwidth, transparency, coexistence with multiple formats, synchronization
- Network characteristics – network diameter, individual run length, cable types, convergence, monitor and control, extent and scale, network administration
- Robustness – fault tolerance

Apart from the obvious importance of providing a cheap, reliable, and simple to use audio networking solution, three main points arose from the results of the survey. These included:

1. The need for an audio network to be based on open and multi-source technology standards.

2. The need for control and monitoring capability to be supported by a network.
3. The lack of clear preference between building an audio network as a dedicated infrastructure, or as a shared network.

The report stated that “open” and “multi-sourced” technology was overwhelmingly supported by the respondents of the survey, and was an interesting requirement given that all the technologies used in the respondents’ application areas are proprietary or practically single sourced. However, some panel members discussing the result of the survey were of the view that “open” is a better tasting word than “proprietary”, and the real requirement is for a widely available interoperable network which can be met by either open or proprietary technology with fair licensing terms. The fact that there is not a multi-sourced, open standard was thought of as the reason for holding back the adoption of audio networking.

Also from the result of the survey, control and monitoring is an expected capability, especially in audio for sound installations such as hotels, convention centres, etc, and is seen as an important factor in driving the adoption of audio networks.

The overall intent of the survey was to identify current applications for audio networks and understand the requirements imposed by those applications, with the hypothesis that no one solution could serve all needs. This hypothesis implied that different applications would have different requirements and one should use the technology best suited to the needs of the application. From point 3 above, this hypothesis is shown to be disproved, with the results of the survey indicating a minor differentiation between building audio networks to handle specific needs of applications, and building audio networks as a shared infrastructure.

The AES TC-NAS survey confirms that digital audio technology will continue to edge out analogue, and with that there is little doubt that networking will be the preferred distribution method [AES TC-NAS, 2005]. In view of this, it is important to ensure that audio networking evolves into an interoperable and cost effective technology that solves the problems of audio applications.

The next section describes a number of audio networks that are currently under development or actually in use and on the market, and how they shape up to the main points raised from the audio networking survey. In addition to these points, other technical issues such as: *latency*, *number of transmitted channels* and the *Quality of Service (QoS)* offered by these networks are also highlighted.

2.1 Current Audio Networks

The list below gives examples of companies that provide, or are currently developing audio networking technology.

- Intelligent Media Technologies' SmartBuss
- DANTE Project of National ICT Australia
- Hear Technologies' HearBus
- Aviom's A-Net™ Pro64
- Axia Audio's Livewire
- Gibson's Media-accelerated Global Information Carrier (MaGIC)
- Cirrus Logic/Peak Audio's CobraNet™ technology
- Digigram's EtherSound technology
- Oxford Technologies' SuperMAC
- Yamaha Corporation's mLAN Digital Network Interface Technology

Of these networks, Oxford Technologies' SuperMAC, and Yamaha Corporation's mLAN Digital Network Interface Technology are based on open-standard specifications. The rest of the companies provide proprietary solutions. The network transport employed by all these audio networks, with the exception of mLAN, is based on the Ethernet networking technology, and hence takes advantage of the inexpensive and widespread use of category-5 UTP cables in both the home and professional environments. The mLAN networking technology uses IEEE 1394/Firewire cables as its transport medium.

2.1.1 An Overview of Current Audio Networks

An overview of these networks, discussed according to the points raised at the beginning of this chapter, is given below. Note that for certain proprietary networks, detailed information could not be obtained for some of the networking criteria.

2.1.1.1 Intelligent Media Technologies' SmartBuss

SmartBuss, Intelligent Media Technologies' (IMT) first embedded network product, was developed to provide Original Equipment Manufacturers (OEMs) with a low cost solution for audio, video and serial data networking [IMT Inc., 2005a]. It is described as a scalable matrix network that routes any input to any output and mixes multiple media configurations onto a common bus. Each network node uses a SmartBuss Core IP processor, based on an industry standard Altera Field Programmable Gate Array (FPGA), managing up to 64 channels of 48 kHz audio through 8 I2S lines. Latency is fixed - 21µs each for input and output, plus 760 ns per node throughput. A SmartBuss Core provides OEM's the ability to network audio, video and control data with the lowest possible implementation costs for their systems. Network connection is via CAT-5 cable in a daisy chain configuration at 100 Mb/s, handling 64 total audio channels.

Management of SmartBuss is via IMT's PathMaster reference application for Windows. Input and output assignments and I/O signal processing, matrix mixing, network monitoring and provisioning are provided. The application has been configured so OEMs can create GUIs for their brand-specific look and feel [IMT Inc., 2005b]. Detailed information on the nature of the PathMaster application could not be obtained.

A summary of the capabilities offered by SmartBuss are as follows:

Latency	Fixed at 21 µs each for input and output plus 760 ns per node throughput.
Number of channels	Up to 64 channels of 48 kHz audio.
Network transport	CAT-5 cabling, in a daisy chain and star configuration
Control and monitoring	PathMaster reference application
Open standard/Proprietary based	Proprietary based, but uses a standards based transport

Table 2-1: Summary of the capabilities offered by IMT's SmartBuss

2.1.1.2 DANTE Project of National ICT Australia

The National ICT of Australia (NICTA) *Digital Audio Networking* (DANTE) project has developed a networked solution for transporting digital audio, based on standard hardware and data networking protocols [NICTA News, 2005]. This is based on Ethernet and TCP/IP and is capable of carrying multiple high-quality audio channels, MIDI, and control data with sample-accurate timing. In addition to this, the network is designed to be plug-and-play, allowing new sources to be discovered without the operator needing to know which socket or patch panel a source is connected to [NICTA, 2004]. The DANTE project is a new undertaking, and as such is in the process of developing proof-of-concept prototypes.

Information on the latency, channel count, and the control and monitoring aspects offered by DANTE's audio networking solution could not be obtained. A summary of the capabilities offered by this network are as follows:

Latency	No information available
Number of channels	No information available
Network transport	CAT-5 cabling, No information on supported configuration
Control and monitoring	No information available
Open standard/Proprietary based	Proprietary based, but uses a standards based transport

Table 2-2: Summary of the capabilities offered by DANTE's audio network

2.1.1.3 Hear Technologies' HearBus

The HearBus network developed by Hear Technologies [Hear Technologies, 2002], is mainly used by their headphone/monitor mixing product - HearBack™. The network is an eight channel bus that permits digital audio to be run over 500 feet of CAT-5e cable. The HearBack™ system consists of a hub and personal mixers connected via the HearBus network. A single hub supplies power to a maximum of eight mixers, or can be daisy chained using the HearBus in and out for virtually unlimited system size. The overall system delay for the HearBack™ system is less than 1.5 ms, with digital inputs capable of receiving 44.1 kHz and 48 kHz of 24-bit audio.

Local control of up to ten channels of audio is provided using the HearBack mixer. No detailed information is available for the remote monitoring and control capabilities offered by the HearBus network. A summary of the capabilities offered by the HearBack™ system using the HearBus network are as follows:

Latency	Low propagation delay of less than1.5 ms
Number of channels	8 channels of 24-bit audio
Network transport	CAT-5e cabling, in a daisy chain configuration
Control and monitoring	No detailed information available. Local control provided using the HearBack mixer.
Open standard/Proprietary based	Proprietary based, but uses a standards based transport

Table 2-3: Summary of the capabilities offered by the HearBus network

2.1.1.4 Aviom's A-Net™ Pro64

The A-Net™ Pro64 audio networking protocol developed by Aviom Inc. [Aviom Inc., 2005] is described to be the only distributed audio system to deliver archival quality audio with ultra low latency, jitter and wander, while simultaneously supporting continuously variable sample rates. It makes use of 150m cable runs on CAT-5e and has no restriction on topology. 24-bit audio is delivered at all times, with up to 64 channels capable of being distributed throughout the entire network at 48 kHz sampling rate. It is also capable of transmitting 32 audio channels at 96 kHz, and 16 at 196 kHz. A bi-directional transfer mode of operation is supported, capable of two 64-channel audio streams, one in each direction. Additionally, the A-Net™ Pro64 protocol reserves bandwidth for integrated transmission of multiple streams of non-audio data. These include MIDI, GPIO and RS-232. No information could be obtained for the control and monitoring features offered by this network.

A summary of the capabilities offered by the A-Net™ Pro64 network are as follows:

Latency	No figures available.
Number of channels	24-bit audio at all times. 64 channels at 48 kHz, 32 channels at 96 kHz and 16 channels at 192 kHz.
Network transport	CAT-5e cabling. No restriction on topology

	configuration
Control and monitoring	No information available.
Open standard/Proprietary based	Proprietary based, but uses a standards based transport

Table 2-4: Summary of the capabilities offered by A-Net™ Pro 64

2.1.1.5 Axia Audio's Livewire

Livewire, a pioneering technology invented by Telos Systems, is used by Axia products to convey low-delay and high-reliability audio over switched Ethernet, particularly geared towards radio broadcasting environments [Axia Audio/TLS Corp., 2005a]. With Livewire, a single CAT-6 Ethernet cable or fibre link is capable of carrying multiple channels of real-time uncompressed digital audio, device control messages, program associated data, routine network traffic – even VoIP data [Axia Audio/TLS Corp., 2005b]. In view of this, audio is prioritized and takes precedence over all other data types. Livewire currently employs a 100Base-T segment capable of carrying 25 bi-directional stereo channels of 48 kHz 24-bit linear PCM audio, and is based on switching Ethernet hubs. Link delay is kept below 1 ms, with less than 3 ms from microphone to monitor out, including network and processor loops.

The PathFinderPC router control application is used to provide a central point of control, via IP, of every Axia node in a Livewire network. This application scans the audio network, gathers and presents information about the audio sources and destinations of each device on the network. An audio routing is established by simply clicking on the destination, and then selecting the source [Axia Audio/TLS Corp., 2005b]. These "route points" can then be locked or unlocked to prevent other users of the system to inadvertently change routings. A snapshot of the application is shown in Figure 2-1. This snapshot is adapted from [Axia Audio/TLS Corp., 2005b].

In addition to this, the PathFinderPC control application allows bi-directional routing of audio and GPIO simultaneously. Also, a built-in audio metering feature together with a configurable silence sense, allows automated "watchdogs" on important audio sources to be set up, and hence enables automatic switching to a backup source if a particular audio signal is not present. Off-site routing control is also possible using an integrated, platform-independent web server.

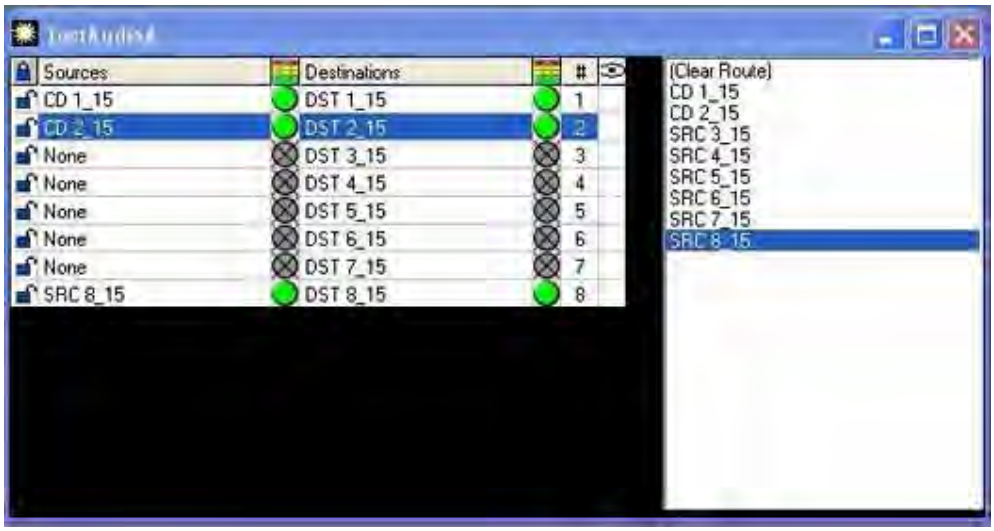


Figure 2-1: Audio routing capability provided by the PathFinderPC control application

A summary of the capabilities offered by Livewire are as follows:

Latency	Link delay kept below 1 ms. Less than 3 ms from microphone to monitor out, including network and processor loops.
Number of channels	Capable of 25 bi-directional stereo channels of 48 kHz, 24-bit linear PCM audio
Network transport	CAT-6 cabling. Supports a daisy chain configuration
Control and monitoring	PathFinderPC router control application
Open standard/Proprietary based	Proprietary based, but uses a standards based transport

Table 2-5: Summary of the capabilities offered by Livewire

2.1.1.6 Gibson's MaGIC

The Media accelerated Global Information Carrier (MaGIC), a trademark of Gibson Guitar Corporation, provides as many as 32 channels of 32-bit bi-directional audio over 100 Mbit Ethernet with sample rates up to 192 kHz [Gibson Guitar Corp., 2003a]. In addition, data and control transport is said to occur up to 30,000 times faster than MIDI. Additional cable features include phantom power, automatic clocking, and network synchronization [Dipert, 2005]. The MaGIC specification [Gibson Guitar Corp., 2003b] specifies a point-to-point latency of 250 µs.

Brian Dipert [Dipert, 2005] indicates that the impressive performance specifications of MaGIC come at a price of less-than-full Ethernet compatibility. Though the footprint of a MaGIC packet is the same as Ethernet UDP, it differs from standard Ethernet because the packet size and transmission rate do not change. Routing is done at the MAC layer (layer 2) and the data on this layer is referred to as frames and not packets [Gibson Guitar Corp., 2003c].

Gibson is currently offering evaluation boards that provide a complete platform for rapid prototyping and development of MaGIC enabled products, and also comes with a 10 year royalty free license on MaGIC. The kit includes a DSP board, 8x8 Analogue I/O, 40 pin ribbon cable, power supply, documentation and application CD. The applications include a Router/Mixer and a suite of test applications [Gibson Guitar Corp., 2003d]. Detailed information on the router/mixer application could not be obtained.

A summary of the capabilities offered by MaGIC are as follows:

Latency	250 µs point-to-point latency across 100 m
Number of channels	Up to 32 channels of 32-bit bi-directional audio with sample rates up to 192 kHz.
Network transport	CAT-5 cabling. Daisy chain, star and uplink[2].
Control and monitoring	Routing application developed for the MaGIC evaluation board system
Open standard/Proprietary based	Proprietary based, but uses a standards based transport

Table 2-6: Summary of the capabilities offered by MaGIC

2.1.1.7 Cirrus Logic/Peak Audio's CobraNet™

CobraNet™ is being offered as the standard technology for the transport of multi-channel audio and control data [Peak Audio Inc., 2004a]. It is capable of delivering, over a single CAT-5 cable, up to 64 channels of 20-bit audio at 48 kHz sampling rate. More channels are supported with 16-bit audio and less with 24-bit audio. Three

[2] Uplink is a MaGIC network topology that employs at least two switching hubs allowing several MaGIC links to be multiplexed onto a single cable.

latency values – 1.33 ms, 2.66 ms (using low-latency mode) and 5.33 ms (otherwise), are supported by the CobraNet™ network.

The CobraNet™ technology is a combination of hardware, software, and network protocol that allows the distribution of many channels of real-time, high quality digital audio over an Ethernet network [Peak Audio Inc., 2004b]. These components provide for an Ethernet network, isochronous data transport, sample clock distribution, and transport for control and monitoring data. A CobraNet™ node includes a digital signal processor (DSP) which forms the heart of CobraNet™. The DSP implements the network protocol stack, performs isochronous to synchronous and synchronous to isochronous conversions, and plays a part in sample clock generation [Klinkradt and Foss, 2003].

Three packet types make up the CobraNet™ protocol namely: *beat, isochronous data* and *reservation packets*. The beat packet is directed at a multicast address and carries the network operating parameters, clock, and transmission permissions. Only one device on the network transmits beat packets, but all devices on the network are required to listen for these packets. The isochronous packet is the carrier of audio data and is subdivided into bundles. A bundle is the smallest network audio routing envelope, with a capacity to transmit up to 8 audio channels.

Figure 2-2 below provides an illustration of the services provided by CobraNet™, and the manner in which these services reside on an Ethernet network. This is adapted from the paper “A Comparative Study of mLAN and CobraNet™ Technologies and their use in the Sound Installation Industry”, [Klinkradt and Foss, 2003].

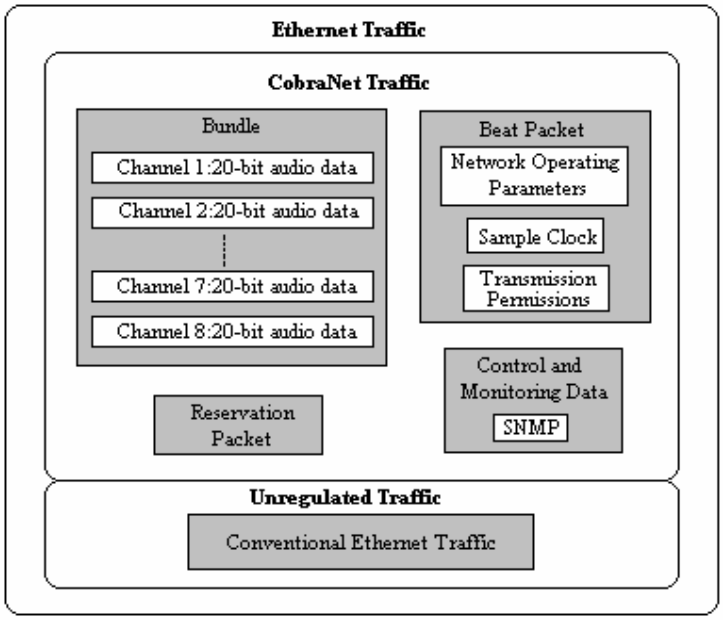


Figure 2-2: Services available on a CobraNet™ network

The Simple Network Management Protocol (SNMP) is used for controlling and monitoring CobraNet-enabled devices. Audio routing between devices is also achieved using SNMP. In this case, bundles are routed from a bundle transmitter of a device to a bundle receiver of a target device, from which the appropriate output and input plugs of the transmitting and receiving devices, respectively, are configured to transmit and receive audio. The CobraNet™ Manager application, developed by D&R Electronica [D&R Electronica, 2005] is an example of an application that provides routing capabilities for CobraNet-enabled devices. It is also implements a built-in scheduler that is used to plan connections. A snapshot of this application is given in Figure 2-3.

This part of the application is also referred to as the matrix. Currently, up to four bundle transmitters and receivers are implemented by each device. These are shown at the top and left side of the matrix, respectively. These are represented as *Tx1, Tx2, Tx3, Tx4* for the bundle transmitters, and *Rx1, Rx2, Rx3, Rx4* for the bundle receivers. To establish a connection, a user first has to click on a cross point that specifies a bundle transmitter of the source device and a bundle receiver of the destination device. Clicking the cross point causes a routing path to be established between the two devices [D&R Electronica, 2005].

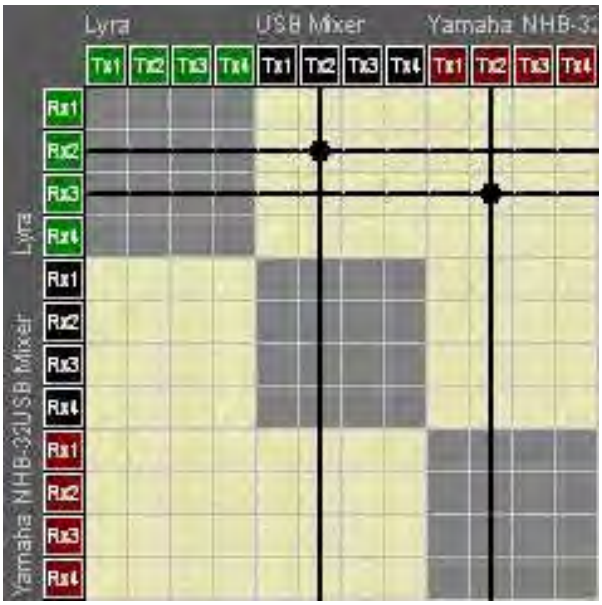


Figure 2-3: Snapshot of the CobraNet™ Manager Control application [D&R Electronica, 2005]

After this step, the desired plugs of the transmitting and receiving devices have to be configured for data transmissions to, and receptions from the CobraNet™ network.

A summary of the capabilities offered by CobraNet™ are as follows:

Latency	2.66 ms and 1.33 ms with low latency mode, 5.33 ms otherwise.
Number of channels	Up to 64 channels of 20-bit audio at 48 kHz sample rate. More channel count with 16-bit audio and less with 24-bit audio.
Network transport	CAT-5 cabling. Daisy chain and star
Control and monitoring	CobraNet™ Manager developed by D&R Electronica
Open standard/Proprietary based	Proprietary based, but uses a standards based transport

Table 2-7: Summary of the capabilities offered by CobraNet™

2.1.1.8 Digigram's EtherSound Technology

EtherSound is described as an elegant, simple, and open standard for networking digital audio [Digigram, 2005]. It is fully compliant with the Ethernet standard (IEEE

802.3) and provides bi-directional, deterministic, low-latency transmission of synchronized audio channels and control data over standard Ethernet.

It is capable of 64 channels of 24-bit/48 kHz PCM audio, including embedded control and monitoring data. Depending on the sampling frequency used, other channel counts are possible, i.e. 32 channels at 96 kHz. Audio quality is ensured by the ultra low jitter resulting from the built-in clock recovery. A 250 µs delay from network input to network output is implemented by the EtherSound network, with analogue input to analogue output measuring a latency of 1.5 to 2 ms. EtherSound networks support Layer 2 (physical) peripherals and use standard CAT-5 or CAT-6 cables, fibre optic cables, switches, media converters, and other standard Ethernet components [Digigram, 2005]. An EtherSound implementation consists of a communications protocol and the use of this protocol in several hardware designs, referred to as ESnet reference designs [Digigram, 2003].

The communications protocol, referred to as the EtherSound protocol, makes use of EtherSound frames that are wrapped up in standard Ethernet frames. These frames consist of two main parts, namely, the *EtherSound header* and the *EtherSound payload*. The former includes all important information relevant to the protocol, while the latter specifies the actual data transmitted via the network.

An EtherSound payload is divided into a number of packets, with each packet consisting of a *packet header* and *packet data*. The packet header defines the packet type and structure, and the packet data specifies the actual data [Digigram, 2003]. The current version of the EtherSound protocol (VI) defines a payload of two packets [Digigram, 2003]. These are:

- The *command packet* that transmits 1 command, which can either be a control command or a status request command.
- The *audio packet* for transmitting up to 64 audio channels at 44.1 kHz or 48 kHz.

The EtherSound hardware, described by ESnet reference designs, provides the electronic schema and binary code of the FPGA that allows manufacturers to integrate EtherSound into their own design. Examples of ESnet reference designs include the

MSx 88 Eeprom 200K and the S2 Prom 100K. The MSx 88 Eeprom 200K reference design provides either 8 audio inputs or 8 audio outputs, while the S2 Prom 100K design provides 2 audio outputs. Each design implements two RJ45 ports – "Ethernet OUT" and "Ethernet IN" – that respectively send out and receive EtherSound frames in an EtherSound network.

Each EtherSound FPGA firmware has an internal database of 256 16-bit device registers, which provide configuration information status, or manage the behaviour of the EtherSound device. These device registers are grouped into three separate categories, namely:

- *Device descriptor registers* - Describes the status and the configuration of the EtherSound Kernel of the device.
- *I/O mapping registers* – Describes the behaviour of the EtherSound device audio I/Os (i.e. the EtherSound channel assignments to audio I/Os).
- *Core dedicated application parameters* – Reserved for specific optional functions not included by default in every EtherSound device.

A control application referred to as the ESControl Management Software [Digigram, 2004] is a configuration application provided by Digigram to set up audio routings between devices – detailed information about the application could not be obtained. However, Digigram also provides an API and SDK for 3rd party developers to implement EtherSound routing applications. StarDraw Control, developed by StarDraw.com Ltd. [StarDraw.com Ltd., 2005], is described as a control program that can control any remotely controlled or monitored hardware from any manufacturer, using any protocol over any communications infrastructure. This application also provides support for EtherSound-enabled devices on an EtherSound network. As an aside, CobraNet™ is soon to be supported by StarDraw Control.

A summary of the capabilities offered by EtherSound are as follows:

Latency	250 µs point-to-point latency, 1.5-2 ms for analogue input to analogue output.
Number of channels	Up to 64 channels of 24-bit/48 kHz PCM audio, plus

	embedded control and monitoring data.
Network transport	CAT-5 cabling. Daisy chain, star, both daisy chain and star.
Control and monitoring	ESControl Management software for audio routings
Open standard/Proprietary based	Proprietary based, but uses a standards based transport

Table 2-8: Summary of the capabilities offered by EtherSound

2.1.1.9 Oxford Technologies' SuperMAC

SuperMAC of Sony Oxford Technologies provides a bi-directional, point-to-point connection for multi-channel audio plus sample clock, over a single CAT-5 data network cable [Oxford Technologies, 2004]. It is based on 100 Mb/s Ethernet physical layer, and transports up to 48 channels of 24-bit audio at a sampling rate of 48 kHz – it is also capable of supporting other sampling rates and channel counts. It implements a 5 Mbit/sec auxiliary data channel that provides an Ethernet-like communication service for control data, and also has a link latency of about 63 µs.

The original protocol developed for SuperMAC transceiver nodes formed the basis of the AES50-2005 [AES Inc., 2005] standard. These transceivers are now fully compliant with the AES50-2005 standard, hence providing an easy and cost-effective way to integrate AES50 interconnections into audio products. Nine Tiles Networks are working with Sony Pro-Audio Lab to develop a SuperMAC-to-AES47 bridge (referred to as the Audiolink 2) that would allow integration with ATM networks [Oxford Technologies, 2005a] [Nine Tiles Networks Ltd., 2005a]. AES47 is published by the Audio Engineering Society (AES) Inc. as AES47-2002 [AES Inc., 2002] and specifies the transmission of digital audio over ATM networks.

Since SuperMAC transceivers provide point-to-point connections, a SuperMAC Router is used to allow individual SuperMAC nodes to be connected together into an audio network. A control application running on a PC can access control and status registers on all devices connected to the Router via SuperMAC connections. A schematic example of an interconnection that describes a SuperMAC network for a studio environment is shown Figure 2-4. Note that devices connected to the Router all have SuperMAC transceivers implemented.

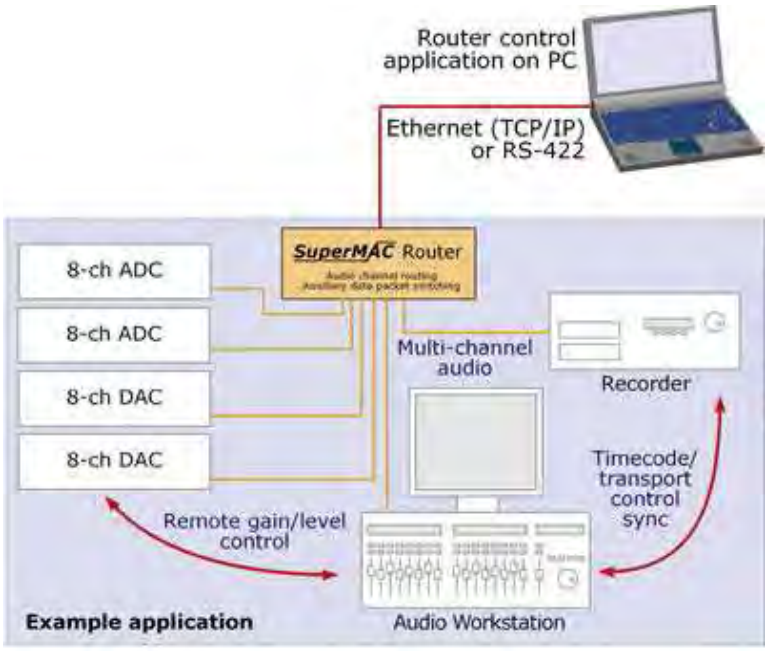


Figure 2-4: Example schematic of a SuperMAC connected network [Oxford Technologies, 2005b]

The router may be controlled from an application, typically running on a general-purpose computer or another studio device, via a TCP/IP-over-Ethernet interface. This application provides control over audio routing, automatically adapting to reflect the number of channels available on each port. A screenshot of a control application developed at Sony is shown below:

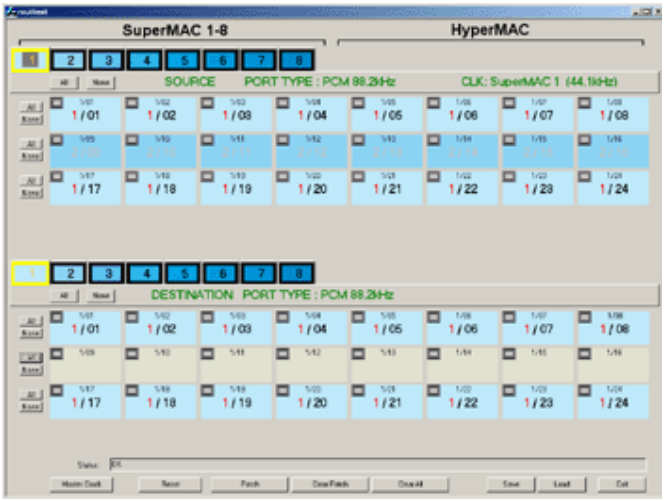


Figure 2-5: Routing application used in a SuperMAC network [Oxford Technologies, 2005b]

From this application, one can establish connections between ports of the SuperMAC Router, in so doing establishing an audio routing path from one device to another.

The HyperMAC technology – a work in progress – is also being developed by the team at Sony Pro-Audio Lab. It uses Gigabit Ethernet physical layer technology, on either CAT-6 data cable or optical fibre. It is a complementary technology to SuperMAC, supporting the same formats and providing a similar service, but with a much higher capacity. It is capable of transmitting up to 384 bi-directional channels. The audio channel capacity of HyperMAC is divided into eight streams. The eight streams may carry different audio formats at different sampling rates. Up to four independent asynchronous sample clocks can also be conveyed by HyperMAC, with each stream capable of being associated with any one of the four. The HyperMAC protocol is not standardized, but it is Sony's intention to submit the protocol specification for AES standardization in due course [Oxford Technologies, 2005c].

A summary of the capabilities offered by the SuperMAC technology are as follows:

Latency	Link latency of 63 µs
Number of channels	48 channels of 24-bit audio at a sampling rate of 48 kHz. Capable of supporting other sampling rates and channel counts
Network transport	CAT-5 cabling. Star topology.
Control and monitoring	No information on control and monitoring. Audio routing can be established using the SuperMAC router application.
Open standard/Proprietary based	Open-standard based, and uses a standards based transport

Table 2-9: Summary of the capabilities offered by the SuperMAC technology

2.1.1.10 Yamaha Corporation's mLAN Technology

The music Local Area Network (mLAN), a digital network interface technology created by Yamaha Corporation, Japan, allows professional audio equipment, PCs and electronic instruments to be easily and efficiently interconnected using a single cable

[Yamaha Corp., 2005b]. This technology uses the IEEE 1394 high performance serial bus as its transport medium, and is based on the IEC 61883-6 protocol.

The IEEE 1394 high performance serial bus is a multimedia connection that enables simple, low-cost, high-bandwidth isochronous (real-time) data interfacing between computers, peripherals, and consumer electronics products [1394TA, 2005]. The register architecture is based on the ISO/IEC 13213 specification, which is formally named "Information Technology – Microprocessor Systems – Control and Status Registers Architecture for Microcomputer busses" [Anderson, 1999]. The IEEE 1394 specification adds onto these features bus-dependent extensions.

A maximum of 65536 nodes is supported on an IEEE 1394 network. In addressing a node, a 64-bit addressing scheme, identical to that defined by the CSR specification[3] is used. This 64-bit addressing scheme is made up of a 16-bit node address and a 48-bit address that refers to the 256 terabytes of address space available within the node. The 16-bit node address is further divided into a 10-bit bus ID and a 6-bit node physical ID.

Two basic data transfer services are supported by an IEEE 1394 node; these are asynchronous and isochronous transfers. Asynchronous transactions, implemented as *read*, *write* and *lock* transactions, provide guaranteed data delivery (with confirmation) of variable-length packets to a specific memory address space. A read transaction returns data from a memory address, a write transaction writes data into a specified memory address, and a lock transaction performs atomic read-modify-write operations into a specified memory address. Isochronous packets on the other hand, allow for the transfer of variable-length packets at regular intervals. Isochronous transactions allow one *isochronous talker* to transmit a data stream to one *isochronous listener*. In order for this to take place, the talker is required to transmit on one of 64 isochronous channels available on the local bus segment, while the listener is required to receive on the isochronous channel being transmitted by the talker.

[3] The CSR specification is an abbreviation of the ISO/IEC 13213 specification

The original IEEE 1394 specification, IEEE Std. 1394-1995 [IEEE, 1995], has been enhanced with additional features and improved performance to create IEEE Std. 1394a-2000 [IEEE, 2000], IEEE Std. 1394b-2002 [IEEE, 2002] and the current work-in-progress IEEE p1394c specification [IEEE SA, 2005]. A description of these specifications is summarized in Table 2-10 below.

Specification	Description
IEEE Std. 1394a-2000	Created to clarify the different interpretations and hence interoperability issues resulting from the IEEE Std. 1394-1995 specification. This also adds additional features and makes improvements intended to increase performance or usability.
IEEE Std. 1394b-2002	Provides a supplement to IEEE Std. 1394-1995 and to IEEE Std. 1394a-2000 that defines features and mechanisms that provide gigabit speed extensions in a backward compatible fashion, and also the ability to signal over single hop distances of up to 100m.
IEEE p1394c	Defines features and mechanisms that provide gigabit speed using CAT-5 cabling over single hop distances of up to 100m. It is set to also provide minor updates to 1394b-2002.

Table 2-10: A summary of the IEEE 1394 specifications

Note that the IEEE Std. 1394-1995 (and also 1394a) defines data transfers speeds of 100 Mb/s, 200 Mb/s and 400 Mb/s, while the IEEE Std. 1394b-2002 defines speeds of 800 Mb/s, 1.6 Gb/s and 3.2 Gb/s. Bridging in an IEEE 1394 network is made possible through the IEEE Std. 1394.1-2004 bridging specification [IEEE, 2005], which defines the required facilities to enable interconnection of multiple IEEE 1394 busses.

The IEC 61883-6 specification, also referred to as the “Audio and Music Data Transmission Protocol”, is the data transmission protocol on which mLAN is based. This specification was originally developed by Yamaha Corporation and was later adopted as part 6 of the real time data transmission protocol, IEC 61883, which specifies a digital interface for electronic/audio equipment using IEEE 1394 [IEC, 2005]. Included within this specification are the mechanisms for the transport of IEC 60958 digital audio, raw audio and MIDI data. The transport mechanisms defined build on the isochronous transmission method of IEEE 1394, where audio and music

data (e.g. MIDI) is formatted in AM824 data quadlets[4] (8-bit label and 24-bit data) and transmitted as *sequences* within isochronous streams. Of the hardware chips that implement or are capable of implementing the IEC 61883-6 protocol, the mLAN-PH2 chip, developed by Yamaha has the highest audio transmit/receive capability of 32 channels of 24-bit audio at a sample rate of 48 kHz – transmission speed is 400 Mb/s. However, up to 4 of such chips may be combined, providing a 128 audio channel transmission and reception capability. A maximum transfer latency of 354.17 µs is defined by the IEC 61883-6 specification [IEC, 2005].

The current implementation of mLAN, *mLAN Version 2*, implements an Enabler/Transporter concept. A Transporter is implemented within an audio device and exposes a number of registers that allow audio and music transmission parameters to be configured. The Enabler, on the other hand, is a software application that is responsible for providing a number of high-level abstractions, including audio and MIDI plugs on behalf of Transporter nodes. The Enabler interfaces to Transporter nodes via a plug-in mechanism, thus facilitating other vendor-specific Transporter implementations. Yamaha Corporation has submitted an open Transporter specification, referred to as the Open Generic Transporter (OGT) specification [AES SC, 2005] to the Audio Engineering Society, which will allow easier device integration to mLAN. The OGT specification is currently an AES SC02-12G[5] working draft.

Device control on an mLAN network is currently performed through the use of MIDI messages, which are encapsulated within isochronous messages. Audio routings between devices is currently achieved using the mLAN Graphic Patchbay application. A snap-shot of the application is shown in Figure 2-6 below. Each device on the network is modelled to have a number of input plugs and output plugs. The output plugs reference a particular *sequence* contained within an isochronous stream. When a plug connection is made from an output plug of a device to an input plug of another device, the input plug is configured to receive the audio data that is transmitted by the output plug. This configuration occurs by specifying the isochronous channel and the sequence position of the *sequence* containing the audio data that is being transmitted.

[4] 32-bit data
[5] The AES Standards Committee (Task group on professional audio in 1394)

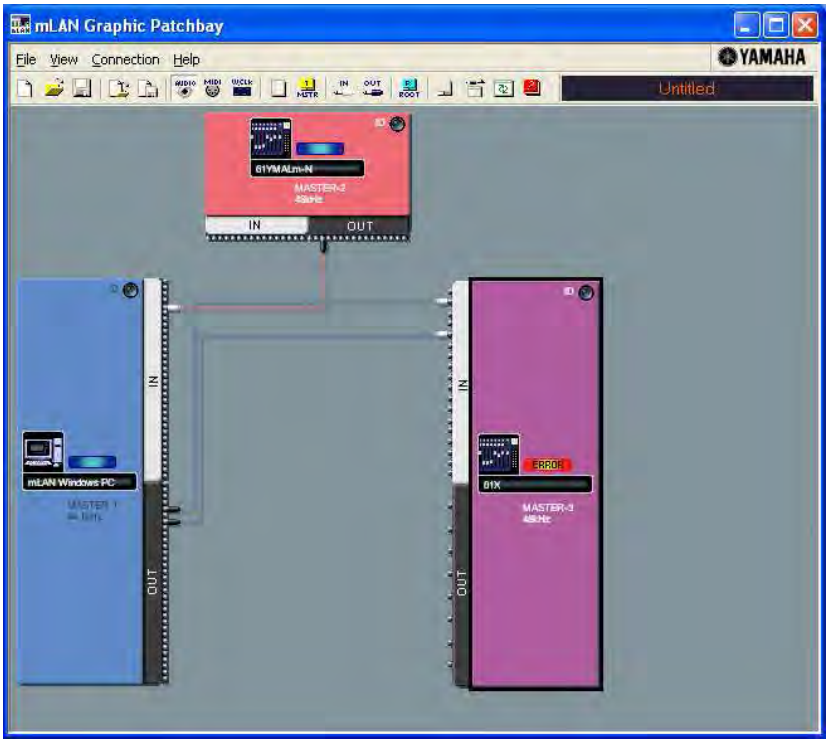


Figure 2-6: The mLAN Graphic Patchbay that enables plug routing

The software plugs of the devices shown on the patch bay can be directly or flexibly configured to the hard inputs and outputs found on a device. If a flexible configuration is implemented, the user has to manually configure the connected software plug of the patch bay to a device's hard input or output plug of choice. From this application, MIDI plug connections as well as word clock synchronization between devices can also be set up.

A summary of the capabilities offered by mLAN are as follows:

Latency	Maximum transfer latency of 354.17 µs
Number of channels	The mLAN-PH2 chip provides 32 channels of 24-bit audio at 48 kHz, but can provide a 128 audio channel capability with four such chips.
Network transport	Uses the IEEE 1394 standard. Daisy chain
Control and monitoring	MIDI messages used for control. The mLAN Graphic Patchbay used for audio routings.
Open standard/Proprietary based	Open-standard based, and uses a standards based transport

Table 2-11: Summary of the capabilities offered by mLAN

2.1.2 A Review of Current Audio Networks

It is evident from the networks described above that companies who develop audio networking technologies aim to tackle the problems associated with point-to-point cabling by first of all, providing a single (long) cable interconnect to transmit both audio and control data. Their aim is also to provide the low-latency, high-channel count and QoS required by the professional industry.

2.1.2.1 Low-Latency, Channel Count & QoS

Each network has its advantages and disadvantages, and the choice of one audio network over another is usually determined by the nature and demands of the application in which the network is to be used. For example, the latencies defined by CobraNet™ may make it unsuitable for use in live broadcast and music productions studios, where one usually monitors a number of real-time audio signals against a mixed signal. However it finds applications in large-scale sound systems such as stadiums and convention centres, where the delay between an original sound and the sound carried by the system is not a critical feature of the network.

The number of simultaneous transmissions one can have on a network is governed by the channel count offered by a network. Although most applications do not demand a network with high channel count, providing this feature facilitates network scalability, thereby allowing more devices to be configured for transmissions.

The QoS requirements for an audio stream are particularity demanding [AES TC-NAS, 1998]. Lost or late data packets can easily destroy the continuity of the audio signal stream, thus seriously degrading the quality of the sound. In achieving QoS, all digital networking components must be synchronized to a common word clock to ensure that audio data is delivered at a rate suitable for recovering sample rate clocks without jitter or dropouts, and maintaining proper time alignment between the signals [AES TC-NAS, 1998]. Delivering QoS is an important feature of an audio network, and hence audio network providers ensure that this capability is met with their networking solutions.

2.1.2.2 Convergence

There is also the issue of convergence, which was found not to be a universal requirement in the survey conducted by the AES Technical Committee on Networked Audio Systems (AES TC-NAS). The survey indicated that 49% of the respondents preferred a separate audio infrastructure, while 42% believed that there is value in convergence. Some designers, in the interest of quality and/or reliability, work to remove dependencies (such as data communications) on systems outside their control [AES TC-NAS, 2005]. As most of these networks are based on the Ethernet standard, they implement a customized version of the Ethernet standard causing it to give higher performance in delivering low latency and maintaining QoS. Michael Johan Teener discusses in his paper "Residential Ethernet: a status report" [Teener, 2005], why Ethernet as-is cannot be used for A/V networks.

In spite of the lack of consensus regarding convergence, the marketing behind audio network technologies use the issue in their favour as a selling factor. EtherSound, A-Net™ and AES50 require a dedicated infrastructure and, in most cases, tout that as an advantage. CobraNet™, AES47, mLAN and Axia, are designed to work on a shared network and tout that as an advantage [AES TC-NAS, 2005].

The mLAN networking technology, unlike the other audio networks, uses a transport base that is primarily designed for A/V networks – IEEE 1394, and hence allows IEEE 1394 to be used in its standard form, with regards to network topologies and sharing data traffic with other IEEE 1394 devices. The other Ethernet-based networking solutions that are designed to work on a shared network usually require a list of tested Ethernet components in order to achieve the performance claimed by their respective vendors.

2.1.2.3 Standards vs. Proprietary

Standards-based audio networks may be favoured over proprietary solutions, primarily because of the lack of licensing fees incurred by audio manufacturers in providing product support within their products. In doing so, they are guaranteed interoperability with other devices that implement the standard. In addition to this, open standards, before publicly released, are rigorously examined by a selected

committee of working group members in order to achieve a complete well-thought-of solution.

Current audio networking solutions such as CobraNet, EtherSound and SuperMAC employ licensing agreements that enable right of use of the intellectual property and/or proprietary information regarding these implementations. Yamaha Corporation's mLAN technology, although currently requiring a license agreement, is at the moment moving towards providing an open-standard specification. This specification, referred to as the Open Generic Transporter specification (see section 2.1.1.10), currently an AES SC02-12G working draft, will not require any licensing for OEMs to integrate to an mLAN network.

2.1.2.4 Control and Monitoring

One of the main points expressed from the survey conducted by the AES TC-NAS is the core importance of control and monitoring capability on a network. Audio networks, as seen from the networks discussed in section 2.1, provide a means by which devices can be controlled and monitored remotely. Although monitoring is not such a common feature, as compared to control, it plays a significant role in installed sound, where for example, the volume of sound and the temperature of devices can be monitored.

The control feature offered by a network enables audio routing to be set up between devices, as well as to control volume, programme selection and the play/pause capabilities of a device. The IEC project team 62379 [Nine Tiles Networks Ltd., 2005b] is currently developing a common control interface for networked audiovisual equipment that will provide, amongst other things, a consistent interface to the functionality in an audiovisual unit.

With regards to setting up audio routings between devices, a PC-based application is usually employed to configure routings between devices. In configuring routings, a two-step approach is adopted. First, the transmitting and receiving devices are configured at the transport layer of the network (using a PC-based application) to be able to transmit and receive audio from the network. The second step requires a

manual configuration of the source and destination plugs on the devices to transmit and receive data from the network node implemented within the device, which at this point would be already configured to transmit/receive from the network.

Audio networks, as much as they do away with the disadvantages of using physical point-to-point cabling, should enable a connection management or routing application to be developed that reflects the basic concepts used in physical point-to-point cabling. Enabling this would eliminate the need for devices to be configured after a routing is constructed from a control application. The connection management applications of the current audio networking solutions lack this feature, and it is a problem whose solution forms an important goal of this study.

2.2 End-to-End Connection Management

After analyzing the various audio networks described above, and considering the criteria used by the AES TC-NAS in surveying a number of audio networking solutions for a number of different applications, we feel that three important criteria have been omitted. These are:

1. True end-to-end connection management capability.
2. The speed of issuing and effecting end-to-end connection management.
3. Control over resource usage as a result of enabling end-to-end connection management.

As described in section 1.1.2.1, true end-to-end connection management provides the capability, from a user level application, to route audio/MIDI data from the hard-end plug of a device or a data-bus line implemented within a device, onto an audio network, and also the capability to retrieve audio/MIDI data from the network onto data-bus lines implemented within a device or to a device's hard-end plugs. The current audio networks provide routing capabilities only at the network's transport layer, and then require one to manually configure devices for the reception and transmission of audio (See Figure 1-2). Audio networks were primarily created to deal with the problems associated with point-to-point cabling, and in doing so, they also need to deal with the technical issues in the network transport of audio, such as multi-channel support, minimal latency, and delivering QoS. However, the connection

management capabilities or audio routing techniques provided by these networks are not as conceptually simple as point-to-point cabling, i.e. being able to connect from the hard-end plug of one device to the hard-end plug of another device. A complete audio networking solution should provide this capability.

The speed at which these end-to-end connections can be established or broken is also another important factor. Speed is generally a desired feature as far as any technology implementation goes. However, certain networked audio applications are time sensitive, and require operations to occur in the shortest possible time. For example, in broadcast studios, it is important that audio routings between plugs of devices can be switched quickly.

The number of simultaneous audio channels that can be transmitted on a network depends on the bandwidth available for transmissions. Gaining control over bandwidth resources enables a user to:

- Manually acquire bandwidth for dedicated transmissions, or
- Release unused bandwidth acquired by devices.

In addition, enabling end-to-end connection management permits bandwidth, as well as other resources, to be dynamically allocated as and when needed. Providing this capability, ensures that only the required amount of resource is reserved on a network for device transmissions.

The goal of this study is to provide a solution that will provide true end-to-end connection management, and incorporate other desirable features such as speed and bandwidth control, for an audio network that fairs well against the criteria adopted by the AES TC-NAS for audio networks. The audio network platform chosen for the development of this solution is Yamaha Corporation's mLAN technology. The reasons for this choice are given in the next section.

2.2.1 Why the mLAN Networking Technology?

The mLAN network, and in particular Otari ND-20B devices [Otari Inc., 2005] within the network, provides the potential for true end-to-end connection management. These

devices implement an interface that exposes to an mLAN network information regarding the hard-end plugs and word clock sources implemented within the device. An Enabler (recall the Enabler/Transporter concept from section 2.1.1.10) is then capable of modelling the host area of an ND-20B device, and can be responsible for patching its hard-end plugs to *sequences* contained within isochronous streams, which are to be transmitted onto the network. In doing so, isochronous resources (especially bandwidth) can be managed, thus allowing for optimal transmissions. In a similar manner, the Enabler can also be responsible for assigning particular *sequences* of audio that are being transmitted on the network to selected hard-end plugs of an ND-20B device. The word clock sources, both internal and external, that are implemented within the host area of an ND-20B device, can also be modelled and therefore configured by the Enabler. This will enable different word clock sources, as well as sample rates, to be selected and used. The Enabler, in modelling the word clock sources of devices, can also provide master/slave synchronization capabilities on behalf of these devices, thus eliminating the need for one to manually configure a device for word clock synchronization.

The Enabler/Transporter architecture of the current mLAN implementation (see section 2.1.1.10), also makes it possible to implement a fast switching capability for audio routings. Recall from section 2.1.1.10 that the Transporter nodes expose a number of configuration registers to an mLAN network. An Enabler, through a plug-in module, can access and modify the information contained within these registers. This is achieved via a number of high-level abstractions provided by the Enabler, on behalf of Transporter devices. In configuring Transporters, the Enabler can ensure that only the required registers are modified and in so doing, can increase the speed at which connection management requests are issued and effected.

In addition to the above mentioned reasons for choosing mLAN, mLAN fairs well against the criteria adopted for audio networks. It implements a maximum transport latency of 354.17 µs, and a total channel handling capacity that is dependent on the speed and sampling rate used for transmissions [Yamaha Corp., 2005c]. Table 2-12 below give figures on the channel count offered by an mLAN network. As 800Mb/s transmissions become more common, these channel counts will double.

Sample Rate [24-bit Audio]	200Mb/s	400Mb/s
44.1 kHz	100	200
48 kHz	87	174
88.2 kHz	50	100
96 kHz	43	87
176.4 kHz	25	50
192 kHz	21	43

Table 2-12: Total channel handling capacity of the mLAN network

QoS, through word clock synchronization, is achieved by using timing information that is transmitted as part of transmitting an audio stream. A device receiving audio data listens for this timing information, and uses it to reconstruct the sample rate in used by the transmitter. This technique is described by the IEC 61883-6 specification [IEC, 2005].

An mLAN network also supports standard networking topologies, and allows other IEEE 1394 nodes to co-exist on a network, without compromising its performance. The introduction of the IEEE standards 1394b-2002 [IEEE, 2002], p1394c [IEEE SA, 2005] and 1394.1-2004 [IEEE, 2005], enables an mLAN network to be scalable.

Another reason for choosing the mLAN technology is because it supports the drive for open standards. This is seen from its data transmission implementation, which is based on the IEC 61833-6 protocol, and also from the efforts to provide mLAN integration through the Open Generic Transporter specification [AES SC, 2005]. This specification is in the process of being standardized by the Audio Engineering Society.

In approaching the problem of high speed true end-to-end connection management, the current connection management architectures of mLAN will be analyzed and implemented, and their performance evaluated. With this basis, a new design and a number of implementation strategies for a solution that provides true end-to-end connection management will be devised, and the capabilities and speeds of this new implementation compared to that of the current design and implementations. It is

hoped that in addition to being able to perform end-to-end connection management, the new connection management solution, based on its improved design and implementation, will show a better speed and stability performance when compared to the design and implementations of the current mLAN connection management architectures. The descriptions of these architectures are given in the subsequent chapters.

Before describing the current mLAN connection management architectures, the next chapter, chapter 3, highlights the architectural features of IEEE 1394 that facilitate the transfer of audio and MIDI data, and permit a controller application to enumerate, reconstruct and represent the topology of an IEEE 1394 network. The IEEE 1394 bridge model and the IEC 61883-6 data transmission protocol are also described.

2.3 Summary

This chapter reviews a number of audio networks against the criteria formulated by the Audio Engineering Society – Technical Council on Networked Audio Systems (AES TC-NAS). From the review it is found that audio networks are designed, to a greater or lesser extent, to deliver the desired latency, multi-channel support and the QoS required by various applications. However, they lack the capability of true end-to-end connection management. Yamaha Corporation's mLAN digital interfacing technology is identified as the audio network to enable such a capability. This is made possible using the Otari ND-20B devices (see section 2.2.1), with a high-speed implementation given by the Enabler/Transporter model currently employed by mLAN, in addition to the high data transmission rate (currently 400Mb/s) of its transport medium – IEEE 1394. Other factors that make mLAN desirable include its convergence capabilities, and the standardization efforts, via the Audio Engineering Society, towards creating an open-standard specification - the Open Generic Transporter specification. The rest of this thesis focuses on realizing a true end-to-end connection management solution on an mLAN network. A brief overview of the IEEE 1394 architecture and the related technologies that enable end-to-end connection management is given in the next chapter.

Chapter 3

3. IEEE 1394 and Related Technologies

As mentioned in the previous chapter, Yamaha Corporation realized the potential of using the features of IEEE 1394 in creating an audio network. The mLAN concept was formed and a software protocol and a number of corresponding hardware chips were developed. The software protocol is described by the open standard IEC 61883-6, also known as the Audio and Music Data Transmission protocol. This protocol describes how audio and MIDI data should be transmitted isochronously over IEEE 1394. The hardware chips developed include the mLAN-NC1, mLAN-PH1 and mLAN-PH2. This chapter gives further information on the architecture of IEEE 1394, the IEC 61883-6 protocol, as well as the hardware chips developed as a result of the mLAN initiative. The features of IEEE 1394 described include the node architecture, node addressing, transaction types, control and status registers and the configuration ROM formats. Also, a mention is given of how bus configuration of IEEE 1394 nodes occurs; this demonstrates how a controlling application would reconstruct a topology for a given network, based on information it reads from the network.

In addition to the above, the IEEE 1394 bridge model described by the IEEE 1394 Bridge standard, as well as the application support specification for the proprietary 1394 NEC bridges, is also given by this chapter. In so doing, highlighting the various

core features that allow for across-bus forwarding of isochronous streams, hence enabling across-bus plug connection management.

3.1 Overview of IEEE 1394

IEEE 1394 provides a serial bus interconnection that allows a wide variety of high performance peripherals to be interconnected. The primary characteristics of this serial bus include:

- Plug and Play support
- Ease of use and low cost device implementations
- High speed application support with scalable performance
- Support for isochronous applications

Devices attached to the IEEE 1394 serial bus support automatic configuration. Each time a new device is added or removed from the bus, the IEEE 1394 bus is re-enumerated without intervention from a host system. This is described more extensively in section 3.1.2. The serial bus provides scalable performance by supporting transfer rates of 3.2Gb/s, 1.6Gb/s, 800Mb/s 400Mb/s, 200Mb/s and 100Mb/s.

The IEEE 1394 specification is based on the ISO/IEC 13213 (ANSI/IEEE 1212) specification [IEEE, 2001]. This specification, formally named "Information technology- Microprocessor systems-Control and Status Registers (CSR) Architecture for microcomputer buses," defines a common set of core features that can be implemented by a variety of buses [Anderson, 1999]. These features include:

- Node Architecture
- Address space
- Transaction types
- Control and Status Registers
- Configuration ROM format

The ISO/IEC 13213 (ANSI/IEEE 1212) specification will be referred to as the CSR architecture in the remainder of this chapter.

3.1.1 Node Architecture

The IEEE 1394 bus architecture is defined in terms of *nodes, units* and *modules*. A node represents a logical entity within a module that is visible to initialization software, and also contains control and status registers and ROM entries. A unit represents the functional subcomponent of a node that may identify either processing, memory or I/O functionality. Units within a node usually operate independently and are controlled by their own software drivers. A module represents the physical device attached to the bus and is capable of containing one or more nodes. An example node architecture layout is given in Figure 3-1.

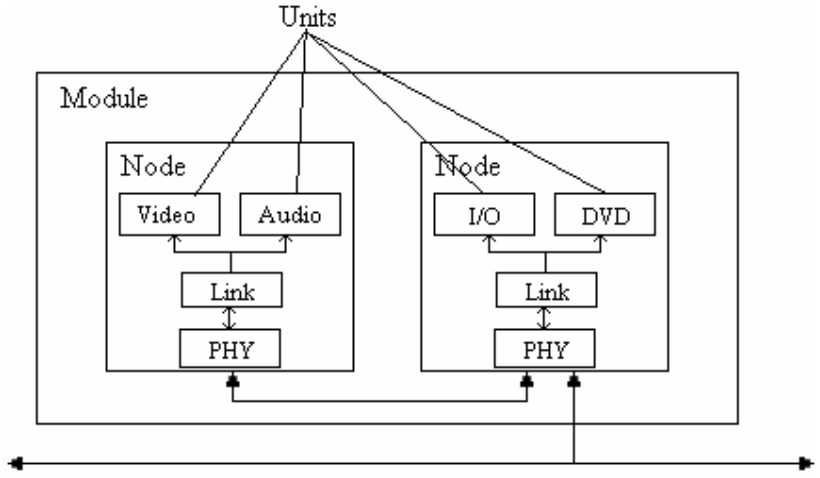


Figure 3-1: An example node architecture layout

The module shown in the diagram consists of two nodes. Each node implements two units namely *video* and *audio*, and *I/O* and *DVD* respectively.

3.1.1.1 Node Addressing

The 1394 architecture follows the CSR specification for 64-bit addressing. In this addressing scheme the most significant 16 bits of the address represent the ID of the node, and the remaining 48 bits address a particular region within the 256 terabyte address space available to each node. The 16-bit ID is subdivided into a 10-bit bus ID and a 6-bit physical ID, allowing for a maximum of 1023 buses each with 63 independently addressable nodes. The structure of the 64-bit address is given in Figure 3-2.

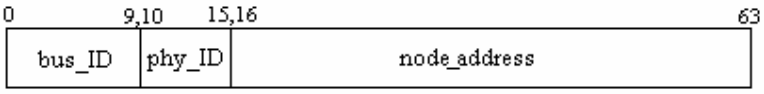


Figure 3-2: Structure of the IEEE 1394 64-bit addressing

The 256 terabyte address space allocated to an IEEE 1394 node is divided into various blocks defined for specific purposes. These blocks include:

- Initial memory space
- Private space
- Initial register space
  - CSR architecture register space
  - Serial bus space
  - ROM (first 1KB)
- Initial unit space

The initial register space provides standardized locations used for serial bus configuration and management, while the private space is reserved for a node's local use. Figure 3-3 shows the total 64-bit addressing scheme for the 1394 implementation.

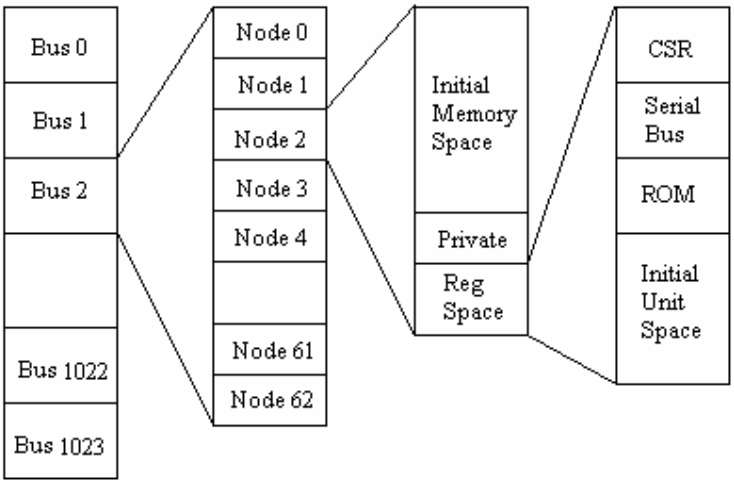


Figure 3-3: Serial bus address space

3.1.1.2 Transaction Types

The IEEE 1394 specification supports two types of data transfer: *asynchronous* and *isochronous*. Asynchronous transfer provides a means by which aperiodic data with

guaranteed delivery is sent across the serial bus. This is used by applications requiring confirmation of data delivery, usually in situations where critical data is used. Conversely, isochronous transfer is useful in situations that do not require data confirmation but rather data delivery at a constant rate. The nature of these transfer types are discussed in the following paragraphs.

Asynchronous Transfer

In asynchronous transfers, a transaction, defined as a request packet with a corresponding response, is conducted between two nodes. These transactions are initiated from a requester and are received by a responder. The responder node processes the request and sends an appropriate response to the requester. Three types of asynchronous transactions are defined by the 1394 specification. These are *read*, *write* and *lock*. A read transaction retrieves the contents of a specified memory address from the responder node. A write transaction writes to a specified memory address on the responder node, and a lock transaction provides a mechanism that permits atomic read-write-lock operations.

Various asynchronous packets are defined for these transactions: 4 for asynchronous read, 3 for asynchronous write and 2 for asynchronous lock. The packets defined for asynchronous read include *read data quadlet request*, *read data quadlet response*, *read data block request* and *read data block response*. The packets defined for asynchronous write include *write quadlet request*, *write data block request* and *write response*. The packets defined for asynchronous lock include *lock request* and *lock response*. Don Anderson [Anderson, 1999] gives a detailed description of the structure of these packets. The structure of a typical asynchronous packet is shown in Figure 3-4.

The fields, *destination_ID* and *destination_offset*, both define the 64-bit memory address, as illustrated in Figure 3-2. The transaction label field, *tl*, is specified by the requester to identify the transaction. The priority field, *pri*, is not used in the cable environment. The source identifier field, *source_ID*, identifies the node sending the packet. The retry code field, *rt*, specifies whether the packet is an attempted retry and

defines the retry phase. The transaction code field, *tcode*, defines the type of transaction.

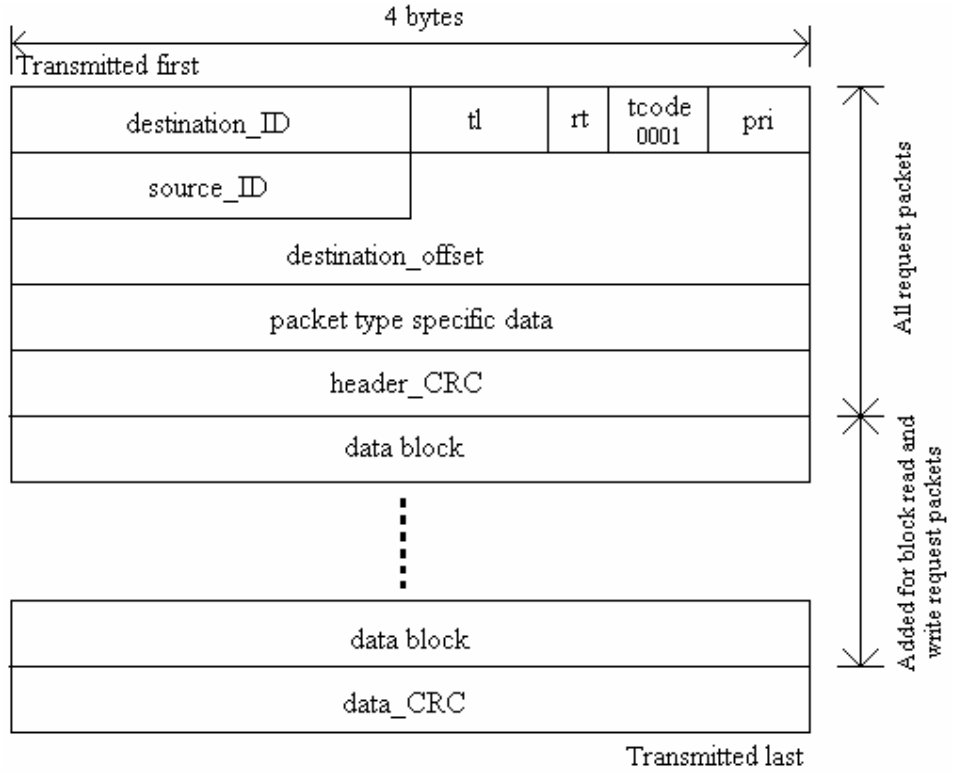


Figure 3-4: Asynchronous packet structure

Asynchronous transactions are guaranteed 20% of the bus bandwidth. A given node is not guaranteed bus bandwidth, but rather guaranteed fair access to the bus, in which each node wishing to perform asynchronous transactions gets access to the bus exactly one time during a single *fairness interval*.

Isochronous Transfer

In isochronous transfers, data flow is unidirectional. In this case, isochronous data, identified by a 6-bit channel number, is broadcast on a bus. Nodes intending to receive isochronous data are required to listen on the specified transmitting channel. Sixty-four (0 to 63) channels are available on which isochronous data can be transmitted or received. The initiator of the isochronous transaction is termed the *talker* and the targeted node termed a *listener*. Nodes wishing to perform isochronous transfers, must request the required bandwidth from the *isochronous resource manager* node. Once the bandwidth has been acquired, the transmitting channel is

guaranteed up to 80% of each bus cycle. A bus cycle lasts 125 μs and is initiated by cycle start packets generated by the *cycle master* node.

Isochronous transactions make use of a single data packet to perform broadcast operations. Figure 3-5 shows the structure of an isochronous packet.

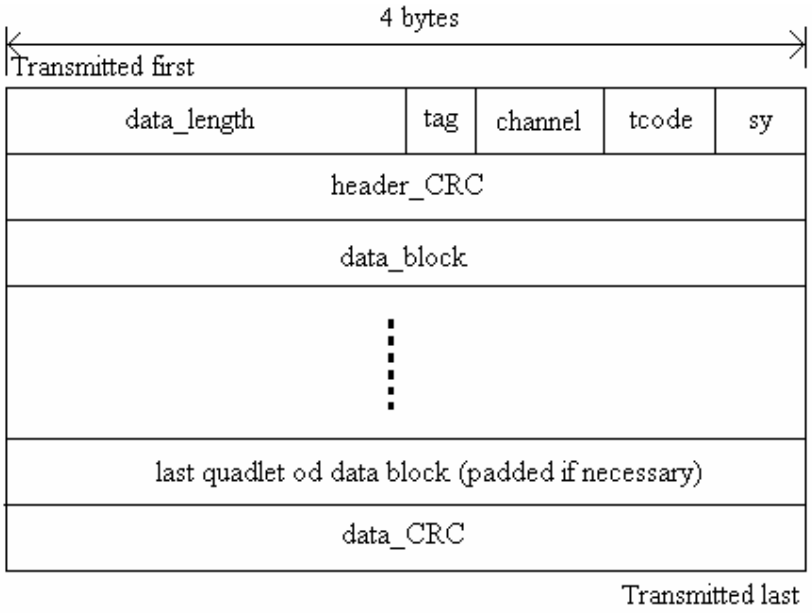


Figure 3-5: Isochronous packet structure

The field *data_length* specifies the number of bytes contained within the *data_block* field of the isochronous packet. The *tag* field, indicates the isochronous data format tag, and is used by higher level protocols. The isochronous channel**,** indicated by the *channel* field, specifies the isochronous channel assigned to the packet. The transaction code field, *tcode*, is defined to be Ah for isochronous transactions. The synchronisation code, *sy*, is application specific, and is utilised by the 5C digital transmission copy protection standard [DTLA, 2005].

3.1.1.3 Control and Status Registers

The control and status registers provide a standard definition for easier implementation and interoperability of devices. All IEEE 1394 nodes implement a group of control and status registers defined by the CSR architecture. These registers are implemented as defined in Table 3-1.

Offset (h)	Register Name	Description
000	STATE_CLEAR	State and control information
004	STATE_SET	Sets STATE_CLEAR bits
008	NODE_IDS	Specifies 16-bit node ID value
00C	RESET_START	Resets state of node
018-01C	SPLIT_TIMEOUT_HI SPLIT_TIMEOUT_LO	Split request timeout
200-3FC	SERIAL BUS DEPENDENT	IEEE 1394 unique serial bus CSRs

Table 3-1: The CSR register space implemented by 1394 nodes

In addition to these registers, 1394 nodes also implement serial bus dependent control and status registers defined by the IEEE 1394 standard. These registers provide bus-specific extensions to the CSR specification, and are implemented as defined in Table 3-2.

Offset (h)	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. Extends the CYCLE_TIME register.
208	POWER_FAIL_IMMINENT	Used to notify nodes 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 isochronous channel usage
22C	MAINT_CONTROL	Used for diagnostics
230	MAIN_UTILITY	Used for debugging
234-3FC	Not used	Reserved for future use

Table 3-2: Serial bus dependent CSR registers

Of particular interest to audio and MIDI data distribution, are the BANDWIDTH_AVAILABLE and CHANNEL_AVAILABLE registers. The use of these registers is discussed within the context of isochronous resource management further on in this thesis.

The registers described are the standard registers for 1394 bus operations. Specific protocols may define additional registers, as in the case of IEC 61883-6, which is discussed in section 3.2. Don Anderson [Anderson, 1999] gives a detailed account of the possible values for the standard CSR registers.

3.1.1.4 Configuration ROM

The CSR architecture defines a standard set of ROM entries that specify configuration information to be used during node initialization. The information included within the ROM includes information for:

- Identifying software drivers for devices
- Identifying diagnostic software
- Specifying bus-related capabilities of devices
- Specifying optional modules, nodes and unit characteristics and parameters

Two ROM formats are defined; *minimum* and *general*. The minimum ROM consists only of a 24-bit Vendor-ID value. The most significant 8 bits contains a value of 01h, which identifies the ROM format as minimal. Any other value is interpreted as a general ROM format. The general format specifies the vendor identifier, a bus information block, and a root directory containing information entries and/or pointers to other directories. Don Anderson [Anderson, 1999] describes the layout of the configuration ROM in more detail, as well as possible values within the configuration ROM space.

The configuration ROM enables a process called *enumeration* to take place [Laubscher, 1999]. Device enumeration, typically initiated by a 1394 equipped PC, is the ability for the PC to identify and appropriately handle any attached 1394 devices. This technique forms a core part of this research and will be revisited.

It was mentioned at the beginning of this section that nodes on an IEEE 1394 bus are able to reconfigure themselves, without the intervention of a host system. This enables high-level software, usually in the form of a controlling application running on a 1394 equipped PC, to monitor the bus properties of 1394 nodes on the network. The next section briefly describes the procedures involved in the configuration process, and how physical information, such as the bus topology, can be accessed by high-level applications. Note that this is a review of what is given in the IEEE Std. 1394-1995 [IEEE, 1995].

3.1.2 Configuration Process

Configuration of 1394 devices occurs locally on the serial bus without the intervention of a host processor. This occurs each time a new device, or node is attached or removed from the serial bus. Three primary procedures: *bus initialization, tree identification* and *self identification* are performed during the configuration process. These procedures are discussed below.

3.1.2.1 Bus Initialization

Bus initialization, also known as bus reset, forces each node on a serial bus into its initialization state and hence, begins the configuration process. This is invoked either by software control or as a result of hardware events, which mainly includes attaching or removing nodes from the 1394 bus, or power cycling devices. The degree of initialization performed by nodes during a bus reset depends on the source of the reset. The primary effects include clearing the topology information held by each port of the node, and also resetting some of the CSR register values.

Figure 3-6 illustrates a number of 1394 nodes with the topology established and a subsequent view of the topology immediately following a reset. Note that after reset the ports of the nodes lose their parent/child topology information. Reset initializes the bus and prepares each node to begin the tree identification process.

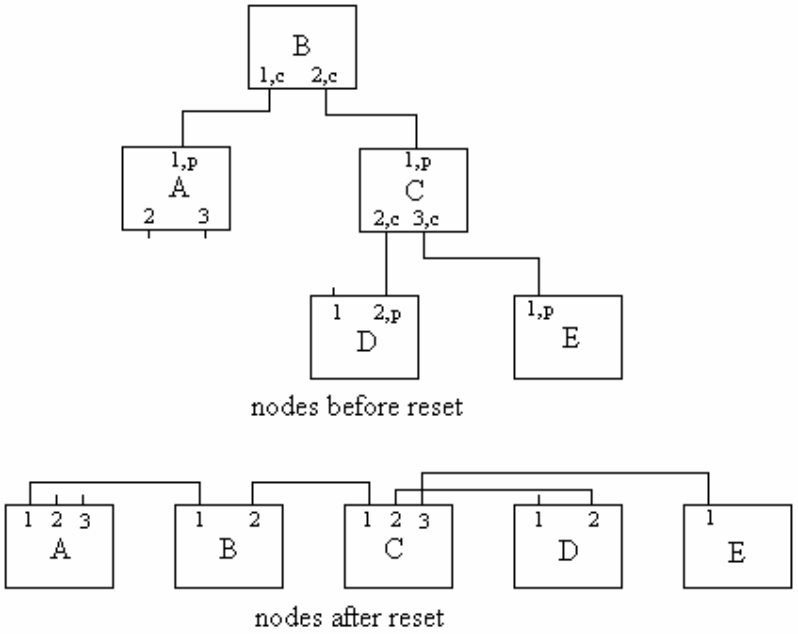


Figure 3-6: Illustration of bus topology after bus reset

3.1.2.2 Tree Identification

The tree identification process determines the topology of the bus according to the current nodes attached to it. This results in each port of a node being identified as either a parent or a child. A port identified as a parent means that the node at the other end of the cable is closer to the root, and that node will have identified its port as a child. Conversely, any port identified as a child is attached to a node that is further away from the root. The root node is defined to have all its connected ports identified as children. Don Anderson [Anderson, 1999] describes the tree identification process and defines how the root node is selected.

As an example, assume a node identified as F is attached to node D of Figure 3-6. This generates a bus reset, clearing the topology information previously attained. The state of the bus at bus reset is shown in Figure 3-7.

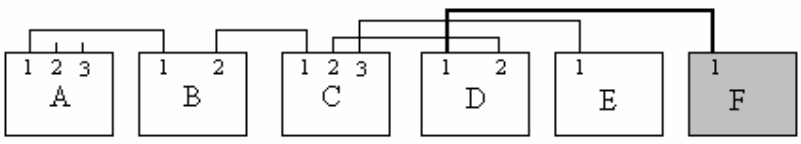


Figure 3-7: Bus reset state after node F is added to the bus

After the tree identification process, the topology of the bus may look like that shown in Figure 3-8. It is assumed that node D is identified as the root node.

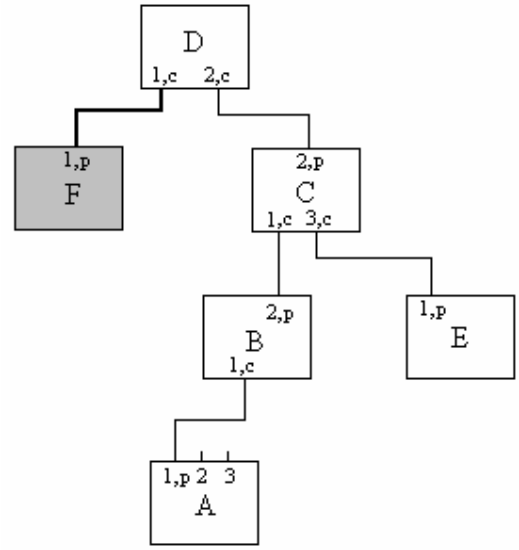


Figure 3-8: Example bus following tree identification

Once the tree identification has completed, the root node initiates the self-identification process.

3.1.2.3 Self Identification

During self identification (self-ID) a node is given the opportunity to select a unique logical ID that identifies its PHY. This ID is referred to as the *physical ID*. In addition to this, the self identification process allows a node to reveal its serial bus characteristics to any management entity attached to the bus [IEEE, 1995]. The serial bus characteristics include codes for power required to turn on the attached link layer as well as codes that indicate the node's data rate limitations. At the start of the self-ID process each node would have identified whether its port connects to a child or a parent port. The root node initiates the self-ID process by sending a self-ID grant to all nodes, beginning with its lowest numbered connected port. A node that receives a self-ID grant forwards it to the child node attached to its lowest numbered port. This process continues until the self-ID grant is received by a leaf node or by a node that has all it child nodes identified. At this stage, the node broadcasts a self-ID packet.

The physical ID of a node is simply the count of the number of times a node passes through the state of receiving self-ID information before having its own opportunity to do so. The first node to broadcast a self-ID packet assigns its physical ID to be 0; all other nodes monitor self-ID grant transmissions and keep track of the self-ID count. In this way each node identifies itself, and a layout of the bus topology can be generated.

The self-ID process following the bus example shown in Figure 3-8 is shown in Figure 3-9. The root node, D, would have observed the first self-ID grant. Node D and all other nodes that receive the self ID grant forward it to their respective child nodes, starting with the lowest number port. The number shown alongside each node represents its physical ID, which indicates the number of times the node receives the self-ID grant before transmitting a self-ID packet.

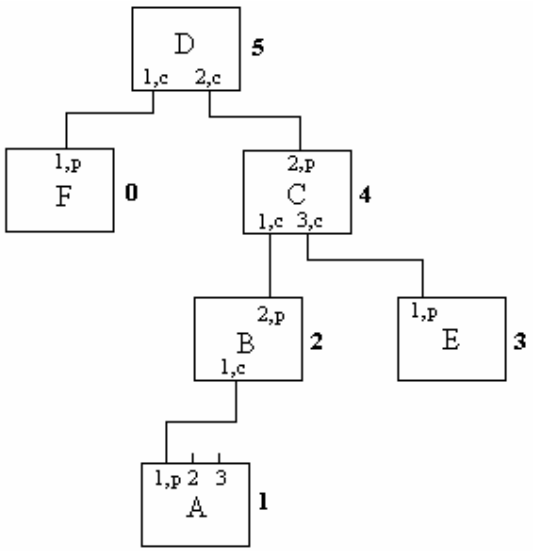


Figure 3-9: Example bus following the self-ID process

Self identification is the last of the configuration processes that occurs on a bus. At this stage, each node would have identified itself and its serial bus properties, and would also be aware of the parent/child connection state of its ports. Following this, the bus topology can be reconstructed by a management node attached to the bus. The next subsection describes how the 1394 architecture allows the bus topology to be regenerated.

3.1.2.4 Reconstructing the Bus Topology

The bus topology is reconstructed from the self ID packets of the 1394 nodes attached to the bus. More specifically, from observing the values contained within the *physical ID* field and the *port number* fields of the device's self ID packet. These specify information about the physical ID assigned to a node and also the connection status of its ports. Three types of self ID packets are defined by the IEEE 1394 standard: *self ID packet zero, self ID packet one* and *self ID packet two*. The structure of self ID packet zero is shown in Figure 3-10. Don Anderson [Anderson, 1999] describes the structure of self ID packet one and self ID packet two.

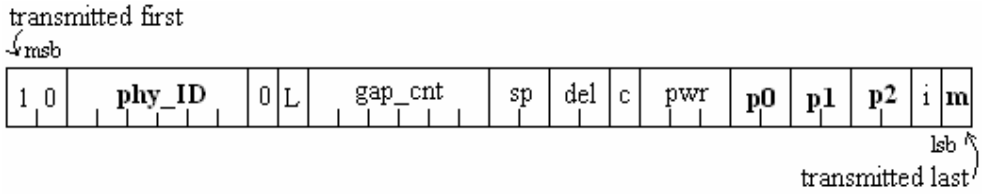


Figure 3-10: Structure of self ID packet zero

Of particular relevance to topology reconstruction are the fields labelled *phy_ID, p0, p1, p2* and *m*. The description of these fields is summarized in Table 3-3. The other fields specify the serial bus characteristics of a node and are described in [Anderson, 1999].

Field Code	Field Name	Comments
phy_ID	Physical ID	Physical identifier of the node sending the packet.
p0, p1, p2	Port number	Specifies port status: 11 = Active and connected to child node 10 = Active and connected parent node 01 = Not active 00 = Port not present
m	More packets	Specifies whether more packets follow to report additional port status.

Table 3-3: Fields description of self-ID packets

It is useful to mention that the *sp* field specifies the speed capabilities of a node as defined below:

00b = 98.304 Mb/s
01b = 98.304 Mb/s and 196.608 Mb/s
10b = 98.304 Mb/s, 196.608 Mb/s and 393.216 Mb/s

Figure 3-11: Node speed capabilities

The use of speed capabilities with regards to obtaining speed map information between 1394 nodes will be revisited later in this thesis. An illustration of how the bus topology is reconstructed using self-ID packets is given in the next paragraph. This example has been adapted from the IEEE Std. 1394-1995 [IEEE, 1995].

Consider the network example shown in Figure 3-12. The port information of the nodes as specified by their self-ID packets is summarized in Table 3-4 that follows.

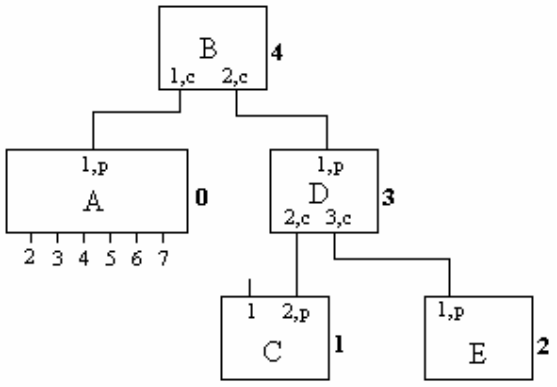


Figure 3-12: Example bus topology after performing self identification

phy_ID	Port 1	Port 2	Port 3	Port 4	Port 5	Port 6	Port 7	Port 8:16
0	parent	unconn	unconn	unconn	unconn	unconn	unconn	unconn
1	unconn	parent	noport	-	-	-	-	-
2	parent	noport	noport	-	-	-	-	-
3	parent	child	child	-	-	-	-	-
4	child	child	noport	-	-	-	-	-

Table 3-4: Port information of the 1394 nodes of the bus topology example shown in Figure 3-12

a) The node A (phy_ID 0) represents a seven port PHY, with a parent connection on port 1, and no other connections. Thus it is a leaf node.

b) The node C (phy_ID 1) represents a two port PHY, with a parent connection on port 2, and no other connections. Thus it is also a leaf node.

c) The node E (phy_ID 2) represents a single-port PHY, which by definition must be a leaf node. Its sole connection is a parent connection.

At this point, information about the three leaf nodes A, C and E have been retrieved. This is shown below:

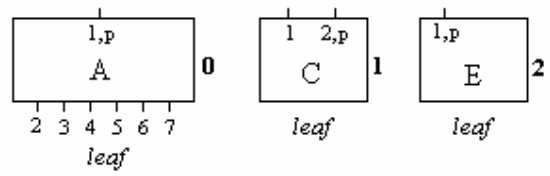


Figure 3-13: Topology information after identifying the leaf nodes

d) The node D (phy_ID 3) represents a three port node with connections on all its three ports. Port 1 is the parent port; ports 2 and 3 are child ports. In order to determine which of the previous nodes connects to these child ports, the technique involved in the self identification process is used, where the self-ID grant is first forwarded to the lowest numbered port of a node. This indirectly implies that the highest ranking node must be connected to the highest numbered next available child port. From this, child port 3 of node D must be connected to the leaf node E, because it has the highest physical ID value. Also child port 2 must be connected to leaf node C. This is shown in Figure 3-14.

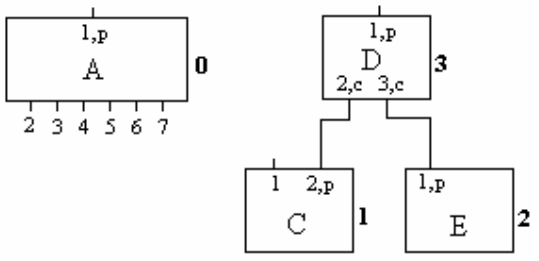


Figure 3-14: Topology information after identifying the first branch node

e) The node B (phy_ID 4) represents a two port PHY with connections on both ports. By definition, these ports are all child ports. Following from the previous step, it implies that child port 2 must be connected to node D and

child port 1 must be connected the node A. The final reconstructed topology is identical to the original topology shown in Figure 3-12.

This section gave a brief overview of the IEEE 1394 standard, highlighting various features of the node architecture and configuration procedures that facilitate the use of 1394 in the management of devices, and also, in distributing audio and control data through isochronous and asynchronous transmissions. It was mentioned that the 1394 standard is capable of supporting a maximum of 1023 buses each containing 63 addressable nodes. 1394 Bridges are used to achieve this. The next section takes a look at the current IEEE 1394 bridges, highlighting the features they provide in facilitating network enumeration, and asynchronous and isochronous transmissions in a multi-bus environment.

3.2 The IEEE 1394 Bridge Model

IEEE 1394 bridges were implemented to build on the multi-bus feature defined by the IEEE 1394 standard. They achieve this by successively increasing the number of IEEE 1394 buses that exists on a network. A bridge consists of two 1394 nodes, also known as *bridge portals*, each connected to a separate bus. These bridge portals are capable of forwarding asynchronous and isochronous packets to the buses attached to each other's adjacent portal. This view is conceptualized in Figure 3-15.

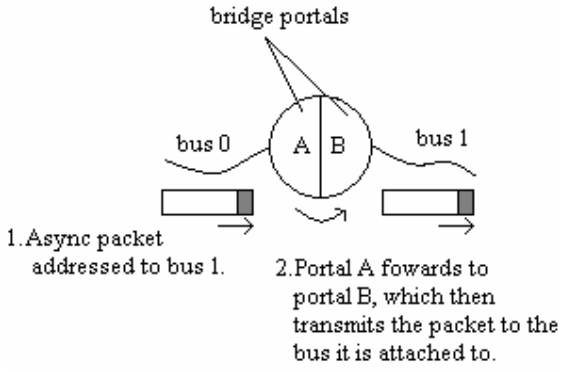


Figure 3-15: Conceptual representation of the operation of IEEE 1394 bridges

3.2.1 The IEEE 1394 Bridge Standard

The 1394 bridge specification, "IEEE Standard for High Performance Serial Bus Bridges" [IEEE, 2005], also referred to as the 1394.1 specification, defines the standard that is intended to model the definition and behaviour of serial bus bridges. In this specification, a bridge is required to consist of two bridge portals (each with its associated PHY and link), FIFO queues for handling isochronous and asynchronous subactions, cycle timers, route maps, and configuration ROM. The FIFO queues form an implementation-dependent *fabric* between the two portals. This model is shown in Figure 3-16.

Each bridge portal is a separate serial bus node. They each have a different *global unique identifier* (described in section 3.2.1.1) as well as an address space to which asynchronous transactions can be directed. In addition to this, they also monitor all serial bus subactions, asynchronous and isochronous, and use route maps to determine which subactions, if any, are to be routed through the bridge's fabric to the adjacent portal. The adjacent portal is also known as the *co-portal*.

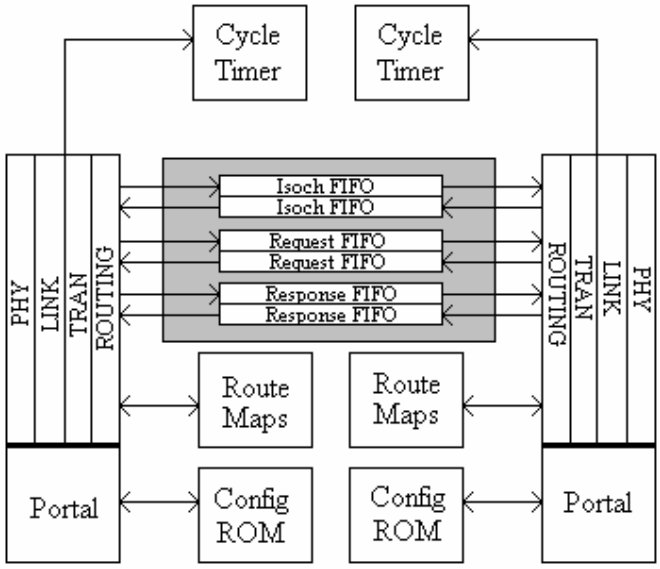


Figure 3-16: IEEE 1394 Bridge model

The bridge's fabric, shown enclosed in grey, interconnects the bridge portals and is conceptualized as consisting of a set of FIFOs that support bidirectional, non-blocking transfer of asynchronous request subactions, asynchronous response subactions and

isochronous subactions. Each portal has a cycle timer driven by a 24.576 MHz oscillator. The route maps contain information that permits the selective forwarding of both asynchronous and isochronous subactions through the fabric to the co-portal.

Various terminologies and concepts are introduced by this specification. They include global node IDs, remote timeout, clan affinity and net update, cycle time distribution and synchronization, and stream connection management. These are reviewed in the following subsections.

3.2.1.1 Global Node IDs

Global node IDs are defined to identify 1394 nodes in a bridged network. Just like node IDs, specified by IEEE 1394, they are used as source and destination identifiers within asynchronous transactions. These IDs are 16 bits wide and are each made up of a 10-bit bus ID and a 6-bit virtual ID. The virtual ID is equivalent to the physical ID specified by IEEE 1394, but is not assigned by the self identification process and does not necessarily change upon a bus reset. The virtual ID is assigned and managed locally on each bus by the *coordinator* while bus ID is assigned and managed centrally for the entire net by the *prime portal* (see section 3.2.1.3).

3.2.1.2 Remote Timeout

Timeouts for local and remote transactions are different. In remote transactions there is naturally a longer time taken to complete a transaction, which is due to intervening bridge portals along the path from the requester to the responder. This longer time period gave rise to the definition of a remote timeout analogous to the split time-out of a local bus. The remote timeout value is path dependent and is obtained by using a bridge management message sent to a target bridge. The 1394.1 specification defines the TIMEOUT bridge management message that is used for this purpose.

3.2.1.3 Clan Affinity and Net Update

Clan affinity is the term used to describe how coordination is effected when two independent networks of buses are connected. The bridge portals that constitute a network form a clan. When portals of two independent clans are connected, the two clans have to be resolved by merging one clan into the other. During this process, a

bridge portal, located on the bus on which the reset occurred as a result of the connection, is selected to act as the coordinator to manage net update. Net update refers to the process that updates net configuration after a change in topology. The coordinator detects the presence of the two clans and performs the necessary procedure to resolve the clans. This procedure is described by the 1394.1 specification.

The notion of a clan introduces two other terminologies: a prime portal and an alpha portal. A clan has one prime portal which is used in identifying the clan. A prime portal is also responsible for maintaining a central registry of allocated bus IDs and also for identifying the *net cycle master*. See section 3.2.1.4. Alpha portals are portals on the route from the local bus towards the prime portal and are useful in routing inter-bridge messages to the prime portal, without knowledge of the prime portal's global ID. The prime portal, alpha portal terminology is illustrated in the figure below.

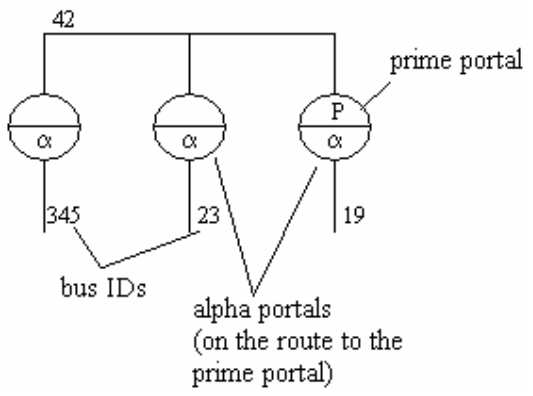


Figure 3-17: Network topology illustrating prime portals and alpha portals

3.2.1.4 Cycle Time Distribution and Synchronization

The net cycle master is the singular cycle master that provides cycle offset for the entire net; its location is determined by the location of the prime portal, and is usually the cycle master connected to the same bus as the prime portal. Net cycle offset originates from the net cycle master and is distributed throughout the net by bridge portals. On the bus that contains the prime portal, all portals obtain cycle time from the net cycle master and communicate it to their co-portals which in turn regulate cycle start events on their own buses. On all buses except the prime bus – the bus on

which the prime portal is attached, the alpha portal receives cycle synchronization information from its co-portal, and the subordinate portals (a portal that is neither the prime portal nor an alpha portal) receive cycle start information from the cycle master – and in turn pass it to their co-portals.

Frequency synchronization is also defined by the bridge standard. It is necessary in order for bridges to reliably transport isochronous streams, thus preventing overrunning or under running of isochronous data. A feedback loop is employed to maintain phase synchronization between two adjacent buses. This illustrated using the figure below:

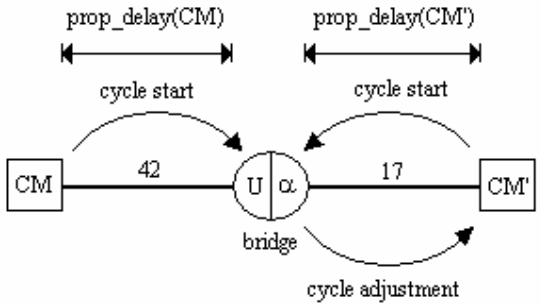


Figure 3-18: Illustration of cycle time synchronization as defined by 1394.1

The figure above shows the cycle master nodes attached to the bus of the adjacent portals of a 1394 bridge. The upstream and downstream cycle masters (with respect to the net cycle master) are labelled CM and CM' respectively. Synchronization is achieved by first determining the phase difference between the two adjacent buses. This is calculated by taking the difference between the sum, for each bus, of the cycle time offset sampled from the cycle master node, and the propagation delay from the cycle master node to a bridge portal. This is given by the formula:

$$\Delta phase = (CYCLE\_OFFSET_{\alpha} + delay_{CM'}) - (CYCLE\_OFFSET_{U} + delay_{CM}) \quad (1)$$

If a negative phase difference is measured, it indicates that the downstream cycle master is running slower than the upstream cycle master and the alpha portal, indicated by α, instructs it to reduce the threshold value for the impending cycle start by one clock tick. Otherwise, when a positive phase difference is measured, the downstream cycle master is faster and the alpha portal instructs it to increase the

threshold. These instructions are shown by the 'cycle adjustment' feedback to the cycle master attached to the alpha portal.

3.2.1.5 Stream Connection Management

Stream connection management defined by the bridge standard requires the use of various net management messages to create, break and return status information of a stream path established between a talker and a listener. These messages include JOIN, LEAVE, LISTEN, RENEW, TEARDOWN and STREAM_STATUS. In addition to the terms *talker* and *listener* defined by the IEEE Std. 1394-1995, 1394.1 defines a *listening portal* and a *talking portal*. Listening portals are defined to be bridge portals that listen to a set of channel numbers in order for their packets to be retransmitted by their co-portals. Talking portals are bridge portals that transmit stream packets for a set of channel numbers. This is illustrated below.

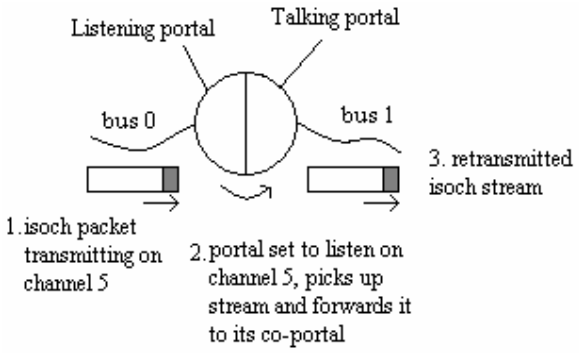


Figure 3-19: An illustration of a listening and talking bridge portal

A JOIN request message is used to establish a stream path from a talker to a listener. Conversely, a LEAVE request message is used to terminate the established stream path. A LISTEN message is used by a talking bridge portal to communicate the channel number and speed used for a stream on the talking portal's local bus to another portal on the local bus that intends to listen and forward the stream. The RENEW request message is used to extend the remaining stream connection lifetime for a particular listener. A TEARDOWN message is an inter-portal message that causes the teardown of all or part of a stream whose path has been severed by the disconnection of one or more nodes. The STREAM_STATUS message is a response message used to communicate to a controller the result of an earlier JOIN, LEAVE or RENEW request. The 1394.1 specification gives information on the structure of these

messages and also, the rules required to be implemented when a stream path is to be established or broken down.

In addition to these concepts and terminologies defined by the bridge standard, it also specifies additional requirements that are to be implemented by bridge portals and bridge-aware devices. These include additional definitions to the general configuration ROM format specified by the IEEE Std. 1394-1995 and also additional CSR registers. The two significant CSR additions are the MESSAGE_REQUEST and MESSAGE_RESPONSE registers. These registers allow for the receipt and handling of net management messages. A new self-ID packet zero structure is also defined, which includes a bridge-capability field that indicates whether a node is a bridge-aware device or not. The 1394.1 specification gives more information on these additional requirements. Work has been done by Melekam Tsegaye [Tsegaye, 2002] in investigating the possibility of developing bridge-aware IEEE 1394 drivers for the Windows and Linux operating systems.

The 1394.1 bridge specification has only been recently published (2005) and just made open to device manufacturers, and as such no bridge devices currently exist with this specification. Prior to the release of this specification and because of the demand by the audio industry to utilize 1394 bridging capabilities, NEC Corporation developed 1394 bridges based on a self-defined propriety specification [NEC Corp., 2002]. This specification evolved from the fundamentals of the 1394.1 standard. The operation of NEC bridges in facilitating isochronous stream forwarding in a multi bus environment is described in the next section.

3.2.2 The Propriety 1394 NEC Bridges

The *NEC MX/Bridge-A* bridges are the first serial bus bridges made available on the consumer market. Their architecture is based on that of 1394.1 in the sense that they also implement two portals connected to separate buses, with the bridge capable of performing selective relaying of packets between the buses connected to these portals. The following subsections describe the functionality of NEC bridges.

3.2.2.1 Virtual Node IDs

Virtual node IDs are equivalent to global node IDs defined by 1394.1; they are 16 bits wide and made up of a 10-bit bus ID and a 6-bit virtual ID. The virtual ID is not assigned during the self identification process and does not necessarily change during a bus reset.

3.2.2.2 Network Topology

The terminologies used in describing a NEC 1394 bridged network, with the exception of the prime portal, are identical to that defined by 1394.1. The definition of a network cycle master and alpha portals is maintained, with the added restriction that alpha portals are not only responsible for assigning a virtual ID to each node within the local bus, but are also assigned a virtual ID of zero. The same restriction applies to a network cycle master, and a network cycle master always resides on the 1394 bus that has been assigned a bus ID of zero. These ID assignments make it easy for a controller to locate and gain access to either an alpha portal or the network cycle master node.

From a network cycle master, one can obtain information about the bus IDs of the active 1394 buses on the network, as well as the number of bridges and the assigned virtual node IDs of the portals implemented by the bridges. Net management messages: ACTIVE_BUS_ID and NET_TOPOLOGY_MAP are defined for this purpose. The structure of the response data field in response to an ACTIVE_BUS_ID and a NET_TOPOLOGY_MAP message request are shown in Figure 3-20 and Figure 3-21 respectively.

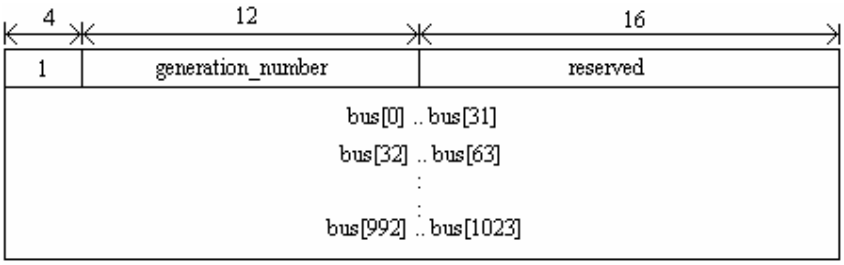


Figure 3-20: Format of the response to an ACTIVE_BUS_ID request

The field *generation_number*, indicates the number of times a change has occurred in the bus ID value of the active 1394 buses on the network. The fields *bus[0]…bus[1023]* form a bitmap that indicates the usage state of the bus IDs; if bus ID *i* (where *i* is an integer between 0 and 1022) is in use, the *bus[i]* bit will have the value 1, otherwise it will have the value 0.

