A LINUX SOUND DRIVER FOR FIREWIRE
(IEEE 1394 HIGH PERFORMANCE SERIAL BUS)

AN INVESTIGATION INTO THE IMPLEMENTATION
OF AN ALSA SOUND DRIVER FOR FIREWIRE

THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENT FOR THE DEGREE OF
BACHELOR OF SCIENCE (HONOURS)
OF RHODES UNIVERSITY

BY

JOHN SHEPHERD

NOVEMBER 2002

ABSTRACT

The IEEE 1394 high performance serial bus has emerged as the interconnection technology most suited to the music production studio environment. The bus is used to distribute audio and control data between 1394 studio devices, *viz*. mixers and synthesizers. Studio PC's running audio applications, such as software sequencers, therefore require an interface to the IEEE 1394 bus.

There are no Linux sound drivers available that allow audio applications to stream audio to the IEEE 1394 bus. This thesis attempts to provide a Linux solution to the problem of streaming audio over 1394. It attempts to develop an ALSA based driver to stream the audio. The driver utilizes the Audio and Music Data Transmission Protocol (AMDTP) as its vehicle for audio data streaming.

An ALSA sound card driver was modified in an attempt to create a streaming interface to the 1394 bus. The ALSA sound driver communicates with an AMDTP implementation for Linux.

Ultimately the driver was not successful in streaming audio to the 1394 bus. The major obstacles were firstly discerning where the audio data was exiting the sound driver, and then the extraction of the audio data from the driver. Difficulty was encountered in trying to transfer the data to the 1394 drivers. Future work in this area could include supporting streams of multiple sequences of audio and midi data transfers.

ACKNOWLEDGEMENTS

Firstly, I would like to extend thanks to my supervisor, Prof. Richard Foss, for his support and encouragement, as well his expertise in the field of audio production and engineering. At the same time I would like to thank Melekam Tsegaye (*Linux guru extraordinaire*) for his many useful ideas, and generally helping when Linux and I did not quite see eye to eye. I would also like to thank Bradley Klinkradt for all his helpful insights.

Secondly, I would also like to thank Ben Harper, for his encouragement and tire-less, remote proof-reading, and my mother also for proof-reading this thesis.

Last, but of course not least, I would like to thank my girlfriend, Wendy, for all her support and encouragement when I thought I would never finish this thing!

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

The IEEE 1394 bus (or *FireWire*, as it was termed by Apple) is an emerging, high-speed networking technology that is poised to become the *de facto* industry standard in the multimedia arena. Music production studios are moving towards using IEEE 1394 as the interconnection means between audio devices. It is used to distribute digital audio and control data between devices in studios.

However, there are no Linux sound drivers that support *FireWire*. This project hinges on modifying one of the two existing Linux sound architectures in such a way as to allow the transmission of digital audio data over *FireWire* utilizing the IEC 61883-6 standard (Audio and Music Data Transmission Protocol). The two architectures are the native, standard sound architecture, the OSS (*Open Sound System*) and a newer architecture, the ALSA (*Advanced Linux Sound Architecture*). OSS is becoming outdated, while ALSA is ready to take its place as the standard Linux sound architecture. ALSA has advantages over OSS which have led to the belief that it is superior to the native Linux sound architecture. It provides a stable and universally accepted platform on which to base a driver interface between the worlds of audio and *FireWire*.

An ALSA based audio-1394 interface would allow music studio owners/directors to fully integrate the IEEE 1394 bus into their existing studios. Radio broadcast corporations could also benefit from such an interface. This is provided they are operating under a Linux platform. This thesis attempts to develop a solution to address this problem. An ALSA sound driver is modified to stream audio to the IEEE 1394 bus, via the Linux Audio and Music Data Transmission Protocol driver.

The organization of this thesis is as follows:

Chapter two introduces the IEEE 1394 bus and introduces ALSA (the relatively young Linux sound architecture), which, at the same time, is compared to OSS (the more mature standard Linux sound architecture). It then goes on to elaborate on the need for a driver of this nature, and where the market for it lies.

Chapter three describes, and furthermore discusses the IEEE 1394 high performance serial bus in more detail. A discussion then follows of the IEC standard pertaining to audio transfer over *FireWire* (61883-6), and how the 1394 bus facilitates it. At the same time a mild case is made for the bus as the future intra-studio interconnection medium. Finally, the IEEE 1394 subsystem as it is implemented for Linux is introduced, as well as how Linux implements the IEC standard for audio transfer over *FireWire*.

Chapter four makes a detour from the IEEE 1394 bus and concentrates on driver writing. A discussion takes place revolving around a number of issues relating to design and implementation of Linux drivers that had effect on the implementation of this driver. This is followed by a more thorough examination of the ALSA driver architecture.

Chapter five discusses and illustrates the various the decisions made regarding design of the Linux sound driver.

Chapter six discusses