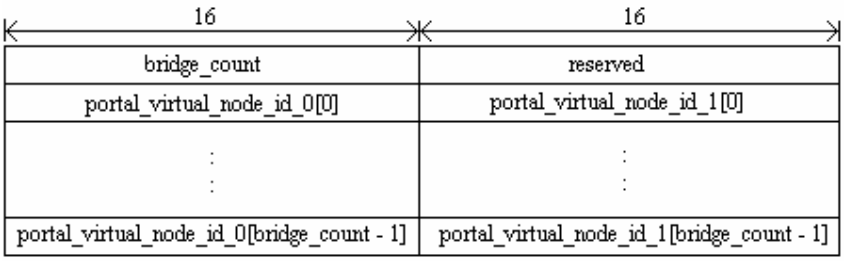


Figure 3-21: Format of the response to a NET_TOPOLOGY_MAP request

The field *bridge_count* specifies the number of bridges on the network. The virtual IDs of the portals of each bridge are given by *portal_virtual_node_id_0* and *portal_virtual_node_id_1*.

Similarly, information about the 1394 nodes on a bus can be retrieved by querying the alpha portal assigned to a particular bus. Two categories of information can be obtained, the first returns information identical to the *bus info block* of the configuration ROMs of the nodes on the bus. This information also specifies the virtual ID of a node and an indication of whether a node is a bridge portal or not. The net management message REMOTE_NODE_INFO is used to access this information. The format of the response data field in response to a REMOTE_NODE_INFO request is shown in the diagram below.

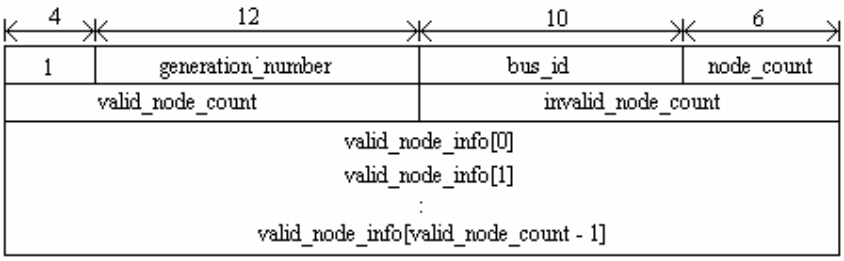


Figure 3-22: Format of the response to a REMOTE_NODE_INFO request

The *generation_number* field indicates the number of times the information contained within the packet has been updated. The *bus_id* field indicates the bus ID of the bus to which the alpha portal transmitting the message is connected. The *node_count* field indicates the number of nodes connected to the bus specified by *bus_id*. The fields, *valid_node_count* and *invalid_node_count*, specify the number of *valid* and *invalid* nodes attached to the bus. The NEC bridge specification [NEC Corp., 2002] gives definitions for valid and invalid nodes. The *valid _node_info* entry contains device detection information for the nodes connected to the bus that is equivalent to the *bus info block* structure defined by the IEEE Std. 1394-1995. The structure of this entry is given in Figure 3-23 below.

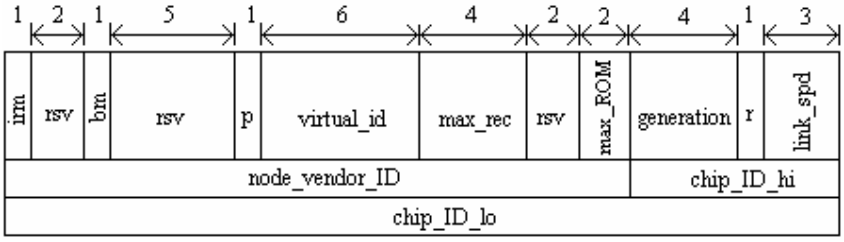


Figure 3-23: Format of the valid_node_info entry

All the fields with the exception of *irm, bm, p* and *virtual_id* are as defined by the IEEE Std. 1394-1995 [IEEE, 1995]. The *irm* bit indicates whether a node is the isochronous resource manager for the bus. Similarly, the bit *bm* indicates whether a node is the bus manager node. The portal bit, *p*, indicates whether a node is a portal. The *virtual_id* field indicates the value of the virtual ID that is assigned to the node.

The other category of information returned from the alpha portal is the topology map. This specifies the self ID packets of the nodes attached to the bus, from which a controller can reconstruct the bus topology according to the algorithm described in section 3.1.2.4. The BUS_TOPOLOGY_MAP net management message is used to retrieve this information from a bus's alpha portal. The format of the response data field in response to a BUS_TOPOLOGY_MAP request is shown in Figure 3-24.

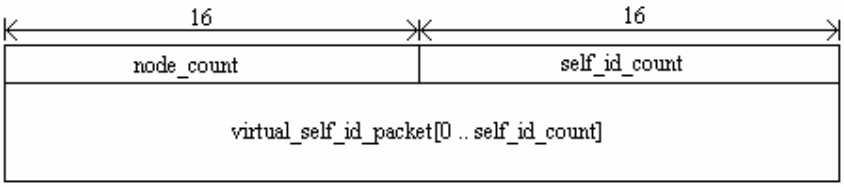


Figure 3-24: Format of the response to a BUS_TOPOLOGY_MAP request

The *node_count* field contains the number of nodes that are attached to the bus. Similarly, *self_id_count* specifies the number of self-ID packets contained within the response message. The virtual self-ID packets are specified by *virtual_self_id_packet* and are defined as shown in Figure 3-25 below. They a stored in ascending order of physical IDs.

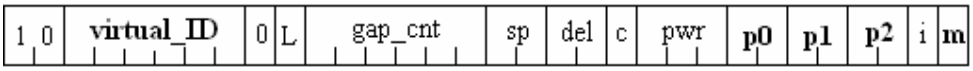


Figure 3-25: Structure of a virtual self-ID packet zero

All the fields but *virtual_ID* are as defined by the IEEE Std. 1394-1995 [IEEE, 1995]. The field *virtual_ID* specifies the virtual ID assigned to a node. Virtual self-ID packet one and two are also defined and are identical to those defined by the IEEE Std. 1394-1995 [IEEE, 1995]. However these packets have the *physical_ID* field replaced with a *virtual_ID* field.

In addition to these net management messages, a ROUTING_MAP message is also defined to access the bus ID routing path of portals on a network. This routing path specifies information of the bus IDs of the 1394 buses to which a particular bridge portal is capable of forwarding asynchronous data. The response data field in response to a ROUTING_MAP request is shown in Figure 3-26.

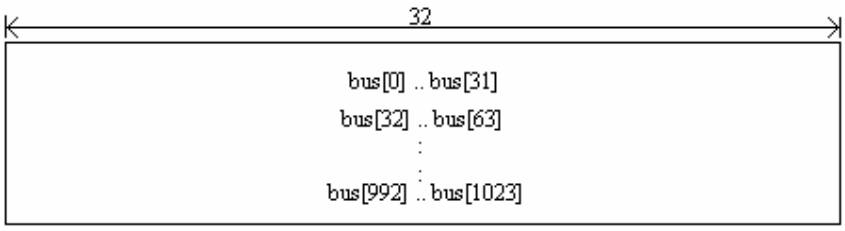


Figure 3-26: Format of the response to a ROUTING_MAP request

The fields *bus[0]…bus[1023]* form a bitmap that indicates the bus IDs of the 1394 buses that a bridge portal can asynchronously forward to.

These net management messages make it possible for a controller to retrieve complete information regarding the components and configuration of a network, in so doing enumerating the network accordingly.

3.2.2.3 Stream Packet Forwarding and Synchronization

NEC bridges facilitate stream packet forwarding by allowing a controller to specify configuration parameters to an appropriate *stream control* register. A stream control register is 32 bits wide and specifies across-bus forwarding parameters for an isochronous and asynchronous stream. The structure of this register is defined below [Yamaha Corp., 2003a].

Figure 3-27: Format of the stream control register

The ready bit, *RDY*, enables or disables forwarding of streams between buses. The output channel field, *OCH*, specifies the isochronous channel number to be used as the transmitting channel of the forwarded stream. The field *SPD* specifies the output speed of the forwarded stream. The connection counter, *CCTR*, indicates the number of listeners receiving the incoming stream to be forwarded. The *PAYLOAD* field specifies the payload size of the incoming stream to be forwarded.

Sixty-four registers, one for each isochronous channel, are defined. When a stream packet of channel = *i* (where *i* is an integer between 0 and 63) is received, the portal refers to the stream control register that corresponds to *i* (SCR[*i*]), and only if the ready bit is 1, will it convert the channel number and the speed of the incoming stream to the output channel and output speed of SCR[*i*]. This is illustrated in Figure 3-28 below.

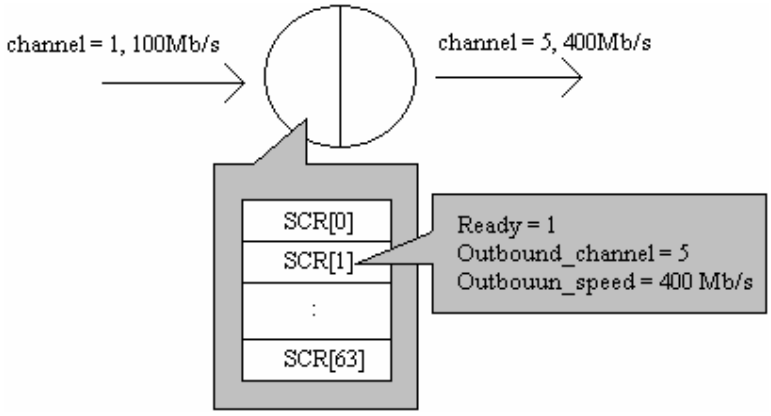


Figure 3-28: Stream packet transfer

From the diagram, the isochronous stream transmitting on channel 1 at 100 Mb/s is configured to be forwarded on channel 5 at 400 Mb/s.

In addition to configuring stream control registers for stream forwarding, synchronization must be established between adjacent buses of a bridge. Synchronization between buses is achieved by synchronizing the cycle masters of independent buses. The portals of an NEC bridge have a master/slave relationship. Each 125 μs, the master portal transmits a control pulse to the slave portal in the same bridge, which the slave uses in adjusting its own cycle timer. A slave portal functions as cycle master of its bus, time data synchronized to the network cycle master is propagated to the rest of the bridges on the network, thereby establishing synchronization for the entire network.

Until now, the architecture of IEEE 1394 has been discussed, describing the bus transport mechanisms and the related bridging technologies that enable multiple bus transmission. In the context of audio and music distribution, the asynchronous capabilities of IEEE 1394 permits a controller to send configuration information to an audio device or bridge portal to set it up for isochronous transmissions. Likewise, the isochronous capabilities can be used as a medium of transporting audio and music data. The Audio and Music data specification was originally developed by Yamaha to use the isochronous capabilities of IEEE 1394; it specifies how audio and MIDI should be formatted and transmitted. The next section takes a look at the Audio and Music specification.

3.3 Audio and Music Data Transmission

The Audio and Music (A/M) Data Transmission Protocol, adopted as an International Electrotechnical Commission (IEC) standards specification document, IEC 61883-6 [IEC, 2005], defines a protocol for the transmission of audio and music data over 1394. This document forms part of the real-time data transmission protocol, IEC 61883, which specifies a digital interface for consumer electronic/audio equipment using IEEE 1394.

The important features of the A/M protocol include:

- Real time data transmission
- Packet formation
- Transmission of timing information

3.3.1 Real Time Data Transmission

Isochronous packets are used for the transmission of multiple *sequences* of audio and MIDI data. Individual audio samples and MIDI data from the various sequences, which occur at a particular point in time, are clustered into *data blocks*. The number of data blocks within an isochronous packet depends on the sampling rate and the transmission mode of the transmitting node. Two transmission modes are defined: *blocking* and *non-blocking* modes. In blocking mode, transmission only occurs when a fixed number of events (audio samples or MIDI data) are accumulated. Conversely in non-blocking mode, transmissions occur if more than one event is accumulated within the nominal isochronous cycle. For a sampling rate of 48 kHz operating in non-blocking mode, 6 data blocks would be encapsulated within an isochronous packet.

Figure 3-29 shows a diagram of an isochronous packet with six data blocks [Fujimori and Foss, 2003].

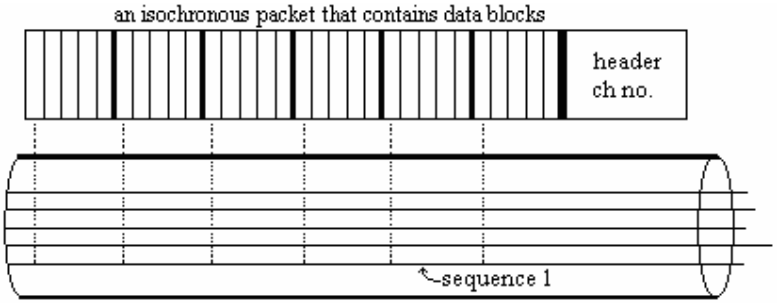


Figure 3-29: An isochronous stream with sequences

At the top of the figure is an isochronous packet whose header carries a channel number. Isochronous packets, with the same channel number make up an isochronous stream. Each data block within the isochronous packet, is made up of a number of elements, these elements being audio samples or MIDI messages. The position of an audio sample or MIDI message within the data block indicates the sequence that it belongs to. Figure 3-29 shows five elements in each data block. The first element of each data block belongs to the first sequence, second element to the second sequence, and so on.

Transmission of MIDI requires multiplexing up to eight MIDI data streams (corresponding to the data carried by 8 MIDI cables) onto a single sequence within successive data blocks. This is illustrated in Figure 3-30 below. In this diagram, the fourth data block position has been assigned to a MIDI sequence. Messages from the three MIDI data streams are being carried in a single sequence. Every eight data block will have, at position 4, the next data for the first MIDI data stream. This implies that 'no data' quadlets fill the fourth position of data blocks 4-8.

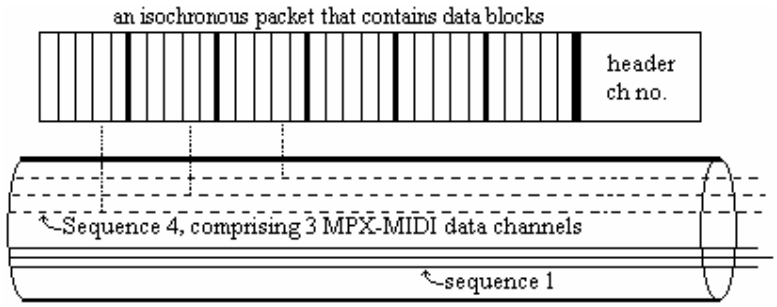


Figure 3-30: Multiplexing MIDI data streams

3.3.2 Packet Formation

An isochronous packet, required for transmitting audio and MIDI data, is required to have a two-quadlet Common Isochronous Packet (CIP) header at the beginning of its data field. This packet introduces information about the type of real-time data in the data field following it. Figure 3-31 shows an isochronous data packet that conforms to the IEC 61883 standard. The fields within the isochronous header have been previously discussed.

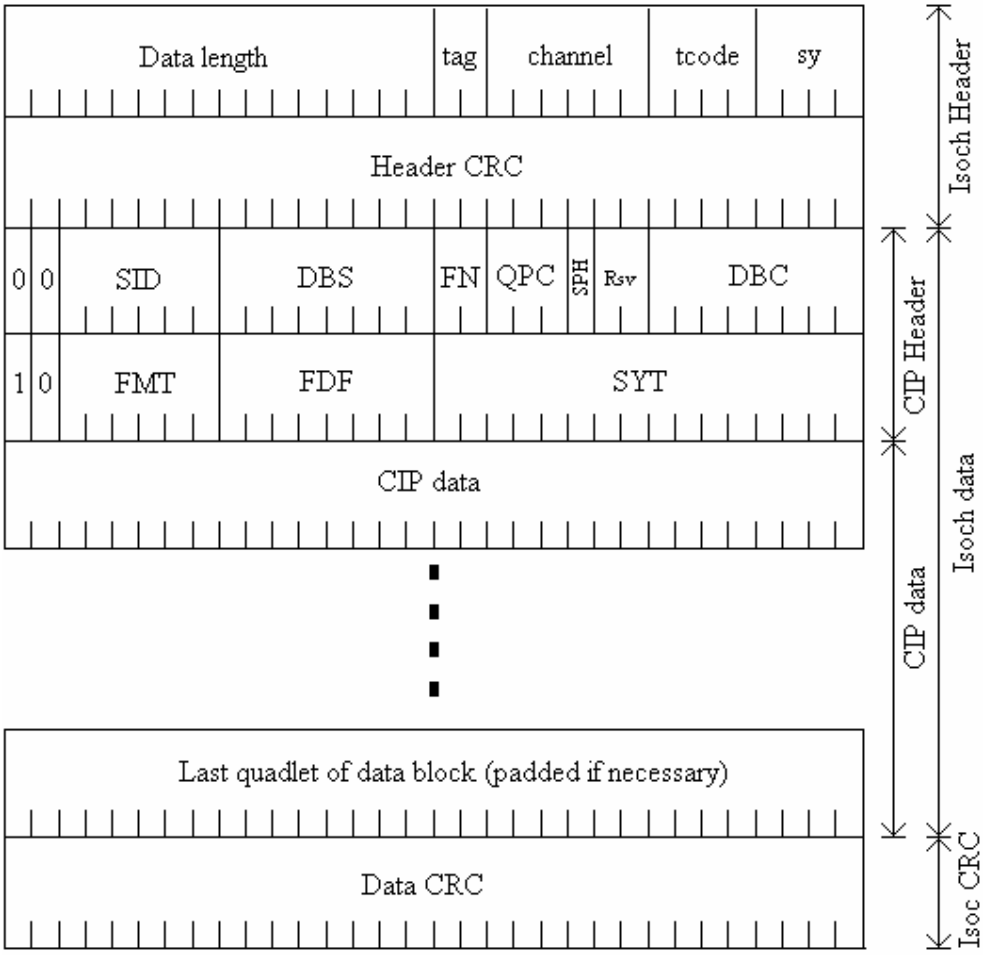


Figure 3-31: The Common Isochronous Protocol (CIP) packet format

For isochronous transactions that are CIP conformant, the *tag* field in the isochronous header is assigned the value of 01b. The remaining fields are defined as follows:

- The *SID* field refers to the source identifier, and contains the 6-bit physical ID of the node transmitting the packet.
- The *DBS* field indicates the data block size.

- The *FN* field specifies the number of fractions into which the original application packet was divided, after optional stamp and padding were added. (Not relevant for audio).
- The *QPC* (quadlet padding count) contains a value that indicates the number of padding bytes added to each application frame to make it divide evenly into 2^{fn} data blocks. (Not relevant for audio).
- The *SPH* field indicates if a transport delay time stamp has been added by the transmitter. The delay time stamp, if included, is specified within the source packet header and has the value of the lower 25-bits of the IEEE 1394 CYCLE_TIME register. (Not used by the A/M protocol).
- The *DBC* field acts as a sequence counter and continuity checker that is used by receivers for audio synchronization.
- The *FMT* field identifies the data type using the CIP.
- The *FDF* (format dependent field) is used to identify sub-formats or convey other information.
- The *SYT* field indicates the time that a particular data block within the packet should be presented at the receiver. This is also used for audio synchronization.

The A/M protocol defines unique values for the fields: *FMT, FN, QPC* and *SPH*. These values are summarized in the table below.

Field	**Value** **[hex]**	**Comment**
FMT	10	Indicates that the format is for audio and music data
FN	00	Indicates that the source packets are not divided into fractions.
QPC	00	Indicates that no quadlet padding is necessary since all data is 32-bit aligned.
SPH	00	Indicates that no transport delay time stamps are present within the packet.

Table 3-5: CIP header values defined by the A/M protocol

The CIP data field contains audiovisual data that is clustered into one or more data blocks or events. The A/M protocol defines 32-bit aligned events of different formats. The defined events are as follows:

- AM824 data

- 32-bit floating point data
- 24-bit * 4 audio pack
- Provision for 32-bit or 64-bit data

The format of these events is indicated by the value of the FDF field held within the CIP header. This field also specifies the nominal sampling frequency code of the transmitted data. The structure of the FDF field is shown below.

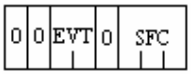


Figure 3-32: Generic FDF definition

Table 3-6 gives the code definitions for EVT and SFC.

EVT code definitions		**SFC code definitions**	
Value [decimal]	**Description**	**Value** [decimal]	**Description**
0	AM824 Data	0	32 kHz
1	24-bit * 4 audio pack	1	44.1
2	32-bit floating point data	2	48
3	Reserved for 32-bit or 64-bit data	3	88.2
		4	96
		5-7	Reserved

Table 3-6: EVT and SFC code definitions

The A/M specification [IEC, 2005] gives the details of these events. The most relevant event type, in the context of a studio or sound installation environment is AM824. The AM824 event type specifies a 32-bit data field consisting of an 8-bit label and a 24-bit data. This encodes IEC 60958 digital audio, raw audio or MIDI data into a 32-bit event.

An IEC 60958 conformant digital audio event has a label value ranging from 00h to 3Fh. The format of this event is illustrated in Figure 3-33.

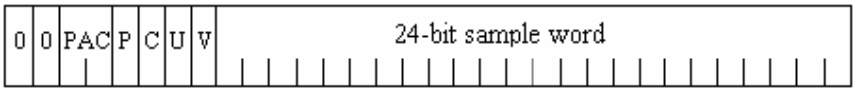


Figure 3-33: Format of an IEC 60958 conformant event

P, C, U, V carries auxiliary information as defined in the IEC 60958 standard [IEC, 2003a]. The preamble code, *PAC*, identifies the sub frame within the IEC 60958 frame.

The raw audio event has a label value ranging from 40h to 43h; this is illustrated in Figure 3-34 below.

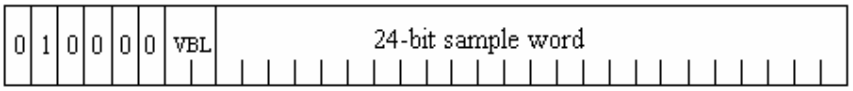


Figure 3-34: Format of a raw audio event

The 24-bit data field contains an audio sample that is expressed in 24-bit 2's complement format. If the data is less than 24-bits, the correct number of zero bits is padded below the least significant bit to make it a 24-bit data structure. The valid bit length code, *VBL*, specifies the number of valid bits that make up an audio sample; 00b indicates 24 bits, 01b indicates 20 bits and 10b indicates 16 bits.

A MIDI conformant event has a label value ranging from 80h to 83h. The format of this event is illustrated in Figure 3-35.

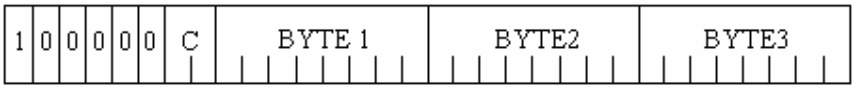


Figure 3-35: Format of a MIDI conformant event

The counter, C, indicates the number of valid MIDI bytes contained within the 24-bit data; 00b indicates that no valid bytes exists within the packet, 01b indicates that only *BYTE1* is valid, 10b indicates that both *BYTE1* and *BYTE2* are valid, 11b indicates that all three bytes are valid.

Audio and MIDI data can be formatted and transmitted over 1394 as described above. In order for a device to receive and playback audio events, the device's word clock has to be synchronized to that of the transmitter, or the word clocks of both the transmitter and the receiver have to be synchronized to a particular reference. The synchronization technique adopted by the IEC 61883-6 specification is discussed in the next subsection.

3.3.3 Transmission of Timing Information

In CIP data transmission, there are two distinct mechanisms [Laubscher, 1999] provided for synchronization via time stamps:

1. Transport delay time stamp
2. Higher level application synchronization time stamp

Every IEEE 1394 node implements a CYCLE_TIME register (refer back to section 3.1.1.3), which is incremented via a 24.576 MHz clock. The CYCLE_TIME register of all the nodes are synchronized by cycle start packets sent by the cycle master node every 125 μs. These cycle start packets contain the cycle time value of the CYCLE_TIME register belonging to the cycle master node. When isochronous packets with audio samples are being transmitted, these packets are time stamped with a 25-bit sample of the transmitting node's CYCLE_TIME register. An additional offset is added to the time stamp, and is intended to be the presentation time at which the receiver presents the associated event to its host application. This method of time stamping is known as the transport delay time stamp.

The A/M protocol does not use transport delay time stamps, but rather utilizes the SYT field located within the CIP header, for synchronization. In this high level application synchronization mechanism, a transmitting node places the twelve least significant bits of its CYCLE_TIME register, incremented with an offset, into the SYT field of the CIP header. The value in the SYT field is intended to be the presentation time of the event by the receiver. A CIP packet may contain multiple events and only one time stamp. In order to associate the SYT time stamp with a particular event, a unit known as the SYT_INTERVAL is used. The SYT_INTERVAL is defined to be

the number of events between two successive SYT time stamps. The transmitter prepares the time stamp for the event that meets the condition:

$$\mathrm{mod}(DBC, SYT\_INTVAL) = 0 \qquad (2)$$

At the receiver, the index of the cluster event to which the SYT time stamp applies can be calculated from:

$$index = \mathrm{mod}((SYT\_INTVAL - \mathrm{mod}(DBC, SYT\_INTVAL)), SYT\_INTVAL) \qquad (3)$$

The receiver is responsible for estimating the timing of events between valid time stamps.

Phase locked loop (PLL) techniques, are normally employed to achieve this. When a receiver receives an isochronous packet, it reads the associated time stamp. When this time stamp value equals its own CYCLE_TIME register value, it indicates a match to its PLL. The PLL then continuously receives a series of matched pulses from which it can determine the sample rate of the transmitter. This is illustrated using Figure 3-36.

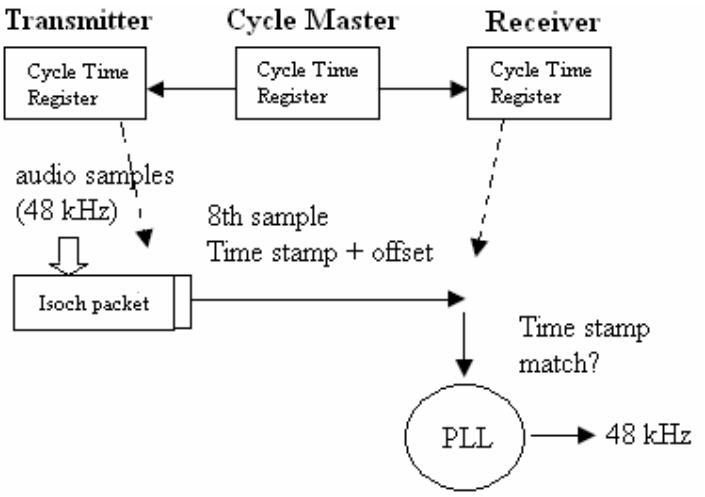


Figure 3-36: Sample clock synchronization

From the figure, the transmitter generates sequences of audio samples at 48 kHz, and packs them into data blocks within isochronous packets. A clock at the transmitter determines the sample rate and every eighth tick of the sample rate clock, the CYCLE_TIME register is sampled and included (with an offset) into the CIP header

of the next isochronous packet to be transmitted. At the receiver, the time stamp is read from the isochronous packet, and a match indicated to the PLL when the time stamp value equals its own CYCLE_TIME register value.

Jitter is associated with the sample clock produced by the PLL. However, it has been shown that the jitter is small enough (less than 600 picoseconds) to allow for the implementation of professional audio equipment using IEEE 1394 [Kuribashi, Ohtani and Fujimori, 1998]. With these mechanisms in place, the next step is to be able to establish audio/MIDI point-to-point or broadcast connections over 1394. The next section takes a look at the available connection management techniques that make this possible.

3.4 Connection Management Techniques

The IEC 61883-1 specification [IEC, 2003b] provides a mechanism that allows for setting up point-to-point or broadcast connections. It defines a number of mechanisms, namely, isochronous data flow management, connection management procedures and the Function Control Protocol (FCP). The FCP forms the basis on which other high-level connection management specifications are based. These specifications include the AV/C[6] Connection and Compatibility Management specification [1394TA, 2002a] and the AV/C Music Subunit specification [1394TA, 2001], which have been developed above the general AV/C specification, namely the AV/C Digital Interface Command Set [1394TA, 2004]. These provide a more flexible solution to connection management, and are described in the paragraphs that follow. Firstly though, a detailed description of the IEC 61883-1 specification is given.

3.4.1 IEC 61883-1

This specification forms the basis of most high-level connection management techniques and consists of three main sections:

- Isochronous data flow
- Connection management procedures
- Function control protocol

[6] AV/C refers to audio and video control

3.4.1.1 Isochronous Data Flow

The IEC 61883-1 specification makes use of the concept of serial bus plugs through which isochronous connections can be established. In this model, isochronous data is transmitted through an output plug of a transmitting device and is received, at a receiver, through an input plug. This concept is shown in Figure 3-37. These plugs are required to be implemented in terms of plug control registers as part of a device's control and status registers. Four types of plug control registers are defined: output plug control register (oPCR), output master plug register (oMPR), input plug control register (iPCR), and input master plug register (iMPR).

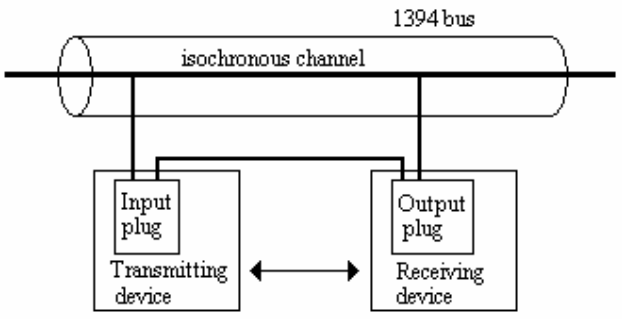


Figure 3-37: Plug model for isochronous flow management

An oPCR implements an output plug of a transmitting device; likewise an iPCR implements an input plug of a receiving device. For all output plugs, there exists an oMPR that controls the attributes common to all output plugs; likewise for all input plugs there exists an iMPR that controls the attributes common to all input plugs. The oPCR specifies a *channel number*, *overhead ID* and *payload* that describe the properties of an output isochronous stream. The iPCR specifies a *channel number* from which an isochronous stream should be received by an input plug. The IEC 61883-1 specification gives definitions of the fields contained within these plug registers.

A controller can specify to the oPCR of a transmitting device, the isochronous channel on which to transmit, and also to the iPCR of a receiving device, the isochronous channel on which to receive. This is achieved through asynchronous writes to the appropriate plug control register on the node, which results in

manufacturer-specific operations that actually set the transmission or reception channel number. In establishing these connections, certain procedures are required to be adhered to; these are discussed in the next subsection.

3.4.1.2 Connection Management Procedures

Connection management procedures describe the steps that an application is required to take in order to manage connections between input and output plugs of a device. For both types of connections: point-to-point and broadcast, procedures are described for the action to be taken when commands are received to:

- Establish a connection
- Overlay a connection
- Break a connection

These procedures involve modifying the appropriate input and output plug control registers, which also includes incrementing or decrementing connection counters. Procedures for restoring connections after a bus reset are also defined. The IEC 61883-1 specification gives more details on this.

Commands to establish, overlay, and break connections are sent to a node using *unit commands* defined by the AV/C Digital Interface Command Set [1394TA, 2004]. AV/C commands and responses are encapsulated within asynchronous packets, formatted according to the Function Control Protocol, described in the next section.

3.4.1.3 Function Control Protocol

The Function Control Protocol (FCP) provides a means by which control and response messages are transported among devices connected through an IEEE 1394 bus. These control and response messages are encapsulated within an FCP frame, which forms the payload of an asynchronous *write data block* packet. A *write data quadlet* packet is used if the length of the FCP frame is exactly four bytes.

Figure 3-38 shows an FCP frame within an asynchronous write data block packet. The format of the *Destination_ID*, *tl*, *rt*, *pri*, *Source_ID*, *data_length* and *CRC* fields are as defined for asynchronous block write packets in section 3.1.1.2. The *cts* field specifies

the command transaction set being transported; this field has a value 0 for the AV/C command set. A command frame, originating from a controller, is written to the FCP_COMMAND register of a target device. Similarly the corresponding response frame, originating from the target device, is written to the FCP_RESPONSE register of the controller. This process is illustrated in Figure 3-39. The FCP_COMMAND and FCP_RESPONSE registers exist within a node's initial unit address space at an offset of 0B00h and 0D00h respectively.

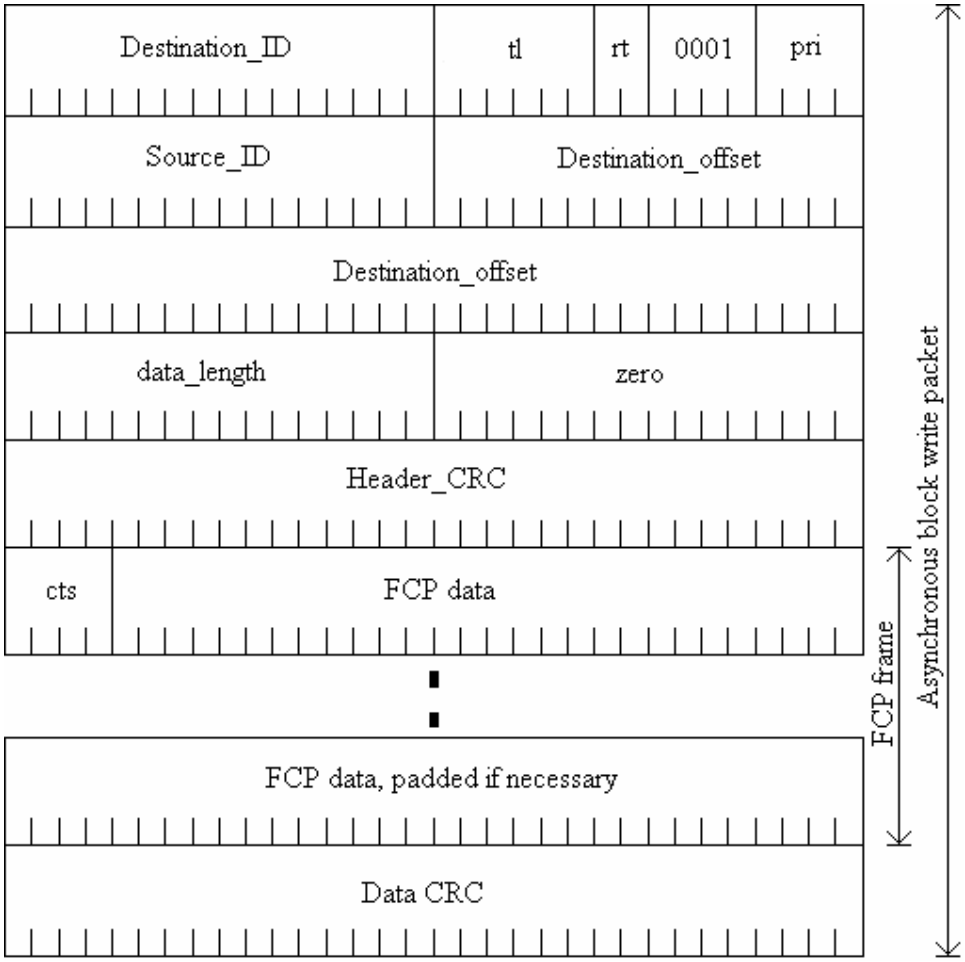


Figure 3-38: An FCP frame within an asynchronous write block packet

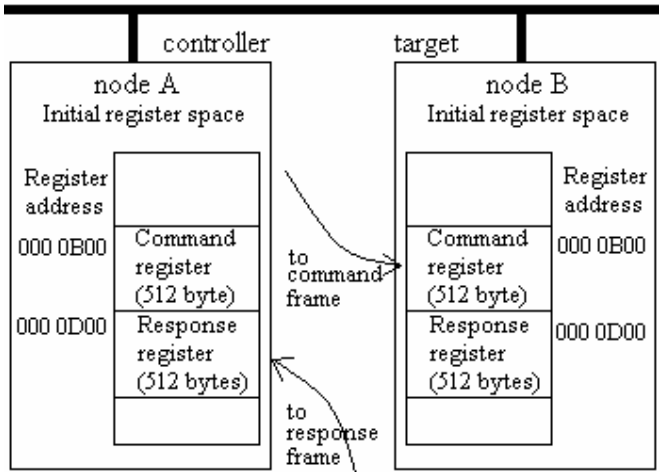


Figure 3-39: Command and response FCP registers

3.4.2 AV/C Digital Interface Command Set

The AV/C Digital Interface Command Set [1394TA, 2004] specifies a generic command set used to control audio/video devices. As already mentioned these AV/C commands or responses are encapsulated within FCP frames and are transmitted between the FCP_COMMAND and FCP_RESPONSE registers. In AV/C terms, the serial bus plugs mentioned earlier are associated with units within 1394 nodes and are referred to as unit plugs. The AV/C protocol also introduces the concept of a *subunit*, where a subunit encapsulates some structure and function that is common to a number of devices. For example, there exists an Audio Subunit, Monitor Subunit, Disc Subunit, etc.

These units and subunits have both source and destination plugs through which connections can be specified. This concept is illustrated in Figure 3-40 below. Both the transmitter and receiver nodes have an audio subunit within their respective audio unit. From the transmitter, the plug labelled *output4* is connected to the first output *unit plug*, set to transmit on channel *2*. The second input unit plug of the receiver, configured to listen on channel *2*, 'picks up' the corresponding data and directs it to plug *input3* of its audio subunit. This concept is made possible by the AV/C Connection and Compatibility Management Specification [1394TA, 2002a], which will be discussed shortly.

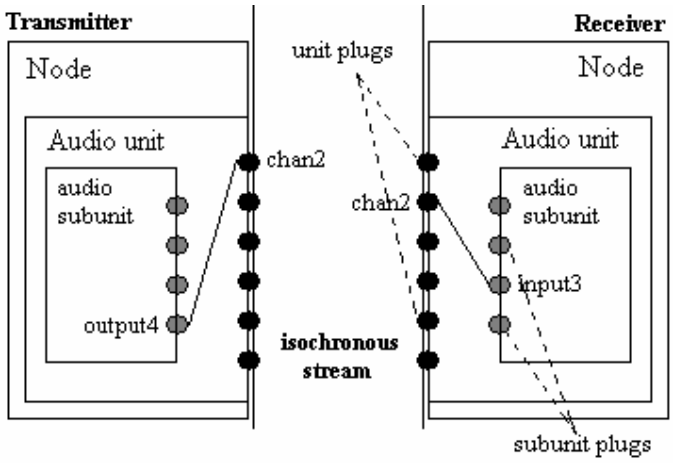


Figure 3-40: Illustration of subunit plugs

The format of an AV/C frame is shown in Figure 3-41 below.

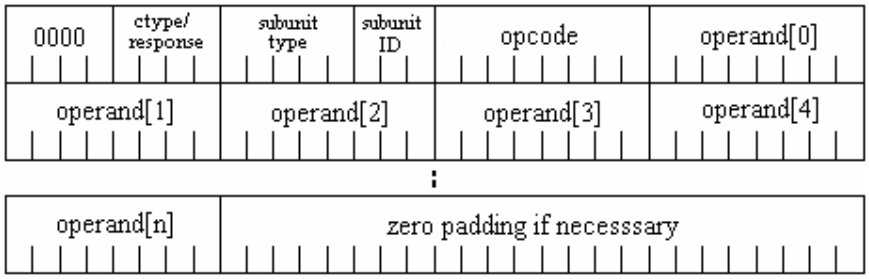


Figure 3-41: Structure of an AV/C frame

The first nibble of the frame, *0000b*, indicates an AV/C command set, and corresponds to the *cts* field shown in Figure 3-38. The field represented by *ctype/response* indicates the type of the FCP command specified by a controller, or the response status of an FCP command issued to a target node. The possible values of *ctype* and *response* are shown in the table below.

***ctype* code definition**		***response* code definition**	
Value [hex]	**Description**	**Value [hex]**	**Description**
0	Control	8	Not Implemented
1	Status	9	Accepted
2	Specific Enquiry	A	Rejected
3	Notify	B	In Transition
4	General Inquiry	C	Changed

5-7	Reserved	D	Implemented/Stable
		E	Reserved
		F	Interim

Table 3-7: The ctype and response values for the AV/C frame

The *subunit type* and *subunit ID* together specify an AV/C 'address' of the recipient of the command or the source of the response. This address is used to locate a particular subunit within a unit of a node (see Figure 3-40). The current subunit types defined by the AV/C specification include: audio, music, video monitor, disc recorder/player, tape recorder player, tuner and video camera. Other subunit types are currently being defined. The *opcode* field specifies the operation to be performed or the status to be returned by the target node. These are divided into ranges valid for commands addressed to units, subunits or both. Refer to the AV/C Digital Interface Command See the AV/C Digital Interface Command Set [1394TA, 2004] for further information on the implementation of the operands of these commands.

Information about AV/C units, subunits, and their associated components such as plugs, are stored within data structures contained within the unit or subunit. These data structures exist as part of various mechanisms. The AV/C Descriptor Mechanism Specification Version 1.0 [1394TA, 2002b] gives details of the various data structures. This specification defines two types of structures for sharing data between devices: the *descriptor* and the *information block.*

A descriptor is a structured data interface that contains information about a device's features as well as other descriptive information. Information blocks are extensible data structures that exist inside descriptors. A conceptual representation of these data structures is given in Figure 3-42; refer to the AV/C Descriptor Mechanism Specification [1394TA, 2002b] for more information on these data structures.

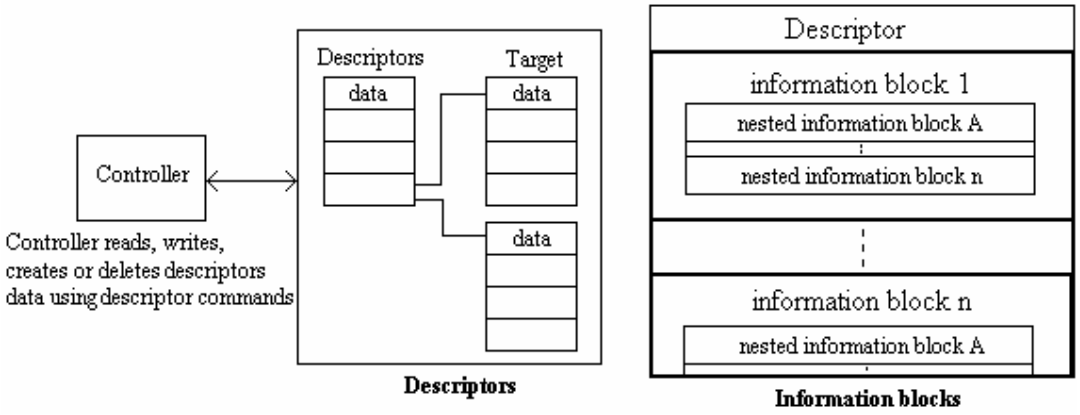


Figure 3-42: Illustration of descriptors and information blocks

3.4.2.1 AV/C Control and Compatibility Management

The AV/C Connection and Compatibility Management (CCM) specification [1394TA, 2002a] was developed to provide a standard set of commands to be used in establishing plug connections between AV/C devices on a 1394 network. The plug connections handled by this specification are internal plug connections i.e. between subunit and unit plugs, as well as external plug connections between unit plugs of AV/C devices. This is displayed in Figure 3-40. Establishing external plug connections require writes to plug control registers described earlier (see section 3.4.1.1).

While CCM and plug control register writes are appropriate for the creation of isochronous streams between transmitters and receivers, they are directed more towards consumer audio/video devices. With professional devices, it is desirable to be able to extract *sequences* of data within isochronous streams, and for this reason, the AV/C Music Subunit Specification [1394TA, 2001] was developed. This is briefly described in the next subsection.

3.4.2.2 AV/C Music Subunit Specification

The AV/C Music Subunit specification was created to fulfil professional audio's particular need for the encapsulation of large numbers of audio, MIDI, SMPTE and sample count sequences into isochronous streams, and the selective extraction of these sequences [Fujimori and Foss, 2003]. As an example, a synthesizer may transmit two

audio sequences in sequence position 1 and 2 of an isochronous stream, and a user may want to direct these sequences to specific inputs on a mixer. Refer back to section 3.3.1 for an illustration of how sequences are carried within isochronous streams.

Each Music Subunit within a unit on a device contains a number of music input plugs and music output plugs, each plug being of a particular type, for example audio, MIDI, SMPTE or sample count. A subunit input plug may be connected to one or more music input plugs, and one or more music output plugs may be connected to a music subunit output plug. This concept is shown in Figure 3-43 below.

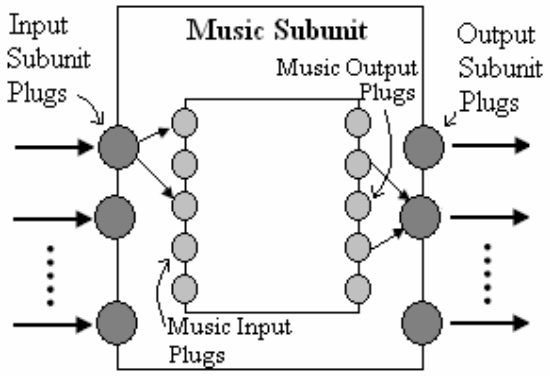


Figure 3-43: Music subunit plugs

Assume two devices, a synthesizer and a mixer, implement Music Subunit plugs as modelled in Figure 3-43. Let the connected music output plugs represent two of the outputs of a synthesizer, and the two connected music input plugs represent two of the inputs of a mixer. Music subunit commands will allow a controller to determine the sequence positions within the isochronous stream of the output from the synthesizer's music output plugs. They will also allow for the selection of these sequences from the relevant subunit destination plug as inputs to the mixer's music input plugs.

The connection management abstractions discussed thus far need to be implemented in terms of hardware and software structures. Various LSI chips have been developed by various vendors that facilitate the extraction of sequences from isochronous streams. The next section takes a look at the supporting hardware available for connection management.

3.5 Connection Management Hardware

The mLAN[7] project, an initiative of Yamaha Corporation, created chips that allow for sequence extraction within an isochronous stream. The first chip, mLAN-PH1, is an application chip that has to be connected to a link layer controller chip of a 1394 node. In this case, all isochronous packets, regardless of isochronous channel number, are picked up by the link layer, from where specific isochronous streams are selected. The mLAN-PH1 chip allows transmission of 8 audio sequences. All sequences are combined into an isochronous stream and the channel number associated with the packets is specified via a register. It is also capable of extracting 8 audio sequences from an isochronous stream and placing them into FIFO buffers. Each FIFO has two registers associated with it, one to determine the channel of an isochronous stream and the other to determine the sequence position within the isochronous stream. Figure 3-44 illustrates the operation of the mLAN-PH1 chip (adapted from AES Convention Paper 5699, [Fujimori and Foss, 2003]).

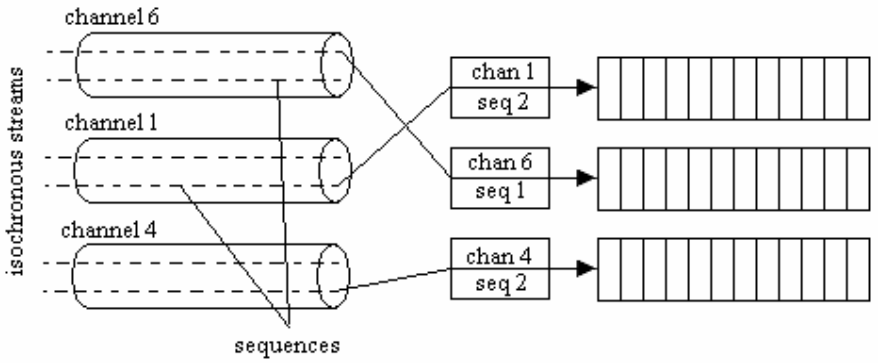


Figure 3-44: Operation of the mLAN-PH1 chip

The mLAN-PH1 also allows for MIDI transmission and extraction. This chip does not multiplex the MIDI data to allow additional MIDI data streams. However, MIDI data like audio data can be selected via appropriate registers from any sequence within any isochronous channel. The received data is place into a FIFO and is read by a CPU on a separate chip. Upon transmission, the PH1 adds time stamps based on the word clock of the audio input. On reception, an in-built PLL generates a word clock based on the time stamps of a particular isochronous stream. This isochronous stream can be

[7] mLAN stands for music Local Area Network

specified via a write to a particular register. Up to four mLAN-PH1 chips can be connected to a single link layer, thus providing a 32-sequence audio I/O capability.

Following the mLAN-PH1, a second more comprehensive chip, the mLAN-NC1, was created. This chip incorporates the capabilities of the mLAN-PH1, together with a link layer controller, a microprocessor unit and an associated flash memory. It also allows for a transmission rate of 400 Mb/s, compared to the 200 Mb/s transmission rate of mLAN-PH1. Furthermore, the performance of the embedded mLAN-PH1 chip is enhanced.

In response to the demand for greater volumes of audio data in the high-end professional audio industry, the mLAN-PH2 chip was created. A single chip can transmit a maximum of 32 sequences which can be combined into a single isochronous stream. Also, 32 sequences can be received, where each sequence can be extracted from any position in any isochronous stream. Up to four PH2 chips can be connected together to provide a 128 sequence transmission and reception capability. The PH2, like the PH1, requires an external link layer controller, microprocessor unit, and memory. Note that for high sampling rates (88.2 kHz and 96 kHz) the mLAN-PH2 chip has a maximum of 16 receive and transmit sequences.

These chips provide a number of solutions to the problem of connection management by encapsulating audio and music data into sequences within isochronous streams, and extraction of these sequences for presentation. Above these fundamental capabilities, structures must be provided that present these capabilities in terms of plug abstractions. The subsequent chapters describe the design and implementation of a connection management application (the Enabler) that enables, at a user level, plug connections to be established between various audio devices.

3.6 Summary

This chapter reviews IEEE 1394 and the related technologies that enable it to be used as a backbone for the transport of audio and MIDI data, in addition to control information. The main aspects of IEEE 1394 discussed are its asynchronous and isochronous transport capabilities, as well as the bus configuration process that occurs

between 1394 nodes following a bus reset. The 1394 bridge model is also reviewed, describing both the IEEE 1394 bridge standard, 1394.1-2005, and the proprietary NEC 1394 bridge specification. It is pointed out that no bridge hardware exists with the 1394.1-2005 specification, and that the NEC MX/Bridge-A 1394 bridges (based on a propriety specification) are the first consumer IEEE 1394 bridges. The various features of an NEC bridge that allow for across-bus forwarding of isochronous streams are highlighted.

Various specifications that enable the transport and control of audio and MIDI data over IEEE 1394 are also discussed. These include the IEC 61883-6 protocol that describes the format of audio and MIDI data, the IEC 61883-1 specification that describes the lower-level connection management procedures using plug control registers, and the higher-level connection management techniques described by the AV/C Connection and Compatibility Management specification, and the AV/C Music Subunit specification. A number of hardware chips are also described: the mLAN-NC1 chip, the mLAN-PH1 chip and the mLAN-PH2 chip. These chips provide the platform from which these high level AV/C connection management techniques, and hence mLAN can be realized. The next chapter describes the first implementation of mLAN – the *mLAN Version 1* architecture.

Chapter 4

4. MLAN Version 1. A First Attempt at Connection Management

As discussed in the previous chapter, various high-level AV/C standards exist that enable plugs of units and subunits to be interconnected. These standards also allow connections to be established between music subunit plugs, thus making it possible to encapsulate audio and music data into isochronous sequences and subsequently extract these sequences. An AV/C standard, which makes use of descriptors and information blocks also exists, and specifies how information is represented within the device. These mechanisms provide a level of plug abstraction within a device and allow a controller to gain access and configure the device appropriately for isochronous transmission and reception. A typical controller is a desktop PC.

This chapter takes a look at the high-level connection management application developed by Yamaha Corporation, which interacts with the first generation of mLAN devices. This connection management application was developed to run on Microsoft Windows and OSX. A similar implementation, resulting from the analysis of this research, was developed for the Linux environment. This move was required, and paved the way for further advances in the development of connection management applications. These advances form the core of this research and are discussed in subsequent chapters.

4.1 MLAN Version 1 Device Architecture

"MLAN Version 1" is the name given to the first generation of audio and music devices that resulted from Yamaha's mLAN initiative. These devices implement a plug abstraction layer that is based on the AV/C protocol [1394TA, 2004]. The plugs implemented by this architecture, referred to as *mLAN plugs*, have an implementation that is based on neither the AV/C Connection and Compatibility Management specification nor the AV/C Music Subunit specification, but rather makes use of a set of vendor dependent commands that form part of the AV/C protocol suite. The Music Subunit specification was adopted by the 1394TA sometime after the manufacture of mLAN interfaces and expansion boards. However, information regarding the properties of these devices is encapsulated within descriptors and information blocks as defined by the AV/C Descriptor Mechanism Specification [1394TA, 2002b].

The implementation architecture of the mLAN Version 1 devices can be conceptualized as consisting of two components: an *upper layer* and a *lower layer*. This is illustrated in Figure 4-1 below.

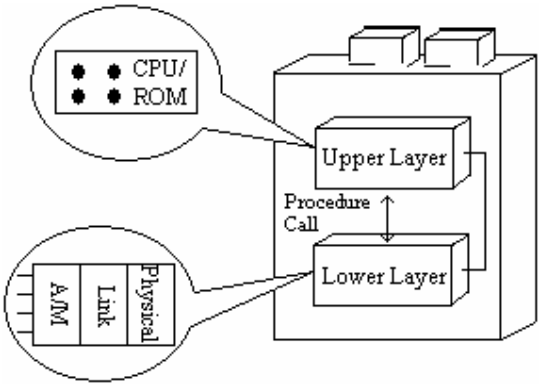


Figure 4-1: Architecture of mLAN Version 1 devices

The upper layer implements the high-level component and is typically made up of a CPU/ROM that implements a plug abstraction layer which contains information about the various plugs implemented by the device. The plug abstraction layer is implemented in terms of the AV/C protocol, descriptors and information blocks, and is exposed to a controlling device. The lower layer is made up of the standard IEEE 1394 physical and link layer, and the A/M data layer. The A/M data layer is

implemented in terms of the mLAN-NC1, mLAN-PH1 or mLAN-PH2 chips described previously, which have the ability to receive various formats of audio and MIDI data from outside sources, encapsulate the data into isochronous sequences, time-stamp them and then transmit the sequences via the link layer and physical layer hardware. In a similar manner, they have the ability to receive more than one isochronous stream and extract particular sequences from the received streams. Particular registers of the mLAN chip have to be modified to enable the transmission or reception of isochronous sequences. These register values are communicated to the lower layer via the high-level implementation within the upper layer.

The upper layer receives a number of vendor dependent commands that specify particular actions. These commands are picked up and handled by the CPU (of the upper layer) via the link layer, which then modifies the appropriate information blocks and also the necessary register content within the A/M layer.

4.1.1 Descriptor and Information Block Layout

The descriptor and information block data structures implemented within the upper layer of an mLAN Version 1 device are specified by the "mLAN 1.0 Connection and Control Specification" [Yamaha Corp., 2000a]. This has a structure similar to that described in Figure 3-42. There is a top level descriptor, referred to as the *connection management status* descriptor, which specifies a number of information blocks that describe the overall properties of the mLAN device. The information blocks defined by the *connection management status* descriptor include:

- The *general connection management status* information block
- The *destination mLAN plug status area* information block
- The *source mLAN plug status area* information block

These information blocks implement other nested information blocks that specify further information. Figure 4-2 illustrates the descriptor/information block structure implemented within mLAN Version 1 devices.

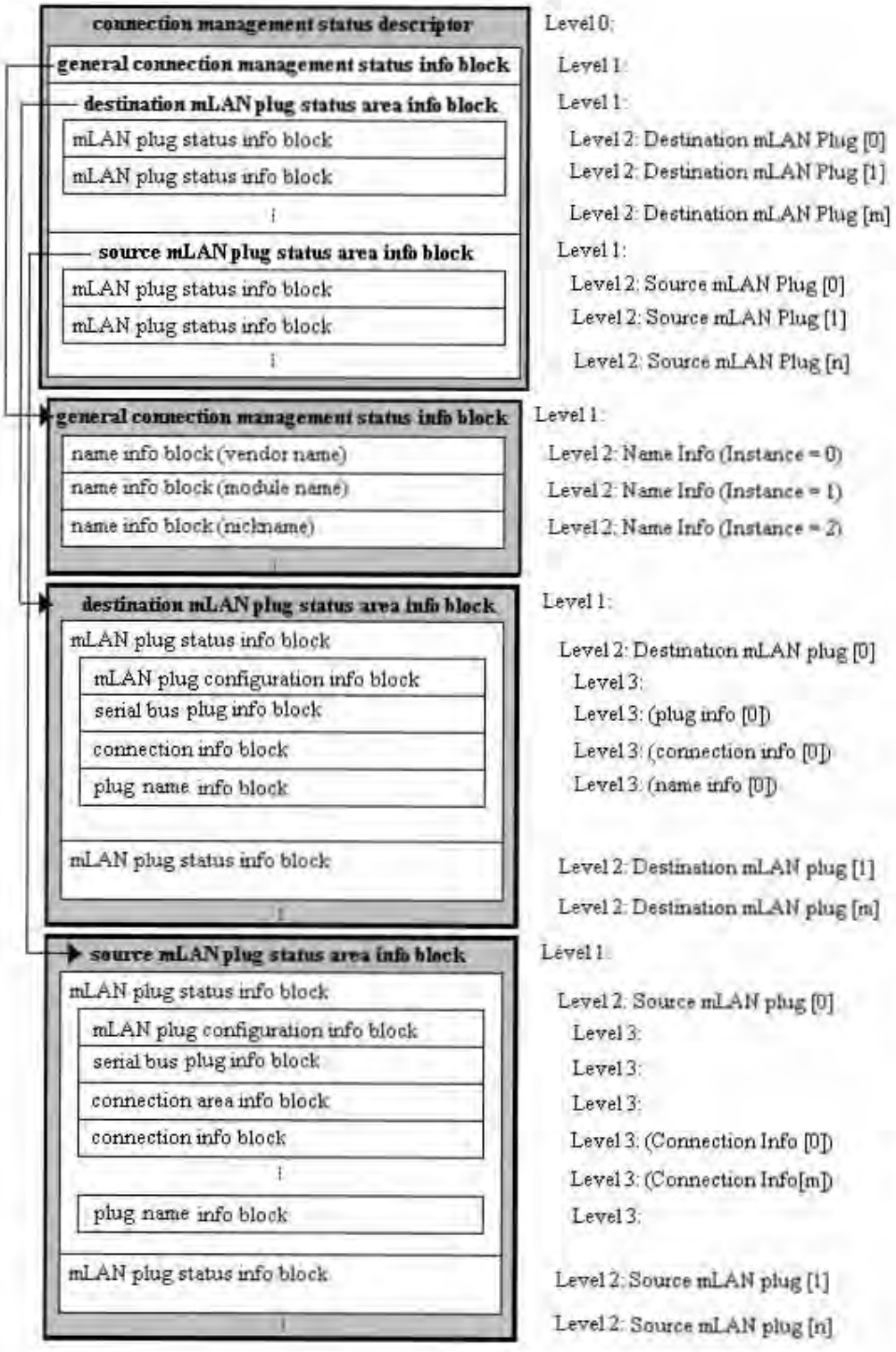


Figure 4-2: Descriptor information block structure implemented with mLAN Version 1 devices

The *connection management status descriptor* is expanded as shown in Figure 4-2 above.

The *general connection management status* information block contains a number of nested information blocks (indicated at ‘Level 2’) that specify the textual descriptors of the vendor, module and nickname of an mLAN device.

The *destination mLAN plug status area* information block gives information on the plug-types and number of input plugs implemented by an mLAN device, and also contains a number of nested information blocks that describe each plug. An *mLAN plug status* information block is implemented for each plug, and contains a number of nested information blocks that specify more specific information. These nested information blocks are shown at ‘Level 3’ in the figure and include:

- A *mLAN plug configuration* information block
- A *serial bus plug* information block
- A *connection* information block
- A *plug name* information block.

The *mLAN plug configuration* information block gives information on the *sequence ID*, *AM824 label* and the *sampling rate* at which an input plug is configured to receive data. The *serial bus plug* information block specifies information such as the index number of the iPCR – input plug control register (see section 3.4.1.1) – assigned to an mLAN plug, as well as the isochronous channel number being received by the mLAN plug. The *connection* information block gives the *GUID*, *node ID*, *plug ID* and *plug type* of a source plug connected to an input mLAN plug. The *plug name* information block specifies a textual descriptor of the name assigned to an input plug.

The *source mLAN plug status area* information block has a similar structure to the *destination mLAN plug status area* information block. However, the *mLAN plug status* information block implemented for each output mLAN plug, contains a *connection area* information block and a number of *connection* information blocks that give information on the input plugs of the other mLAN devices that are connected to the source mLAN plug. The number of target plug connections to a source mLAN plug is held within the *connection area* information block.

A *word clock* information block is also defined by mLAN Version 1 devices and is not described any further. Refer to the "mLAN 1.0 Connection and Control Specification" [Yamaha Corp., 2000a] for more information on this.

4.1.1.1 Reference Paths to access Information Blocks

Reference paths are defined by the "mLAN 1.0 Connection and Control Specification" to access the above mentioned information blocks. A reference path to an information block contains two sets of information:

1. The number of (nested) levels that indicate where the information to be accessed can be found.
2. A set of commands that specify where the information is located.

As an example, the reference path defined to access the *vendor name* information block is given in Figure 4-3 below.

info_block_reference_path for Vendor Name Raw Text Info Block		
address	**contents**	
	number_of_levels = 4	
	90 (connection management status descriptor)	level 0
	30(reference by type and instance count)	level 1
	90(MSB for IB_TYPE_GENERAL)	
	01(LSB for IB_TYPE_GENERAL)	
	00(level 1 instance)	
	30(reference by type and instance count)	level 2
	00(MSB for IB_TYPE_NAME)	
	0B(LSB for IB_TYPE_NAME)	
	00(instance for VENDOR_NAME_ID)	
	30(reference by type and instance count)	level 3
	00(MSB for IB_TYPE_RAW_TEXT)	
	0A(LSB for IB_TYPE_RAW_TEXT)	
	00(level 3 instance)	

Figure 4-3: Structure of the reference path used in accessing the vendor name information block

The number of levels required to access the raw content of the textual descriptor that describes the vendor name of an mLAN device is given to be '4'. From Figure 4-2 the *module name* information block is located at 'Level 2', which is the third level, but the raw content of the textual descriptor is a level further. After specifying the number of levels, the actual path (through each level) that specifies where the raw content of

the vendor name textual descriptor is located, follows. At level 0, the *connection management status descriptor* is referenced with the byte value ‘90’. This is followed by a reference to the *general connection management status* information block given by the byte values ‘30’, ‘90’, ‘01’ and ‘00’. At level 2, an indication is given to access the *vendor name* information block. This indication is specified by the bytes values ‘30’, ‘00’, ‘0B’ and ‘00’. The last byte value at this level specifies the instance position of the *vendor name* information block (see Figure 4-2). The byte values specified at level 3 indicate that the raw text of the *vendor name* information block is to be accessed. The “mLAN 1.0 Connection and Control Specification” [Yamaha Corp., 2000a] gives details on the byte values that are used in accessing other information blocks.

Note that these reference paths are specified as part of an AV/C command used in addressing mLAN devices.

4.1.1.2 Procedure used in Accessing Information Blocks

A standard sequence of procedures is required to be used in accessing the information blocks of mLAN Version 1 devices. These procedures are defined by four vendor-dependent commands and they include:

- OPEN INFO BLOCK control command
- OPEN INFO BLOCK status command
- READ INFO BLOCK control command
- WRITE INFO BLOCK control command

The OPEN INFO BLOCK control command is used to open an information block for read, write or close. The status of any one of these operations is retrieved via the OPEN INFO BLOCK status command. The READ INFO BLOCK control command is used to read the contents of an information block. Likewise, the WRITE INFO BLOCK control command is used to modify entries of an information block.

As an illustration, the structure of an OPEN INFO BLOCK control command is given below in Figure 4-4.

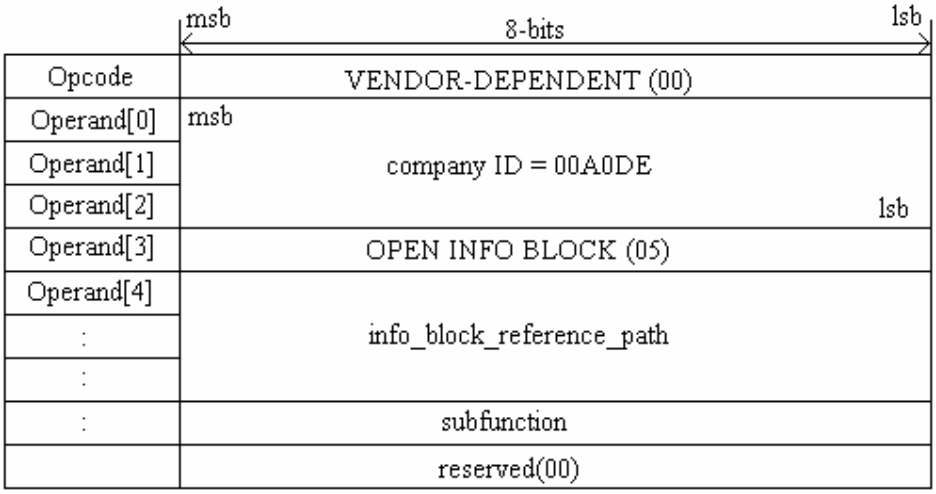


Figure 4-4: Structure of the OPEN INFO BLOCK control command

The *company ID* field specifies the *company ID* of Yamaha Corporation (00A0DEh). The *info_block_reference_path* specifies the path to an information block to be accessed, which is typically of the form shown in Figure 4-3. The *subfunction* field specifies one of the following values: 02h, 01h or 00h, each defining a subcommand that opens an information block for write, read or close. The format of the READ INFO BLOCK and WRITE INFO BLOCK commands are specified by the connection and control specification.

A controller requiring access to the contents of an information block is first required to check the status of the information block. This is achieved by using the OPEN INFO BLOCK status command. If the response to this request indicates that the information block is not being accessed by another controller, then the next step is to open the information block for read or write using the OPEN INFO BLOCK for read or write control command. If successful, the read or write is performed by issuing either the READ INFO BLOCK or WRITE INFO BLOCK control command. The information block is subsequently closed using the OPEN INFO BLOCK for close control command.

Yamaha has created a number of mLAN-compatible devices that implement the protocol described above. These include the mLAN8P, the CD8-mLAN, mLAN8E and mLAN-EX expansion boards. The mLAN8E and mLAN-EX expansion boards plug into synthesizers and samplers, providing them with mLAN communication

capability. The CD8-mLAN interface card was developed for the Yamaha range of mixer devices. The mLAN8P is a standalone audio/MIDI processor device that incorporates a built-in mLAN interface, and has legacy adapter capabilities, thereby enabling legacy devices to participate in an mLAN network. The following section takes a look at the design and implementation of a controller application, referred to as the Enabler, which enables, from a user's point of view, the various types of plugs (audio, MIDI and word clock) to be connected between these devices.

4.2 MLAN Version 1 Enabler Architecture

Yamaha Corporation provided an API specification entitled "mLAN Enabler Software Specification, version 0.5.0" [Yamaha Corp., 2002a], which was aimed at providing a portable interface for Enabler software developers and user-level applications developers to interact with various mLAN Version 1 device implementations. This API specification describes a set of attributes and functions required to be implemented by a number of classes that model the generic behaviour of the IEEE 1394 and mLAN features of 1394 PCI/PCMCIA interface cards, buses and mLAN devices.

The mLAN device object model defined by this specification is given in Figure 4-5. The *C1394Device* class represents and extracts the IEEE 1394 functionality common to all IEEE 1394 devices. Sub-classing this is the class *CmLANDevice*. The *CmLANDevice* class represents and abstracts the functionality common to all types of mLAN devices, which is indicated by the sub-class relationships to *CmLANPigeon*, *CmLANSparrow* and *CmLANPC*. It defines methods that allow access to various parameters of mLAN devices, including access to plug information, and also forms the top-level interface which user-level applications can use in interacting with a specific type of mLAN device.

The *CmLANPigeon*[8] class represents devices that implement the mLAN Version 1 architecture. At the time this Enabler software specification was released, these devices were the current mLAN devices available on the consumer market.

[8] mLAN Pigeon is the term used to refer to mLAN Version 1 devices

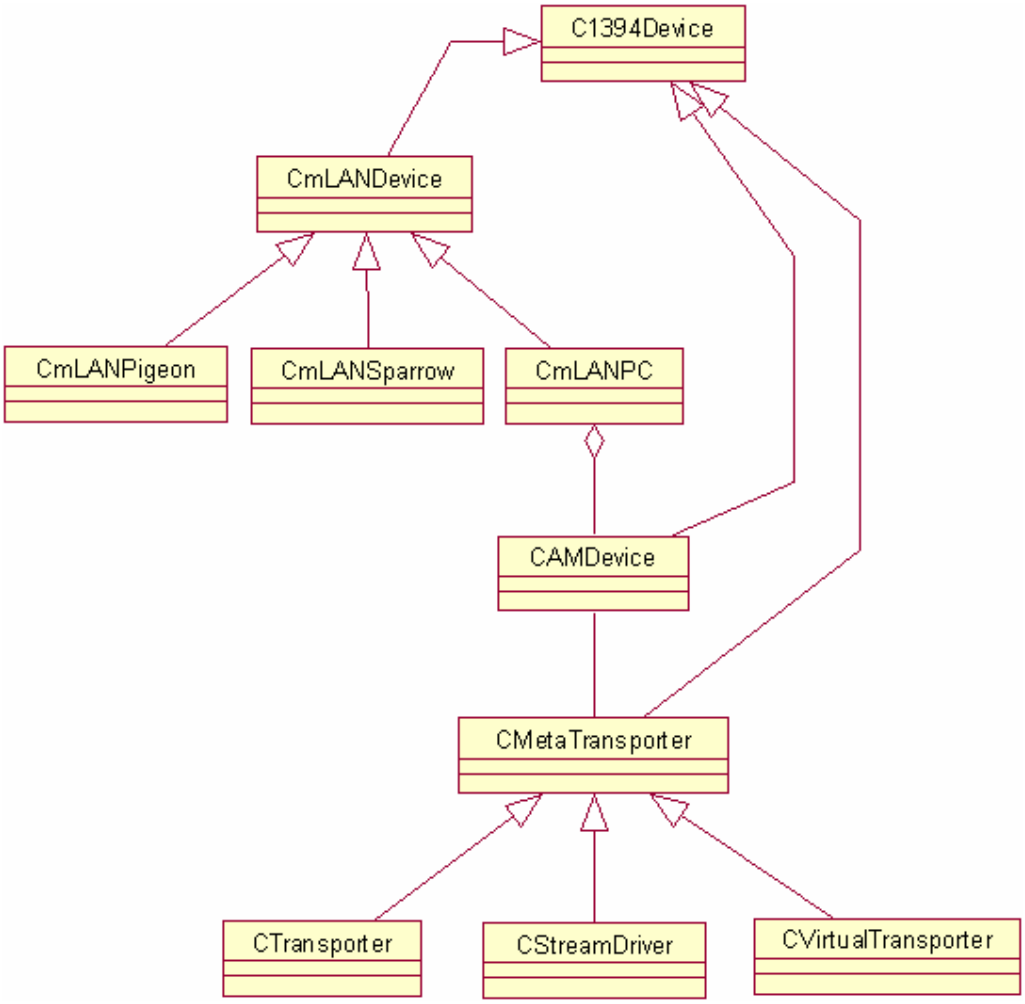


Figure 4-5: mLAN device object model defined by the Enabler specification

The *CmLANSparrow* class was added to represent the future generation of mLAN devices. The *CmLANPC* class represents the implementation of the mLAN architecture on a PC. In the PC implementation, the PC is modelled to contain a number of devices (including the PC driver) that implement IEC 61883-6 level connections. Note that this model only reflected a conceptual view of the next implementation phase of the mLAN Version 1 Enabler, and as such was not implemented at the time of conducting this research. From an application's point of view, it was envisioned that the PC would umbrella a number of 'basic' devices that implement IEC 61883-6 level connections, creating and managing plugs for these devices, and revealing them as plugs of the *CmLANPC*. In other words, the granularity of these 'basic' devices would not be exposed to an application, but would have their plugs implemented as part of the *CmLANPC* device.

In view of this, Yamaha's goal was to provide an AV/C implementation for these 'basic' devices in order to enable high-level mLAN plug connections between these devices and other types of mLAN Pigeon devices. From the object model, this is represented by the aggregation of the *CAMDevice* class to the *CmLANPC* class. The *CAMDevice* class has an association to an abstract class, *CMetaTransporter*, which has its class methods realized by the classes *CTransporter*, *CStreamDriver* and *CVirtualTransporter*. These classes describe the operations of a 'basic' device, which is solely responsible for the transport of audio and MIDI data. The next chapter, "MLAN Version 2. A Plural Node Approach", describes this in greater detail.

The *CAMDevice* class is intended to model devices that implement the A/M data protocol defined by IEC 61883-6 [IEC, 2005]. It also manages serial bus plugs, plug control registers, isochronous streams, and isochronous resources, and provides a means to access the properties of each isochronous stream. This includes the sampling frequency, number of sequences, and the data type carried by the stream. The *CMetaTransporter* class is generic, and abstracts the functionality that is common to devices that support transmission and reception of isochronous streams using IEC 61883-6. These devices are defined to be of three different types, represented by the classes *CTransporter*, *CStreamDriver* and *CVirtualTransporter*. As already mentioned, these devices are 'basic' in the sense that they are responsible for the transport of audio and MIDI data, and hence are referred to as Transporters.

The *CTransporter* class models stand-alone Transporter devices, the *CStreamDriver* class represents the implementation of 'Transporter' capabilities on a PC, and the *CVirtualTransporter* class models the behaviour of a generic IEC 61883-6 device, in so doing, causing it to behave like a Transporter.

In addition to these classes, the Enabler specification also defines various other classes that model 1394 PCI/PCMCIA interface cards, 1394 buses and the basic IEEE 1394 capabilities of devices. These are represented by the classes *C1394*, *C1394Interface*, *C1394Bus*, *C1394Device*, *CmLANInterface*, *CmLANBus* and *CmLANDevice*, and have the relationship shown below in Figure 4-6.

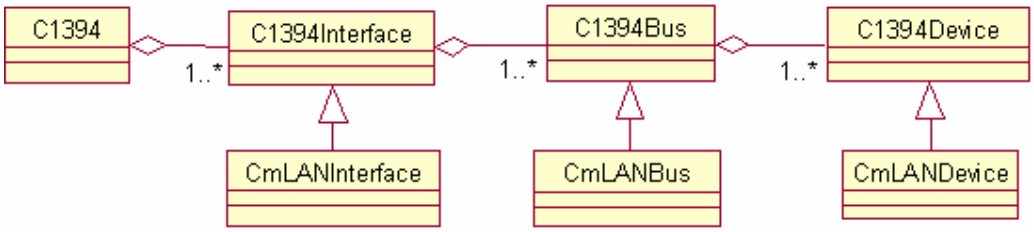


Figure 4-6: Classes that model IEEE 1394 interfaces, buses and devices

The *C1394* class represents the entire Enabler system and is the first port of entry that enables the use of the Enabler's functionality. The *C1394Device-CmLANDevice* class relationship is expanded as indicated previously in Figure 4-5. The *CmLANInterface* and *CmLANBus* class define mLAN-related extensions to their respective base classes. They also define methods to create and return pointers to objects that model different mLAN device implementations.

The Enabler specification identifies each interface, bus or device class object using an object identifier, also known as an *object ID*. The implementation of this identifier is not given by the specification, but is required to be unique for a particular 1394 entity, thus enabling an application to uniquely identify a class object that represents a particular 1394 interface, bus or device. Applications requiring access to the Enabler are first required to create an object of the *C1394* class from which objectIDs are retrieved in order to create the relevant class objects. The Enabler specification [Yamaha Corp., 2002a] gives details of how an application would typically interact with an Enabler.

It is important to mention that at the time of implementing, there were no IEEE 1394 bridges that permitted the use of multiple 1394 buses. Hence, the only bus that existed was the local bus attached to a workstation. The issue of 1394 bridging will be revisited in subsequent chapters.

The next subsection takes a look at the top-level view of the Windows Enabler implementation developed by Yamaha, highlighting various constraints imposed by the operating system that lead to the development of a Linux equivalent. The Linux Enabler design and implementation forms the basis of this research.

4.2.1 The Windows mLAN Version 1 Enabler

Yamaha Corporation developed a patch bay (controller) application for the Windows and Macintosh platforms. This application allows for audio and MIDI plug connections, as well as word clock synchronization to be established between mLAN Version 1 devices. A snapshot of the patch bay application is shown in Figure 4-7 below.

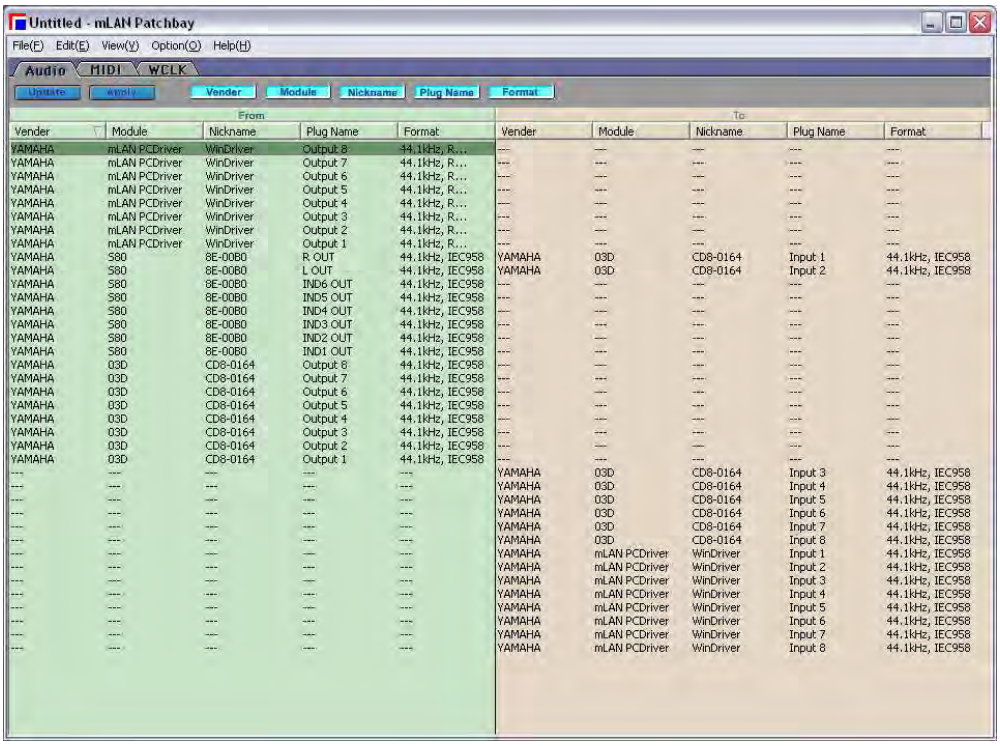


Figure 4-7: Snapshot of Yamaha's patch bay application developed for mLAN Version 1 devices

Three tabbed pages, "Audio", "MIDI" and "WCLK", are shown; the audio page is currently displayed. The left-hand side of the display indicates the possible "From" (source) plugs of the devices attached to the IEEE 1394 bus. The right-hand side indicates the possible "To" (destination) plugs of the mLAN deceives. On either side of the "From" or "To" section, the "Vendor", "Module" and "Nickname" of the mLAN device having a plug with "Plug Name" and "Format" are displayed.

Plugs that have no target connections have the symbol "---" placed adjacent to them. Conversely, plugs with target connections have the information of the corresponding target plug placed adjacent to them. From the snapshot, the output plugs "R OUT"

and “L OUT” of the “S80” device are shown to have connections; they are connected to “Input 1” and “Input 2” of the “O3D” device. The “MIDI” and “WCLK” tabs have similar GUI displays.

Connections between the various types of plugs are established from a series of mouse button clicks. The mLAN Patchbay Owner’s Manual [Yamaha Corp., 2000b] gives further details on the features and operation of the patch bay application.

The mLAN device object model implemented by this Enabler does not exactly follow the object model proposed by the Enabler specification (shown in Figure 4-5). The *CmLANPC* class does not have an aggregation of *CAMDevice* class objects as previously indicated; instead it is implemented to directly interact with an mLAN *streaming driver* that provides mLAN Version 1 capabilities for a PC.

The need to utilize IEEE 1394 bridges in studio and sound installation environments became increasingly important with regard to mLAN. This implied that the Enabler and also any 1394 Windows or OSX drivers would be required to be modified to incorporate bridging functionalities. However, these changes could not be implemented immediately because the code sources of the 1394 driver implementation for the Windows and Macintosh platforms were not made available. The Linux platform was considered in exploring this possibility, primarily because of its open source nature. Having a Linux Enabler also provided an alternative to the Windows and OSX Enabler versions. The first step in achieving a “bridge-aware” Enabler was to implement for Linux, an Enabler defined by the Enabler specification. The next subsection describes the implementation strategies adopted for the Linux Enabler.

4.2.2 The Linux mLAN Version 1 Enabler

Prior to developing the Enabler, the Linux operating system had to be configured for IEEE 1394 operations. This is discussed in the next paragraph.

4.2.2.1 Configuring Linux for IEEE 1394

The IEEE 1394 drivers for Linux, at the time of the implementation, were experimental and were not directly supported by the kernel. Kernel version 2.4.17 or greater, was considered to be stable with respect to IEEE 1394 support. In this implementation, the Linux kernel version 2.4.17 was used and the necessary configurations made to enable the IEEE 1394 drivers. The "IEEE 1394 for Linux" website [Linux1394, 2005] gives detailed information on how Linux can be set up for 1394. The later kernels however, do incorporate IEEE 1394 drivers as part of their build.

On top of these drivers, there exists a user-space library that facilitates IEEE 1394 communication from user applications directly to the 1394 bus. This library, known as *libraw*, provides "raw" access to the 1394 bus. The hierarchy of the 1394 implementation in Linux is shown in Figure 4-8. This figure is adapted from [Linux1394, 2005].

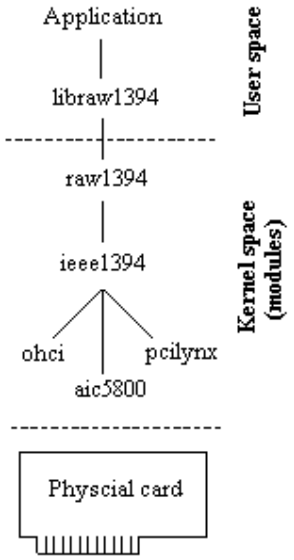


Figure 4-8 : IEEE 1394 implementation in Linux

The *libraw* library defines various standard IEEE 1394 methods that allow applications to retrieve information about the nodes attached to a bus, perform asynchronous and isochronous transactions, register bus reset and FCP data change call-back routines and to set up address range spaces. It communicates with the high-level driver *raw1394* that provides an interface for user space applications to access

the raw 1394 bus. Below the *raw1394* module, is the module *IEEE 1394* that forms the core of the entire 1394 subsystem. It manages all high-level and low-level drivers in the subsystem, handles transactions, and provides a mechanism for event triggering. This interfaces to a low-level (hardware) driver module that is specific to an actual IEEE 1394 interface card. Figure 4-8 shows the *ohci1394* hardware module, which is an implementation based on the 1394 Open Host Controller Interface (OHCI) Specification. Two other hardware modules exists, these are called *aic5800* and *pcilynx*. The module *aic5800* corresponds to IEEE 1394 interface cards with the Adaptec AIC-5800 PCI-IEEE 1394 chip implementation. The *pcilynx* module corresponds to IEEE 1394 interface cards with the Texas Instruments PCILynx chip implementation.

4.2.2.2 Implementation Strategies

The first part of the implementation was spent using the *FireSpy400* IEEE 1394 bus analyzer developed by Dap Technology [Dap Technology, 2005] in studying the nature of asynchronous packets transmitted on the 1394 bus. These packets were generated from a number of plug connections and disconnections that were established using the already existing Windows version of the mLAN Patchbay application – version 0.0.4. This study was done to gain a practical insight into the format of the FCP packets required to make and break connections. The payload structure of the various FCP packets captured by FireSpy400 coincided with that specified in the "mLAN 1.0 Connection and Control Specification" [Yamaha Corp., 2000a]. Recall from section 3.4.1.3 that FCP is used by AV/C devices in transmitting command and response messages between each other.

A connect and disconnect between an output plug of a source device and an input plug of a destination device required several reads and writes to the appropriate information blocks of both devices; recall that information blocks are data structures that store pertinent information about a device. The "writes" need to be issued in an ordered sequence. From the observations made using the FireSpy bus analyzer, the required sequence of writes for establishing a connection was as follows:

1. A write to the *connection info block* of the output plug of the source device with the appropriate information about the input plug of the destination device.

2. A write to the *configuration info block* of the input plug of the destination device, with the appropriate configuration information about the output plug of the source device.
3. A write to the *serial bus plug info block* of the input plug of the destination device, with the appropriate serial bus information about the output plug of the source device.
4. A write to the *connection info block* of the input plug of the destination device with the appropriate information about the output plug of the source device.

The two writes to the *connection info block* (1 and 4) contain information about the global unique identifier, node ID and plug ID of the respective target devices. This information allows a device to be "aware" of its target connection, and hence allows connections to be re-established when a bus reset occurs, or when the device is power-cycled. The write to the *configuration info block* and *serial bus plug info block* of the destination device, contains information such as the *sequence ID, AM824 label, sampling rate*, and *isochronous channel* of the source plug that is to be set to the destination plug. This enables it to "pick-up" a sequence (or subsequence) of isochronous data from the isochronous stream that contains the audio/MIDI data channelled by the source plug.

In a similar manner, the sequence of writes necessary for breaking a connection was as follows:

1. A write to the *connection info block* of the output plug of the source device.
2. A write to the *connection info block* of the input plug of the destination device.

The payloads of these write packets specify values that indicate an invalid connection.

In addition to the above described sequence of steps, the steps involved in accessing information blocks (described in section 4.1.1) using the OPEN INFO BLOCK, READ INFO BLOCK and WRITE INFO BLOCK commands was also confirmed.

Enabler Implementation

It was mentioned that the *libraw* library defines various methods that allow user-level applications to have direct access to the 1394 bus. Some of the methods defined by *libraw* are shown in Table 4-1 below.

Method	**Description**
raw1394_new_handle(…)	Gets a handle to the Linux 1394 low-level implementation
raw1394_destroy_handle(…)	Destroys a handle obtained by calling raw1394_new_handle()
raw1394_get_irm_id(…)	Returns the physical ID of the node acting as the isochronous resource manager.
raw1394_setport(…)	Sets a particular IEEE 1394 PCI/PCMCIA card for 1394 communication.
raw1394_get_nodecount(…)	Returns the number of nodes attached to the IEEE 1394 bus of the PCI/PCMCIA card interface.
raw1394_reset_bus(…)	Issues a bus reset
raw1394_set_bus_reset_handler(…)	Sets a call-back that is to be invoked when a bus rest occurs.
raw1394_set_fcp_handler(…)	Sets a call-back that is to be called when the local FCP_COMMAND or FCP_RESPONSE register gets written to.
raw1394_read(…), raw1394_write(…), raw1394_lock(…)	Issues IEEE 1394 read, write and lock transactions respectively.
raw1394_start_fcp_listen(…)	Handles command and response messages sent to the local node's FCP_COMMAND and FCP_RESPONSE registers.

Table 4-1: Various methods implemented by the *libraw* library

Most of the function implementations of the various (Linux equivalent) methods of the Enabler classes make use of these *libraw* methods. The *C1394* class, which represents the entire IEEE 1394 system, obtains a handle to the Linux 1394 low-level implementation by calling the *raw1394_new_handle(…)* method. A specific interface through which bus transactions is to be performed, is set via a call to the *Open(…)* method of the *C1394Interface* class. This method uses the *libraw* methods

raw1394_set_port(…) and *raw1394_start_fcp_listen(…)* to first set an interface card for use and then start FCP receptions on the selected interface. The *ResetBus(…)* and the *SetBusResetNotify(…)* methods of the *C1394Bus* class makes use of *raw1394_reset_bus(…)* and *raw1394_set_bus_reset_handler(…).* The *ReadQuadlet(…), ReadBlock(…), WriteQuadlet(…)* and *WriteBlock(…)* methods defined by the *C1394Interface* and *C1394Device* class, use the *raw1394_read(…)* and *raw1394_write(…)* functions respectively.

A *GetConfigurationROM(…)* method was introduced to the *C1394Device* class. This method retrieves the configuration ROM information of an IEEE 1394 device by issuing several *ReadQuadlet(…)* and *ReadBlock(…)* requests to the corresponding device's address space.

The *CmLANBus* class creates and returns a pointer to an object of the class *CmLANPigeon* by examining the configuration ROM of the mLAN devices attached to the bus. For each device, it retrieves the configuration ROM information, checks if the device is an mLAN Pigeon type and then creates and returns a pointer to an mLAN Pigeon class object. The configuration ROM of an mLAN Pigeon device must have a *unit directory* that specifies a specification ID of 0000A02Dh, a software version of 00010001h and a textual descriptor that contains the string "mLAN".

The *CmLANPigeon* class models mLAN Version 1 devices by retrieving and modifying the information blocks implemented within the device. In achieving this, an FCP packet, similar to the one described in Figure 4-4, is constructed and also includes the reference path to the information block to be accessed. The FCP packet is constructed using the method *CreateFCPCommandPacket (…)*. This method makes use of other methods in generating the packet structure. These include:

- *AppendFCPHeader (…)*
- *AppendInfoBlock (…)*
- *PersonalizeFCPPacket (…)*

The use of these methods is illustrated using the diagram shown in Figure 4-9 below.

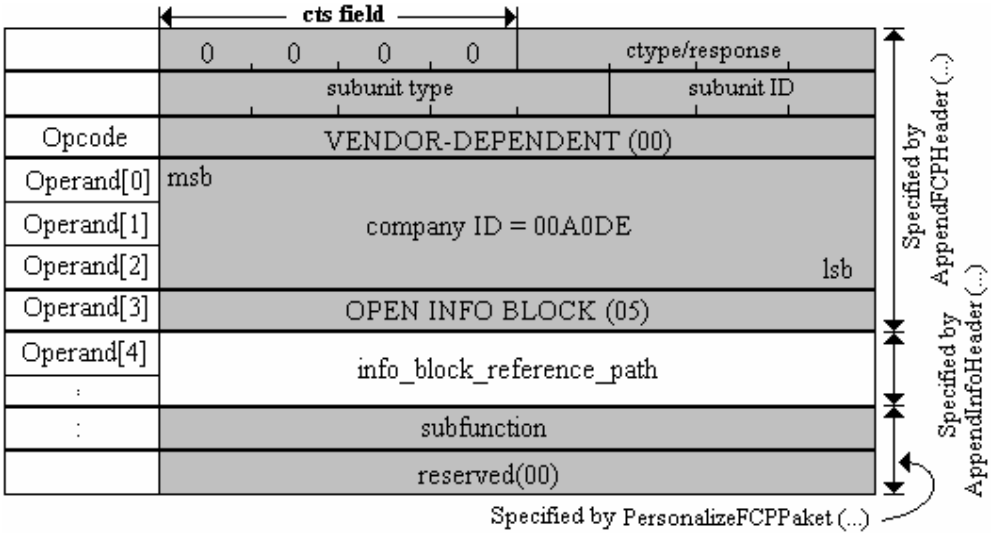


Figure 4-9: Creating an FCP packet to access an information block

Figure 4-9 shows the FCP data that is contained within an asynchronous write block packet. The *AppendFCPHeader (…)* method builds up the FCP data up to and including the value specified for 'Operand[3]'. This includes the standard AV/C fields – *cts*, *ctype*, *subunit_type* and *subunit_ID*, the 'Opcode' value – 00b, which indicates a vendor dependent command, followed by the company ID and the command to be executed. The command to be executed is specified at 'Operand[3]'. This specifies a command to open an information block, read an information block, or write to an information block. Figure 4-9 specifies a command to open an information block.

The A*ppendInfoBlock (…)* method appends the appropriate *info_block_reference_path* required to access information contained within a particular information block. The structure of the reference path specified is identical to that described in Figure 4-3, where the number of levels that indicate where the information to be accessed can be found is specified, together with the set of commands that specify where the information is located.

The method *PersonalizeFCPPacket (…),* personalizes the FCP data according to the value specified in 'Operand[3]'. If the OPEN INFO BLOCK command is specified for 'Operand[3]', the appropriate *subfunction* is specified. The subfunction indicates a secondary operation that is to be performed in addition to opening the information block; this can either be a read, write or close. If a read is specified, the information

block is opened for reading. If a write is specified, the information block is opened for editing. Similarly, if a close is specified, an information block that has been previously opened is closed. If the READ INFO BLOCK command is specified for 'Operand[3]', the *data_length* and the *offset values* that are to be read are specified. If the WRITE INFO BLOCK command is specified, the *subfunction, group_tag, replacement_data_length, data_address, original_data_length* and the *replacement_info_block_data* values are specified. The "mLAN 1.0 Connection and Control Specification" [Yamaha Corp., 2000a] gives the definitions of these fields.

Information accessed from the information blocks of mLAN Version 1 devices are cached in various *structs* implemented within the *CmLANPigeon* class. These *structs* are defined to mirror the structure of information blocks that they represent. They are defined to contain:

- The vendor name, module name and nickname of a device.
- The number and various types of source and destination plugs implemented within the device.
- Information regarding the properties of each plug as well as its connection information.

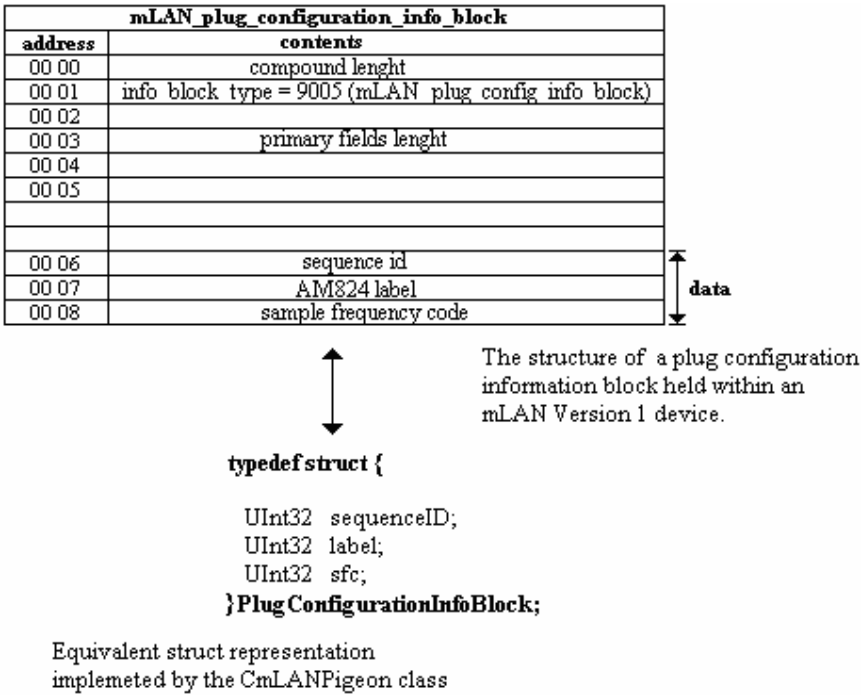


Figure 4-10: Struct representation for the *mLAN plug configuration* information block

Figure 4-10 above shows the 'information block ↔ *struct*' relationship for the *mLAN plug configuration* information block implemented within an mLAN device. The values held by the fields of these structs are updated when the content of the corresponding information block it represents is modified.

In addition to the methods that access and retrieve the data content held within information blocks, the *CmLANPigeon* class also implements other methods to perform *pigeon-specific* plug connections and disconnections. In these implementations, a number of FCP packets are created, using the *CreateFCPCommandPacket (…)* method described above, and issued to the mLAN device. These FCP packets also specify the appropriate reference path for the information blocks to be accessed.

Recall the procedure for establishing plug connections, discussed at the beginning of section 4.2.2.2. The information blocks to be accessed include:

- The *connection* information block of the source plug
- The *mLAN plug configuration* information block of the destination plug
- The *serial bus plug* information block of the destination plug
- The *connection* information block of the destination plug

For each information block accessed, the corresponding information of the target plug is written. For example, when the *mLAN plug configuration* information block of the destination plug is being accessed, the data content of the *mLAN plug configuration* information block of the source plug is written to the destination plug. A similar approach is adopted for breaking plug connections. Recall the procedure for breaking plug connections, also discussed at the beginning of section 4.2.2.2.

It is important to mention that for each information block accessed by the Enabler, the sequence of steps described in section 4.1.1.2 is followed. This required an OPEN INFO BLOCK for status command to be issued first, followed by an OPEN INFO BLOCK for read or write, then a subsequent READ/WRITE INFO BLOCK command and then an OPEN INFO BLOCK for close.

The *CmLANPigeon* class also realizes the methods of its base class, *CmLANDevice*, which forms the mLAN device interface between an application and the actual pigeon implementation of the Enabler. The next subsection describes the Control Panel/patch bay application that was developed for the Linux mLAN Version 1 Enabler.

4.2.2.3 Control Panel/Patchbay Application

The user-level application for the Linux mLAN Version 1 Enabler was developed using the Qt Toolkit [Dalheimer, 2002]. A screenshot of the main window is shown in Figure 4-11. Two group boxes are shown, these being labelled “mLAN Device List” and “Current mLAN Plug Connections”, respectively.

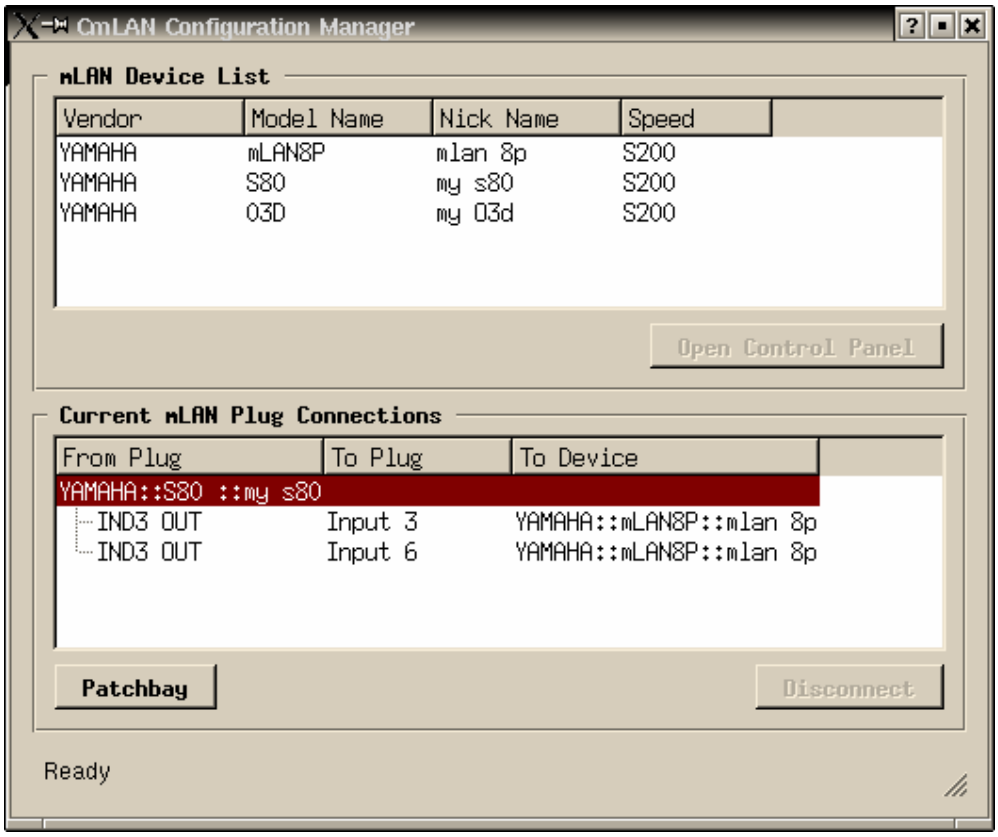


Figure 4-11: Main window of the mLAN control panel application

The “mLAN Device List” group box displays a list of the mLAN Pigeon devices attached to the workstation. For each displayed device, a control panel dialog can be opened, from which detailed information (including plug information) about the

device is displayed. The control panel and plug information dialog are shown in Figure 4-12 and Figure 4-13, respectively.

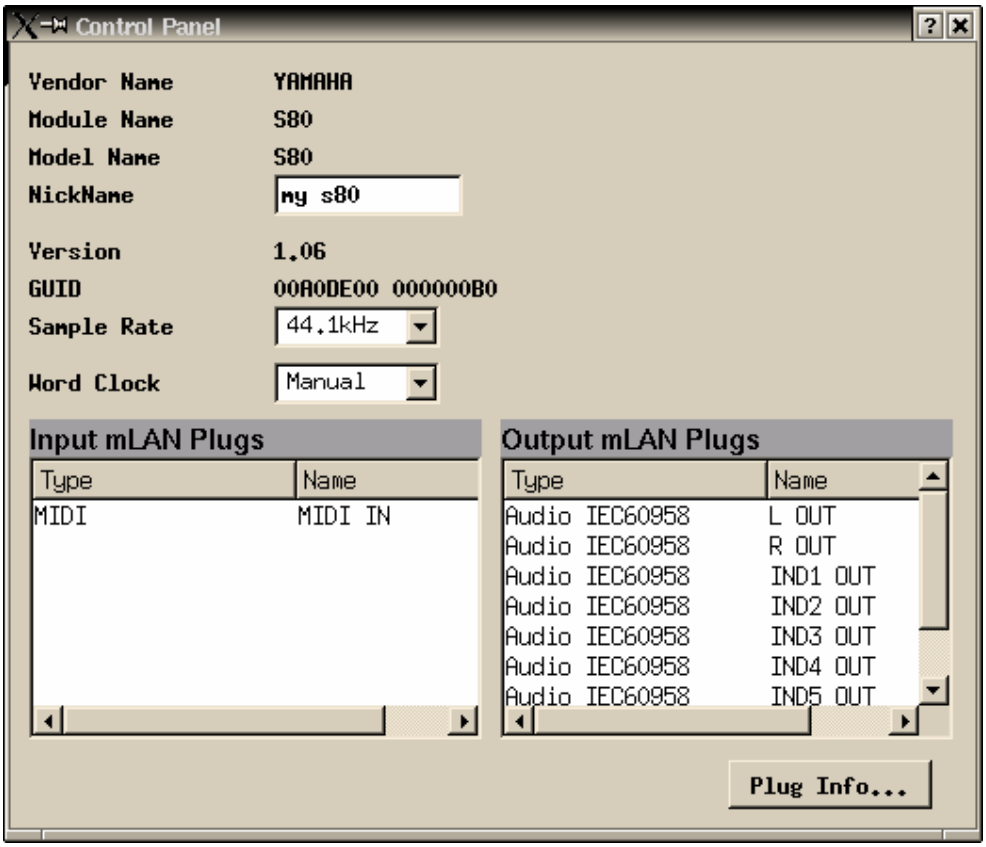


Figure 4-12: Control panel dialog for mLAN Pigeon devices

The control panel dialog gives information about the vendor, module and model of the device. The nickname, version, and GUID are also displayed, in addition to the currently selected sample rate and word clock synchronization. The control panel allows the nickname, sampling rate and word clock synchronization to be modified. Two list views are displayed. These contain a list of the input and output plugs implemented by the device. Double clicking on a selected plug or clicking on the "Plug Info…" command button reveals detailed information about the selected plug. This information includes the ID, data type, sample rate, transmission or reception isochronous channel number, and sequence number, and a list of connected target plugs. An example plug information dialog box is shown in Figure 4-13.

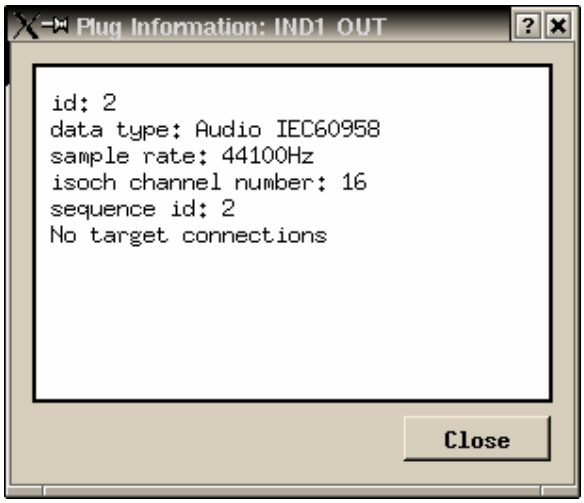


Figure 4-13: Plug information dialog

The “Current mLAN Plug Connections” group box, of the main window shown in Figure 4-11, displays a list of the currently connected plugs of all the devices attached to the workstation. The display currently shows two plug connections from plug “IND3 OUT” of the “S80” to “Input3” and “Input6” of the “mLAN8P”. A selected connected plug can be disconnected by clicking on the “Disconnect” command button, or via a patch bay dialog, which is invoked by clicking the “Patchbay” command button. The subsequent patch bay dialog is shown below in Figure 4-14.

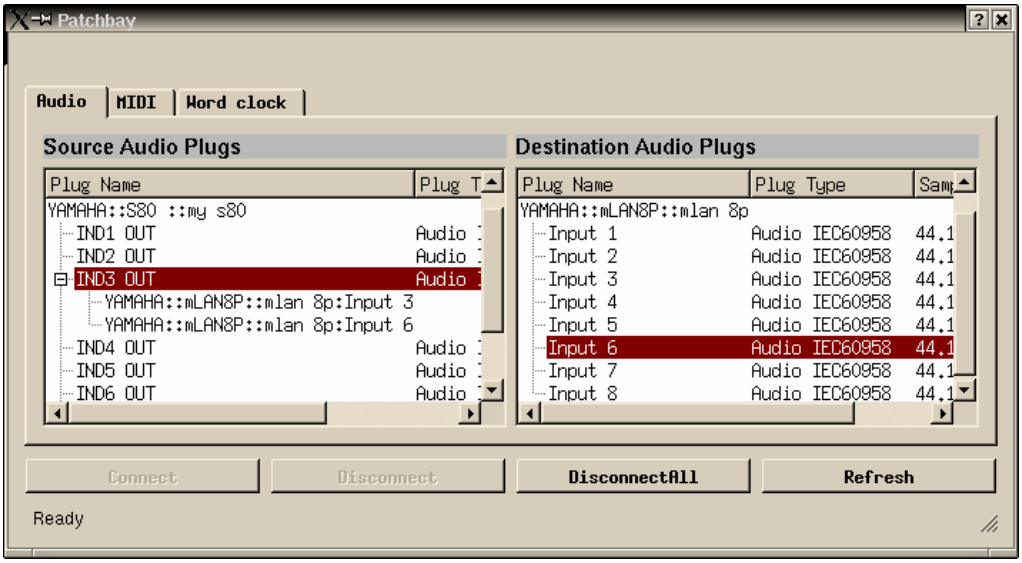


Figure 4-14: mLAN patch bay dialog

This dialog permits audio and MIDI to be connected and disconnected, as well as setting up word clock synchronization among various mLAN devices. It consists of a tabbed panel, list views, and command buttons. The tabbed panel has 3 tabs, each referring to “Audio”, “MIDI” and “Word clock”, respectively. Each tab has two list views that display the source and destination plugs, including connection information of the plugs of the attached mLAN devices. The four command buttons: *Connect*, *Disconnect*, *DisconnectAll* and *Refresh* allow the various types of plugs to be connected, disconnected and refreshed accordingly. A user manual has been written for this application and can be obtained from Network Audio Solutions [Network Audio Solutions, 2002].

The mLAN technology provides a solution to the problem of connection management of various vendor-specific mLAN implementations. In addition to the Yamaha mLAN Version 1 devices mentioned thus far, a number of other manufacturers have also developed mLAN expansion boards or devices that implement the mLAN Version 1 architecture. These include:

- Korg [Korg USA, 2005] with an EXB-mLAN expansion board that can be plugged into its Triton synthesizers/samplers,
- Otari [Otari Inc., 2005], with an mLAN compatible “Networked Audio Distribution Unit”. This device, also known as the ND20, allows audio to be distributed across an IEEE 1394 network.
- Apogee [Apogee Electronics Corp., 2005], with an mLAN interface card for their “AMBus”-based mic preamp and A/D conversion products such as the Trak2 and AD-8000,
- PreSonus’s [PreSonus Audio Electronics Inc., 2004] mLAN compatible breakout box, the FIREStation. This allows for the input and output of multiple formats of audio, which can be recorded and played backed using recording software.

The next section describes the various advantages and disadvantages of the mLAN Version 1 architecture, which make it desirable or undesirable for it to be used in a professional studio environment.

4.3 Qualitative and Quantitative Analysis

A significant feature of the version 1 architecture is the plug abstraction layer implemented within devices. Recall from sections 3.4.2 and 4.1 that the plug information is stored within descriptors and information block structures. This implies that the state of the plugs is held within non-volatile memory, and it is possible for devices themselves to restore their connection state after power-down.

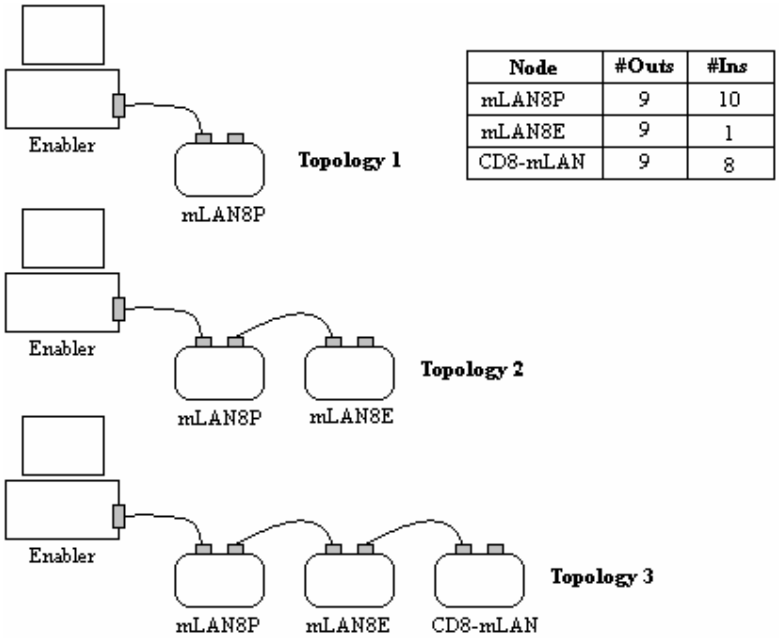
However, the implementation of the plug abstractions requires a large amount of memory, and hence increases the cost of providing mLAN compatibility. Any changes to the implementation, for example bug fixes, upgrades, and possibly new approaches to connection management, require firmware upgrades in all mLAN compatible devices. Furthermore, non-mLAN chip vendors who wish to provide mLAN compatibility within their chip sets will require considerable effort and mLAN expertise. Also, not all applications require the level of sophistication provided by the mLAN plug abstractions; for example fixed installations, where an initial set up is all that is required, using low level isochronous channel and sequence selections.

In addition to these qualitative disadvantages, the processing time required for a control application to retrieve information from mLAN Version 1 devices, and also to establish and break plug connections, is significantly long. This includes the time taken for FCP command and response messages to be relayed to and fro on the bus. The delay can be undesirable in certain professional operational environments. For example, in broadcasting studios where fast switching capabilities are required between audio plugs that carry feeds from different satellite studios.

Various timing measurements have been performed on the Linux Enabler to establish an idea of the time required to enumerate a number of mLAN Version 1 devices. Timing measurements for plug connects, disconnects, and retrieval of plug connections are also given. Up to three mLAN devices (mLAN8P, CD8-mLAN expansion board for the Yamaha O3D mixer and the mLAN8E expansion board for the S80 synthesizer) were used in the measurement. The procedure adopted in performing these measurements, and the timing results are discussed below.

4.3.1 Measuring Speed of Network Enumeration

Timing measurements were taken to determine the speed at which an application enumerates a network by creating Enabler objects for the IEEE 1394 interfaces, buses and devices on the network. Three sets of measurements were taken for the three different network topologies as shown in Figure 4-15. The timing results are shown in Table 4-2.



Node	#Outs	#Ins
mLAN8P	9	10
mLAN8E	9	1
CD8-mLAN	9	8

Figure 4-15: Network topologies used in measuring enumeration speed

	Timing Measurements [seconds]			
Topology	**1st**	**2nd**	**3rd**	**Average**
1	5.38	5.30	5.24	*5.31*
2	8.48	8.39	8.47	*8.45*
3	14.13	14.10	14.07	*14.10*

Table 4-2: Timing measurements for mLAN Version 1 device enumeration

The timing measurements taken for each topology were performed under the same networking conditions. The only varying parameter in performing these measurements is the different number of input plugs supported by each device; mLAN8P implements 10 input plugs, mLAN8E implements 1 and CD8-mLAN

implements 8. Irrespective of this, it is clear that enumerating mLAN Version 1 devices is in the order of seconds, with about 3.41 seconds required to enumerate an mLAN8E node. This order of magnitude is largely due to the overhead caused by the FCP command and response AV/C messages used by the Enabler in reading the information blocks of the devices. Recall from Figure 4-2 that a number of information blocks are are implemented by an mLAN Version 1 device which requires access via FCP messages. Though this overhead may be tolerable in studio environments, it is undesirable for large installations.

4.3.2 Measuring Speed of Plug Operations

Performance measurements were taken for a number of plug operations – plug connections, plug disconnections, and retrieving plug connections of source plugs. The setup shown in Figure 4-16 was used in performing the measurements for three sets of five operations.

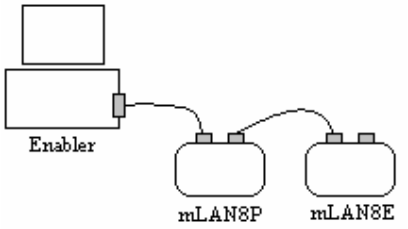


Figure 4-16: Device setup used in performing timing measurements

The results of the timing measurements are summarized below.

	Timing Measurements		
	Connections [s]	**Disconnections [s]**	**Get Connections[µs]**
1st Five Operations	11.6	8.2	11
2nd Five Operations	12.4	7.9	11
3rd Five Operations	11.9	8.4	12
Average of 15 Operations	*2.4*	*1.6*	*2.3*

Table 4-3: Timing measurements for a number of plug operations

Note that the source plugs used in timing the speed of retrieving plug connections, did not have any target connections. This approach has been used consistently for all plug

connection retrieval timings; since it avoids additional overhead of storing plug connection information.

From the results, the average time taken by the Enabler to establish a plug connection is 2.4 seconds. It also takes an average time of 1.6 seconds to disconnect a pair of connected plugs, and 2.3 μs to retrieve plug connections of source plugs. For plug connections and disconnections, this order of magnitude is largely due to the overhead in FCP communications between the source and destination devices, and the Enabler. Recall from section 4.2.2.2 that a number of information blocks of the source and destination devices are modified by the Enabler in order to connect or disconnect plugs. In environments where fast plug-switching capabilities are required, such as in broadcasting studios, the mLAN Version 1 Enabler and corresponding devices are not suitable. A greater speed-up is required in order to deploy the mLAN technology in such environments.

The high speed observed in retrieving plug connections of source plugs is due to the caching implementation used by the Enabler. After network enumeration, all the attributes of the devices on the network are retrieved and cached by the Enabler. This goes to show that an application developed to interact with the Enabler can update its display in the order of microseconds, and the bottle-neck in speed occurs when bus transactions occur on the IEEE 1394 bus.

The above considerations led to the creation of a new mLAN implementation architecture.

4.4 Summary

This chapter describes the mLAN Version 1 architecture, as well as a workstation-based controller application that provides a complete networking solution for the audio industry. The mLAN Version 1 architecture is the first generation of Yamaha's mLAN implementation. An mLAN device, according to this architecture, implements descriptors and information blocks according to the AV/C Descriptor Mechanism specification. These information blocks and descriptors are required to be accessed using FCP messages containing vendor dependent commands. The design of a

workstation-based controller application (the Enabler) is also described in this chapter. A Windows version of this Enabler has been implemented by Yamaha, and an equivalent implementation described, as part of this research, for the Linux platform. Using the Linux Enabler, a number of timing measurements were conducted to determine the speed performance of various operations performed on mLAN Version 1 devices. It was found that the speed taken to enumerate an mLAN Version 1 device with 9 output plugs and 1 input plug was 3.41s, while the time taken to make and break a plug connection was found to be 2.4s and 1.6s respectively. The results of these measurements, in addition to a number of other factors led to the creation of a new mLAN implementation, which in addition to delivering a better speed performance, would also facilitate the adoption of mLAN within the audio industry. This new mLAN implementation is described in the next chapter.

Chapter 5

5. MLAN Version 2. A Plural Node Approach

The mLAN Version 2 architecture, also known as the plural node architecture, is a two part mLAN Audio and Music (A/M) device implementation, where the "connection management" capabilities and the actual transport of data are implemented in separate nodes. This differs from the mechanism adopted for mLAN Version 1 devices, where both these components are implemented in the same node.

The Enabler of the mLAN Version 2 architecture, in addition to issuing commands that effect connections, is also responsible for providing a number of high-level abstractions for various mLAN A/M devices. These include abstractions for audio and MIDI plugs and also for word clock synchronization. An mLAN Version 2 A/M device exposes a thin interface to a controlling application, which allows the device to be configured for basic transmission and reception of A/M data. This device is referred to as a Transporter and is mentioned briefly in the previous chapter.

This chapter briefly describes the mLAN Version 2 architecture and the corresponding software design and implementation done by Yamaha Corporation for the Windows and Macintosh platforms. It furthermore describes the mLAN Version 2 Linux implementation created as a component of this research. An analysis of the advantages and disadvantages of this approach is provided, with performance testing

results that compare the mLAN Version 2 implementation to that of mLAN Version 1.

5.1 The Plural Node Model

The most important feature of the plural node architecture is the relocation of the implementation of the abstraction of mLAN plugs and word clock synchronization. The implementation of these abstractions no longer resides within the device but rather within a separate node, which is usually an IEEE 1394 equipped workstation. The workstation implements the Enabler that is responsible for implementing the high-level abstractions and enabling connections between mLAN plugs. The device, however, implements an mLAN Node Controller and associated firmware that together comprise the Transporter. A Transporter is responsible for the transport, i.e. the transmission and reception of audio and music data in a manner that is compliant with the Audio and Music data transmission protocol (IEC 61883-6), and relies on an Enabler to set up its A/M data parameters for the transmission and reception of audio and MIDI data. There may be more than one Enabler on an mLAN network, but each Transporter is controlled by only one Enabler. This concept is illustrated in Figure 5-1 below.

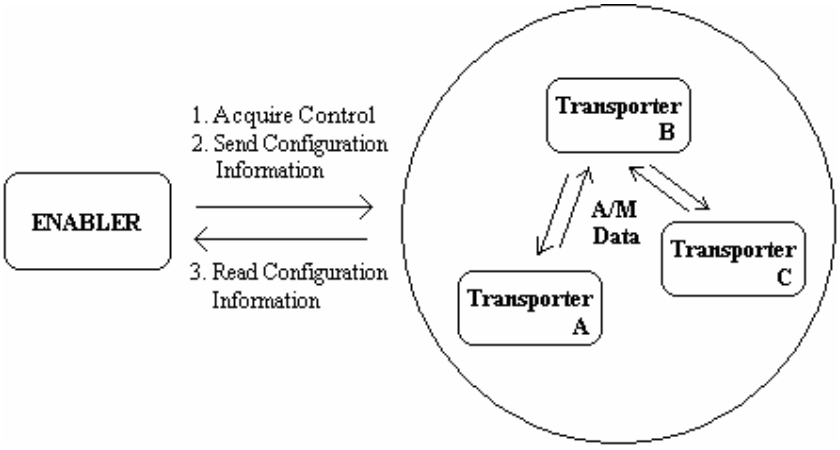


Figure 5-1: Plural node (Enabler-Transporter) interaction

The Enabler is shown to have acquired control over the Transporters labelled A, B and C, after which it is then responsible for retrieving and issuing configuration requests to set up A/M data transmissions. The following sections describe the architecture of the Transporter and Enabler in more detail.

5.1.1 The Transporter Architecture

This section gives an overview of the Transporter architecture, highlighting the basic/required components as well as those which are deemed optional/recommended.

5.1.1.1 Basic Transporter Requirements

The "mLAN-1.0 Transporter Specification" document [Yamaha Corp., 2002b] gives an overview of what a Transporter implementation should comprise. The Transporter in its simplest form consists of the A/M Transporter Layer - an implementation of the IEC 61883-6 protocol, and a simple control interface used as a means of modifying the parameters of the A/M implementation via the IEEE 1394 serial bus. A representation of these layers is given in Figure 5-2.

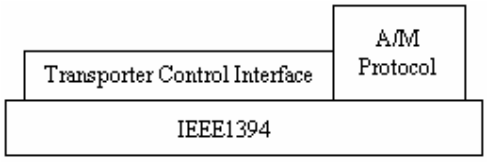


Figure 5-2: Basic components of a Transporter node

IEEE 1394 Layer

The IEEE 1394 layer implements the required IEEE 1394 functionalities defined by the IEEE Std. 1394-1995 [IEEE, 1995] and 1394a-2002 [IEEE, 2000]. They implement the serial bus protocol stack that includes the physical layer, link layer and transaction layer. The protocol stack also includes a serial bus management component, which provides the basic control functions and standard CSRs needed to control nodes or to manage bus resources. The standard CSR registers implemented by Transporters are equivalent to those described in Table 3-1of section 3.1.1.3. A configuration ROM format is also defined. This is detailed in the document "mLAN-2.0 Configuration ROM specification" [Yamaha Corp., 2003b], and builds upon the standard configuration ROM format defined by IEEE 1394. This specification defines, amongst other things, the structure of the unit directory that is required to be implemented by mLAN Transporter nodes. It specifies a number of fields that contain information which identifies an IEEE 1394 node to be of type mLAN Transporter, and also identifies the hardware abstraction layer plug-in to be used by a controlling

application in handling the device. The format of the unit directory is shown in Figure 5-3 below.

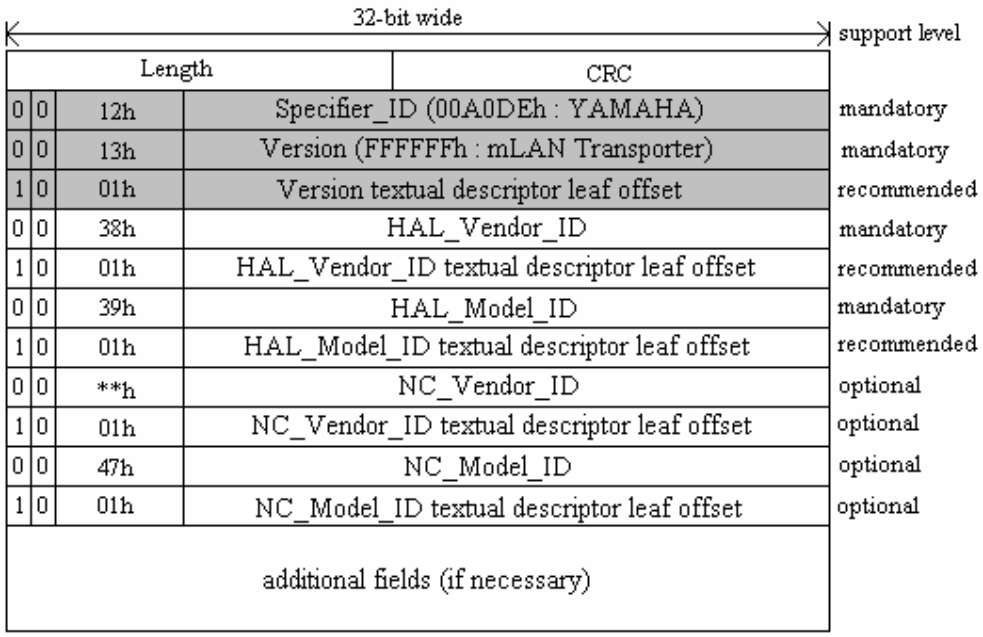


Figure 5-3: Format of the unit directory defined for mLAN Transporter nodes

The software version has the value FFFFFFh. This together with the specifier ID value of 00A0DEh, indicates that a given IEEE 1394 node implements the mLAN Transporter specification. The combination of HAL_Vendor_ID and HAL_Model_ID, collectively known as the HAL ID, is used by a controlling application to uniquely identify the hardware abstraction layer plug-in that is to be used in handling the device. The use of the HAL ID in determining a Transporter's hardware abstraction layer is revisited shortly. The fields NC_Vendor_ID and NC_Model_ID are optional and are not discussed.

Transporter Control Interface

The Transporter Control Interface provides a means for accessing Transporter resources via the serial bus. The main portion of this interface is implemented by mapping control and status registers concerned with the implementation of the A/M protocol, to a portion of the Transporter's address space. This enables a controlling application to read or write to a Transporter node by issuing simple asynchronous transactions, as opposed to the FCP command/response approach adopted for the mLAN Version 1 architecture.

Access to a Transporter's resources is governed by the value of the upper 16-bits of an ENABLER_ADDRESS register. The contents of the ENABLER_ADDRESS register is a 64-bit value that specifies the Transporter's Enabler node (upper 16-bit) and the address of the Enabler's interrupt register (lower 48-bits). Access is granted to a node specified by the upper 16-bits of this register. The lower 48-bits specify the CSR address of the Enabler's interrupt register, which enables the Transporter to send interrupt messages to the Enabler. At start-up, or at bus reset, the ENABLER_ADDRESS register is initialized to 0xFFFF00000000000.

A/M Protocol Layer

The A/M Protocol Layer forms the core of the Transporter's implementation, and implements configuration parameters of the A/M protocol that enables transmission and reception of isochronous streams, configuring isochronous sequences, and setting up SYT synchronization. This layer can be implemented in hardware, in the form of an LSI chip, or as firmware. Examples of hardware implementations include the DICE II chip – manufactured by Wavefront Semiconductor [Wavefront Semiconductor, 2005] and Yamaha's mLAN-PH1, mLAN-PH2 and mLAN-NC1 chips, described in section 3.5. Recall that these chips expose a number of registers that allow for the encapsulation and extraction of sequences. BridgeCo's DM1000 chip [BridgeCo AG, 2005] is an example that allows for software implementations. Software implementations of the A/M protocol also expose configuration parameters that are mapped to a portion of a Transporter's address space.

Transporter Operating Modes

Two types of Transporter modes are defined: *B-mode and B-Pro mode*. A *B-mode* Transporter requires an Enabler in order to start A/M data transmission. *B-Pro mode* Transporters, on the other hand, are able to manage A/M data transmissions in the absence of the Enabler. For a Transporter to be *B-Pro mode* capable, it has to implement Connectionless Isochronous Transmission (CIT), where the node simply broadcasts A/M data on a fixed isochronous channel and receives data on more than one fixed isochronous channels. In addition to these requirements, there also has to be an indication of the isochronous bandwidth required for transmissions. In achieving

this, a CIT Manager layer is implemented as part of the Transporter's basic components as shown in Figure 5-4 below.

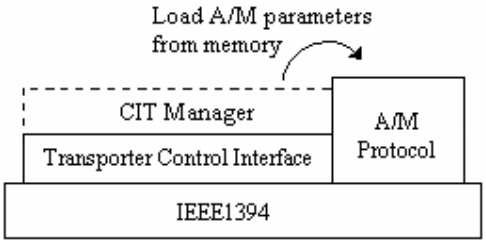


Figure 5-4: Transporter node with a CIT Manager layer

The primary role of the CIT Manager is to load the A/M Transport layer with an entire set of *pre-configured* A/M parameters for Connectionless Isochronous Transmissions. Transporters that implement the CIT manager have the following features:

- Boot Parameter Memory
- A/M Parameter Loading
- Isochronous Resource Allocation

The boot parameter memory refers to non volatile memory used for storing a copy of the A/M configuration parameters of a Transporter. The entries in this memory are specified by an Enabler and are required to be modified following a configuration change to registers of the A/M Transport layer. At power-on or after writes to the RESET_START register (see the core IEEE 1394 CSR registers shown in Table 3-1), the CIT manager is required to copy the parameters from the Boot Parameter Memory to the Transporter's A/M Transport Layer. This ensures that the Transporter device starts up in a state equivalent to the last stable configuration prior to power down.

In addition to A/M parameter loading, isochronous resources (channel and bandwidth) have to be reallocated after each bus reset. The isochronous channel and bandwidth that are to be reallocated by the Transporter for each of its transmitting isochronous streams are specified by the Enabler and held within non volatile memory registers within the CIT manager layer. The bandwidth calculated for an isochronous stream is derived using the following formula:

$$bandwidth = overhead + (packetOverhead + payload) \times (1600 / dataRate) \qquad (4)$$

The *overhead* field refers to the extra overhead required to transmit the isochronous stream. The value of this field is dependent on the number of cable hops from the transmitting to the receiving device. The IEC 61883-1 specification [IEC, 2003b] gives details of the various overhead values. The overhead value of 352, which assumes the maximum cable hop count of 16, is used as default in most cases.

The *packetOverhead* field has the value 5, and is made up of the sum of the quadlet sizes of the *isoch header*, *CIP header* and *isoch CRC* quadlets of an isochronous packet containing CIP formatted data. From Figure 3-31, this corresponds to 2, 2 and 1 respectively.

The *payload* field is the product of the number of sequences transmitted by the isochronous stream and the maximum data blocks contained within each isochronous packet, i.e.:

$$payload = numsequences \times blocks \qquad (5)$$

The IEC 61883-1 specification [IEC, 2003b] gives details on the maximum data block values.

The *dataRate* field corresponds to the transmission speed of the isochronous stream in Mb/s, i.e. 100, 200, 400, 800 or 1600.

If the channel or bandwidth resource cannot be allocated within the first one second following a bus reset, the Transporter device aborts any ongoing isochronous transmissions. Also, if the bandwidth register for an isochronous stream has the value zero, the isochronous channel in the corresponding channel register is not allocated by the Transporter.

Optional components for Transporter implementations are also specified by the mLAN Transporter specification. These are discussed in the next paragraph.

5.1.1.2 Optional (Recommended) Components

A number of hardware interfaces and extensions can be implemented within Transporter devices. The interfaces include an audio interface, a MIDI interface, an RS232/422 serial interface and a node application interface. The audio and MIDI interfaces are dependent on a particular host. The serial interface and node application interface can be optional implementations. The node application interface provides the host system of the Transporter with inbound/outbound asynchronous packet bridging services, thus allowing access to the properties of a Transporter's host application.

Other hardware extensions include the implementation of an LED indicator, which is used to indicate the connection state of the PHY ports of an IEEE 1394 node. The mLAN Transporter specification defines the colours green and red to indicate a port connected to a leaf node and non-leaf node respectively, the colour green to indicate a root node and the colour blue to indicate an active PHY port. These LED indicators can also be used in identifying the mLAN Transporter by configuring it to blink several times.

Another recommended, but optional Transporter component is the implementation of a PRIVATE_SPACE_MAP table at address 0xFFFFE0000000 of the node's private space. Recall from Figure 3-3 on page 43 that the private space forms part of a node's 256 terabyte address space and is reserved for the node's local use. The PRIVATE_SPACE_MAP table specifies the mappings to those areas within the private space that implement the Transporter Control Interface to the various resources implemented by the Transporter. The PRIVATE_SPACE_MAP defined by the mLAN Transporter specification is shown in Table 5-1. All entries are read quadlet or read block accessible, with offsets relative to the top of the private space (0xFFFFE0000000).

Offset	Name	Length	Description
+000	BOOT_PAR_SPACE_OFFSET	32bits	Boot parameter area offset
+004	BOOT_PAR_SPACE_SIZE	32 bits	Boot parameter area size
+008	CORE_SPACE_OFFSET	32 bits	Core area offset

+00C	CORE_SPACE_SIZE	32 bits	Core area size
+010	MLAN_SPACE_OFFSET	32 bits	mLAN register area offset
+014	MLAN_SPACE_SIZE	32 bits	mLAN register area size
+018	NODE_APP_SPACE_OFFSET	32 bits	Node application area offset
+01C	NODE_APP_SPACE_SIZE	32 bits	Node application area size

Table 5-1: PRIVATE_SPACE_MAP table defined by the mLAN Transporter specification

The boot parameter space offset and size provides access to the contents of the boot parameter memory implemented by the CIT Manager discussed earlier. The core space and size provides access to the A/M parameters implemented by the A/M protocol layer of the Transporter. The mLAN space implements registers that provide overall control over the Transporter device. Table 5-2 summarizes the registers implemented by the mLAN space.

Offset	**Name**	**Size**	**Description**
+000	ENABLER_ADDRESS	64 bits	Specifies the node ID and the 48-bit address of an Enabler's interrupt register.
+00C	BANDWIDTH_REQUEST (N)	16 bits	Specifies the amount of bandwidth that should be allocated for a particular isochronous stream, N.
…	CHANNEL_REQUEST (N)	16 bits	Specifies the isochronous channel that should be allocated for a particular isochronous stream, N.
	SERIAL_PORT_DATA_ADDR	64 bits	Specifies the 64-bit address to which data received on a specified serial port should be relayed to.
	SERIAL_PORT_DATA_OUT	32 bits	Data written to this register will be output to a preconfigured serial port such as RS-232 or RS-422 for external device control.
	IDENTIFY	16 bits	Invokes an "identity operation"

			on a Transporter when a non-zero value is written to this register.

Table 5-2: Registers implemented by the mLAN space

All entries are read quadlet or read block accessible, with offsets relative to the top of the mLAN space specified by MLAN_SPACE_OFFSET. The node application space offset and size specifies the implementation of the node application interface of the Transporter, which allows access to a Transporter's host. In this way, a controller can model the host of a Transporter node and hence get access to the end plugs implemented by the device. Recall that providing control over the host features of a device forms the basis of this research, and the techniques employed are discussed more extensively in chapters 6 and 7.

The Transporter architecture allows other vendor-specific implementations to be implemented within the Transporter device. However, these features have to be made known to a controlling application or an Enabler in order for them to be used. The plural node architecture, as compared with the mLAN Version 1 model, provides a flexible, easy, and dynamic way for different vendors to manufacture Transporter devices with proprietary implementations. These vendors have to provide a 'plug-in' module that interfaces to an Enabler, which enables it to enumerate the device and allow for connection management through plug abstractions. The next subsection describes the basic requirements of an mLAN Version 2 Enabler that is required to interact with Transporters.

5.1.2 The Enabler Architecture

The role of the mLAN Version 2 Enabler differs significantly from that of mLAN Version 1. In addition to facilitating connection management, it is also responsible for providing plug and word clock abstractions based on the low-level A/M implementation of Transporter devices. These Enabler-abstracted plug and word clock models are presented to a client application, such as a patch bay, from where point-to-point plug connections and also word clock synchronization can be achieved.

The Enabler allows a number of different types of plug abstractions to be implemented on behalf of Transporter devices. Plugs of the most basic plug abstraction are referred to as *mLAN plugs*. These plugs are directly related to isochronous sequences or subsequences of an isochronous stream that is transported by the A/M layer of a Transporter node. A further discussion on these plugs is given shortly. Above this plug layer, other plug layers such AV/C plugs, or plugs of a Transporter's host can be implemented. In addition to this plug abstractions layers, the architecture of the mLAN Version 2 Enabler also makes it possible to provide interoperability between 1394-based A/V devices that implement different connection management procedures. For example, an Enabler of mLAN Version 2 can enable plug connects and disconnects between mLAN Version 1 and mLAN Version 2 devices or between BridgeCo's BeBob break-out boxes [BridgeCo AG, 2005] that implement the AV/C Music Subunit specification.

The Enabler, according to this model, has to implement at least three distinct layers that facilitate uniform interaction with Transporter devices and enable plug abstractions [Fujimori and Foss, 2003]. These layers include the:

- mLAN Plug Abstraction Layer
- A/M Manager Layer, and
- Hardware Abstraction Layer

These layers are described in the subsections that follow. Note that other plug abstractions layers are optional implementations of the Enabler.

5.1.2.1 The mLAN Plug Abstraction Layer

The top layer, the mLAN plug abstraction layer, implements a number of mLAN plugs on behalf of Transporter devices. These plugs can be viewed as terminators for the transmission and reception of audio sequences or MIDI subsequences that are carried by an IEC 61883-6 formatted isochronous stream. Figure 5-5 highlights the mLAN plug concept that makes use of the isochronous stream model shown in Figure 3-29.

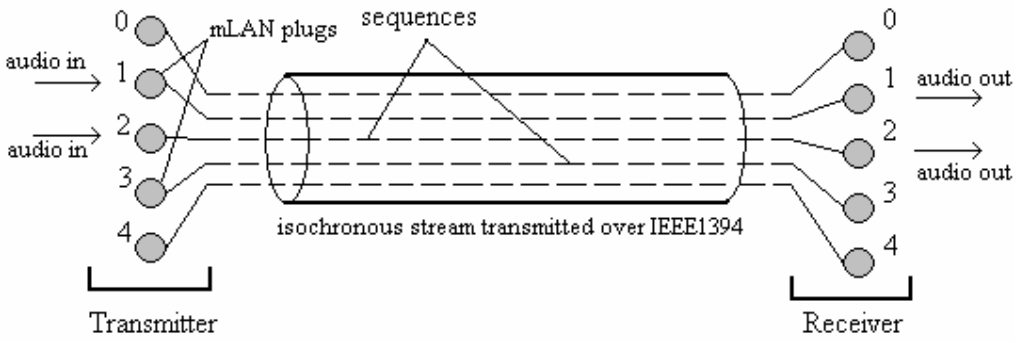


Figure 5-5: Concept of mLAN plugs implemented by the Enabler

The mLAN plugs are shown in grey and directly correspond to the audio sequences carried by the isochronous stream. On the transmitter side, they can be viewed as a means of encapsulating audio feeds into isochronous sequences to be transmitted over the IEEE 1394 bus. Conversely on reception, they can be viewed as a means of extracting audio data from isochronous sequences transmitted over the IEEE 1394 bus.

Word clock synchronization is also modelled by this layer. This is illustrated in the figure below.

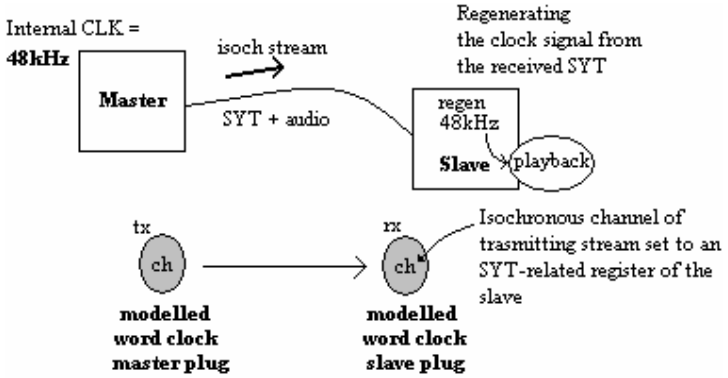


Figure 5-6: Modelling the word clock of Transporter devices

In this case, the device that is to act as a word clock slave to a corresponding master has to be configured to receive SYT timing information from the isochronous stream transmitted by the master. Recall from Figure 3-31 that the SYT timing information is encapsulated within the CIP header of an isochronous packet, and gives an indication of the presentation time of the events contained within the packet structure.

5.1.2.2 The A/M Manager Layer

Below the top layer, is an A/M manager layer. This layer is responsible for reading audio and music data transmission and reception parameters from the associated Transporters, and for updating these parameters in response to requests from the mLAN plug abstraction layer. Each of the Transporters under the control of the Enabler will have a Transporter object that keeps its state information and handles requests from both the plug abstraction layer and the actual Transporter.

5.1.2.3 The Hardware Abstraction Layer

The Hardware Abstraction Layer (HAL) is the bottom layer of the Enabler. This layer is responsible for abstracting away the details of various hardware implementations of Transporters. This is achieved by defining and exposing a uniform interface, which serves as the common means of communication between the Enabler and a vendor-specific Transporter HAL implementation. Recall from section 5.1.1.1 that a vendor would be required to provide a 'plug-in' module (that interfaces to an Enabler) for its proprietary Transporter; the plug-in module will implement the HAL interface defined by the Enabler, and translate method calls of this interface into proprietary requests that the Transporter Control Interface of that particular Transporter can interpret.

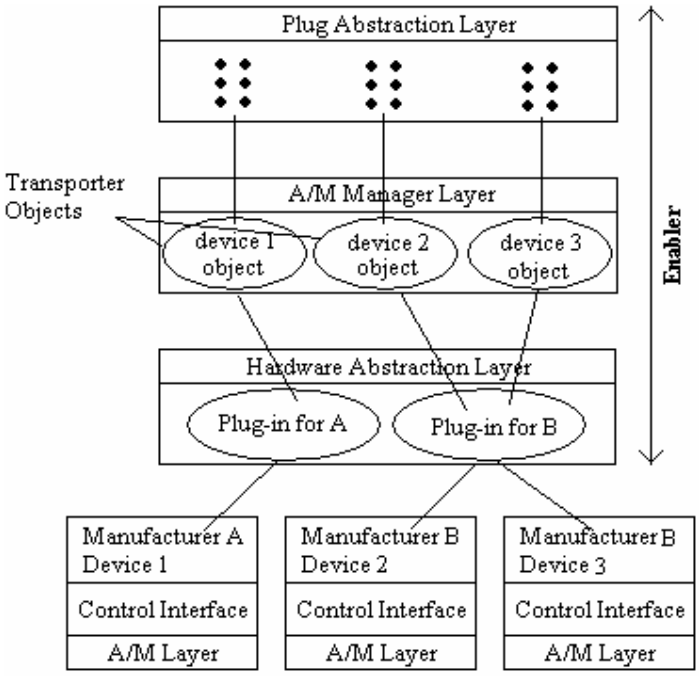


Figure 5-7: Enabler's layers and interfaces

The operation of the 3 layers of the Enabler is illustrated in Figure 5-7 above. Three Transporters are shown at the bottom of the figure, one that contains a node controller implementation from manufacturer A, and two that contain a node controller implementation from manufacturer B. Both manufacturers A and B have created workstation plug-ins that implement their respective messaging protocols to communicate with the Transporter Control Interface implemented within the devices.

The A/M Manager in Figure 5-7 has instantiated three Transporter objects that provide A/M related control over and access to their respective hardware Transporters. The plug-ins are responsible for fulfilling these control and access requests. Via interaction with these Transporter objects, the mLAN plug abstraction layer can then build mLAN plug objects, which in turn are accessible to applications.

Yamaha Corporation developed an Enabler specification for the plural node architecture. This specification, which was to serve as an example Enabler, is derived from that originally defined to be the mLAN Enabler specification (discussed in section 4.2 of the previous chapter) and is intended to provide connection management for only Transporter devices. This 'limited' version of the Enabler is referred to as the "Basic Enabler" and is described by the "mLAN Basic Enabler Specification, version 1.0.1" [Yamaha Corp., 2004a]. The next section describes the Basic Enabler specification from which the implementation of the subsequent Windows, Macintosh and Linux mLAN Version 2 Enablers are based.

5.2 The Basic Enabler Specification

The object model defined by the Basic Enabler specification is similar to that of the original Enabler (discussed in section 4.2 of the previous chapter) and only differs in the design and hence implementation of an mLAN device. Figure 5-8 shows the new mLAN device object model defined by the Basic Enabler specification.

What is immediately apparent from the object model is the absence of the *CmLANPigeon* and *CmLANSparrow* classes. Also, the Transporter device is modelled directly to be an mLAN device and is not governed by the *CmLANPC* class

implementation as previously indicated by the original Enabler object model. See Figure 4-5.

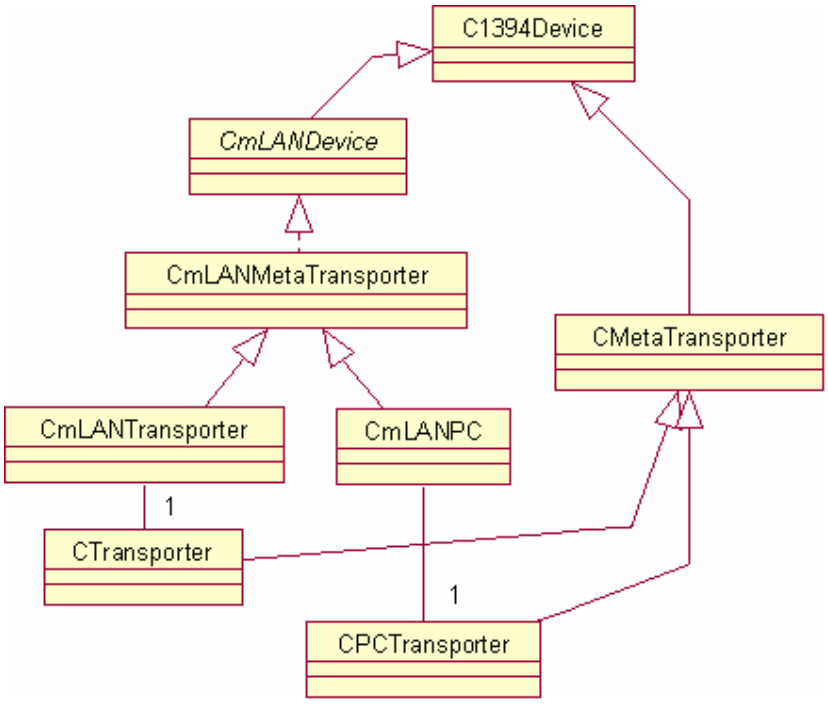


Figure 5-8: mLAN device object model defined by the Basic Enabler specification

The implementation of the mLAN device class – *CmLANDevice*, is now realized by the classes *CmLANTransporter* and *CmLANPC*, which respectively represent an mLAN device implemented by a stand-alone Transporter device and an mLAN device implemented by an IEC 61883-6 PC streaming driver. The *CmLANMetaTransporter* class abstracts and implements the functionality that is common to both mLAN Transporter implementations. This includes the plug abstraction layer responsible for abstracting mLAN plugs from sequences and subsequences within isochronous streams, and the A/M manager layer that provides common operations to modify the A/M protocol layer of both the *CmLANTransporter* and *CmLANPC* Transporter implementations. The plug abstraction layer will be revisited shortly.

The classes *CmLANTransporter* and *CmLANPC*, both have an association with a Transporter object, which is used in directly accessing the low-level implementation of a particular Transporter device. Using this object, the Enabler is able to configure a Transporter device appropriately for transmissions and receptions. The three classes:

CMetaTransporter, *CTransporter* and *CPCTransporter* together form the Enabler's hardware abstraction layer. The *CMetaTransporter* class defines the attributes and methods that are common to devices that support transmission and reception of isochronous streams via the A/M protocol, whereas *CTransporter* and *CPCTransporter* define additional functionality that is specific to either a stand-alone Transporter implementation or a PC Transporter implementation.

The methods defined by the *CTransporter* class, some of which are derived from *CMetaTransporter*, define the HAL interface that provides the uniform means of interaction between the Enabler and various vendor Transporter implementations. An overview of the HAL interface methods defined by the Basic Enabler specification is given in the next subsection, section 5.2.2. These methods are required to be implemented as a plug-in module by a particular Transporter vendor. These vendor-specific method implementations are used by the Enabler to communicate appropriately with the vendor Transporter device. Hence, the implementation of the *CTransporter* class shown to be associated with *CmLANTransporter* is provided by a particular vendor Transporter implementation.

The Basic Enabler specification does not strictly define the implementation of the plug-in module, as this is platform specific. It does however recommend the implementation of a method that identifies Transporter devices that can be communicated to, by matching the supported HAL ID of the plug-in with that of the Transporter devices. Recall from section 5.1.1.1 that the HAL ID identifies the hardware abstraction layer to be used by a controlling application in interacting with a particular Transporter. Methods that create and dispose of vendor Transporter objects are also recommended. These methods are indicated in Figure 5-9 below.

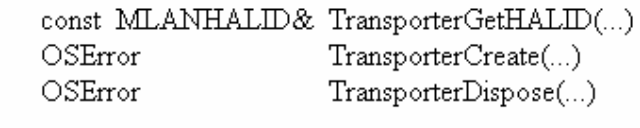


Figure 5-9: Recommended Transporter plug-in methods

The plug abstraction layer and the Transporter HAL interface, defined by the Basic Enabler, are described more extensively in the next two subsections. The plug abstraction layer subsection describes how device plugs are represented by the Enabler and presented to applications from where plug connections and disconnections can be established. The Transporter HAL interface subsection describes how isochronous sequences and subsequences, contained within isochronous streams, are represented and identified by the HAL from which mLAN plugs can be created.

5.2.1 The Plug Abstraction Model

The Basic Enabler specification defines a number of plug classes, one for each mLAN device class: *CmLANDevice*, *CmLANMetaTransporter*, *CmLANTransporter* and *CmLANPC* that implement mLAN plug functionality for the respective device type. A hierarchal relationship exists between these plug classes. The methods of the topmost class of this hierarchy are defined to be abstract, with the implementation given by an appropriate subclass. These methods define the plug abstraction interface exposed to applications and allow access to plug information. Figure 5-10 shows the plug abstraction model defined by the Basic Enabler specification.

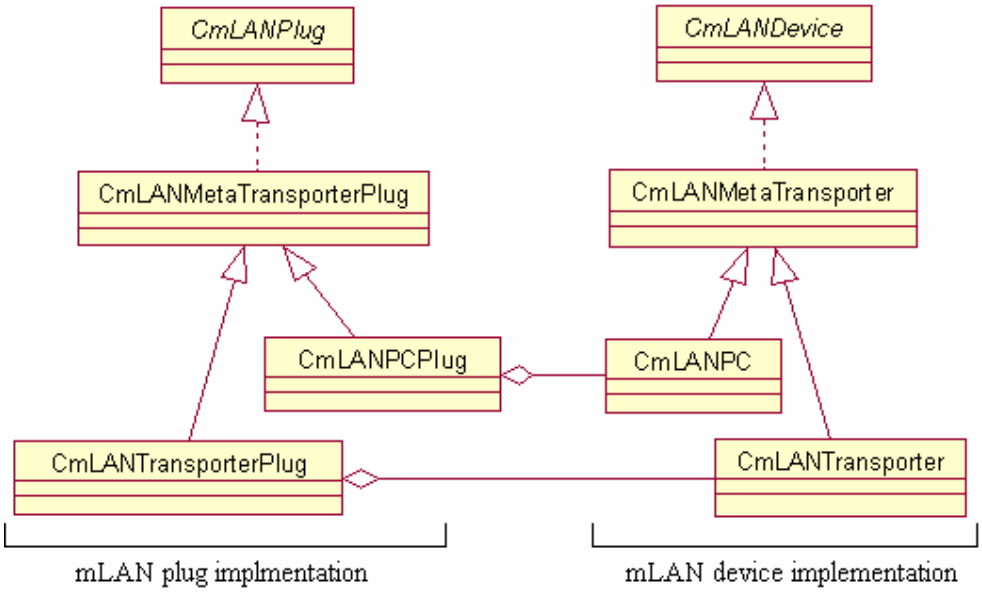


Figure 5-10: Plug abstraction model of the Basic Enabler specification

The right-hand side of Figure 5-10 shows a section of the mLAN device object model discussed in Figure 5-8. The left-hand side shows the corresponding mLAN plug implementation for each device class. The *CmLANPlug* class, the topmost class of the hierarchy, is defined to be abstract with its methods realized by the classes *CmLANTransporterPlug* or *CmLANPCPlug*. The *CmLANTransporterPlug* class models the mLAN plug behaviour of stand-alone Transporter devices, whereas *CmLANPCPlug* models those for a PC Transporter. Figure 5-11 summarizes the methods that make up the plug abstraction interface. The parameter types of the various methods are defined by the Basic Enabler specification.

Public Methods

0 IsDestination (void)
1 IsSource (void)
2 GetPlugType (void)
3 GetPlugID (void)
4
5 GetPlugName (UInt8 *strName)
6 SetPlugName (const UInt8 *strName)
7
8 GetSampleRateList (UInt32 listSize, UInt32 *aSampleRate, UInt32 *pNumSampleRates)
9 GetDataTypeList (UInt32 listSize, UInt32 *aDataType, UInt32 *pNumDataTypes)
10 GetDataType (UInt32 *pDataType)
11 GetSampleRate (UInt32 *pSampleRate)
12 SetSample Rate (UInt32 sampleRate)
13
14 GetConnections (MLANPlugConnectionsPtr pConnections)
15 DisconnectAll (void)

Private Methods

16 GetIsochChannel (UInt32 *pIsochChannel)
17 GetSequenceNumber (UInt32 *pSeqNumber)
18 GetSubsequenceNumber (UInt32 *pSubNumber)
19
20 Connect (CmLANPlugPtr pMLANPlug)
21 Disconnect (CmLANPlugPtr pMLANPlug)

Figure 5-11: Summary of the plug abstraction interface methods

The public methods (accessible by applications) allow access to various properties of a plug, which include the plug ID, the plug direction – output or input, the plug type – Audio or MIDI, the data type carried by the plug – IEC 60958/Raw Audio/MIDI/SYT Clock, the plug name, the target connections to a plug, and the sample rate in use by the plug. Methods are also defined, on lines 8 and 9, to retrieve a list of sampling rates and data types supported by the plug. A client application is also capable of

performing operations to set the plug name and sample rate, as well as to disconnect all target connections to a plug.

Private methods are used internally within the class to establish and break connections to other plugs. These are implemented by corresponding *Connect/Disconnect* method calls of the *CmLANDevice* class. Note that an application using a Basic Enabler, enumerates devices on a network in a manner similar to when using an mLAN Version 1 Enabler. In this case, *objectIDs* for interfaces, buses or devices are retrieved from the Enabler and the corresponding object created by the application. This routine has been discussed in section 4.2 using the object model illustration shown in Figure 4-6.

The mLAN plugs that are modelled by the Enabler on behalf of Transporter devices have to correspond to sequences or subsequences contained within the transmitted or received isochronous streams of a Transporter. The next subsection describes the Transporter HAL interface defined by the Basic Enabler, and gives and indication of how these sequences and subsequences can be identified.

5.2.2 The Transporter HAL Interface

As mentioned earlier, the Basic Enabler's Transporter HAL interface incorporates methods of the *CTransporter* class of the mLAN device object model (Figure 5-8). This class represents stand-alone Transporter devices, which can be implemented according to a number of different vendor Transporter specifications. The methods defined by this class are shown below in Figure 5-12. Note that the parameters of some of the methods have been omitted for brevity.

Methods introduced by the *CMetaTransporter* class

```
0     virtual OSError    GetNameString (UInt32 strType, UInt8 *strName);
1     virtual OSError    SetNameString (UInt32 strType, const UInt8 *strName);
2     virtual OSError    GetMaxSpeed (UInt32 *pSpeed);
3     virtual OSError    GetMaxIsochChannels (bool isInput, UInt32 *pNumIsochChannels);
4     virtual OSError    IsRunning (bool isInput, bool *pRunning);
5     virtual OSError    Start (bool isInput);
6     virtual OSError    Stop (bool isInput);
7     virtual OSError    GetSYTSynchChannel (UInt32 *pIsochChannel);
8     virtual OSError    SetSYTSynchChannel (UInt32 isochChannel);
9     virtual OSError    GetSYTSynchStatus(bool *pSynch)
10
```

```
11  virtual OSError   GetOutputSpeed (UInt32 isochID, UInt32 *pSpeed);
12  virtual OSError   SetOutputSpeed (UInt32 isochID, UInt32 speed);
13  virtual OSError   GetOutputMaxDataBlocks (UInt32 isochID, UInt32 *pBlocks);
14  virtual OSError   SetOutputMaxDataBlocks (UInt32 isochID, UInt32 blocks);
15  virtual OSError   GetOutputOverhead (UInt32 isochID, UInt32 *pOverhead);
16  virtual OSError   SetOutputOverhead (UInt32 isochID, UInt32 overhead);
17
18  virtual OSError   GetIsochChannel (bool isInput, UInt32 isochID, UInt32 *pIsochChannel);
19  virtual OSError   SetIsochChannel (bool isInput, UInt32 isochID, UInt32 isochChannel);
20
21  virtual OSError   GetEventTypeList (bool isInput, UInt32 isochID, ...);
22  virtual OSError   GetSFCList (bool isInput, UInt32 isochID, ...);
23  virtual OSError   GetEventType (bool isInput, UInt32 isochID, UInt32 *pEventType);
24  virtual OSError   SetEventType (bool isInput, UInt32 isochID, UInt32 eventType);
25  virtual OSError   GetSFC (bool isInput, UInt32 isochID, UInt32 *pSFC);
26  virtual OSError   SetSFC (bool isInput, UInt32 isochID, UInt32 sFC);
27
28  virtual OSError   GetSequenceTypeList (bool isInput, UInt32 isochID, ...);
29  virtual OSError   GetMaxSequences (bool isInput, UInt32 isochID, UInt32 seqType, …);
30  virtual OSError   GetNumSequences (bool isInput, UInt32 isochID, UInt32 seqType, …);
31  virtual OSError   SetNumSequences(bool isInput, UInt32 isochID, UInt32 seqType, …);
32
33  virtual OSError   GetSequenceDataTypeList (isInput, isochID, seqType, seqID,…);
34  virtual OSError   GetSequenceDataType (isInput, isochID, seqType, seqID,…);
35  virtual OSError   SetSequenceDataType (isInput, isochID, seqType, seqID,…);
36  virtual OSError   GetSequenceNumber (isInput, isochID, seqType, seqID,…);
37  virtual OSError   SetSequenceNumber (isInput, isochID, seqType, seqID,…);
38  virtual OSError   GetMaxSubsequences (isInput, isochID, seqType, seqID,…);
39  virtual OSError   GetNumSubsequences (isInput, isochID, seqType, seqID,…);
40  virtual OSError   SetNumSubsequences (isInput, isochID, seqType, seqID,…);
44
45  virtual OSError   GetSubsequenceNumber (isInput, isochID, seqType, seqID, subID, …);
46  virtual OSError   SetSubsequenceNumber (isInput, isochID, seqType, seqID, subID, …);
```

Methods introduced by the *CTransporter* class

```
47  virtual OSError   SetEnablerAddress ( void);
48  virtual OSError   ClearEnablerAddress (void );
49  virtual OSError   IsUnderControl (void );
50
51  virtual OSError   Identify (UInt16 data);
52  virtual OSError   GetIdentificationStatus (bool *pData.);
53
54  virtual OSError   InterruptSetNotify (void (*callback)(...), void *pParam);
55  virtual OSError   InterruptClearNotify (void (*callback)(...), void *pParam);
56  virtual OSError   InputPortSetNotify (UInt32 portID, void (*callback)(...), void *pParam);
57  virtual OSError   InputPortClearNotify (UInt32 portID, void (*callback)(...), void *pParam);
58  virtual OSError   OutputPortSend (UInt32 portID, UInt32 *pCount, const UInt8 *aData);
59
60  virtual OSError   GetTransporterMode (UInt32 *pMode);
61  virtual OSError   SetTransporterMode (UInt32 mode);
62
63  virtual OSError   CallPlugInControlPanel (PlugInCPParamPtr pParam = nil);
64  virtual OSError   CallPlugInSpecificFunction (UInt32 funcID, void *pParam, UInt32 paramSize);
```

Figure 5-12: Summary of the Transporter HAL interface methods

The methods introduced by each class (*CMetaTransporter* and *CTransporter*) are discussed the subsections below.

5.2.2.1 The '*CMetaTransporter* class' Interface

The methods defined by the *CMetaTransporter* class are also common to *CPCTransporter*. These methods provide access to the parameters of a Transporter's A/M implementation, allowing a controller to retrieve and modify the IEC 61883-6 transport capabilities of the Transporter via its control interface. These methods are explained below according to the common entities that they provide access to.

Common Entity: The Overall Transporter

The methods that address the overall Transporter are described by lines 0 through to 9 of Figure 5-12. These include methods to get and set the various types of names (nickname, version, module and vendor) of the Transporter, methods to get the maximum handling speed as well as the maximum number of input or output isochronous channels/streams supported by the Transporter. The stream direction (input or output) is specified by the *isInput* parameter. For a particular stream direction, methods are defined to check the streaming state, given by the *IsRunning(…)* method, and to stop or start all streams of a particular direction, given by the *Stop(…)* and *Start(…)* methods respectively. The methods *GetSYTSynchChannel(…), SetSYTSynchChannel(…)* and *GetSYTSynchStatus(…)* are used respectively to retrieve the SYT channel on which the Transporter receives SYT synchronization information, set the SYT channel on which to receive SYT synchronization information, and to check the status of SYT receptions.

Common Entity: An Isochronous Stream

The methods that address the parameters of either the transmission or reception FIFO buffer of an isochronous stream of a Transporter are described by lines 11 through to 31 of Figure 5-12. An isochronous stream FIFO buffer for a particular direction is identified using an *isochID* parameter, which has a value ranging from 0 to the value of **pNumIsochChannel* returned from the call to the *GetMaxIsochChannels(bool isInput, UInt32 *pNumIsochChannels)* method. This range is illustrated using the inequality expression stated below.

$$0 \leq isochID < *pNumIsochChannels \quad (6)$$

Recall from section 3.3.2 and from section 5.1.1 (page 127) that an isochronous stream has properties such as the:

- transmission speed, maximum data blocks and overhead,
- isochronous channel number,
- supported event types and sample rates, including the event type and sample rate in use by the isochronous stream (see Table 3-6),
- types of sequences (audio/MIDI or a combination both) and the number of each type carried by the isochronous stream – see Figure 3-29 and Figure 3-30 of section 3.3.1.

The methods defined for this common entity allow access to the corresponding buffer parameters, and where possible allow these parameters to be configured. With regards to connection management, the *GetIsochChannel(…)* and the *SetIsochChannel(…)* methods can be used to configure a certain input isochronous stream buffer to receive a particular incoming isochronous stream. Also, the *GetNumSequences(…)* method is used by the Enabler's plug abstraction layer during plug enumeration. These concepts will be revisited in the next section.

Common Entity: An Isochronous Sequence

An isochronous sequence of a particular sequence type, contained within a stream of a particular stream direction, is identified using a *seqID* parameter. This parameter has a value ranging from 0 to the value of **pNumSequences* returned from the call to the *GetNumSequences(bool isInput, UInt32 isochID, UInt32 seqType, UInt32 *pNumSequences)* method. This range is illustrated using the expression stated below.

$$0 \leq seqID < *pNumSequences \quad (7)$$

The other parameters: *isInput*, *isochID* and *seqType*, of the *GetNumSequences(…)* method are used respectively to indicate the direction, identify the isochronous stream FIFO buffer and the sequence type of the enumerated sequences. The methods that describe the parameters of an isochronous sequence are indicated by lines 33 to 40 of

Figure 5-12. These include methods to get and set the transmission or reception sequence data type (IEC 60958, Raw Audio or MIDI), the sequence number or position at which a sequence is to transmit or receive, and the number of subsequences (if any) implemented by a sequence. These methods are used frequently by the plug abstraction layer of the Enabler in plug enumeration and also during plug connections.

Common Entity: An Isochronous Subsequence

The properties defined for isochronous subsequences only include methods to get and set the subsequence number on which an isochronous subsequence can transmit or receive. This functionality is provided by the *GetSubsequence(…)* and *SetSubsequence(…)* methods (lines 45 and 46 of Figure 5-12). A subsequence is identified using a *subID* parameter, which is also governed by the direction of the isochronous stream containing the sequence of which the subsequence forms part. The value of the *subID* parameter lies in the range from 0 to the value of **pNumSubsequences* returned from the call to the *GetNumSubsequences(…, UInt32 *pNumSubsequences)* method, as illustrated in the expression stated below.

$$0 \leq subID < *pNumSubsequences \qquad (8)$$

The *GetSubsequence(…)* and *SetSubsequence(…)* methods are also used by the Enabler's plug abstraction layer during plug connections.

5.2.2.2 The '*CTransporter* class' Interface

The methods introduced by the *CTransporter* class provide access to the parameters of the recommended Transporter components discussed in section 5.1.1.2. The methods described by lines 47 to 58 access the various fields defined by the Transporter's mLAN Space (see Table 5-2).

Using these methods the Enabler can set or clear the ENABLER_ADDRESS register of a Transporter. Recall that a Transporter only responds to request messages from a controller that has its node ID set to the upper 16-bits of its ENABLER_ADDRESS register. The method *IsUnderControl(…)* can be used to determine whether the

Enabler has control over a Transporter. The Transporter can also be identified appropriately using the *Identify(…)* method. Interrupt messages or serial bus data generated from the Transporter are detected and handled by the Enabler using call-back routines specified via the *InterruptSetNotify(…)* and *InputPortSetNotify(…)* methods. Likewise, if supported, a Transporter can receive a series of byte data from the Enabler, and can relay this data to an output port.

The operating mode of the Transporter (*B-mode* or *B-Pro mode)* can be retrieved and specified using the *GetTransporterMode(…)* and *SetTransporterMode(…)* methods, respectively. Recall that these operating modes indicate whether or not the Transporter requires an Enabler in order for it to start its A/M data transmissions. A control panel dialog is required to be implemented by the vendor of a particular Transporter implementation, allowing the various Transporter parameters to be easily viewed and modified. The *CallPlugInControlPanel(…)* method is used to display the control panel dialog.

The Basic Enabler specification defines a congruent model from which software developers can build Enablers. The next section takes a look at the Enabler implementation done by Yamaha Corporation for the Windows and Macintosh platforms, and the Linux Basic Enabler implementation performed as part of this research. The particular implementation strategies of the Linux equivalent are also discussed.

5.3 Plural Node Implementation

Before describing the software implementation, a brief description of the Transporter hardware used in the Enabler-Transporter implementation is given below.

5.3.1 Transporter Hardware

The first Transporter hardware implementation, the MAP4 Evaluation board, was developed by Yamaha Corporation. This was developed to test mLAN connection management using the plural node approach described above, and also, to serve as a prototype on which other Transporter implementations are based. The MAP4 consists

of two boards – a daughter board and a mother board. The daughter board, known as the MAP4D, contains:

- an mLAN-NC1 chip (discussed in section 3.5),
- an mLAN-PH2 chip (discussed in section 3.5) and
- a 2-port S400 PHY.

The mother board, the MAP4M, contains:

- 4 A/D converters allowing for 4 input channels of audio,
- 8 D/A converters allowing for 8 output channels of audio,
- 4 MIDI in ports and
- 4 MIDI out ports.

The mother board acts as the typical professional audio device, with the daughter board providing the node controller (Transporter) capabilities. The link layer capability resides in the mLAN-NC1 chip, and is used by both the NC1 and the PH2. Of these two chips, the NC1 is configured to handle the MIDI processing, while the PH2 handles all the audio processing for the node controller. A diagram of the MAP4 board layout is given in appendix A [Yamaha Corp., 2002c].

A layout of the address space (Transporter Control Interface) allocated to the node controller (Transporter) implementation of the MAP4 is shown below in Figure 5-13 (adapted from the document "NC1-Transporter Address Map, Version 0.500B" [Yamaha Corp., 2002d]).

The *PRIVATE_SPACE_MAP* implements the private space map entries defined in Table 5-1.

The *Boot Parameter Space* refers to the non volatile memory used for storing a copy of the A/M transmission parameters of the MAP4. This indicates that the MAP4 implements a CIT Manager and hence is capable of starting transmissions without an Enabler. The structure of the Boot Parameter Space is as shown in Figure 5-14. This contains a copy of the configuration ROM and an initialization table that contains the A/M transmission initialization entries.

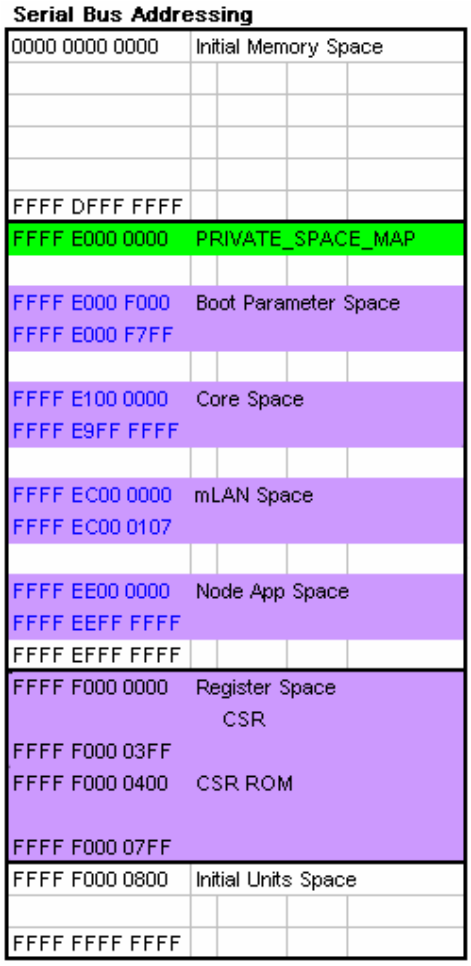


Figure 5-13: Node address space (Transporter Control Interface) defined for the MAP4

Boot Parameter Addressing	Size	Offset
Reserved for Transpoter use	256 bytes	+000
CSR ROM Image	512 bytes	+100
Init Table	1280 bytes	+300

Figure 5-14: MAP4's boot parameter space

Within the initialization table (shown as *Init Table*), 3 entry types are defined (NCP04/05 Transporter Specifications [Yamaha Corp., 2002e]): *block parameter entry*, *initialization entry*, and the *terminator*.

The format of a block parameter entry is of the form:

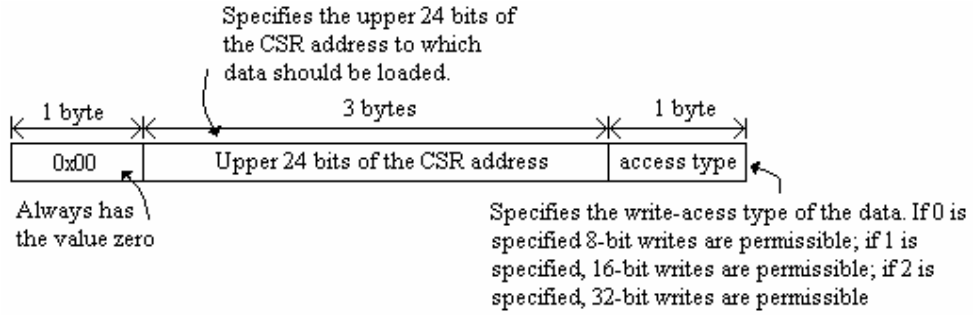


Figure 5-15: Format of a block parameter entry

The format of an initialization entry is of the form:

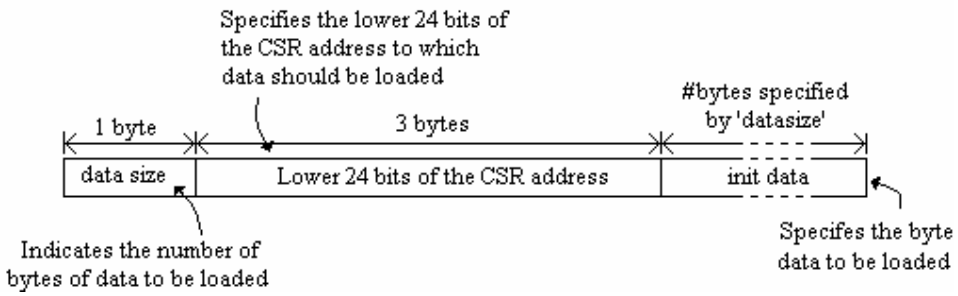


Figure 5-16: Format of an initialization entry

The format of the terminator is of the form:

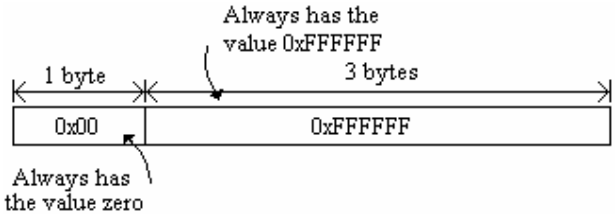


Figure 5-17: Format of a terminator

The block parameter entry specifies the base address and the permissible access type for the initialization entries that follow. The terminator is used to signal the end of the initialization table. Refer to the NCP04/05 Transporter Specifications [Yamaha Corp., 2002e] for more detailed information.

The C*ore Space* of the MAP4's address space, contains amongst other things, direct mappings to the registers of the mLAN-NC1 and mLAN-PH2 chips. This exposes the

registers of the chips, thus allowing them to be configured appropriately for transmission and reception of audio or MIDI data by the corresponding MAP4's HAL implementation. The *mLAN Space* shown in Figure 5-13 implements the mLAN space fields defined in Table 5-2.

The *Register Space* implements the standard IEEE 1394 CSR registers described in Table 3-1 and Table 3-2, as well as the configuration ROM with the unit directory described by Figure 5-3. This enables a MAP4 Transporter to be detected on the IEEE 1394 bus, and enumerated accordingly by a controller. The next two subsections describe the software implementation of the mLAN Version 2 Enabler, and a corresponding HAL for the MAP4 Transporters.

5.3.2 The Yamaha Enabler Implementation

The source code implementations of the Yamaha Windows and Macintosh Enablers were studied in order to determine the current implementation strategies. Both Enablers have similar implementations and only differed in the implementation of the hardware abstraction layer. The findings are discussed below, using the Windows Enabler as reference. Five sequence diagrams: "Create mLAN Transporter", "Create mLAN Plugs", "Connect mLAN Plugs", "Disconnect mLAN Plugs" and "Get Plug Connections" are used in explaining the Yamaha Enabler implementation.

5.3.2.1 "Create mLAN Transporter" Sequence Diagram

An extension to the object model of the Basic Enabler (Figure 5-8) showing the implementation of a Transporter plug-in, is shown in Figure 5-18 below.

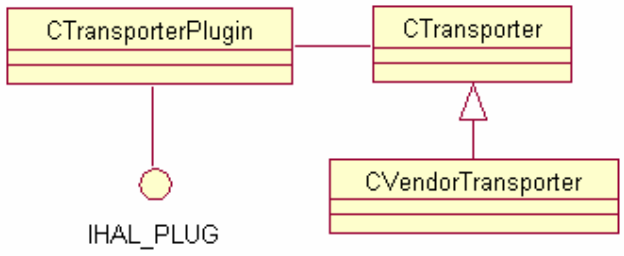


Figure 5-18: Transporter plug-in extension to the Basic Enabler object model

The *CTransporter* class now has an association with a *CTransporterPlugin* class, which implements the interface IHAL_PLUG. The corresponding sequence diagram for the creation of an mLAN Transporter object by an application is shown in Figure 5-19 below.

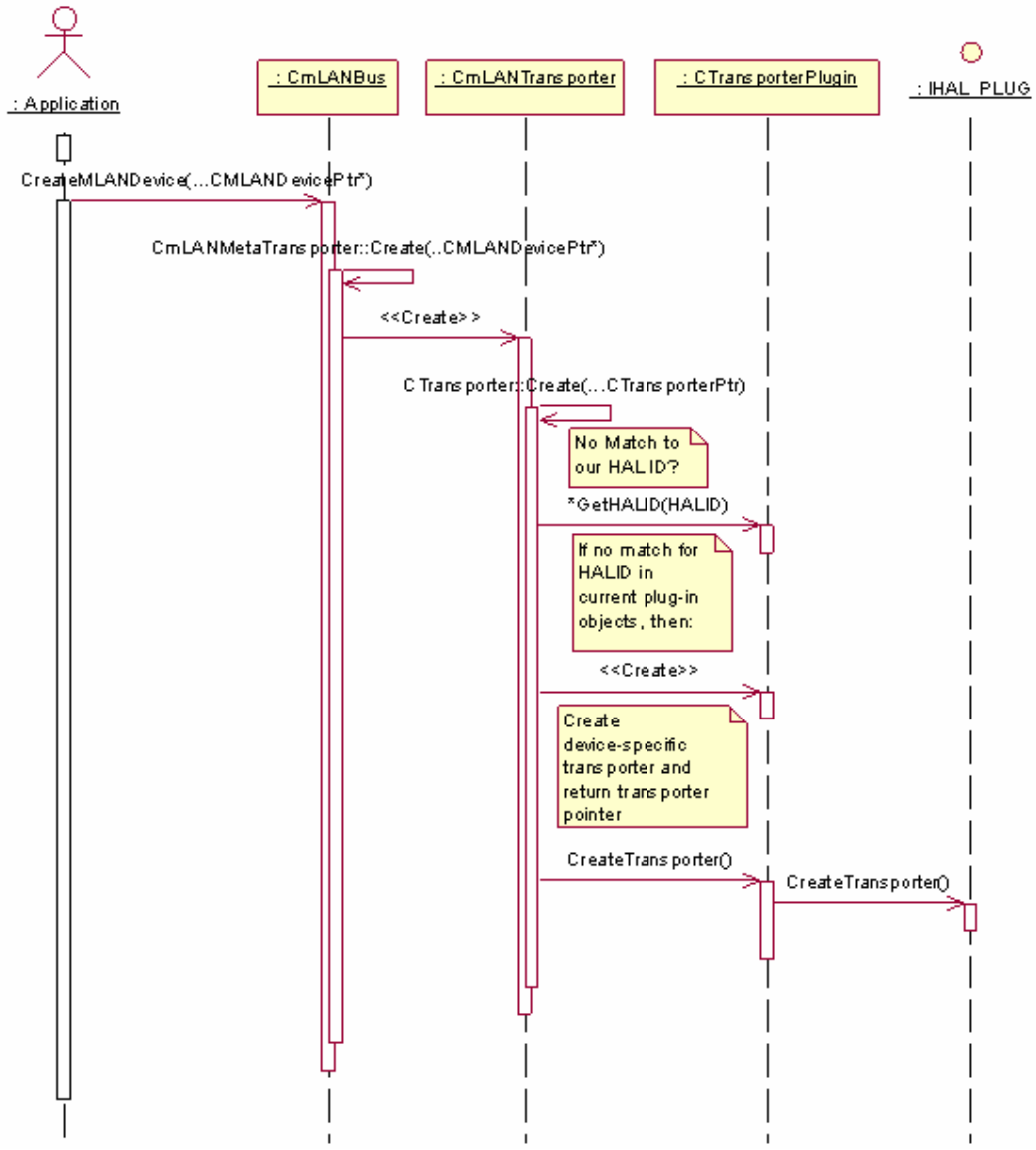


Figure 5-19: Create mLAN Transporter sequence diagram of the Windows Basic Enabler

After enumerating the interfaces and buses attached to the workstation, the application, in order to create mLAN (Transporter) device objects, requests the

appropriate mLAN bus[9] to create the mLAN device. This request will return a pointer to an mLAN device via one of its parameters, *pDevicePtr,* as shown below:

```
CreateMLANDevice(IEEE1394DeviceOID deviceOID, CmLANDevicePtr *pDevicePtr);
```

Remember from the implementation of the mLAN Version 1 Enabler (page 101), it was stated that *objectIDs* (object identifiers) were used by the Enabler to identify interfaces, buses and devices. The *deviceOID* parameter in the above mentioned function call specifies the object identifier associated with the mLAN device.

Within the *CreateMLANDevice(…)* function there is a call to the static function *Create(…)* of the *CmLANMetaTransporter* class. Within this static function is a request to create a new *CmLANTransporter* object as shown below:

```
*pDevicePtr = new CmLANTransporter(pBus, deviceOID);
```

The parameter *pBus*, specifies the bus on which the mLAN device is attached. Following the statement above, the pointer to the mLAN device returned to the application is actually a pointer to a *CmLANTransporter* object.

In the constructor of *CmLANTransporter* is a call to a static member function *Create(…),* of the *CTransporter* class. This is required in order to obtain the associated vendor Transporter object, hence marking the initiation of the A/M Manager Layer of the Enabler – see Figure 5-7. A vendor Transporter object has to be created from a loaded code library (plug-in) installed on the workstation running the Enabler. The loaded plug-in to use is identified using the Transporter's HAL ID – a combination of its HAL VendorID and HAL ModelID stored within the configuration ROM. For each plug-in that is loaded, a plug-in object of class *CTransporterPlugin* is created and a pointer to the object stored in an array. Within the *CTransporter::Create()* function, each of these stored plug-ins is searched to determine whether it is associated with the *HALID* of the vendor Transporter being created. If no such plug-in object exists, an attempt is made to load one with the specified Transporter HALID.

[9] Only one bus exists at this stage, since IEEE 1394 bridges device are not implemented as yet.

As discussed in section 5.2 (Figure 5-9) the loaded *CTransporterPlugin* object exposes a number of functions, including a *CreateTransporter(…)* method that causes a vendor-specific Transporter to be created, and a pointer to this object returned. The plug-in mechanism adopted for the Windows Enabler is discussed in the next paragraph.

Plug-in Mechanism adopted for the Yamaha Windows Enabler

The Yamaha Windows Enabler implementation makes use of the Windows Component Object Model (COM) and the Windows Registry, in order to discover and load HALs for vendor-specific Transporter implementations. Note that this procedure is described in the document "mLAN Transporter Plug-in Mechanism" [Yamaha Corp., 2004b], and is discussed briefly in this section.

The Enabler stores a number of loaded *CTransporterPlugIn* objects in an array similar to the one shown below:

```
static CTransporterPlugInPtr s_aPlugInPtr[kNumMaxPlugIns];
```

Within the *Create(…)* function of *CTransporter*, this array is searched to determine whether a *CTransporterPlugIn* object associated with the device's *HALID* exists. The search takes the following form:

```
0   for ( ixPlugInEntry=0 ; ixPlugInEntry < s_numPlugInEntries ; ixPlugInEntry++ ) {
1
2   if (s_aPlugInPtr[ixPlugInEntry]→GetHALID() == myHALID) {
3
4           pPlugIn = s_aPlugInPtr[ixPlugInEntry];
5           break;
6       }
7   }
```

If the search is unsuccessful, a new Transporter plug-in object is created. If this creation operation is successful, it is then added to the list of plug-in objects. This procedure is shown in the code below:

```
0   if (!pPlugIn) {
1
2       if (s_numPlugInEntries < kNumMaxPlugIns) {
```

```
3
4               try {
5                   pPlugIn = new CTransporterPlugIn(myHALID);
6                   MyThrowIfMemFail(pPlugIn);
7                   s_aPlugInPtr[s_numPlugInEntries++] = pPlugIn;
8               }
9               …
10          }
11          …
12      }
```

In the constructor of *CTransporterPlugIn*, an attempt is made to load a HAL plug-in specified by the Transporter's HALID (line 5). First, the COM libraries are loaded via the *CoInitialize* call:

```
if ( FAILED( CoInitialize( NULL )))
```

Following this, the Windows registry is searched for Transporter plug-ins/components. Information about the various Transporter components would previously have been loaded into the registry. The Enabler expects a particular hierarchy of keys that leads to a Transporter plug-in key, the defined hierarchy is as follows:

HKEY_LOCAL_MACHINE / SOFTWARE / YAMAHA / HAL_PLUG / [HAL_KEY]

The following set of instructions results in the retrieval of a handle to the HAL_PLUG application key.

```
0   if (ERROR_SUCCESS == RegOpenKeyEx(HKEY_LOCAL_MACHINE,"Software", 0,
        KEY_READ, &hSoftware)){
1       if (ERROR_SUCCESS == RegOpenKeyEx(hSoftware, "YAMAHA", 0,
        KEY_READ, &hCompany)){
2           if (ERROR_SUCCESS == RegOpenKeyEx(hCompany, "HAL_PLUG", 0,
            KEY_READ, &hApplication)
```

Next, there is an iterative retrieval of HAL plug-in keys:

```
0   while (ERROR_NO_MORE_ITEMS != RegEnumKeyEx(hApplication, index++, szBuffer, &size,
    NULL, NULL, NULL, NULL)){
1       if (ERROR_SUCCESS == RegOpenKeyEx(hApplication, szBuffer, 0, KEY_READ,
        &hAppSubName)){
2           if (ERROR_SUCCESS == RegQueryValueEx(hAppSubName,"CLS_ID", 0, NULL,
            (LPBYTE)szCLSID, &sizeClsID)){
3               if (ERROR_SUCCESS == RegQueryValueEx(hAppSubName,"REF_IID", 0,
                NULL, (LPBYTE)szIID, &sizeIID)){
```

```
4                   if (ERROR_SUCCESS == RegQueryValueEx(hAppSubName,"DLL_NAME",
                    0, NULL,  (LPBYTE)szDllName, &sizeDllName)){
```

On each iteration, the CLSID (a GUID identifying the component), and the location of the next stored component's *dll* are retrieved from value fields associated with the component's key entry.

The Windows COM function, *CoCreateInstance(…)*, is invoked to create an instance of the Transporter plug-in's component class, and to provide a pointer to the *IHAL_PLUG* interface.

```
HRESULT hr = CoCreateInstance(  CLSID_HAL_PLUG, NULL,
                                CLSCTX_ALL, IID_IHAL_PLUG,
                                reinterpret_cast<void**>(&MyPointer) );
```

This pointer is used to retrieve the *HALID* associated with the plug-in:

```
long retVal  = MyPointer→GetHALDescription(&HalInfo);
```

If this HALID value matches the HALID specified in the constructor of the *CTransporterPlugin* object, the *IHAL_PLUG* interface, *MyPointer*, is stored as an attribute of the object and used by the plug-in object for the creation and disposing of the corresponding vendor Transporter objects. Following this, the *Create(…)* function of the *CTransporter* class requests the *CTransporterPlugin* object to create a vendor-specific Transporter object as indicated below

```
status = pPlugIn→CreateTransporter(pBus, deviceOID, &pParam→transporterID);
```

This function, in turn, invokes the *CreateTransporter(…)* interface function of the *IHAL_PLUG* interface.

```
m_MyPlugInInfo → CreateTransporter( (LPSTR)pBus, (LPSTR)deviceOID, (LPSTR)this,
                                    (char**)pTransporter, &status )
```

This must be implemented by the *CreateTransporter* function of the actual component class. In the case of the MAP4 HAL plug-in, this implementation incorporates the following function call:

```
*pTransporter = (LPSTR)new CMAP4Transporter(pBus, deviceOID, pPlugIn);
```

After successfully creating a C*mLANTransporter* object and its associated vendor-specific Transporter object, the Enabler is required to enumerate mLAN plugs on behalf of the device. This next subsection describes how this is achieved by the Enabler.

5.3.2.2 "Create mLAN Plugs" Sequence Diagram

The sequence diagram that describes how the mLAN plugs are enumerated by the Enabler is shown below in Figure 5-20.

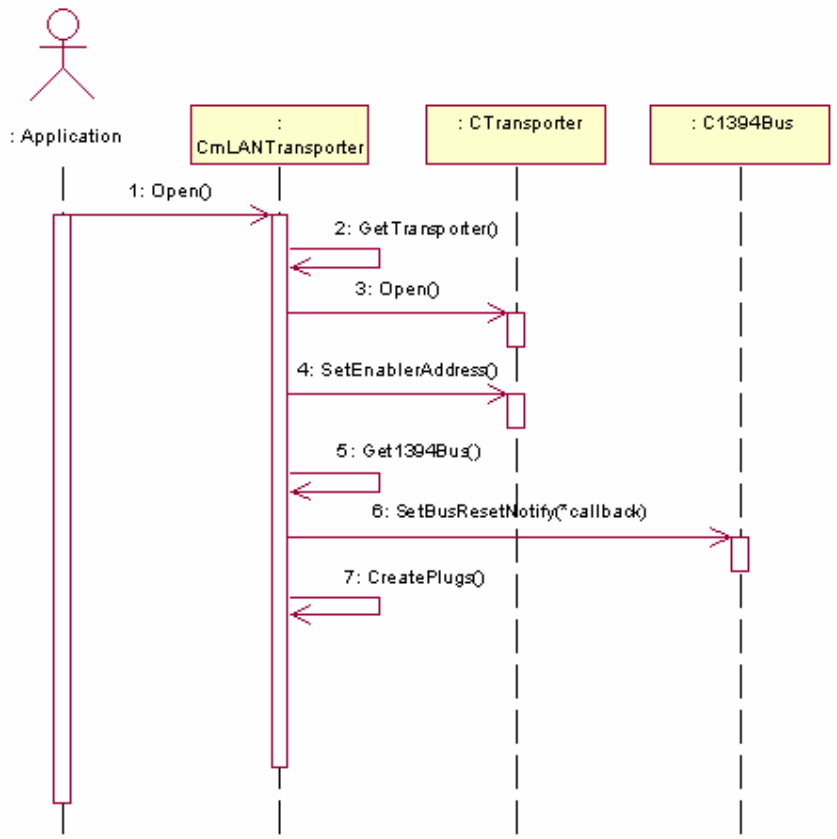


Figure 5-20: Sequence diagram that describes mLAN plug creation

The application initiates this by calling the *Open(…)* method of an mLAN Transporter object. This results in setting the Enabler address of the associated Transporter, and also specifying a call-back handler to its parent 1394 bus, which ensures that the Enabler address of the associated Transporter is always set in the event of a bus reset. Following this, the mLAN plugs are created. Plugs are created for input and output audio, MIDI and word clock. The creation of the mLAN plugs involves a two-step process: first determining the number of plugs to create, and then identifying sequences and subsequences for each plug.

Determining the Number of mLAN Plugs to Create

The algorithm adopted by the Enabler to determine the number of plugs to create is shown below.

```
0   status = pTransporter→GetMaxIsochChannels(m_isDst, &numIsochChannels);
1   if (status != kOSErr_NoError) {
2         goto Quit;
3   }
4
5   if (m_plugType == CmLAN::kPlugType_Clock) { // If word clock plug to be  created
6         if (m_isDst) {
7               m_numPlugs = 1;  // If input, set number of plugs to create to one
8         }
9         else { //If output, check if the Transporter can transmit audio sequences
10              const UInt32 seqType = ::kSequenceType_Audio;
11              UInt32          maxSequences;
12
13              for ( isochID = 0 ; isochID < numIsochChannels ; isochID++ ) {
14                    status = pTransporter→GetMaxSequences(m_isDst, isochID, seqType,
                      &maxSequences);
15                    if (status == kOSErr_NoError && maxSequences) {
16                          m_numPlugs = 1;
17                          break;
18                    }
19              }
20        }
21  }
22  else { // Create Audio and MIDI plugs
23              UInt32   seqType, numSequences, seqID, numSubs;
24              seqType = ::ConvertFromCmLANDevicePlugType(m_plugType);
25
26              for ( isochID = 0 ; isochID < numIsochChannels ; isochID++ ) {
27                    status = pTransporter→GetNumSequences(m_isDst, isochID, seqType,
                      &numSequences);
28                    if (status == kOSErr_NoError) {
29                          for ( seqID=0 ; seqID < numSequences ; seqID++ ) {
30                                status = pTransporter→GetNumSubsequences(m_isDst, isochID,
                                  seqType, seqID, &numSubs);
```

```
31                                  if (status == kOSErr_NoError) {
32                                      m_numPlugs += numSubs; // accumulating plug count
33                                  }
34                              }
35                          }
36                      }
37  }
```

First, the maximum number of isochronous FIFO buffers capable of transmitting or receiving isochronous streams is retrieved for a particular mLAN plug direction (input/output) as shown in line 0. The direction is specified by the parameter *m_isDst* and the number of FIFO buffers is returned in *numIsochChannels*. The number of plugs for the specified plug type is then determined.

The number of input word clock plugs (indicated by line 2) is always 1. This is because A/M devices are required, according to the IEC 61883-6 specification, to be able to receive SYT synchronization via the 1394 bus. For output word clock, the number of plugs to create is either 0 or 1 depending on whether the associated Transporter is capable of transmitting audio sequences in any of its output isochronous FIFO buffers. This check is performed by calling the *GetMaxSequences(…)* method of the associated Transporter object for each output isochronous FIFO buffer (indicated by lines 13 to 19). The number of plugs is set to 1 when an isochronous FIFO buffer capable of transmitting isochronous sequences is found.

For audio and MIDI plug types, the number of input (or output) plugs to create is equivalent to the cumulative sum of the number of subsequences that is contained within each isochronous sequence carried by all the isochronous streams that are being received (or transmitted) by the Transporter. This is indicated by lines 26 through to 36. For each input (or output) isochronous FIFO buffer, represented by *isochID*, the number of input (or output) sequences that are being received (or transmitted) is obtained, as shown in line 27. For each isochronous sequence, represented by *seqID*, the isochronous FIFO buffer is queried to determine the number of subsequences that are being received (or transmitted). This is shown in line 30. The number of subsequences returned from this function call is cumulatively added to form the total number of plugs to create.

After determining the number of plugs, objects of the *CmLANTransporterPlug* class are created with a plug ID value as identifier that ranges from 0 to *m_numPlugs*, where *m_numPlugs* is the total number of plugs. During the creation of the mLAN plug object, the plug ID value together with the specified plug type and direction are used to determine the isochronous sequence and subsequence that is to be associated with the plug. This is discussed in the next paragraph.

Identifying Isochronous Sequences and Subsequences

The *CmLANMetaTransporterPlug* class (parent class to *CmLANTransporterPlug)* defines a number of attributes that bind the mLAN plug to a particular subsequence of an isochronous sequence contained within an isochronous stream that is transmitted or received by an isochronous FIFO buffer. The defined attributes are *m_isochID*, *m_seqID*, *m_seqType* and *m_subID*;

- The attribute *m_isochID* identifies the isochronous FIFO buffer that transmits or receives an isochronous stream.
- The attribute *m_seqID* identifies a particular sequence buffer within the isochronous FIFO buffer that transmits or receives an isochronous sequence.
- The attribute *m_ seqType* indicates the type – audio or MIDI of the sequence.
- The attribute *m_subID* identifies a particular subsequence buffer within a sequence buffer that transmits or receives a subsequence of data.

Note that these definitions given to *m_isochID, m_seqID* and *m_subID* are to aid conceptual understanding; the actual values of these IDs depend on the actual Transporter HAL implementation. For example, the MAP4 Transporter uses the mLAN-NC1 chip for MIDI transmission and reception, and the mLAN-PH2 chip for audio transmission and reception. For transmissions, there will be two isochronous FIFO buffers (audio and MIDI) that transmit data, the *isochID* value used in identifying these buffers is specified by the MAP4's HAL implementation, which can either have the value 0 or 1.

These attributes are initialized within the constructor of the *CmLANMetaTransporterPlug* class, a generalized version of the routine is shown below, and is explained in the paragraph that follows.

```
0    if (plugType == CmLAN::kPlugType_Clock && isDst) { //Set to default if plug is an input clock
1
2        m_isochID   = 0;
3        m_seqType  = CMetaTransporter::kSequenceType_Audio;
4        m_seqID     = 0;
5        m_subID     = 0;
6    }
7    else { // Initialize ID parameters for other plug types
8       UInt32 seqType, plugIDAct, plugIDTemp = 0;
9
10       seqType =  (plugType == ::kPlugType_Clock) ? kSequenceType_Audio :
11                                   ::ConvertFromCmLANDevicePlugType(plugType);
12        m_seqType = seqType;
13        plugIDAct = (plugType == CmLAN::kPlugType_Clock) ? 0 : plugID;
14
15       UInt32 numIsochChannels, isochID, numSequences, seqID;
16
17       err = pTransporter→GetMaxIsochChannels(isDst, &numIsochChannels);
18       for (isochID = 0; isochID < numIsochChannels; isochID++ ) {
19
20          err = pTransporter→GetNumSequences(isDst, isochID, seqType, &numSequences);
21          for (seqID = 0; seqID < numSequences; seqID++) {
22
23              UInt32  numSubs, subID;
24
25              err = pTransporter→GetNumSubsequences(isDst, isochID, seqType, seqID, &numSubs);
26              if (plugIDAct < plugIDTemp + numSubs) { //Find plug ID range
27
28                  subID = plugIDAct - plugIDTemp;
29
30                  m_isochID  = isochID;
31                  m_seqID    = seqID;
32                  m_subID    = subID;
33
34                  goto Quit;
35              }
36
37              plugIDTemp += numSubs;
38          }
39       }
40   }
```

If an input word clock plug is being created, indicated by lines 0 through to 6, the *m_isochID*, *m_seqID*, and *m_subID* attributes are not used by the plug and are initialized to zero. The mechanism adopted for input word clock plug connections is discussed shortly. However, the *m_seqType* attribute has a value that corresponds to an audio sequence type.

Lines 7 through to 40 describe the generalized initialization routine for input and output audio and MIDI plugs as well as output word clock plugs. The sequence type that corresponds to the specified plug type is first obtained (line 10) and is used to

initialize the *m_seqType* attribute of the class. The mechanism employed in determining the isochronous ID, sequence ID and subsequence ID to be assigned respectively to *m_isochID*, *m_seqID*, and *m_subID*, involves keeping track of an upper boundary plug ID, using the *plugIDTemp* variable, which corresponds to the cumulative sum of the number subsequences capable of being transmitted or received by sequence buffers of isochronous FIFO buffers. A match is determined (line 26) when the actual plug ID, held by the *plugIDAct* variable, specified via the class constructor (see line 13) is just less than the sum of the current value of *plugIDTemp* and the number of subsequences of the current isochronous ID and sequence ID. The current isochronous ID and sequence ID is used to initialize the attributes *m_isochID* and *m_seqID*. The subsequence ID that is allocated to *m_subID* is given by the difference in value between the actual plug ID (*plugIDAct*) and the upper boundary plug ID (*plugIDTemp)*.

An mLAN Transporter object is successfully created if all its mLAN plug objects are created. Following this, mLAN plugs of Transporter objects can be requested for connections and disconnection. The implementation procedure adopted by the Enabler in handling plug connections and disconnections are discussed in the next two paragraphs.

5.3.2.3 “Connect mLAN Plugs” Sequence Diagram

The sequence diagram that describes how mLAN plug connections are handled by the Enabler is shown below in Figure 5-21.

The application initiates this process by calling the *Connect(…)* method of the mLAN device object that contains the destination plug to connect. The parameters passed into this method call include the plug ID of the destination plug (represented by *myDstPlugID*), a pointer to the mLAN device object containing the source plug, (represented by *pSrcDevice*), the plug ID of the source plug (represented by *srcPlugID*), and an indication of the type of plugs (audio, MIDI or word clock) to connect (not shown). In the implementation of the device connection routine, the source and destination mLAN plug objects indicated by *srcPlugID* and *myDstPlugID* are retrieved and connected to each other in turn, indicated by sequences 4 and 5.

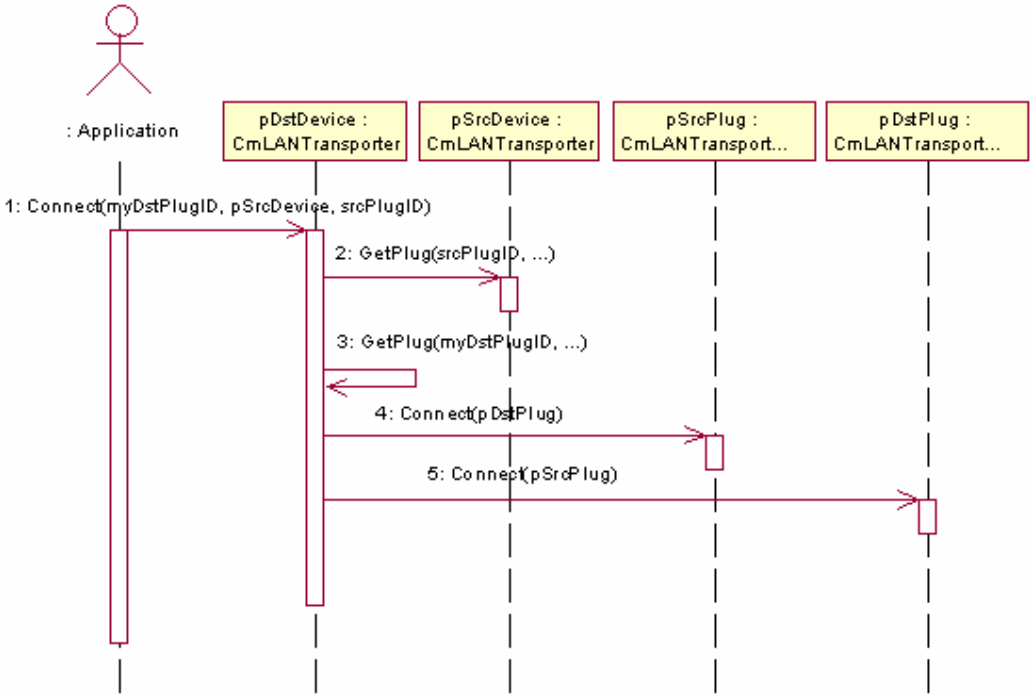


Figure 5-21: mLAN plug connection sequence diagram

The mLAN plug connection method is implemented by the *CmLANMetaTransporterPlug* class. As indicated by sequences 4 and 5 of Figure 5-21, this method takes as an argument a pointer to the target mLAN plug from/to which to connect. With reference to sequence 4, where the *Connect(…)* method is called on the source plug object with the destination plug as parameter, the source mLAN plug object is required to ensure that the associated Transporter device is streaming. The source code implementation of this is given below and is explained in the paragraph the follows.

```
0    err = pTransporter→IsRunning(::kIsOutput, &bRunning); //Check if Transporter is transmitting
1    if (err != kOSErr_NoError || bRunning) goto Quit;
2
3    //  Start transfer of the source device if not already transmitting
4    UInt64  channelsAvailable, checkBit;
5    UInt32  generation;
6
7    err = GetIsochChannel(&isochCh);
8    if (err != kOSErr_NoError) goto Quit;
9
10   if (isochCh == kInvalidIsochChannel) isochCh = 0;
11   // Find new isochronous channel if the original channel is invalid
12   err = pTransporter→GetBusPtr()→GetChannelsAvailable(&channelsAvailable, &generation);
13   if (err != kOSErr_NoError) goto Quit;
14
```

```
15   checkBit = 1ULL << isochCh;
16   if ((channelsAvailable & checkBit) == 0) { //Decide new channel available
17        if (channelsAvailable == 0) {
18             err = CmLAN::kErrChannelNotAvailable;
19             goto Quit;
20        }
21
22        do { //Find an avaliable channel
23             isochCh++;
24             checkBit >>= 1;
25             if (kMaxIsochChannel < isochCh) {
26               isochCh = 0;
27               checkBit = 1ULL << 63;
28             }
29        } while ((channelsAvailable & checkBit) == 0);
30
31        err = SetIsochChannel(isochCh); //Set channel to the Transporter
32        if (err != kOSErr_NoError) goto Quit;
33
34        err = pTransporter→Start(CMetaTransporter::kIsOutput); // Start transmissions
```

First, a check is performed to determine the output streaming state of the Transporter (line 0). If the Transporter is streaming the method returns with no error. On the other hand, if the Transporter is not streaming, streaming is started. In the attempt to start streaming, the isochronous channel in use by the Transporter is retrieved (line 7) and a check is performed to determine whether the isochronous channel is in use by the bus (described by lines 10 to 16). If the isochronous channel is in use, a new available channel is determined (lines 16 through 29) and then set to the Transporter to be used for transmissions (line 31), after which the Transporter is made to start streaming. When streaming is started by the Transporter, it also allocates from the isochronous resource manager, the necessary isochronous bandwidth. The methods *GetIsochChannel(…)* (line 7) and *SetIsochChannel(…)* (line 31) of the *CmLANMetaTransporterPlug* class resolve to calls to the *GetIsochChannel(…)* and *SetIsochChannel(…)* methods respectively, of the associated Transporter object (see Figure 5-12). The isochronous FIFO buffer that is to have its isochronous channel modified, is identified using the *m_isochID* attribute of the *CmLANMetaTransporterPlug* class.

In the case where the *Connect(…)* method is called on the destination plug object with the source plug as parameter, (sequence 5 of Figure 5-21) the destination plug is required to copy configuration parameters from the specified source plug to overwrite

its own configuration parameters. The source code implementation of this by the Enabler is given below.

```
0    err = pMLANPlug→GetIsochChannel(&isochCh); //Get isoch. channel of source plug
1    if (err != kOSErr_NoError) goto Quit;
2
3    if (isochCh == kInvalidIsochChannel) {
4
5        err = CmLAN::kErrIllegalStatus;
6        goto Quit;
7    }
8    if (GetPlugType() == CmLAN::kPlugType_Clock) {
9       //Set SYT channel, if this plug is a word clock slave plug
10       err = SetIsochChannel(isochCh);
11   }
12   else {
13
14       UInt32 seqNo, subNo, sfc, dataType;
15       //Get configuration parameters of the source plug
16       err = pMLANPlug→GetSampleRate(&sfc);
17       if (err == kOSErr_NoError) err = pMLANPlug→GetSequenceNumber(&seqNo);
18       if (err == kOSErr_NoError) err = pMLANPlug→GetSubsequenceNumber(&subNo);
19       if (err == kOSErr_NoError) err = pMLANPlug→GetDataType(&dataType);
20       if (err != kOSErr_NoError)  goto Quit;
21       //Set configuration parameters to this destination plug
22       err = SetIsochChannel(kInvalidIsochChannel);
23       if (err == kOSErr_NoError && GetPlugType() == CmLAN::kPlugType_Audio) {
24
25            err = SetSampleRate(sfc);
26       }
27       if (err == kOSErr_NoError) err = SetSequenceNumber(seqNo);
28       if (err == kOSErr_NoError) err = SetSubsequenceNumber(subNo);
29       if (err == kOSErr_NoError) err = SetDataType(dataType);
30       if (err == kOSErr_NoError) err = SetIsochChannel(isochCh);
31
32       if (err == kOSErr_NoError) {
33            //Enabler reception if not already enabled
34            err = pTransporter→IsRunning(::kIsInput, &bRunning);
35            if (err == kOSErr_NoError && !bRunning) {
36
37              err = pTransporter→Start(::kIsInput);
38            }
39       }
40      }
```

The isochronous channel of the specified source plug is first retrieved. The method returns with an error if the returned isochronous channel indicates an invalid value (line 0 to 7). If the plug is of type word clock, the isochronous channel of the destination plug is set to correspond to that of the source. The destination plug, in this

case, is viewed as the *slave*, and hence this call resolves to a call to the *SetSYTSynchChannel(…)* method of the associated Transporter object.

For all other plug types, the sample rate, sequence number, subsequence number and data type are retrieved from the specified source plug and then set to the destination plug together with the retrieved isochronous channel. The sample rate is only set to when the plug type is audio.

Note that the methods that get and set these configuration parameters make use of the class attributes *m_isochID, m_seqID, m_seqType* and *m_subID* to identify the sequence and subsequence associated with the mLAN plug. After setting the configuration parameters, the Transporter associated with the destination plug is requested to start receiving isochronous streams if not already doing so.

5.3.2.4 "Disconnect mLAN Plugs" Sequence Diagram

The sequence diagram that describes how mLAN plug disconnections are handled by the Enabler is shown below in Figure 5-22.

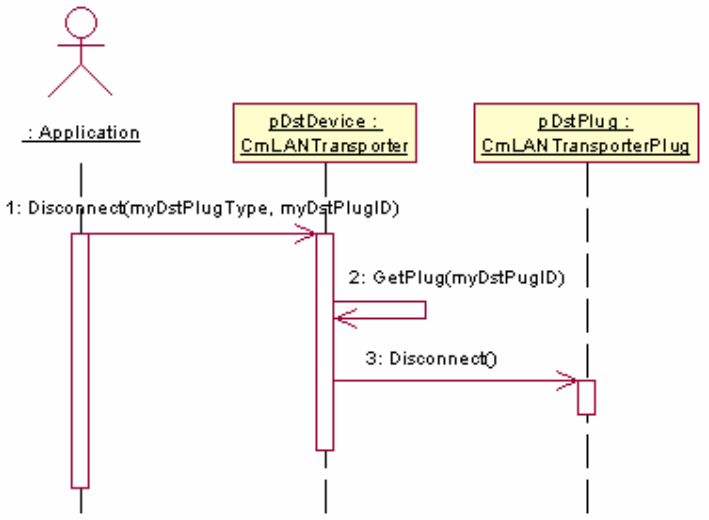


Figure 5-22: mLAN plug disconnection sequence diagram

This is invoked by the application via the *Disconnect(…)* method call on the mLAN Transporter device object that contains the destination plug to be disconnected. The plug type and the destination plug ID are passed as parameters to the function,

indicated by *myDstPlugType* and *myDstPlugID* respectively. This further results in the *Disconnect(…)* method being called on the actual destination mLAN plug object. The implementation of the *Disconnect(…)* routine of the mLAN plug class, *CmLANMetaTransporter*, results in setting the receiving subsequence number of the destination plug to an invalid value, as shown below:

SetSubsequenceNumber (kInvalidSubsequenceNumber)

This method results in a call to the *SetSubsequenceNumber(…)* method of the associated Transporter object, using the values obtained for the class attributes: *m_isochID, m_seqID, m_seqType* and *m_ subID* as parameters, while setting the subsequence number of the receiving 'subsequence FIFO buffer' to an invalid value. The causes the destination plug to stop receiving data.

With connections and disconnections in place, applications can display information about the target plugs connected to the various plugs of an mLAN device. In order to do this for a particular plug, an application needs to retrieve the plug's connections. The implementation of the get-plug-connections routine of the Enabler is discussed in the next paragraph.

5.3.2.5 "Get Plug Connections" Sequence Diagram

The sequence diagram that describes how connections to an mLAN plug are retrieved by the Enabler is shown below in Figure 5-23 below.

In determining connected plugs, the Enabler compares the configuration parameters – isochronous channel, sequence number and subsequence number, of the plug for which connections are to be determined (shown by sequence 2) to the configuration parameters (shown by sequence 8) of mLAN plugs of the same plug type, of all the other mLAN devices on the IEEE 1394 bus. A plug connection is identified if there is an isochronous channel, sequence number and subsequence number match between the two plugs (shown by sequence 9).

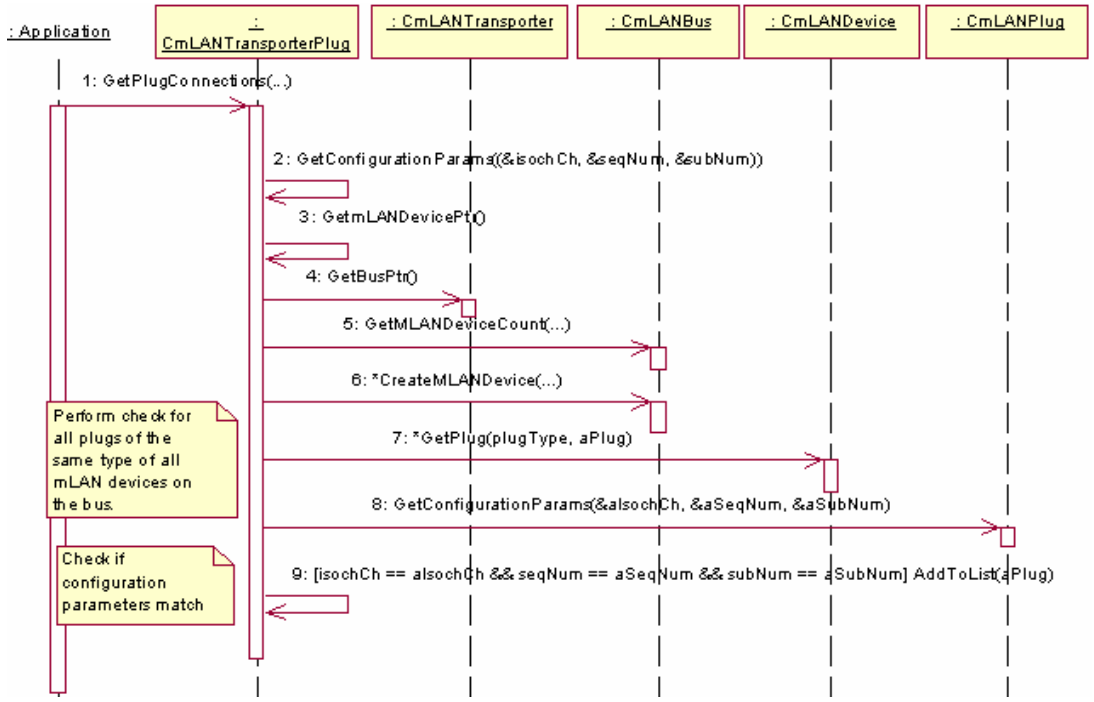


Figure 5-23: Sequence diagram that describes how mLAN plug connections are retrieved

The implementation of this is given by the source code extract given below, and is explained in the paragraph that follows. Note that the code has been altered for simplicity.

```
0     err = GetIsochChannel(&myIsochChannel); //Get configuration parameters
1     if (err == kOSErr_NoError)  err = GetSequenceNumber(&mySeqNo);
2     if (err == kOSErr_NoError)  err = GetSubsequenceNumber(&mySubNo);
3     if (err != kOSErr_NoError)    goto Quit;
4
5     if (myIsochChannel == kInvalidIsochChannel) {
6       err = kOSErr_NoError;
7       goto Quit;
8     }
9
10    err = pTransporter→IsRunning(IsDestination(), &bRunning);
11    if (err != kOSErr_NoError)  goto Quit;
12    if (!bRunning) goto Quit;
13
14    if (IsDestination()) { //Find the source plug that is connected to this destination plug
15
16       for (ixDevice = 0; ixDevice < numDevices; ixDevice++) {
17
18        err = pBus→CreateMLANDevice(deviceOID, &pDeviceTemp); //Create an mLAN device
19         if (err != kOSErr_NoError) continue;
20
21         pDeviceTemp→Open();
22         err = pDeviceTemp→GetNumPlugs(::kSrcPlug, GetPlugType(), &numPlugs);
23         if (err == kOSErr_NoError) {
```

```
24
25              for (plugID = 0; plugID < numPlugs; plugID++) {
26                //For all source plugs of this device, check for a match in configuration params
27                err = pDeviceTemp→GetPlug(::kSrcPlug, GetPlugType(), plugID, &pMLANPlug);
28                if (err == kOSErr_NoError) err = pMLANPlug→GetIsochChannel(&isochChannel);
29                if (err == kOSErr_NoError && isochChannel == myIsochChannel) {
30
31                 err = pMLANPlug→GetSequenceNumber(&seqNo);
32                  if (seqNo != mySeqNo)  continue;
33                  err = pMLANPlug→GetSubsequenceNumber(&subNo);
34                  if (subNo != mySubNo)  continue;
35
36                // Match found here, return connected plug an break off loop
37          }
38        delete pDeviceTemp;
39      }
40    }
41    else { //Find the destination plugs that are connected from this source plug
42      for (ixDevice = 0; ixDevice < numDevices; ixDevice++) {
43
44        err = pBus→CreateMLANDevice(deviceOID, &pDeviceTemp); //Create an mLAN device
45        if (err != kOSErr_NoError)  continue;
46
47        pDeviceTemp→Open();
48        err = pDeviceTemp→GetNumPlugs(::kDstPlug, GetPlugType(), &numPlugs);
49        if (err == kOSErr_NoError) {
50
51              for (plugID = 0; plugID < numPlugs; plugID++) {
52                //For all destination plugs of this device, check for a match in configuration params
53                err = pDeviceTemp→GetPlug(::kDstPlug, GetPlugType(), plugID, &pMLANPlug);
54                if (err == kOSErr_NoError)  err = pMLANPlug->GetIsochChannel(&isochChannel);
55                if (err != kOSErr_NoError || isochChannel != myIsochChannel)  continue;
56
57               err = pMLANPlug→GetSequenceNumber(&seqNo);
58                if (err != kOSErr_NoError || seqNo != mySeqNo) continue;
59
60                err = pMLANPlug→GetSubsequenceNumber(&subNo);
61                if (err != kOSErr_NoError || subNo != mySubNo)  continue;
62
63                // Match found here, add connected plug to a list
64              }
65        }
66        delete pDeviceTemp;
67      }
68    }
```

The configuration parameters of the plug for which connected plugs are to be determined are retrieved, indicated by lines 0 to 3. The method returns, if an invalid channel (lines 5 to 8) is returned or if the associated Transporter is not streaming (lines 10 to 12). These indications automatically imply that there are no target connections to the plug.

The "'if" clause shown in line 14, determines the directional flow of the plug. This is required in order to compare configuration parameters of the plug with the appropriate plugs of other devices, i.e. if the plug is an input plug, the configuration parameters of the output plugs (lines 29 to 36) of the other devices are compared. Conversely the configuration parameters of the input plugs (lines 58 to 66) of other mLAN devices are compared if the plug is an output. If a match is identified for an input plug, the information about the target plug is returned and the configuration matching routine terminated. If a match is identified for an output plug, the information about the target plug is added to a list, and the configuration matching routine performed for the next plug until all the input plugs of all the devices have been examined.

The five sequence diagrams described above highlight the implementation of the Yamaha Windows Enabler as far as plug connections are concerned, showing how mLAN Transporter objects are created and how the plug abstraction layer and the A/M Manager layer interact to enable plug connections and disconnections. It is important to mention that the Enabler is also capable of accessing other device-specific information not mentioned in this section. This includes the Transporter's vendor and module names, as well as the nickname.

As part of the research of this project, a similar Enabler implementation was done for the Linux operating system; this is described in the next subsection.

5.3.3 The Linux Enabler Implementation

The plug abstraction layer and the A/M manager layer implementation of the Windows Enabler, described in the previous section, were adopted in the implementation of the Linux Enabler. In other words, the concepts involved in the process of plug creation and plug connections and disconnections are equivalent across the two platform implementations. Also, the process involved in creating mLAN Transporter objects is somewhat identical, and only differs in the implementation of the plug-in class, *CTransporterPlugIn*. This is described in the following paragraph.

5.3.3.1 The Linux Transporter Plug-in Implementation

The plug-in architecture implemented within the Linux Enabler makes use of the dynamic linking API [TLDP, 2003] via which objects to vendor-specific Transporter HAL implementations can be dynamically loaded. The API defines methods for obtaining a handle to a shared library, looking up symbols within the library, handling error messages and closing the handle. This implies that vendor Transporter HAL implementations have to be created and built as shared libraries, and placed in a particular location within the Linux directory structure, from where they can be accessed by the Enabler. The Linux Documentation Project – Program Library HOWTO, [TLDP, 2003], describes how shared libraries can be created and installed in a Linux environment. The directory path for shared libraries of Transporter HAL implementations is defined to be:

/usr/local/lib/HALs/

The Enabler iteratively examines the shared libraries installed in this location in order to create a plug-in object for a Transporter HAL implementation that it finds to be compatible with a particular Transporter device that is being enumerated. The compatible HAL implementation is determined by comparing the HAL ID value specified by the shared library implementation to the HAL ID value contained within the configuration ROM of the Transporter device. This is illustrated using the diagram shown in Figure 5-24.

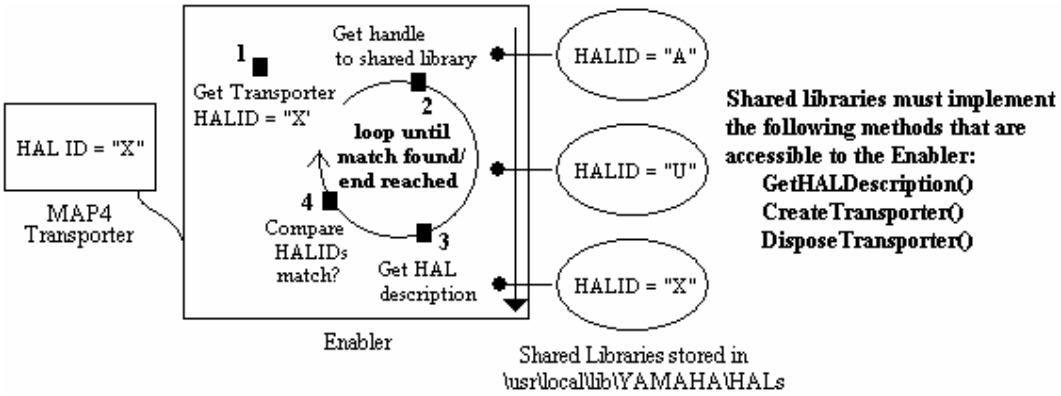


Figure 5-24: Linux Transporter plug-in implementation model

In the diagram, a MAP4 Transporter with an HAL ID value of “X” is shown to be connected to the Enabler. Also, a number of shared libraries that provide HAL implementations for Transporters having HAL ID of “A”, “U”, and “X” have been installed in the standard directory location: /usr/local/lib/HALs/. These HAL implementations, in addition to providing Transporter capabilities, are required to implement the *class factory* methods: *GetHALDescription(…), CreateTransporter(…)* and *DisposeTransporter(…).*

These methods return information about the Transporter HAL, and allow Transporter HAL objects to be created and destroyed. For Enablers to create a plug-in for the MAP4 Transporter, it first has to retrieve the HAL ID of the connected Transporter (indicated by *step 1*). The next step is to determine the installed HAL shared libraries and get a handle (*step 2*) to the first. The description of the HAL shared library (*step 3*) is retrieved and compared with the HAL ID of the MAP4 (*step 4*). In this case the HAL ID of the first HAL shared library is “A” and does not correspond to that of the MAP4 (“X”). A handle to the next shared library is retrieved, and the process repeated until a HAL ID match is identified or after all the shared libraries have been examined, thus indicating that a HAL implementation does not exist. From the setup shown in the diagram, a match would be identified for the third shared library, which would then be used for creating and disposing Transporter HAL objects using the associated *CreateTransporter(…)* and *DisposeTransporter(…)* methods.

The above algorithm is implemented in the constructor of the *CTransporterPlugIn* class, and also follows from the “Create mLAN Transporter” sequence diagram shown in Figure 5-19. The code implementation of this algorithm, demonstrating the use of the dynamic linking API, is described below and is explained in the paragraph the follows. Note that this code has been simplified.

```
0    pFileStream = fopen(kPstrPlugInDumpFile, kPstrOpenRead);
1    if (pFileStream) {//Create text file to cache the file locations of shared libraries
2
3        while (!feof(pFileStream)) {
4
5            if (fgets(pLibraryLocation, ::kNameStrMaxLength, pFileStream)) { //Get a library location
6
7                pLibraryHandle = dlopen(pLibraryLocation, RTLD_NOW); //Get a handle to the library
8                if (!pLibraryHandle) continue;
```

```
9               //Get handle to the GetHALDescription method implemented by the library
10              pFuncGetHALDescription = (GetHALDescriptionFunc*) dlsym(pLibraryHandle,
                                                                kPstrFunc_GetHALDescription);
11
12              if (!pFuncGetHALDescription) {
13
14                  dlclose(pLibraryHandle);
15                  continue;
16              }
17
18              pFuncGetHALDescription(&description); //Check if HALIDs match
19              if (halID == description.halID) {
20                  //If HALIDs match, initialize this plug-in object
21                  m_pHandle = pLibraryHandle;
22
23                  m_pFuncCreateTransporter  = (CreateTransporterFunc*)dlsym(pLibraryHandle,
                                                                kPstrFunc_CreateTransporter);
24
25                  m_pFuncDisposeTransporter = (DisposeTransporterFunc*)dlsym(pLibraryHandle,
                                                                kPstrFunc_DisposeTransporter);
26
27                 break;
28              }
29
30              dlclose(pLibraryHandle);
31            }
32        }
33
34        fclose(pFileStream);
35        system(kPstrDisposeDumpFile);
36    }
```

Referring to line 0, a file stream is obtained to a text file that has been created to contain the path to the Transporter HAL shared libraries located in the /usr/local/lib/HALs/ directory. The string constant *kPstrPlugInDumpFile* that is used in the *fopen(…)* function call, is defined to be "find /usr/local/lib/HALs/lib*.so.? -fprint PlugInFileDump". With reference to the illustration shown in Figure 5-24, this operation creates and returns a file stream to the text file *PlugInFileDump* that contains the following entries:

/usr/local/lib/HAL/lib/TransporterHALA.so.1.0.1
/usr/local/lib/HAL/lib/TransporterHALU.so.1.0.2
/usr/local/lib/HAL/lib/TransporterHALX.so.1.0.0

If the file stream is obtained successfully, each line of the *PlugInFileDump* file is read to obtain the absolute location of a HAL shared library (line 5). The absolute location retrieved from the file is stored in the variable *pLibraryLocation*, which is then used

by the *dlopen(…)* method (in line 7) to obtain a handle to the shared library specified by *pLibraryLocation*. After obtaining the handle, a pointer to the *GetHALDescription(…)* class factory method of the shared library is obtained via the *dlsym(…)* method call (line 10). The string constant *kPstrFunc_GetHALDescription*, defined to be "GetHALDescription", specifies the name of the method to be resolved by *dlsym(…)*. Using this pointer, the HAL description is obtained (line 18) and a check performed to determine whether a match exists with the HAL ID of an actual Transporter device (line 19). If a match is identified, the handle to the shared library together with function pointers to the shared library's *CreateTransporter(…)* and *DisposeTransporter(…)* class factory methods are obtained and held by the plug-in class (lines 21 to 27). The function pointers are stored in the attributes *m_pFuncCreateTransporter* and *m_pFuncDisposeTransporter* respectively. If a match does not exist, the process is repeated for the next available shared library.

Once the Transporter plug-in object is created, it can then be requested to create Transporter objects via its *CreateTransporter(…)* method. This in turn makes use of the function pointer *m_pFuncCreateTransporter*, with the appropriate parameters, to create the Transporter object as shown below:

```
*pTransporter = m_pFuncCreateTransporter(pBus, deviceInformation, configurationROM, …);
```

This results in the creation of the HAL Transporter object, which in the case of the MAP4 would be:

```
*pTransporter = new MAP4Transporter (pBus, deviceInformation, configurationROM …);
```

Other plug-in approaches can be adopted by the Linux Enabler for interacting with Transporter HALs; the method described in this section is (Linux) platform specific and can be used across a number of UNIX platforms. For portability, the *glib* library [GNOME Project, 2004] can be used as an alternative in implementing a portable interface to Transporter HAL implementations.

Both Enablers were preliminarily tested using the MAP4 Evaluation board. In order to do this a corresponding HAL had to be developed to interface to the Enabler. The next

section briefly describes the design and implementation of the HAL with particular reference to establishing plug and word clock connections.

5.3.4 MAP4 Transporter HAL Implementation

Yamaha developed a MAP4 HAL plug-in for their Basic Enabler implementation. For the Linux Enabler, a MAP4 HAL plug-in was also developed. The design structure of Yamaha's MAP4 HAL implementation is described below. The Linux MAP4 HAL plug-in utilizes this model.

5.3.4.1 Object Model of the MAP4 HAL

The object model that describes the HAL implementation of the MAP4 Transporter is shown in the figure below.

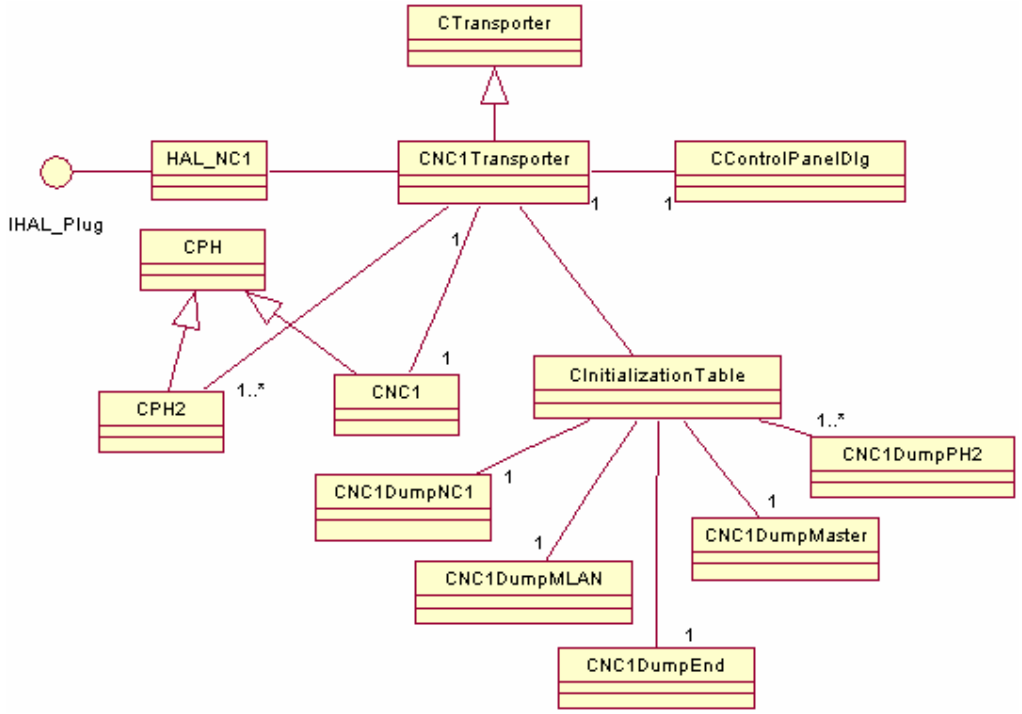


Figure 5-25: Object model describing the MAP4 HAL implementation

The *CNC1Transporter* class represents the actual MAP4 Transporter device, and provides the HAL implementation for the MAP4 Transporter according to the Transporter HAL interface defined by *CTransporter*. A number of classes are shown to be associated with the *CNC1Transporter* class. These include a control panel dialog class (*CControlPanelDlg*), an initialization table class (*CInitializationTable*), an mLAN-NC1 chip class (*CNC1*), and an mLAN-PH2 chip class (*CPH2*). An

association is also shown to the *HAL_NC1* class, which together with *IHAL_Plug* interface allows the HAL to be accessed by the Enabler. The process involved in creating an object of the *HAL_NC1* class has been described in the "Create mLAN Transporter" sequence diagram (section 5.3.2.1).

The classes *CPH2* and *CNC1* define methods that access the registers of the onboard mLAN-NC1 and mLAN-PH2 chips of the MAP4. This is achieved through the use of asynchronous read and writes to the appropriate address offsets within the Core Space of the MAP4 (described in section 5.3.1). The *CPH* class, which is shown to be the parent to *CPH2* and *CNC1*, defines a set of operations that are common to both.

The initialization table class (*CInitializationTable*) implements methods that define the structure and the parameters of the initialization table entries that are to be stored in the initialization table area of the MAP4's Boot Parameter Space (see Figure 5-14). Recall from section 5.3.1 that this initialization table area refers to non-volatile memory that contains the A/M transmission parameters that are to be loaded up by the Transporter (at power-on) in order to restore the previous state of the Transporter. The *CInitializationTable* class makes use of other classes: *CNC1DumpNC1*, *CNC1DumpPH2* and *CNC1DumpMLAN* that respectively define the structure and boot-loadable parameters for the mLAN-NC1 chip, mLAN-PH2 chip and the BANDWIDTH_REQUEST and CHANNEL_REQUEST values of the mLAN Space. The structure of the top and the bottom of the initialization table area is defined by *CNC1DumpMaster* and *CNC1DumpEnd*, respectively.

The *CControlPanelDlg* class implements a control panel dialog that displays the current A/M parameter configuration of the MAP4, and allows them to be modified.

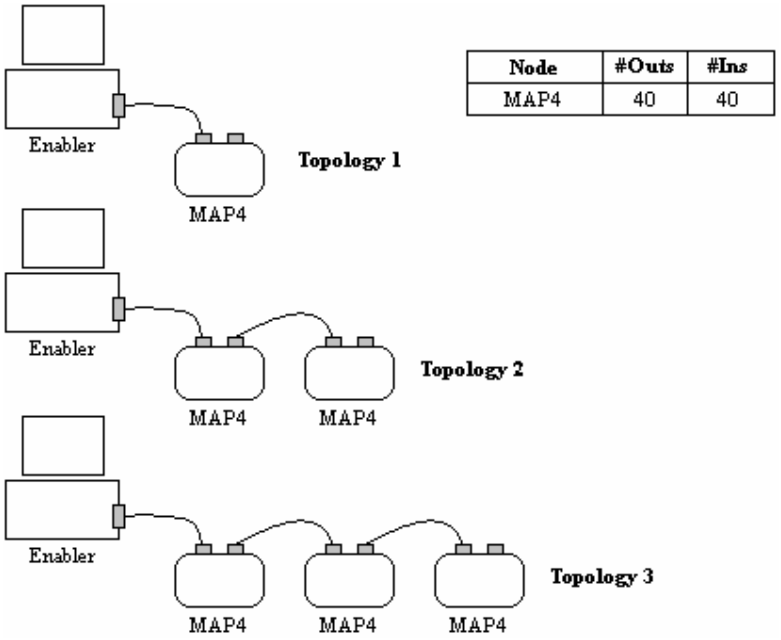
5.3.5 Quantitative and Qualitative Analysis

The performance of the mLAN Version 2 architecture is expected to be superior to the mLAN Version 1 architecture. In the mLAN Version 2 architecture, high-level abstractions of various components of a device are implemented by the Enabler, where 1394 bus communication with the device occurs using basic asynchronous read, write and lock operations. In contrast, mLAN Version 1 provides an architecture

where the high-level abstractions of its components are implemented in the form of information blocks. An Enabler communicates using FCP command and response messages to retrieve and modify information contained within these information blocks. Timing measurements of the mLAN Version 2 Enabler were conducted to confirm this. The approach adopted was similar to that used in mLAN Version 1, in which a number of time measurements were taken to determine the speed at which an application enumerates a network, and also to determine the speed at which the Enabler performs plug operations. A description of the procedure and outcomes of these measurements are discussed below.

5.3.5.1 Measuring Speed of Network Enumeration

Network topologies similar to those used in testing the mLAN Version 1 architecture, were used in conducting timing measurements to determine the speed at which an application using the mLAN Version 2 architecture enumerates a network. However, in this case, the Transporter nodes of each device were configured to have the maximum number of input and output plugs. Figure 5-26 illustrates the network topologies used, with the various timing measurements given in Table 5-3.



Node	#Outs	#Ins
MAP4	40	40

Figure 5-26: Network topologies used in measuring enumeration speed

	Timing Measurements [seconds]			
Topology	**1st**	**2nd**	**3rd**	**Average**
1	0.70	0.71	0.71	*0.71*
2	1.03	1.02	1.03	*1.03*
3	1.35	1.35	1.35	*1.35*

Table 5-3: Timing measurements for mLAN Version 2 device enumeration

The timing measurements taken for each topology were performed under the same networking conditions, with each MAP4 Transporter configured for the worst case, thus allowing a maximum of 40 input plugs and 40 output plugs to be enumerated by the Enabler. By taking the difference in the timing average between 'Topology 2' and 'Topology 1', it is observed that the time taken by the Enabler to enumerate a Transporter node with 40 inputs and 40 outputs is 0.32 seconds. This value compared with that obtained for an mLAN Version 1 device with 9 outputs and 1 input (3.41 seconds) shows a speed increase of about 90.6% (10.7 times faster), which clearly makes the mLAN Version 2 architecture superior to mLAN Version 1, in terms of enumeration and performance.

5.3.5.2 Measuring Speed of Plug Operations

Measurements were also taken to determine the speed of a number of plug operations performed by the Enabler. The plug operations measured were similar to those used in mLAN Version 1. The setup shown in Figure 5-27 was used in performing the measurements for three sets of thirty-two operations.

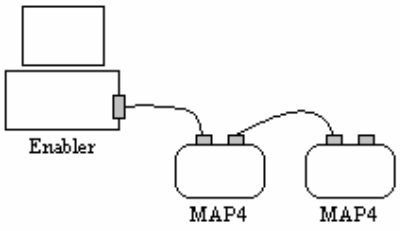


Figure 5-27: Device setup used in performing timing measurements

The results of the timing measurements are summarized in Table 5-4. Note that, as before, the source plugs used in timing the speed of retrieving plug connections, did not have any target connections.

	Timing Measurements		
	Connections [s]	**Disconnections [s]**	**Get Connections[s]**
1st Thirty-two Operations	2.44	1.59	0.64
2nd Thirty-two Operations	2.37	1.60	0.64
3rd Thirty-two Operations	2.39	1.60	0.63
Average of 96 Operations	*0.08*	*0.05*	*0.02*

Table 5-4: Timing measurements for a number of plug operations

The mLAN Version 2 architecture, compared to mLAN Version 1 shows a significant speed up in performing plug connections and disconnections. The average time taken to establish a plug connection is about 80 milliseconds, and 50 milliseconds for a plug disconnection. Recall from Table 4-3 that the average time taken to connect and disconnect plugs of mLAN Version 1 devices was 2.4 seconds and 1.6 seconds respectively. The timing results shown in Table 5-4 indicate a speed increase of about 96.7% (30 times faster) for plug connections, and 96.8% (32 times faster) for plug disconnections.

The average time taken for retrieving plug connections is found to be 20 milliseconds, which is about 8696 times slower than the speed obtained for mLAN Version 1 plugs (0.0023 milliseconds). Recall that the mLAN Version 1 Enabler retrieved plug connections from a number of cached attributes. However, the order of magnitude provided by the mLAN Version 2 Enabler still remains in the millisecond range, which is tolerable for most operating environments.

5.4 Summary

This chapter describes the mLAN Version 2 architecture – an mLAN-based networking architecture that specifies a thin transport implementation for audio devices, and an Enabler that is responsible for providing high-level plug and word clock abstractions on behalf of these devices. The audio devices, referred to as Transporters, implement the Audio and Music data protocol and a thin interface that allows transmission and reception to be configured. Unlike mLAN Version 1, the plugs of these devices are not held within the device, but rather modelled by the Enabler. In addition to the plug abstraction layer implemented by the Enabler, it also

implements an A/M manager layer and a hardware abstraction layer that enables a plug-in module to be loaded for a particular vendor-specific Transporter implementation. The implementations of these layers are discussed in this chapter.

Several timing measurements were conducted on a Linux implementation of the mLAN Version 2 Enabler. These measurements include the time taken by the Enabler to enumerate a Transporter device, as well as to perform plug connects and disconnects. In comparison to the timing results obtained for mLAN Version 1, the mLAN Version 2 Enabler shows a remarkable speed-up of about 90.6% in performing device enumeration, 96.7% in performing plug connects, and 96.8% in performing plug disconnects, and hence motivates the need for the mLAN Version 2 architecture.

In addition to these results, mLAN Version 2 defines an architecture that enables point-to-point plug connections at the transport layer of the network. However, the *node application* interface defined by the Transporter specification, should make it possible for an Enabler to model a Transporter's host, and in so doing, provide true-end-true plug connectivity. The Otari ND-20B audio routing device is an example of an mLAN Version 2 device that implements a *node application* interface.

The next chapter describes the attempts performed on the Windows Basic Enabler to model the *node application* of a Transporter node, in addition to providing an IEEE 1394 bridge implementation. The limitations of the Basic Enabler in allowing for these enhancements are discussed.

Chapter 6

6. Further Enhancements and Limitations of the Current Enabler Specification

From the previous chapter it was established that the mLAN Version 2 Enabler provides a more efficient and effective solution to connection management. However, since its deployment and use in the music industry, it became necessary (for certain configurations) to have IEEE 1394 bridges supported by the Enabler. Such configurations may be comprised of:

- A large number of devices, where managing isochronous resources in use by transmitting devices becomes vital.
- A small number of mLAN devices that are frequently added or removed from a bus. In this case, IEEE 1394 bridges help to maintain bus reset independence between the buses that form part of the network.

A further feature required to be implemented by an mLAN Version 2 solution, was to enable true end-to-end connection management. This feature is particularly targeted for broadcast studios, where switching between audio feeds of different studios is time-critical, and also, for sound installations, in order to reduce the time taken to set up audio routings between devices. Providing this feature requires an accurate modelling of Transporter devices.

The mLAN Version 2 Enabler provides plug abstractions that only model the transport i.e. the A/M layer of mLAN Transporter devices. This usually implies that the associations between the hard-end connectors implemented by the mLAN Transporter device and its transport capabilities, have to be pre-determined or (if possible) configured using a control panel application which sometimes resides on the actual device. The actual word clock source and sample rate in use by various converters within the device is configured in a similar manner. Modelling the host area of mLAN Transporter nodes by the Enabler eliminates the need to separately pre-configure these host features. In addition to this, it also facilitates optimal use of isochronous resources by the device.

This chapter gives an indication of how IEEE 1394 Bridges and the node application of mLAN Transporter devices have been implemented by both the Windows and Linux Enablers. It highlights the inefficiencies that result from the architecture of the Basic Enabler specification, and the motivation for developing a new Enabler that adequately incorporates these features and hence provides true end-to-end connectivity.

6.1 IEEE 1394 Bridge Implementation

The architecture of IEEE 1394 bridges has been discussed in section 3.2. The first consumer bridges, NEC MX/Bridge-A, were used in the bridge implementation of both the Linux and Windows versions of the Enabler. These Enablers support across-bus plug connections by:

1. Enumerating all the active devices and buses on the network.
2. Performing the necessary bridge portal settings that permit isochronous streams to be forwarded from one bus to another.

The implementations of these operations are discussed in the subsections that follow using the Linux (Basic) Enabler as reference.

6.1.1 Network Enumeration

Consider the 3-bus network example shown in Figure 6-1. This network comprises two IEEE 1394 bridges with portals U/V and X/Y respectively, and also, four mLAN

devices labelled A, B, C and D that are distributed over the network. The IEEE 1394 nodes A and U reside on bus 0, nodes V, B, C and X reside on bus 1, and nodes Y and D reside on bus 2.

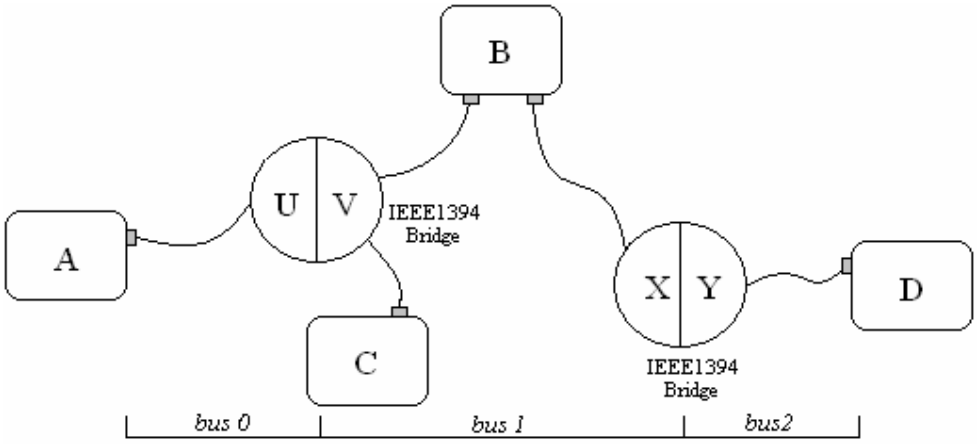


Figure 6-1: 3-bus network example

An Enabler (implemented according to the Basic Enabler specification) in addition to creating an object identifier for the IEEE 1394 PCI/PCMCIA interface card on which the network is attached, also has to create object identifiers for each IEEE 1394 bus and node present on the network. Recall from the implementation of the mLAN Version 1 Enabler (page 101) that the object identifiers (object IDs) are used to uniquely identify physical interfaces, buses or devices. The created object IDs also have to be managed appropriately in the event of a device or bus configuration change.

6.1.1.1 Creating Object Identifiers

The object IDs for the various IEEE 1394 entities (interfaces, buses and devices) can be created when the Enabler starts up, in other words when an application creates a new *C1394* object; refer to the object model shown in Figure 4-6. In creating these objects, the number of interfaces, buses and devices on each bus has to be determined, from where an appropriate object is created to uniquely identify the IEEE 1394 entity. For a one-bus environment, determining the number of interfaces and devices on the local bus requires platform-specific calls to the platform's IEEE 1394 implementation. In a multi-bus environment, the NEC Bridges have to be queried to determine the number of IEEE 1394 buses and nodes attached to each bus. Recall from the operation of NEC bridges given in section 3.2.2 that the portals of an NEC

bridge, particularly the network cycle master (NCM) and the alpha portal of a bus, implement a number of net management messages that permit a controlling application to be aware of the network topology. These net management messages include ACTIVE_BUS_ID, REMOTE_NODE_INFO and BUS_TOPOLOGY_MAP. The ACTIVE_BUS_ID message is implemented by the NCM, and gives an indication of the active buses that exist on the network. The REMOTE_NODE_INFO and BUS_TOPOLOGY_MAP messages are implemented by the alpha portal node, which respectively give information about the physical properties of each node on a bus as well as the topology of the nodes attached to a bus. The format of these messages has been described in section 3.2.2.2. Thus, by obtaining the ACTIVE_BUS_ID, REMOTE_NODE_INFO and BUS_TOPOLOGY_MAP bridge messages, an Enabler can enumerate an IEEE 1394 network and hence create object IDs for the buses and nodes on the network.

The Linux Enabler uses the memory addresses of pointers to a number of class objects to implement the object ID concept. These class objects, one defined for each interface, bus and device, hold information that describes the physical properties of the IEEE 1394 entities they represent. These physical properties include:

- The GUID and IEEE 1394 (libraw) handle assigned to a particular interface.
- The bus ID, bus generation, node ID of the alpha portal of a bus, and the topology map information pertaining to a particular bus.
- The GUID, physical ID, and virtual ID of a particular IEEE 1394 node.

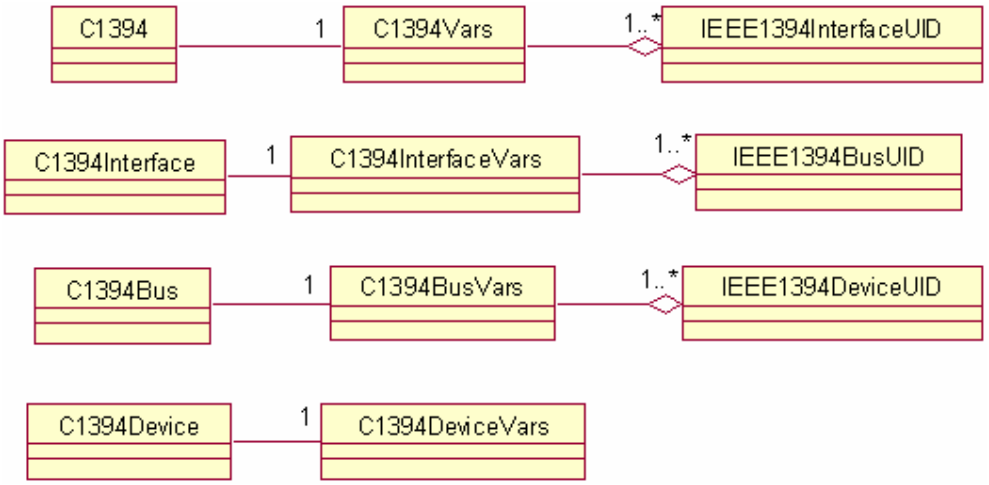


Figure 6-2: Object model of object ID implementation of the Linux (Basic) Enabler

The object model that describes the implementation of the Linux object ID model is shown in Figure 6-2 above. The classes *C1394Vars*, *C1394InterfaceVars*, *C1394BusVars* and *C1394DeviceVars* provide platform specific implementations for the respective corresponding Enabler classes – *C1394*, *C1394Interface*, *C1394Bus* and *C1394Device*. The object IDs of interfaces, buses, and devices are derived from the memory addresses of pointers to objects of the 'unique identifier' classes: *IEEE1394InterfaceUID*, *IEEE1394BusUID* and *IEEE1394DeviceUID*, respectively. This is illustrated in the figure below, using a 1394 device as an example.

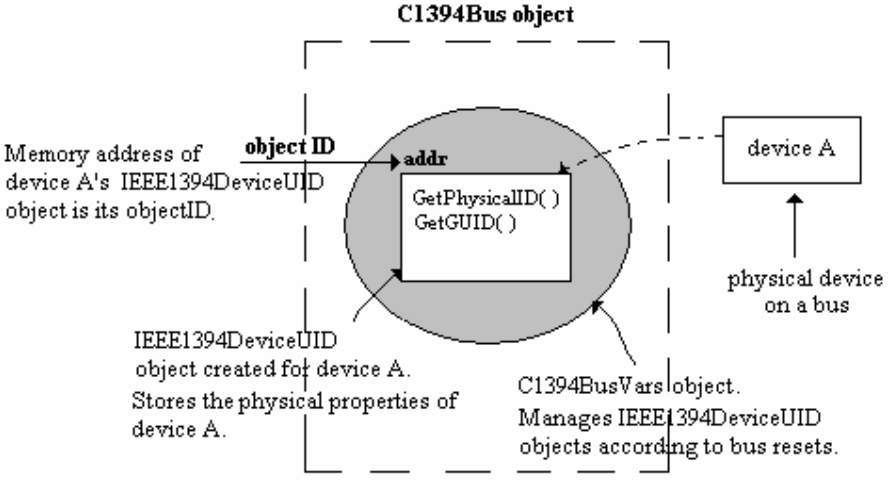


Figure 6-3: Object ID model for a 1394 device

These unique identifier classes define a number of accessor methods that allow the physical properties of the actual interface, bus, or device they represent to be set to or retrieved from the corresponding class object. The platform-specific classes: *C1394Vars*, *C1394InterfaceVars* and *C1394BusVars* contain a number of objects of the unique identifier classes from where they are managed according to the physical configuration of devices on a network.

As implied by Figure 6-2, the Linux Enabler creates and initializes the platform-specific objects and the corresponding aggregation of objects of the unique identifier classes, when objects of *C1394*, *C1394Interface* and *C1394Bus* classes are created, usually by an application. After creating a *C1394* object, the number of interface cards is determined and an *IEEE1394InterfaceUID* object created for each card. The object IDs, which are actually memory pointers to the *IEEE1394InterfaceUID*

objects, are then presented to an application from where corresponding *C1394Interface* objects are created. During the creation of a *C1394Interface* object, the entire network has to be enumerated by accessing the network management messages – ACTIVE_BUS_ID, REMOTE_NODE_INFO and BUS_TOPOLOGY_MAP to establish the number of *IEEE1394BusUID* objects to create and also to initialize these objects with the corresponding bus information. The *C1394Interface* class is configured, using a call-back routine, to always listen for these net management messages, interpret them and handle them appropriately. Figure 6-4 and Figure 6-5 shows the sequence diagrams that briefly illustrate how *IEEE1394BusUID* objects are created within a bridged environment.

It is important to mention that at this stage of the Linux Enabler implementation, the original IEEE 1394 Linux drivers and libraw library were replaced with a bridge-aware equivalent developed by Melekam Tsegaye [Tsegaye, 2002]. This version of the drivers and library, amongst other things, implements the bridge message command and response registers that enables reception of network management messages.

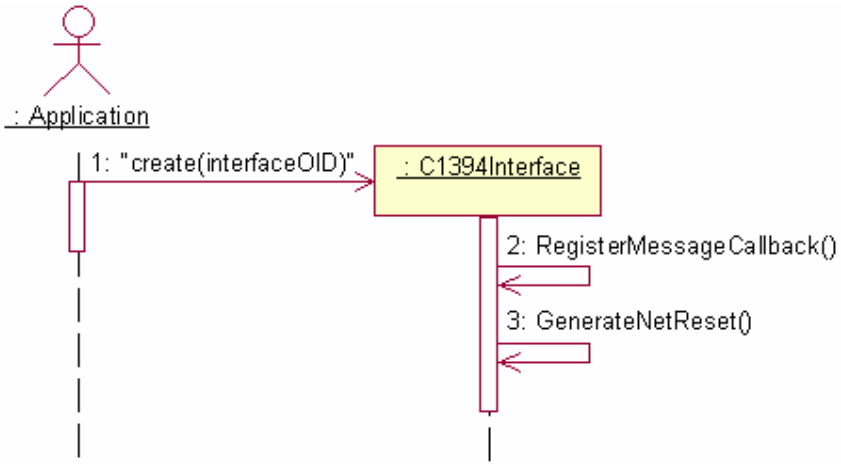


Figure 6-4: IEEE1394BusUID object creation in a multi-bus environment

Creating *IEEE1394BusUID* objects takes place in two stages. The first stage, shown in Figure 6-4, performs various initializations on the *IEEE1394Interface* object – a message call-back routine is registered to the IEEE 1394 driver, after which a net reset is generated. The generated net reset causes all the alpha portals that exist on the

network, as well as the network cycle master to broadcast their network management messages. The procedure for handling the network management messages received from the bus is described by Figure 6-5.

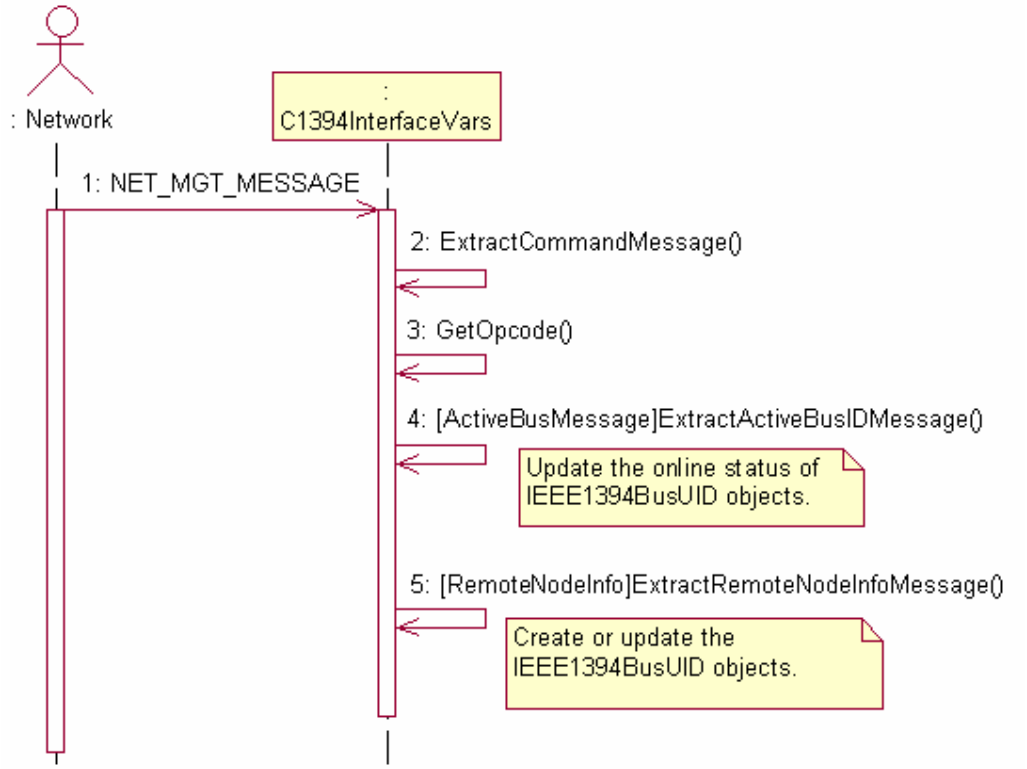


Figure 6-5: Handling network management messages

A network management message, originating from a bridge portal to the Linux Enabler is received by the *C1394InterfaceVars* class via the message call-back handler previously set in sequence 2 of Figure 6-4. From the byte data received, the command message is extracted (sequence 2) after which the opcode is examined (sequence 3) to determine the message type. If the opcode indicates an ACTIVE_BUS_ID message, the corresponding message is extracted from the command message and used to update the online status of the *IEEE1394BusUID* objects. Similarly, if the opcode indicates a REMOTE_ NODE_INFO message, the corresponding message is extracted and used in creating or updating the contents of existing *IEEE1394BusUID* objects. Recall that the ACTIVE_BUS_ID message contains information about the bus IDs of 1394 buses that are active on the network, while the REMOTE_NODE_INFO message contains information about the physical

properties of the nodes on a bus. At start-up time, no *IEEE1394BusUID* objects exist, and hence the message would result in their creation. Each *IEEE1394BusUID* object created is initialized with the corresponding bus physical properties, which also includes the physical properties of the nodes attached to the bus. Once the *IEEE1394BusUID* objects have been created, an application can then request the *C1394Interface* object for a list of bus object IDs, from which *C1394Bus* objects are created.

The bus object ID used in creating a *C1394Bus* object is a pointer to an existing *IEEE1394BusUID* object, and contains information about the bus and nodes attached to the bus. This information is used to initialize the associated *C1394BusVars* object (referring back to Figure 6-2) and also for creating *IEEE1394DeviceUID* objects to serve as device object IDs. An *IEEE1394DeviceUID* object is created for every node contained within the *IEEE1394BusUID* object, and is initialized to contain the corresponding node's (physical) information. Upon creating a *C1394Bus* object, a list of pointers to these objects (device object IDs) is presented to an application, from where appropriate 1394 device objects are created.

The object IDs created for interfaces, buses and devices have to be managed according to configuration changes that occur on the network. For example, if a 1394 device is added or removed from the network, the corresponding object ID has to be created or updated to reflect the current network configuration. The technique involved in this process is discussed in the next subsection.

6.1.1.2 Managing Object Identifiers

The Basic Enabler specification defines an object ID to uniquely identify an interface, bus or device. Hence an application that uses an object ID to create a *C1394Interface*, *C1394Bus* or *C1394Device* object for a particular 1394 entity expects the object ID to remain unchanged even if the entity is removed and subsequently added to the network. The only way for an Enabler to present the same object ID to an application for a particular 1394 entity is for it not to dispose of the object ID representation of the 1394 entity when the entity is removed from the network, but rather set a flag

which indicates that the entity is offline. This is illustrated using the example network shown in Figure 6-6.

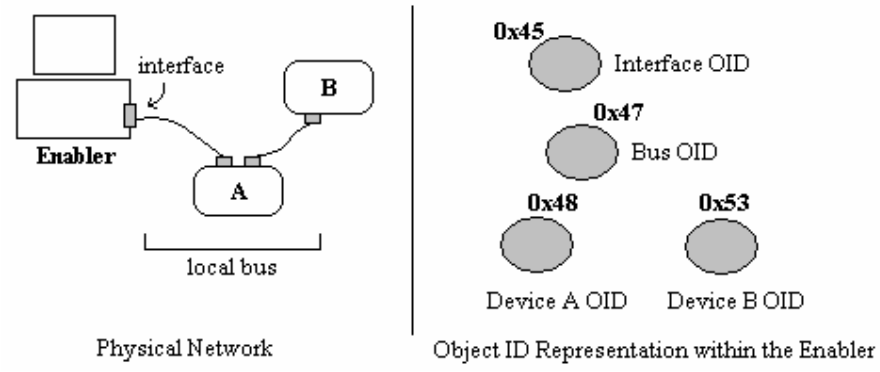


Figure 6-6: Example network illustrating object ID management

To the left of Figure 6-6, a one-bus network is shown to consist of two devices – A and B, that are attached to an Enabler. The corresponding object ID representation is shown to the right of the figure. Here, an object ID with value 0x45 is created for the Enabler's interface, an object ID with value 0x47 is created for the local bus, and also object IDs having the value 0x48 and 0x53 are created for the devices A and B respectively. If device B is now removed from the bus, the Enabler, in order to maintain the definition of object IDs, has to preserve the object ID originally created for device B, but has to somehow indicate that the device B is no longer on the bus. Hence, Figure 6-6 now becomes:

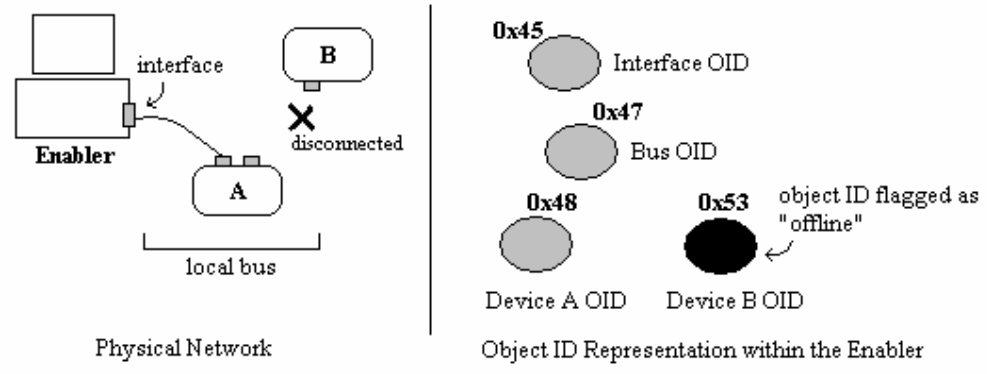


Figure 6-7: Updated object ID representation

Device B's object ID can be re-activated when device B is reconnected to the local bus of the Enabler. The object ID's of buses, in a multi-bus environment, are updated in much the same way as described for devices, in cases where an entire bus is disconnected from the network.

Configuration changes that occur on a bridged IEEE 1394 network are detected by the Linux Enabler using the resulting network management messages – ACTIVE_BUS_ID and REMOTE_NODE_INFO – that are broadcast on the network. The Enabler receives these messages, interprets them, and then proceeds to update its object IDs. The technique involved in deciphering and handling the bridge messages is described by the sequence diagram shown in Figure 6-5.

When a device is removed or added to a 1394 bus, the *IEEE1394BusUID* object corresponding to the 1394 bus on which the configuration change occurred, has to be located and then updated with the current information regarding the physical properties of the bus and the nodes on the bus. From the sequence diagram shown in Figure 6-5, the *C1394InterfaceVars* object detects the resulting REMOTE_NODE_INFO message and then locates the corresponding *IEEE1394BusUID* object. The *IEEE1394BusUID* object is located by searching for an *IEEE1394BusUID* object that contains a 1394 device GUID that matches the GUID value of at least one of the portals specified by the REMOTE_NODE_INFO message.

Recall from the previous subsection that *IEEE1394BusUID* objects, during their creation, are updated with the physical information of the nodes attached to the bus. The GUID of a portal specified by the REMOTE_NODE_INFO message, and not the bus ID, is used in searching for the corresponding *IEEE1394BusUID* object, primarily because of the volatile nature of the bus ID during configuration changes. After locating the *IEEE1394BusUID* object, the physical information of the bus and nodes held by the object is updated with the contents of the REMOTE_NODE_INFO message. This is illustrated in pseudo-code below.

```
0    IEEE1394InterfaceVars::UpdateBusUID( pRemoteNodeInfo )
1    {
2        foundBusUID = false;
3        numBusUIDs = GetBusUIDCount( ); //Get number of bus UID objects
```

```
4
5         for j: 1 to numBusUIDs && ! foundBusUID {
6           //Get a bus UID object
7           aBusUID = GetBusUID( j );
8           numPortals = GetNumPortals ( pRemoteNodeInfo );
9
10          for ixPortal: 1 to numPortals && ! foundBusUID {
11           //Check if the bus represented by the bu UID object is contained in pRemoteNodeInfo
12             aPortalGUID = GetPortalGUID (ixPortal, pRemoteNodeInfo );
13             if  (aBusUID→ContainsThisPortal( aPortalGUID ) )  foundBusUID = true;
14          }
15        }
16
17    if (foundBusUID) {
18          // If found, update the attributes of the bus UID object
19          aBusUID→UpdatePhysicalInformation( pRemoteNodeInfo )
20    }
```

Once the *IEEE1394BusUID* object is updated, the attributes of the individual *IEEE1394DeviceUID* objects that uniquely identify the 1394 nodes specified by the *IEEE1394BusUID* object, have to be also updated. These objects, as mentioned before, are managed by the *C1394BusVars* object that is associated with the *C1394Bus* object that has had its *IEEE1394BusUID* object updated. This is illustrated in the figure below.

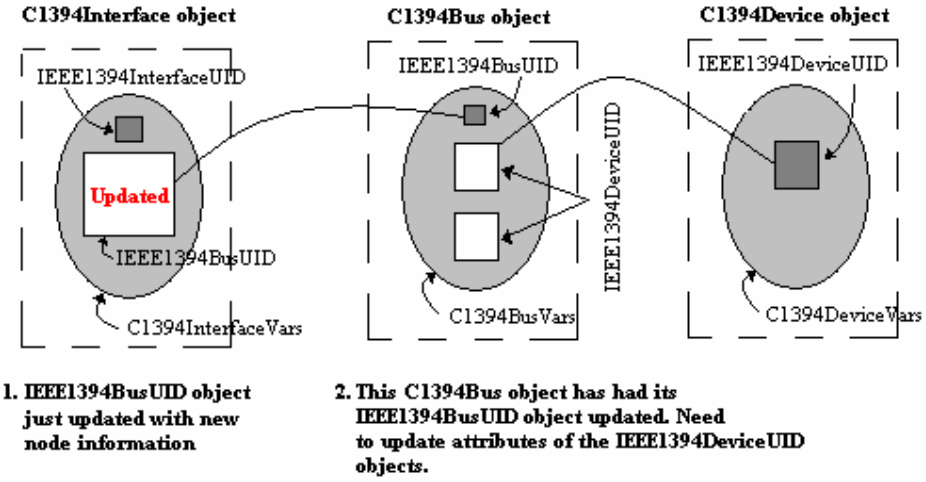


Figure 6-8: Updating information held within object IDs

In updating the *IEEE1394DeviceUID* objects, the *C1394BusVars* object detects a change in device configuration by comparing the current bus generation value held by the object, to the bus generation value held by the associated *IEEE1394BusUID* object. Since the *IEEE1394BusUID* object was updated, it would, at this point,

contain the new generation value of the bus. If there is a change in generation, the *C1394BusVars* object proceeds to update its *IEEE1394DeviceUID* objects.

The *IEEE1394DeviceUID* objects are updated by finding the updated node information held within the *IEEE1394BusUID* object that corresponds to an *IEEE1394DeviceUID* object. The node information is determined by finding a GUID value of a node information (held within the *IEEE1394BusUID* object) that matches the GUID value held by the *IEEE1394DeviceUID* object to be updated. If the node information is found within the *IEEE1394BusUID* object, the attributes of the *IEEE1394DeviceUID* object are updated to contain the updated node information. This also updates the physical ID value held by the *IEEE1394DeviceUID* object, which may have changed as a result of the configuration change.

If the *IEEE1394BusUID* object does not have any updated information for a particular *IEEE1394DeviceUID* object, it implies that the 1394 node that is identified by the *IEEE1394DeviceUID* object has been removed from the bus. In this case the *IEEE1394DeviceUID* object is flagged 'offline'. On the other hand, if the *IEEE1394BusUID* object has node information for an *IEEE1394DeviceUID* object that does not exist, it implies that a new 1394 node has been added to the bus, and hence a new *IEEE1394DeviceUID* object is created to identify it. This process is summarized in pseudo-code below.

```
0    IEEE1394BusVars::UpdateDeviceBusUID( )
1    {
2        foundDeviceUID = false;
3
4        //Get the IEEE1394BusUID object. Should contain the updated node information
5        myBusUID = GetBusUIDPtr( )
6
7        //Search for the updated node information for each IEEE1394DeviceUID object
8        for j: 1 to GetDeviceUIDCount ( ) && ! foundDeviceUID {
9
10           aDeviceUID = GetDeviceUID( j ) //Get an IEEE1394DeviceUID object
11           numDevices = myBusUID→GetNodeInformationCount( )
12
13           for ixDevice: 1 to numDevices && !foundDeviceUID {
14
15              //Find the updated node information that corresponds to aDeviceUID
16              aDeviceGUID = myBusUID→GetDeviceGUID (ixDevice);
17              if  (aDeviceUID→GetGUID ( ) == aDeviceGUID ) {
18
```

```
19                foundNodeInformation = true;
20              }
21            }
22
23            if (foundNodeInformation) { // Found node information, updating aDeviceUID
24
25              pNodeInformation = myBusUID→GetNodeInformation (ixDevice);
26              aDeviceUID→UpdateNodeInformation(pNodeInformation);
27              aDeviceUID→MarkOnline( );
28            }
29            else {
30              // Node information not found for aDeviceUID, flag offline
31
32            }
33          }
34          // Create a new IEEE1394DeviceUID object and add to current list for each
35          // node information that has no IEEE1394DeviceUID object to update
36      }
```

The above process describes the case where an IEEE 1394 node (other than a bridge portal) is added or removed from the network. When a bridge portal is inserted or detached from a network, two configuration changes occur:

- A device configuration change on the 1394 bus on which the bridge portal was added or removed,
- A bus configuration change on the entire 1394 network.

The former is handled by the Linux Enabler as described above. In handling the latter, *IEEE1394BusUID* objects have to be created or updated to reflect the state of 1394 buses on the network. *IEEE1394BusUID* objects are created for new 1394 buses added to the network, or are updated to either reflect a change in the 10-bit bus ID – resulting from the configuration change, or flagged offline for 1394 buses that have been removed from the network. Creating *IEEE1394BusUID* objects may result in the creation of subsequent *IEEE1394DeviceUID* objects. Similarly, changing the online state of an *IEEE1394BusUID* object also results in changes to the online state of the related *IEEE1394DeviceUID* objects. Applications can be notified of these configuration changes by registering appropriate call-back routines. From an application point of view, it implies that method calls to objects of *C1394Bus* and *C1394Device* are successful only if the corresponding 1394 buses and devices are online.

Another task of supporting 1394 bridging within the Enabler is to provide support for across-bus plug connections. The techniques adopted for this implementation, with regards to both the Linux and Windows (Basic) Enablers are given in the next subsection.

6.1.2 Isochronous Stream Forwarding

Forwarding isochronous streams across buses is necessary in order to enable plug connections and word clock synchronization between devices residing on different buses on a network. As mentioned in section 5.3.2.3, the stream control registers of the bridge portals along the path from the transmitting device to the receiving device have to be configured, in addition to setting up the source and destination plugs for connections. Recall that a plug connection involves copying across the isochronous channel number, sequence number and subsequence number of the source plug to the destination plug. Also remember from section 3.2.2.3 that NEC bridges implement a number of stream control registers (SCR) – one for each isochronous channel – that define configuration parameters required for across-bus forwarding. An illustration of the operation of isochronous stream forwarding is given with the example shown below.

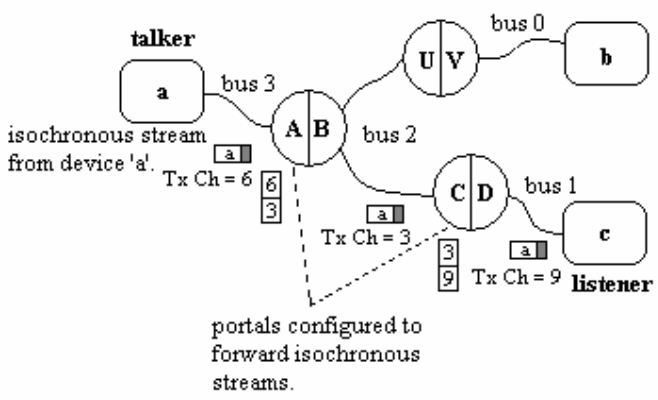


Figure 6-9: Network example illustrating isochronous stream forwarding

Figure 6-9 shows a 4-bus network comprising three devices: a, b, and c, that are interconnected using three bridges with portals labelled A, B, U, V, C and D. A source plug on the device labelled ‘a’ – the *talker,* is intended to be connected to an input plug on the device labelled ‘c’ – the *listener*. These devices reside on different buses – bus 3 and 1 respectively, and as such, require isochronous stream forwarding.

In order to accomplish this, the bridge portals along the path from device ‘a’ to ‘c’ are determined i.e. portals A and C respectively. The appropriate stream control register of each portal is then configured to establish an isochronous stream path from bus 3 to bus 1. Before configuring the stream control register of a portal, the adjacent bus of the portal is examined to determine whether an available isochronous channel and sufficient bandwidth exist to support the transfer of the stream. If this check fails on any of the buses, isochronous stream forwarding cannot occur. Referring to Figure 6-9, the adjacent bus of portal A, bus 2, is examined for availability of isochronous resources. Also the adjacent bus of portal C, bus 1, is examined. For the purpose of the illustration, assume sufficient resources exist on these buses. An isochronous stream path is then established from bus 3 to bus 1 such that the isochronous stream from device ‘a’ that is being transmitted on channel 6 is mapped onto channel 3 on bus 2, which is further mapped onto channel 9 on bus 1. This implies that the input plug on the listener to which the source plug on the talker is to be connected, should be configured to listen for data on channel 9.

The Windows and Linux Basic Enablers implement IEEE 1394 bridging by introducing a number of bridge related classes. The classes include a *C1394Bridge* class, a *C1394RoutingMap* class and a *C1394BridgeCommand* class. The *C1394Bridge* class defines methods that model the operation of NEC bridge portals, and most importantly exposes the configuration parameters of stream control registers. The *C1394RoutingMap* class defines a method that determines the bus path between any two buses. The *C1394BridgeCommand* class defines various methods that enable retrieval of net management messages from bridge portals. These classes are used collectively in various code implementations within the Enabler where across-bus isochronous communication is required. In so doing, they hide from an application the intricate techniques required to enable isochronous stream forwarding. In addition to this, applications that have been developed to work with the non-bridged Enabler (discussed in the previous chapter) remain unchanged and can interface with the bridged Enabler.

The code implementations of the Enabler where across-bus isochronous communication may be required are the methods that establish plug connections, break plug connections, and retrieve plug connections. The sequence diagrams for

these operations in a non-bridged environment have been discussed in sections 5.3.2.3, 5.3.2.4 and 5.3.2.5 respectively. The modified sequence diagram that describes plug connections in a bridged environment is shown in Figure 6-10.

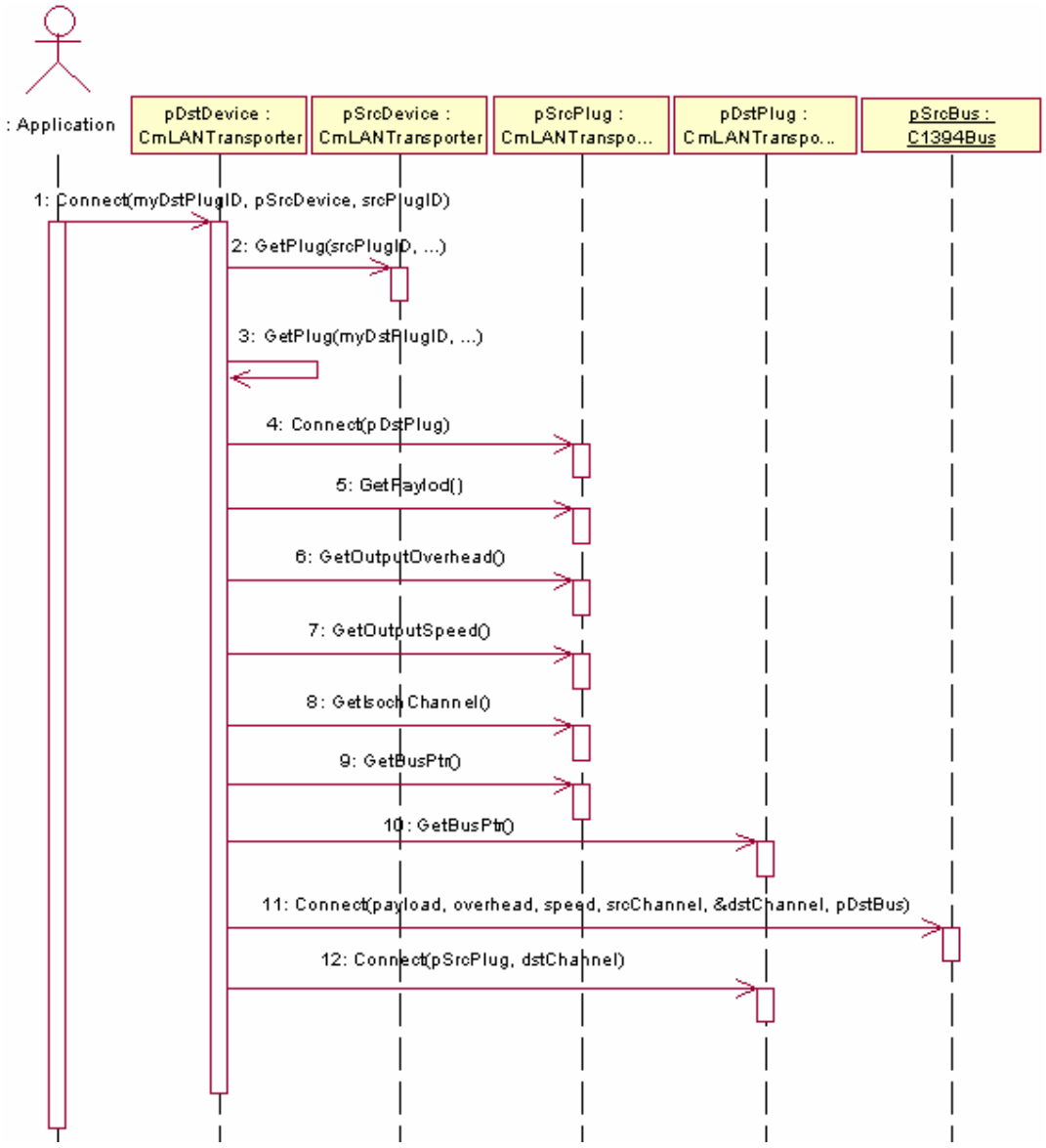


Figure 6-10: Implementing plug connections in a bridged environment

This sequence diagram differs from that discussed in section 5.3.2.3 with the inclusion of extra sequences (5, 6, 7, 8, 9, 10, and 11) that retrieve the necessary information to establish an isochronous stream path from the 1394 bus on which the transmitting device is located to the 1394 bus on which the receiving device is attached. The necessary information includes the payload of the transmitting

isochronous packet, the source device's transmission overhead, the speed and isochronous channel number of the transmitting stream. Isochronous stream forwarding is enabled in sequence 11, where:

- The portals along the path from the source to the destination are determined
- For each portal, a check is made on the adjacent bus for sufficient isochronous resources before configuring the portal's stream control registers.

Sequence 11 also returns the isochronous channel number of the target bus to which the source isochronous channel has been mapped. This channel, indicated as *dstChannel*, is then set to the input plug as part of the connection procedure.

To disconnect plugs of devices that reside on different buses, it is necessary to:

- Disable the input plug, as discussed in section 5.3.2.4. This prevents the input plug from receiving data.
- Disable any isochronous stream forwarding settings as well as release any isochronous resources that might have been in use and are now no longer in use.

The implemented sequence diagram that describes this process is shown in Figure 6-11. Sequences 4 through 8 describes how the two buses that individually contain the transmitting and receiving devices are disconnected by locating the object ID of the 1394 bus on which the transmitting device is attached, creating an object of the corresponding bus and then performing the disconnection. A pointer to the bus on which the destination device is attached is held by the object *pDstDevice* of the *CmLANTransporter* class and is retrieved as shown in sequence 7. The across-bus disconnection is performed in sequence 8, where the stream control registers of the portals along the path from the source device to the destination are disabled, and hence stop isochronous stream forwarding. Isochronous resources on the adjacent buses of the portals along the path are also released. It is worth mentioning that the stream control registers of portals are only disabled if the plug to be disconnected is the only plug receiving the forwarded stream.

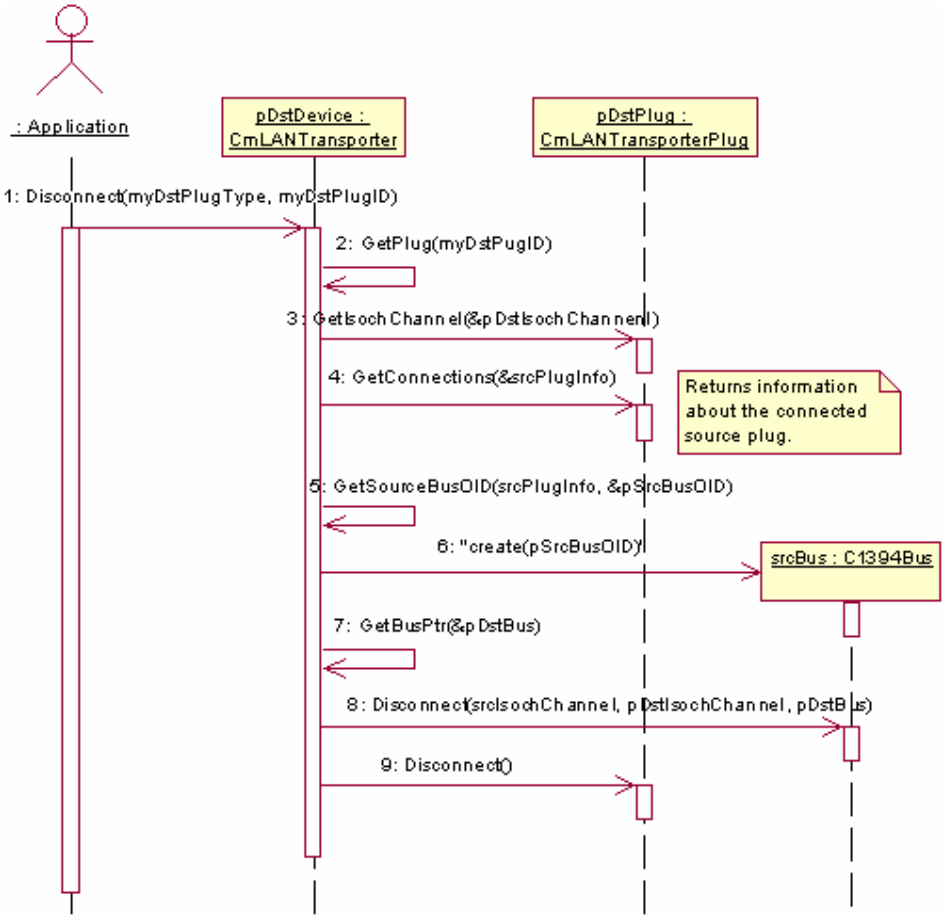


Figure 6-11: Implementing plug disconnections in a bridged environment

After disabling across-bus forwarding, the input plug is disconnected (sequence 9) to disable it from receiving data.

Plug connections in a bridged environment are retrieved in much the same way as for a single bus environment (discussed in section 5.3.2.5) – isochronous channels, sequence numbers and subsequence numbers are compared between various plugs to identify a match. The subtle difference between retrieving plug connections in a one-bus environment as compared with a multi-bus environment, is the value of the isochronous channel used in performing the channel matching. The isochronous channel used is the mapped isochronous channel (on the target bus) that corresponds to the isochronous channel of an isochronous stream that is being transmitted or received by the plug for which connections are to be retrieved. Referring to the example network shown in Figure 6-9, if plug connections are to be retrieved for the source plug on the talker device, a mapped channel value of 9 together with the

corresponding sequence number and subsequence number is used to determine matches for plugs of devices on bus 1. Similarly, if connections are to be determined for the input plug on the listener device, the corresponding mapped channel, 6, is used for matching plugs on bus 3.

The Windows and Linux Enablers each underwent a significant change in implementation from their initial versions since the need to provide support for IEEE 1394 bridging. As will be seen in a later section of this chapter (section 6.3.1.3), the Basic Enabler specification does not define an architecture that adequately supports bridging. This is reflected in the poor run-time efficiency of these Enablers. Also, the added bridging support makes implementing a Basic Enabler far more complicated, which inadvertently makes debugging and testing a cumbersome process. Before discussing these problems, the next section describes how the node application interface of mLAN Transporters is supported by the Windows Basic Enabler.

6.2 Modelling a Transporter's Host

The host of a Transporter node refers to that part of an mLAN device that houses and makes use of the resources provided by the device's Transporter implementation. Recall from section 5.1.1 that the "mLAN-1.0 Transporter Specification" document [Yamaha Corp., 2002b] specifies the requirements for Transporter implementations. The level of implementation of a Transporter's host depends on the nature of the device, and ranges from complex DSP implementations for digital mixers, to straight forward routings, to I/O hard-end plugs for break-out boxes.

The host also implements at least one word clock source, usually an internal clock, which provides a steady clock signal for the various converters implemented within the device. Other external word clock sources from ADAT, AES3, TDIF and/or BNC sources may be supported. Note that it is mandatory for the host of a Transporter node to implement an mLAN external word clock source, which is derived from the SYT timing information received by the Transporter node. Refer to section 3.3.3 for a description of how a corresponding word clock signal is generated from SYT timing information.

6.2.1 Features of a Transporter's Host

Figure 6-12 shows a typical host implementation for a simple mLAN Transporter break-out box.

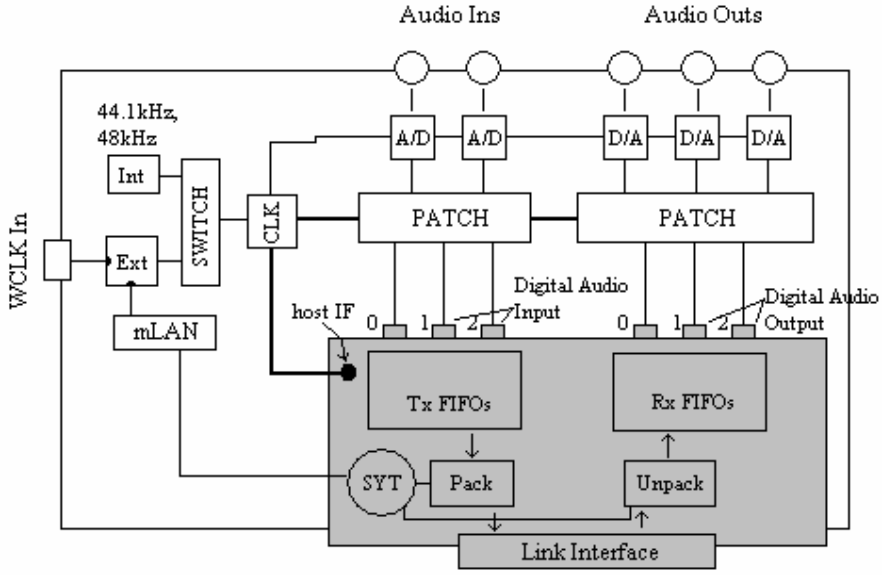


Figure 6-12: Host implementation for a simple break-out box

The audio device shown above has 6 hard-end plugs: 2 analogue mono inputs, 3 analogue mono outputs and 1 external word clock input. Analogue audio from an input plug of the device is converted to a digital representation using the A/D converters. These A/D converters are fed a clock signal from the clock source *CLK*, which can either be from an internal or external source. Each of these clock sources may support a number of sampling rates; the sampling rates that are shown to be supported are 44.1 kHz and 48 kHz. The digital audio signal, represented in a particular format – IIS, DAC-A, DAC-B, etc, is channelled to an input of the IEC 61883-6/Transporter implementation of the device. Similarly, data received from the Transporter implementation is channelled to a particular D/A converter, which is driven by a particular word clock signal, and then sent out via an audio plug. The Transporter implementation is indicated by the shaded area.

The Transporter implementation shown in the above diagram consists of an SYT block and a number of transmit and receive FIFOs that contain audio data that is to be packetized or that has been extracted from sequences of isochronous streams. The SYT block is responsible for generating SYT timing information for the transmitting

packets and also for regenerating word clock signals from the received isochronous packets to be used by the Transporter's host implementation.

As stated earlier, the level of implementation of a Transporter's host varies according to the type of the device. As far as connection management is concerned, the features that are required to be implemented include:

1. Routing data signals from the plugs of the host implementation to the mLAN transport layer of the device.
2. The flexibility of selecting between a number of word clock sources of the Transporter's host, and their corresponding sampling rates.

These features are described below.

6.2.1.1 Routing Data Signals

Data signals routed from the host implementation of a Transporter node to the mLAN transport layer can be implemented in either one of two ways:

- a fixed one-to-one association, or
- a flexible routing implementation.

The advantages of one approach over the other are discussed further on in section 6.3.2. The break-out box diagram shown in Figure 6-12 has a flexible routing implementation, where any of the plugs (of a particular direction) implemented by the Transporter's host can be freely assigned to any of the digital input/output pins of the transport layer. It is possible for the host implementation of a Transporter to have more than one layer of patching, each intended for a different purpose. However, a controlling application should be responsible for setting up the patching appropriately. Note that the plugs implemented by the Transporter's host are the true data source or end points of the corresponding transport implementation, and providing a connection management solution for these end points is the main outcome of this research.

6.2.1.2 Selecting Word Clock Sources and Sample Rates

This enables a particular word clock source and sample rate to be selected for use by a Transporter's host. The selected word clock source can either be from an external

source – including the clock signal generated from SYTs, or from the device's internal clock. Multiple clock signals may be used simultaneously within the device; the device shown in Figure 6-12 implements one word clock signal. These word clock properties must be exposed to a controlling application from where they can be configured.

In order to configure the host features mentioned above (and any other proprietary feature) a control interface has to be made available to the controlling application. The control interface, represented by *host IF* in Figure 6-12, is exposed via the Transporter node. This enables simple asynchronous or isochronous transactions to be used in configuring the Transporter's host implementation. Isochronous transactions are used in cases where MIDI control is required.

A number of mLAN Transporter devices exist that allow the features of the host implementation to be configured. This is achieved using either a control panel application on the front panel of the device itself or a desktop application. To facilitate to true end point plug connections, this control has to be implemented by the Enabler. The Otari ND-20B [Otari Inc., 2005] is one such device that implements a simple control interface that provides access to its host implementation. This interface has been modelled and implemented by the Windows Enabler. The next section describes how this implementation is achieved with an Enabler built according to the Basic Enabler specification.

6.2.2 Modelling the Host Implementation

Note that the term "node application" is be used interchangeably with the term "host implementation" in the remainder of the thesis.Yamaha Corporation developed a separate Enabler to handle Otari's ND-20B devices. This Enabler, built from the earlier versions of their Windows Enabler implementation, takes into account the node application control exposed by the ND-20Bs. The ND-20B devices are the first mLAN Transporter devices to truly capture and provide control over the interaction of the device's features to its mLAN transport layer. Before discussing the Enabler implementation, a brief overview of the ND-20B device is given in the next subsection.

6.2.2.1 Overview of the Otari ND-20B

The Otari ND-20B is a network audio distribution unit which provides a networking solution for audio signals [Otari Inc., 2005]. Multiple ND-20Bs can be connected via IEEE 1394 to form a network where audio signals are distributed, or supplied audio signals converted into different formats. Some of the features of the ND-20B, adopted from the "NB-20B Operating Manual" [Otari Inc., 2004], are listed below. Refer to the manual for a detailed description.

- High-quality A/D and D/A conversion

  The ND-20 provides high-quality A/D and D/A conversion as a stand-alone device, or combines with multiple ND-20's into a 96 kHz digital audio network.

- 32-Channel handling capacity

  Each ND-20 unit provides up to 32-channel capacity (16-channel with 96 kHz sample rate) for analogue or digital inputs or outputs.

- Expandability

  The ND-20 has four slots for I/O cards. The I/O cards include Mic Inputs (remotely controlled), Line Inputs, Line Outputs and AES Input/Output. All I/O's are capable of 96 kHz, 48 kHz, 44.1 kHz and 32 kHz sample rates, and pull-up/pull-down rates.

- Sample rate conversion

  Sample rate conversion is built into all AES input channels with individual control over each channel.

The Transport implementation of the ND-20B makes use of two mLAN-PH2 chips for A/M processing. Two chips are used in order to keep the number of transmit and receive sequences constant (32) for both low and high sampling rates. Recall from section 3.5 where connection management hardware was being discussed, that the mLAN-PH2 chip operating in a high sampling frequency mode has a maximum receive and transmit capability of 16 sequences, and 32 for a low sampling frequency mode. When used in an ND-20B device, only half of the transmit and receive capabilities of each mLAN-PH2 chip is used, thus maintaining a constant of 32.

The node application interface that facilitates flexible assignments of input and output plugs to isochronous sequences, and also allows word clock sources to be selected is summarized in Table 6-1 and Table 6-2 respectively. This specification was obtained via direct communication with Otari, Inc. Japan [Otari Inc., 2005].

Offset Address	Area	Details
:		
000 0200	PH2 Tx Sequence No. 0	Channel No.
000 0201	PH2 Tx Sequence No. 1	:
:		:
000 021F	PH2 Tx Sequence No. 31	:

[Channel No.] Ch1 of Slot A is “0”, Ch2 of Slot A is “1”... and Ch8 of Slot D is “31”

Table 6-1: Otari's ND-20B input/output plug assignments to sequences

Offset Address	Area	Bit							
		7	6	5	4	3	2	1	0
000 0009	Audio Clock Source	-	-	-	-	-	Clock Source		
000 0009	Audio Clock Fs	-	-	-	-	-	FS		
000 000A	Audio Clock Status	-	-	-	-	-		Ul	Ud

[Clock Source] 0 = SYT Slave, 1 = Internal (Xtal), 2 = External Word, 4 = Digital Input Sync
[FS] 0 = 32 kHz, 1 = 44.1 kHz, 2 = 48 kHz, 5 = 88.2 kHz, 6 = 96 kHz, 7 = Invalid
[Ul] 0 = Locked, 1 = Unlocked
[Ud] 0 = Detected, 1 = Undetected

Table 6-2: Otari’s ND-20B word clock source selection node application interface

Table 6-1 defines the interface that enables audio channels to be assigned to particular sequence positions of an isochronous stream. The audio channels are defined in terms of the available inputs and outputs of any one of the four I/O slots of the ND-20B. Table 6-2 defines the interface that facilitates configuration of a word clock source. The field *Clock Source* can either be configured to be SYT, internal, external word or digital input. The *FS* field indicates the sampling rate of the internal word clock. The *Ul* and *Ud* fields indicated whether the selected word clock source is locked and detected.

Using this interface, an application can configure the ND-20B's node application. The next subsection discusses the implementation of the Yamaha Enabler that provides support for ND-20B devices.

6.2.2.2 Yamaha's Enabler Implementation for ND-20Bs

The plug abstraction layer defined by the Basic Enabler specification, as discussed in section 5.1.2.1, describes software plugs that correspond to audio sequences or MIDI subsequences carried by an IEC 61883-6 formatted isochronous stream. In creating these plugs, the transport layer of a Transporter device is queried to retrieve the number of isochronous channels, number of sequences and number of subsequences within each sequence. This process is described in section 5.3.2.2.

Since the Basic Enabler specification defines an architecture that allows plug abstractions to be based on the transport capabilities of a Transporter device, accessed through a corresponding HAL, the assignment between node application plugs and sequences, and the word clock source selection of ND-20B devices are implemented within the HAL. The object model that describes this implementation is shown in Figure 6-13. This object model is similar to that designed for the MAP4 Transporter HAL (Figure 5-25) except for the classes *CND20ClockControl*, *CND20TxRouting*, *CND20ModuleInfo* and the corresponding classes associated with the *CND20InitializationTable* class. As implied from the diagram:

- The *CND20ClockControl* class defines methods and properties of the word clock sources implemented by the node application of the NB-20B device.
- The *CND20TxRouting* class is responsible for setting up routings between I/O slots inputs to isochronous sequences.
- The *CND20ModuleInfo* class describes the properties of the actual I/O cards inserted into the ND-20B slots.

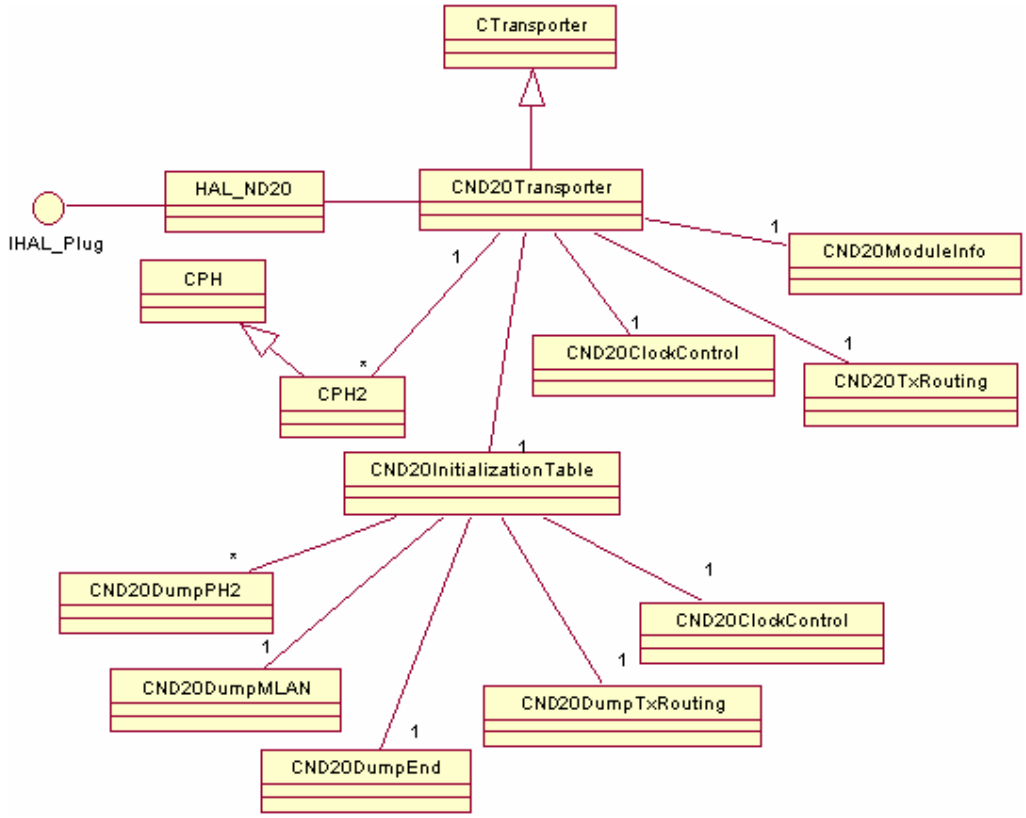


Figure 6-13: Object model of Yamaha's ND-20B HAL implementation

The implementation of node application features within the ND-20B Transporter HAL implies that the HAL functionality differs from the requirements of the Transporter HAL interface as defined by the Basic Enabler specification. Referring to the methods that constitute the Transporter HAL interface – Figure 5-12, methods like *GetMaxSequences (…)* and *GetNumSequences (…)*, which are required to retrieve the maximum number of isochronous sequences and the actual number being transmitted/received by an isochronous stream instead, returns the maximum number and current number of node application plugs of the ND-20. The maximum number of node application plugs is determined from the specification of the I/O slots of the ND-20B. Information about the specific I/O cards inserted into these slots, which in turn gives information about the number of plugs, is retrieved from read-only memory defined by the ND-20B address space.

Any sequence property that is required for connection management (see the mLAN plug connection sequence diagram, Figure 5-21) such as the *GetSequenceNumber(...)* method, first determines whether an associated isochronous sequence is allocated, and

then returns a value that either describes the sequence property or indicates an invalid sequence property. This is illustrated in the figure below:

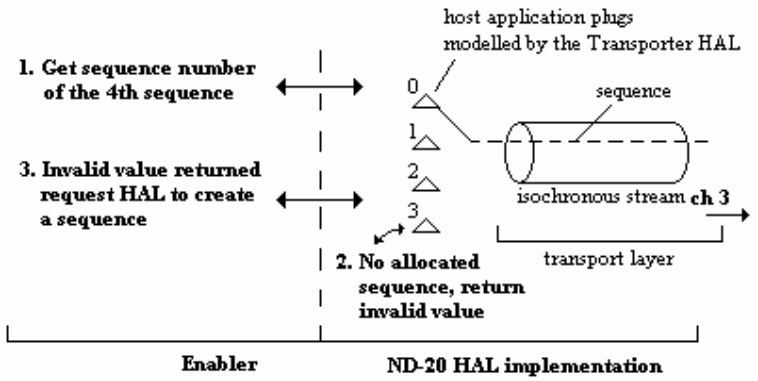


Figure 6-14: ND-20 HAL implementation of plugs with no isochronous sequences

The Enabler, upon detecting an invalid sequence property has to request the HAL to allocate an isochronous sequence; this is performed according to a dynamic sequence allocation algorithm described in a document entitled "A Brief Description of Dynamic Sequence Control" [Yamaha Corp., 2005d]. This document describes how isochronous sequences should be optimally allocated to node application plugs. A similar but different technique is discussed in section 8.2 of chapter 8 and will be revisited.

Performing dynamic sequence allocation is invoked by the Enabler using a *SetSequenceInfo(…)* method that also forms part of the Transporter HAL interface. In handling plug disconnections, the isochronous sequences allocated to the transmitting host application plugs are also dynamically released. Note that the dynamic allocation and deallocation of isochronous sequences involves some amount of bandwidth, which is either acquired or released. In cases where dynamic sequence allocation is required, if insufficient bandwidth exists, the isochronous sequence is not allocated. If the isochronous sequence is allocated successfully, the transmission routing table shown in Table 6-1 is updated to indicate the sequence number assigned to a particular host application plug.

The word clock sources of NB-20B devices are limited to two:

- word clock sourced from an internal clock, or

- word clock source from SYT synchronization.

This restriction is implied from the basic word clock support provided by the Transporter HAL interface. The methods *GetSYTSynchChannel (…)* and *SetSYTSynchChannel (…)* of the HAL interface allow a controller to retrieve and assign, to a Transporter, an isochronous channel on which the Transporter is currently receiving or is to receive SYT synchronization. Also, these methods are used to indicate to a Transporter to not synchronize on SYTs. When a valid isochronous channel number is set to the Transporter of the ND-20B using the *SetSYTSynchChannel (…)* method, the transport layer of the ND-20B is configured to receive synchronization on SYTs and the word clock source of the Transporter's host configured to use the word clock signal generated from SYTs. If an invalid channel value is set, defined by the constant *kInvalidIsochChannel* of the Enabler, the transport layer of the ND-20B is configured to not receive SYTs and the word clock source of the Transporter's host changed to use the internal clock.

The Enabler developed for the ND-20Bs has been modified in a number of places, to take into account the host application features of an NB-20B device. This is because the Basic Enabler specification defines an architecture that does not adequately support a Transporter's host application features. The next section highlights the shortcomings of the Basic Enabler architecture with respect to these Enabler advances. Some quantitative timing measurements are presented to reflect the resulting inefficiencies, motivating the need for a new Enabler architecture.

6.3 Shortcomings of the Basic Enabler

The fundamental cause of the problems encountered with the current Enabler architecture lies in its design. At a top level, the Basic Enabler Specification models an Enabler that does not truly model the behaviour of devices that form part of an mLAN network. It specifies a means by which client applications can access and modify the attributes pertaining to mLAN devices, leaving the responsibility of creating and managing Enabler class objects with a client application. As already discussed in chapter 4, the creation of class objects defined by the Basic Enabler

Specification occurs through the use of object IDs. The section that follows highlights the disadvantage of this object ID approach as way of enumerating a network.

6.3.1 Object ID Complications

The complications that arise from the use of the object ID approach of the Enabler manifest in three Enabler-related implementations:

1. Configuration change handling by an application
2. Maintaining the data of Enabler objects
3. IEEE 1394 Bridge implementation within the Enabler

The Enabler's object ID complications are discussed in terms of these Enabler-related implementations.

6.3.1.1 Configuration Change Handling by an Application

Client applications developed for the Enabler are left with the responsibility of actually modelling the topological behaviour of the devices and buses on a network. This is because the Basic Enabler Specification requires Enabler objects that directly correspond to IEEE 1394 interfaces, buses, and devices to be created by client applications. This level of flexibility provided to client applications leaves room for implementation errors.

For an application to detect and reflect the effects of bus resets on its GUI display, it must register a call back handler to a corresponding 1394 bus object. This call back handler must specify the operation to be performed when a bus reset occurs. The nature of the operation to be performed is largely dependent on the nature of the application. For most cases, device objects are created if 1394 devices are added to a bus, or subsequently disposed of if a 1394 device no longer resides on the bus. The object that models an IEEE 1394 interface, bus, or device can be queried for the online status of its corresponding device. Querying the online status of Enabler objects can occur in one of two ways. The simpler and more straight forward approach is to call the *IsOnline ( )* method of the object, which returns *true* if the actual 1394 entity represented by the object is still on the bus, otherwise it returns *false*. This is illustrated in Figure 6-15 below.

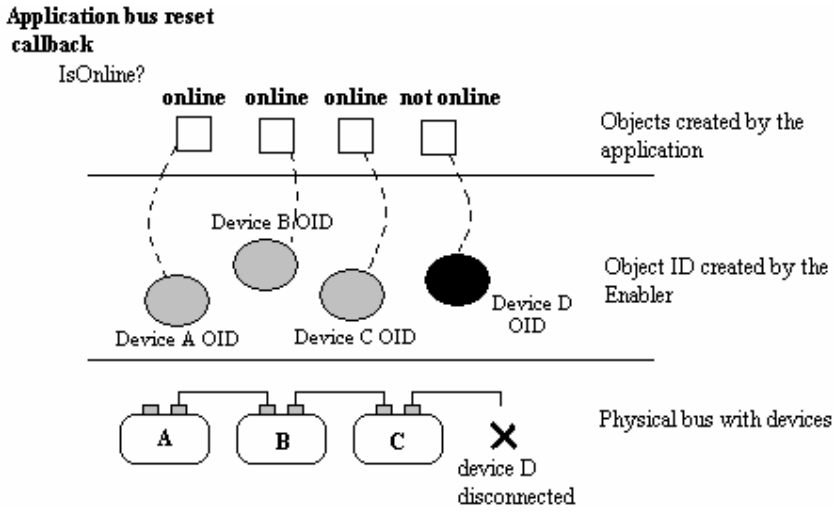


Figure 6-15: Handling configuration changes by using the *IsOnline()* method call

Device 'D' has been disconnected from the bus and the corresponding change detected and handled by the Enabler. The Enabler, in handling the configuration change, marks the object ID created for device 'D' as offline and then invokes the bus-reset call-back specified by the application. The bus-reset call-back examines the online status of each of its device objects; device A, B and C are found to be online while the object for device D returns offline.

A slightly more cumbersome approach is to obtain, from the bus object, a list of active object IDs that correspond to the devices on the bus and compare these object IDs with the object ID associated with each of the application-created mLAN Enabler device objects. From the Enabler objects, if an object ID is found to be contained in the list of active object IDs, then the 1394 device represented by the device object is still on the bus. On the other hand, if an object ID associated with an Enabler class object is not found to be included in the list of the active object IDs, it implies that the corresponding 1394 device no longer resides on the bus. Following from the example shown in Figure 6-15, the list of active object IDs returned from the Enabler is *Device A OID*, *Device B OID* and *Device C OID*. Comparing these against the object ID associated with each mLAN Enabler device object of the application, indicates that device 'D' no longer resides on the 1394 bus.

The second method is preferred because it also gives an indication of the devices that have been added to the bus. Consider Figure 6-15 with device 'D' on the bus. If an

extra device ‘E’ is now added to the bus, the state of the object IDs of the Enabler after detecting and handling the configuration change now becomes:

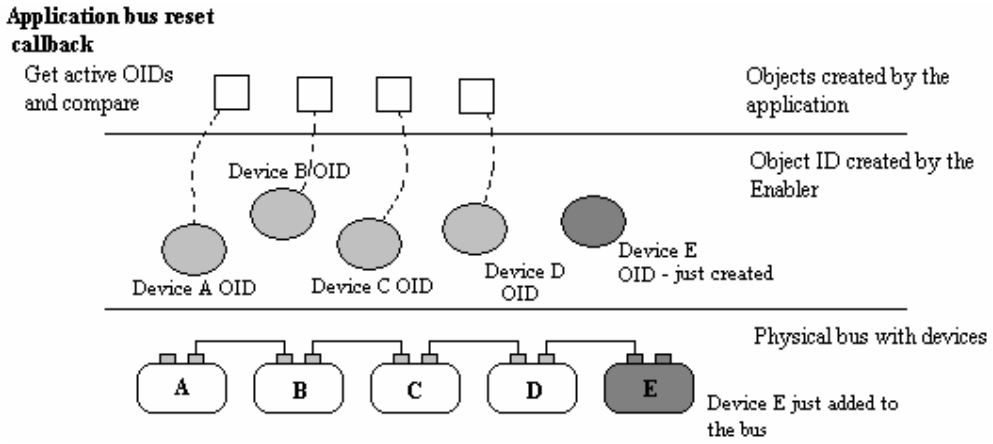


Figure 6-16: Configuration change by adding an extra 1394 device

If the application uses the second approach to handling configuration changes, the active object IDs returned from the Enabler would be *Device A OID, Device B OID, Device C OID, Device D OID* and *Device E OID*. Comparing these against the object ID associated with each Enabler class object of the application, indicates that a new device has been added to the 1394 bus, resulting in a new device object being created. Configuration change handling by an application is described here with respect to devices attached to a single bus. A similar technique is also adopted for a multi-bus network environment. The configuration change handling mechanism imposed on client applications by the Basic Enabler Specification seems fairly straight forward as described in the above paragraph. However, this mechanism can be further simplified if the objects corresponding to IEEE 1394 interfaces, buses and devices are actually created and managed within the Enabler library. Applications will only have to query the Enabler for a list of device objects, as compared to a list of object IDs, when dealing with configuration changes. This approach makes client applications easier to develop, which greatly enhances the response time of applications in dealing with user requests.

6.3.1.2 Maintaining Data Integrity of Enabler Objects

Housing Enabler objects within the Enabler library has other benefits. It partially solves the problem of misrepresenting the state of attributes cached by an Enabler

object that provides control over the capabilities of its corresponding 1394 interface, bus or device. Since the Basic Enabler Specification requires Enabler objects to be created in the memory space of an application, multiple objects can be created to reference a particular 1394 entity and, depending on how the Enabler classes are implemented, can lead to an inconsistent state representation. If Enabler objects are stored within the Enabler library, the attributes pertaining to a particular 1394 interface, bus or device are localized within the single object created to model it.

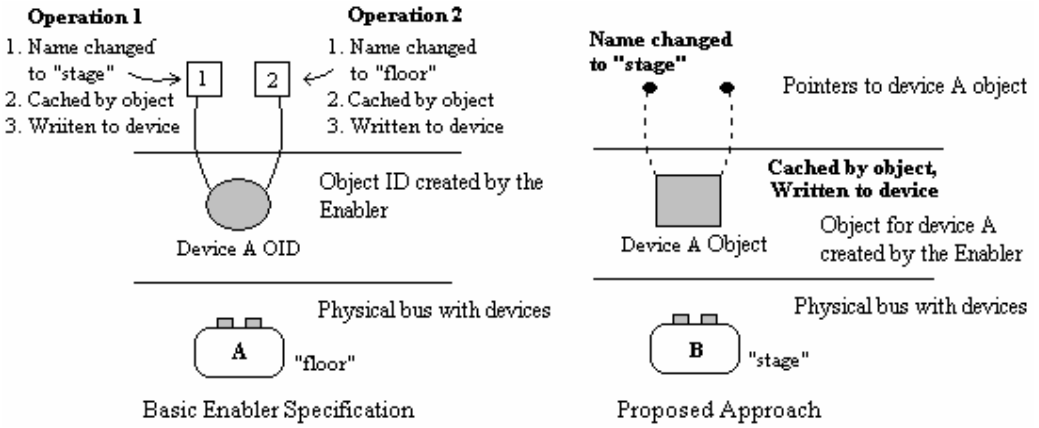


Figure 6-17: Current device object implementation and the proposed approach

Figure 6-17 above illustrates the behaviour of the attributes cached by an Enabler object with respect to the Basic Enabler Specification and the proposed approach.

The left part of Figure 6-17 shows an application interacting with a Basic Enabler Specification Enabler. In this section, the application creates two device objects 1 and 2 that model the 1394 device labelled ‘A’. From the object labelled 1, the name of the device is changed to “stage”. This change is cached by the object and also set to the 1394 device. If a similar operation, but with a different name, occurs on the object labelled 2, the textual name cached by both objects will not be the same. The name stored by the 1394 device will however reflect the outcome of the second operation. The right part of Figure 6-17 shows the proposed technique. Here, the application obtains a pointer to the one and only device object that corresponds to the 1394 device labelled ‘B’. Any changes made to device ‘B’ are performed via the object, which also updates the relevant cached values held by the object that corresponds to the device.

A data integrity problem that cannot be solved by housing Enabler class objects within an Enabler library, occurs when multiple applications that require services of the Enabler are operating simultaneously. An example of this would be making plug connections from both an mLAN patch bay application and a digital audio workstation, both requiring use of the Enabler. With respect to the Basic Enabler Specification, the applications operating simultaneously each have their own memory spaces, in which they create the Enabler library and also the associated Enabler objects. This Enabler library is created by creating an object to C1394. This is illustrated in Figure 6-18 below.

From the figure, the patch bay and digital audio workstation applications each create and use their respective Enabler objects that offer control over the 1394 devices attached to a network. Each instance of the Enabler library that is individually created by both applications also creates and manages a set of object IDs that uniquely identify 1394 interface, buses and devices on the network.

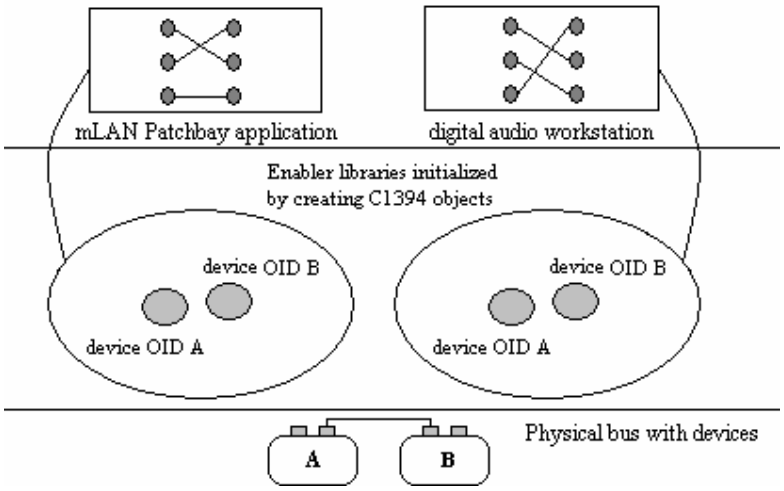


Figure 6-18: Object layout for multiple application access to the Enabler

It is obvious from Figure 6-18 that the data content held by the Enabler objects of both the patch bay and digital audio workstation application may be inconsistent if the same operation is performed sequentially on a device from both applications. In other words, a device object may have multiple state representations if operations are performed from both applications [Booch, Rumbaugh and Jacobson, 1999].

This problem has been addressed, and a suitable solution provided by Yamaha for the MAC and Windows Enablers. This involves the use of a further Enabler library that allows client applications to create objects that correspond to memory resident Enabler objects. This Enabler library exposes the same methods and functionalities defined by the Basic Enabler Specification, but differs significantly in its implementation. It makes use of COPYDATA messages to relay method calls from an application to the corresponding memory resident Enabler objects. The memory resident Enabler objects are created and initialized by an application that uses the original Enabler library. This application is required to be executed prior to any other Enabler-based applications. This approach is illustrated in Figure 6-19 (adapted from the document "mLAN Transporter Plug-In Mechanism" [Yamaha Corp., 2004b]). With this approach, a single Enabler object now represents the state of any particular device, and all applications have access to this single state.

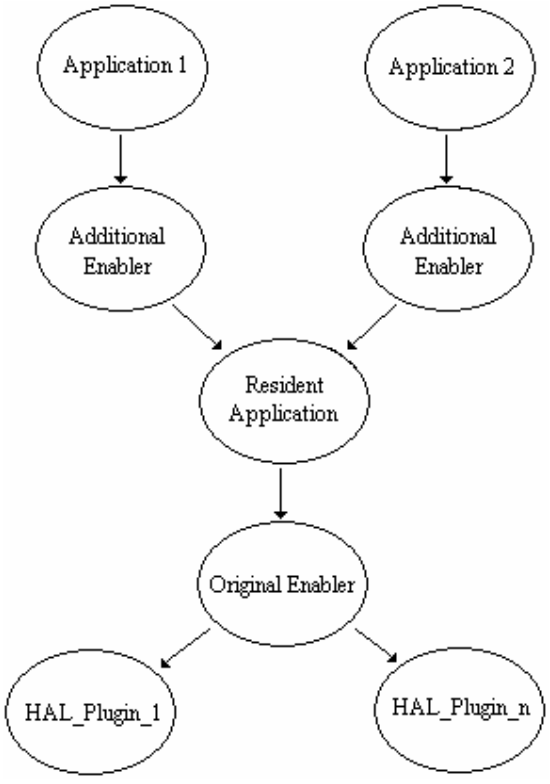


Figure 6-19: Adopted technique that allows simultaneous access by applications to an Enabler

Although this provides a solution to the problem of multiple accesses to an Enabler, it lacks elegance and simplicity. The introduction of the second Enabler library

complicates the overall Enabler architecture, as can be seen from Figure 6-19. Also, an inconvenience is introduced by the requirement to execute the memory resident application prior to any other Enabler applications. Although this problem cannot be solved by housing Enabler objects within the Enabler, an alternative technique that uses a client–server approach is preferred. In this approach, applications interact with a centralized Enabler using XML, where sessions of these various applications are managed by the Enabler. This has been discussed in a paper by Fujimori, Foss, Klinkradt and Bangay [Fujimori, Foss, Klinkradt and Bangay, 2003].

To summarize, the potential problems of the object ID approach adopted by the Basic Enabler Specification have been discussed, and the suggestion made that these problems would be eliminated if Enabler objects were created and hosted within the Enabler library. The following section discusses the bridging complications within the current Enabler, and how these complications can be eliminated if Enabler objects are created and hosted within the Enabler library.

6.3.1.3 IEEE 1394 Bridge Complications

Recall from section 6.1 that an Enabler that supports IEEE 1394 bridging has to be capable of at least enumerating all the active devices and buses on the network, as well as configuring bridge portals to allow isochronous streams to be forwarded from one bus to another.

Regarding network enumeration, the object ID mechanism of the Basic Enabler specification makes it difficult to uniquely identify an IEEE 1394 bus. This difficulty is further complicated when a bus configuration change occurs on the network. A bus object ID, as defined by the Basic Enabler specification, is required to uniquely identify a 1394 bus. Unlike IEEE 1394 nodes that have a GUID attribute that uniquely identifies them, IEEE 1394 buses do not. From the analysis conducted in this research, Enabler strategies exist that permit 1394 buses to be uniquely identified. However, the uniqueness of these strategies may be governed by the manner in which configuration changes occur on the network. Some may require a stable 1394 bus configuration that is capable of only handling device configuration changes, while others may be able to handle both 1394 bus and device configuration changes. The

latter, however, is implemented at an expense to the Enabler, which usually involve memory resources. The bus identification strategy adopted in the implementation of the Linux Enabler (section 6.1.1.2) was to use the GUID of at least one IEEE 1394 bridge portal that is attached to an IEEE 1394 bus. This technique provides a sufficient level of bus identification, which is also capable of handling both bus and device configuration changes.

Referring to section 6.1.1.2, which describes how the object IDs of the Linux Enabler are managed, it was established that for the object IDs to remain unique, the entity from which the object ID of a device is derived, was not to be disposed. This entity is usually implemented as an object that is created and managed by the Enabler. This implies that memory, once acquired by the Enabler, is not released. Workstations running the Enabler library in environments that involve large installations, or where different devices are frequently added or removed from a network over a long period of time, can end up consuming large amounts of memory. This is illustrated in Figure 6-15 and Figure 6-16, which both demonstrate the Enabler's response to configuration changes, first by removing a device and then inserting another. An object ID was created for the newly inserted device.

The Enabler architecture provided by the Basic Enabler specification makes the implementation of isochronous stream forwarding across buses highly inefficient. This is the essence of bridging support by an Enabler, and the manner in which it is implemented is very significant. Although client applications are capable of creating and using high-level objects that model 1394 bridges, the Enabler itself has to make use of IEEE 1394 bridges within its implementation of across-bus plug connection management. Refer to section 6.1.2 for a description of how this is implemented by the current Enablers.

The nature of the Basic Enabler does not permit Enabler objects to be created and retained within the Enabler library. Hence, the Enabler always has to create bridge portal objects (objects of the classes *C1394Bridge, C1394RoutingMap* and *C1394BridgeCommand*) each time bridge manipulations are to be performed. The most common bridge manipulation includes across-bus plug connections, disconnections and across-bus retrieval of plug connections. These manipulations

occur frequently between an application and the Enabler. The creation of bridge portal objects by the Enabler always requires asynchronous transactions in the form of *read* and *write* requests to be issued to the corresponding IEEE 1394 bridge portal. This overhead affects the speed performance of the Enabler, and is more noticeable in large multi-bus environments.

A number of timing measurements have been conducted on the bridged Linux Basic Enabler to demonstrate its performance. The procedure and outcomes of these measurements are discussed in the next subsection.

6.3.1.4 Performance Tests Measurements

A number of measurements were conducted to determine the speed at which an application using the bridged Linux Basic Enabler enumerates a number of bridged networks. Timing measurements were also conducted to determine the speed at which the Basic Enabler handles a number of across-bus plug operations. The results of these measurements are described below.

Speed of Enumerating a Bridged Network

The speed of network enumeration was measured for the network topologies described by Figure 6-20.

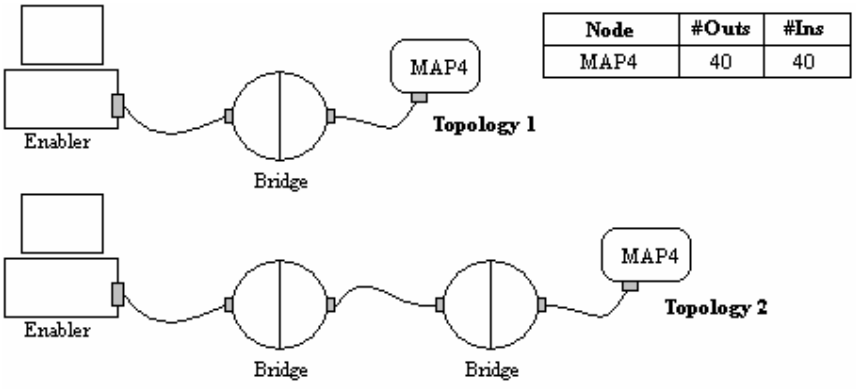


Figure 6-20: Network topologies used in measuring network enumeration

The resulting timing measurements are given in Table 6-3 below.

	Timing Measurements [seconds]			
Topology	**1st**	**2nd**	**3rd**	**Average**
1	5.80	5.80	5.84	*5.81*
2	5.94	5.93	5.93	*5.93*

Table 6-3: Timing measurements for bridged network enumeration

Note that the MAP4 Transporter used in both topologies had been configured such that 40 input plugs and 40 output plugs were enumerated by the Enabler.

The average speed at which an application enumerates a 1-bridged network with one MAP4 Transporter and a 2-bridged network with one MAP4 Transporter are 5.81 seconds and 5.93 seconds respectively. From these results, it is observed that an extra 0.12 seconds is introduced with the addition of the second bridge in "Topology 2". This value can be viewed as the extra processing time added to the enumeration speed to handle an IEEE 1394 bridge. The small value of this processing time can be explained in terms of the technique adopted by the Linux Basic Enabler to enumerate object IDs for IEEE 1394 interfaces, buses and 1394 nodes on the network. Recall from section 6.1.1.1 (Figure 6-4 and Figure 6-5) that the Enabler builds object IDs from network management messages generated from a net reset. The net reset is invoked by the Enabler.

Also recall from section 5.3.5.1 that the average time used by the Enabler to enumerate a MAP4 Transporter node is about 0.32 seconds. The addition of a bridging capability to the Linux Basic Enabler adds a constant overhead of about 5.37 seconds to the network enumeration speed. This value is calculated using the formula below with reference to "Topology 1":

$$E_k = E_{tt} - (N_b \times B_t + N_T \times T_t) \quad (9)$$

where the variables are defined as follows:

E_k The constant overhead added by the Enabler

E_{tt} The total time required to enumerate a given topology

N_b The number of bridges on the network

B_t The extra processing time to handle an IEEE 1394 bridge (= 0.12 secs)

N_T The number of Transporters on the network (configured for 40 inputs and 40 outputs)

T_t The time required to enumerate a single Transporter (= 0.32 secs)

Hence using equation 9 and the timing results of "Topology 1":

$$E_k = 5.81 - (1 \times 0.12 + 1 \times 0.32)$$
$$= 5.37$$

This constant overhead is required by the Enabler to process the network management messages and also to handle race conditions of these messages, since a number of these messages may be broadcast on the bus.

Timing measurements for a number of across-bus plug operations were conducted. These are described in the next paragraph.

Speed of Performing Across-Bus Plug Operations

A number of across-bus plug operations – connections, disconnections, and retrieval of plug connections – were timed for two different bridged network topologies, to determine the speed at which the bridged Linux Basic Enabler can perform these. Figure 6-21 illustrates the configuration of the topologies used.

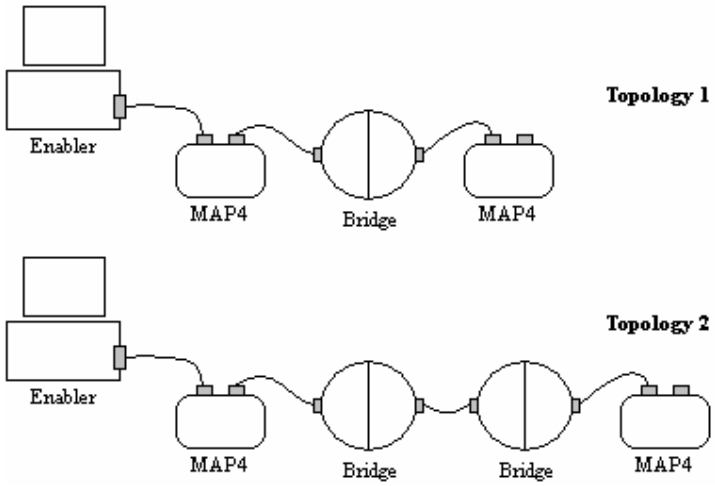


Figure 6-21: Network topologies used in timing across-bus plug operations

The results of the timing measurements for each topology are given in the tables below. Note that source plugs were used in timing the "Get Connections" routine and these plugs were configured not to have any target connections, as in the case of the previous connection retrieval measurements.

	Timing Measurements using "Topology 1"		
	Connections [s]	**Disconnections [s]**	**Get Connections[s]**
1st Sixteen Operations	2.59	3.79	1.04
2nd Sixteen Operations	2.63	3.80	1.05
3rd Sixteen Operations	2.55	3.79	1.04
Average of 48 Operations	*0.16*	*0.24*	*0.07*

Table 6-4: Timing measurements for plug operations performed using "Topology 1"

	Timing Measurements using "Topology 2"		
	Connections [s]	**Disconnections [s]**	**Get Connections[s]**
1st Sixteen Operations	3.88	2.16	3.11
2nd Sixteen Operations	3.90	2.14	3.10
3rd Sixteen Operations	3.99	2.14	3.10
Average of 48 Operations	*0.25*	*0.13*	*0.19*

Table 6-5: Timing measurements for plug operations performed using "Topology 2"

From the timing measurements taken for "Topology 1", it was observed that the Enabler took 160 milliseconds and 240 milliseconds respectively to connect and disconnect plugs. Comparing these values to those obtained for a single bus environment in Table 5-4, connections with the Linux Enabler in a bridged environment are 2 times slower, and disconnections 4.8 times slower. Also, the speed at which plug connections are retrieved is 3.5 times slower in a bridged environment as compared with a single bus environment.

The timing results for "Topology 2" shows an increase in the time the Basic Enabler takes in establishing connections and retrieving plug connections, but a decrease in the time taken for breaking plugs connections. The decrease in time is an unexpected result and cannot be immediately explained.

The speed of across-bus plug operations could be further reduced if an Enabler created and hosted bridge portal objects. Via these objects, it could access cache information, instead of performing direct bus read and write transactions as is currently happening.

In addition to these timing measurements, the bridged Linux Enabler was found to be frequently unstable, and at times took a much longer time than the non-bridging Basic Enabler to recover from network configuration changes. Also, at times, it took longer to perform any bus transactions resulting from network enumeration or plug operations. This can be explained by the Enabler's inefficient handling of the race conditions of the network management messages broadcast on the network, which eventually results in mismanagement of the object IDs that uniquely identify interfaces, buses and device on the network.

6.3.2 Modelling mLAN Transporter devices

It was mentioned in section 6.2 that the architecture of the Basic Enabler specification is not sufficient to adequately model the features of a Transporter's host implementation. It was also mentioned that there are a variety host features that can be implemented by a device. However, the features that are required as far as plug connection management is concerned are the ability to assign plugs of the node application of a Transporter to inputs and outputs of the mLAN transport layer, as well as the capability to select a particular word clock source for use.

Assigning node application plugs to inputs and outputs of the transport layer can either be implemented as fixed one-to-one associations, or as a flexible routing. In either case, the node application of a Transporter is capable of managing the use of isochronous resources required for transmissions. Most of the mLAN Transporter devices on the consumer market employ some form of strategy to allocate the required isochronous resources for transmission of audio/MIDI data from their node application plugs. A Transporter control panel that provides direct access to the transport layer of the device is typically used for this purpose. This technique has a number of disadvantages; the allocated resources may be above or below the actual requirements. This implies that the Transporter control panel would be used each time

isochronous resources are required to be allocated or de-allocated. Another significant disadvantage is that the isochronous resources that are available for transmissions on an IEEE 1394 bus can easily be consumed if non-optimal resources are pre-allocated. This directly affects the number of simultaneous isochronous transmissions on a 1394 bus, and is therefore not viable in large sound installations. As an example, an IEEE 1394 bus can safely permit a maximum of 4 isochronous streams of data, each carrying 32 sequences of audio, at a speed of 400 Mbps, and at a sampling rate of 48 kHz. This amounts to allocating at most 4 distinct isochronous channels and a total bandwidth of 0x10D0 bandwidth units, which is just below the available maximum of 0x1333 bandwidth units.

For devices that implement fixed one-to-one associations, the node application plugs are assigned permanent sequence, or sub-sequence positions (in the case of MIDI) within isochronous streams. This further implies that if the number of sequences of an isochronous stream (and hence the associated bandwidth) is reduced, some of the logical channels attached to the node application plugs would be lost. This is illustrated in the figure below:

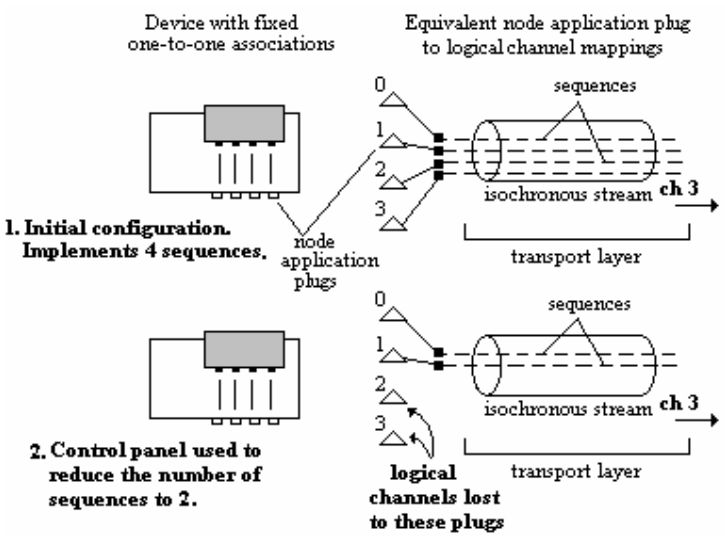


Figure 6-22: Illustrating the loss of logical channels for fixed one-to-one associations

A device that implements a flexible patching mechanism has the advantage of having data signals from its node application rerouted to any available inputs and outputs of its transport layer. This usually involves the use of a separate controlling application

either running on the device or on a workstation. This dynamic behaviour of rerouting data signals from the node application of a device to its transport layer is what has to be captured by an Enabler. This greatly improves plug connectivity amongst devices and in so doing, efficiently managing isochronous resources.

The node application also implements a number of word clock sources that have to be modelled by the Enabler. It should allow a particular clock source, internal or external, to be selected by an application, and where possible, also allow the sampling rates to be changed. Most importantly, it should allow master/slave synchronization settings to be established between mLAN devices. The Basic Enabler specification does not accurately model the behaviour of word clock synchronization. To an extent, it permits Master/Slave relationships and sample rate changes to be made. However, it lacks in its determination of the master or slave capability of a device and also fails in propagating changes made on a master device to all its associated synchronized slaves. This lack of modelling by an Enabler pushes the responsibility of true word clock synchronization modelling to an application, which further increases the complexity in developing Enabler-based applications. It is necessary that this has to be modelled by an Enabler in order to provide proper management of word clock synchronization.

Section 6.2.2.2 saw the current Enabler solution to providing node application support for Otari ND-20B devices. This was implemented within the ND-20B HAL, where some methods like *GetNumSequences (…)* returned information about the host application instead of properties of the isochronous streams. The node application word clock source selection, limited to internal clock and external mLAN, is also implemented via the *SetSYTSynchChannel (…)* method. A customized Enabler was developed especially to interact with ND-20B devices. This Enabler does not interact with other types of mLAN devices and hence does not provide a common connection management solution for all mLAN devices. Also, the implementation of node application features and IEEE 1394 bridging, does not allow full use of the node application and bridging capabilities. An Enabler that adequately addresses these problems has to be developed. The design and implementation of one such Enabler is discussed in the subsequent chapters.

6.4 Summary

This chapter critically analyzes the current Enabler specification, highlighting its limitations that do not allow for an efficient true end-to-end connection management implementation, and also, an efficient IEEE 1394 bridge implementation. Two significant limitations were raised as a result of the analysis. These are:

- The fact that the IEEE 1394 bridge implementation is made difficult and inefficient because the Basic Enabler specification is defined to allow Enabler-related objects to be created and managed by an application instead of within an Enabler.
- An inadequate capability to model the *node application* implementation of a Transporter node, which is what is required to enable end-to-end connection management.

The effects of these limitations is confirmed in a number of performance tests done on a Basic Enabler implementation that provides 1394 bridging support, as well as providing access to the node application of the Otari ND-20B devices. It is shown that although there is not a significant difference in the timing results of the bridged Enabler when compared to the non-bridged Enabler, the speed of across-bus plug operations could be further reduced if an Enabler created and hosted device objects.

Design level innovations of a new Enabler design that serves to address these limitations and offer improved performance are described in the next chapter.

Chapter 7

7. Design Level Innovations to Enable End-to-End Connectivity

The Transporter/Enabler concept was discussed extensively in the previous chapter. In this model an Enabler node is required to model and facilitate plug connections between mLAN Transporter devices interconnected using the IEEE 1394 standard. The plugs modelled by the Enabler correspond to high-level abstractions of logical channels contained within isochronous streams. Software Enablers implemented according to the Basic Enabler specification have been developed to allow for such plug connections. The Enablers implemented for the Linux and Windows platforms were extended to provide support for IEEE 1394 bridging. In addition to this, a customized Enabler that is capable of interacting with the host application of Otari's ND-20B devices was also developed. As was later discovered in section 6.3 of chapter 6, the Basic Enabler architecture does not congruently define a structure that properly models IEEE 1394 bridges and the node application features of Transporter devices; the need to design an Enabler that achieves these goals is required. This chapter focuses on:

- the design concepts of such an Enabler, and
- the effect a new design has on current Transporter HALs, and
- how these HALs should be extended to provide support for a Transporter's node application.

It is important to mention that the plural node architecture, discussed in chapter 5, allows a number of different Enabler implementations to exist, while the Transporter HAL implementations, and hence Transporter HAL interface to these Enablers remains constant. The Enabler design described in this chapter highlights concepts for an example Enabler, and should not be viewed as the 'only' Enabler design.

7.1 Design Concepts

After considering the limitations of the Basic Enabler specification with regard to handling true end-to-end plug connectivity and IEEE 1394 bridging (as discussed in section 6.3), it became apparent that a new Enabler architecture had to be developed. This section gives an overview of how the problems encountered with the Basic Enabler specification are addressed, leading to the design of an enhanced Enabler.

7.1.1 Modelling IEEE 1394 Devices and Buses

Two main problems were encountered with the Bridge implementation of the Basic Enabler specification. The first problem was related to the object ID implementation of the Enabler and the difficulties it introduced in uniquely identifying an IEEE 1394 Bus. The second problem was related to the decreased speed of operation that was introduced to the Enabler as a result of the 1394 Bridge implementation. These problems were discussed in section 6.3.1. It was also mentioned that these problems could be resolved if Enabler objects are created and managed within the Enabler. With this approach, the Enabler, after being initialized by an application, will enumerate and create the relevant objects for the buses and devices present on the network. Client applications, instead of creating Enabler objects, will now have to request these objects from the Enabler library.

This new approach, in addition to making applications a lot easier to develop, greatly improves the efficiency of the Enabler. In the event of a configuration change, the Enabler models the true state of the network by deleting any objects corresponding to IEEE 1394 buses and devices that are no longer on the network. Similarly, the Enabler creates objects for 1394 buses or devices added to the network.

The implementation of bridging is greatly simplified by this approach. In performing bridge manipulations, the Enabler library only has to retrieve the bridge portal objects that correspond to 1394 bridge portals that are to be modified or accessed. Recall that with the Basic Enabler specification, bridge portal objects were always required to be created whenever bridge manipulations had to be performed, and this always resulted in bus reads and writes. The speed performance of the Enabler in a multi-bus environment should be greatly enhanced with this new approach. This is confirmed in chapter 9 by a number of performance measurements done on the corresponding Enabler implementation.

7.1.2 Simultaneous Application Interaction

In section 6.3.1.2 it was mentioned that the Basic Enabler specification permits applications to create more than one Enabler object to reference a particular IEEE 1394 interface, device or bus. This potentially leads to inconsistencies in the state representation of the different Enabler objects referencing each of these IEEE 1394 entities. This problem is also resolved by creating the Enabler objects within the Enabler library itself. Since only one device or bus object exists for a particular IEEE 1394 device or bus, the integrity of the cached attributes of the device or bus objects are consistent, even though they may be modified several times by an application.

Another version of the state inconsistency problem of the Basic Enabler specification occurs when multiple applications access the Enabler simultaneously. As mentioned in section 6.3.1.2, the Windows and Macintosh Enablers make use of a memory resident application to model the sates of interfaces, devices and buses. An alternative approach to handling simultaneous application interaction with an Enabler is to make use of XML Remote Procedure Calls. This technique is based on a Client-Server approach, in which the Enabler, together with an XML parser, constitutes an mLAN Connection Management Server (mCMS). Enabler-based applications, are referred to as clients. These clients implement an mCMS stub, which is responsible for communicating with the mLAN Connection Management Server using the appropriate protocol. An XML based mLAN Inter-Application Protocol (mIAP) [Yamaha Corp., 2004c] has recently been developed by Yamaha's London-based

R&D, to facilitate the client/server concept. The client/server architecture is described in Figure 7-1.

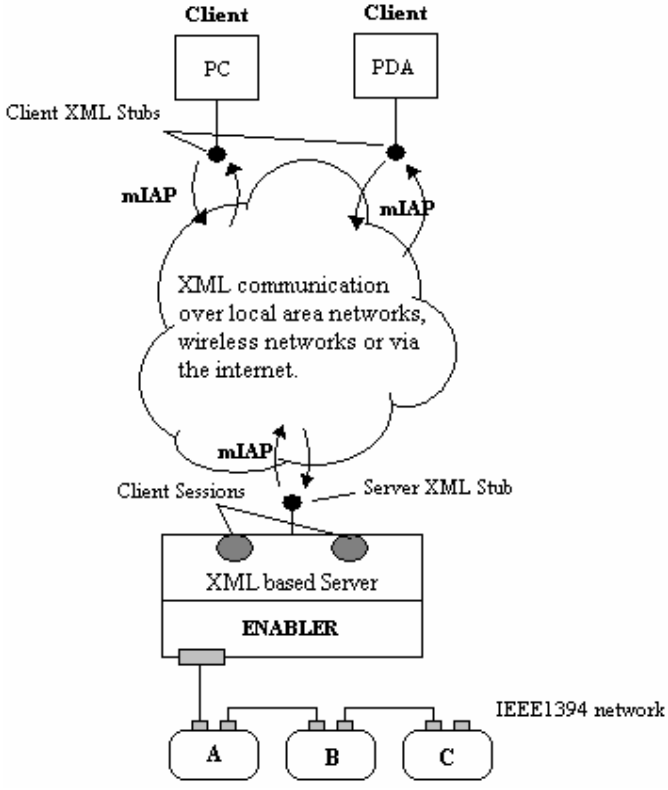


Figure 7-1 : Simultaneous Enabler access based on an XML based Client/Server model

From the figure, the client applications individually communicate with the mLAN server using mIAP. The mLAN server, which provides a central repository of Enabler objects, is responsible for managing a number of client requests for the attributes and services of these objects. The implementation of the client/server concept falls outside the scope of this research and is not discussed any further.

7.1.3 Node Application/Node Controller Concept

The Basic Enabler specification defines an architecture that does not take into account the operation and behaviour of a Transporter's host implementation. In section 6.2, it was mentioned that for true end-to-end plug connectivity to be achieved, it is necessary for an Enabler to model a Transporter's host. The mLAN Transporter specification, as discussed in section 5.1.1.2, specifies an optional Transporter implementation – the *node application interface*. This optional implementation provides the host system of a Transporter node with inbound/outbound asynchronous

packet bridging services. In view of this optional requirement, it is necessary to define an Enabler architecture that provides support for mLAN Transporter devices with a node application interface implementation. The high-level implementation of this interface by the new Enabler architecture is referred to as the *node application component*. The implementation of the original Transporter HAL interface that provides access to the IEC 61883-6 implementation of mLAN Transporter devices, is referred to as the *node controller component*. This is illustrated in Figure 7-2.

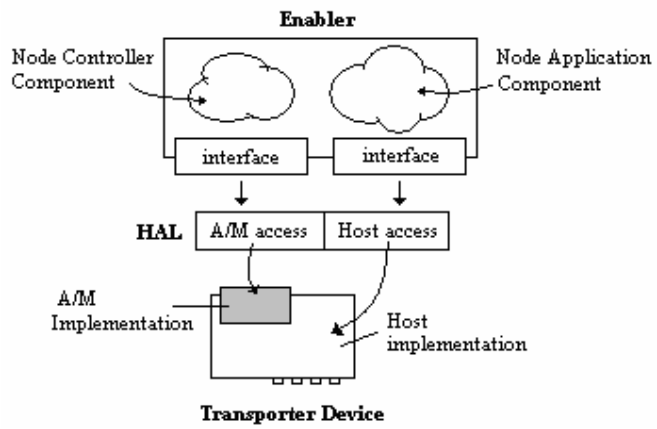


Figure 7-2: Enabler's node controller/node application concept

If a Transporter node implements a node application interface, an Enabler should be capable of accessing the node application of the corresponding mLAN Transporter device via a HAL plug-in. The actual node application implementation of a device is specified by its vendor and largely depends on the nature of the device. For example, the node application component of a digital mixer, which requires control faders and DSP algorithms to be implemented, will be different from that of a break-out box where volume control may be implemented. However, as discussed in section 6.2.1, the node application interface implemented by the Transporter node must be capable of at least accessing the attributes and properties of audio and MIDI plugs as well as all the word clock sources implemented within the host application. It should be noted however, that the node application interface implemented by a Transporter node, similar to its Transporter control interface, should also expose low-level registers to a controller. These registers can be used to gain total control over and be responsible for the proper configuration of the Transporter's host implementation.

Access to the low-level registers exposed by a Transporter's node application interface can be implemented within a HAL module. This HAL module implements an interface referred to as the *node application HAL API*, which declares various methods that allow the Enabler to interact uniformly with different node application interface implementations of various vendor Transporter nodes. The proposed *node application HAL API* methods are summarized in Table 7-1. The design structure of the node application component of the Enabler will be discussed in section 7.2.2.

Method	Description
GetVersion ( )	Returns the version revision of the firmware software that implements the Transporter's node application interface.
GetWordClockSource ( )	Returns the word clock source currently in use.
SetWordClockSource ( )	Sets a word clock source to be used
GetWordClockSourceList ( )	Returns a list of supported word clock sources
GetWordClockSampleRateList ( )	Returns a list of sampling rates supported by a word clock source
GetWordClockStatus ( )	Returns the synchronization status of the currently selected word clock source.
GetWordClockSampleRate ( )	Returns the sample rate of the currently selected word clock source.
SetWordClockSampleRate ( )	Sets the sample rate of the currently selected word clock.
GetClockSourceStr ( )	Retrieves the textual description of the currently selected clock
GetPlugTypeList ( )	Returns a list of the supported plug types.
GetNumPlugs ( )	Returns the number of plugs of a specific type.
GetPlugName ( )	Returns the textual name of a plug.
ClearPlugConfigurations ( )	Clears the logical channel association of a host application plug.
GetPlugConfiguration ( )	Returns the logical channel association of a host application plug.
SetPlugConfiguration ( )	Sets the logical channel association of a host application plug.

Table 7-1: Node Application HAL API methods defined by the Enabler

The Transporter HAL API originally defined by the Basic Enabler specification is maintained and used by this new Enabler architecture. This API, now referred to as the *node controller HAL API*, declares various methods that allow access and

modification of the transport layer implemented by a Transporter node. The main motivation for maintaining the use of the original Transporter HAL API is to provide backward compatibility with the existing HAL implementations of the current mLAN Transporter devices.

The Enabler, in order to provide backward compatibility with existing Transporter HAL implementations, has to treat the implementation of the node controller component for mLAN Transporter devices independently of the corresponding node application component. This ensures that devices having Transporter plug-ins with no access to their node application can interoperate with plug-ins that do have node application access. However, if a node application HAL implementation exists for a particular mLAN Transporter device, the node application component implementation that is modelled by the Enabler is exposed to client applications. This node application implementation makes use of the underlying node controller component as the medium of communication to access the configuration parameters of the transport layer of the Transporter devices. This is illustrated in Figure 7-3.

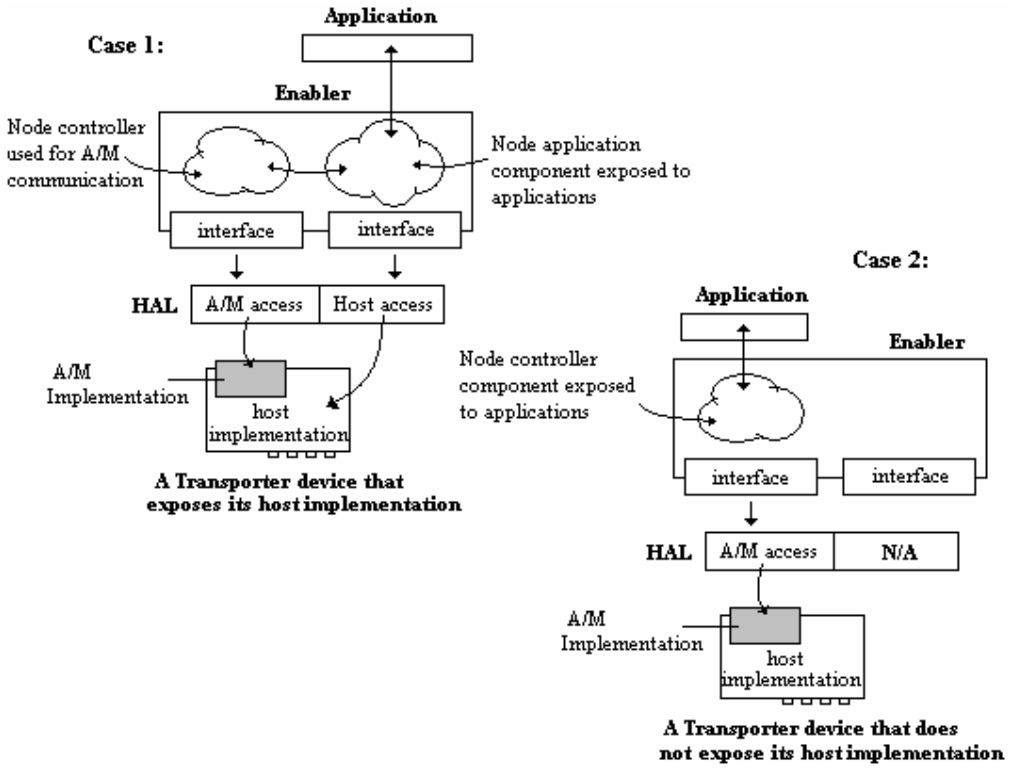


Figure 7-3: Enabler's node application/node controller component interaction

From the figure, “Case 1” shows a Transporter device that exposes its host implementation, and hence a HAL that allows direct access to it. The Enabler models for this device both a node controller component and a node application component, and while exposing the node application component to applications, it makes use of the node controller component for handling communications with the A/M implementation of the device. In “Case 2”, a Transporter device that does not expose its host implementation is shown. In this case, the Enabler only models the node controller component for the device and presents this component to applications. The Enabler should be capable of providing interoperability between devices that expose their node applications and devices that do not.

The node controller component of the new Enabler is modelled in much the same way as the Basic Enabler. In order for this to be modelled independently of the node application, the Enabler has to provide abstractions of audio and MIDI plugs as well as word clock synchronization for the transport layer of the Transporter device. These audio and MIDI plugs provide an abstraction of the connection behaviour of monaural channels of audio or “cables” of MIDI carried via logical sequences implemented within isochronous streams. The word clock synchronization implementation of the node controller component is responsible for establishing master/slave word clock settings as well as allowing sample rate changes to be made at the transport layer level of an mLAN Transporter device. Modifications made using the node controller component of the Enabler to the corresponding transport layer of an mLAN Transporter device, do not have an effect on the device’s node application. However, certain operations performed on the transport layer may affect the nature of the transmitted isochronous streams packets. For example, increasing the sample rate of the transmitted isochronous streams increases the number of transmitted data blocks.

In modelling the node application component of the Enabler, the node application level audio and MIDI plugs should reflect the connection behaviour of the plugs implemented by a Transporter’s host implementation. These plugs, on the device side, are usually routed to digital inputs or outputs of the transport layer, in order to feed into or receive from the data contained within isochronous stream packets. This same behaviour is reflected in the Enabler regarding its way of modelling the plugs of the node application component. For a connection to be established, a node application

plug has to acquire an available node controller plug, which it uses for its transmission or reception (see section 6.2.1). The node application component has full control over the availability of node controller plugs, which require isochronous resources in order to be created. An object of the node application component has the capability of managing the isochronous resource usage of the device that it represents.

Node application plugs can be modelled in a number of ways, such as surround sound cables, stereo pairs, or snake bundles. Currently, the node application plugs specified by the Enabler assume a monaural channel of audio or a “cable” of MIDI.

The word clock implementation of the node application component allows access to the word clock sources implemented by a Transporter’s host. It permits different word clock sources with a supported sampling rate to be selected and used by the node application of a Transporter. Similar to the implementation of the node application plugs, the node application component also binds its word clock implementation to the synchronization block of the corresponding node controller component. This ensures that any configurations made to the current word clock source of the node application of a Transporter device, are also configured appropriately for the device’s transport layer. This is illustrated in the figure below:

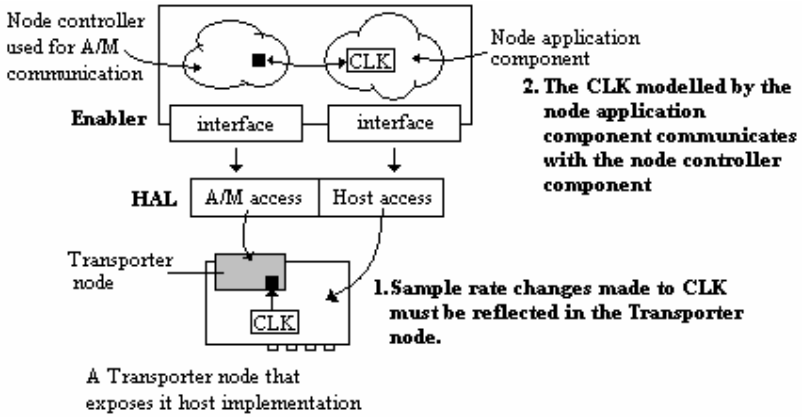


Figure 7-4: Word clock interaction between node application and node controller

In section 6.3.2 it was mentioned that the Basic Enabler does not accurately model the word clock behaviour of mLAN Transporters – it enables masters/slave relationships to be established, but fails at determining the master or slave capability of a device,

and also fails in propagating changes made on a master device to all its associated synchronized slaves. A specification for the implementation of a congruent word clock model is given in the next subsection.

7.1.4 Word Clock Implementation

The Basic Enabler specification models a device's word clock as a plug type, which only allows master/slave relationships to be established and sampling rates to be changed. This leaves the responsibility of modelling other word clock features, such as determining a device's master/slave capability, to client applications. With the introduction of the node application component of the Enabler, it is necessary to define an application-accessible word clock class, which would be responsible for accurately modelling the characteristics of word clock synchronization implemented by mLAN Transporter devices. In addition to providing methods to establish master/slave relationships and modify the word clock sampling rates, this class would also define methods to:

- Determine the presence of a word clock signal.
- Determine whether a device is acting as a word clock slave, or is capable of acting as a word clock slave or master.

A further functionality to be defined by the word clock class is the ability to determine the integrity of a sampling rate value to be (if possible) set to the word clock implementation of either a device's node application or transport layer. The return value of this integrity check is an error message that is defined by the Enabler. A "no error" error message is returned if a specified sample rate can be set successfully. In all other cases, a suitable error message is returned if the integrity check fails. The integrity check may fail if an invalid sample rate is specified or if in certain cases resources in use by a device need to be deactivated at certain sampling frequencies. This is relevant with mLAN Transporter devices implementing the mLAN-PH2 chip. The mLAN-PH2 chip requires the number of transmitting isochronous sequences to be reduced from a maximum of 32 to 16 when an 88.2 kHz or 96 kHz sampling frequency is used.

It is important to realise that if an mLAN Transporter device configured to act as a word clock master has the sample rate of its internal clock changed, or the integrity check performed, it implies that the sampling rate is also changed or the integrity check also performed for all its synchronized word clock slaves. Hence, if the sampling rate or the integrity check of a slave device cannot be established, the whole operation fails. This is illustrated in Figure 7-5 below.

In the diagram, device 'A' is configured to provide word clock synchronization for devices 'B', 'C', 'D' and 'E'. Device 'A' currently has a 48 kHz clock signal, which is regenerated by the other devices. Recall from section 3.3.3 that word clock synchronization information is transmitted over IEEE 1394 using SYTs. An integrity check for a 88.2 kHz sampling rate is performed on the master device 'A'. The Enabler handles this by first checking whether the master device, device 'A', supports the specified sampling rate.

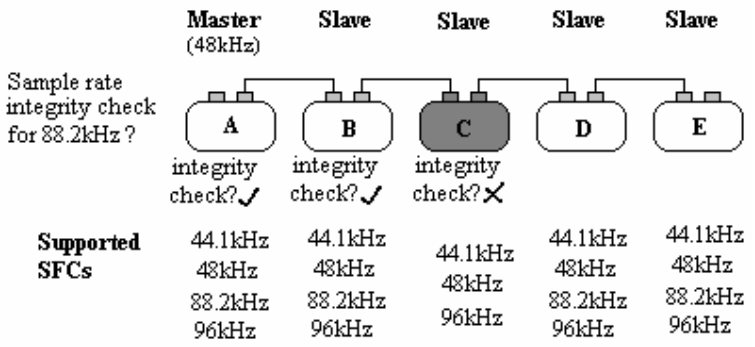


Figure 7-5: Sample rate integrity check performed by the Enabler

If it does, a check is performed on the attached slave devices. From the figure, device 'C' – a slave to device 'A', fails the integrity check and hence an appropriate error message is returned to the integrity check performed on device 'A'.

Table 7-2 summarizes the methods defined by the word clock class proposed by the new Enabler.

Method	Description
IsSlave (…)	*Returns whether an mLAN device is receiving word clock synchronization.*

IsSlaveCapable (…)	*Returns whether an mLAN device is capable of receiving word clock synchronization.*
IsMasterCapable (…)	*Returns whether an mLAN device is capable of generating word clocks for synchronization.*
IsSlaveCapableClockDetected (…)	*Returns whether word clocks are being received by an mLAN device.*
IsMasterCapableClockDetected (…)	*Returns whether word clocks are being generated by an mLAN device.*
GetSlaveCapableSampleRateList (…)	*Returns the supported word clock sample rates capable of being received.*
GetMasterCapableSampleRateList (…)	*Returns the supported word clock sample rates capable of being generated.*
SetSampleRate (…)	*Sets the sample rate of a word clock source.*
GetSampleRate (…)	*Returns the sample rate of the current clock source.*
GetMasterGUID (…)	*Returns the GUID of the mLAN device generating word clocks for synchronization.*
GetSynchronizedSlaves (…)	*Returns a list of mLAN devices receiving word clocks generated by an mLAN device.*

Table 7-2: Summary of the methods defined by the word clock class implemented by the Enabler

7.1.5 Enabler Portability

An additional feature of an Enabler that is not defined by the Basic Enabler specification is the need for portability across multiple platforms. The platform dependent implementations of an Enabler reside in two main areas. The first relates to the communication mechanism implemented between the Enabler and its platform's IEEE 1394 implementation, and the second relates to the implementation of the plug-in mechanism used by the Enabler in loading Transporter HAL plug-ins.

7.1.5.1 Handling Different IEEE 1394 Implementations

Various platforms have different interfaces to their respective IEEE 1394 implementations. Recall from section 4.2.2, where the first Enabler was developed for

Linux, it was said that a user-space library known as *libraw* was used in the Linux environment to allow applications to interact with kernel space 1394 driver modules.

Windows, on the other hand, provides an IEEE 1394 bus driver and a port driver for various IEEE 1394 OHCI compliant host controllers. The Windows 1394 driver stack exports a programming interface that is available in kernel mode only. Hence applications running in user mode are not able to access this interface; instead a kernel-mode driver that conforms to the Windows Driver Model (WDM) has to be implemented to facilitate communication between Win32 applications and the bus driver interface. This is described in Figure 7-6; adapted from Microsoft's MSDN [MSDN, 2005].

.

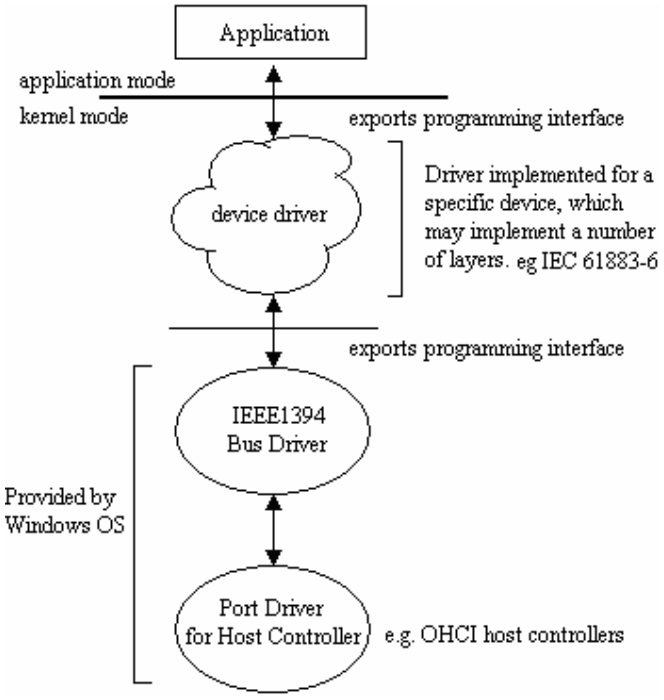


Figure 7-6 : IEEE 1394 device stack for Windows OS

A kernel-mode mLAN driver, developed by Yamaha, provides features that facilitate IEEE 1394 bus communication between the Enabler and Transporter devices. It defines various methods that allow applications to issue asynchronous read, write and lock transactions on the IEEE 1394 bus, as well as configure the Enabler for isochronous streaming implemented according to the IEC 61883-6 specification. This

driver is available and can be downloaded from the Yamaha mLAN Central website [Yamaha Corp., 2005a].

The Macintosh operating system has a structure similar to Linux, where a user-space library exists for applications to access the kernel-space IEEE 1394 implementation. In the kernel, several layers of objects represent each IEEE 1394 device attached to a bus. For each IEEE 1394 hardware interface on a Macintosh, the *IOFireWire family*[10] publishes an *IOFireWireController* object in the I/O Registry. The *IOFireWireController* object provides bus management services for the multiple devices and protocols that can exist on one FireWire hardware interface. The *IOFireWire family* then tries to read the configuration ROM of each device on the bus. For each device that responds with its bus information block, the *IOFireWire family* publishes an *IOFireWireDevice* object in the I/O Registry. The *IOFireWireDevice* object keeps track of the device's node ID and copies properties from the device's configuration ROM, such as the device's globally unique identification (or GUID), into its property list. Most importantly the *IOFireWireDevice* object scans the configuration ROM for unit directories. For each unit directory it finds, it publishes an *IOFireWireUnit* object in the I/O Registry.

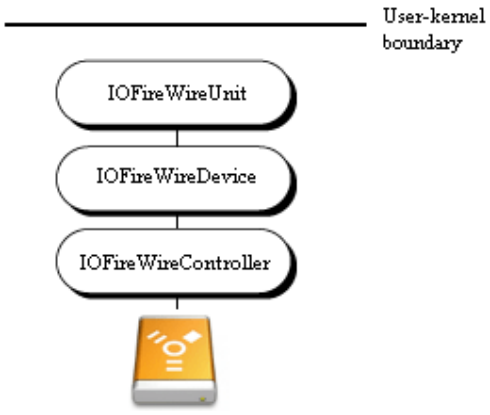


Figure 7-7: IEEE 1394 device stack for the MAC OS

Figure 7-7 shows the stack of objects instantiated for 1394 devices attached to a MAC. Refer to Apple's Developer Connection [Apple Computer Inc., 2005] for more information regarding the implementation of IEEE 1394 on Macintosh platforms.

[10] The IOFireWire family provides support for, and access to, device attached to the IEEE 1394 (firewire) bus

Three application-space libraries are provided by the *IOFireWire family* that permit applications to communicate with IEEE 1394 devices. These include *IOFireWireLib*, *IOFireWireSBP2Lib*, and *IOFireWireAVCLib*. The library *IOFireWireLib* is used for standard 1394 commands and isochronous communication, the *IOFireWireSBP2Lib* library implements the serial bus protocol 2 (SBP-2), and is used to efficiently transfer large amounts of data at high speeds, and the *IOFireWireAVCLib* library is used for sending AV/C commands to an AV/C unit implemented by an IEEE 1394 node. AV/C has been discussed under "Connection Management Techniques" in section 3.4. These libraries, *IOFireWireLib*, *IOFireWireSBP2Lib* and *IOFireWireAVCLib*, define the necessary methods relating to their respective operation. An Enabler running on the Macintosh platform uses the *IOFireWireLib* library to communicate with mLAN Transporter devices.

There are three distinct OS 1394 implementations for the three most widely used platforms. For an Enabler to be portable across these platforms, it must define a common interface that enables communication with the appropriate platform's IEEE 1394 implementation. The Enabler library, when initialized by an application, would detect the platform's operating system and then load the appropriate Enabler-defined low-level 1394 module to interact with the platform's OS 1394 implementation. The methods proposed to make up the Enabler-defined low-level 1394 interface are summarized in Table 7-3. The design structure of this interface will be discussed in section 7.2.1.

Method	Description
SetBusSpeed (…)	Sets the bus asynchronous transmission speed.
ResetLocalBus (…)	Generates a bus reset on the bus attached to the IEEE 1394 interface card.
GetLocalBusGeneration (…)	Retrieves the bus generation of the local bus.
GetLocalNodeID (…)	Returns the 6-bit physical ID of the IEEE 1394 interface card.
GetLocalBusIRMNodeID (…)	Returns the 6-bit physical ID of the isochronous resource manager node attached to the local bus.
GetLocalBusManagerNodeID (…)	Returns the 6-bit physical ID of the bus manager node attached to the local bus.
GetLocalBusNodeCount (…)	Retrieves the number of IEEE 1394 devices on the local bus.

GetTopologyInformation (…)	Retrieves the topology information of the IEEE 1394 devices on the local bus.
MapAddressSpace (…)	Allocates an address space for a specified address range.
DisposeAddressSpace (…)	De-allocates a specified allocated address space.
SetBusResetNotify (…)	Sets the call-back to be executed when a bus reset is invoked on the local bus.
SetFCPMessageNotify (…)	Sets the call-back to be executed when data is written to the FCP command or response register of the IEEE 1394 interface card.
ClearBusResetNotify (…)	Clears the specified bus reset notification call-back
ClearFCPMessageNotify (…)	Clears the specified FCP message notification call-back
AsyncQuadletRead (…)	Issues an asynchronous quadlet read transaction from a specified 48-bit node address of a specified node.
AsyncBlockRead (…)	Issues an asynchronous block read transaction from a specified 48-bit node address of a specified node.
AsyncQuadletWrite (…)	Issues an asynchronous quadlet write transaction to a specified 48-bit node address of a specified node.
AsyncBlockWrite (…)	Issues an asynchronous block write transaction to a specified 48-bit node address of a specified node.
AsyncLock32 (…)	Issues a 32-bit asynchronous lock transaction on a specified 48-bit node address of a specified node.
AsyncLock64 (…)	Issues a 64-bit asynchronous lock transaction on a specified 48-bit node address of a specified node.

Table 7-3 : Methods of the Enabler-defined 1394 interface

7.1.5.2 Handling Different OS Plug-in Implementations

The implementation of the plug-in mechanism by the Enabler takes a similar approach to that of the previous section. With the architectural differences regarding plug-in implementation on Linux, Windows and the MAC, it is necessary for an Enabler to define a common interface that abstracts these differences. The Transporter plug-in mechanism implemented by the Windows Enabler uses Microsoft's ATL COM. This has been described in detail within "The Create mLAN Transporter Sequence Diagram", section 5.3.2.1. The Linux and the Macintosh Enabler adopt a Dynamic Loading (DL) mechanism, where a shared library implementation of a vendor-specific

Transporter HAL is made accessible to the Enabler; the DL mechanism for Linux has been discussed in section 5.3.3.1.

The methods of a plug-in class, defined by the Enabler, that provide uniform interaction across the above-mentioned OS-specific plug-in implementations are summarized in Table 7-4. The design structure of this interface will be discussed in section 7.2.2.

Method	Description
GetHALID (…)	Returns the hardware abstraction layer ID (HALID) used in identifying this plug-in module.
GetNumberOfClients (…)	Returns the number of device transporters created by this plug-in module.
CreateTransporter (…)	Creates a Transporter object implemented by this plug-in module
DisposeTransporter (…)	Destroys a Transporter object
CreateNodeApplication (…)	Creates a node application object for a specified Transporter
DisposeNodeApplication (…)	Destroys a Transporter's node application object

Table 7-4: Methods defined by the Transporter plug-in interface of the Enabler

Up until this point, the Enabler architecture defined by the Basic Enabler specification has been looked at extensively. The shortcomings regarding the implementation of true end-to-end plug connectivity in a bridged IEEE 1394 environment were pointed out, which gave way to design innovations for an enhanced Enabler. The next section describes in detail the object model that governs a new Enabler and how the various classes it defines interoperate in order to provide a more efficient connection management solution.

7.2 Object Model of a Redesigned Enabler

The object model that describes the design of the new Enabler has some features similar to those defined by the Basic Enabler specification. These features include two well defined Application Programming Interfaces (APIs), namely:

- The Client Application API
- The Transporter Hardware Abstraction Layer (HAL) API

The Client API provides a standard programming interface through which client applications can access the various objects hosted by the Enabler. These objects provide a communication handle to the physical components of a network, namely: IEEE 1394 interfaces, buses and devices, and provide a means by which these components can be configured.

The Transporter HAL API defines a common set of methods that abstract the hardware implementation of various Transporter types and allows them to interface uniformly with the Enabler. This is facilitated through the use of a plug-in structure also defined by the Enabler. As mentioned previously, the plug-in structure enables the dynamic loading of vendor-specific HAL implementations, which is used in gaining control over the capabilities of the associated Transporter.

In addition to these two APIs introduced by the Basic Enabler specification, an IEEE 1394 Operating System API is also defined. This added capability facilitates portability of the Enabler across various platforms. As discussed in section 7.1.5.1, the API defines a set of basic abstract methods that serve as a uniform communication interface between the Enabler and a platform-specific IEEE 1394 implementation.

The object model of the Enabler that collectively reveals these three distinct APIs is shown in Figure 7-8. Various classes are defined in this object model, most of which have associations or aggregations with other classes. These classes are discussed in the sections that follow, grouped according to these distinct APIs.

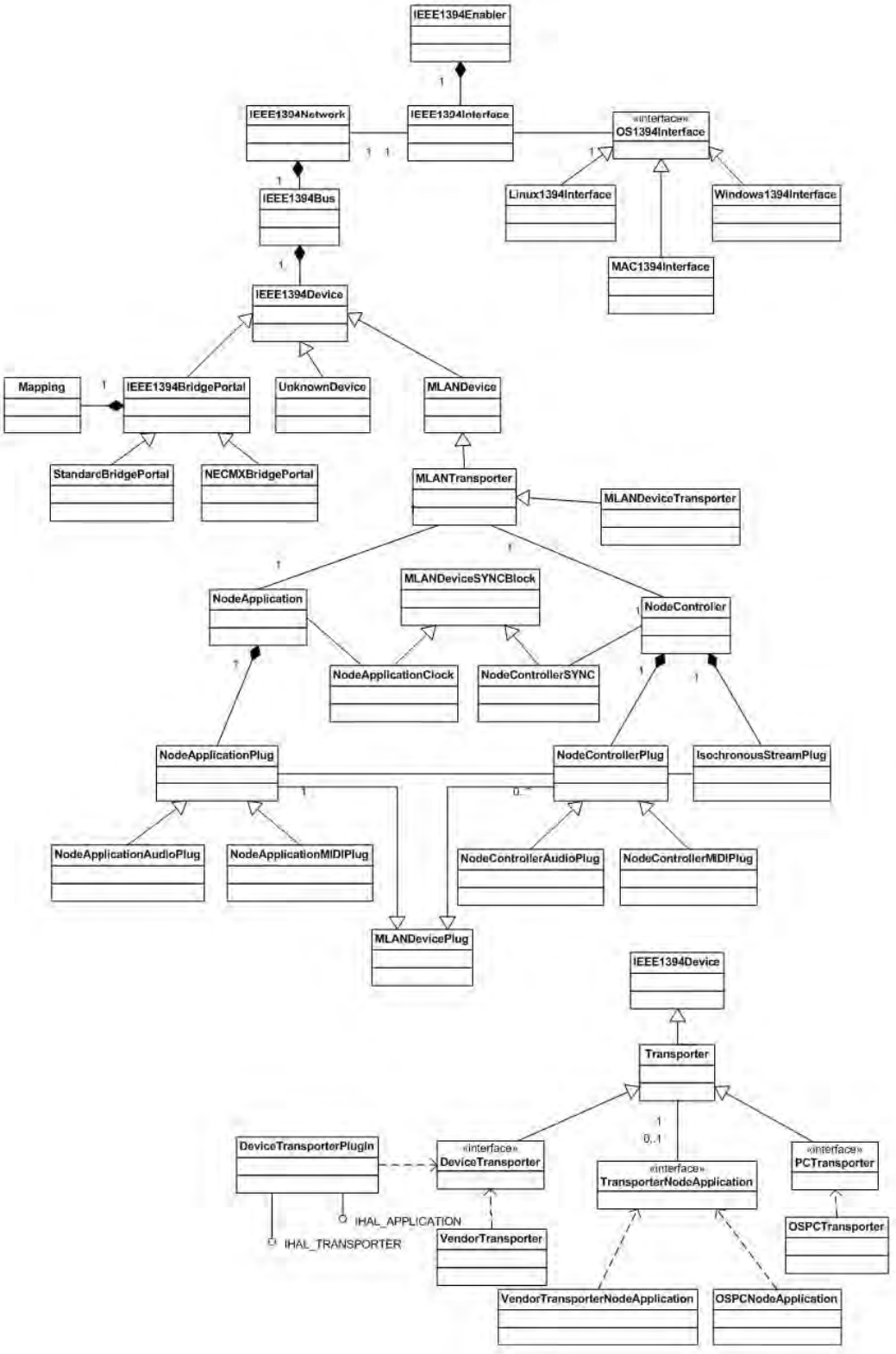


Figure 7-8: Object Model of the Redesigned Enabler

7.2.1 IEEE 1394 Operating System Interface

The group of classes that form part of this interface are the most basic of the Enabler classes. They encapsulate the methods that allow for asynchronous IEEE 1394 *read*, *write* and *lock* operations. In addition, they provide access to information regarding the IEEE 1394 devices attached to a workstation. These include the number of attached IEEE 1394 nodes and the generation value of the bus. Other methods are defined that allow for address range mapping and also, for specifying low-level call-back routines to handle bus resets, FCP messages and also IEEE 1394 bridge command and response messages. The methods defined in the abstract class *OS1394Interface* define the uniform IEEE 1394 interface that is used as a standard means of communication between a platform-specific 1394 implementation and the Enabler. The part of the object model that describes the Enabler's IEEE 1394 operating system interface is shown in Figure 7-9 below.

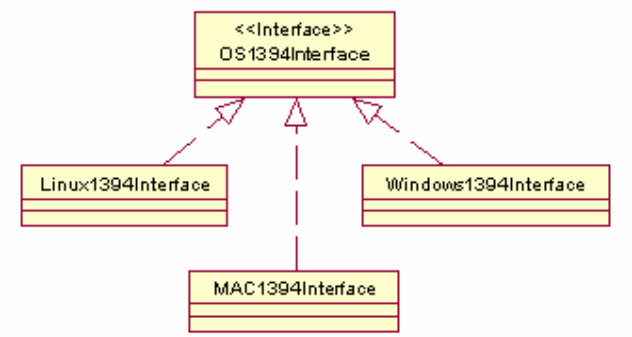


Figure 7-9: Object model of the Enabler's IEEE 1394 OS interface

The implementation of the *OS1394Interface* class is realized by one of the following classes: *Linux1394Interface*, *MAC1394Interface* or *Windows1394Interface*. During the Enabler's start-up routine, the appropriate *OS1394Interface* object is created and used by the Enabler in performing IEEE 1394 related operations on the physical bus.

7.2.2 Transporter Hardware Abstraction Layer

The Transporter hardware abstraction layer provides a common interface to various vendor-specific Transporter implementations. As in the case of the IEEE 1394 operating system interface, it facilitates uniform interaction between the Enabler and

these vendor-specific Transporter implementations. The object model that describes the hardware abstraction layer is shown in Figure 7-10.

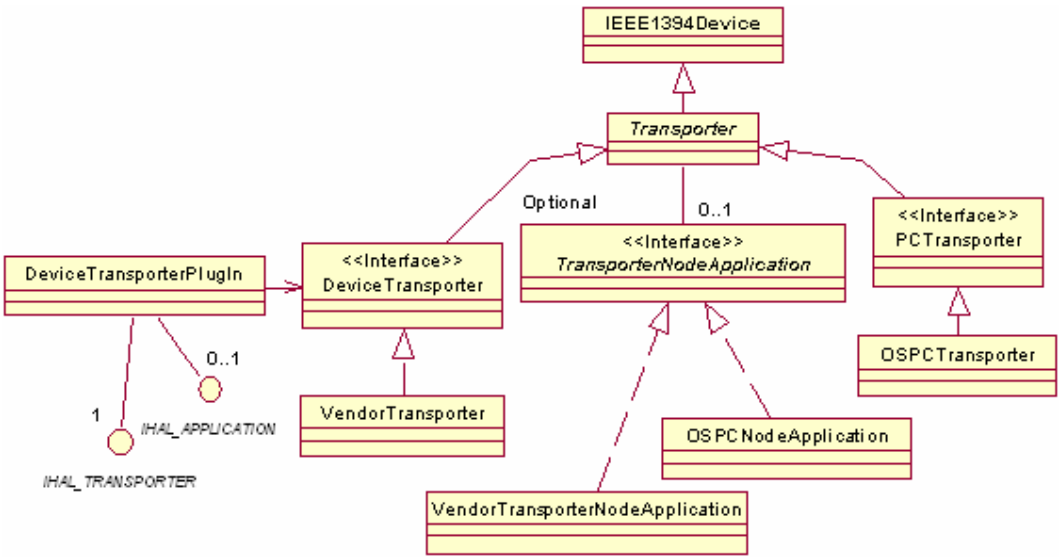


Figure 7-10: Object model of the Transporter HAL implemented by the Enabler

This layer is made up of various classes. The *Transporter* class encapsulates the operations and functionalities of a typical Transporter device. It is an abstract class that declares methods to access or modify the IEC 61883-6 implementation, also known as the transport layer, of vendor-specific Transporters. Recall from the Transporter specification, section 5.1.1, that the transport layer implementation of Transporter devices is mandatory. The methods defined in the *Transporter* class include methods to set the transmission or reception isochronous channel, sequence number and subsequence number, methods to modify the SYT synchronization channel, isochronous transmission speed, sample rate and event type of the isochronous stream data.

Two types of Transporter devices currently exist, modelled by the classes *DeviceTransporter* and *PCTransporter*. The fundamental difference between these two device types is that device Transporters are implemented on stand-alone devices, whereas PC Transporters, as their name implies, are implemented on PC workstations. These classes introduce extra methods that are more specific than the basic Transporter methods. The methods of the classes *DeviceTransporter* and

PCTransporter define the node controller component HAL API that enables high-level applications, including the Enabler, to communicate with the transport layer of Transporters. This API also defines the standard methods that are required to be implemented by the hardware abstraction layers of various vendor Transporters. These are illustrated by the classes *VendorTransporter* and *OSPCTransporter* in Figure 7-10.

The hardware abstraction layer defines a node application component HAL API. This component is optional to Transporter devices, and as such, is not required to be implemented. However, the inclusion of this component facilitates true end-to-end plug connectivity by enabling node application plugs of devices to form part of connection management. The functionality of this component is encapsulated within the *TransporterNodeApplication* class shown in Figure 7-10. The methods declared by this class are also abstract, and are intended to be implemented by vendor-specific implementations. These methods define a hardware abstraction layer API, from which the HAL implementation of the host implementation of various Transporter nodes is based. These methods allow direct access to the capabilities and resources implemented by a Transporter's host. These include the number and types of plugs supported by a Transporter's host, information regarding word clock synchronization, such as the supported sample rates and word clock sources, as well as any host-specific implementation.

The plug-in structure defined by the Enabler facilitates dynamic loading of Transporter HAL plug-ins. This plug-in structure defines a plug-in class that makes up a Transporter HAL Software Development Kit, which is used for the development of vendor-specific Transporter plug-ins. The plug-in class defines a number of *Query Interface* methods. These methods enable the Enabler to investigate the availability of a particular (optional) Transporter capability, e.g. to determine whether a HAL plug-in for a specific Transporter device, has an implementation that provides access to the Transporter's host. Once loaded, the Transporter HAL plug-in is required to make available its currently supported interfaces, which the Enabler can access in order to enumerate the appropriate Transporter modules. This query interface mechanism allows the Enabler and also Transporter HAL plug-ins to be easily extended to provide support for other future-defined Transporter implementations.

The Enabler's Transporter HAL plug-in class is represented by the *DeviceTransporterPlugIn* class in Figure 7-10. A query interface mechanism to determine the presence of the interfaces, *IHAL_TRANSPORTER* and *IHAL_APPLICATION*, is required to be implemented. The presence of the interface IHAL_TRANSPORTER, previously named IHAL_PLUG in the Basic Enabler specification, acknowledges the implementation of the node controller component HAL API within a HAL plug-in that provides access to the transport layer of a particular Transporter device. Similarly, the presence of the IHAL_APPLICATION interface acknowledges the implementation of the node application component HAL API within a HAL plug-in that provides access to the host application of a particular Transporter device. The methods declared by the *DeviceTransporterPlugIn* class, given in Table 7-4, define the standard API upon which a platform dependent plug-in implementation is based.

7.2.3 The Client Interface

The client interface of the Enabler is made up of a number of methods derived from various Enabler classes. This interface allows applications to modify the parameters of IEEE 1394 buses and devices that are present on a network. The classes that make up the client interface can be grouped into two categories: *Fundamental* and *Specialized*. The classes grouped as fundamental implement the basic IEEE 1394 operations that are required by the Enabler. These include asynchronous read, write and lock transactions, setting up address spaces, and isochronous bandwidth and channel allocation.

The specialized classes have a more particular role. Their implementation is based on a particular vendor specification that defines the appropriate protocol required for communication.

The object model that describes the client interface of the Enabler is shown in Figure 7-11. Table 7-5 indicates the fundamental and specialized classes that make up this interface.

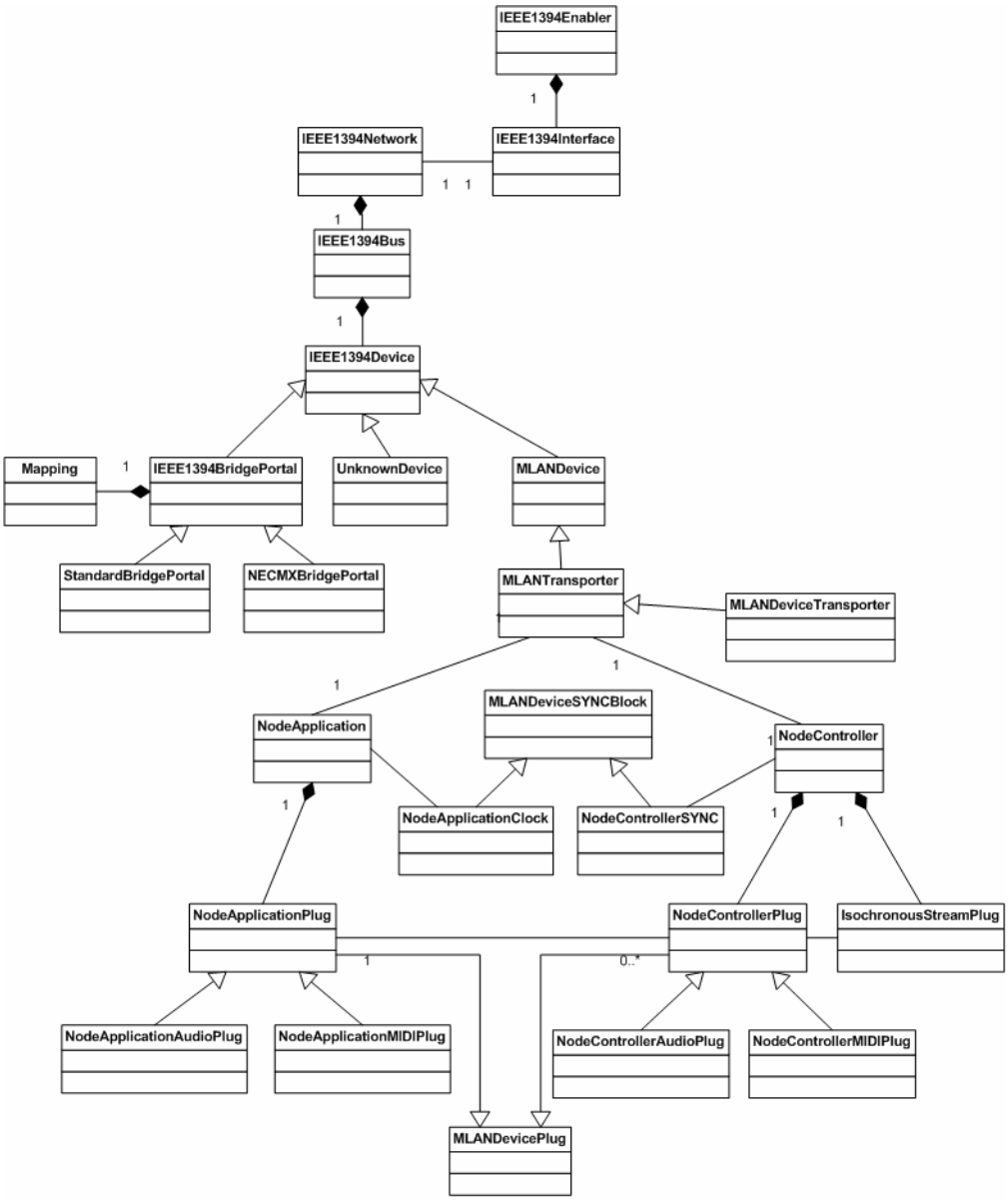


Figure 7-11: Object model of the Enabler's client interface

Fundamental Classes
IEEE1394Enabler, IEEE1394Interface, IEEE1394Network, IEEE1394Bus, IEEE1394Device
Specialized Classes
IEEE1394BridgePortal, NECMXBridgePortal, MLANDevice, MLANDevicePlug, MLANDeviceSYNCBlock

Table 7-5 : Fundamental and specialized classes of the Enabler's Client Interface

7.2.3.1 Fundamental Classes

The *IEEE1394Enabler* class represents the Enabler system. It is responsible for enumerating and creating appropriate objects for the active buses and devices that form part of a network. A client application requiring services of the Enabler would first have to create an object of this class. The *IEEE1394Enabler* class has an aggregation of IEEE 1394 interface objects, represented in Figure 7-11 by the *IEEE1394Interface* class. This class encapsulates the functionality of an IEEE 1394 PCI/PCMCIA card installed on a workstation, and serves as a gateway for asynchronous communication between the device objects hosted by the Enabler and their corresponding associated physical 1394 devices. It is also responsible for detecting, and handling appropriately, any network configuration changes resulting from a bus reset or a bus configuration change that may occur on the attached network. In addition to this, it also allows user-defined address ranges to be set up on any available address space within the Private Space Map (see Figure 3-3) assigned to the IEEE 1394 PCI/PCMCIA card. This feature is particularly useful when devices require communication with the Enabler. In summary, the *IEEE1394Interface* class forms the underlying core structure of the Enabler in terms of asynchronous transaction handling and physical device enumeration.

The interface's network, represented by the *IEEE1394Network* class in Figure 7-11, models the entire collection of the active buses and devices associated with an interface. These active buses and devices are represented within the Enabler by the classes *IEEE1394Bus* and *IEEE1394Device* respectively. An object of the *IEEE1394Bus* class enumerates all the IEEE 1394 devices attached to it. In addition to this, it also defines methods that allow for the retrieval of the bus topology, isochronous bandwidth and channel allocations, and also allows for registering call-back routines to handle bus reset invocations. The *IEEE1394Device* class encapsulates the functionality common to all IEEE 1394 devices. These include methods to retrieve the node ID and GUID of a device, and also methods to perform asynchronous *read*, *write*, and *lock* transactions on a specified node address of a particular 1394 device.

The *IEEE1394Device* class forms the base class for the other device classes implemented by the Enabler. These device classes are more specific in their implementation, and hence fall under the *Specialized Class* category. These classes are discussed in the next section.

7.2.3.2 Specialized Classes

Two types of IEEE 1394 devices are defined by the Enabler, these being mLAN devices and IEEE 1394 bridges. All other device types enumerated by the Enabler are modelled as 'unknown'. These device types are represented by the classes *IEEE1394BridgePortal*, *MLANDevice* and *UnknownDevice*, respectively.

The *IEEE1394BridgePortal* class is an abstract class that provides a common interface to various bridge portal implementations. It declares various methods that allow client applications, including the Enabler, to perform isochronous stream forwarding settings that allow isochronous data to be retransmitted from one bus to another. This is particularly useful in implementing plug connections across buses. Two types of IEEE 1394 bridge implementations exist, as described in section 3.2. These are represented by the classes *StandardBridgePortal* and *NECMXBridgePortal*. The former describes bridges that are implemented according to the specification, "IEEE Standard for High Performance Serial Bus Bridges" [IEEE, 2005]. The latter, the *NECMXBridgePortal* class, describes the implementation of NEC *MX/Bridge-A* bridges. NEC MX/Bridge-A bridges are implemented according to the proprietary specification, "1394 Bridge Application Support Specification" [NEC Corp., 2002], and aim to provide bridging support for devices that implement the IEC 61883-6 protocol. These bridges are the first consumer IEEE 1394 serial bus bridges. The specifications of these two bridges have been discussed in detail under "IEEE 1394 Bridge Model", in section 3.2.

The *MLANDevice* class encapsulates the functionality of mLAN devices. In particular, it defines various methods to access the attributes of mLAN devices, allow for word clock master/slave synchronization and to modify plug connections. Two types of consumer mLAN devices exist, *mLAN Version 1* and *Version 2*. The architectures of these mLAN device types have been discussed in Chapters 4 and 5

respectively. The object model of the new Enabler currently supports only *mLAN Version 2* devices; this is represented by the *MLANTransporter* class of Figure 7-11. The *MLANTransporter* class is further sub-classed to form the *MLANDeviceTransporter* class. Recall that two types of Transporter devices exist, device Transporters and PC Transporters. The *MLANDeviceTransporter* class describes the functionality specific to device Transporters.

The node controller and node application components of the Enabler, modelled for mLAN Transporter devices, are implemented by the Enabler classes *NodeController* and *NodeApplication,* respectively. These classes each define a plug abstraction layer responsible for creating high-level plug objects that model the connection behaviour of a Transporter's logical isochronous stream channels and host application plugs. The plugs modelled and managed by the *NodeController* class are the isochronous stream plugs and the node controller plugs of the transport layer of an mLAN Transporter device. The isochronous stream plugs represent the serial bus plugs implemented by an mLAN Transporter, and are responsible for transmitting or receiving isochronous streams of data from an IEEE 1394 bus. The node controller plugs represent the endpoint of logical channels contained within isochronous streams. These two plug types are represented by the classes *IsochronousStreamPlug* and *NodeControllerPlug* respectively. For the node application component, the node application plugs associated with the *NodeApplication* class represent the hard end plugs or true audio/MIDI data end points hosted by a device. These plugs make use of node controller plugs as the underlying IEEE 1394 transport medium for audio/MIDI data. A device's node application plugs are represented within the Enabler by the *NodeApplicationPlug* class. The nature of the interaction between the node application component and the node controller component of the Enabler is discussed in section 7.1.3.

In addition to the plug abstraction layer implemented by the node controller and node application components, a word clock synchronization block is defined. This block defines methods to modify word clock synchronization parameters of mLAN devices. These parameters include the selection of a word clock source and a corresponding sampling rate. The synchronization block is also capable of establishing master/slave synchronization relationships between various mLAN devices. The synchronization

block of the node controller component is modelled by the *NodeControllerSYNC* class. Similarly, the *NodeApplicationClock* class models the synchronization block of the node application component.

The *MLANDevicePlug* class represents the plugs implemented by an mLAN device. It declares various methods that allow access to the properties of mLAN plugs. This class is abstract, and is intended to be realized by both the *NodeControllerPlug* and *NodeApplicationPlug* classes. The actual realization exposed to a client application depends on whether a node application HAL plug-in implementation exists for a particular mLAN Transporter device. Note that the node application component of a Transporter is an optional requirement. If a node application HAL plug-in implementation exists, the Enabler enumerates the corresponding node application component and hence exposes plug objects of the class *NodeApplicationPlug* to client applications; otherwise plug objects of the class *NodeControllerPlug* are exposed. Similarly, the *MLANDeviceSYNCBlock* class is abstract, and its methods intended to be realized by both the *NodeControllerSYNC* and *NodeApplicationClock* classes. If a node application component is created for a particular mLAN Transporter device, the word clock object of the class *NodeApplicationClock* is exposed to a client application; otherwise the word clock object of the class *NodeControllerSYNC* is exposed.

As discussed in section 7.1.3, the node controller component of the Enabler is modelled independently of the node application component. The main motivation for this is to make the Enabler backward compatible with existing Transporter HAL plug-ins that do not have a node application HAL implementation. In addition to this feature, this design model permits the node controller component created by the Enabler for a Transporter to interoperate with the node application component for another Transporter and vice versa. This capability improves the flexibility of the Enabler. The next subsection briefly describes how vendors provide node application HAL support for their devices.

7.2.4 Providing Node Application HAL Support

Device vendors or 3rd party HAL developers can implement the node application HAL interface defined by the Enabler, by implementing the plug-in interface IHAL_APPLICATION, and providing an implementation for the methods defined by the class *TransporterNodeApplication*. Recall from section 7.1.3 that the methods of the class *TransporterNodeApplication* make up the node application HAL interface; these methods are described in Table 7-1.

The implementation of the IHAL_APPLICATION plug-in interface (and IHAL_TRANSPORTER) is similar to that of the IHAL_PLUG defined by the Basic Enabler specification. This interface provides declarations for the methods:

- *CreateNodeApplication (…)*
- *DisposeNodeApplication (…),*

which are required to be implemented by the HAL. The method *CreateNodeApplication()* creates and returns a *TransporterNodeApplication* object that has a vendor-specific implementation, and the method *DisposeNodeApplication()* destroys a vendor-specific *TransporterNodeApplication* object.

In implementing the methods of the *TransporterNodeApplication* class, it should be assumed that the Enabler or a controller who calls any 'set' methods of *TransporterNodeApplication* would also call (if necessary) the corresponding method of the *DeviceTransporter* class. In view of this, the corresponding call to *DeviceTransporter* should not be implemented from within *TransporterNodeApplication*. For example, if the Enabler changes the sampling rate of the internal clock of a Transporter's host from 44.1 kHz to 48 kHz, the Enabler is required to also set the sampling rate (and any other related changes) to the transport layer of the Transporter by accessing the relevant methods of the *DeviceTransporter* class. This approach provides a thin implementation for the methods defined by the *TransporterNodeApplication* class.

Figure 7-12 shows a typical object model design of a HAL that allows access to the host application of a Transporter device.

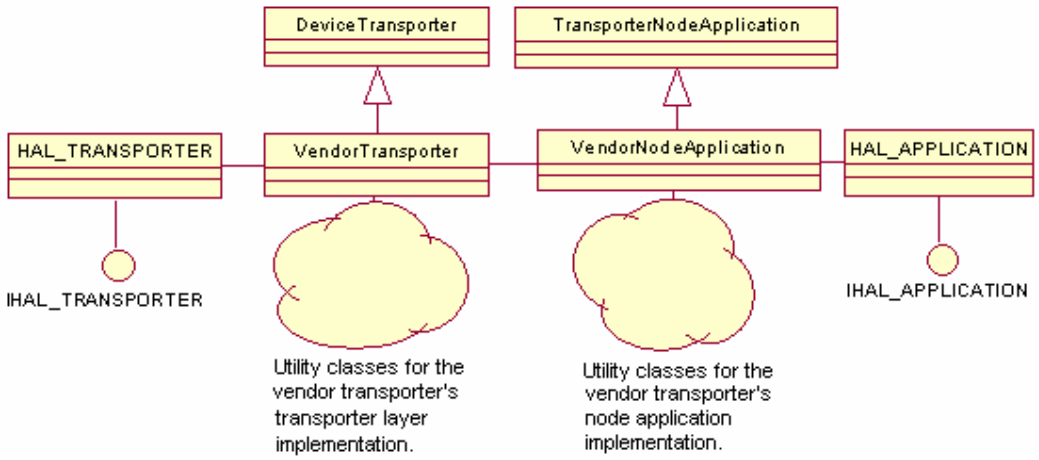


Figure 7-12: Typical HAL design that includes a node application implementation

The node application implementation is given by the class *VendorNodeApplication*, which may make use of other related classes, shown by the associated bubble. The HAL_APPLICATION class implements the Enabler-defined interface IHAL_APPLICATION that allows the Enabler to create and dispose an object of the corresponding vendor node application implementation. The corresponding transport layer implementation, given by *VendorTransporter*, follows a similar approach, and is not very much different from the implementation provided by the Basic Enabler specification.

Having described the design of an Enabler that congruently provides node application support for Transporter devices, and how Transporter HALs can be modified to implement the required node application HAL interface, the next step is to illustrate how the Enabler enumerates a network of devices that may contain multiple buses. This is described in the next section.

7.3 Network Enumeration

Network enumeration refers to the ability of the Enabler to identify the various IEEE 1394 buses and devices attached to a network, and to create the relevant class objects that provide control over them. This operation always occurs at Enabler start up and also after every device or bus configuration change that occurs on the network. Recall from section 7.1.1, that the Enabler class objects should be created and managed by an Enabler rather than by an application, as is currently the case with the Basic

Enabler specification. This section highlights the procedures in terms of sequence diagrams that are used by the Enabler to perform network enumeration.

7.3.1 Enabler Start up

At Enabler start up, which usually occurs when an Enabler object is created by a client application, the Enabler performs its initial enumeration by creating the relevant class objects for the IEEE 1394 buses and devices present on the network. The initial enumeration performed by the Enabler is summarized in the sequence diagram shown in Figure 7-13.

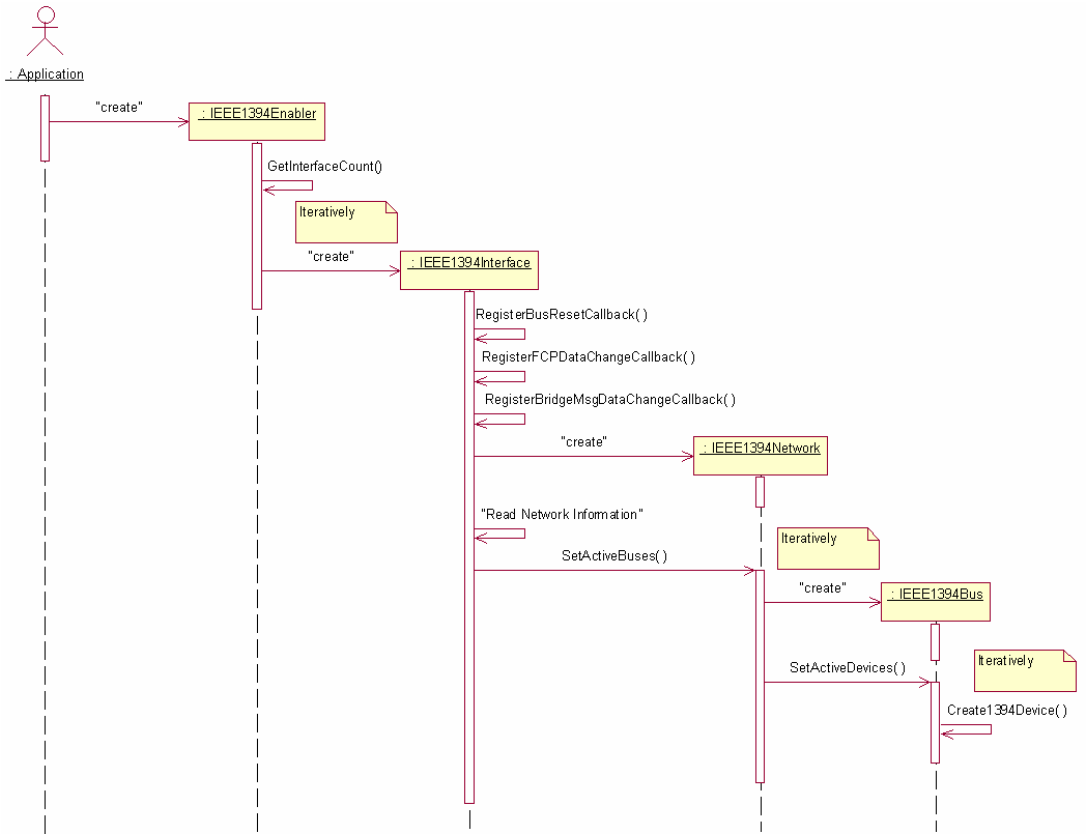


Figure 7-13 : Sequence diagram describing the Enabler's initial enumeration phase

The sequence diagram commences with the creation of an *IEEE1394Enabler* class object by an application. The Enabler object embodies the entire Enabler system and as such, during start up time, it proceeds to enumerate all the IEEE 1394 networks attached to a workstation running the Enabler. From the created Enabler object, the number of IEEE 1394 PCI/PCMCIA interface cards is retrieved. For each IEEE 1394

PCI/PCMCIA interface card, an *IEEE1394Interface* class object is created to model its functionality.

It is at this stage that various call backs are registered to handle:

- Bus resets occurring on the local bus.
- FCP data written to the FCP command register of the interface card (section 3.4.1.3), and
- NEC bridge messages written to the BRIDGE_MESSAGE command or response register of the interface card.

These call back routines enable the Enabler to be aware of any configuration changes that may occur on the network and to update the state of its device objects accordingly.

An *IEEE1394Network* class object is created for each IEEE 1394 PCI/PCMCIA interface card. The network object hosts and manages the aggregation of bus and device objects that correspond to actual IEEE 1394 buses and devices on a network. Information regarding these buses and devices are retrieved and used in initializing the created *IEEE1394Network* class object. The information retrieved at this stage represents the physical attributes of the buses and devices, and is sufficient to create high-level objects that are associated with specific buses or devices on a network. This information is summarized in Table 7-6 and Table 7-7 for buses and devices, respectively. The implementation details that describe how the physical network information is retrieved and stored by the Enabler are discussed in more detail in section 8.1.1.

Physical information retrieved for each IEEE 1394 bus
• Bus ID. • Bus reset generation. • Number of attached 1394 nodes. • Bus topology.

Table 7-6 : The physical information retrieved from IEEE 1394 buses on a network

Physical information retrieved for each IEEE 1394 device
• An indication of whether a device is a bridge portal. • An indication of whether a device acts as the *Isochronous Resource Manager* node of the bus it is attached to. • An indication of whether a device acts as the *Bus Manager* node of the bus it is attached to. • The device's GUID. • The device's node ID.

Table 7-7: The physical information retrieved from IEEE 1394 device on a network

During the initialization of the *IEEE1394Network* class object, an *IEEE1394Bus* object is created for every active bus found on the network. This subsequently leads to the creation of the appropriate *IEEE1394Device* object for each 1394 node present on a bus. The appropriate *IEEE1394BridgePortal* class object, *NECMXBridgePortal* in this case, is created for any NEC bridge portal found on the network. Likewise, the appropriate *MLANDevice* object is created for mLAN Transporter devices found on the network. The *UnknownDevice* class object is created for all other device types. This device creation procedure is illustrated in Figure 7-14.

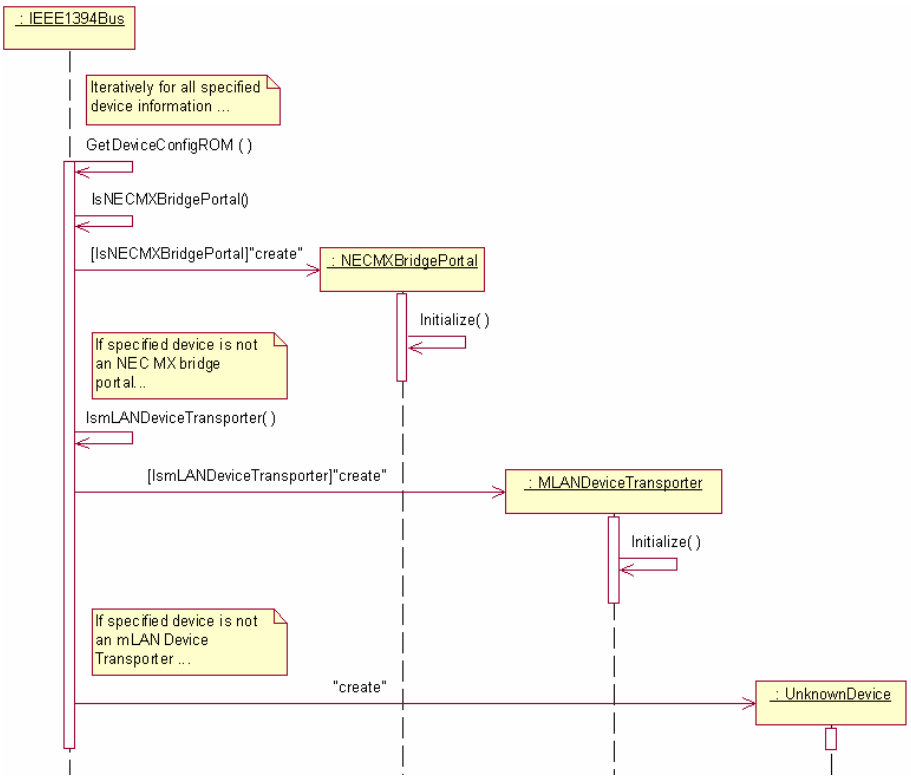


Figure 7-14: Sequence diagram describing device creation by the Enabler

The creation of an mLAN Transporter device by the Enabler occurs in two phases. The first phase involves creating a Transporter HAL object that is capable of communicating to the transport layer and host application implementation of the mLAN Transporter device. This is performed by a corresponding plug-in object that is loaded by the Enabler. If the Transporter HAL object is created successfully, the second phase is then initiated. This phase is responsible for creating the node application component objects and node controller component objects that are used by the Enabler to model the operation of the host application and transport layer exposed by the mLAN Transporter's HAL. This procedure is illustrated by the sequence diagram shown in Figure 7-15.

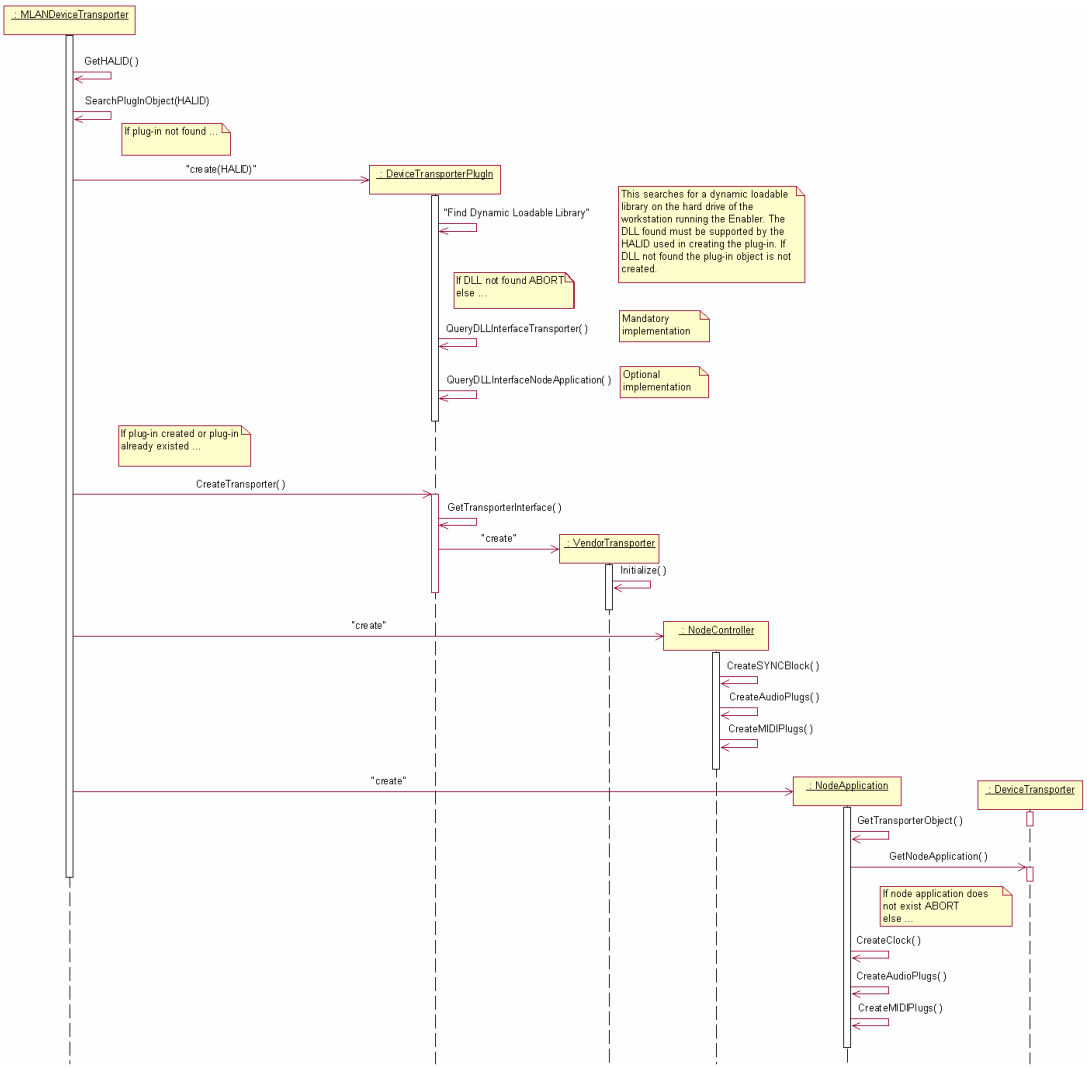


Figure 7-15: Sequence diagram describing the enumeration of mLAN Transporter devices

The Transporter HAL object is created using a HAL identifier, also referred to as the HAL ID. Recall from section 5.1.1.1 that the HAL ID of a Transporter device is located in the Transporter's configuration ROM and is used by a controller to uniquely identify the hardware abstraction layer that is used in communicating with the Transporter.

This identifier enables the Enabler's plug-in loading mechanism to locate or create a plug-in object that is used in creating and disposing the supported Transporter HAL objects. For the plug-in object to be created successfully, a *dynamic loadable library* for a particular mLAN Transporter implementation must exist, and be made available to the Enabler's plug-in loading mechanism. During the creation of the plug-in object, the *interfaces* implemented by the Transporter's HAL, IHAL_TRANSPORTER and IHAL_APPLICATION, are queried using the *QueryInterface* methods described in section 7.2.2.

The second phase of the mLAN Transporter device creation involves creating the high-level node controller component objects and node application component objects of the newly acquired Transporter HAL object. The node controller component objects include objects of the classes *NodeController*, *NodeControllerSYNCBlock*, *NodeControllerPlug* and *IsochronousStreamPlug*. Likewise the node application component objects include objects of the classes *NodeApplication, NodeApplicationClock* and *NodeApplicationPlug.* Depending on whether a node application HAL implementation exists for an mLAN Transporter device, either the node controller component objects or the node application component objects are made available to client applications.

Network enumeration, though mainly invoked within the Enabler's start up phase, also takes place when a bus or a device configuration change occurs on a network. This is caused each time a device or entire bus is added or removed from the network. The Enabler, in return, must be capable of responding effectively to these events. A description of network enumeration with respect to configuration changes is discussed in the next section.

7.3.2 Configuration Changes

A network configuration change is said to occur when an IEEE 1394 bus or device is added or removed from the network. Hot plugging or swapping of devices is a feature supported by the IEEE 1394 specification and as such, IEEE 1394 devices are automatically reconfigured to operate normally after a change in configuration. The Enabler should be capable of handling such configuration changes, and in doing so, update its device objects accordingly. The procedure adopted by the Enabler to handle configuration changes is illustrated by the sequence diagram shown in Figure 7-16.

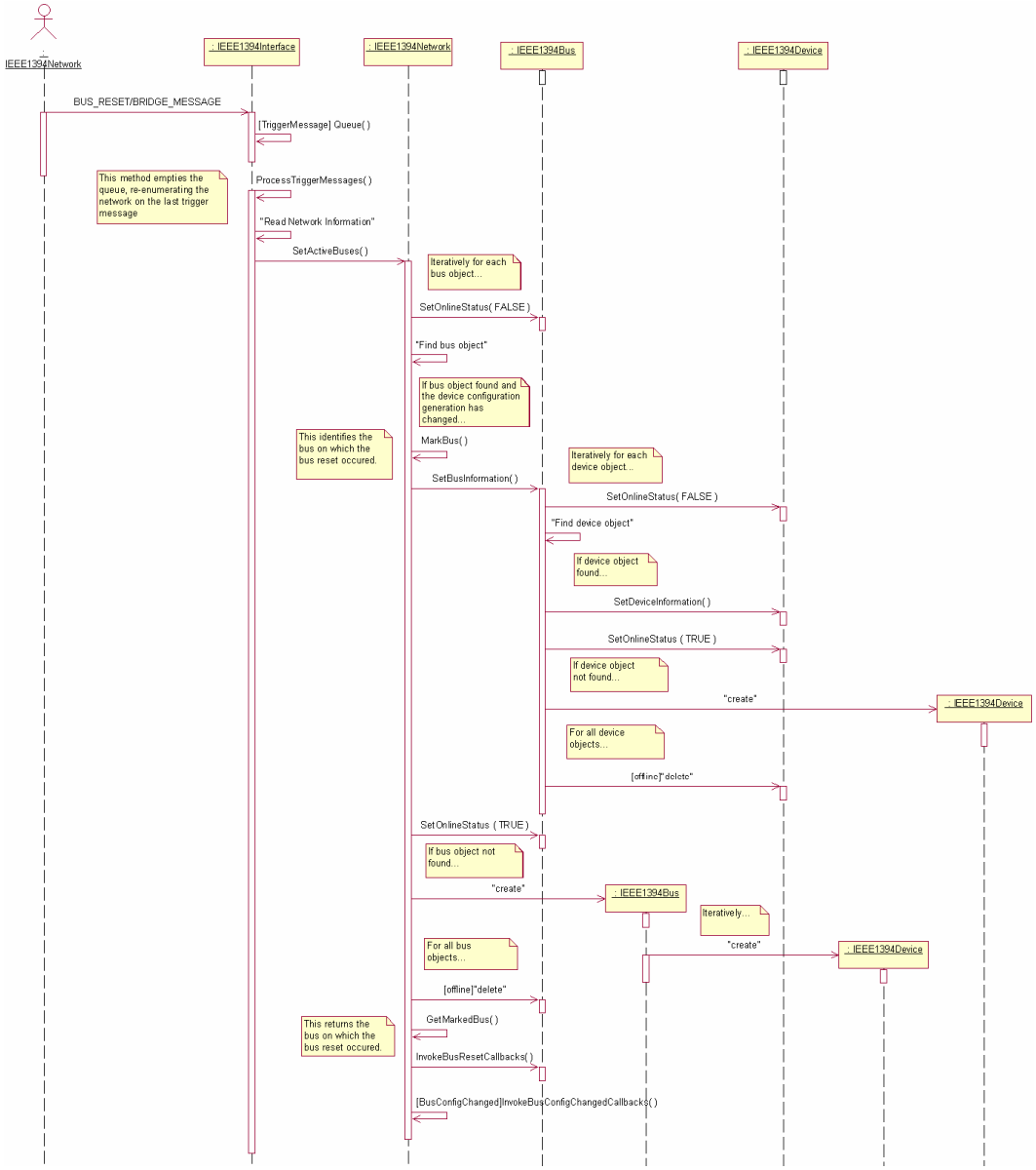


Figure 7-16: Sequence diagram describing the configuration change handling

Bus reset messages or bridge messages that indicate a change in topology, otherwise known as *trigger* messages, are identified and used by the Enabler in handling network configuration changes. The Enabler actively gathers these trigger messages and places them in a queue. The number of trigger messages placed in the queue is equivalent to the number of times a configuration change occurred on the network. Hence, the last trigger message, more often than not gives an indication of when the network is in a stable configuration state. These trigger messages are subsequently emptied from the queue, with the enumeration of the IEEE 1394 network occurring on emptying the last trigger message. The information retrieved during the enumeration of the IEEE 1394 network includes the physical attributes of the buses and devices present on the network.

After the network information is retrieved, the physical information of the 1394 bus objects and their associated device objects that are managed by the *IEEE1394Network* class object are updated. During the update procedure, the corresponding *IEEE1394Device* class object or *IEEE1394Bus* class object are created for any 1394 device or bus attached to the network. Similarly, *IEEE1394Device* or *IEEE1394Bus* class objects are deleted for any 1394 device or bus removed from the network. After this stage, any application-specified bus reset or bus configuration change call-backs are invoked.

Network enumeration forms a vital component of the Enabler and as such, if not implemented and managed properly, can cause significant data inconsistencies between the objects hosted by the Enabler and the corresponding 1394 devices or buses on a network.

The next section briefly describes how a client application typically interacts with this Enabler and the operations that occur within the Enabler in order to perform connection management related tasks.

7.4 Client-Enabler Interaction

Various client applications can be developed to interact with the Enabler; with the patch bay application being the most frequently developed. A patch bay application

facilitates plug connections of audio and MIDI plugs between mLAN devices. Other client applications may include a simple bus analyzer program that is capable of displaying the topology, physical information, and properties of the devices on a network. An application requiring the services of the Enabler is required to install and set up the Enabler software library according to a particular operating system's requirements. The Enabler interfaces made available to an application are summarized in the table below. The use of a particular interface depends on the nature of the client application to be developed.

Interface header file	Description
IEEE 1394device.h	Required for IEEE 1394 device enumeration and to perform basic IEEE 1394 operations. Such operations include performing asynchronous transactions, invoking bus resets, etc.
IEEE 1394bridgeportal.h	Required for IEEE 1394 device enumeration and to perform basic IEEE 1394 bridge portal operations. Such operations include stream control register settings, etc.
mlandevice.h	Required for mLAN device enumeration and to perform mLAN device related operations. Such operations include mLAN plug connections.
transporter.h	Required for Transporter device enumeration and to perform basic Transporter related operations. Basic Transporter related operations refer to operations common to both a PC and a Device Transporter. Such operations include parameter settings specific to the IEC 61883-6 protocol.
devicetransporter.h	Required for the enumeration of device transporters and to perform device Transporter related operations. Such operations include parameter settings specific to the IEC 61883-6 protocol, Enabler address register settings, etc.

Table 7-8 : Implementation interfaces exposed by the Enabler

Irrespective of the implementation interface used by an application, a common sequence of steps exists for an application to follow when querying the Enabler for its available objects. This sequence of steps is summarized below.

- Create a new *IEEE1394Enabler* object.
- Call the *GetInterfaceObjectList (…)* method of the *IEEE1394Enabler* object to retrieve a list of pointers to *IEEE1394Interface* objects.

- Call the *GetNetworkObject (…)* method of the desired *IEEE1394Interface* object to retrieve a pointer to the associated *IEEE1394Network* object.
- Call the *GetBusObjectList (…)* method of the *IEEE1394Network* object to retrieve a list of pointers to *IEEE1394Bus* objects.
- Call the appropriate method of the desired *IEEE1394Bus* object to retrieve a list of device pointers. For example, to retrieve a list of *MLANDevice* objects, call the *GetMLANDeviceList (…)* method, etc.
- *Use the various objects to perform any associated processing.*
- Delete the *IEEE1394Enabler* object.

The next paragraphs illustrate how the above mentioned steps are used by a simple application to access an mLAN network, make plug connections and disconnections between devices, as well as set up word clock synchronization.

7.4.1 Accessing Enabler Class Objects

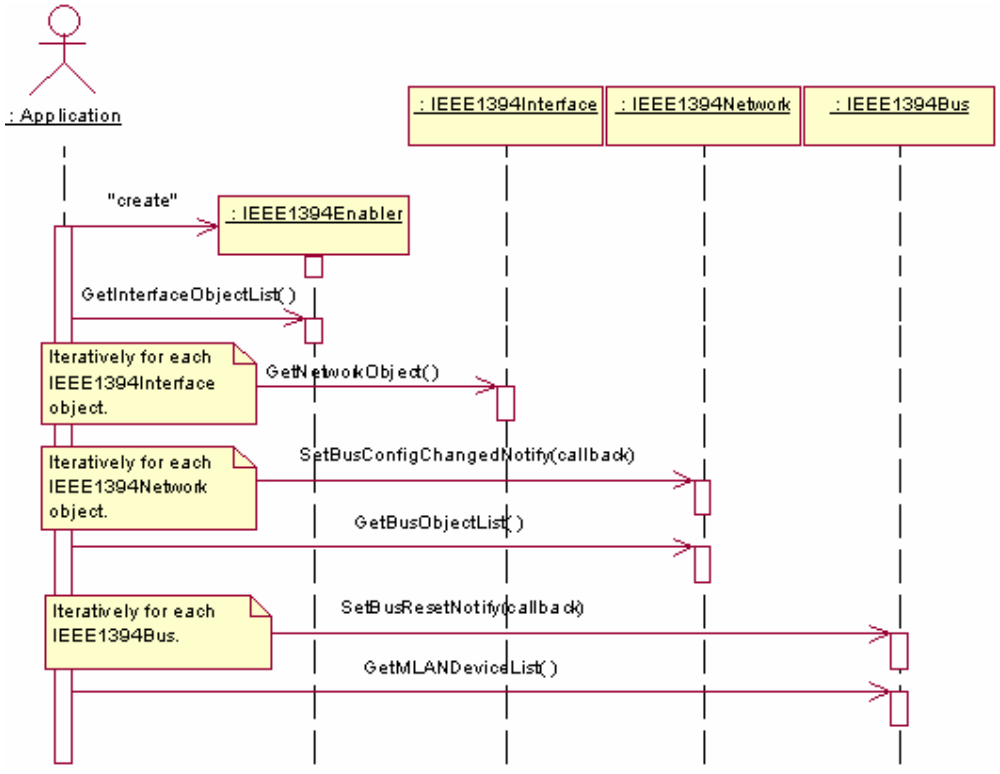


Figure 7-17: Sequence diagram for accessing Enabler class objects

Figure 7-17 above displays a typical sequence diagram containing the sequence followed by an example application when accessing objects of the Enabler. The application first creates an *IEEE1394Enabler* object and then obtains a list of pointers to *IEEE1394Interface* objects. The process involved in creating the Enabler object is described in section 7.3.1. For each interface pointer retrieved, a pointer to the associated *IEEE1394Network* object that contains information about the buses and devices on the network is obtained. At this point, an application-defined call-back to handle changes in bus configuration is registered to the *IEEE1394Network* object; a number of call-backs can be registered. For each *IEEE1394Network* object, a list of pointers to *IEEE1394Bus* objects is obtained. Bus reset call-backs to handle device configuration changes are specified for these *IEEE1394Bus* objects. From each *IEEE1394Bus* object, pointers to mLAN device objects are obtained and used by the application to perform mLAN-related operations.

7.4.2 Making Plug Connections

Before making plug connections, the application has to first retrieve the plug type (audio or MIDI) of the plugs to be connected, the plug ID of both the source and destination plugs, and the reference pointers to the source and destination mLAN device objects, as shown in Figure 7-18 below.

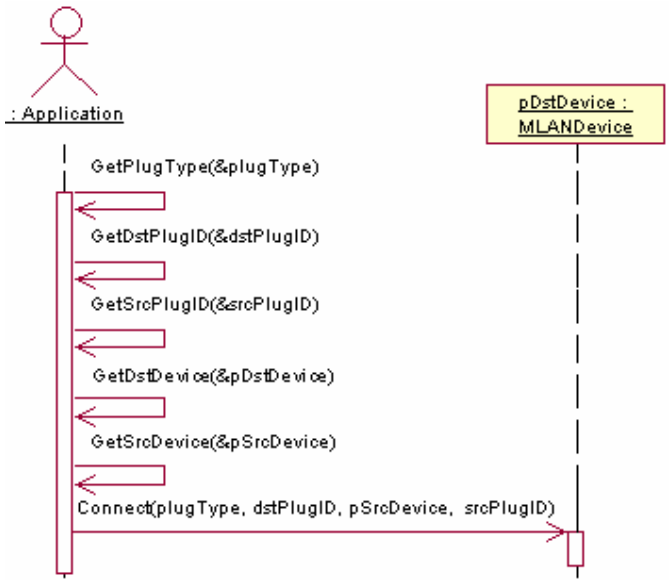


Figure 7-18: Making plug connections from an application

After obtaining these parameters, the application requests for the connection to be made by invoking the *Connect(…)* method of the destination mLAN device object, using the obtained parameters. Note that the mLAN plugs to be connected can either be both node application plugs, both node controller plugs, or one a node application plug and the other a node controller plug. If the plug is a node controller plug, the connection routine followed is similar to the implementation described in section 5.3.2.3 for the Basic Enabler specification, where the source plug is made to commence transmissions and its low-level configuration parameters (isochronous channel, sequence number and subsequence number) copied to the destination plug. If the plug is a node application plug, the dynamic sequence allocation technique (mentioned in section 6.2.2.2) is used to obtain a node controller plug for the node application plug, after which the node controller plug connection routine is performed. The dynamic sequence allocation technique implemented by the Enabler is discussed in section 8.2.1 of the next chapter.

7.4.3 Breaking Plug Connections

To break a connection to a plug, the application has to retrieve the plug type and the plug ID of the destination plug to be disconnected, as well as the reference pointer to the corresponding mLAN device object. The plug type and ID are passed via a *Disconnect(…)* method call to the mLAN device object in order to effect the plug disconnection. This is shown in Figure 7-19.

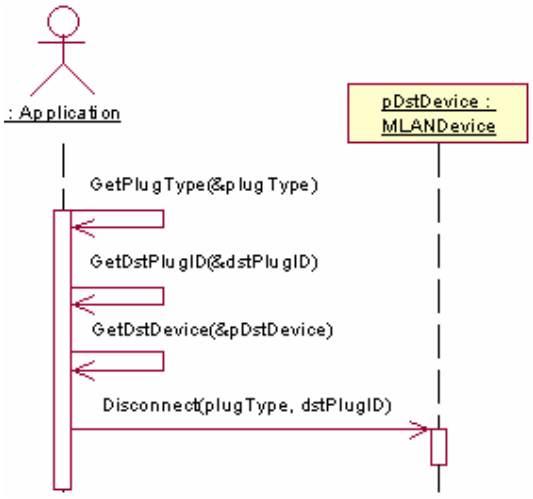


Figure 7-19: Breaking plug connections from an application

7.4.4 Setting up Word Clock Synchronization

The three frequently used word clock related configurations performed on an mLAN device by an application are:

- Changing a device's word clock sample rate.
- Configuring a device to be word clock master to one or more other devices
- Breaking the slave synchronization setting of a device

The procedures required by the example application to achieve the above stated settings are described below.

7.4.4.1 Changing the Word Clock Sample Rate

The sample rate of an mLAN device's word clock can only be changed if the word clock source is the device's internal clock; in all other cases, the Enabler returns an appropriate error message. For the application to change the word clock sampling rate of a device, it first has to obtain the reference pointer of the mLAN device to be configured, and then obtain a pointer to the synchronization block, *MLANDeviceSyncBlock*, implemented by the mLAN device object. The pointer to the synchronization block is obtained by calling the *GetWCLKSyncBlock(…)* method of the mLAN device object. After successfully obtaining a pointer to the *MLANDeviceSyncBlock* object, the sample rate is set by calling its *SetSampleRate(…)* method with the desired sample rate value. This is illustrated in the sequence diagram shown below.

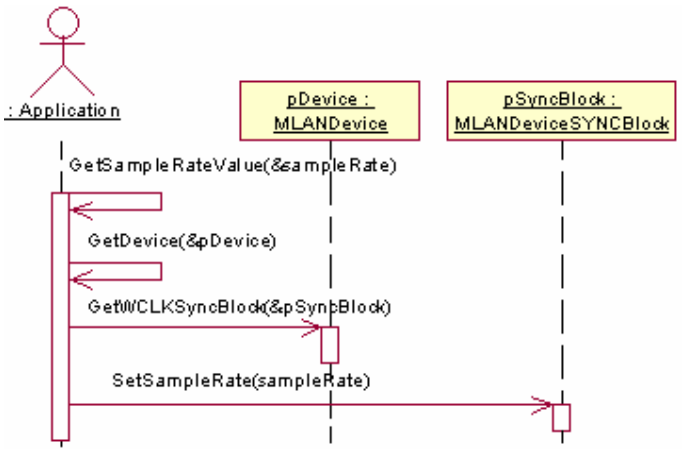


Figure 7-20: Changing an mLAN device's sampling rate from an application

Recall from section 7.1.4 that when changing the sample rate value of a device that is word clock master to a number of other devices, the sample rate of the slave devices are also changed. If the sample rate of a slave device cannot be changed, the whole operation fails. Section 8.3.3 in the next chapter gives further details of the operation performed by the Enabler in changing the sample rate of a device.

7.4.4.2 Setting up a Device to be Word Clock Slave

The Enabler provides a simple way of configuring an mLAN device to receive word clock synchronization from another device. The device receiving the word clock synchronization is referred to as a word clock slave and the device generating the word clock information as a word clock master. Configuring a slave device is done by calling the *Synchronize(…)* method of the mLAN device object that models the mLAN device to be configured, and passing in a pointer to the device object that models the mLAN device that generates the word clock information. The sequence of steps required by an application in achieving this is illustrated in Figure 7-21 below.

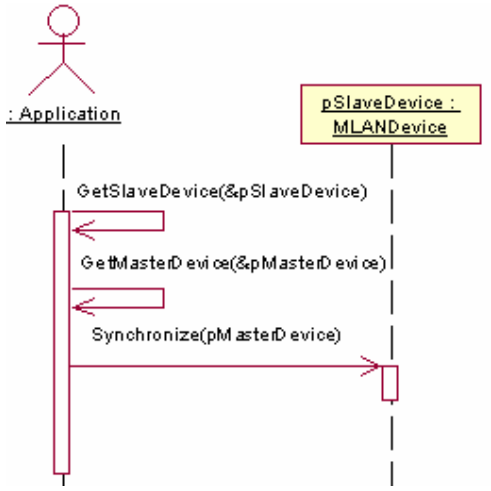


Figure 7-21: Configuration a word clock slave from an application

The mLAN device that is specified to act as a word clock master cannot be operating as a word clock slave. If such a device is specified, an error message is returned to the application. The application can check whether an mLAN device is acting as a word clock slave by calling the *IsSlave()* method of the device's synchronization block, and if required, can break the slave synchronization according to the procedure described in the next subsection. In configuring an mLAN device to be word clock slave, the

Enabler sets up the device to receive SYT timing information from the mLAN device specified to be word clock master. Further information regarding the procedure involved in achieving this is described in section 8.3.4 in the next chapter.

7.4.4.3 Breaking a Slave Synchronization

Breaking the slave synchronization of a device follows a similar set of operations as with configuring the device. This is also achieved by using the *Synchronize(…)* method of the device object that models the slave mLAN device. In invoking the *Synchronize(…)* method, no parameters are specified, which is an indication to the Enabler to configure the mLAN device to synchronize to 'nothing'. The Enabler then changes the word clock source of the device from SYT synchronization to its internal clock. Figure 7-22 gives the sequence of steps an application uses in breaking word clock slave synchronization.

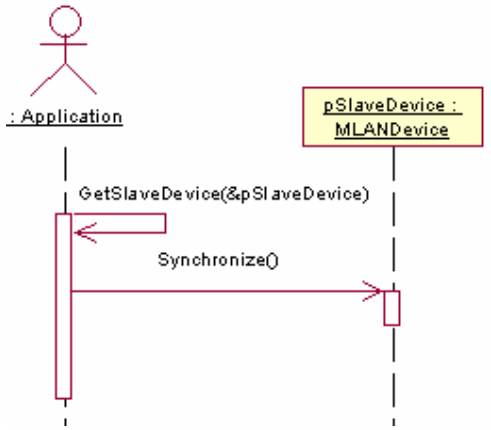


Figure 7-22: Breaking word clock slave synchronization from an application

The procedure used by the Enabler in actually disabling word clock slave synchronization is discussed further in section 8.3.4.

This chapter focuses on the design architecture of an mLAN Enabler that congruently provides support for IEEE 1394 bridges and also models accurately the node application/node controller functionality of mLAN Transporter devices. The node application refers to the host system of a Transporter node, while the node controller represents the transport layer of an mLAN Transporter device. In modelling these components, a patch bay application using this Enabler can establish plug connections

from *user-recognizable* plugs of a device, which are usually the device's hard-end plugs or data bus lines within the device, to *user-recognizable* plugs of other devices. Recall from chapter 2 that the connection management schemes of the current audio networks allow audio/MIDI data routings to occur at the network's transport layer, and usually requires a manual configuration of the devices' plugs to either receive from or transmit to the network.

The node application/node controller implementation of this new mLAN Enabler eliminates this extra step required in configuring a device's hard-end plugs to receive or to transmit data. In addition to this, it allows for the implementation of enhanced features that are not possible with the Enablers of the Basic Enabler specification. The next chapter describes the core component level innovations of an Enabler implementation that is based on this design and hence enables all these features.

7.5 Summary

This chapter has provided a description of the design of an Enabler that:

- Allows Enabler class objects to be created and hosted within the Enabler, thereby facilitating simultaneous application interactions with the Enabler.
- Illustrates a node application/node controller concept, which models respectively, the node application and transport layer implementations of an mLAN Version 2 device.
- Provides a platform for a congruent word clock implementation, as well as techniques that allow for platform-independence of the different low-level IEEE 1394 implementations and the plug-in approaches of the Windows, Linux and Macintosh operating systems.

The Enabler defines three distinct interfaces through which access to the Enabler's functionality is granted. These interfaces include the IEEE 1394 OS interface, the Transporter HAL interface, and the Client API, with each defining a functional role. A description is also given of how network enumeration and network configuration changes are handled by this new Enabler design, in addition to the variety of ways in which a client application can interact with the Enabler.

Chapter 8

8. Component Level Innovations for End-to-End Connectivity

The previous chapter looked at the design architecture of an Enabler that truly models mLAN Transporter devices, and also provides rigorous support for IEEE 1394 bridges. Further on in that chapter, we saw how the Enabler was capable of enumerating and handling configuration changes of a network of devices, and the many suitable ways an application can interact with the Enabler. The Enabler implements a number of new concepts, different from the Basic Enabler specification, which makes it more efficient and more reliable compared with other Enablers developed. A detailed look at these innovative implementations is given in this chapter. This includes the bridging algorithms that capture and represent network information, and also guarantee across-bus flow of isochronous streams from a talker to a listener by examining speed maps. The node application component/node controller component interaction is also discussed, and how dynamic sequence allocation, dynamic sequence deallocation and sequence optimization are performed. Recall from section 6.2.1 that these aid in managing bandwidth usage on a network. Finally the encapsulated word clock implementation is discussed and how the various methods it implements work to provide a simpler way for applications to establish work clock synchronization.

8.1 Bridging Implementation

Recall from section 6.1 of chapter 6 that IEEE 1394 bridge support within an Enabler entails:

1. Network enumeration
2. Across-bus flow of isochronous streams

The former handles all manipulations pertaining to the physical characteristics of the IEEE 1394 buses and nodes present on a network, and the latter manages the routing path of isochronous stream flow on a network. The new Enabler implements bridging by providing support for the above mentioned requirements. The implementation of these requirements by the Enabler is described in the following sections.

8.1.1 Network Enumeration

The network enumeration algorithms implemented within the Enabler can be sub-divided into three categories, namely:

- Physical network enumeration
- Encapsulating topology map information of an IEEE 1394 bus
- Calculating the maximum transfer speed between any two 1394 nodes

8.1.1.1 Physical Network Enumeration

Physical network enumeration, as defined in section 7.3, refers to the ability of the Enabler to identify the various IEEE 1394 buses and devices attached to a network and to create the relevant class objects that provide control over them. The Enabler achieves this by reading the physical information that corresponds to the 1394 buses and devices on the network. Note that these physical properties have been discussed under the chapter heading “IEEE 1394 and Related Technologies” in chapter 3.

Section 7.3.1 described how physical network enumeration is performed by the enhanced Enabler. It was also mentioned that the physical network enumeration occurs at Enabler start up and also when handling network configuration changes. The operation that actually retrieves the physical information is described by the “Read

Network Information" sequence, which forms part of the sequence diagram that describes the Enabler's start-up routine and its configuration change handling mechanism. These are shown in Figure 7-13 and Figure 7-16 respectively. The following paragraphs illustrate how the Enabler encapsulates the physical information of IEEE 1394 buses and devices on the network, which is then used to create the relevant Enabler class objects.

The *Read Network Information* operation is elaborated in Figure 8-1. During this operation, an indication is first made to determine the bus multiplicity of the network.

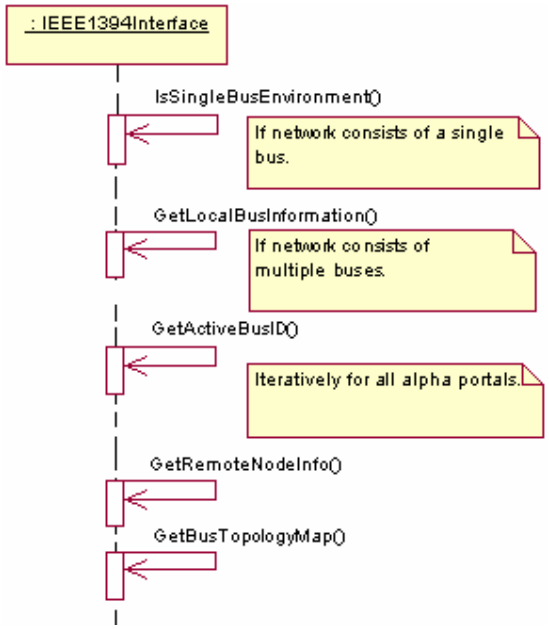


Figure 8-1 : Elaboration of the "Read Network Information" sequence diagram

The Enabler achieves this by searching for the presence of at least one IEEE 1394 bridge portal attached to the local bus of the workstation running the Enabler. If no bridge portals exist, a single bus environment is assumed, otherwise a multi-bus environment is assumed. Following this, the physical information of the network is retrieved. If the network consists of a single bus, the local bus including its attached devices is enumerated. Alternatively, if a multi-bus network exists, the physical attributes of the active buses as well as the nodes on each bus are retrieved.

Enumerating a multi-bus network is a 3-step process. The first involves retrieving the ACTIVE_BUS_ID message from the bridge portal designated to be the n*etwork cycle master*. Recall from section 3.2.2.2 that this returns a 1024-bitmap array that specifies the 10-bit bus ID value of the active buses present on the network. The retrieval of this message is represented by the *GetActiveBusID (…)* operation shown in Figure 8-1. An example of an ACTIVE_BUS_ID message is shown in Figure 8-2. From the byte data shown, it indicates that the active buses on the network have bus ID values of 0, 1 and 2 respectively.

byte data	intepretation
100B0000	generation = **11**
E0000000	busID bitmap: bus[0] = **1**, bus[1] = **1**, bus[2] = **1**
00000000	

Figure 8-2 : An example of an ACTIVE_BUS_ID message

After the ACTIVE_BUS_ID message is retrieved, the *alpha* portal on each of the active buses is queried to return both the REMOTE_NODE_INFO and the BUS_TOPOLOGY_MAP messages. Also, recall from section 3.2.2.2 that the REMOTE_NODE_INFO message contains a list of clustered information, which is identical to the *bus information block* of the configuration ROM of each of the IEEE 1394 nodes attached to the bus. The BUS_TOPOLOGY_MAP message contains a list of self-ID packets that describe the physical properties of the IEEE 1394 nodes attached to the bus. From the REMOTE_NODE_INFO message, the Enabler can determine the GUID, the physical ID, as well as the global ID of each IEEE 1394 node on the bus, and also whether a node is a bridge portal device or operating as the isochronous resource manager or bus manager of a particular bus. Refer to chapter 3 for the definition of these terms.

The self-ID packets contained within the BUS_TOPOLOGY_MAP message enable the Enabler to build a topology tree for the IEEE 1394 nodes on a bus. Examples of actual REMOTE_NODE_INFO and BUS_TOPOLOGY_MAP messages are shown in Figure 8-3 and Figure 8-4 respectively. The description of the various fields given under the column “interpretation” shown in the figures below is given in section 3.2.2.2.

Figure 8-3: An example of a REMOTE_NODE_INFO message

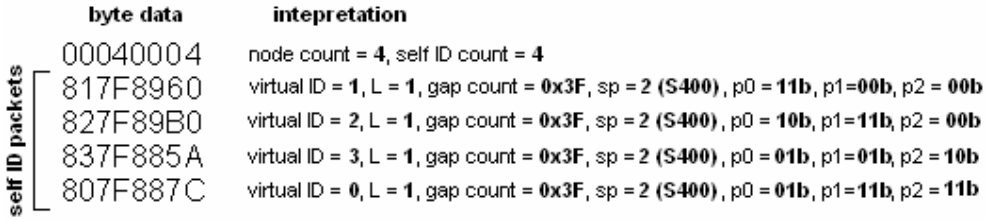


Figure 8-4: An example of a BUS_TOPOLOGY_MAP message

The physical attributes of the buses and nodes retrieved from the network are encapsulated in various *structs* defined by the Enabler. Three *structs* are defined: *DeviceInformation, TopologyInformation* and *BusInformation*. The layout of these *structs* is given in Figure 8-5.

The *DeviceInformation* struct holds the physical information of the IEEE 1394 nodes on the network. As shown in the table, it defines a number of fields that describe the physical characteristics of a 1394 node. The *virtualID* field defined by the struct holds the value of the virtual ID assigned to a node, and as such, its value is only valid in a multi-bus environment.

The *TopologyInformation* struct defines fields that hold the entries contained in the TOPOLOGY_MAP register of the bus manager node assigned to an IEEE 1394 bus. These include a list of self-ID packets of the nodes attached to a bus, as well as the number of these packets. The fields: *nodeCount* and *selfIDCount* respectively specify the number of nodes on the bus and the number of self-ID packets contained within the list *selfIDPacketList*.

```
typedef struct {

   bool   isPortal;
   bool   isIRM;
   bool   isBM;

   UInt64  guid;
   UInt16  physicalID;
   UInt16  virtualID;
```

```
} DeviceInformation, *DeviceInformationPtr;

typedef struct {

   UInt32  nodeCount;
   UInt32  selfIDCount;

   UInt32  selfIDPacketList[kNumMaxNodes];
 } TopologyInformation, *TopologyInformationPtr;

typedef struct {

   bool    isLocal;

   UInt16  busID;
   UInt32  nodeCount;
   UInt32  generation;

   TopologyInformation busTopology;
   DeviceInformation     aDeviceInfo[kNumMaxNodes];
 } BusInformation, *BusInformationPtr;
```

Figure 8-5: Layout of network enumeration structs defined by the enhanced Enabler

The Enabler uses this information to encapsulate topology information of a bus from which applications can then reconstruct the topology. More information on this is given in section 8.1.1.2.

The *BusInformation* struct, in addition to defining physical parameters for a 1394 bus, also has fields that indicate the bus topology and device information of each node on a bus. In other words a *BusInformation* struct specifies the complete physical information of an IEEE 1394 bus that is sufficient to enumerate the bus and its attached nodes. The *busID* field of this struct holds the 10-bit bus ID value assigned to the bus. In a single-bus environment, this field contains the local bus ID value – 1023 (0x3FF), otherwise it contains a value between 0 and 1022 (0x3FE) inclusive. The field *isLocal* is used to identify the bus information that corresponds to the 1394 bus that is local to the Enabler.

The next paragraph illustrates with network examples how the Enabler uses these structs to create class objects. The enumeration of a single-bus network is first illustrated, followed by the enumeration of a multi-bus network.

Network Enumeration Example

In the following network illustrations, assume that the Enabler library has been initialized by an application and that the network enumeration is about to begin. Recall from Figure 7-13 that the network enumeration, at Enabler start up, is performed by the *IEEE1394Interface* object, which at this point of initialization would have already created the *IEEE1394Network* class object.

Figure 8-6 below shows a simple single-bus network that consists of three 1394 nodes, which includes the 1394 interface node attached to the Enabler.

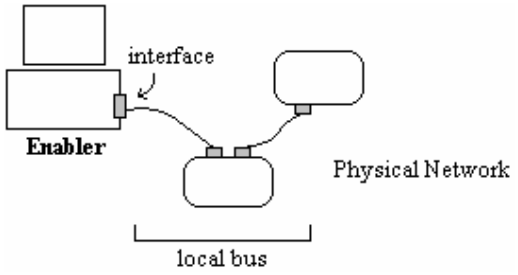


Figure 8-6: A single bus network enumerated at the Enabler's start up

In this network configuration, the Enabler fails to detect the presence of an IEEE 1394 bridge portal on its local bus and hence proceeds to enumerate the local bus. The enumerated values include the physical bus properties, the topology map of the bus, and the physical information of the three nodes. The *IEEE1394Network* class object, created prior to performing the network enumeration, is initialized with this information, which it uses in creating an *IEEE1394Bus* object and the three corresponding *IEEE1394Device* objects. In the event of a bus reset, the entire local bus is re-enumerated and the network object re-initialized to update its bus object and the associated device objects.

Figure 8-7 below shows a multi-bus network environment that consists of two buses, with three and two 1394 nodes respectively on each bus.

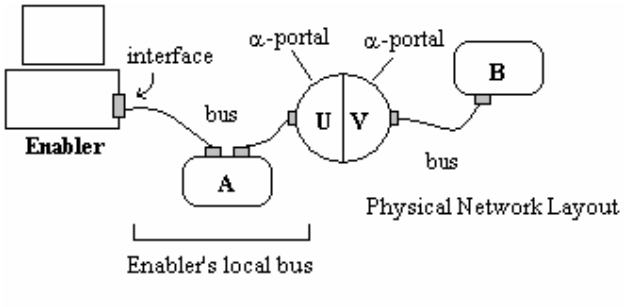


Figure 8-7: A two-bus network enumerated at the Enabler's start up

The nodes attached to the local bus of the Enabler's workstation include the 1394 interface card attached to the workstation, the mLAN device labelled 'A' and the 1394 bridge portal labelled 'U'. Similarly, the nodes attached to the second bus include the 1394 bridge portal labelled 'V', and the mLAN device labelled 'B'. In this network configuration the Enabler detects a portal on its local bus and hence attempts to perform enumeration for a multi-bus network. During network enumeration the two buses, including the associated 1394 nodes, are enumerated by reading the REMOTE_NODE_INFO message of the alpha portals on each bus (see section 3.2.2.2 for a description of the format of the REMOTE_NODE_INFO message). The *IEEE1394Network* class object created prior to enumerating the network is initialized with the enumerated values, from which it creates two *IEEE1394Bus* objects and the corresponding *IEEE1394Device* objects for each bus.

When handling configuration changes, the entire network is also re-enumerated and the *IEEE1394Network* object re-initialized to update its aggregation of bus and device objects.

8.1.1.2 Encapsulating Topology Map Information

The topology map of an IEEE 1394 bus contains information about the physical layout of the nodes attached to the bus. Being aware of the bus topology can be useful in optimizing the serial bus performance of a bus. For example, from a graphical representation of the bus topology on a user application, one can clearly identify and reconfigure the topology of a bus to either reduce the number of cable hops between devices or to group together 1394 nodes with similar speed capabilities.

The mechanism involved in reconstructing the topology of a bus requires examining the physical ID and the port status information contained within self-ID packets generated from the nodes attached to the bus. In a single-bus environment, these self-ID packets are retrieved from the TOPOLOGY_MAP register of the bus manager node assigned to the bus. In a multi-bus environment, these packets are retrieved from the BUS_TOPOLOGY_MAP message of the alpha portal on each bus. Recall from section 3.2.2.2 that the self-ID packet defined by the BUS_TOPOLOGY_MAP message specifies a *virtual_ID* field instead of a *physical_ID* field, which is required for topology reconstruction. The value specified by the *virtual_ID* field of a node's self-ID packet may be different from the node's physical ID, and hence cannot be used in the reconstruction process.

In order to determine the physical ID of nodes in a remote bus, the index at which the self-ID packet appears within the BUS_TOPOLOGY_MAP message is used, as stated by the "1394 Bridge Application Support Specification" [NEC Corp., 2002]. The self-ID packets contained within a BUS_TOPOLOGY_MAP message, as with the self-ID packets retrieved from the TOPOLOGY_MAP register of a bus manager node, are arranged in ascending order of physical IDs. Hence, the first self-ID packet contained within a BUS_ TOPOLOGY_MAP message or a TOPOLOGY_MAP register reading, corresponds to an IEEE 1394 node with physical ID 0, the second corresponds to a node with physical ID 1, etc. This is illustrated in Figure 8-8 using the BUS_TOPOLOGY_MAP message shown in Figure 8-4.

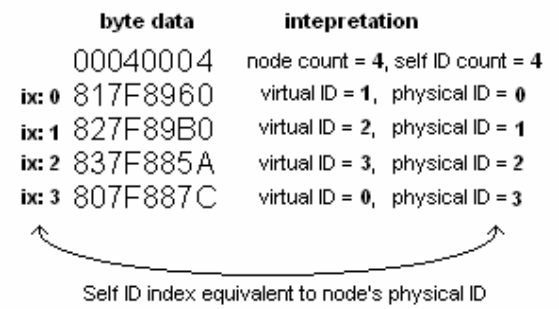


Figure 8-8: Determining physical IDs from a list of self -ID packets

In view of this, irrespective of whether the network is made up of a single bus or multiple buses, the physical ID of the 1394 nodes that form part of a bus can be determined by using the index value at which a self-ID packet appears within the self-

ID list contained within the topology map. This information, together with the port status information, is used by the Enabler to provide an encapsulation of topology map information for applications. The application then uses this encapsulation to reconstruct and display the topology of an IEEE 1394 bus.

The next two paragraphs describe in detail the structure used by the Enabler to encapsulate and present topology information to applications. The process required by an application to reconstruct and display a bus topology from the Enabler's topology encapsulation is given, as well as how the Enabler initializes the encapsulation from self-ID packets. Refer to section 3.1.2.4 for an explanation of how the IEEE 1394 standard allows a bus topology to be reconstructed from self-ID packets.

Reconstructing Topology from Encapsulation

The Enabler encapsulates the topology information of a bus using a self-defined *TopologyMap* struct. Client applications, or the Enabler itself, that require use of the topology information of an IEEE 1394 bus, would typically call the *GetBusTopology (…)* method of the corresponding IEEE1394Bus object. In calling this method, a reference to the Enabler-defined *TopologyMap* struct is also passed as parameter. The definition of this struct is given in Figure 8-9 below.

```
typedef struct {

  UInt32              generation;
  UInt32              numEntries;
  TopologyMapNodeEntry  aEntry[kNumMaxNodes];
} TopologyMap;
```

```
typedef struct {

  UInt32  packet0;
  UInt16  physicalID;
  UInt16  virtualID;
  UInt32  portConnectedToParent;

  UInt32  parentPort;
  UInt16  parentPhysicalID;
  UInt16  parentVirtualID;
} TopologyMapNodeEntry;
```

Figure 8-9: Layout of the topology encapsulation structs defined by the Enabler

The *TopologyMap* struct defines three fields. The first field, *generation*, specifies the actual bus generation value that corresponds to the number of times the physical network has changed. The second and third fields together specify the number of and the actual topology map entries that are contained within the *TopologyMap* struct.

A topology map entry is envisaged to contain the physical connection information of the PHY ports implemented within an IEEE 1394 node. This contains information about the PHY IDs as well as the serial bus characteristics of a node, and if any, also gives an indication of the reference to its parent node. The layout of this struct is given to the right of Figure 8-9. The fields defined by this struct are described in Table 8-1. The description is based on the illustration shown in Figure 8-10, where a topology map entry is derived for the *current* (child) node *X*.

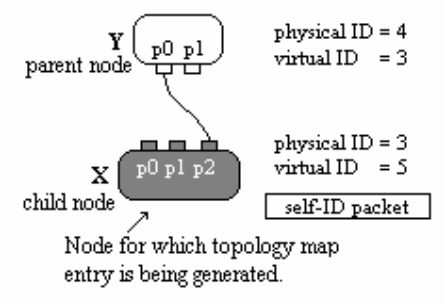


Figure 8-10: Illustration a parent-child topology

Field	Description
packet0	Specifies the 32-bit self-ID packet zero of the node for which the topology map entry is being generated. This also reveals information about its power consumption and speed limitations. For node X, this contains the self-ID packet.
physicalID	Specifies the 6-bit physical ID as seen on the local bus of the node for which the topology map entry is being generated. This returns the value 3 for node X.
virtualID	Specifies the 6-bit virtual ID as seen from a bus other than the local bus of the node for which the topology map entry is being generated. This field is ignored if the network consists of a single bus. This contains the value 5 for node X.
portConnectedToParent	Specifies the port number of the PHY port of the *node* for which the topology map entry is being generated that is physically connected to a parent node. This field is ignored if the *node* is the root node. From Figure 8-10, this specifies the value 2, since port 2 of X is connected to a parent node.
parentPort	Specifies the port number of the PHY port of a parent node that is connected to the *node* for which the topology map entry is being

	generated. This field is ignored if the *node* is the root node. From Figure 8-10, this specifies the value 0, since port 0 of Y is connected to child node X.
parentPhysicalID	Specifies the 6-bit physical ID as seen on the local bus of a node that is connected to the node for which the topology map entry is being generated. This contains the value 4, which corresponds to the physical ID of node Y.
parentVirtualID	Specifies the 6-bit virtual ID as seen from a bus other than the local bus of a node that is connected to the node for which the topology map entry is being generated. This field is ignored if the network consists of a single bus. This contains the value 3, which corresponds to the virtual ID of node Y

Table 8-1: Summary of fields defined by the TopologyMap struct

The *TopologyMap* struct in its entirety contains sufficient information for an application to reconstruct a graphical representation of the wiring layout of the 1394 nodes on a bus. An application achieves this by navigating through the list of topology map entries that specify the configuration state of the nodes on the bus. For a given topology map entry, the connection to a parent node (if any) including its PHY port can be determined and graphically displayed as a straight line connecting two squares. One of the squares represents the node to which the topology map entry belongs and the other square, the parent node. An example is given below of how an application would reconstruct a bus topology using the *TopologyMap* encapsulation.

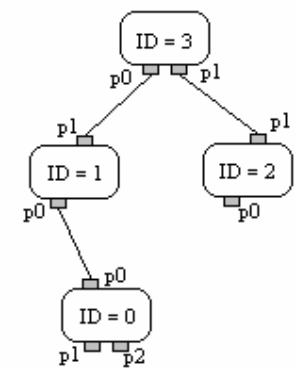


Figure 8-11: Physical layout of IEEE 1394 nodes on a bus

Figure 8-11 above shows a single bus network that consists of four nodes. The corresponding *TopologyMap* information retrieved from the Enabler is shown in Figure 8-12. Note that this does not contain the node's self-ID packets.

generation 65
numEntries 5
aEntry

topology map entry fields	first entry	second entry	third entry	fourth entry
physicalID	0	1	2	3
virtualID	0	1	2	3
portConnectedToParent	0	1	1	---
parentPort	0	0	1	---
parentPhysicalID	1	3	3	---
parentVirtualID	1	3	3	---

Figure 8-12: Topology map information retrieved from the Enabler by an application

The application first takes note of the generation value of the topology information (65) and the number of valid topology map entries (4) contained within the list *aEntry*. The fields of each entry are examined and then interpreted accordingly.

From the column labelled "first entry", the physical ID (0) and virtual ID (0) of a node is given, the port (0) connected to the parent port (0) is given, including the physical ID (1) and the virtual ID (1) of the associated parent node. This information enables an application to represent this connection relationship by drawing a line between two squares arranged in a hierarchal manner, where the square at the bottom of the hierarchy represents the child node and the other represents the parent node. This is shown in Figure 8-13.

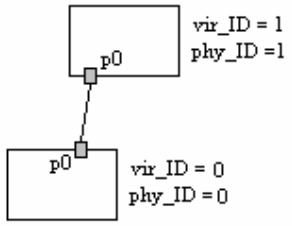


Figure 8-13: Topology reconstruction after examining the first node entry

The column labelled “second entry” indicates that a node with physical ID 1 and virtual ID 1 is connected to a parent node with physical ID 3 and virtual ID 3. By examining the rest of the fields of this entry, an application can represent the connection by a line drawn from the square with physical ID 1 and virtual ID 1 to another square, one level up the hierarchy. Figure 8-13 now becomes:

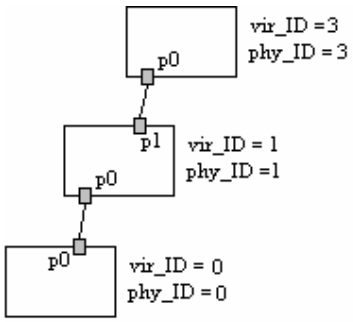


Figure 8-14: Topology reconstruction after examining the second node entry

The column labelled “third entry” indicates that a node with physical ID 2 and virtual ID 2 is connected to a parent node with physical ID 3 and virtual ID 3. From Figure 8-14, the square that represents the parent node with physical ID 3 and virtual ID 3 already exists. In order to show the connection from this square to a square with physical ID 2 and virtual ID 2, a line is drawn from the parent square to another square labelled *vir_ID = 2* and *phy_ID = 2*, one level down the hierarchy. This is shown in Figure 8-15.

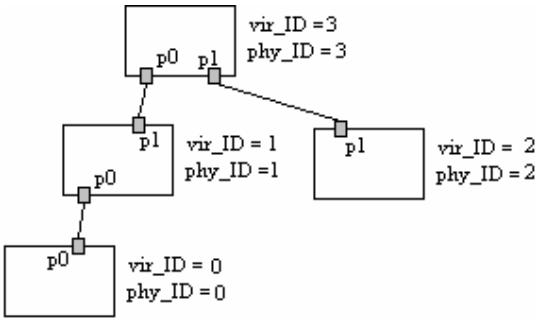


Figure 8-15: Topology reconstruction after examining the third node entry

The column labelled “fourth entry” specifies a node with physical ID 3 and virtual ID 3. The node represented by this column is identified to be the root node because it has no corresponding parent entries. An application can use this information accordingly

in displaying the root node. The final reconstructed topology shown in Figure 8-15 is identical to the physical bus layout shown in Figure 8-11.

The next paragraph describes how the Enabler initializes the *TopologyMap* struct, used in encapsulating and presenting topology map information, from self-ID packets.

Initializing the *TopologyMap* struct from Self-ID packets

The actual algorithm implemented by the Enabler that populates the fields defined by the *TopologyMap* struct is given below in pseudo-code form and is explained in the paragraph that follows.

```
0   IEEE1394Bus::GetBusTopology ( pMap )
1   {
2       ixSelfID = 0;
3       pMap→generation = m_generation; //Assign bus generation
4       pMap→numEntries = m_busTopology.nodeCount; //Assign node count
5       for ixNode: 0 to m_busTopology.nodeCount -1  { // For each (child) node…
6
7           physicalID = ixNode;
8           virtualID = GetVirtualID(m_busTopology.selfIDPacketList[ixSelfID])
9           //Set up the physical properties of the current child node
10          pMap→aEntry[ixNode].packet0 = m_busTopology.selfIDPacketList[ixSelfID];
11          pMap→aEntry[ixNode].physicalID = physicalID;
12          pMap→aEntry[ixNode].virtualID = virtualID;
13          //Set up invalid values for the parent information of the current node
14          pMap→aEntry[ixNode].portConnectedToParent  = kInvalidPHYPortNumber
15          pMap→aEntry[ixNode].parentPort  = kInvalidPHYPortNumber;
16          pMap→aEntry[ixNode].parentPhysicalID = kInvalidNodeID;
17          pMap→aEntry[ixNode].parentVirtualID = kInvalidNodeID;
18          //Get port information and the maximum number of ports implemented
19          portsInfo = GetPortsInformation(m_busTopology.selfIDPacketList[ixSelfID]);
20          maxNumPorts = GetMaxNumPorts(m_busTopology.selfIDPacketList[ixSelfID]);
21
22          for ixPort: maxNumPorts – 1 to 0 { //Examine status of all ports
23
24              aPortInfo = GetPortInfo(portsInfo, ixPort)
25              switch (aPortInfo ) {
26
27                  case kActiveAndConnectedToChild: //Active and connected to a child
28
29                      ixNodeTemp = ixNode;
30
31                      do {// Find parent node
32
33                          ixNodeTemp--;
34                          if (pMap→aEntry[ixNodeTemp].parentPhysicalID == kInvalidNodeID) {
35
36                              pMap→aEntry[ixNodeTemp].parentPort = ixPort;
```

```
37                          pMap→aEntry[ixNodeTemp].parentVirtualID  = virtualID;
38                          pMap→aEntry[ixNodeTemp].parentPhysicalID = physicalID;
39
40                          break;
41                      }
42                  } while (ixNodeTemp);
43              break;
44              case kActiveAndConnectedToParent: //Active and connected to a parent
45
46                  pMap→aEntry[ixNode].portConnectedToParent = ixPort;
47              break;
48              default;
49            }
50          }
51          ixSelfID++;
52      }
53  }
```

A pointer to the *TopologyMap* struct, *pMap*, is passed as argument to the method. The two fields: *generation* and *numEntries* specified by the pointer are immediately initialized (given by lines 3 and 4) with the values held by *m_generation* and *m_busTopology.nodeCount* respectively. The variables *m_generation* and *m_busTopology* are attributes of the *IEEE1394Bus* class, and are initialized during the physical enumeration of the network. The attribute *m_generation* specifies the generation value of the bus, which is equivalent to the number of bus resets that has occurred on the bus. The attribute *m_busTopology* is of type *TopologyInformation* (see Figure 8-5), and specifies self-ID packet information of the nodes on a bus.

For every nodes's self-ID packet retrieved from *m_busTopology*, the *packet0*, *physicalID* and *virtualID* fields of the corresponding node's topology map entry is initialized appropriately, after which the port connection information for the node is retrieved.

During this process, information about the status of all the ports implemented within the node is retrieved (line 19). Recall that this indicates whether a port is connected to a parent, child or neither. The maximum number of possible ports that is likely to be implemented by the node is also retrieved (line 20). For each port, starting from the highest numbered to the lowest, the port status is examined. If it is active and connected to a child, shown by the case statement on line 27, it implies that the current node is acting as a parent to another node, and the corresponding child node is determined, using the *deterministic selection technique*, by locating (in reverse order)

the first node topology map entry with a non-valid *parentPhysicalID* value. The deterministic selection technique is described by the IEEE Std. 1394-1995 [IEEE, 1995] to describe the manner in which self-ID packets are generated by nodes on a bus. When this technique is used in reconstruction, it implies that the highest ranking branch/leaf, i.e. one with the highest physical ID numbered parent port, is the one that must be connected to the highest numbered next available child port. If the child node's topology map entry is located, its *parentPort*, *parentPhysicalID* and *parentVirtualID* fields are initialized with the relevant information of the current (parent) node.

If when examining a port of the current node, the status is found to be active and connected to a parent, as shown by the case statement on line 44, the *portConnectedToParent* field of the node's topology map entry is initialized accordingly. This process is repeated until all the ports of all the self-ID packets have been examined, after which the initialized pointer is returned to the calling application. The operation of this algorithm is demonstrated with an example below using the bus setup shown in Figure 8-11.

An illustration of how the *TopologyMap* struct is initialized

The TOPOLOGY_MAP register information retrieved from the bus manager node of the bus setup shown in Figure 8-11 is summarized and explained below.

Packet		**Description**
0	0x0006EC2D	length (6), CRC (0xEC2D)
1	0x00000002	generation_number (2)
2	0x00040004	node_count (4), self_id_count (4)
3	0x807F4894	phy_id (0), p0 (active & connected to parent), p1 (not active), p2 (not active)
4	0x817F89E2	phy_id (1), p0 (active & connected to child), p1 (active & connected to parent), p2 (port not present)
5	0x827F8060	phy_id (2), p0 (not active), p1 (active & connected to parent), p2 (port not present)
6	0x837F89F0	phy_id (3), p0 (active & connected to child), p1 (active & connected to child), p2 (port not present)

Table 8-2: TOPOLOGY_MAP register information that corresponds to Figure 8-11

This topology map information is stored within the *m_busTopology* attribute defined by the *IEEE1394Bus* class. The *GetBusTopology (…)* method of the *IEEE1394Bus* object is called by an application passing in as parameter a reference to the *TopologyMap* struct, *pMap*.

As shown in the pseudo-code algorithm, the *generation* and the *numEntries* fields of *pMap* are initialized with the values 2 and 4 respectively. These values are derived from the first two quadlets of the TOPOLOGY_MAP information, and are shown by lines 1 and 2 of Table 8-2.

Setting *numEntries* to 4 indicates to the Enabler that there are four nodes on the bus for which topology map entries are to be retrieved. Let Table 8-3 represent the initial state of the topology map entries of the four nodes. The self-ID packet of each node is examined to populate the corresponding fields.

Fields	Index [0]	Index [1]	Index [2]	Index [3]
packet0				
physicalID				
virtualID				
portConnectedToParent				
parentPort				
parentPhysicalID				
parentVirtualID				

Table 8-3: Initial state of topology map entries

First iteration:

From line 5 of the algorithm, *ixNode* has the value 0. Running through lines 7 to 17, the *packet0, physicalID* and *virtualID* fields of the topology map entry for the first node (*ixNode = 0*) is initialized. The initialization values for these fields are obtained from the first self-ID packet shown in Table 8-2 (the row labelled 3). The rest of the fields of *pMap* are invalidated. From lines 19 and 20, the port information: *p0 (active & connected to parent), p1 (not active)* and *p2 (not active),* and the maximum number of ports, *3*, are retrieved. Running through lines 22 to 50, the information of the ports is examined in descending order of port numbers. The only valid connection is on port

0 with a connection to a parent node. Hence from line 46, the *portConnectedToParent* field of the corresponding topology map entry is initialized to be 0. After the first iteration, Table 8-3 becomes:

Fields	Index [0]	Index [1]	Index [2]	Index [3]
packet0	0x807F4894			
physicalID	0			
virtualID	0			
portConnectedToParent	0			
parentPort	---			
parentPhysicalID	---			
parentVirtualID	---			

Table 8-4: Topology map entries after the first iteration

Second iteration:

The variable *ixNode* now has the value 1. The *packet0, physicalID* and *virtualID* fields of the topology map entry for the second node (*ixNode = 0*) is initialized accordingly using the second self-ID packet of Table 8-2 (the row labelled 4). From line 19 and 20 of the algorithm, the port information: *p0 (active & connected to child), p1 (active & connected to parent), p2 (port not present),* and the maximum number of ports, *3*, are retrieved. Running through lines 22 to 50, the valid connections exist on port 1 with a connection to a parent node, and port 0 with a connection to a child node. In handling the connection on port 1 (line 49 of the algorithm), the *portConnectedToParent* field of the corresponding topology map entry is initialized to have the value 1. With the connection on port 0 (lines 29 to 46 of the algorithm), the child node is determined by searching through the current topology map entry list (in reverse order) to find the highest physical ID numbered entry that has a non-valid parent port assignment. If this entry is found, the parent parameters of the entry are initialized. From Table 8-4, the corresponding node entry is located at *Index[0]*, hence the *parentPort*, *parentPhysicalID* and *parentVirtualID* fields of this entry is initialized to have the values 0, 1, 1 respectively. Table 8-4 now becomes:

Fields	Index [0]	Index [1]	Index [2]	Index [3]
packet0	0x807F4894	0x817F89E2		
physicalID	0	1		
virtualID	0	1		
portConnectedToParent	0	1		
parentPort	0	---		
parentPhysicalID	1	---		
parentVirtualID	1	---		

Table 8-5: Topology map entries after second iteration

Third iteration:

This iteration is identical to the first, with the exception that the connection to a parent node exists on port 1. The *portConnectedToParent* field of the corresponding topology map entry is initialized to 1. After this iteration, Table 8-5 becomes:

Fields	Index [0]	Index [1]	Index [2]	Index [3]
packet0	0x807F4894	0x817F89E2	0x827F8060	
physicalID	0	1	2	
virtualID	0	1	2	
portConnectedToParent	0	1	1	
parentPort	0	---	---	
parentPhysicalID	1	---	---	
parentVirtualID	1	---	---	

Table 8-6: Topology map entries after third iteration

Fourth iteration:

The port information of the fourth node *(ixNode = 3)*: *p0 (active & connected to child), p1 (active & connected to child), p2 (port not present)*, and the maximum number of ports, *3*, are retrieved. Running through lines 22 to 50 of the algorithm, the valid connections exist on port 1 and on port 0, with both having connections to child nodes. The topology map entry that corresponds to the child node attached to port 1 is found to be located at *Index[2]*. The *parentPort, parentPhysicalID* and *parentVirtualID* parameters of this entry are initialized with the values 1, 3, 3 respectively. Likewise, the topology map entry that corresponds to the child node

attached to port 0 is found to be located at *Index[1]*, and the *parentPort*, *parentPhysicalID* and *parentVirtualID* parameters initialized to be 0, 3, and 3 respectively. The final topology map entry listing is shown in Table 8-7, and is identical to the example used in Figure 8-12.

Fields	Index [0]	Index [1]	Index [2]	Index [3]
packet0	0x807F4894	0x817F89E2	0x827F8060	0x837F89F0
physicalID	0	1	2	3
virtualID	0	1	2	3
portConnectedToParent	0	1	1	---
parentPort	0	0	1	---
parentPhysicalID	1	3	3	---
parentVirtualID	1	3	3	---

Table 8-7: The complete topology map entries

This section demonstrated how topology map information of a local or remote bus with respect to the Enabler is encapsulated and presented to applications for topology reconstruction, allowing applications to display a graphical representation of the wiring layout of a bridged network. Another component implemented by the Enabler that forms part of the physical enumeration of a bridged network is the ability to calculate the maximum transfer speed allowable between any two nodes on the network. This implementation is given in the next subsection.

8.1.1.3 Calculating Maximum Transfer Speed

In establishing plug connections from an output plug of a talker to an input plug of a corresponding listener, the Enabler, in addition to copying across the isochronous channel, sequence number and subsequence number from the output plug to the input, also has to configure the transmission speed of the isochronous stream of the talker to an optimal speed value, which ensures that the isochronous stream is received by the listener. This optimal speed value is referred to as the maximum transfer speed between the talker and the listener, and is determined by the Enabler in order to guarantee data delivery to the listener. This is illustrated below using an example network shown in Figure 8-16.

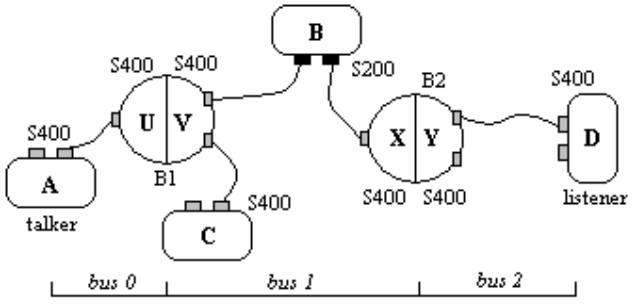


Figure 8-16: Network example demonstrating optimal speed transfer

The network shown above consists of three buses with a number of devices on each bus. Assume that a plug on the talker device ‘A’, residing on bus 0, is to be connected to a plug on the listener device ‘D’ residing on bus 2. After copying the isochronous channel, sequence number and subsequence number from the source plug to the destination, the transmission speed of the talker device ‘A’ is examined, to ensure whether its isochronous streams would be received by ‘D’. In doing so, the maximum handling speed of all the 1394 nodes along the path from ‘A’ to ‘D’ is retrieved, and then the lowest of the speeds set to be the optimal transmission speed of device ‘A’. The device path and maximum handling speed of each node from ‘A’ to ‘D’ is as follows:

A	→	U	→	V	→	B	→	X	→	Y	→	D
(S400)		(S400)		(S400)		(S200)		(S400)		(S400)		(S400)

The lowest handling speed is found to be S200 of device ‘B’, and implies that the talker device ‘A’ would have to be configured to transmit data at S200 for reception at ‘D’.

The SPEED_MAP Control and Status Register

The IEEE Std. 1394-1995 [IEEE, 1995] defines a SPEED_MAP control and status register that is implemented on the bus manager node assigned to a particular 1394 bus. This register contains information that is used by applications or other nodes to determine the maximum transaction speed to be used in communicating with a given node. Don Anderson [Anderson, 1999] gives details on how the SPEED_MAP is

accessed from the bus manager, and how the maximum transaction speed between any two nodes on a bus is determined.

In a bridged environment where multiple buses exist on a network, calculating the maximum transfer speed of 1394 nodes that reside on different buses has to take into account the maximum handling speed of the IEEE 1394 bridge portals that adjoin the buses on which these nodes reside. From the example network discussed previously (Figure 8-16) this refers to portals ‘U’/‘V’ of ‘B1’ and ‘X’/‘Y’ of ‘B2’.

The speed map information defined by the IEEE Std. 1394-1995 [IEEE, 1995] is only sufficient for determining the maximum transfer speed between two nodes that reside on the same bus. However for nodes that reside on different buses, the speed maps of the intermediate buses, as well as the buses of the two nodes would have to be examined in order to calculate the maximum transfer speed. Using the network shown in Figure 8-16 as an illustrative example, it implies that the speed map of *bus 0*, *bus 1* and *bus 2* would have to be examined to determine the maximum transfer speed from the *talker* to the *listener*.

The Enabler does not make use of the speed map information of the bus manager node in calculating the maximum transfer speed between any two nodes, simply because the bus manager node of an IEEE 1394 bus is not guaranteed to be always present. Instead it acquires the bus topology and determines the maximum transfer speed by observing the *speed* field of the self-ID packet that describes each node along the path. This implementation is described below.

The Enabler’s Maximum Transfer Speed Implementation

The *IEEE1394Network* object created by the Enabler implements a method called *GetMaximumTransferSpeed(…)*, which is responsible for retrieving the maximum transfer speed between any two specified 1394 nodes (referred to as *start_node* and *end_node*) on a network. This method examines the intermediate 1394 buses, using the corresponding *IEEE1394Bus* object, along the path from *start_node* to *end_node* by retrieving the maximum transfer speed between the furthest two nodes on each bus that form part of the device path from *start_node* to *end_node*. This is illustrated in Figure 8-17 below.

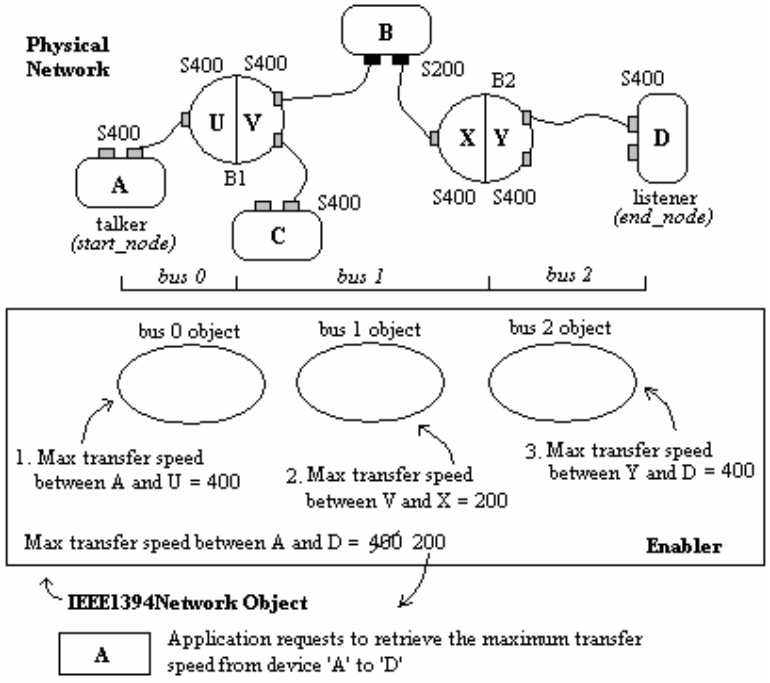


Figure 8-17: Obtaining the maximum transfer speed between two nodes

An application requests the *IEEE1394Network* object of the Enabler to return the maximum transfer speed between the talker ‘A’, and the listener ‘D’. The network object examines the bus objects along the path from ‘A’ to ‘D’: *bus 0 object*, *bus 1 object* and *bus 2 object* respectively and retrieves, for each bus object, the maximum transfer speed of the two furthest nodes on the bus that lies along the path from ‘A’ to ‘D’. In doing so, the network object keeps track of the maximum transfer speed.

For the first bus object, *bus 0 object*, the maximum transfer speed is retrieved from device ‘A’ to the bridge portal ‘U’, and is found to be S400. This value is stored by the network object as the maximum transfer speed. For the second bus object, *bus 1 object*, the maximum transfer speed is retrieved from the bridge portals ‘V’ to ‘X’. This value is found to be S200 and replaces the current maximum transfer speed, S400. On bus 2, the maximum transfer speed is retrieved from the bridge portal ‘Y’ to the device ‘D’. The speed is found to be S400, and since it is not lower than current maximum transfer speed, S200, it is not held by the network object. Hence after examining all the buses, a maximum transfer speed of S200 is returned to the application.

In working out the maximum transfer speed between any two nodes on a bus, the nodes along the path from the one device to the other have to be retrieved. Using this path, the maximum speed-handling capability of each node is examined and the lowest speed monitored. The Enabler makes use of its topology encapsulation in deducing the path from one node to another, and then examines the *speed* field contained within the self-ID packet of each node to monitor the lowest maximum speed-handling capability.

The technique used by the Enabler in working out the device path from one device to another requires the bus topology to have a 'tree' layout. The path from both devices to the root node is individually obtained, from where the actual device path is determined by identifying the first node common to both paths. This mechanism is illustrated below.

Determining the node path

Assume the bus configuration shown in Figure 8-18 and that the device path from A to D is to be deduced as part of determining the maximum transfer speed between these two devices.

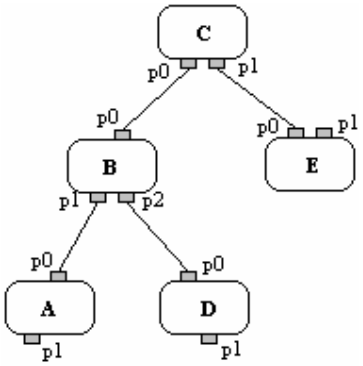
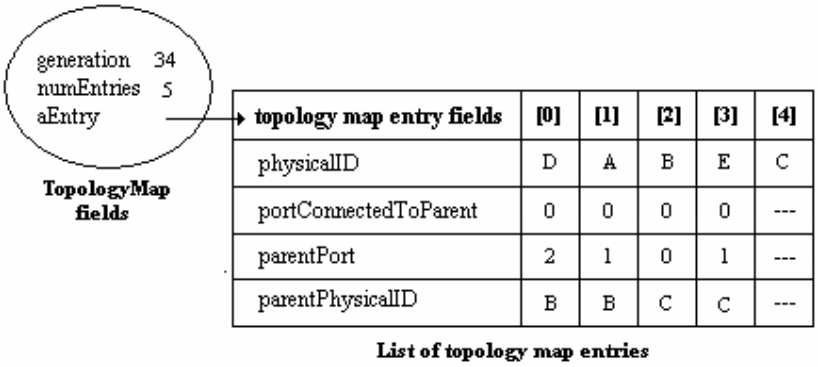


Figure 8-18: Example bus configuration

The following steps highlight the operation of the Enabler in determining the device path.

Step 1: The topology information that corresponds to the above bus configuration is retrieved and shown below. Note that for simplicity, the actual physical IDs of the nodes are not being used.



topology map entry fields	[0]	[1]	[2]	[3]	[4]
physicalID	D	A	B	E	C
portConnectedToParent	0	0	0	0	---
parentPort	2	1	0	1	---
parentPhysicalID	B	B	C	C	---

Figure 8-19: Topology information of example bus configuration

Step 2: The path to the root node from both device ‘A’ and ‘D’ is retrieved, which is then used in determining the actual device.

From Figure 8-19, the topology map entry that corresponds to node ‘A’ is the column labelled [1]. The path to the root node, ‘C’, is retrieved by keeping track of the successive parent nodes of ‘A’, until the node with a *parentPhysicalID* entry of ‘---’ is reached. The successive parent nodes of device ‘A’ are ‘B’ and ‘C’ respectively. Hence, the path from device ‘A’ to the root node is given as A $\rightarrow$ B $\rightarrow$ C. Similarly, the path from device ‘D’ to the root node is given as D $\rightarrow$ B $\rightarrow$ C. From these paths: A $\rightarrow$ B $\rightarrow$ C and D $\rightarrow$ B $\rightarrow$ C the first node common to both paths is ‘B’; hence the path from the talker to the listener is determined to be A $\rightarrow$ B $\rightarrow$ D.

Step 3: After obtaining the node path, the self-ID packet-zero packets of the nodes in the path are examined to determine the lowest PHY speed. The lowest PHY speed determined is also equivalent to the maximum transfer speed that should be used for transmissions from device ‘A’ to ‘D’.

The self-ID packets of the nodes A, B and D are examined. Recall from section 3.1.2.4 that a self-ID packet of a node contains a field that specifies the speed capabilities of its PHY. Assuming that the maximum speed capabilities of the nodes

‘A’, ‘B’ and ‘D’ are S400, S200 and S400 respectively, the lowest speed will be determined to be S200, which then implies that node ‘A’ would have to be configured to transmit at S200 in order for the transmitted data to reach node ‘D’.

From the above explanation, it is apparent that for maximum speed efficiency, the topology of a network should be configured to allow 1394 nodes that have similar speed-handling capabilities to be grouped together.

A pseudo-code implementation of the above description, as implemented by the Enabler, is given below.

```
0    IEEE1394Bus::GetMaximumTransferSpeed (transmitterNodeID, listenerNodeID, … )
1    {
2        GetBusTopology (&pMap) //Get bus topology
3        GetPathToRoot (transmitterNodeID, pMap, srcPathToRoot) //Get transmitter path to
         root
4        GetPathToRoot ( listenerNodeID, pMap, dstPathToRoot ) //Get listener path to root
5        //Get path between talker and listener
6        GetNodePath (srcPathToRoot, dstPathToRoot, nodePath)
7
8        maxTxSpeed = GetMinPHYSpeed ( nodePath, pMap ) //Get minimum PHY speed
9    }
```

The function call shown on line 2 retrieves the topology information of the bus. The operations on lines 3, 4 and 6 determine the node path from *transmitterNodeID* to *listenerNodeID*, and then the maximum transfer speed is determined using the node path.

The manner in which topology map information is encapsulated by the Enabler allows the simple algorithm described above to be used in determining the path between two nodes. In view of this, tree scanning algorithms that make use of the *depth-first search* or *breath-first search* techniques are not required.

Obtaining the maximum transfer speed of a device is one of the operations performed by the Enabler that guarantees isochronous transmissions from one node to another. Another operation that guarantees transmissions, particularly with 1394 bridging, is configuring across-bus flow of isochronous streams. The implementation of this by the Enabler is discussed in the next section.

8.1.2 Across-Bus Flow of Isochronous Streams

The Enabler defines other algorithms that assist in enabling across-bus flow of isochronous streams. These algorithms are used by the Enabler in retrieving plug connections between any plugs on a network, and also in performing across-bus plug connections and disconnections. In all of these operations, the Enabler has to retrieve a list of listener portals along the path from a talker device to a listener that are to be:

- Configured to enable isochronous stream forwarding
- Configured to disable isochronous stream forwarding
- Examined to determine the bus routing undertaken by an isochronous stream.

Recall from section 3.2.1.5 that a listener portal is defined to be a bridge portal that listens to a set of channel numbers in order for their packets to be retransmitted by its co-portal (see Figure 3-19). For NEC MX/Bridge-A bridges, these portals contain 64 stream control registers (one for each isochronous stream channel). Each register specifies across-bus forwarding parameters for the isochronous stream it defines.

When configuring to allow isochronous stream forwarding (see section 6.1.2), valid entries are written to the parameters of the appropriate stream control registers of these listener portals. In contrast, the parameters of these stream control registers are cleared to disable isochronous stream forwarding. To determine the bus routing undertaken by an isochronous stream, the stream control registers of these listener portals are examined to check if the portals are enabled for isochronous stream forwarding.

The mechanism used by the Enabler to retrieve a list of listener portals between any two devices on a network is described below. Also, a description is given of how this list of listener portals is used by the Enabler to enable isochronous bus forwarding, disable isochronous bus forwarding or to determine the bus routings of isochronous streams.

8.1.2.1 Retrieving a List of Listener Bridge Portals

The ROUTING_MAP message implemented by portals of NEC MX/Bridge-A bridges is retrieved by the Enabler in an attempt to determine a list of listener portals

between two nodes. The ROUTING_MAP message, similar to that defined for the ACTIVE_BUS_ID message, indicates the 10-bit bus ID of the 1394 buses to which the bridge portal can asynchronously forward data. The format of the response data for a ROUTING_MAP message request is shown below:

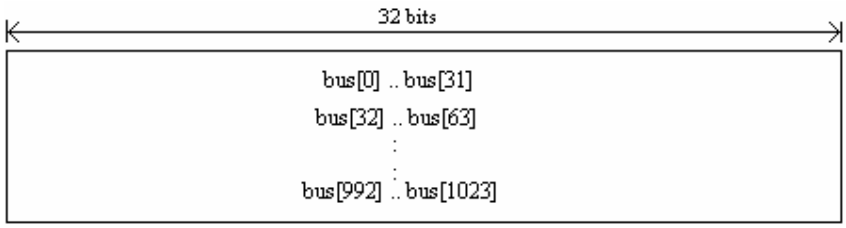


Figure 8-20: Format of the response to a ROUTING_MAP message request

The fields *bus[0]* … *bus[1023]* form the bitmap that indicates the 1394 buses to which asynchronous data can be forwarded to. As an example, consider the bridge setup shown in Figure 8-21 below:

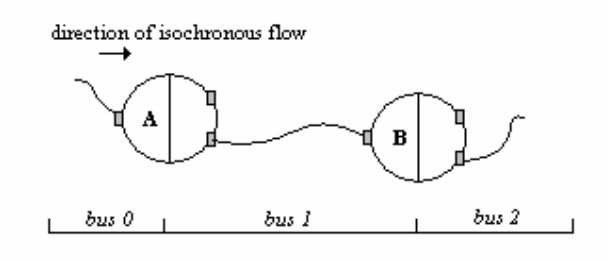


Figure 8-21: Simple network layout

The contents of the ROUTING_MAP messages of the bridge portals labelled ‘A’ and ‘B’ are as follows:

Portal	ROUTING_MAP	Description
A	1, 2	Portal A can forward to the 1394 buses with bus IDs 1 and 2.
B	2	Portal B can forward to the 1394 bus with bus ID 2.

Table 8-8: Results of ROUTING_MAP example

The algorithm used by the Enabler in determining the list of listener portals is illustrated using the 3-bus network shown in Figure 8-22. This network consists of

three IEEE 1394 bridges with portals labelled ‘A’, ‘B’, ‘X’, ‘Y’, ‘U’ and ‘V’, and four mLAN devices labelled ‘a’, ‘b’, ‘c’ and ‘d’ that are distributed over the network.

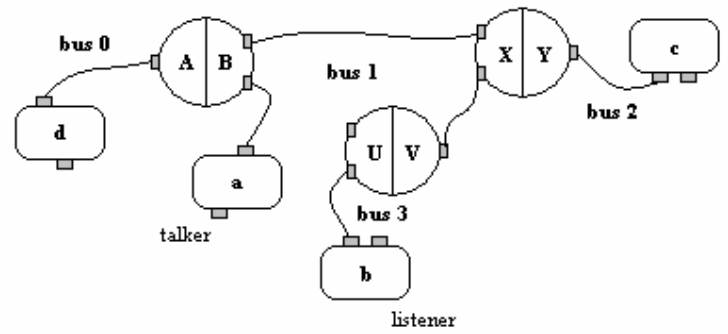


Figure 8-22: Example network to demonstrate ROUTING_MAP usage

Assume that the mLAN device labelled *a*, is configured to be the talker and device labelled *b* is configured to be the listener, and that the listener portals along the path from ‘a’ to ‘b’ are to be retrieved.

Step 1: The first step performed by the Enabler in determining the listener portals along the path, is to identify the bus ID of the IEEE 1394 bus on which the listener node is attached.

From the diagram, this corresponds to the bus labelled ‘bus 3’, in this case the bus ID value 3.

Step 2: After identifying the listener node’s bus ID, the next step is to retrieve and examine the ROUTING_MAP message of each of the 1394 bridge portals attached to the same bus as the talker. In examining the ROUTING_MAP messages, the Enabler keeps track of the bridge portal that is capable of forwarding asynchronous packets to the bus on which the listener device resides.

From the diagram, the bridge portals on the same bus as the talker are ‘B’, ‘X’ and ‘V’. The ROUTING_MAP message of each portal – ‘B’, ‘X’ and ‘V’, is examined to determine which of the portals is capable of forwarding to the ‘bus 3’. The portal capable of transmitting to the target bus is ‘V’. This is because the bus ID of the target bus, 3, is contained in the ROUTING_MAP message of portal ‘V’. The

ROUTING_MAP message of the other portals does not contain the bus ID of the target bus.

Step 3: The 1394 bus adjacent to the bridge portal identified in the previous step is retrieved, and step 2 is performed on this bus. Step 3 is performed repeatedly until the target bus is reached. The list of portals the Enabler keeps track of is the list of listener portals from the talker to the listener node.

From the diagram, the bus adjacent to portal 'V' is examined according to the step 3. This bus is found to be the target bus, and hence the listener portal list consists of only the bridge portal 'V'.

This operation is implemented by the Enabler as described by the pseudo-code below:

```
0    currBus = talker.GetBusPtr();
1    targetBus = listener. GetBusPtr();
2
3    targetbusID = listener.busID
4    while (currBus != targetBus) {//While end bus is not reached
5
6        numPortals = currBus→GetPortalsOnBus(portalsOnBus)
7        for i:1 to numPortals {//For each portal on the bus
8
9            if (portalsOnBus[ i ] →HasRouting( targetbusID )) {//Can forward?
10
11               AddtoListenerPortalPath( portalsOnBus[ i ] ); //Add to list
12               currBus = GetAdjacentBus( portalsOnBus[ i ] ); //Get next bus
13
14               break;
15           }
16       }
17   }
```

This component of the Enabler is used as a utility in performing a number of other Enabler bridge-related operations. These are discussed in the following paragraphs.

8.1.2.2 Enabling Isochronous Stream Forwarding

Isochronous stream forwarding by IEEE 1394 bridge portals is enabled by the Enabler when a request is made to establish a connection between plugs of devices that reside on different buses. In performing the connection, a list of listener portals from the source device to the destination is retrieved, and the corresponding stream control

registers configured to allow isochronous stream flow. The technique involved in configuring the stream control registers follows an approach similar to what was discussed under the heading “Isochronous Stream Forwarding” in section 6.1.2. This section revealed how support for IEEE 1394 bridges was provided by the current Windows and Linux Basic Enablers.

8.1.2.3 Disabling Isochronous Stream Forwarding

Disabling isochronous stream forwarding is performed by the Enabler when it is requested to disconnect a pair of connected plugs. The listener portals along the path from the transmitting device to the receiving are obtained, and the stream control registers that were configured for forwarding, are disabled to stop the forwarding of isochronous streams. In this case, the listener portals are examined from listener to talker, and for each portal the 1394 nodes on the portal’s adjacent bus are checked to determine whether the forwarded isochronous stream is being received. If the forwarded stream is in use, the stream control register of the portal that is configured to forward the stream is not cleared and the disabling process discontinued. If the forwarded stream is not in use, the stream control register of the portal that is configured to forward the stream is cleared and the adjacent bus of the next portal examined. This process is explained further using the illustration below.

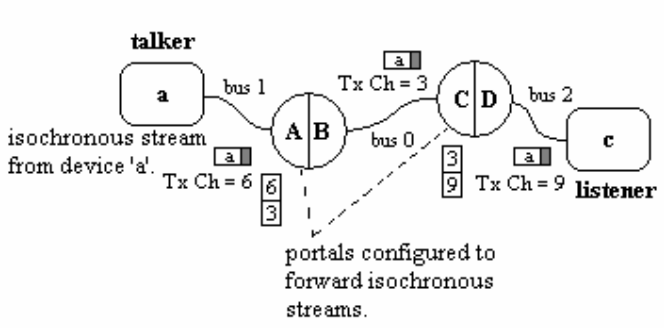


Figure 8-23: Illustrating how isochronous stream forwarding is disabled

Assume that in the configuration above the isochronous stream from the talker device ‘a’ is forwarded and being received by the listener device ‘c’. The Enabler performs the following steps in an attempt to disable the isochronous stream forwarding.

Step 1: The Enabler retrieves a list of listener portals from the talker to the listener.

The list consists of the portals ‘A’ and ‘C’.

Step 2a: The listener portals are examined listener to talker, and for each portal, the 1394 nodes on the portal’s adjacent bus are checked to determine whether the forwarded isochronous stream is being received.

The adjacent bus of portal ‘C’, *bus 2*, is examined to check if any devices attached to the bus are receiving the forwarded isochronous stream. The nodes on *bus 2* are ‘D’ and ‘c’ respectively and neither of these devices are currently receiving the forwarded isochronous stream. Note that device ‘c’, at this stage, would have been configured to stop receiving the isochronous stream.

Step 2b: If the forwarded stream is being received by a node, the stream control register of the portal that is configured to forward the stream is not cleared and the disabling process discontinued. If the forwarded stream is not being received, the corresponding stream control register is cleared and the adjacent bus of the next portal examined.

Since no nodes on *bus 2* were found to be receiving the forwarded stream, the stream control register of portal ‘C’, currently labelled *3/9*, is cleared and the same process described by steps 2a and 2b performed for the next listener portal ‘A’. The corresponding stream control register of ‘A’, labelled *6/3*, is also cleared, which then prevents the isochronous stream from being forwarded from ‘a’ to ‘c’.

Note that in enabling or disabling isochronous stream forwarding, the Enabler also allocates or releases the corresponding isochronous resources on the intermediate buses that have been configured to receive or stop receiving the forwarded isochronous stream. The other bridge-related operation that requires the use of listener bridge portals is the retrieval of the routing path of isochronous streams. This is discussed next.

8.1.2.4 Retrieving the Routing Path of Isoch Streams

The routing path of an isochronous stream is required to be accessed by the Enabler when connections to or from a plug of a device operating in a multi-bus environment are being retrieved. In enabling isochronous stream forwarding, the original channel of the isochronous stream of a transmitting device is mapped to other channels on the intermediate 1394 buses receiving the isochronous stream.

From the network example given in Figure 8-23, the original isochronous channel 6 on *bus 1* is mapped to 3 on *bus 0*, which is further mapped to 9 on *bus 2*. If connection information is to be retrieved from a plug on device ‘a’ that is transmitting isochronous streams, the Enabler has to get access to the mapped isochronous channels in order to compare isochronous channels, and determine the plugs of other devices receiving the stream.

For the Enabler to generate a list of mapped isochronous channels for an isochronous stream being transmitted on a bus, it first has to retrieve lists of listener portals from the bus to the other 1394 buses on the network. For each of the list of portals retrieved, the stream control registers of each portal within the list is examined to determine whether it is capable of forwarding the isochronous stream. The Enabler, in examining these bridge portals, keeps track of the mapped isochronous channel as well as the bus ID of the 1394 bus receiving the forwarded isochronous stream. This is explained using the figure below.

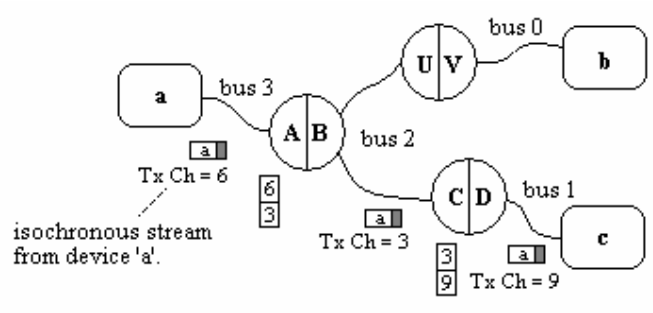


Figure 8-24: Illustrating the routing path of isochronous streams

From Figure 8-24 above, assume the routing path is to be obtained for the isochronous stream transmitted from device ‘a’. The following steps describe the operation of the Enabler.

Step 1: The Enabler retrieves a list of listener portals from the bus on which the specified isochronous stream originates to all the other 1394 buses on the network.

The bus on which the isochronous stream originates is *bus 3*. The other 1394 buses on the network are *bus 0, bus 2* and *bus 1*. The portals along the path from *bus 3* to the other buses are as follows:

bus 3 to *bus 0* : A → U
bus 3 to *bus 2* : A
bus 3 to *bus 1* : A → C

Step 2: For each of the list of portals retrieved, the stream control registers of each portal within the list is examined to determine whether it is capable of forwarding the isochronous stream. In doing so, the Enabler keeps track of the mapped isochronous channel as well as the bus ID of the 1394 bus receiving the forwarded isochronous stream.

For the first portal list, portal A is capable of forwarding the original isochronous stream to its adjacent bus. The mapped isochronous channel and the bus ID of the adjacent bus are ‘3’ and ‘2’ respectively. The second portal in the list is not capable of forwarding and hence is ignored.

In the second portal list, portal A is capable of forwarding the original isochronous stream to its adjacent bus. The mapped isochronous channel and the bus ID of the adjacent bus are ‘3’ and ‘2’ respectively.

For the third portal list, portal A is capable of forwarding the original isochronous stream to its adjacent bus. The mapped isochronous channel and the bus ID of the adjacent bus are ‘3’ and ‘2’ respectively. The second portal in the list, portal ‘C’, is also capable of forwarding the isochronous stream to its adjacent bus. The mapped isochronous channel and the bus ID of the adjacent bus are ‘9’ and ‘1’ respectively.

After examining all the portals contained within the portal lists, the mapped isochronous channels and the bus ID of the 1394 buses receiving the isochronous stream from device ‘a’ are as follows:

Mapped isochronous channel	Bus ID
3	2
9	1

Hence, any devices placed on these buses can receive the isochronous stream from device ‘a’, if they are configured to receive on channels 2 or 1.

This section summarizes the core components regarding the IEEE 1394 bridge implementation of the new Enabler. Isochronous resource management, another key component required by an Enabler, is also supported by this Enabler. This involves cooperation between the host system (node application) and the transport layer of mLAN Transporter devices to ensure optimal use of isochronous resources. The next section describes the technique the Enabler uses in achieving this cooperation.

8.2 Isochronous Resource Management

The IEEE Std. 1394-1995 specification [IEEE, 1995] stipulates that an isochronous channel and bandwidth be allocated for any IEEE 1394 node requiring isochronous transmissions. Recall from chapter 3 that the isochronous resource manager node assigned to a bus keeps track of the isochronous channels and bandwidth available for use on the bus. A maximum of 63 channels and 4915 bandwidth units are available for isochronous transmission, from which a channel and a bandwidth value that depends on the *overhead*, *payload size* and *transmission speed* of an isochronous stream is allocated. In view of this, more devices requiring isochronous transmissions implies that more isochronous resources would have to be allocated from the isochronous resource manager, and if these resources are not managed effectively, can put a constraint on the number of 1394 nodes transmitting simultaneously in a large network.

The dependency of the amount of bandwidth required for isochronous transmission on the *overhead*, *payload size* and *transmission speed* of an isochronous stream is given

by equations (4) and (5) of section 5.1.1.1. Refer to this section for further information on these.

The amount of bandwidth that is calculated for an isochronous stream increases with an increase in *overhead*, *payload size*, and a decrease in *transmission speed*. The *overhead* value, in bandwidth units, depends on the number of cable hops from a transmitter to a receiver [Yamaha Corp., 2002f], but is not strictly calculated according to this; a default constant of 352 bandwidth units is used in most cases. The *transmission speed* of an isochronous stream is a desired property, which is pre-configured by a user. S400 is the standard transmission speed used by most devices. The *payload size* of an isochronous stream is proportional the number of isochronous sequences carried by the isochronous stream, which varies according to the amount of audio/MIDI 'channels' required to be transmitted.

The Enabler manages the bandwidth usage of a device by monitoring the payload sizes, and hence number of sequences transmitted by each isochronous stream of the device. Three distinct approaches are used. These include:

1. Dynamically allocating isochronous sequences on-the-fly.
2. Pre-configuring isochronous sequences to be used by Transporter devices
3. Optimizing isochronous sequences already in use by a Transporter.

The following subsections describe the operation and implementation of these isochronous resource management techniques. Reference will be made to the node application and node controller concepts of the Enabler introduced in section 7.1.3. Recall that the node application component of the Enabler models the host implementation of a Transporter node, while the node controller component models the Transporter's transport layer. Also recall that the host plugs of an mLAN Transporter device are modelled by the node application plugs of the node application component of the Enabler. Similarly, the plug abstraction provided for the transport layer of a Transporter is given by the node controller plugs of the node controller component of the Enabler. A plug connection made from a node application plug modelled by the Enabler, requires use of a node controller plug for transmissions. This relationship is also reflected in hardware between the corresponding host plug of the device and an isochronous sequence of its transport layer. The node

application/node controller implementation of the Enabler is described more extensively in section 7.1.3 of chapter 7.

8.2.1 Dynamic Sequence Allocation

Dynamic sequence allocation is performed by the Enabler when handling connections to node application plugs. This section describes the procedure used by the Enabler in assigning a node controller plug to the node application plug to be connected, and is only performed if the node application plug does not already have an associated node controller plug.

Consider a simple audio Transporter device with four host application plugs and a transport layer that transmits an isochronous stream with three sequences. Also assume that, by an initial configuration, two of the isochronous sequences are assigned to the first and third node application plugs. Figure 8-25 shows the Enabler model of this device's initial configuration.

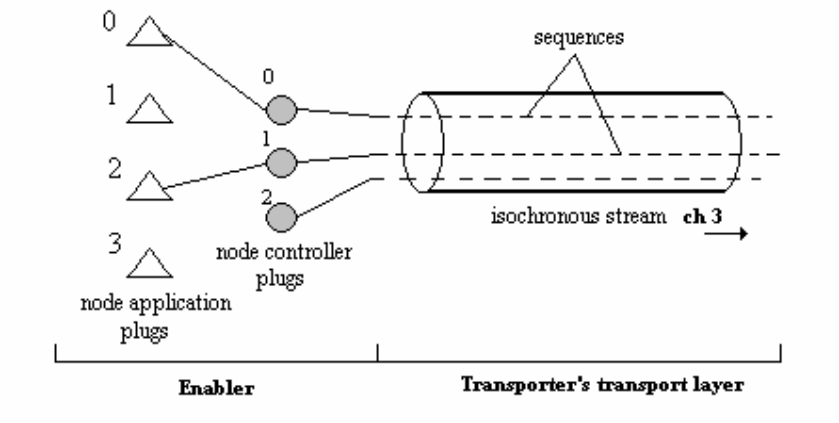


Figure 8-25: Enabler model of an hypothetical Transporter device

Connections established from node application plugs *0* or *2* require no additional configuration to be performed by the Enabler on the Transporter's host implementation or transport layer. These plugs already have associated node controller plugs, which can be used for 'carrying' any audio data from the node application plugs over IEEE 1394. Bear in mind that a node controller plug translates to an isochronous sequence in hardware. On the other hand, connections established from node application plugs *1* or *3* each require an assignment to either an unused

node controller plug or a newly created node controller plug for transmissions. These translate to either an unused isochronous sequence or a newly created isochronous sequence.

If an application requests a connection to be made from node application plug *1*, the optimal allocation procedure for a node controller plug would be to assign node application plug *1* to the unused node controller plug, plug *2*, for data transmissions. This has to be effected on hardware by configuring the corresponding host application plug of the device to have its audio data routed to the third sequence carried by the isochronous stream. The updated Enabler model is shown below.

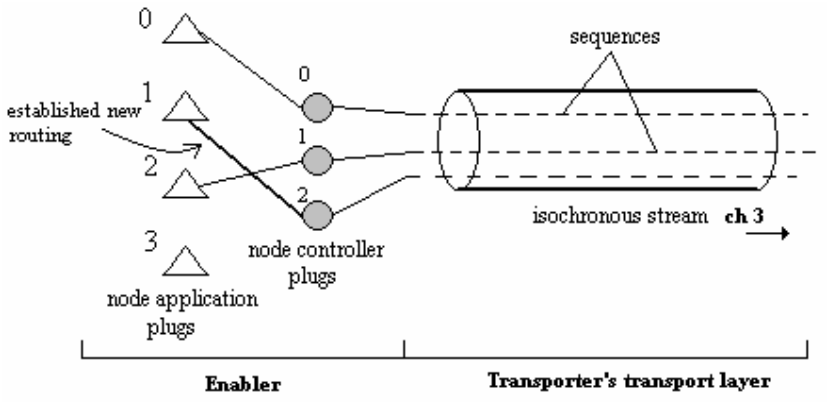


Figure 8-26: A node application plug assigned to an unused node controller plug

Note that, no further isochronous resources are allocated for the Transporter by the Enabler device in performing this step. This should have been already pre-allocated by the device based on its initial configuration.

If the state of Figure 8-26 is subsequently modified such that a connection is established from the node application plug labelled *3*, a new isochronous sequence and hence a corresponding node controller plug would have to be created and associated with the node application plug. Whether or not this node controller plug is created successfully, depends on the availability of isochronous bandwidth. This additional bandwidth required for creating an extra sequence is given by the formula:

$$\Delta bw = DR \times blocks \qquad (10)$$

where the parameter *DR* specifies the rate of data transmissions and the parameter *blocks*, the maximum data blocks of the packets of the isochronous stream. This formula is derived from the original bandwidth calculation formula give in equation (4). The updated Enabler model is given below.

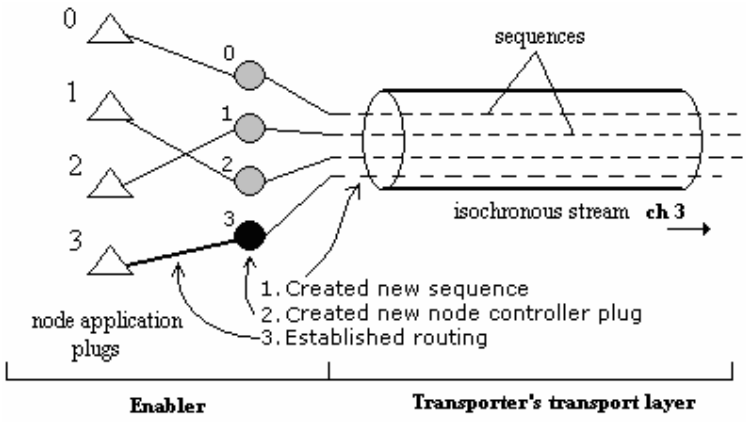


Figure 8-27: A node application plug assigned to a newly created node controller plug

The number of sequences transmitted by the isochronous stream of the device's transport layer is increased by one and the additional bandwidth required is allocated. If this operation is successful, the Enabler then models this sequence by creating a node controller plug for it and then associates it with the node application plug to be connected. In performing this association, the corresponding host application plug of the hardware is configured appropriately.

8.2.1.1 Enabler Implementation

The algorithm implemented by the Enabler to handle dynamic sequence allocation is part of the connection routine of node application plugs. This routine is illustrated in the pseudo-code shown below. Recall from section 7.2 – the section that described the object model of the Enabler – that both node application plugs and node controller plugs are types of mLAN device plugs, and hence are capable of interconnecting.

```
0    NodeApplicationPlug::Connect ( pPlug )
1    {
2        myCtrlPlug = GetNodeControllerPlug( );
3        if ( !myCtrlPlug ) {//If a node controller plug does not exist
4          //First get an available node controller plug
5          myCtrlPlug = pNodeApplication→GetAvaliableNodeControllerPlug( );
6          if ( !myCtrlPlug ) {
```

```
7                   //Create one if there is none avaliable
8                   myCtrlPlug = pNodeController→CreatePlug( );
9             }
10            if ( myCtrlPlug ) {
11
12                  SetNodeControllerPlug (myCtrlPlug) //Set the plug
13            }
14          }
15
16          if ( myCtrlPlug ) {
17
18                  myCtrlPlug→Connect ( pPlug ); // Perform connections
19          }
20    }
```

The attribute *pPlug* represents a pointer to a plug object of the target plug to which the node application plug is to be connected. This plug object specifies either a node application plug or a node controller plug that is modelled by the Enabler for the target device.

Line 2 of this routine retrieves the node controller plug object associated with the node application plug. If one does not exist, an attempt is made to retrieve an available node controller plug (indicated by line 5). An available node controller plug is defined to be a node controller plug that is not in use by any node application plug as described in Figure 8-26. If this operation fails, the node controller component is requested to create a plug, which ultimately results in creating an isochronous sequence and allocating the extra bandwidth required.

The retrieved node controller plug, either previously available or newly created, has to be associated with the node application plug; this is indicated by line 12 of the algorithm. This association is two-fold. Firstly, at the hardware level, the HAL implementation of the Transporter device configures the routing between the host plug and the isochronous sequence modelled by the acquired node controller plug. If this is completed successfully, an association is then made between the node application plug object and node controller plug object modelled by the Enabler. After performing this step, the node controller plug is then connected to the specified plug, *pPlug*, according to the “Connect mLAN Plugs” sequence diagram described in section 5.3.2.3.

Dynamic sequence allocation allows devices to allocate isochronous resources on-the-fly as and when needed. By doing this, any allocated isochronous resources are used up first, before allocating any more. The next two sections describe how the Enabler allows these allocated resources to be released either by pre-configuring isochronous sequence or performing sequence optimization. The pre-configuring technique is discussed in the next section.

8.2.2 Pre-configuring Isochronous Sequences

Pre-configuring isochronous sequences is used when the transmission plugs of a Transporter device are known in advance. This technique allows an application to reserve a fixed amount of isochronous bandwidth that is directly related to the number of audio ‘channels’ being transmitted by a device. These plugs are pre-configured with the intention that they would be used exclusively throughout the device’s transmissions.

From an application, a user can pre-configure the transmission status of a device by specifying which output plugs of the device are required to be pre-configured. The Enabler, on receiving this request, allocates the minimum resources required for transmissions on these plugs.

The plugs specified by an application can be either node application plugs or node controller plugs. If node application plugs are specified, any plugs can be selected and node controller plugs will be assigned to these. The flexibility of selection is due to the dynamic patching that exists between plugs of the node application and plugs of the node controller, and also depends on a patch capability at the hardware level. This is illustrated in Figure 8-28 below.

The top left part of the figure shows the initial configuration of a device. This device has four host plugs and an isochronous stream that transmits one sequence, for which the Enabler creates four node application plugs and one node controller plug. The node controller plug is assigned to the first node application plug. For this device, the node application plugs *1* and *3* have been requested by an application to be pre-configured for dedicated transmissions.

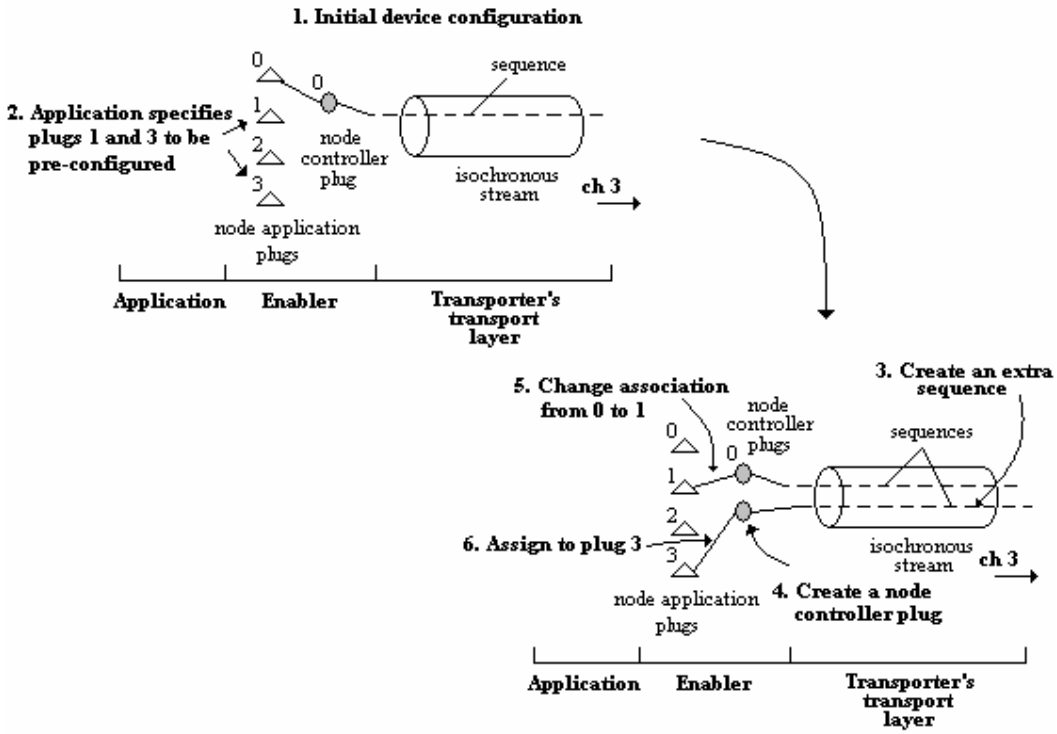


Figure 8-28: Illustrating pre-configuration of node application plugs

In handling the request, the Enabler determines the minimum number of node controller plugs required for pre-configuration. In this case, two node controller plugs are required, which causes the Enabler to create an extra node controller plug to model the extra sequence created by the device. The node controller plug *0*, previously assigned to node application plug *0*, is now assigned to node application plug *1*. Likewise the new node controller plug created is assigned to node application plug *3*.

If a node application component for a device does not exist, an application only has access to the device's node controller plugs. In view of this, if node controller plugs are specified to be pre-configured, the flexibility in the choice of plugs to which resources should be allocated is restricted. Only the desired number of node controller plugs, not a particular selection, is allowed to be specified. This restriction is because the node controller plugs modelled by the Enabler directly correspond to isochronous sequences, and pre-configuring in this case only requires an increase or decrease in the number of node controller plugs. This is illustrated below.

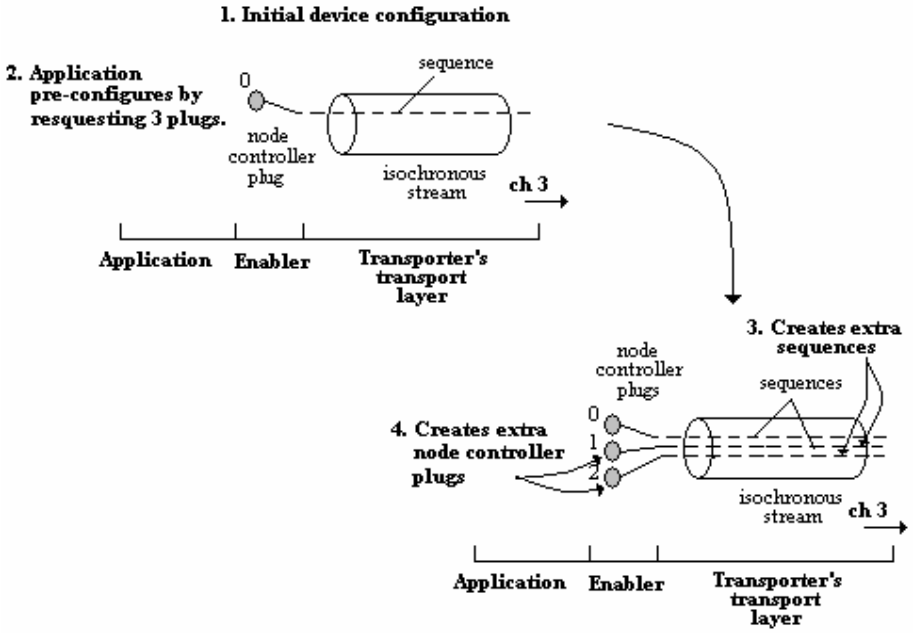


Figure 8-29: Illustrating pre-configuration of node controller plugs

The top left part of the figure shows the initial configuration of a device. It has an isochronous stream with one transmitting sequence, which is modelled by the node controller plug created by the Enabler. An application pre-configures the device by requesting three node controller plugs. Note that in this case, the only options for pre-configuring are increasing or decreasing the number of node controller plugs. In handling this, the Enabler determines the number of node controller plugs to create. In this case, two extra node controller plugs are created to model the two extra sequences created by the device.

8.2.2.1 Enabler Implementation

The Enabler implements the above mechanism using a *struct* that enables applications to specify the plugs to be pre-configured. The name of the struct is *MLANSourcePlugConfig* and is defined below:

```
typedef struct {

  UInt32        numPlugs;
  UInt32        *aPlugType;
  UInt32        *aPlugID;
} MLANSourcePlugConfig, * MLANSourcePlugConfigPtr;
```

Figure 8-30: Enabler-defined struct that allows applications specify output plugs

The *numPlugs* field indicates the number of plugs to be pre-configured. The *aPlugID* and *aPlugType* fields are pointers to parallel arrays, which are meant to indicate the plug ID and the respective plug type of the plugs to be pre-configured.

An example of a *MLANSourcePlugConfig* entry specified by an application that pre-configures a set of node application plugs would be:

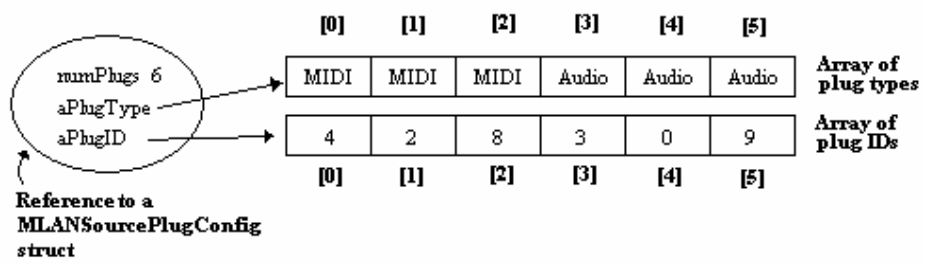


Figure 8-31: An example node application plug configuration

Six node application plugs are specified. The first three of these plugs are MIDI plugs with plug ID *4*, *2* and *8* respectively, the remaining plugs are audio, each having a plug ID of *3*, *0* and *9* respectively. Notice the flexible order in which the plugs have been specified.

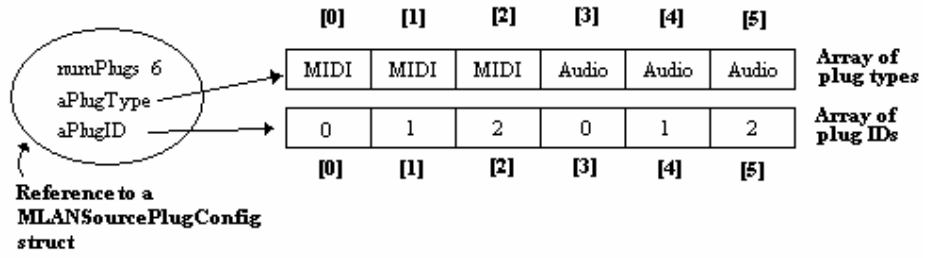


Figure 8-32: An example node controller plug configuration

Figure 8-32 above shows an example of a *MLANSourcePlugConfig* entry specified by an application that pre-configures a set of node controller plugs. This also specifies six plugs, of which the first three are MIDI plugs and the remaining audio. However, note that the plugs have been specified sequentially by the plug IDs. This, in essence, also specifies the number of node controller plugs required for each plug type; from the diagram, three node controller plugs are to be allocated for MIDI and three node controller plugs for audio.

Pre-configuring node application plugs

The Enabler implements pre-configuration of isochronous sequences for node application plugs according to the pseudo-code given below and is explained in the paragraph that follows.

```
0   NodeApplication::SetSourceSequences ( pSourcePlugConfig )
1   {
2       numNodeControllerPlugsRequired = 0;
3
4       numAppPlugs = GetNumNodeApplicationPlugs ( );
5       for plugID : 0 to numAppPlugs { // Release resources of unspecified plugs
6
7           GetPlug(plugID, &pPlug)
8           CachePlugConnections(pPlug)
9
10          plugSpecified = IsPlugSpecified (pPlug, pSourcePlugConfig)
11          if (plugSpecified) {
12
13              numNodeControllerPlugsRequired++;
14          }
15          else {
16
17              DisconnectTargetPlug(pPlug)
18              pPlug → ReleaseNodeControllerPlug()
19          }
20      }
21
22      pNodeController → SetNumberOfSourcePlugs(numNodeControllerPlugsRequired)
23      for plugID : 0 to numAppPlugs { // Assign node controller plugs
24
25          GetPlug(plugID, &pPlug)
26          plugSpecified = IsPlugSpecified (pPlug, pSourcePlugConfig)
27          if (plugSpecified) {
28
29              if (pPlug→HasNodeControllerPlug()) {
30
31                  pPlug → ReleaseNodeControllerPlug()
32              }
33              aNodeCtrlPlug = pNodeController → GetAvaliablePlug();
34              pPlug → SetNodeControllerPlug(aNodeCtrlPlug)
35
36              ReconnectPlugFromCache(pPlug);
37          }
38      }
39  }
```

A reference to the *MLANSourcePlugConfig* struct that specifies the plugs to be configured is passed as *pSourcePlugConfig* as part of the method call. The minimum number of node controller plugs is calculated using the variable

numNodeControllerPlugsRequired, which is initially set to zero. The pseudo-code consists of two main parts.

The first part, given by lines 4 to 20, is responsible for caching the connection information of all the node application plugs of the device. It keeps an accumulative count of the number of node controller plugs required to be set to the device's node controller component for any node application plugs that are found to be specified within *pSourcePlugConfig*. Also, the node controller plugs for any node application plugs that are not specified are released. The plug connections are cached in line 8, the number of node controller plugs required is monitored in line 13, and the releasing of node controller plugs for unspecified node application plugs performed in lines 17 and 18.

The second part of the algorithm, described by lines 22 to 38, assigns node controller plugs to the specified node application plugs of the device. In achieving this, the node controller component is configured to have the required number of node controller plugs (line 22). For a specified node application plug, an available node controller plug is retrieved and then set to the node application plug (lines 33 and 34). If the node application plug previously had a node controller plug, it is first released. After assigning a node controller plug, the target plug connections (if any) previously held by the node application plug are re-established (line 36).

Pre-configuring node controller plugs

In pre-configuring node controller plugs of a device, the Enabler determines the number of node controller plugs specified by an application and then sets it to the node controller component of the device as illustrated below.

```
0    NodeController::SetSourceSequences ( pSourcePlugConfig ) {
1
2        numPlugs  = CalculateNumPlugs(pSourcePlugConfig)
3        SetNumberOfSourcePlugs(numPlugs  )
4    }
```

Pre-configuring isochronous streams is a useful technique that may be more desirable in certain operating environments than dynamic sequence allocation. Such

environments include sound installations, where a dedicated number of plugs are to be used for audio/MIDI transmissions. In addition to the techniques described thus far, the Enabler also has the capability of optimizing the isochronous sequences in use by devices, either performed automatically by an application or manually by user request. This is another isochronous resource management technique that ensures that devices, at any one time, make optimal use of isochronous resources. This technique is described in the next sub-section.

8.2.3 Optimizing Isochronous Sequences

Optimizing isochronous sequences is the technique used by the Enabler to release isochronous sequences and hence isochronous resources that have been allocated for node application plugs that do not have any target connections. In other words, this procedure ensures that only node application plugs with known target plug connections are granted isochronous resources.

This operation can be invoked at any time during the transmitting state of a device, which then performs *dynamic sequence de-allocation* in fulfilling isochronous sequence optimization. In handling this operation, the node application plugs of a device that have target connections are identified and isochronous resources reserved for only these plugs.

This may require a reshuffling of the node controller plugs and hence the currently transmitting isochronous sequences assigned to these node application plugs, which then ultimately leads to reconfiguring the A/M reception parameters of any connected plugs. Recall from the “Connect mLAN Plugs” sequence diagram discussed in section 5.3.2.3 that the A/M parameters required for plug connections are the *isochronous channel*, *sequence number* and *subsequence number*. If the node controller plugs originally in use by a set of node application plugs are reshuffled, either one or more of the following parameters: *isochronous channel*, *sequence position* or *subsequence position* of the logical channel previously referenced by the node application plug may change, which then implies that the configuration of the target plugs originally set to receive on these node application plugs would have to be modified. This is illustrated using the figure below.

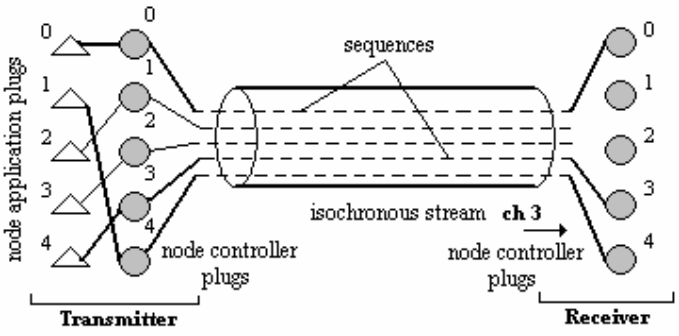


Figure 8-33: A device with un-optimized isochronous sequences

Assume the initial plug configuration shown in Figure 8-33. The node application plugs shown all have associated node controller plugs. The node application plugs labelled *0*, *1* and *4* have target connections to node controller plugs *0*, *4* and *3* respectively of the receiver. The node controller plugs assigned to node application plugs *2* and *3* are allocated bus bandwidth which is not being made use of.

After performing the optimization procedure, the illustration shown in Figure 8-33 now becomes Figure 8-34 shown below. In this figure, the number of transmitting sequences and hence node controller plugs of the transmitter, is reduced to 3. This value corresponds to the number of the transmitter's node application plugs that have target connections.

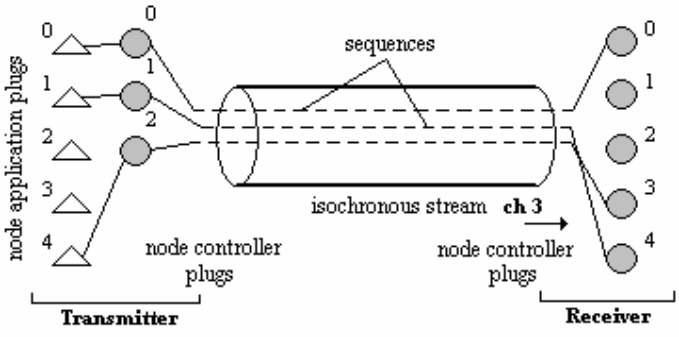


Figure 8-34: Optimized isochronous sequences

The sequence assignments of these plugs have been reshuffled to make use of the current number of sequences. Node application plug *1*, previously assigned to the 5th sequence is now assigned to the 2nd sequence. Similarly plug *4*, previously assigned to the 4th sequence, is now assigned to the 3rd sequence. The A/M reception parameters

of the target plugs of these node application plugs have been modified accordingly. Node controller plug *3* of the receiving device, previously configured to receive on the 4th sequence is now configured to receive on the 3rd sequence; similarly the receiving sequence position of plug *4* has been changed from the 5th to the 2nd. The reduction in the number of transmitting sequences from 5 to 3 resulted in an increase in available bandwidth, which can be used by other devices that require isochronous transmissions.

8.2.3.1 Enabler Implementation

The Enabler implements isochronous sequence optimization as part of the implementation of the *SetSourceSequences (…)* method of the node application component modelled for a Transporter device. The pseudo-code that describes this algorithm is given below and is explained in the paragraph that follows.

```
0    NodeApplication::SetSourceSequences ( NULL )
1    {
2    …
3        numAppPlugs = GetNodeApplicationPlugCount ( );
4        for plugID : 0 to numAppPlugs { //Get number of node controller plugs required
5
6            myPlug = GetPlugObject( plugID )
7            myPlugConnectionList[ plugID ] = myPlug→GetConnections();
8
9            if ( myPlugConnectionList [ plugID ].numConnections == 0 ) {
10
11               myPlug→ReleaseNodeControllerPlugs( ); //Release plug if no connections
12           }
13           else {    numNodeControllerPlugsRequired++;   }
14       }
15
16       for plugID : 0 to numAppPlugs {
17           //Assign node controller plugs to node application plugs with connections
18           plugConnections = myPlugConnectionList[ plugID ];
19           if ( plugConnections.numConnections == 0 ) { //  no connections
20
21               continue;
22           }
23           myPlug = GetPlugObject( plugID );
24           myNodeCtrlPlug = myPlug→GetNodeControllerPlug ( );
25
26           if ( myNodeCtrlPlug→GetPlugID( ) >=  numNodeControllerPlugsRequired ) {
27               //Get available node controller plug and assign
28               newNodeCtrlPlug = GetAvaliableNodeControllerPlug ( );
29               myPlug→DisconnectTargetConnections ( ); //Reestablish connection
30               myPlug→ReleaseNodeControllerPlugs ( );
31               myPlug→SetNodeControllerPlugs ( newNodeCtrlPlug );
32               myPlug→ReEstablishTargetConnections ( );
```

```
33            }
34        }
35        //Set the number of node controller plugs to the node controller
36        pNodeController→SetNumOutputPlugs( numNodeControllerPlugsRequired );
37    …
38    }
```

Notice the *NULL* parameter used in the function call. A *NULL* parameter is an indication to the Enabler to perform isochronous sequence optimization. If a valid parameter is specified, a parameter of type *MLANSourcePlugConfig*, then pre-configuration of isochronous sequences is performed.

This algorithm can be best described to consist of three main parts. The first part of the algorithm, described by lines 0 through to 14, caches the connection information of all the output node application plugs, while incrementing the number of node controller plugs required to be set to the device's node controller component for any node application plug found with a valid target connection. Also, node controller plugs are detached from any node application plug that is found not to have target connections. Referring to the example used in Figure 8-33, this would result in detaching the node controller plugs assigned to node application plugs *2* and *3*, after which three node controller plugs would be required for optimization.

In the second part of the algorithm, described by lines 16 through to 34, the connection information obtained from the first part is explored. For node application plugs that have known target connections, the plug ID of the associated node controller plug is examined to determine whether it falls within the desired number of node controller plugs. If this test fails, an available node controller plug is obtained. This returns the first node controller plug (searching from position 0) that is not associated with any node application plug, and is guaranteed to always return a node controller plug with a plug ID value that is less than the desired number of node controller plugs. After this step, all target connections to the node application plug are momentarily disconnected, its node controller plug released, the newly obtained node controller plug set to the node application plug, and then the target connections restored. Again, referring to Figure 8-33, this would result in node application plugs *1* and *4* of the transmitter to be assigned to node controller plugs *1* and *2* respectively.

Also the A/M reception parameters of the corresponding target plug are configured appropriately.

The final part of the algorithm indicates to the node controller component of the device to set the desired number of node controller plugs. The node controller component is responsible for releasing any unwanted node controller plugs and in so doing, freeing any isochronous resources that are not in use by the plugs. The outcome of this step is as illustrated in Figure 8-34, where the fourth and fifth node controller plugs have been released.

Isochronous sequence optimization releases all the unused sequences of a Transporter device. However it may be desirable in certain operating environments, to be able to manually release isochronous resources in use by a particular unconnected output plug. Input plugs that have been configured for reception in the absence of a transmitter can also be deactivated by this process. These plugs are referred to as *dangling plugs* and they have the potential to accidentally transmit or receive data. An application can achieve this kind of specific optimization by using the pre-allocation technique discussed previously. It specifies to the Enabler all the desired plugs and not the plug(s) for which isochronous resources are to be released.

It is important to mention that the three forms of isochronous resource management techniques described here effectively result in an increase or decrease in the number of transmitting sequences. In a bridged environment, where across-bus plug connections are established, if the payload size of an isochronous stream is changed as a result of performing isochronous resource management, the Enabler also modifies the stream control registers of the listener portals receiving the isochronous stream to contain an updated payload size. In doing so, extra bandwidth is allocated or released on the intermediate buses receiving the stream.

The next section describes the word clock synchronization component of the Enabler and how applications can easily establish word clock synchronization between devices.

8.3 Enabling Word Clock Synchronization

An mLAN device configured to be a word clock slave receives synchronization data via the SYT field of isochronous packets in an isochronous stream. This typically involves configuring one of the device's registers (an SYT related register) to contain the isochronous channel of an isochronous stream that contains SYT timing information.

The level of word clock synchronization implemented on mLAN devices depends on the device type, with the first generation of mLAN devices (mLAN Version 1) having the ability to self-configure themselves for word clock synchronization. Transporter devices, on the other hand, are not able to do so. They do expose an SYT-related register, to which a controlling application can specify an isochronous stream's channel. This implies that a controlling application, usually an Enabler, is responsible for implementing word clock operations on behalf of Transporters. These word clock operations include those that are required by connection management applications such as:

- Determining whether an mLAN device is acting as a slave, or whether it is capable of acting as a slave or master.
- Determining the presence of either the master capable or slave capable word clock signal.
- Retrieving a list of supported sampling rates, as well as setting the synchronization sampling rate.
- Determining the word clock master assigned to a slave device, or the word clock slaves configured to receive synchronization from a corresponding master device.

The new Enabler provides high-level word clock operations for Transporter devices according to the above mentioned requirements. The adopted implementation strategies are discussed in the following subsections.

8.3.1 Determining Master or Slave Capability

Devices receiving audio and music data, defined by IEC 61883-6 [IEC, 2005], are required to implement a mechanism that extracts SYT timing information from an incoming isochronous stream that contains CIP-formatted isochronous packets. The extracted SYT information is used to regenerate a word clock signal that is identical to the word clock signal in use by the device generating the SYT information. This is described by the figure below. Note that the technique employed by a device in regenerating sample rates from SYT information has been discussed in section 3.3.

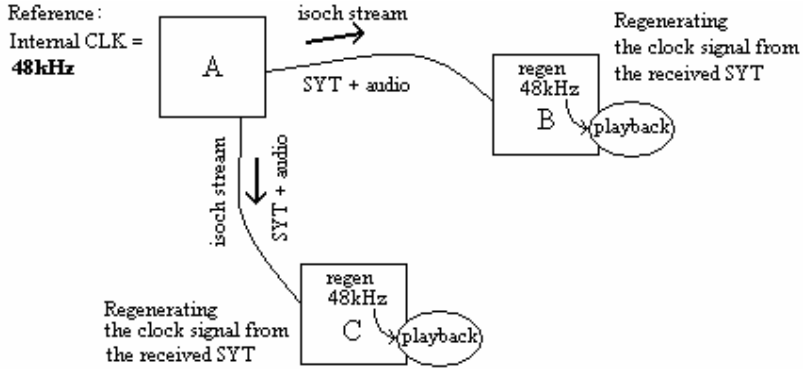


Figure 8-35: Synchronization between devices on a network

Three devices, labelled *A*, *B* and *C*, are shown. Device *A* transmits an isochronous stream that contains SYT time stamps as well as some audio samples. These time stamps are based on its 48 kHz internal clock signal. Device *B* and *C* both receive the isochronous stream that is being transmitted by device *A*. The received SYT time stamp information is used to regenerate a corresponding 48 kHz clock signal, which is then used by the device for playing back the received audio samples.

For an application to be able to set up a reference or a master word clock signal and a number of associated word clock slaves, the Enabler should be able to present to applications the devices that are capable of acting as word clock masters or word clock slaves. The capability of a device to act as a word clock master is fundamentally based on the ability of the device to transmit an isochronous stream that contains valid SYT timing information. Other rules are used by the Enabler in defining a master capable device (see section 8.3.1.1). Similarly, the capability of a device to act as a word clock slave is fundamentally based on the ability of the device to be able to

receive and process SYT timing information. Also, the Enabler introduces other governing rules that define a slave capable device (see section 8.3.1.2). The rules utilized by the Enabler for defining a master capable and a slave capable device are discussed below.

8.3.1.1 Rules for Defining Word Clock Master Capability

The rules used by the Enabler for defining the word clock master capability of a device are:

1. The device must be capable of transmitting an audio data sequence.
2. The device's word clock should not be locked to a reference signal.

The first point, if true, implies that the SYT timing information carried by the transmitted isochronous stream of the device is valid. This assumption is based on the fact that isochronous streams must contain SYT timing information that indicate the presentation time of the audio data carried by the stream [IEC, 2005]. The second point ensures that for a device to be master capable it should not be receiving word clock synchronization from another source. In order to do this, the Enabler examines the value of the SYT-related register of the Transporter device. If the register value has a channel number between 0 and 63, it implies that it has been configured to receive SYT information and hence is acting as a word clock slave.

8.3.1.2 Rules for Defining Word Clock Slave Capability

The rules used by the Enabler in defining the word clock slave capability of a device are:

1. The device must be capable of receiving an audio data sequence.
2. The device's word clock should not be a reference signal to other devices.

These points correspond to the points for master capability. If a device is capable of receiving an audio sequence, it implies that the device is capable of processing SYT time stamp information. The second point ensures that for a device to be slave capable its word clock should not be used as a reference for other devices. It is important to mention that it is possible (in practice) for a slave to also act as a word clock master to

another device. This feature is not implemented by the Enabler for the sake of simplicity.

The next section illustrates the use of these rules by the Enabler in determining the master/slave capability of a device.

8.3.1.3 Illustrating the Master/Slave Capability Rules

Consider the 3-device setup shown in Figure 8-36. In this setup the devices are capable of transmitting and receiving audio and have been configured to make use of their internal word clock sources for synchronization. The master capable and slave capable states of these devices are given in Table 8-9.

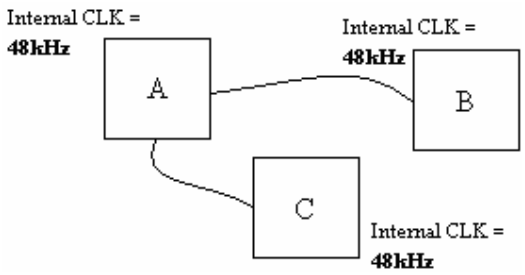


Figure 8-36: Synchronization setup, devices using internal clock

Device	Is Master Capable?	Is Slave Capable?	Reason
A	Yes	Yes	The device's word clock is neither used as a reference signal nor locked to a reference signal.
B	Yes	Yes	The device's word clock is neither used as a reference signal nor locked to a reference signal.
C	Yes	Yes	The device's word clock is neither used as a reference signal nor locked to a reference signal.

Table 8-9: Master and slave capable state of the devices shown in Figure 8-36

Figure 8-37 shows a slightly altered word clock synchronization configuration. Device *A* is now configured to act as a reference clock signal to device *C*. The

configuration of device *B* has not changed. The corresponding master capable and slave capable state of these devices is given in Table 8-10.

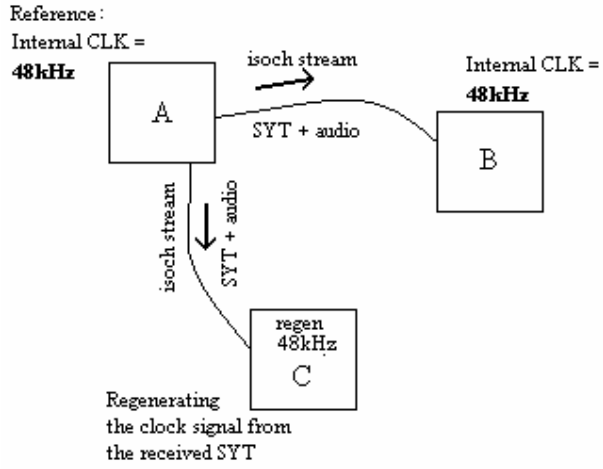


Figure 8-37: Synchronization setup, one master, one slave and one neutral

Device	Is Master Capable?	Is Slave Capable?	Reason
A	Yes	No	The device's word clock is not locked to a reference signal, but it is acting as a reference to device C, hence making it not slave capable.
B	Yes	Yes	The device's word clock is neither used as a reference signal nor locked to a reference signal.
C	No	Yes	The device's word clock is locked to a reference signal, hence making it not master capable. However, its word clock is not acting as a reference to other devices.

Table 8-10: Master and slave capable state of the devices shown in Figure 8-37

Applications can retrieve the master/slave capability of an mLAN Transporter device by calling the *IsMasterCapable(…)* or the *IsSlaveCapable(…)* method of the *MLANDeviceSyncBlock* object implemented by the device's *MLANDevice* object. These capabilities can be represented accordingly by the application.

A further word clock functionality implemented by the Enabler is to allow applications to monitor the word clock master/slave signals of a device. This is described in the next section.

8.3.2 Monitoring Master or Slave Signals

If a device is master or slave capable it invariably implies that the device is capable of transmitting or receiving synchronization information. The real time status of the transmission or reception of the SYT synchronization information by a device should be reflected by applications, thus making it easier to detect and diagnose word clock synchronization errors.

For example, assume a device, *X*, has been configured successfully to receive word clock synchronization that is generated from the internal clock of another device, *Y*. If the clock source of device *Y* has been changed to an external input with no signal, device *X*, even though successfully configured, will not receive any synchronization information from device *Y*. If an application reflects the real time synchronization status of these devices, it can then indicate the absence of the master and the slave word clock signals of device *Y* and *X*, respectively.

Monitoring the status of word clock signals is a feature that has to be provided by an actual Transporter implementation. Transporter nodes, via a suitable HAL implementation, should allow the Enabler to probe for the presence of incoming SYTs as well as determining whether the received SYTs are locked to an isochronous stream. The *MLANDeviceSyncBlock* object of an mLAN Transporter's *MLANDevice* implementation of the Enabler, implements the methods *IsSlaveCapableClockDetected(…)* and *IsMasterCapableClockDetected(…)* that allows applications to retrieve the synchronization status of a device.

Recall from section 7.2.3.2, that the *MLANDeviceSyncBlock* object is actually a pointer to an object of the class *NodeControllerSYNC*, or the class *NodeApplicationClock*. The *NodeControllerSYNC* class models the word clock implementation of the transport layer of an mLAN Transporter device, while the *NodeApplicationClock* class models that for the host implementation of a Transporter

node. Also recall that the Enabler creates both a node application component (if possible), and a node controller component for an mLAN Transporter device, and if the node application component is created, it makes use of the node controller component to access the transport layer of the mLAN Transporter device. Revisit section 7.1.3 of chapter 7 to review the node application/node controller concept.

The sequence diagram shown in Figure 8-38 below describes how the Enabler provides word clock monitoring functionality to user applications.

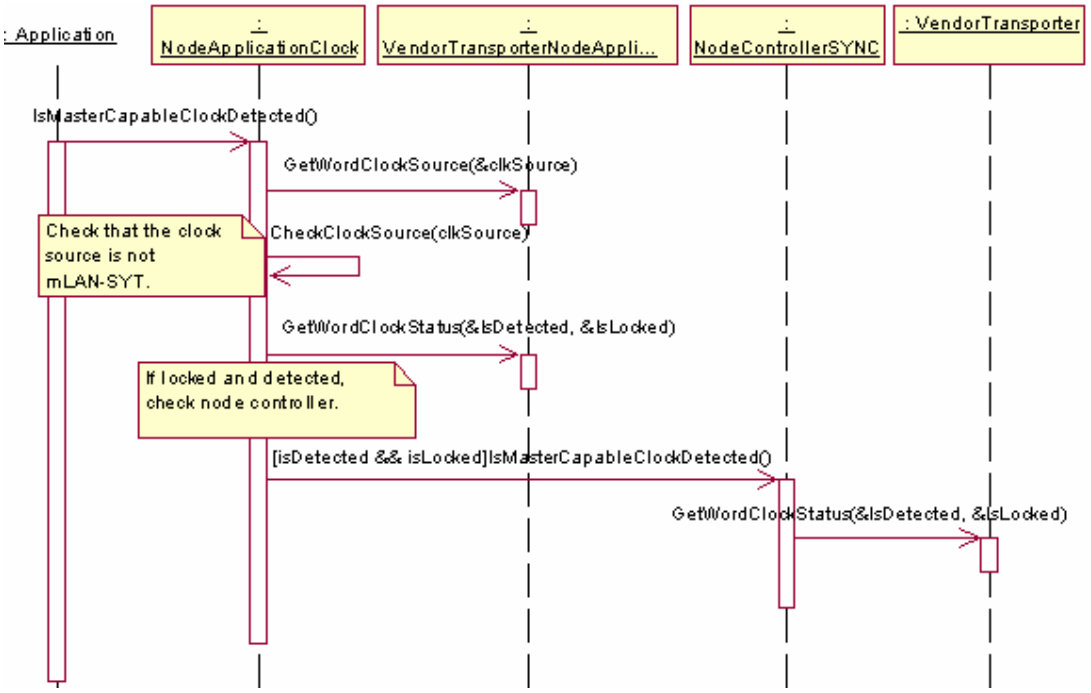


Figure 8-38: Monitoring a word clock master signal

This diagram describes the operations that occur when the word clock signal of a device that is configured as master, is to be monitored. If the *MLANDeviceSyncBlock* object of the device's *MLANDevice* implementation references a *NodeApplicationClock* object, as shown in the figure, a check is made to the host implementation of the Transporter node to verify that the clock source is not mLAN-SYT (shown by the 2nd and 3rd sequences).

If this holds true, the word clock status of the current clock source of the host system is retrieved – this indicates whether the word clock signal is *detected* and *locked*. If

found to be detected and locked, the corresponding node controller word clock implementation, given by the *NodeControllerSYNC* object is queried to determine if it is receiving the word clock signal. This further queries the HAL Transporter object (*VendorTransporter*) for the required information. If all these checks pass, a positive response is then returned to the application, otherwise a negative response is returned. For Transporter nodes that do not expose their host implementations, retrieving the word clock status information for a master signal only results in querying the transport layer of the device in order to determine whether a clock signal is being received. In other words the last two sequences of Figure 8-38 are the only operations performed, from which an appropriate response is returned to an application.

A similar set of operations to that described above is performed when retrieving the word clock monitoring status of Transporter devices configured to be slaves. This is described by the sequence diagram given in Figure 8-39.

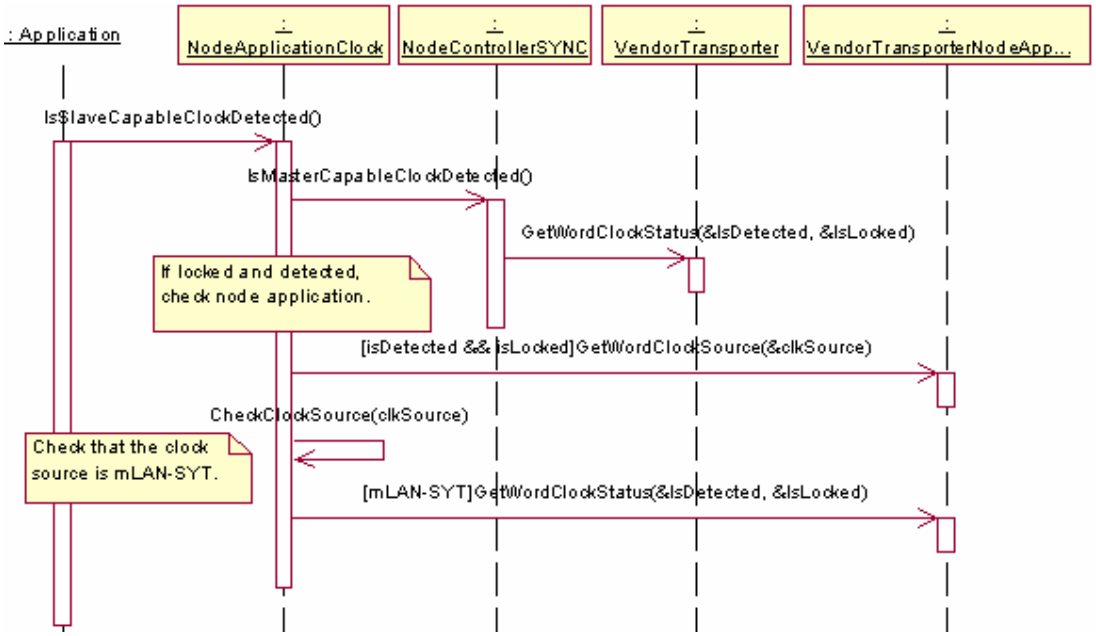


Figure 8-39: Monitoring a word clock slave signal

An application initiates this by calling the *IsSlaveCapableClockDetected(…)* method of the *NodeApplicationClock* object, which is presented as an *MLANDeviceSyncBlock* object. In this case, the node controller word clock implementation is first queried to determine whether it is receiving valid SYTs (given by the 2nd and 3rd sequences). If

this returns true, the word clock source of the node application implementation of the device is examined to check whether it is configured to receive from mLAN-SYT. If after performing this step, the clock source is from mLAN-SYT, a check is then performed to determine the status of the clock signal. If all these checks pass, a positive response is returned to the application, otherwise a negative response is returned.

8.3.3 Changing Word Clock Sampling Rates

Being able to set the sampling rates of a word clock signal is an important feature of word clock synchronization. For Transporter nodes to operate effectively, the sampling rate of the word clock source in use by its host implementation must correspond to that of its transport layer. If a Transporter node exposes its node application, an Enabler can successfully synchronize the sampling rates of both the node application of a Transporter node and its transport layer. If the node application is not exposed, the sampling rate of the host application has to be synchronized manually.

The word clock model implemented by the Enabler has been discussed extensively in section 7.1.4. This section introduced a sample rate integrity check operation that verifies whether a particular sample rate value can be set to a device. Recall from this section that in performing a sampling rate check for a device configured as master, the sampling rate check is also performed for the associated slave devices. The Enabler, in changing the sampling rate of a word clock signal, first checks the integrity of the sample rate value to be set. If this operation returns with no error the sampling rate is then applied to the device. If the device is acting as a master then the sampling rate is also applied to the corresponding slaves.

The *MLANDeviceSyncBlock* object of an mLAN Transporter's *MLANDevice* implementation provided by the Enabler, implements the method *SetSampleRate(…)* that allows an application to set the sampling rate of the mLAN Transporter. The implementation of this method is illustrated using the sequence diagram shown in Figure 8-40. Recall the relationship between the Enabler's node application

component implementation, and the node controller component implementation that are modelled for an mLAN Transporter device.

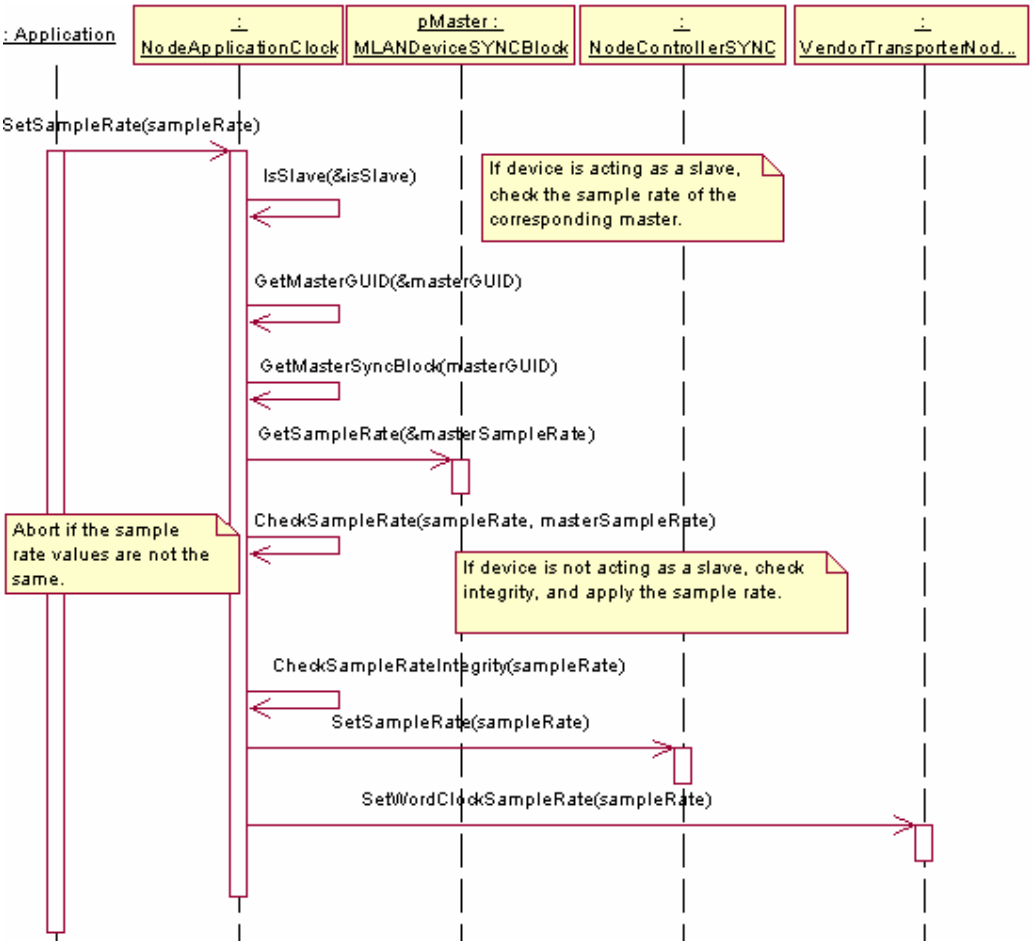


Figure 8-40: Changing the sample rate of a device that exposes its node application

Figure 8-40 assumes that the *SetSampleRate(…)* method is called on an mLAN Transporter device that exposes its node application. The first check performed in setting the sample rate is to determine whether the device is acting as a slave, i.e. receiving word clock synchronization. The sample rate of a slave device is controlled by its master, and hence if acting as a slave, the sample rate value to be set is compared to the sample rate of the corresponding master. If these values differ, an error message is reported. If on the other hand, the device is not acting as a slave, a check is performed to determine the integrity of the sample rate value to be set. The integrity check, as already discussed in section 7.1.4, determines whether the sample

rate value to be set is supported by the device, as well as any corresponding slave devices (if any).

The integrity check also determines whether any resources in use by the device would be lost if the sample rate were to be changed (see section 7.1.4). If the integrity check returns with no error, the sample rate is set to the transport layer of the device via the associated *NodeControllerSYNC* object, and also to the current word clock source of the Transporter's host implementation via the *VendorTransporterNodeApplication* object.

The *SetSampleRate(…)* method call of the *NodeControllerSYNC* object shown in the figure above, performs a number of operations down at the node controller level of a Transporter device. Note that this is also equivalent to an application calling the *SetSampleRate(…)* method on an mLAN Transporter device that does not expose its node application. The sequences of steps that describe the required operations are illustrated in Figure 8-41.

In this case, if the device is acting as a slave, the sample rate value of the corresponding master is also compared against the sample rate value to be set. If the sample rate values are not equal, an error message is reported and the process aborted. On the other hand, if the device is not acting as a word clock slave, the integrity of the sample rate value to be set is checked. If this check passes, the node controller of the Transporter device is configured to apply the sample rate, via the *NodeController* object. In performing this, all the input and output isochronous stream plugs of the Transporter node are configured to implement the sample rate. After performing this operation, a check is performed to determine whether the device is acting as a master, and if it is, the sample rate value is also set to all the corresponding slaves.

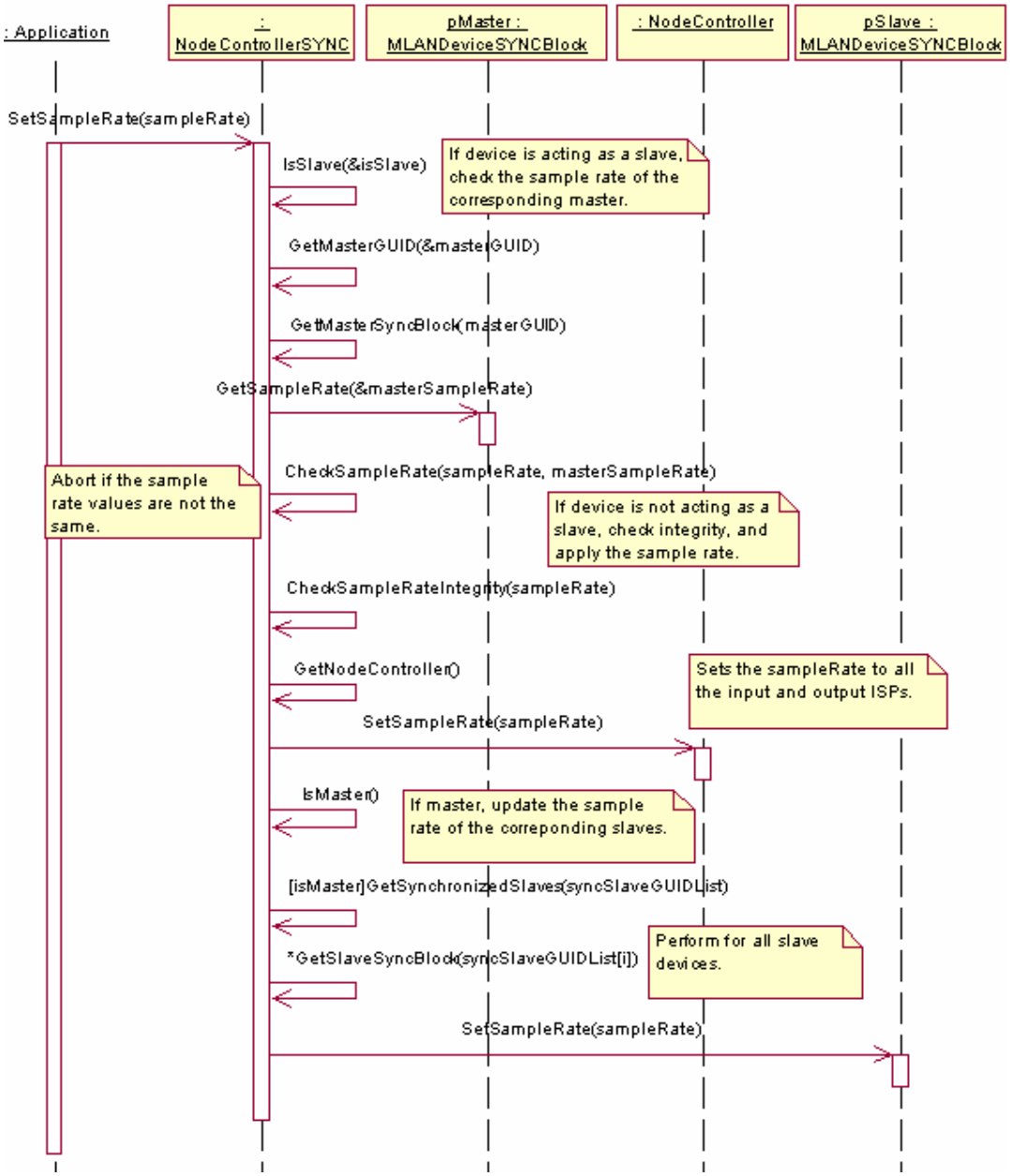


Figure 8-41: Changing the sample rate of a device that does not expose its node application

8.3.4 Making and Breaking Synchronization

The Enabler provides a simple and direct way for applications to establish master/slave relationships between Transporter devices. This is achieved using the *Synchronize(…)* method defined by the *MLANDevice* class. Also, using this method, word clock synchronization between devices can be broken. Recall from sections 7.4.4.2 and 7.4.4.3 that in order to synchronize two devices, the *Synchronize(…)*

method is called on the *MLANDevice* object that is to be configured as slave while passing in as parameter, the *MLANDevice* object of the intended master. To cause a device to stop receiving word clock synchronization, the *Synchronize(…)* method is called on the slave *MLANDevice* object without any parameters, which is an indication to the slave device to synchronize to no other devices, and hence causing it to use its internal clock.

8.3.4.1 Making Synchronization Settings

In achieving synchronization between two devices, the Enabler ensures that the transmitting channel of the isochronous stream containing SYT timing information from a specified master, is set to the SYT-related register of the device to be configured as slave. It also ensures that the nominal sampling rate value of both the master and the slave devices are the same. This process is described in the sequence diagram shown in Figure 8-42, and assumes that the *Synchronize(…)* method is called on a Transporter that exposes its node application.

In handling the synchronization routine, objects of the synchronization block implementation of both the master and slave devices are retrieved. The synchronization is achieved by invoking the *ConnectToSlave(…)* method on the master synchronization block object, passing the slave synchronization block object as parameter, and also by invoking the *ConnectToMaster(…)* method on the slave synchronization block object, passing the master synchronization block object as parameter. In handling the *ConnectToSlave(…)* method of the master synchronization block, a check is performed to determine whether it is master capable, and if it is, the word clock source of the host implementation of the device is configured to use its internal clock. See section 8.3.1.1 for the definition adopted for master capability.

Following this, the synchronization block of the device's node controller is then configured appropriately. This ensures that device transmissions containing valid SYTs are enabled, and that any bridge portals along the path from the master to the slave are configured for across-bus forwarding.

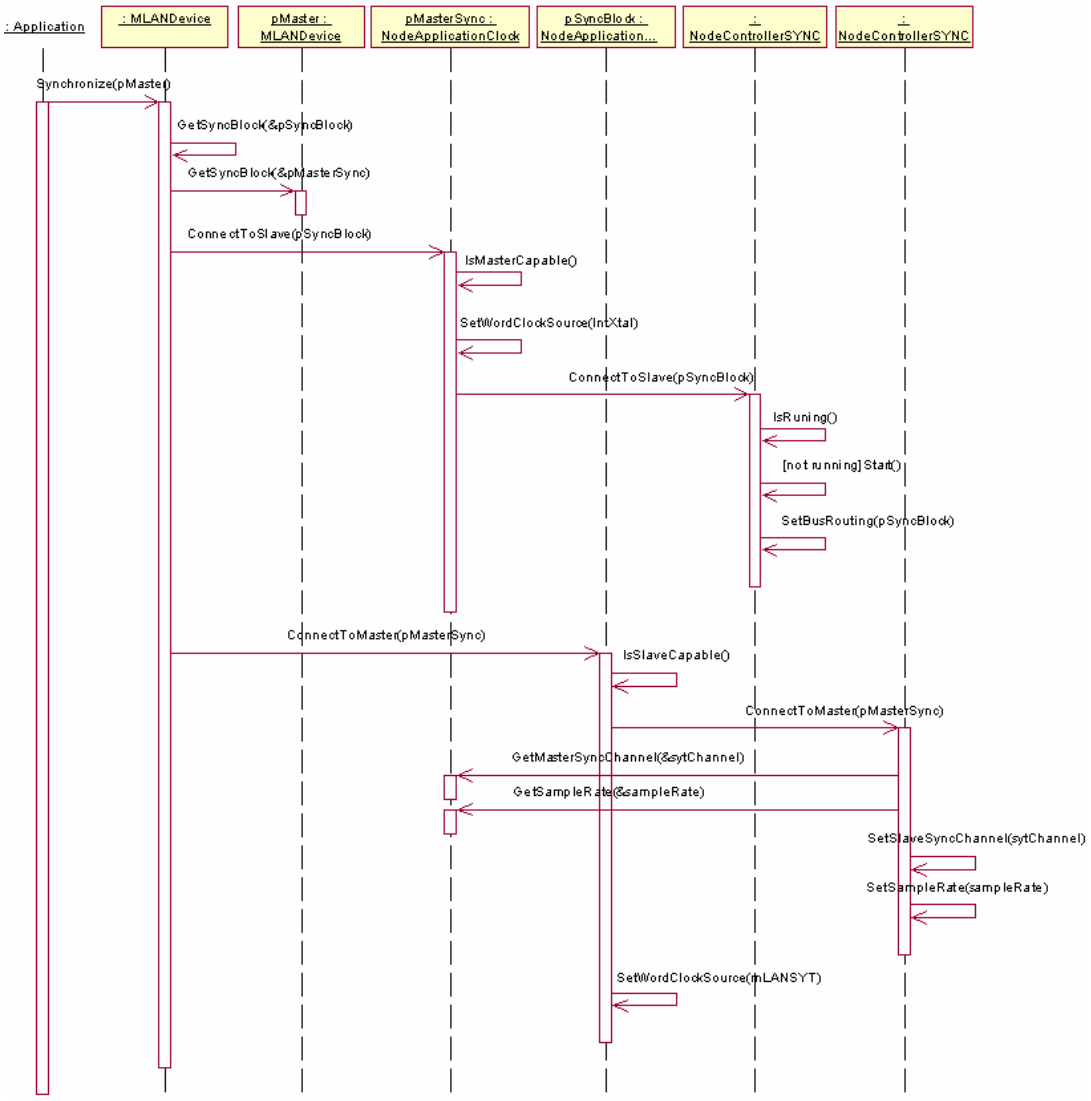


Figure 8-42: Handling word clock synchronization by the Enabler

If the *ConnectToSlave(…)* procedure is handled successfully, the *ConnectToMaster(…)* method of the slave synchronization block is then invoked. In handling this, a check is performed to determine whether the device is slave capable, and if it is, the node controller implementation of the device is configured to receive SYTs. Configuring the node controller requires setting the synchronization channel and the sample rate of the slave device to correspond to that of the master. Following this, the word clock source of the host implementation of the Transporter device is configured to receive on mLAN-SYT.

8.3.4.2 Breaking Synchronization Settings

In breaking word clock synchronization settings of a slave device, the synchronization block object of the slave device and the corresponding master are retrieved and disconnected accordingly. These steps are illustrated in the sequence diagram shown in Figure 8-43.

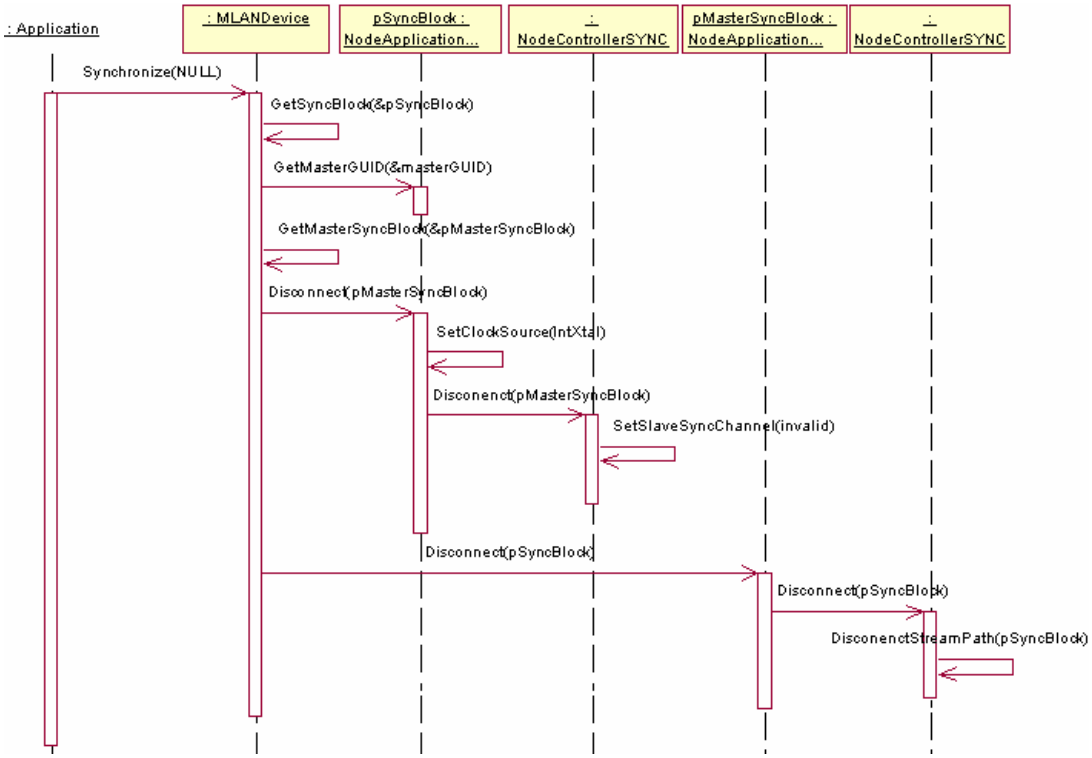


Figure 8-43: Breaking word clock synchronization settings by the Enabler

In handling the *Disconnect(…)* method of the slave synchronization block, the word clock source of the host implementation of the Transporter device is configured to use its internal clock. Following this, the node controller implementation of the device is configured to not receive SYTs. This is achieved by the setting the slave's synchronization channel to an invalid value. The master synchronization block handles disconnections by ensuring that the bridge portals along the path from the master to the slave are configured to stop forwarding isochronous streams. This only happens provided no other device on the same bus as the slave is receiving the isochronous stream either for SYT synchronization, or for extracting audio/MIDI data.

Note that in cases where synchronization is to be established or broken between devices that do not have a node application component implementation, a similar set of operations, excluding method calls to the *NodeApplicationClock* object, are followed.

8.3.5 Retrieving Synchronization Connections

The Enabler also provides the capability of determining the synchronization connection state of devices on a network. In other words, it is capable of determining the word clock master of a particular slave device, and the word clock slaves that have been configured to receive synchronization from a particular master.

The corresponding *MLANDeviceSyncBlock* object of an mLAN Transporter's *MLANDevice* implementation implements the methods *GetMasterGUID(…)* and *GetSynchronizedSlaves(…)* that allow the GUID of a master node to be retrieved for a slave device, and a list of slave devices to be retrieved for a master device. The implementation of these methods is described in the following paragraphs.

8.3.5.1 Retrieving the Master assigned to a Slave Device

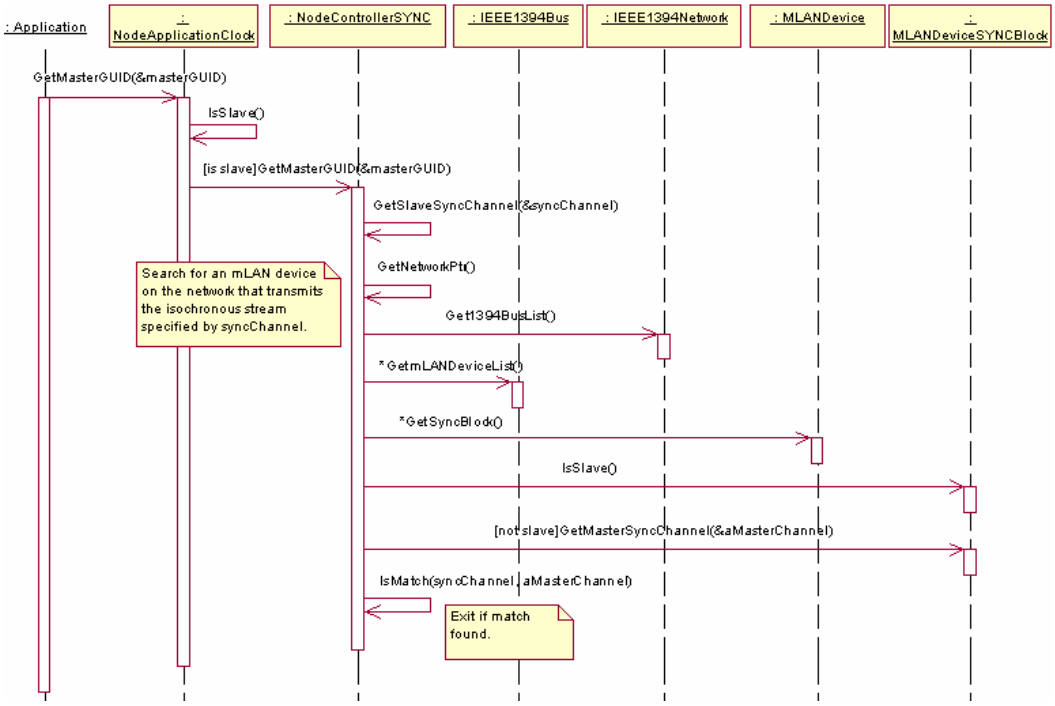


Figure 8-44: Retrieving the word clock master assigned to a slave device

The sequence diagram that describes how the Enabler retrieves the word clock master of a slave device is given in Figure 8-44 above. This is based on the assumption that the *GetMasterGUID(…)* method is called on a device that implements a node application.

First, a check is performed to determine whether the device for which the word clock master is to be retrieved is actually a slave. If this check fails, the process is aborted and an appropriate error message returned. If the device is acting as a slave, the synchronization block of the node controller, given by *NodeControllerSYNC*, is requested to retrieve the GUID of the master node. In achieving this, the *slave synchronization channel* of the device is compared with the *master synchronization channel* of all other devices on the network. The master device is identified when the synchronization channels match. Note that for potential master devices located on a bus remote to the slave, the mapped isochronous channel value of the *master synchronization channel* is used in performing the comparison.

8.3.5.2 Retrieving Slaves of a Master Device

A similar procedure to that described above is used by the Enabler in retrieving a list of devices that are receiving word clock synchronization from a particular master. The sequence diagram that describes this process is given in Figure 8-45 below. This assumes that the *GetSynchronizedSlaves(…)* method is called on a device that implements a node application.

In this approach, a check is performed to determine whether the device for which the slave devices are to be retrieved is master capable. If the device is found not to be master capable, the process is aborted and a suitable error message returned to the application. If the device is found to be master capable, the synchronization block of the node controller is then requested to retrieve the list of slave devices. In performing this, the *master synchronization channel* of the device is retrieved and compared with the *slave synchronization channel* of the other devices on the network. A slave device is identified if there is a match in the synchronization channels. Note that for slave devices located on a bus remote to the master, the mapped isochronous channel value of the *slave synchronization channel* is used in performing the comparison.

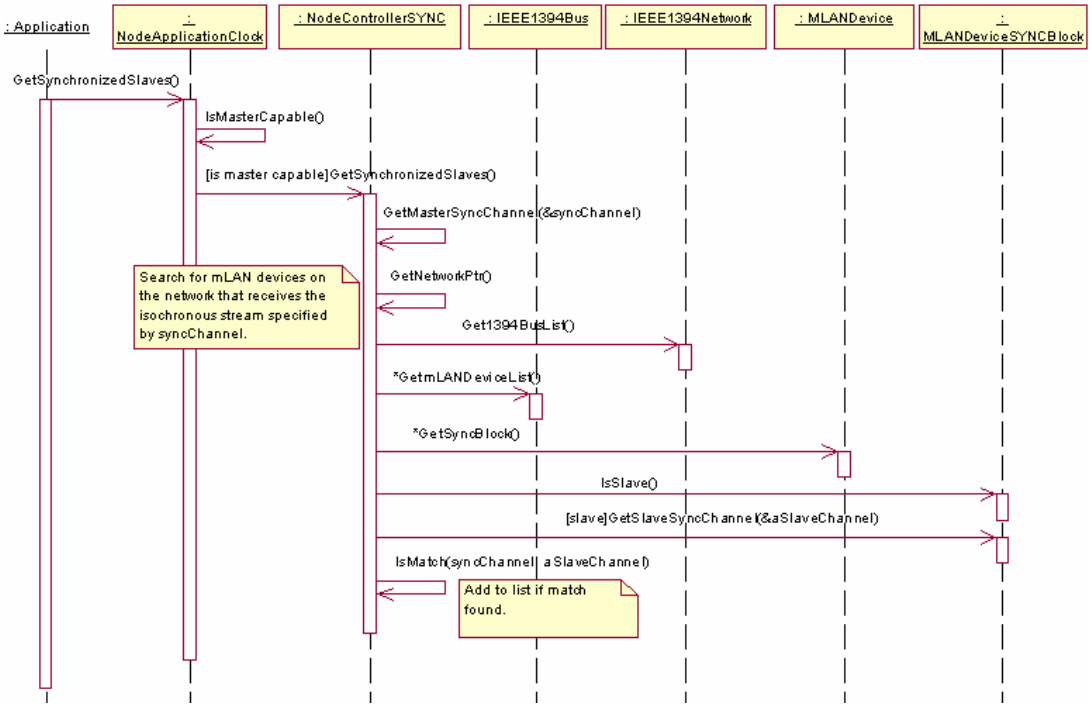


Figure 8-45: Retrieving the word clock slaves configured to a master device

A number of component level innovations of an Enabler implementation have been given thus far. These range from how a multiple bus environment is enumerated and modelled by the Enabler, to the tools provided by the Enabler to properly manage isochronous resources, as well as word clock synchronization. These enhanced Enabler features that result from the new design described in chapter 7 needed to be tested with a sample application.

The next chapter, chapter 9, describes a user-level patch bay application developed to interact with this Enabler. It demonstrates the basic features of the Enabler regarding plug connections, plug disconnections and word clock synchronization. It also demonstrates the enhanced features of the Enabler that result from its node application/node controller implementation. In addition to this, chapter 9 also verifies the stability and performance of the Enabler through a number of test procedures.

8.4 Summary

This chapter describes a number of component level innovations for an Enabler implementation that is based on the new Enabler design. The component level innovations discussed include:

- Ways of representing and modelling IEEE 1394 nodes in a bridged environment.
- Techniques that allow for an effective management of the usage of isochronous resources on a network.
- Providing effective word clock synchronization control on behalf of Transporter nodes on a network.

With respect to representing and modelling IEEE 1394 nodes in a bridged environment, various implementation strategies are described that allow:

- the topology map information of a network to be retrieved and reconstructed by an application,
- the maximum transfer speed to be determined between any two nodes residing on the network, and also
- the routing path of transmitting isochronous streams to be determined.

The isochronous resources that are available on a network for data transmissions can be managed by the Enabler by using one of the following techniques:

- dynamic sequence allocation,
- pre-configuring isochronous sequences, and
- optimizing isochronous sequences.

These techniques are all under user control, and hence grant the user full management over the isochronous resources available on a network.

The Enabler provides an effective word clock control mechanism by encapsulating the behaviour and functionality of the word clock implementation of a device within a single class. In addition to defining and implementing the standard word clock capabilities such as: *changing sample rates*, *establishing*, *breaking* and *retrieving word clock synchronization settings*, the word clock class defines and implements additional features that:

- determines the master/slave capability of a device, as well as providing
- the capability to monitor the word clock signals of master or slave devices.

Chapter 9

9. Evaluation

As mentioned in section 7.4 various applications can be developed to interact with an Enabler, with connection management related applications being the most common. A patch bay application was developed, as part this research, to test the functionality and efficiency of the redesigned Enabler described in the previous two chapters. This patch bay application is different from the patch bay application developed earlier for Linux, and implements a number of features that allow audio and MIDI plug connections, and also word clock synchronization to be performed. In addition to this, it gives a visual representation of the bandwidth usage on a network and hence permits isochronous resources to be managed. The efficiency of the Enabler is also established using a number of test procedures. These test procedures specify a number of operations to be performed by the Enabler for a number of network topologies. Timing measurements of the Enabler in enumerating a network and establishing/restoring multiple plug connections as well as disconnections are also given, and the results compared with that of the Linux and Windows Basic Enablers.

This chapter describes the features of the patch bay application developed to interact with the new Enabler, and gives qualitative and quantitative analysis of the Enabler's performance in comparison with the Windows and Linux Basic Enablers.

9.1 Features of the Patch Bay Application

An owner's manual for the Patchbay application that describes its installation, operation and features has been written, and is described by the document "Linux mLAN Patchbay Owner's Manual, Version 0.6" [Network Audio Solutions, 2005]. Refer to this document for detailed information regarding the patch bay application (see appendix B). The features described here highlight the connection management operations and isochronous resource management techniques of the Enabler.

9.1.1 Main Application

After starting the mLAN patch bay application, the dialog shown below appears.

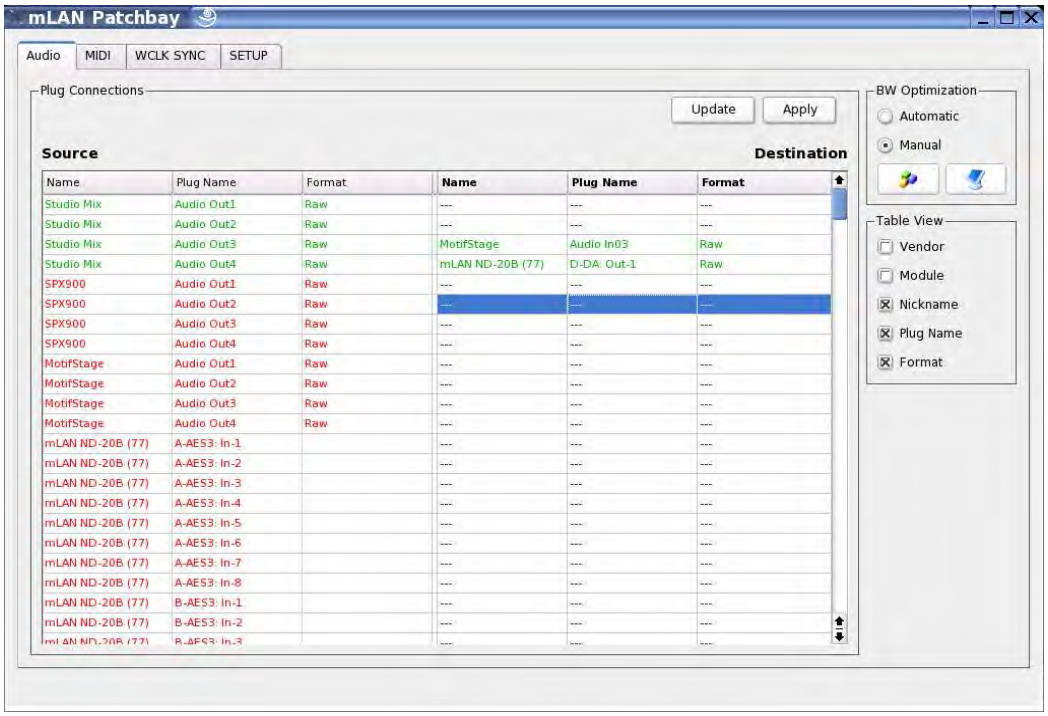


Figure 9-1: Start-up window of the Linux mLAN patch bay application

This dialog implements a number of tabbed pages as indicated by the tabs at the top right of the figure. From these tabs, a user can display connection information of the audio and MIDI plugs, and also word clock synchronization of the devices on the network. These displays allow plugs to be connected or disconnected, and also allow word clock synchronization between devices to be configured. The tab labelled "SET UP" reveals further information regarding the 1394 buses and mLAN devices

attached to a network. A description of the various tabbed pages is given in the subsequent sections.

9.1.1.1 The “Audio” and “MIDI” Tabbed Page

The “Audio” and “MIDI” tabbed pages each have a user interface that is identical to that shown in Figure 9-1. The command buttons labelled “Update” and “Apply” respectively update the display to reflect the corresponding information of the mLAN devices on the network, and apply any changes made on the display to the relevant mLAN devices on the network.

The data list constitutes a major part of the user interface and is divided into a “Source” and “Destination” section. Each of these sections is defined by a set of columns given by:

Figure 9-2: Data columns defined by the Audio and MIDI tabs

These columns provide textual descriptions of the source or destination plugs of the mLAN devices on the network. The plugs listed here can either be the end-point/true plugs of an mLAN device or the plugs modelled by its transport layer. The choice of plugs displayed depends on whether an mLAN device exposes its host implementation to the new Enabler, via a node application interface.

The content of the columns defined by the data list is given in the table below.

Column	Description
Vendor	Displays the manufacturer of an mLAN device.
Module	Displays the module name of an mLAN device.
Name	Displays the nickname of an mLAN device.
Plug Name	Displays the plug name of a plug of an mLAN device.
Format	Displays the plug format of a plug of an mLAN device.

Table 9-1: Description of the data columns defined by the Audio and MIDI tabs

The visibility of these columns on the display is controlled by the check boxes contained within the group box labelled “Table View”. Checking a box causes the corresponding column to be visible, unchecking it causes the column to be hidden. The nature of the data list and how it allows for audio/MIDI plug connections and disconnections is described further on in this section.

The tool buttons located within the group box labelled “BW Optimization” allows isochronous resource management to be performed on an mLAN Device. If the “Automatic” radio button is selected, the network bandwidth is optimized each time the “Apply” command button is click. Selecting the “Manual” optimization option allows a user to, at anytime, optimize network bandwidth by clearing stream control registers of IEEE 1394 bridge portals that are not in use, but have been configured for isochronous stream forwarding. Manual optimization also allows a user to, at anytime, perform isochronous sequence optimization on an mLAN device.

9.1.1.2 The “WCLK SYNC” Tabbed Page

The user interface of the “WCLK SYNC” tab is given in Figure 9-3 below. This page also implements the “Update” and “Apply” command buttons that update the display and apply changes made on the display to the relevant mLAN devices on the network.

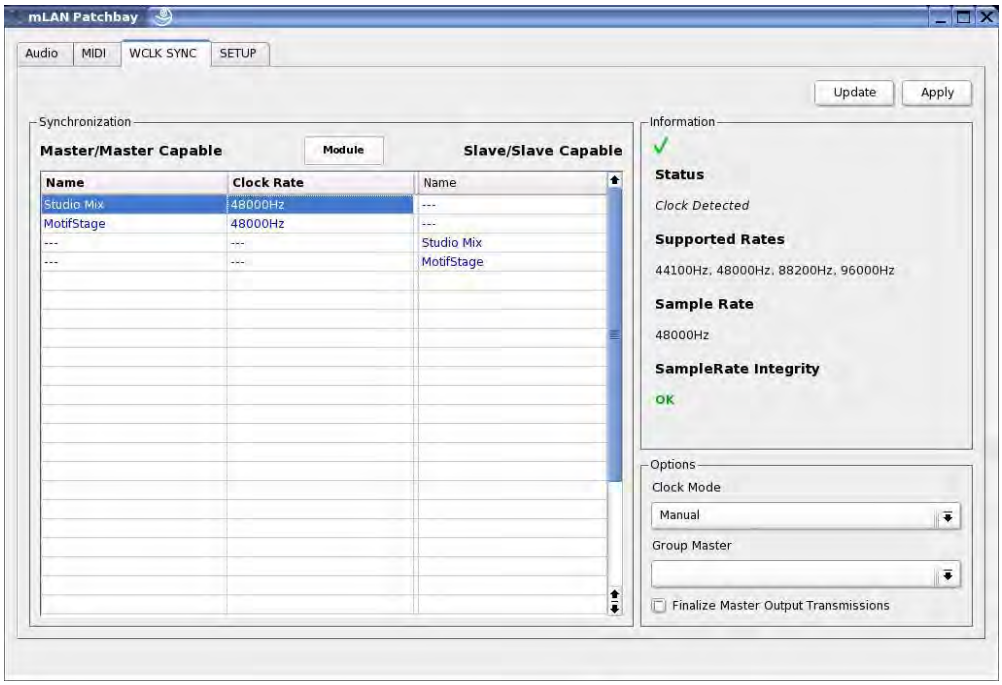


Figure 9-3: User interface of the "WCLK SYNC" tabbed page

The data list of this page has a “Master/Master Capable” section and a “Slave/Slave Capable” section. The “Master/Master Capable” section is defined by three columns, and displays the word clock masters and master capable mLAN devices on the network. Similarly the “Slave/ Slave Capable” section displays the textual name of word clock slaves and slave capable mLAN devices on the network. The three columns defined by the “Master/Master Capable” section are illustrated in Figure 9-4 below:

Figure 9-4: Data columns defined by the “Master/Master Capable” section

Note that as in the case of the data list of the “Audio” and “MIDI” tabbed pages, the word clock synchronization information displayed here can either reflect the synchronization information of the host implementation of a device, or the synchronization information of its transport layer. The actual synchronization information displayed depends on whether the device exposes its host implementation to the new Enabler, via a node application interface.

The description of the columns defined by the data list is given in the table below.

Column	Description
Module	Displays the module name of an mLAN device.
Name	Displays the nickname of an mLAN device.
Clock Rate	Displays the current clock rate of a master capable mLAN device.

Table 9-2: Description of the data columns defined by the “Master/Master Capable” section

The visibility of the module column is controlled by the tool button labelled “Module”. If this tool button is depressed, the module column is shown, otherwise it is hidden.

The “Information” display, shown at the right side of Figure 9-3, reveals information about the capabilities and state of the word clock implementation of the currently selected mLAN device. As shown in the diagram, it gives information on whether the

clock signal is detected, the sample rates supported by the device, the current sampling rate value and the integrity of the current sample rate.

The “Options” group box specifies various configuration options for word clock synchronization. The “Clock Mode” and “Group Master” combo-boxes respectively indicate what synchronization mode – manual or automatic, is in use by the mLAN devices on a network, and what the master word clock device should be. The “Finalize Master Output Transmissions” check box, when checked, indicates that isochronous resources be allocated to all the transmission plugs of any mLAN device configured to be a word clock master. This added capability also prevents any form of bandwidth optimization to be performed on the device.

The user interface of the “SET UP” tab is not reviewed here. Refer to the patch bay’s user manual [Network Audio Solutions, 2005] for further information on this. The next section describes the operation of the patch bay application, and how a user can perform audio and MIDI plug connections, word clock synchronization and isochronous resource management.

9.1.2 Patch Bay Operations

The data list of the “Audio”, “MIDI” and “WCLK SYNC” tab shows various plugs in the case of “Audio” or “MIDI”, or devices in the case of “WCLK SYNC”. The entries shown on these pages are colour coded, as shown in Figure 9-1 and Figure 9-3. These colour codes give an indication of the state of the plugs or devices, and also gives a visual representation of isochronous resource usage by devices as described below.

For the “Audio” and “MIDI” tabs, a red entry indicates that the plug associated with the mLAN device is in an *inactive* state. If it is a source plug, this could either mean that the mLAN device as a whole is not transmitting audio/MIDI data or the selected plug has not been allocated isochronous resources for transmissions. If it is a destination plug, this could either mean that the device as a whole is not receiving audio/MIDI data or the selected plug has not been configured for reception. A green entry indicates that the plug is in an *active* state. If it is a source plug, this indicates that isochronous resources have been allocated to the plug for transmissions and that

it is indeed transmitting audio/MIDI data over the network. If it is a destination plug, this means that the plug has been configured to receive audio or MIDI data. A green entry with no corresponding source or destination plug indicates a *dangling* plug connection. A black entry indicates that the connection state of the plug has been modified by a user but not updated on the network.

For the "WCLK SYNC" data list, a blue entry indicates the current synchronization state of an mLAN device on the network. The colour of an entry changes if a synchronization request is made on the data list, and this is an indication to the user to either apply the change or to update the display.

The following subsections describe the operations of the application.

9.1.2.1 Establishing Plug Connections

To establish an audio or MIDI plug connection, a user right-clicks on a source plug and navigates through a list of resulting popup menus to select the target plug to be connected. This is illustrated in the figure below.

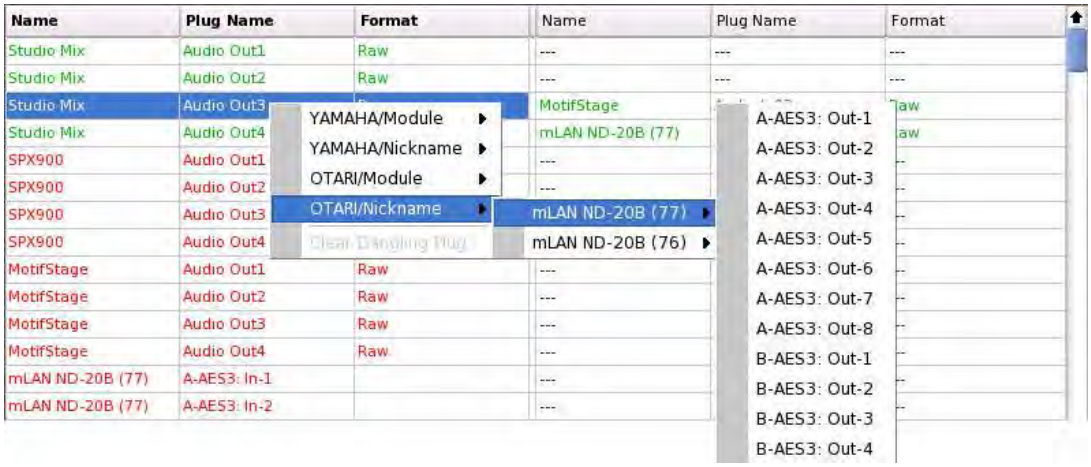


Figure 9-5: Establishing an audio or MIDI plug connection

After the target plug is selected, the plug is placed adjacent to the intended source plug and the colour of the entry changed to black. To effect the connection, the "Apply" command button has to be clicked, and if successful, changes the colour of the entry to green.

9.1.2.2 Breaking Plug Connections

To break a connection to an audio or MIDI plug, a user right-clicks on the desired destination plug to be disconnected and selects the “Disconnect” option of the resulting popup menu. This is illustrated in the figure below.

Figure 9-6: Breaking an audio or MIDI plug connection

After the “Disconnect” option is selected, the destination plug appears as unconnected and the “Apply” button must be clicked to effect the disconnection.

9.1.2.3 Clearing Dangling Plug Connections

To clear a dangling audio or MIDI plug connection, a user right-clicks on the dangling source or destination plug and then selects the “Clear Dangling Plug” option of the resulting popup menu, as illustrated in Figure 9-7 below.

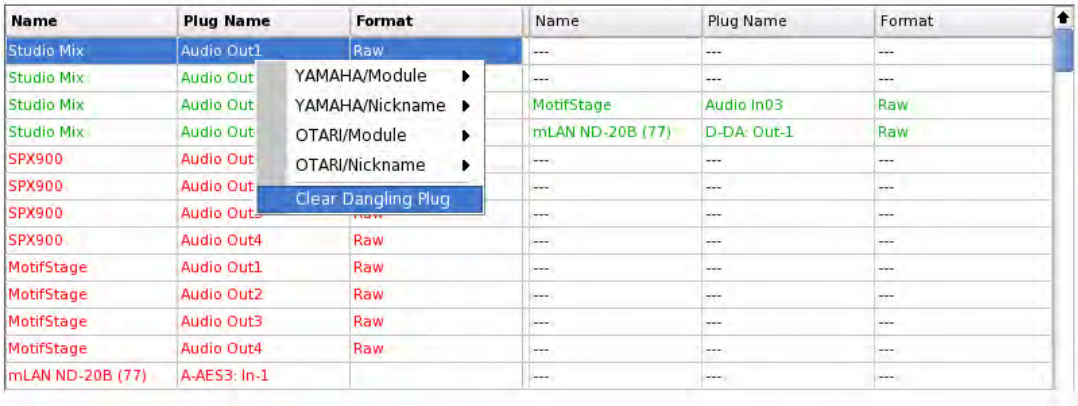


Figure 9-7: Clearing dangling plug connections

After selecting the “Clear Dangling Plug” option, the colour of the entry is changed to black, hence indicating a modification made to the display. If this modification is applied, the entry changes to red, which further implies that the plug is not configured for transmissions or receptions.

9.1.2.4 Optimizing Bandwidth Usage

Section 9.1.1.1 revealed two ways in which a user can optimize the bandwidth usage of a device; by performing either automatic or manual optimization. If the automatic option is selected, bandwidth usage on the entire network is optimized each time the “Apply” command button is clicked. For a user to perform manual optimization, the following procedure must be followed:

1. On the data list of either the “Audio” or “MIDI” tab, select the mLAN device on which optimization is to be performed.
2. Ensure that the “Manual” option located within the “BW Optimization” section of the page is selected.
3. Click the appropriate optimization tool button.

These steps are illustrated below:

Step 1: Select the mLAN device

Name	Plug Name	Format	Name	Plug Name	Format
Studio Mix	Audio Out1	Raw	---	---	---
Studio Mix	Audio Out2	Raw	---	---	---
Studio Mix	Audio Out3	Raw	MotifStage	Audio In03	Raw
Studio Mix	Audio Out4	Raw	mLAN ND-20B (77)	D-DA: Out-1	Raw

Figure 9-8: Step 1 of manual bandwidth optimization

Step 2: Select the “Manual” option within the “BW Optimization” section of the page.

Figure 9-9: Step 2 of manual bandwidth optimization, selecting the “manual” option

Step 3: Click the appropriate optimization tool button, see section 9.1.1.1 for the role of each tool button. This tool button to be clicked is indicated below.

Figure 9-10: Step 3 of manual bandwidth optimization

9.1.2.5 Master and Slave Settings

To configure an mLAN device to receive word clock synchronization from another device, a user right-clicks on the device to be configured, listed under the "Slave/Slave Capable" section of the data list of the "WCLK SYNC" tab. The corresponding master or master capable device is selected by transversing through the resulting popup menus. This operation is illustrated using the figure below:

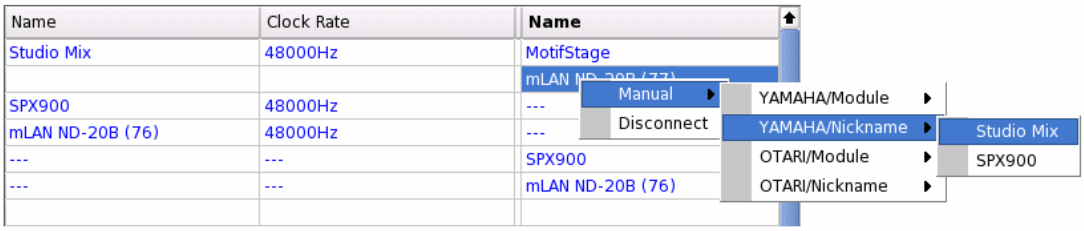


Figure 9-11: Performing master and slave synchronization settings

After the corresponding master or master capable device is selected, the device to be configured as slave is displayed adjacent to the corresponding master. The configuration has to be applied for the synchronization setting to take effect.

9.1.2.6 Specifying a Clock Rate

The application only allows the clock rate of the devices listed under the "Master/Master Capable" section of the data list to be changed. In order to do this, a user right-clicks on a master or master capable device and selects the desired sampling rate value to be set from the resulting popup menu, as illustrated below.

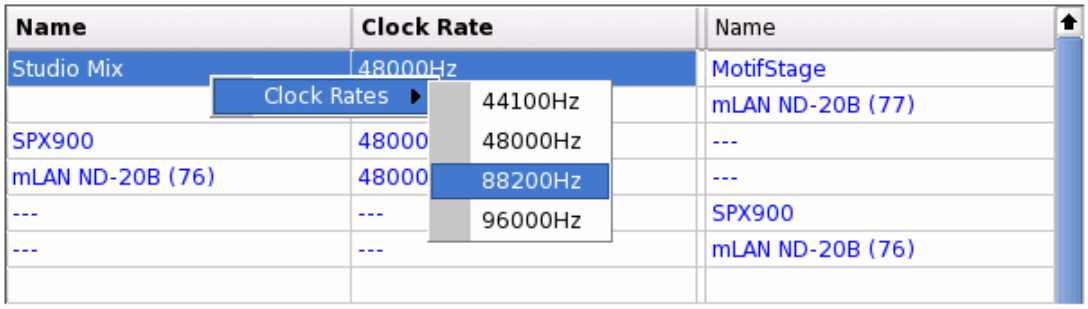


Figure 9-12: Specifying a clock rate value

If the change in sampling rate breaks the sample rate integrity check, (see section 7.1.4) a suitable error message is displayed in the "Information" section of the page in order to prompt the user before applying the setting. An example error message from failing the sample rate integrity check is shown below.

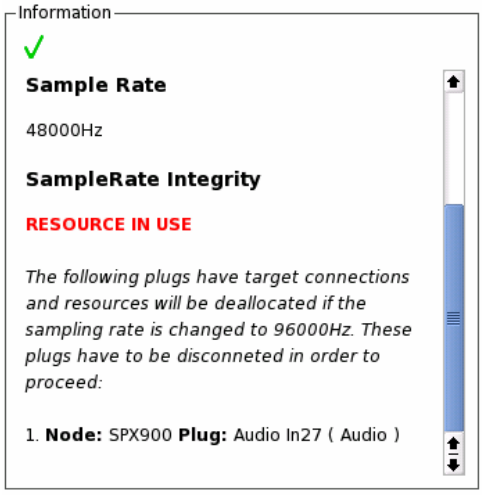


Figure 9-13: Error message from failing the sample rate integrity check

In this case, the selected sample rate "96000Hz" *would* result in a reduction of resources in use by the device, which *would* affect the connected plug "Audio In 27" of "SPX900". In other words, if the user still wants to set the desired sampling rate, the connected plug "Audio In 27" of "SPX900" must be disconnected.

The patch bay application implements other features that are not reviewed here. Refer to the operating manual [Network Audio Solutions, 2005] for further information. The features reviewed here, in comparison to the patch bays of the Windows and Linux

Basic Enablers, immediately reveal some of the enhanced features of the redesigned Enabler. These include the way in which isochronous resources are graphically represented on the user interface, the option of clearing dangling plugs, and also the ease of setting up master/slave relationships as well as configuring sampling rates. The next section discusses some of the test procedures conducted using the redesigned Enabler and gives some performance results.

9.2 Implementation, Testing & Evaluation

The Enabler was developed using GCC, the GNU Compiler Collection, under SUSE Linux Professional 9.1 [Novell Inc., 2005], with kernel version 2.6 and KDE version 3.2. The kernel's original IEEE 1394 drivers were used, but the libraw library was upgraded to version 0.10.1. This library is available on the "IEEE 1394 for Linux" website [Linux1394, 2005].

The implementation and testing of the Enabler, to a large extent, was performed alongside development. Standard testing techniques described by "Testing Computer Software" [Kaner, Falk and Nguyen, 1999] were used in verifying various components of the Enabler. Unit testing followed by integrated testing was used extensively during the early developmental cycles of the Enabler, where a number of test applications were developed to test each method implemented for each Enabler class.

After completing the initial complete version of the Enabler, *alpha* testing was performed. This involved testing the system as a whole, where the Enabler was viewed as a white box. *Beta* testing followed, where the Enabler was treated as a black box. The testing parties involved in this process included Yamaha Corporation and 3rd party developers who implemented the client/server model above the Enabler.
A number of stress and performance tests were also conducted on the Enabler. The configurations of these tests and outcomes are discussed below.

9.2.1 Stress Testing

The stress testing performed on the Enabler mainly looked at its stability with regards to:

- Enumerating a large bridged network
- Handling frequent network configuration changes
- Applying a large number of plug connections and disconnections
- Applying a large number of word clock synchronization settings

All the IEEE 1394 bridges and Transporter devices within the Rhodes University Audio Engineering Lab were used in performing this test. These included:

- 2 NEC MX/Bridge-A devices
- 5 Yamaha MAP4 Evaluation boards
- 2 Otari ND-20B units

These devices were arranged in a number of configurations that best suited the test to be performed. Table 9-3 summarizes the outcome of the stress tests performed on the Enabler. Tests 1 and 2 that observed the Enabler's stability with regards to initial network enumeration and network configuration changes were performed using the configuration shown in Figure 9-14:

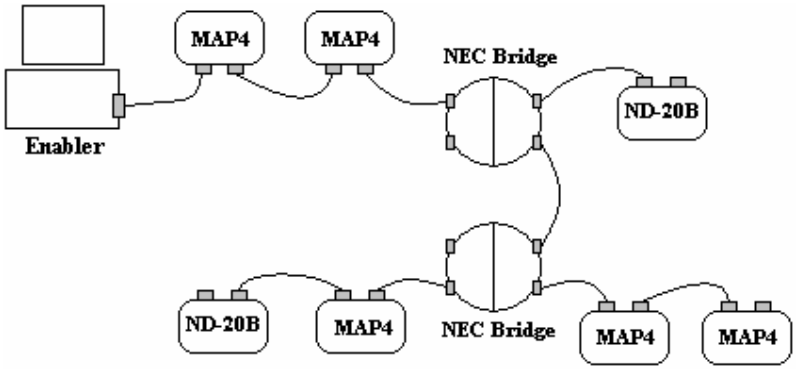


Figure 9-14: Network topology used in stress testing enumeration

Tests 3 and 4, which observed the Enabler's stability with regards to multiple connections and disconnections of plugs, including word clock synchronization, made use of the network configurations shown in Figure 9-15. The two ND-20B devices were used in each configuration in Figure 9-15 to ensure that bandwidth usage on the network was kept at a minimum. The Enabler was attached to each configuration before performing the test.

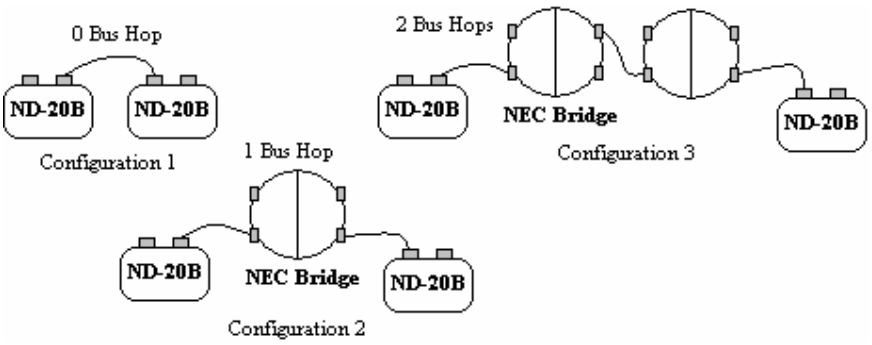


Figure 9-15: Network topology used in stress testing plug connections

ID	Stress Test	Test application	Outcome
1	Multiple enumerations were performed by the Enabler.	A test application was used in performing this test. The network was enumerated and reenumerated 20 times within a *for loop* implementation.	The Enabler passed the enumeration test of all 20 enumerations.
2	A series of configuration changes were performed. Devices were power cycled, removed and added to the network. Some of these operations caused bus resets and also bus configuration changes.	The patch bay application developed to interact with the Enabler was used in viewing the state of the network after a configuration change was performed.	The Enabler passed the test for each configuration change performed.
3	Applying a large number of plug connections and disconnections.	A test application was used in performing this test. In this particular test, the two ND-20B devices were used in three network configurations; first within the same bus, across one bus hop and then across two bus hops as shown in Figure 9-15. A set of 64 plug connections and disconnections were made for each network configuration.	The Enabler passed each set of plug connections and disconnections for each network configuration.
4	Applying a large number of	A test application was used in	The Enabler passed each

	word clock synchronization settings.	performing this test. In this particular test, the two ND-20B devices were used in three network configurations; first within the same bus, across one bus hop and then across two bus hops as shown in Figure 9-15. A set of 64 word clock connections and disconnections were made for each network configuration.	set of word clock connections and disconnections performed for each network configuration.

Table 9-3: Results of stress performance testing

These tests indicate that the Enabler will withstand a range of typical user interactions within limited size configurations. Stress testing does still need to be performed in larger configurations.

9.2.2 Performance Testing

The performance tests done on the Enabler mainly looked at the time taken by the Enabler to enumerate a number of network topologies and also to perform plug connections and disconnections in a single and multi-bus environment.

9.2.2.1 Measuring Speed of Network Enumeration

For network enumeration, measurements were taken for a number of configurations of a single bus network as well as a bridged network. The configurations that were used are illustrated by Figure 9-16, with the timing measurements given in Table 9-4. The timing measurements taken for each topology were performed under the same networking conditions as with the previous Enablers, in which the MAP4 Transporters were configured such that the Enabler would enumerate the maximum number of input and output plugs.

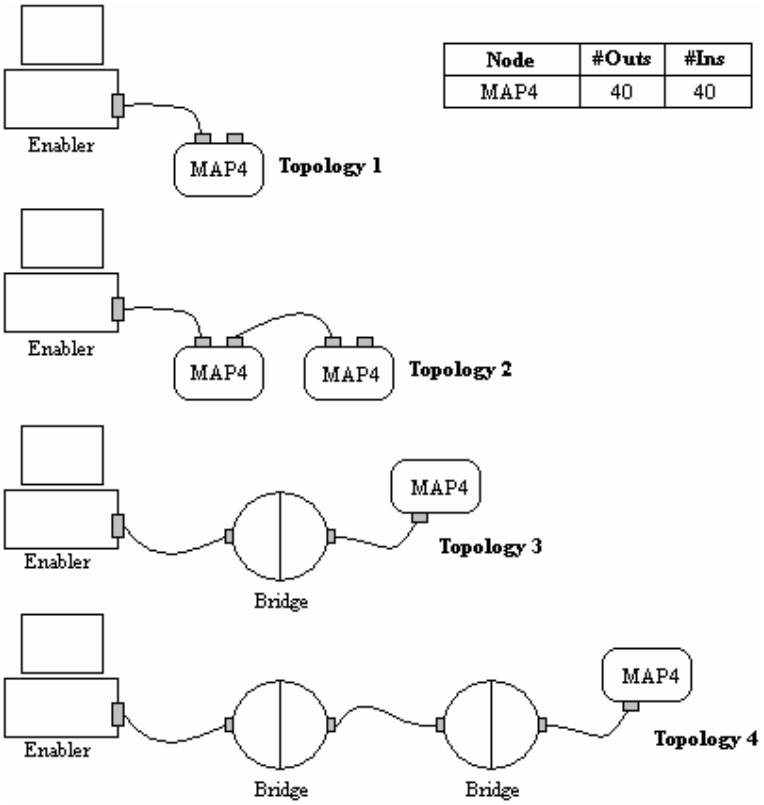


Figure 9-16: Topologies used in measuring network enumeration

	Timing Measurements [seconds]			
Topology	**1st**	**2nd**	**3rd**	**Average**
1	0.57	0.57	0.59	*0.58*
2	0.91	0.90	0.90	*0.90*
3	1.20	1.19	1.19	*1.19*
4	1.53	1.54	1.54	*1.54*

Table 9-4: Timing measurements for a number of network topologies

The average time taken by the Enabler to enumerate a single bus network described by “Topology 1” and “Topology 2” is less than the time taken by the Basic Enabler to enumerate the same topologies. From Table 5-3, the time measurements taken by the Basic Enabler are 0.71 seconds and 1.03 seconds respectively. Also, for the bridged topologies, “Topology 3” and “Topology 4” respectively, the average time measurements required for enumeration are less than that of the bridged Linux Basic

Enabler. From Table 6-3, the time measurements taken by the bridged Linux Basic Enabler are 5.81 seconds and 5.93 seconds respectively.

The average time taken by the Enabler to enumerate a Transporter node with 40 inputs and outputs is given by the difference in the average time measurements of "Topology 2" and "Topology 1". This value is calculated to be 0.32 seconds, and is identical to that obtained for the Linux Basic Enabler.

Recall from section 8.1.1 that the redesigned Enabler actively retrieves the network management messages required for enumeration from alpha portals. This implies that for every IEEE 1394 bridge added to the network, extra enumeration time would be required by the Enabler to read the network management messages from the alpha portal on the new bus created by adding the 1394 bridge. This extra time is calculated by the difference in the average time measurements of "Topology 4" and "Topology 3", which is calculated to be 0.35 seconds. In order words, the time taken by the Enabler to read information from an alpha portal on a 1394 bus is 0.35 seconds. The constant overhead added by the Enabler to network enumeration time is calculated to be 0.17 seconds. This is calculated using the formula below:

$$E_k = E_{tt} - (N_b \times B_t + N_T \times T_t) \quad (11)$$

where the variables are defined as follows:

E_k	The constant overhead added by the Enabler
E_{tt}	The total time required to enumerate a given topology
N_b	The number of buses on the network
B_t	The time required to read enumeration information from an alpha portal (= 0.35 secs)
N_T	The number of Transporters on the network (configured for 40 inputs and 40 outputs)
T_t	The time required to enumerate a single Transporter (= 0.32 secs)

Using this equation and the timing results of "Topology 3", the constant overhead (E_k) is calculated to be:

$$E_k = 1.19 - (2 \times 0.35 + 1 \times 0.32)$$
$$= 0.17 \text{ secs}$$

From timing measurements obtained for the bridged Linux Basic Enabler, section 6.3.1.4, E_k was found to be 5.37 seconds and B_t found to be 0.12 seconds. Note that these readings assume stable processing conditions of the Basic Enabler i.e. without inefficiencies in handling race conditions. The E_k value for the redesigned Enabler is far less than what the Basic Enabler provides, and is calculated to be 31.6 times faster. However B_t for the redesigned Enabler is greater than B_t for the Basic Enabler. This is expected because the redesigned Enabler actively reads network management messages, whereas the Basic Enabler accesses network management messages (which are broadcast on the network) by invoking a net reset. Since E_k and B_t can be assumed to be constants, it implies that for stable conditions, the Basic Enabler may be faster than the redesigned Enabler at enumerating networks with a large number of IEEE 1394 bridges, but for smaller networks the redesigned Enabler shows better performance. Figure 9-17 shows a graph that indicates the cut-off number of IEEE 1394 bridges for which the redesigned Enabler shows better performance over the Basic Enabler implementation.

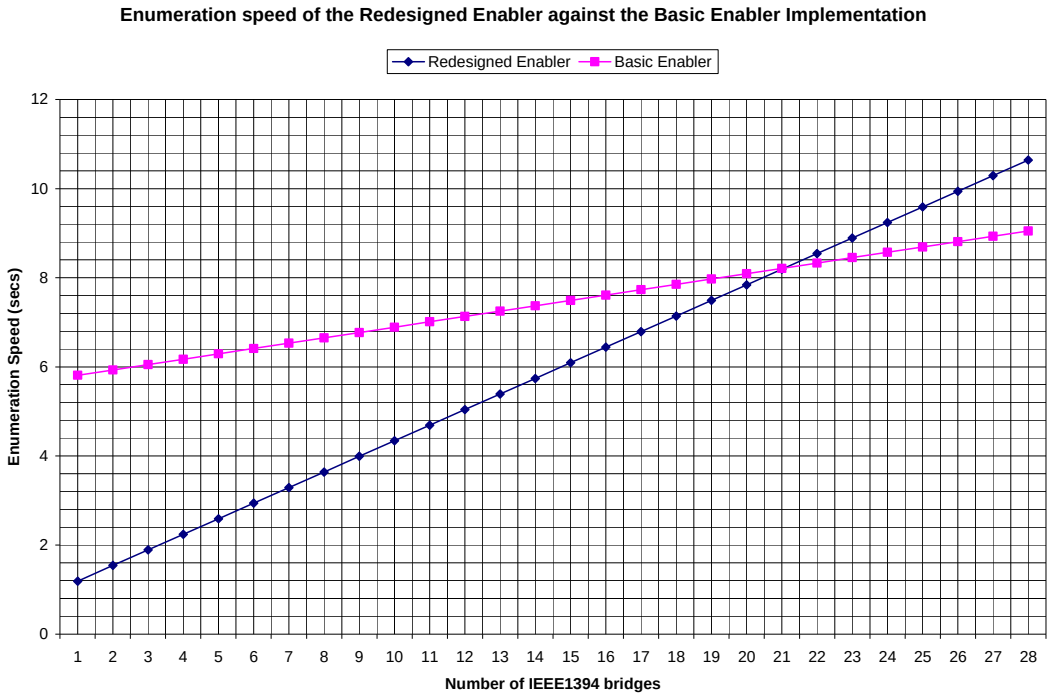


Figure 9-17: Theoretical evaluation of enumeration speed

These calculations were performed with one MAP4 Transporter configured for 40 input plugs and 40 output plugs. The enumeration times for both Enablers were calculated using the formula:

$$E_{tt} = E_k + (N_b \times B_t + N_T \times T_t) \quad (12)$$

Where N_b represents the number of IEEE 1394 bridges with respect to the Basic Enabler implementation, and represents the number of 1394 buses with respect to the redesigned Enabler. From the plot, the number of bridges below which the redesigned Enabler shows better speed performance is 21.

9.2.2.2 Measuring Speed of Plug Operations

Timing measurements were also conducted to determine the performance of plug operations in a single and multi-bus environment. These plug operations include connections, disconnections and the retrieval of a list of connections to a plug. The topologies used in performing these measurements are illustrated in the figure below.

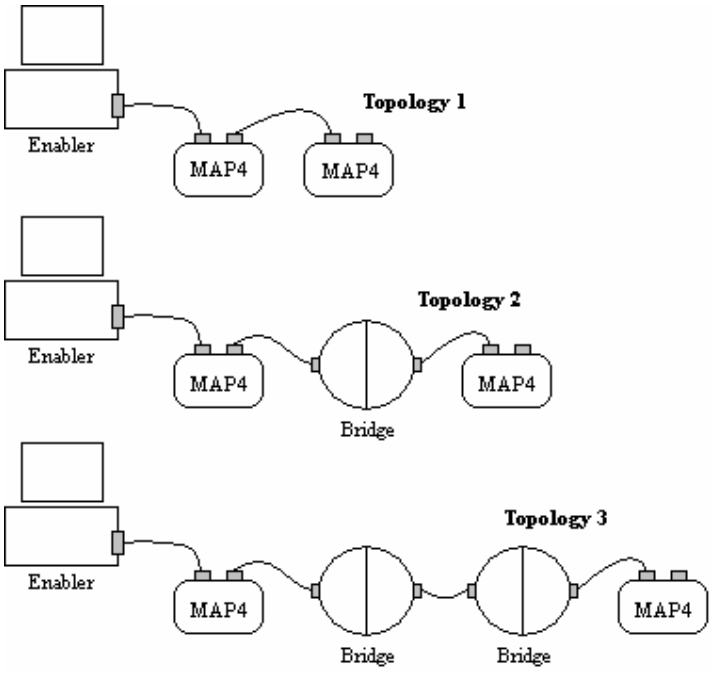


Figure 9-18: Network topologies used in measuring the speed of various plug operations

The results of the measurements are given in the tables below. Note the millisecond measure of time used with “Get Connections”. As with the Basic Enablers, the source

plugs used in timing the “Get Connections” routine were set up to not have any target connections, with three sets of thirty-two operations performed for timing the Connections, Disconnections and GetConnections.

	Timing Measurements using “Topology 1”		
	Connections [s]	**Disconnections [s]**	**Get Connections[ms]**
1st Thirty-two Operations	1.57	0.85	1.51
2nd Thirty-two Operations	1.57	0.85	1.65
3rd Thirty-two Operations	1.57	0.85	1.74
Average of 96 Operations	*0.05*	*0.03*	*0.05*

Table 9-5: Timing measurements for plug operations performed using "Topology 1"

	Timing Measurements using “Topology 2”		
	Connections [s]	**Disconnections [s]**	**Get Connections[ms]**
1st Thirty-two Operations	1.74	1.23	66.47
2nd Thirty-two Operations	1.64	1.23	30.02
3rd Thirty-two Operations	1.67	1.21	18.47
Average of 96 Operations	*0.05*	*0.04*	*1.20*

Table 9-6: Timing measurements for plug operations performed using "Topology 2"

	Timing Measurements using “Topology 3”		
	Connections [s]	**Disconnections [s]**	**Get Connections[ms]**
1st Thirty-two Operations	1.91	1.51	235.72
2nd Thirty-two Operations	1.93	1.51	293.95
3rd Thirty-two Operations	1.97	1.51	281.29
Average of 96 Operations	*0.06*	*0.05*	*8.45*

Table 9-7: Timing measurements for plug operations performed using "Topology 3"

These results, compared with those obtained for the Linux Basic Enabler and the bridged Linux Basic Enabler, show that the redesigned Enabler has a better speed performance in handling plug connections, plug disconnections and retrieving connections to a plug.

From the results of the single bus topology shown in Table 9-5, the average time for a plug connection is 0.05 seconds, 0.03 seconds for a plug disconnection, and 0.05 milliseconds for retrieving plug connections. These readings are respectively 1.6 times, 1.67 times and 400 times faster than the corresponding readings – 0.08 seconds, 0.05 seconds and 0.02 seconds - obtained for the Linux Basic Enabler (see Table 5-4). The reason for this speed-up can be explained from the design architecture and implementation adopted for the new Enabler.

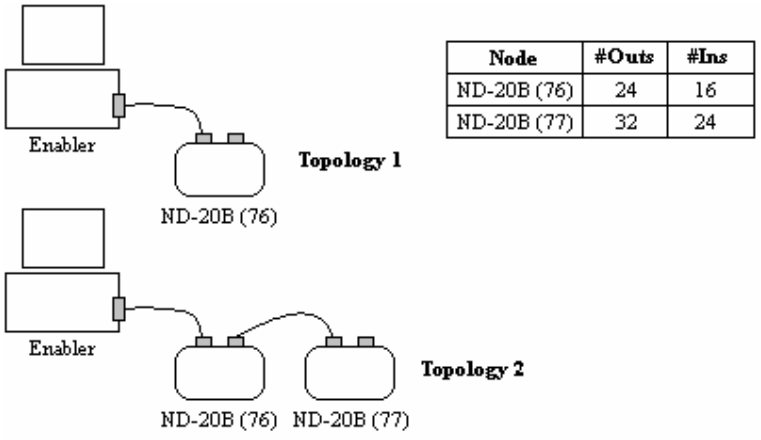
For the 1-bridged network, the timing results show that the redesigned Enabler is 3.2 times faster at performing across-bus plug connections, 6.0 times faster with across-bus plug disconnections, and 58.0 times faster at retrieving plug connections, when compared with the corresponding values of the bridged Linux Basic Enabler (see Table 6-4). Similar findings can be determined for the 2-bridged network given by "Topology 3" – 4.2 times faster at performing across-bus plug connections, 2.6 times faster with across-bus plug disconnections, and 22.5 times faster at retrieving plug connections (see Table 6-5). The reduction in time in performing these across-bus plug operations is largely due to the Enabler itself housing objects of the 1394 interfaces, buses and nodes, including bridge portal nodes, from where it can access cached information, as opposed to performing direct 1394 bus transactions to read bridge portal information.

9.2.2.3 Evaluating the Node Application Implementation

Similar timing measurements for network enumeration and a number of plug operations were performed with Otari's ND-20B devices in order to determine the speed at which the Enabler handles the interaction between its node application component and its node controller component.

For network enumeration, the topologies described by Figure 9-19 were used in the measurements. The ND-20B devices used were configured for the worst case, where an isochronous sequence was allocated to each of the host application output plugs. In doing this, the Enabler would have to enumerate a number of node controller plugs in addition to the node application plugs for the ND-20B. Recall from section 7.1.3, that the node controller plugs enumerated by the Enabler correspond to isochronous

sequences/subsequences of the transport layer of a given Transporter node, while the node application plugs refer to the host application plugs of a Transporter node.



Node	#Outs	#Ins
ND-20B (76)	24	16
ND-20B (77)	32	24

Figure 9-19: Topologies used in measuring network enumeration of ND-20B devices

The ND-20B (76)[11] device has a maximum of 24 output host application plugs and 16 input host application plugs. Similarly, the ND-20B (77) device has a maximum of 32 output host application plugs and 24 input host application plugs. For the host application output plugs, 24 isochronous sequences for the ND-20B (76), and 32 isochronous sequences for the ND-20B (77) were allocated. The results of the enumeration timing measurements are given in the table below.

	Timing Measurements [seconds]			
Topology	**1st**	**2nd**	**3rd**	**Average**
1	0.52	0.52	0.53	*0.52*
2	0.68	0.69	0.67	*0.68*

Table 9-8: Timing measurements for a number of ND-20B network topologies

Comparing these results to “Topology 1” and “Topology 2” of Table 9-4, the speed at which the Enabler enumerates ND-20B devices (configured for the worst case) is faster than that for the MAP4 Transporter, also configured for the worst case. By taking the difference in the average times of “Topology 2” and “Topology 1”, the time taken by the Enabler to enumerate ND-20B (77) can be determined; this is

[11] The number in parenthesis, in this case 76, refers to a hardware ID value assigned to an ND-20B device

calculated to be 0.16 seconds. The time taken by the Enabler to enumerate a MAP4 Transporter was calculated from Table 9-4 to be 0.32 seconds. For the MAP4 Transporter 80 node controller plugs (40 inputs and 40 outputs) were enumerated by the Enabler, while 56 node controller plugs (32 outputs and 24 inputs) were enumerated for the ND-20B (77). In addition to this, 56 node application plugs were also enumerated for the ND-20B (77). From this evaluation, it can be seen that the node application component implementation provided by the Enabler to interact with the host application of Transporter nodes, does not significantly affect the speed at which the Enabler enumerates these devices.

In determining the speed of plug operations, timing measurements for plug connections, disconnections and retrieving connections to a plug were obtained for a single bus and a 1-bridged network. The worst case configuration of the device was used in timing each plug operation. For plug connections, dynamic sequence allocation was performed by the Enabler for each source node application plug used, and for retrieving plug connections, the source node application plugs were configured to each have an associated node controller plug. In accordance with the measurements of the "Get Connections" routine of the MAP4, given in Table 9-5 and Table 9-6, the source plugs used were set up not to have any target connections. The network topologies used in the timing are illustrated by Figure 9-20 below.

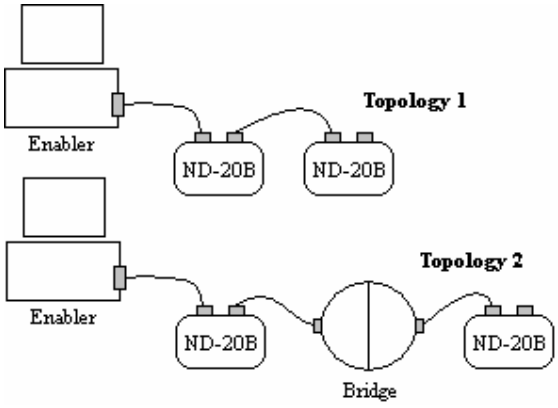


Figure 9-20: Network topologies used in measuring the speed of plug operations of ND-20Bs

The results of the timing measurements are given in the tables below. Note the millisecond unit of time used with "GetConnections".

	Timing Measurements using “Topology 1”		
	Connections [s]	**Disconnections [s]**	**Get Connections[ms]**
1st Thirty-two Operations	6.39	1.23	0.76
2nd Thirty-two Operations	6.39	1.23	0.77
3rd Thirty-two Operations	6.35	1.23	0.79
Average of 96 Operations	*0.20*	*0.04*	*0.02*

Table 9-9: Timing measurements for plug operations performed using "Topology 1"

	Timing Measurements using “Topology 2”		
	Connections [s]	**Disconnections [s]**	**Get Connections[ms]**
1st Thirty-two Operations	6.88	1.70	101.82
2nd Thirty-two Operations	6.97	1.69	101.32
3rd Thirty-two Operations	6.93	1.69	17.12
Average of 96 Operations	*0.22*	*0.05*	*2.29*

Table 9-10: Timing measurements for plug operations performed using "Topology 2"

From the timing results of “Topology 1”, Table 9-9, the average time taken by the Enabler to establish a connection from a node application plug that has not been allocated isochronous resources is found to be 0.20 seconds. This is 4 times slower than the average time taken to establish a connection from a node controller plug (Table 9-5), which by definition already has allocated isochronous resources. The reason for the decrease in speed is the need for the Enabler to perform dynamic sequence allocation by assigning a node controller plug, and hence an isochronous sequence, to the node application plug. In performing this assignment, extra bandwidth is also allocated from the isochronous resource manager node. Similar timing measurements were conducted to determine the average time taken by the Enabler to establish a connection from a node application plug that already has allocated isochronous resources. This was found to be 0.07 seconds. This is 0.02 seconds more than that than shown in Table 9-5, and also implies that the dynamic sequence allocation process requires about 0.13 seconds. Recall from 8.2.1 that dynamic sequence allocation only occurs for node application plugs that do not have an associated node controller plug, and if the node controller plug is acquired it is not released, unless otherwise released by a user. From these results, a user requiring

high-speed plug connections from node application plugs can pre-configure the source plugs required for connections. Pre-configuration of node application plugs has been discussed in section 8.2.2.

The average times for “Disconnections” and “GetConnections” for “Topology 1” are found to be 0.04 seconds and 0.02 milliseconds respectively, which also show a slight decrease in speed when compared to the corresponding time measurements taken for the MAP4 Transporters (Table 9-5). The slight decrease in speed is expected because of the interaction that occurs between the node controller component and node application component of the Enabler.

The timing results obtained for “Topology 2” show a slight increase in value when compared to the results obtained for “Topology 1”. The increase in value can be explained from the extra time taken by the second bridge to configure across bus forwarding, which is about 0.02 seconds in the case of plug connections, and 0.01 seconds in the case of plug disconnections. This shows that across-bus plug connections and disconnections of node application plugs of a device, does not significantly affect the speed performance of the redesigned Enabler.

9.3 Summary

A patch bay application has been developed to demonstrate the features of the new Enabler. The features demonstrated include:

- The true end-to-end connection management capability of the Enabler.
- The isochronous resource management techniques implemented by the Enabler such as: *dynamic sequence allocation*, *pre-allocating isochronous sequences* and *optimizing isochronous sequences*.
- Configuring word clock synchronization between devices, and being able to change the sample rate of word clock sources.

In addition to these, the stability and performance of the Enabler has been proven through a number of test procedures performed. The stability tests include:

- A continuous cycle of multiple network enumerations

- Causing repeated network configuration changes by power cycling devices, removing and inserting devices.
- Applying a large number of plug connections and disconnections, and also word clock synchronization settings.

From the results of these tests, it is observed that the Enabler will withstand a range of typical user interactions within limited size configurations.

The performance tests conducted on the Enabler include time measurement for plug connects, disconnects, and retrieving plug connections. From the analysis, the Enabler takes, in a single bus environment, *0.05s*, *0.03s* and *0.05 ms* to connect a plug, disconnect a plug and retrieve connections to a plug, respectively. When these tests were performed in a 2-bus environment, across buses, they showed minimal increase in time.

The node application implementation of the Enabler is also evaluated, and it is shown that there is negligible overhead added to the overall performance of the Enabler in realizing true end-to-end connection management.

Chapter 10

10. Conclusion

Networked audio is a relatively new field within the audio industry. This thesis has introduced the motivations for the appearance of this field of study. It has also traced the development of various audio networking technologies. The origin of audio networks lie in the complications and distortion that were present when using analogue cabling, where direct patching of analogue plugs was performed. The complications include, amongst other things, the difficulties that arose from managing plug connections resulting from using a mix of cables. These cables include: analogue audio, digital point-to-point, additional word clock cables for synchronization, and MIDI cables for control. However, one of the advantages of staying in the analogue domain is the conceptual simplicity it provides for a user in connecting plugs of devices – point-to-point connections. One of the main goals of this thesis is to retain this conceptual simplicity within digital audio networks.

Two main problems arise when using the point-to-point connection technique. These include:

1. Cable clutter, and
2. The problems associated with using the various types of analogue cables – signal interference and short cable length.

The former makes it difficult to manage plug connections between devices, and often, sticky labels are employed to uniquely identify the source plug of a device that is

connected to a particular input, or the input plug of a device that is connected from a source. The latter is a problem that is more apparent in large sound installations, and more often than not, expensive equipment is used in addition to the components of the sound installation to combat interference and cable length.

A number of networking schemes have been reviewed to deal with the problems of analogue interconnects, and reap the benefits of staying in the digital domain. Such networks include CobraNet™ [Peak Audio Inc., 2004a], EtherSound [Digigram, 2005], mLAN [Yamaha Corp., 2005b], A-Net™ [Aviom Inc., 2005], Axia [Axia Audio/TLS Corp., 2005a], AES47 [AES Inc., 2002] and AES50 [AES Inc., 2005], and they deliver, to a greater or lesser extent, the desired latency, multi-channel transport and the QoS required by various sound installation applications. The choice of one digital network over another depends on the demands of the application in which the audio network is to be deployed. However, as far as the connection management of these networks go, they employ what we will refer to as a two-step approach in fulfilling connections between host plugs of devices. This two-step approach is illustrated using Figure 10-1 below.

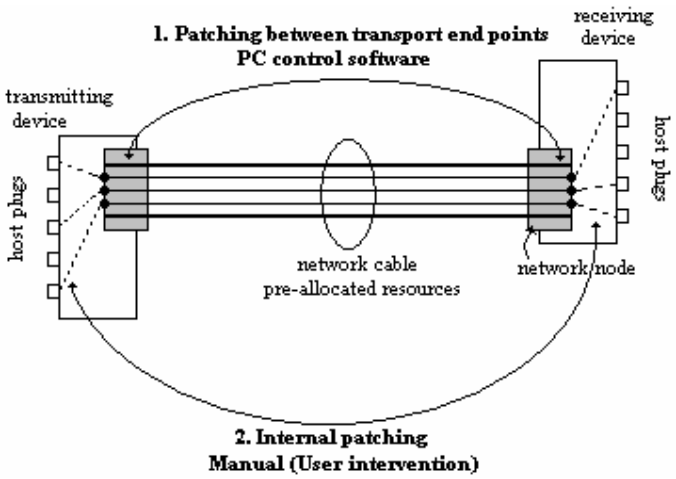


Figure 10-1: The two-step approach to connection management of current digital networks

First, transport-level routings are required to be established between the inputs and outputs of these networked-enabled devices, after which the host plugs of the transmitting and receiving devices have to be internally routed to either receive or transmit data from the inputs and outputs of their transport implementations. The

problem however, lies in the level of control offered to users of these networks in managing plug routings. Currently, digital audio networks make use of a PC-based application to establish transport-level routings between devices. The internal patching between the host plugs of a device and the 'plugs' of its transport implementation, has to be configured manually on the device. Hence, the two-step approach to connection management.

This two-step approach introduces a further complexity to a user in configuring the host plugs of devices. Furthermore, the transport-level 'plugs' or network audio channels implemented by a device's network transport, and presented to a user via a PC-based routing application, do not intuitively describe the 'user-recognizable' plugs of a device. This two-step approach makes connection management a less straight forward process. It is desirable that users who make use of audio networks in their applications, be able to make audio/MIDI plug connections between the host plugs on devices, and not between 'plugs' or audio channels exposed by a network's transport implementation.

The technique used in directly managing plug connections at the host-plug level of a device is referred to as the single-step approach. This reflects the conceptual simplicity described by the point-to-point cabling technique of the analogue domain. This single-step approach is described in Figure 10-2 below.

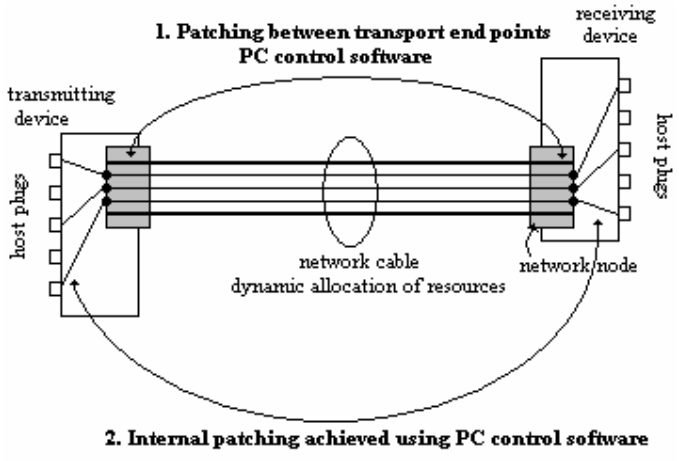


Figure 10-2: The single-step approach to connection management

In this setup, the internal patching of the host plugs of both the transmitting and receiving devices is not performed manually, but rather by the PC control software used to set up transport-level routings. A user of such a system would be presented with ‘user-recognizable’ plugs from where direct plug-to-plug patching can be managed. Such a capability does not only ease the task of connection management, but also provides a platform from where the isochronous resources required for device transmissions can be managed efficiently, e.g. bandwidth. As part of managing resources, bandwidth would be dynamically allocated for any host plug that is configured for transmissions. This is in contrast to the two-step approach where pre-allocating resources is necessary (see Figure 10-1). Realizing this single-step approach to connection management is the focus of this thesis.

In achieving true end-to-end connection management, the issue of speed also has to be taken into account. Providing host-plug to host-plug connection management can reduce the time taken to set up a sound installation. In addition to this, some applications, particularly live broadcast environments, require fast audio routing switching capabilities, to be able to switch between different audio feeds from different satellite studios in the shortest possible time. An audio network that enables true end-to-end connection management, as well as fulfilling the high speed criterion of these applications, is required.

Yamaha Corporation’s mLAN Digital Network Interface Technology [Yamaha Corp., 2005b] was chosen as the audio network to use in this investigation. True end-to-end connection management with mLAN can be realized using the Otari ND-20B mLAN-enabled devices [Otari Inc., 2005]. These devices implement a *node application* interface that exposes, amongst other things, the configuration information of the host plugs supported by the device. From an IEEE 1394 bus, an application can assign an isochronous sequence or audio channel to an output host-plug of an ND-20B. This channel contains the corresponding data to be transmitted on the network. In addition to this, mLAN was also chosen because it enables high speed connection management via its Enabler/Transporter architecture. This is made possible by the data transaction speeds (currently 400 Mb/s) defined by its transport medium, IEEE 1394. The move towards an open-standards-based implementation of the mLAN Transporter, the Open

Generic Transporter [AES SC, 2005], was another desirable factor in choosing mLAN over the other audio networking technologies.

In approaching the problem of host-plug to host-plug plug connectivity, the current mLAN connection management architectures were analyzed and implemented, and their performance evaluated. From these evaluations, it was revealed that the *mLAN Version 2* architecture[12], the second and current generation of mLAN, outperformed the *mLAN Version 1* architecture[13], the first generation of mLAN. It measured a speed increase of about 90.6% in enumerating a device, 96.7% in performing plug connections, and 96.8% in performing plug disconnections. However, the Basic Enabler specification, which describes the connection management application for *mLAN Version 2* devices, also known as Transporters, lacks the necessary components that permit an efficient implementation of true end-to-end connection management. It also defines an architecture that does not allow for an efficient IEEE 1394 bridge implementation.

A critical analysis performed on the Basic Enabler specification, raised two significant limitations that inhibited true end-to-end connection management, as well as IEEE 1394 bridging. These are:

1. The fact that the IEEE 1394 bridge implementation is made difficult and inefficient because the Basic Enabler specification is defined to allow Enabler-related objects to be created and managed by an application rather than to be created and hosted within the Enabler. The significance of this is that objects modelling IEEE 1394 bridges would always have to be created each time bridge manipulations are to be performed, which implies that bus transactions would always occur repeatedly on the bus.
2. The Basic Enabler specification lacks the capability to model the *node application* of a Transporter node, which is what is required to enable end-to-end connection management.

From these limitations, a number of design concepts were devised, upon which a new Enabler design and a number of component level innovations are based. The Enabler

[12] Also known as the Enabler/Transporter architecture
[13] The vendor-specific AV/C implementation of mLAN

now defines a *node application component* and *node controller component*. The node application component models the node application implementation of a Transporter node (using the Otari ND-20B devices as a reference), and hence provides access to the end-point plugs of a device. The node controller component of the Enabler models the node controller implementation (the transport layer) of *mLAN Version 2* devices. In implementing the node application and node controller components, a number of other features that relate to connection management were devised and implemented. These include providing:

- an effective isochronous resource management service, which includes dynamic allocation of resources, and
- an effective word clock synchronization implementation.

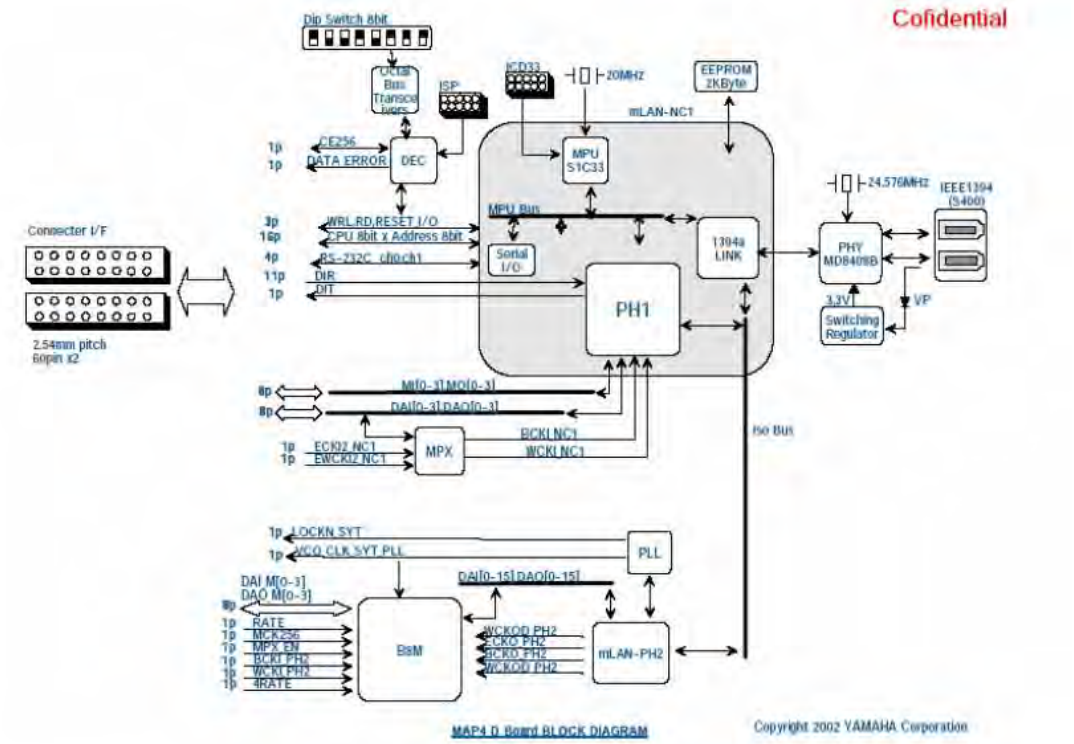
A patch bay application has been written to prove the capability of dynamic resource allocation and true connection management defined by the new design. In addition to this, the new design provides connection management between the host plugs of a device and the transport-layer 'plugs' modelled on behalf of devices. In other words, it supports backward compatibility with other *mLAN Version 2* devices.

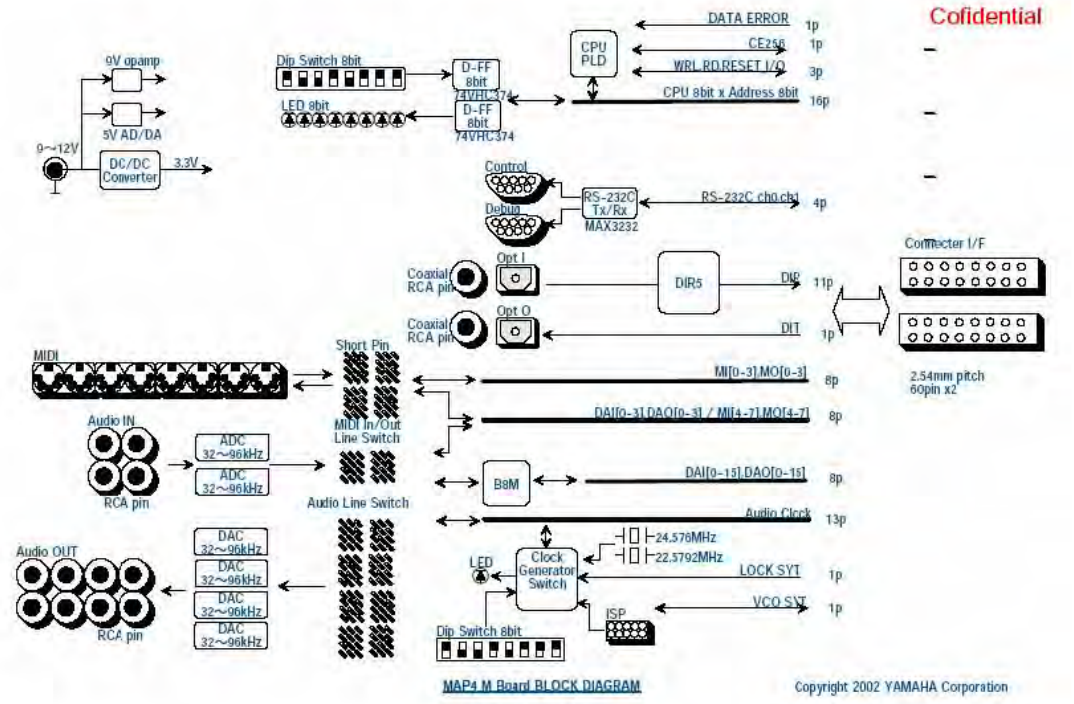
Performance and stress tests have confirmed the efficiency and stability of the new design, revealing a superior efficiency, stability and performance when compared with the current mLAN connection management implementations. In addition to this, there is negligible overhead added to the overall performance of the new design in realizing true end-to-end connectivity.

This research builds upon the *mLAN Version 2* architecture of Yamaha Corporation's mLAN networking technology, by providing a new connection management design that truly models *mLAN Version 2* devices, in so doing, demonstrating true connection management which also enables isochronous resource management. The design, implementation, and the ND-20B hardware, can serve as a template for future connection management applications of digital audio networks. An ideal would be to extend the ND-20B capability to all devices, to ensure that the user experience of patching host-plug to host-plug could be realized in all application domains.

Appendix

A. Layout of the MAP4 Evaluation Board





B. Linux mLAN Patchbay Owner's Manual

Release Notes

The user interface of the Linux mLAN Patchbay has been modified significantly to provide a more graphic representation of the use of isochronous bandwidth on an mLAN network, as well as providing various user options for managing these resources.

The following steps assumes you have performed the necessary steps required in setting up the Enabler, if not refer to the document *HowToSetUpEnabler.pdf* for the required setup procedure.

This application was developed and tested under SUSE Linux 9.1 and may work for other Linux distributions.

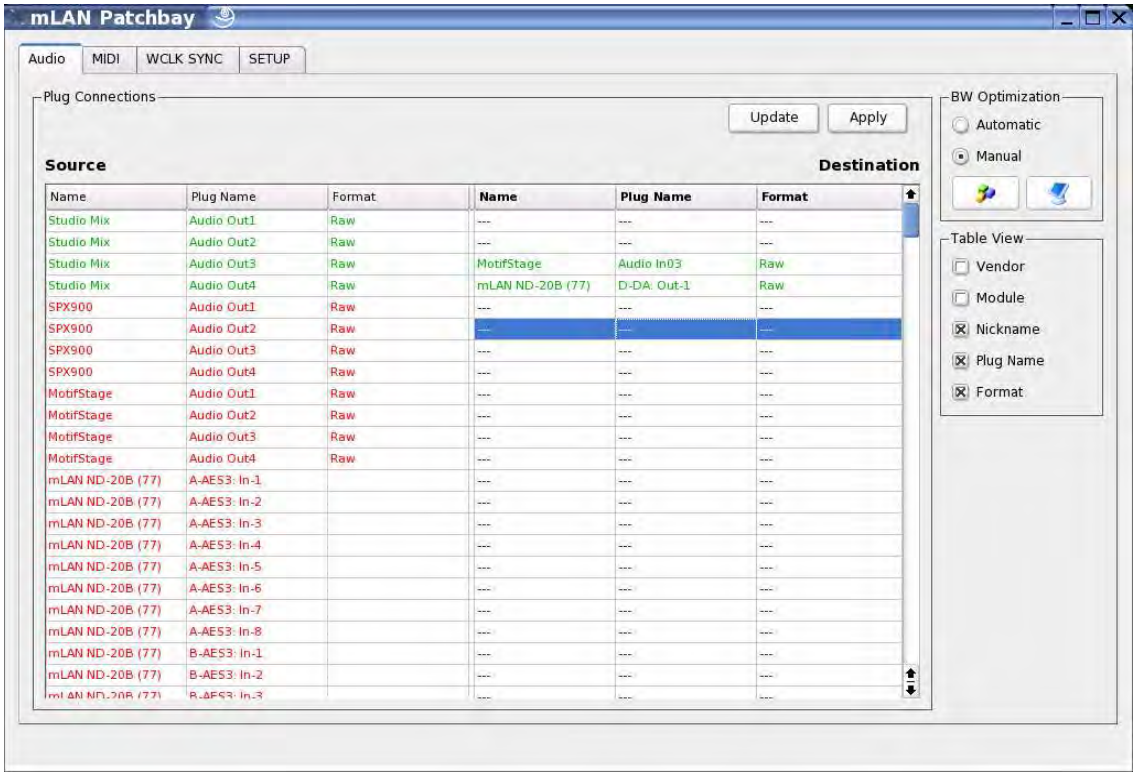
Starting the mLAN Patchbay

After setting up the mLAN Enabler and its various dependencies, follow the necessary steps to set up the mLAN Patchbay. Login as root or with administrative privileges before proceeding.

1. Change directory to the *mlanpatchbay-0.6* directory, i.e. your prompt should read something similar to "*[root@<your user name> mlanpatchbay-0.6] #".*
2. Type "*make"* to compile the package sources.
3. Type "*make install"* to install the program, any data files and documentation.
4. Type "*mlanpatchbay"* at the shell prompt to run the patchbay application.

Startup Window

After starting the mLAN Patchbay application, the window shown below appears.



Tabs

By clicking the tabs you can access the following pages:

Audio	Connections for audio plugs
MIDI	Connections for MIDI plugs
WCLK SYNC	Synchronization settings such as word clock master and slave
SET UP	View information regarding a bus and/or an mLAN device. Transmission plugs of mLAN devices can also be performed on this page.

Tab Pages

Audio and MIDI

Plug Connections

Tool buttons

Update Updates the plug connection display to reflect the current plug connection state of the mLAN devices on the network.

Apply Applies any changes made to the plug connection display to the relevant mLAN devices or buses on the network.

Data List

Vendor Displays the manufacturer of an mLAN device.

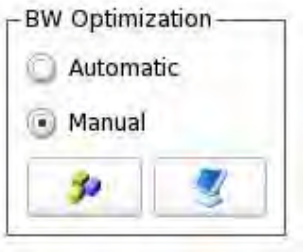
Module Displays the module name of an mLAN device.

Name Displays the nickname of an mLAN device.

Plug Name Displays the plug name of a plug of an mLAN device.

Format Displays the plug format of a plug of an mLAN device.

BW Optimization



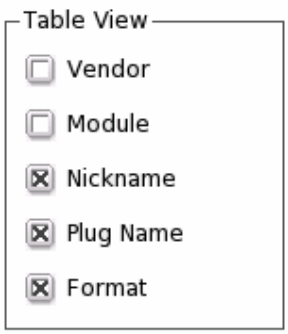
Automatic Performs an automatic network bandwidth optimization each time the **Apply** tool button is clicked.

Manual Allows bandwidth optimization to be performed at any desired stage during an mLAN device's transmission.

 Optimizes bus bandwidth by clearing unused bridge forwarding settings of all bridge portals on the mLAN Network.

 Optimizes bus bandwidth by releasing unused transmission sequences of a selected mLAN device.

Table View



Vendor Switches the Vendor column between displayed and hidden.

Module Switches the Module column between displayed and hidden.

Nickname Switches the Nickname column between displayed and hidden.

Plug Name Switches the Plug Name column between displayed and hidden.

Format Switches the Format column between displayed and hidden.

WCLK SYNC

Tool buttons

Update Updates the WCLK display to reflect the current WCLK synchronization settings of the mLAN devices on the Network.

Apply Applies any changes made on the WCLK display to the relevant mLAN devices or buses on the network.

Synchronization

Tool buttons

Module Switches the Module column between displayed and hidden.

Data List

Module Displays the module name of an mLAN device.

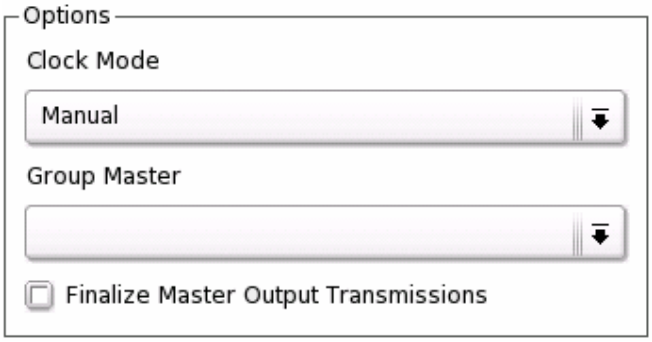
Name Displays the nickname of an mLAN device.

Clock Rate Displays the current clock rate of a master capable mLAN device.

Information

Displays information regarding the capabilities and state of the word clock implementation of the currently selected mLAN device.

Options



Specifies various configuration options for word clock synchronization.

Clock Mode	Specifies the word clock operational mode.
Group Master	Specifies the device to be word clock master on the network
Finalize Master Output Transmissions	When checked, this indicates that isochronous resources should be allocated to all the transmission plugs of any mLAN device configured to be a word clock master. This

also prevents any form of bandwidth optimization to be performed on the device.

SET UP

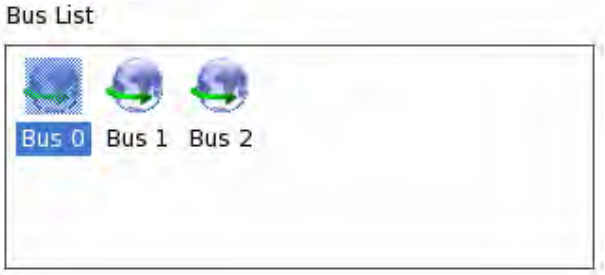
Network

Tool buttons

Update Updates the display to reflect the current configuration of the buses and mLAN devices on the network.

Apply Applies any changes made on the display to the relevant mLAN devices or buses on the network.

Bus List

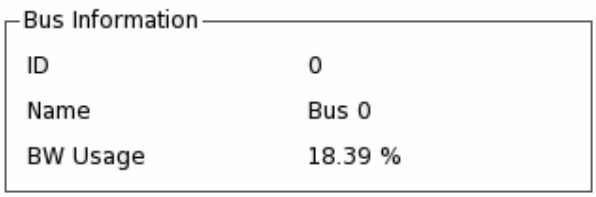


Displays a list of all the IEEE 1394 buses on the network.

Device List

Displays a list of all the mLAN devices on the currently selected bus.

Bus Information

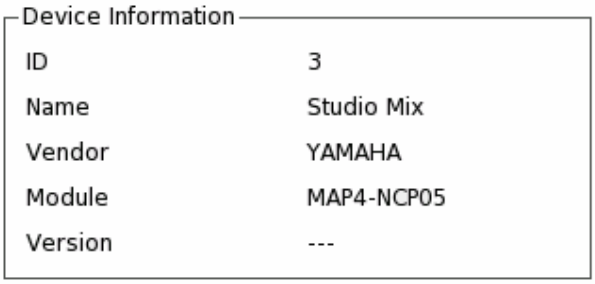


ID Displays the bus ID of the currently selected bus.

Name Displays the name of the currently selected bus.

BW Usage Displays the percentage of the total bus bandwidth in use by any streaming device on the currently selected bus.

Device Information



ID Displays the node ID of the currently selected mLAN device.

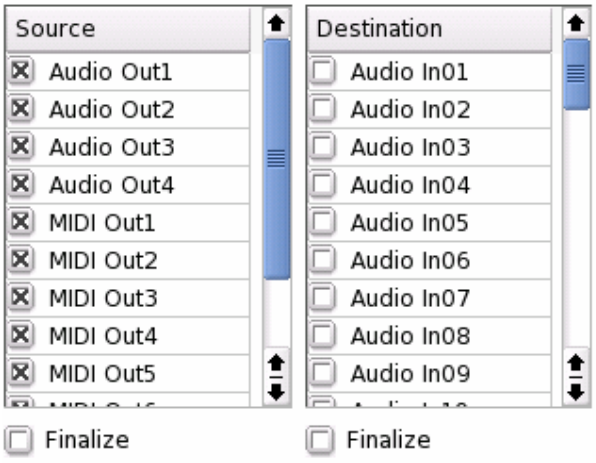
Name Displays the name of the currently selected mLAN device.

Vendor Displays the manufacturer of the currently selected mLAN device.

Module Displays the module name of the currently selected mLAN device.

Version Displays the version information of the currently selected mLAN device.

Plug Configuration



Source	Displays all the output plugs of the currently selected mLAN device. A checked plug indicates that the plug is actually transmitting audio/MIDI data or intended to be transmitting.
Destination	Displays all the input plugs of the currently selected mLAN device. A checked plug indicates that the plug is receiving audio/MIDI data.
Finalize	When checked, this indicates that the currently selected plugs are intended to be used exclusively for transmissions. This prevents dynamic isochronous resource allocation as well as bandwidth optimization to be performed on the mLAN device.

Operations

The table view of the Audio, MIDI and WCLK SYNC tab shows various plugs, in the case of Audio or MIDI, or devices in the case of WCLK SYNC. The entries shown on these pages are colour coded.

For the Audio and MIDI tabs, a red entry indicates that the plug associated with the mLAN device is in an “inactive” state. If it is a source plug, this could either mean that the mLAN device as a whole is not transmitting audio or MIDI data or the selected plug has not been allocated isochronous resources for transmissions. If it is a destination plug, this could either mean that the device as a whole is not receiving audio/MIDI data or the selected plug has not been configured for reception.

A green entry indicates that the plug is in an “active” state. If it is a source plug, this indicates that isochronous resources have been allocated to the plug for transmissions and is indeed transmitting audio or MIDI data over the network. If it is a destination plug, this means that the plug has been configured to receive audio or MIDI data. A green entry with no corresponding source or destination plug indicates a dangling plug connection.

A black entry indicates that the connection state of the plug has been modified by the user and may not reflect the actual state of the network. In this case either click the **Apply** tool button to effect to the network any changes made on the display, or click the **Update** tool button to show the current connection state of the network.

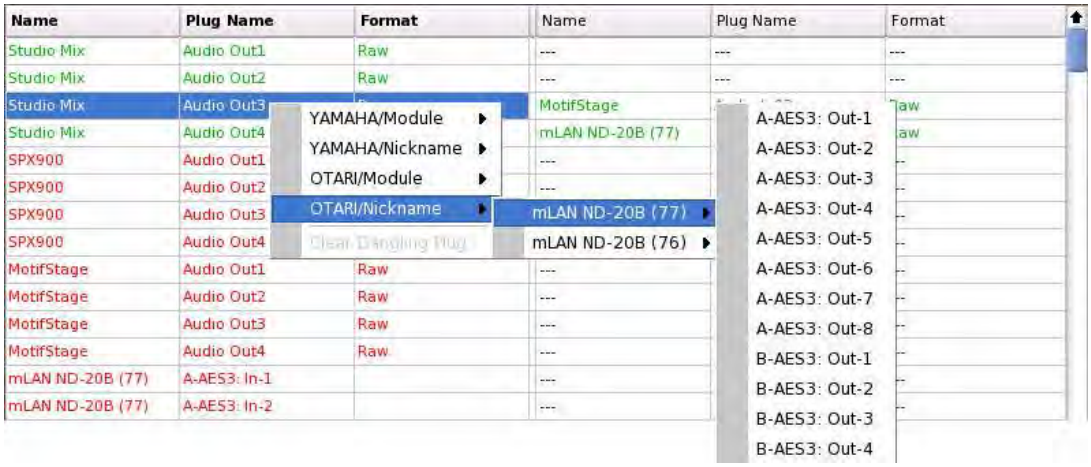
For the WCLK tab, a blue entry indicates the current synchronization state of the mLAN devices on the Network. Any synchronization modification made to the table results in one or more red or black entries shown. An entry shown in red, means that the mLAN device cannot act in the capacity of a word clock master or slave, and will be removed from the table once the **Apply** tool button is clicked. An entry shown in black indicates that synchronization connection of the mLAN device has been modified by the user and may not reflect the actual word clock synchronization state of the Network. In this case either click the **Apply** tool button to effect to the network any the changes made on the

display, or click the **Update** tool button to show the current state of the mLAN Network.

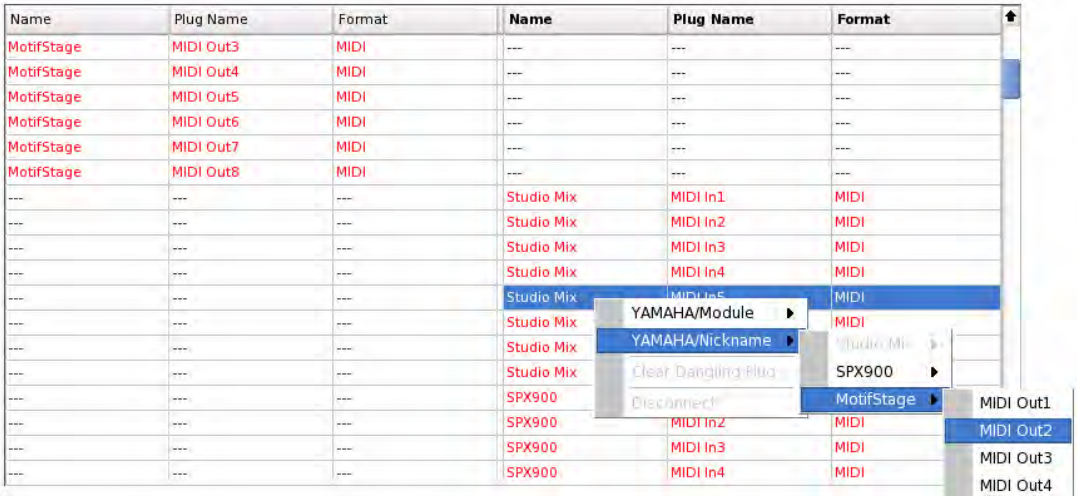
Making a connection

When you right-click a plug either in the source or destination *Audio* or *MIDI* data view, the mLAN plugs available for connection will be shown in a popup menu. By transversing the submenus that appear, select the desired mLAN plug and then click it to update the data view. To effect the connection, click the **Apply** tool button.

Establishing a connection by right-clicking the left hand-side of the Audio data view.



Establishing a connection by right-clicking the right hand-side of the MIDI data view.



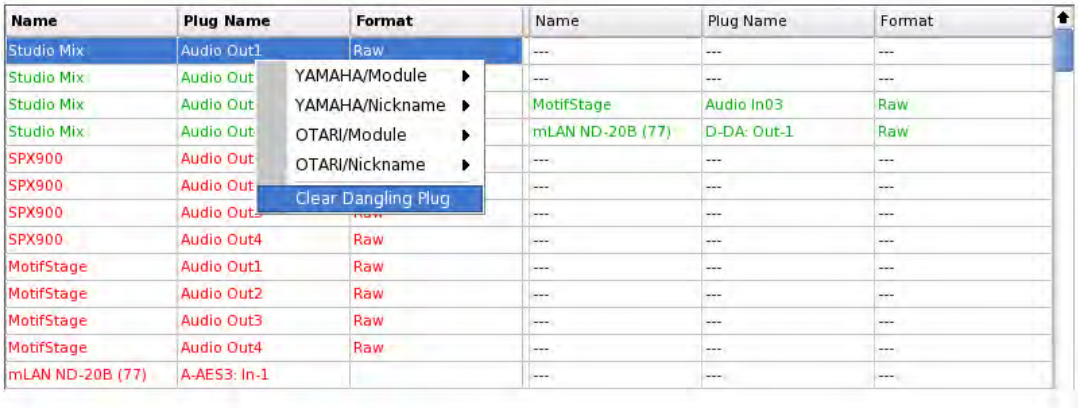
Right-clicking an area that is displayed as "---" has no effect. mLAN plugs that greyed out cannot be selected.

Breaking a connection

To disconnect a connected destination plug, right-click the desired plug and select the *Disconnect* command of the resulting popup menu. To effect the disconnection, click the **Apply** tool button.

Clearing dangling plug connections

To clear a dangling plug connection, right-click the desired plug and select the *Clear Dangling Plug* command of the resulting popup menu. To effect this change, click the **Apply** tool button. The dangling connection state of a plug can only be cleared on entries shown in green that do not have any corresponding source or destination plug.



Optimizing bandwidth usage

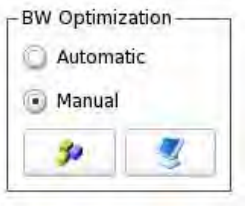
The bus bandwidth in use by an mLAN device is optimized by releasing unused transmission sequences of the mLAN device. To optimize an mLAN device's bus bandwidth, the following procedure is used:

1. First select the device to be optimized by clicking on any of the source or destination plugs of the mLAN device.
2. Ensure that the "Manual" option of the BW Optimization group box is selected.
3. Click the appropriate tool button.

STEP 1: Select the mLAN device

Name	Plug Name	Format	Name	Plug Name	Format
Studio Mix	Audio Out1	Raw	---	---	---
Studio Mix	Audio Out2	Raw	---	---	---
Studio Mix	Audio Out3	Raw	MotifStage	Audio In03	Raw
Studio Mix	Audio Out4	Raw	mLAN ND-20B (77)	D-DA: Out-1	Raw

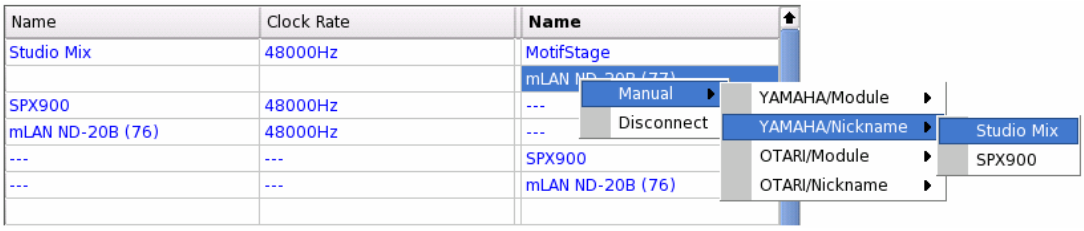
STEP 2: Select the BW Optimization *Manual* option



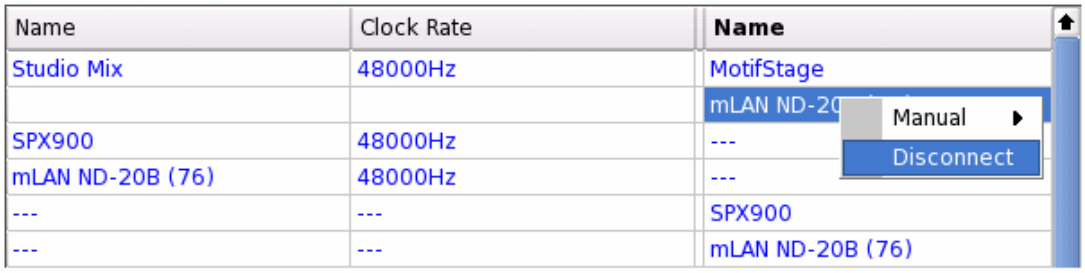
STEP 3: Click the appropriate tool button

Master and Slave settings

When you right-click an mLAN device in the *Slave/Slave Capable* section of the *WCLK* data view, the mLAN devices that can be selected as the master will be displayed. Transverse the submenus of the popup menu to select the mLAN device that you wish to select as master. To effect the synchronization setting, click the **Apply** tool button.



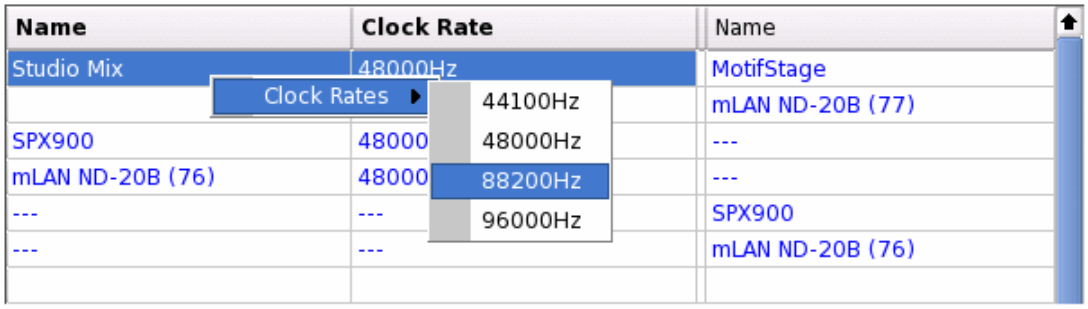
An mLAN device acting as a slave is not master capable and hence not listed under the *Master/Master Capable* section of the table view. To make a slave device master capable, first disconnect the slave device from its assigned word clock master. This can be done by right-clicking on the slave device and selecting the *Disconnect* command on the resulting popup menu. Click the **Apply** tool button to effect the modification.



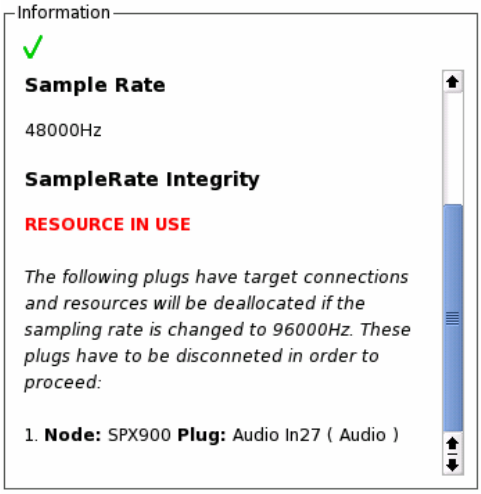
If no word clock master is assigned to an mLAN device listed under the *Slave/Slave Capable* section and the device is not listed under the *Master/Master Capable* section of the table view, this indicates that the mLAN device is configured to receive synchronization but has no corresponding word clock master. To make the slave mLAN device master capable, Click the **Apply** tool button.

Specifying a Clock Rate

The sample rate can only be changed on the mLAN devices listed under the Master/Master Capable section of the table view. To change the sample rate, right-click the desired mLAN device and traverse the *Clock Rate* popup to select the sample at which the device should be synchronizing. Any slave mLAN devices will be configured appropriately.



Changing the sample rate of an mLAN device may result in a change in the availability of resources in use by the mLAN device. This may have an impact on the availability of one or more plugs of the device. In the case where plugs will not be available, resulting from a sample rate change, if these plugs have active connections, a warning message similar to the one shown below, is displayed in the *Information* section of the WCLK SYNC tab.



The plugs listed in the warning message will need to be disconnected before the specified sample rate can be changed.

Finalizing Transmission Plugs

The transmission plugs of a device can be finalized by using either one of two approaches:

1. Finalize Master output transmissions
2. Finalizing selective plugs

To finalize Master output transmissions:

1. Select the WCLK SYNC tab
2. Check the check box labelled *Finalize Master output transmissions*
3. Click the **Apply** tool button.

To finalize selective plugs:

1. Select the SET UP tab
2. Navigate through the list of buses in the *Bus List* view to select the desired mLAN device to be finalized.
3. Select the plugs to be finalized
4. Check the check box labelled *Finalize*
5. Click the **Apply** tool button.

Changing the name of an mLAN device

The following steps are required to change the name of an mLAN device:

1. Select the SET UP tab
2. Navigate through the list of buses in the *Bus List* view to select the desired mLAN device.

3. Click on the mLAN device icon to select it, and then click again on its name to have it modified.
4. Click the **Apply** tool button.
5. To identify an mLAN device, double click on the icon representing the device.

List of References

[1394TA, 2001]. 1394 Trade Association, *'AV/C Music Subunit 1.0'*, 1394TA. 2001.

[1394TA, 2002a]. 1394 Trade Association, *'AV/C Connection and Compatibility Management Specification 1.1'*, 1394TA. 2002.

[1394TA, 2002b]. 1394 Trade Association, *'AV/C Descriptor Mechanism 1.2'*, 1394TA. 2002.

[1394TA, 2004]. 1394 Trade Association, *'AV/C Digital Interface Command Set General Specification, Version 4.2'*, 1394TA. 2004.

[1394TA, 2005]. 1394 Trade Association, *'1394 Technology'*. [Online] Available: http://www.1394ta.org/Technology/index.htm. [Accessed 2005-12-10].

[AES Inc., 2002]. Audio Engineering Society, *'AES47-2002: AES standard for digital audio -- Digital input-output interfacing -- Transmission of digital audio over asynchronous transfer mode (ATM) networks'*, Audio Engineering Society. 2002.

[AES Inc., 2005]. Audio Engineering Society, *'AES50-2005: AES standard for digital audio engineering - High-resolution multi-channel audio interconnection'*, Audio Engineering Society. 2005.

[AES SC, 2005]. Audio Engineering Society – Standards Committee, *'Draft AES Standard for Audio over IEEE 1394 --- Specification of Open Generic Transporter'*, Audio Engineering Society. 2005.

[AES TC-NAS, 1998]. Audio Engineering Society – Technical Committee on Network Audio Systems, *'Networking Audio and Music using Internet2 and Next-Generation Internet Capabilities'*. [Online]. Available: http://www.aes.org/technical/documents/i2.pdf. [Accessed 2005-12-14].

[AES TC-NAS, 2005]. Audio Engineering Society – Technical Committee on Network Audio Systems, *'Audio Networking Survey'*. [Online]. Available: http://nas.cim.mcgill.ca/, http://tinyurl.com/9lzur. [Accessed 2005-12-10].

[Anderson, 1999]. Anderson D., *'Firewire System Architecture – IEEE 1394a'*, 2nd Edition, Addison Wesley, Canada. 1999.

[Apogee Electronics Corp., 2005]. Apogee Electronics Corporation, *'Gear Guide'*. [Online]. Available: http://www.apogeedigital.com/products/. [Accessed 2005-12-10].

[Apple Computer Inc., 2005]. Apple Computer Inc., *'Developer Connection'*. [Online]. Available: http://developer.apple.com/documentation/DeviceDrivers/Conceptual/WorkingWFireWireDI/index.html. [Accessed 2005-12-10].

[Aviom Inc., 2005]. Aviom Inc., *'Aviom Pro64 Products'*. [Online]. Available: http://www.aviom.com/pro64_index.cfm. [Accessed 2005-12-10].

[Axia Audio/TLS Corp., 2005a]. Axia Audio/TLS Corp., *'Axia – Professional Networked Audio'*. [Online]. Available: http://www.axiaaudio.com/. [Accessed 2005-12-10].

[Axia Audio/TLS Corp., 2005b]. Axia Audio/TLS Corp., *'Professional Networked Audio'*. [Online]. Available: http://www.axiaaudio.com/brochures/axia_9-9-2005_screen.pdf. [Accessed 2005-12-10].

[Booch, Rumbaugh and Jacobson, 1999]. Booch G., Rumbaugh J., and Jacobson I., *'The Unified Modelling Language User Guide'*, Addison Wesley, Massachusetts. 1999.

[BridgeCo AG, 2005]. BridgeCo AG, *'BridgeCo – Building Networked Entertainment'*. [Online]. Available: http://www.bridgeco.net. [Accessed 2005-12-10].

[D&R Electronica, 2005]. D&R Electronica B. V., *'CobraNet™ Manager'*. [Online]. Available: http://www.cobranetmanager.com/. [Accessed 2005-12-10].

[Dalheimer, 2002]. Dalheimer K., *'Programming with Qt'*, 2nd Edition, O'Reilly. 2002.

[Dap Technology, 2005]. Dap Technology, *'FireSpy Bus Analyzer Family'*. [Online]. Available: http://www.dapdesign.com/. [Accessed 2005-12-10].

[Digigram, 2003]. Digigram, *'EtherSound Overview, Rev. 1.5c'*. 2003. [Online]. Available: http://www.audioactive.nl/technischmu/Digigram_EtherSound_1.5c.pdf. [Accessed 2005-12-10].

[Digigram, 2004]. Digigram, *'EtherSound Product Range'*. 2004. [Online]. Available: http://www.digigram.com/pdf_brochure/Digigram_EtherSoundRange.pdf. [Accessed 2005-12-10].

[Digigram, 2005]. Digigram, *'EtherSound Technology: Overview'*, [Online]. Available: http://www.ethersound.com/technology/overview.php. [Accessed 2005-12-10].

[Dipert, 2005]. Dipert B., *'CAT5 tracks: Audio goes the distance, reliably and on time'*. EDN. 2005-07-07. [Online]. Available: http://www.edn.com/contents/images/621641.pdf. [Accessed 2005-12-10].

[DTLA, 2005]. Digital Transmission Licensing Administrator. [Online]. Available: http://www.dtcp.com/. [Accessed 2005-12-10].

[Fujimori and Foss, 2003]. Fujimori J. and Foss R., *'Convention Paper: A new Connection Management Architecture for the Next Generation of mLAN'*, Audio Engineering Society: Presented at the 114th Convention, Amsterdam. 2003.

[Fujimori, Foss, Klinkradt and Bangay, 2003]. Fujimori J., Foss R., Klinkradt B. and Bangay S., *'Convention Paper: An mLAN Connection Management Server for Web-Based, Multi-User, Audio Device Patching'*, Audio Engineering Society: Presented at the 115th Convention, New York. 2003.

[Gibson Guitar Corp., 2003a]. Gibson Guitar Corp., *'This is MaGIC'*. 2003. [Online]. Available: http://www.gibsonmagic.com/thisismagic.html. [Accessed 2005-12-10].

[Gibson Guitar Corp., 2003b]. Gibson Guitar Corp., *'Media-accelerated Global Information Carrier, Engineering Specification'*, Gibson Guitar Corporation. 2003. [Online]. Available: http://www.gibsonmagic.com/magic3_0c.pdf. [Accessed 2005-12-10].

[Gibson Guitar Corp., 2003c]. Gibson Guitar Corp., *'MaGIC License FAQ'*. 2003. [Online]. Available: http://www.gibsonmagic.com/faq.html. [Accessed 2005-12-10].

[Gibson Guitar Corp., 2003d]. Gibson Guitar Corp., *'MaGIC Evaluation board system'*. 2003. [Online]. Available: http://www.gibsonmagic.com/eval.html. [Accessed 2005-12-10].

[GNOME Project, 2004]. GNOME Project, *'GLib Reference Manual'*. 2004. [Online]. Available: http://developer.gnome.org/doc/API/glib/index.html. [Accessed 2005-12-10].

[Hear Technologies, 2002]. Hear Technologies, *'Hear Back FAQ'*. 2002. [Online]. Available: http://www.heartechnologies.com/hb/hearback_FAQ.htm#7. [Accessed 2005-12-10].

[IEC, 2003a]. International Electrotechnical Commission, *'IEC 60958-4: Digital audio interface – Part 4: Professional applications (TA4)'*, 2nd Edition, IEC. 2003.

[IEC, 2003b]. International Electrotechnical Commission, *'IEC 61883-1: Consumer audio/video equipment – Digital Interface – Part 1: General'*, 2nd Edition, IEC. 2003.

[IEC, 2005]. International Electrotechnical Commission, *'IEC 61883-6: Consumer audio/video equipment – Digital Interface – Part 6: Audio and music data transmission protocol'*, 2nd Edition, IEC. 2005.

[IEEE, 1995]. Institute of Electrical and Electronics Engineers, *'IEEE Standard for a High Performance Serial Bus – Firewire'*, IEEE. 1995.

[IEEE, 2000]. Institute of Electrical and Electronics Engineers, *'IEEE Standard for a High Performance Serial Bus- Amendment 1'*, IEEE. 2000.

[IEEE, 2001]. Institute of Electrical and Electronics Engineers, *'IEEE Standard for a Control and Status Registers (CSR) Architecture for Microcomputer Buses'*, IEEE. 2001.

[IEEE, 2002]. Institute of Electrical and Electronics Engineers, *'IEEE Standard for a High Performance Serial Bus- Amendment 2'*, IEEE. 2002.

[IEEE, 2005]. Institute of Electrical and Electronics Engineers, *'IEEE Standard for High Performance Serial Bus Bridges'*, IEEE. 2005.

[IEEE SA, 2005]. Institute of Electrical and Electronics Engineers Standards Association, *'1394 with 1000Base-T PHY Technology'*. 2005. [Online]. Available: http://grouper.ieee.org/groups/1394/c/1394cIntroKevinBrown.pdf. [Accessed 2005-12-10].

[IMT Inc., 2005a]. Intelligent Media Technologies Inc., *'SmartBuss'*. [Online]. Available: http://www.intelligentmedia.us/smartbuss.asp. [Accessed 2005-12-10].

[IMT Inc., 2005b]. Intelligent Media Technologies Inc., *'Overview of Intelligent Media Technologies' Embedded Network Solution for Integrated Audio-Video-Data Systems'*, IMT Inc., 2005. [Online]. Available: http://www.intelligentmedia.us/files/smartbuss_intro.pdf. [Accessed 2005-12-10].

[Kaner, Falk and Nguyen, 1999]. Kaner C., Falk J. and Nguyen H., *'Testing Computer Software'*, 2nd Edition, Wiley, Canada. 1999.

[Klinkradt and Foss, 2003]. Klinkradt B. and Foss R., *'Convention Paper 5785: A Comparative Study of mLAN and CobraNet Technologies and their use in the Sound Installation Industry'*, Audio Engineering Society: Presented at the 114th Convention, Amsterdam. 2003.

[Korg USA, 2005]. Korg USA, *'Korg'*. [Online]. Available: http://www.korg.com/. [Accessed 2005-12-10].

[Kuribashi, Ohtani and Fujimori, 1998]. Kuribayashi H., Ohtani Y. and Fujimori J., *'A supplement to Audio and Music Data Transmission Protocol over IEEE 1394: SMPTE time code transmission, Annex A – Delay and jitter evaluation of an audio clock on IEEE 1394'*, Audio Engineering Society: Presented at the 105th Convention, San Francisco. 1998.

[Laubscher, 1999]. Laubscher R., *'An Investigation into the Use of IEEE 1394 for Audio and Control Data Distribution in Music Studio Environments'*, M.Sc. (Computer Science) thesis, Rhodes University, Grahamstown. 1999.

[Lehrman and Tully, 1993]. Lerham P. and Tully T., *'MIDI for the Professional'*, Amsco Publications. 1993.

[Linux1394, 2005]. Linux 1394, *'IEEE 1394 for Linux'*. [Online]. Available: http://www.linux1394.org. [Accessed 2005-12-10].

[MMA, 2001]. MIDI Manufacturers Association, *'Complete MIDI 1.0 Detailed Specification'*, MMA. 2001.

[MSDN, 2005]. Microsoft Developer Network, *'The IEEE Driver Stack'*. [Online]. Available: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/Buses_d/hh/Buses_d/1394-design_35cc5156-ee0d-420f-bdff-f55602ac52ce.xml.asp. [Accessed 2005-12-10].

[NEC Corp., 2002]. NEC Corporation, *'1394 Bridge Application Support Specification Version 0.80'*. January 2002.

[Network Audio Solutions, 2002]. Network Audio Solutions, *'Linux mLAN Connection Management Owner's Manual'*. 2002.

[Network Audio Solutions, 2005]. Network Audio Solutions, *'Linux mLAN Patchbay Owner's Manual Version 0.6'*. 2005.

[NICTA, 2004]. National ICT Australia, *'National ICT Australia Digital Audio Networking'*, 2004. [Online]. Available: http://nicta.com.au/director/research/projects/dante.cfm. [Accessed 2005-12-10].

[NICTA News, 2005]. National ICT Australia, *'NICTA News Issue 03'*, July 2005. [Online]. Available: http://nicta.com.au/uploads/documents/NICTA_News_Issue031.pdf. [Accessed 2005-12-10].

[Nine Tiles Networks Ltd., 2005a]. Nine Tiles Networks Ltd., *'Audiolink 2'*. [Online]. Available: http://www.ninetiles.com/Audiolink2.htm. [Accessed 2005-12-10].

[Nine Tiles Networks Ltd., 2005b]. Nine Tiles Networks Ltd., *'IEC 62379 – Common Control for Networked Audiovisual Equipment'*. [Online]. Available: http://www.iec62379.org/. [Accessed 2005-12-14].

[Novell Inc., 2005]. Novell Inc., *'Novell and SUSE Linux'*. [Online]. Available: http://www.novell.com/linux/suse/. [Accessed 2005-12-10].

[Otari Inc., 2004]. Otari Inc., *'ND-20B Network Audio Distribution Unit Operating Manual'*, 2nd Edition. September 2004. [Online]. Available: http://www.otari.com/support/pdf/nd20e2.pdf. [Accessed 2005-12-10].

[Otari Inc., 2005]. Otari Inc., *'ND-20B Network Audio Distribution Unit'*. [Online]. Available: http://www.otari.com/product/audio/nd_20b/index.html. [Accessed 2005-12-10].

[Oxford Technologies, 2004]. Oxford Technologies, *'Digital Audio Interconnection'*. 2004. [Online]. Available: http://www.sonyoxford.co.uk/pub/supermac/index.html. [Accessed 2005-12-10].

[Oxford Technologies, 2005a]. Oxford Technologies, *'SuperMAC with AES47'*. [Online]. Available: http://www.sonyoxford.co.uk/pub/supermac/aes47.html. [Accessed 2005-12-10].

[Oxford Technologies, 2005b]. Oxford Technologies, *'The SuperMAC/HyperMAC Router'*. [Online]. Available: http://www.sonyoxford.co.uk/pub/supermac/router.html. [Accessed 2005-12-10].

[Oxford Technologies, 2005c]. Oxford Technologies, *'AES Standardisation'*. [Online]. Available: http://www.sonyoxford.co.uk/pub/supermac/aes-standard.html. [Accessed 2005-12-10].

[Peak Audio Inc., 2004a]. Peak Audio Inc., *'Background Specifications and Terminology'*. 2004. [Online]. Available: http://www.peakaudio.com/CobraNet/Background.html. [Accessed 2005-12-10].

[Peak Audio Inc., 2004b]. Peak Audio Inc., *'Frequently Asked Questions'*. 2004. [Online]. Available: http://www.peakaudio.com/CobraNet/FAQ.html#CobraNet. [Accessed 2005-12-10].

[PreSonus Audio Electronics Inc., 2004]. PreSonus Audio Electronics Inc., *'Products'*. 2004. [Online]. Available: http://www.presonus.com/products.html. [Accessed 2005-12-10].

[StarDraw.com Ltd., 2005]. StarDraw.com Ltd., *'Support for Stardraw Control'*. 2005. [Online]. Available: *http://www.stardraw.com/products/stardrawcontrol/FAQ.asp?FAQ=005*. [Accessed 2005-12-10].

[Teener, 2005]. Teener M., *'Residential Ethernet – a status report'*, February 2005. [Online]. Available: http://www.teener.com/ResidentialEthernet/Residential%20Ethernet.pdf. [Accessed 2005-12-10].

[TLDP, 2003]. – The Linux Documentation Project, *'Program Library HOWTO Version 1.20'*. April 2003. [Online]. Available: http://www.tldp.org/HOWTO/Program-Library-HOWTO/index.html. [Accessed 2005-12-10].

[Tsegaye, 2002]. Tsegaye M., *'A Comparative Study of the Linux and Windows Device Driver Architectures with a focus on IEEE1394 (high speed serial bus) drivers'*, M.Sc. (Computer Science) thesis, Rhodes University, Grahamstown. 2002.

[Wavefront Semiconductor, 2005]. Wavefront Semiconductor, *'DICE II Overview'*. [Online]. Available: http://www.wavefrontsemi.com/index.php?id=12,19,0,0,1,0. [Accessed 2005-12-10].

[Yamaha Corp., 2000a]. Yamaha Corporation, *'mLAN 1.0 Connection Control Specification – Draft Version 05 (Rev. 47)'*, Yamaha Corporation. 2000.

[Yamaha Corp., 2000b]. Yamaha Corporation, *'mLAN Patchbay Owner's Manual'*, Yamaha Corporation. 2000.

[Yamaha Corp., 2002a]. Yamaha Corporation, *'mLAN Enabler Software Specification Version 0.5.0'*, Yamaha Corporation. November 2002.

[Yamaha Corp., 2002b]. Yamaha Corporation, *'mLAN-1.0 Transporter Specification – Draft Version 03 (Rev. 26)'*, Yamaha Corporation. August 2002.

[Yamaha Corp., 2002c]. Yamaha Corporation, *'MAP4 Evaluation Board Layout'*, Yamaha Corporation. 2002.

[Yamaha Corp., 2002d]. Yamaha Corporation, *'NC1-Transporter Address Map'*, Yamaha Corporation. 2002.

[Yamaha Corp., 2002e]. Yamaha Corporation, *'NCP04/05 Transporter Specifications'*, Yamaha Corporation. 2002.

[Yamaha Corp., 2002f]. Yamaha Corporation, *'Guide for Bandwidth Efficiency'*, Yamaha Corporation. 2002.

[Yamaha Corp., 2003a]. Yamaha Corporation, *'mLAN Bridge Aware Connections Management Specification – Draft Version 0.3 (Rev. 17)'*, Yamaha Corporation. 2003.

[Yamaha Corp., 2003b]. Yamaha Corporation, *'mLAN-2.0 Configuration ROM specification – Draft Version 0.6 (Rev. 60)'*, Yamaha Corporation. 2003.

[Yamaha Corp., 2004a]. Yamaha Corporation, *'mLAN Enabler Software Specification Version 1.0.1'*, Yamaha Corporation. 2004.

[Yamaha Corp., 2004b]. Yamaha Corporation, *'mLAN Transporter Plug-In Mechanism'*, Yamaha Corporation. 2004.

[Yamaha Corp., 2004c]. Yamaha Corporation, *'mLAN Inter-Application Protocol Version 0.2'*, Yamaha R&D Centre London. May 2004.

[Yamaha Corp., 2005a]. Yamaha Corporation, *'Driver Management'*. [Online]. Available: http://www.mlancentral.com/drivers.php. [Accessed 2005-12-10].

[Yamaha Corp., 2005b]. Yamaha Corporation, *'Otari, Yamaha to Promote mLAN Digital Network Interface Technology'*. [Online]. Available: http://www.yamaha.co.jp/english/news/00092102.html. [Accessed 2005-12-10].

[Yamaha Corp., 2005c]. Yamaha Corporation, *'mLAN FAQ – The Ten Most Asked Questions about mLAN'*. [Online]. Available: http://www.mlancentral.com/mlan_info/faq.php. [Accessed 2005-12-14].

[Yamaha Corp., 2005d]. Yamaha Corporation, *'A Brief Description of Dynamic Sequence Control – Applicable Model: OTARI ND-20B and its associated HAL, Version 1.00'*, Yamaha Corporation. January 2005.

Glossary

across-bus forwarding

The act of transmitting aynchronous or isochronous data between two adjacent buses connected by an IEEE 1394 bridge.

device

A receptor or transmitter of audio or music data.

end-to-end connection management

A connection management scheme that allows the true plugs of a device to be connected.

hard end plug

A hardware plug usually found at the back of audio devices

host implementation

The part of an audio device that makes use of the functionality implemented by the network node that forms part of a device.

node

Refers to that part of an audio device that implements the functionality of transmitting and receiving audio or music data to/from a particular audio network.

node application

See the definitition given to the term *host implementation*

node controller

The low-level network functionality implemented within a node that enables the receipt and transfer of audio and music data.

plug

A hardware or software entity capable of receiving or transmitting a channel of audio or music data.

true plug

The plugs of an audio device that are defined and used by the host implementation of an audio device.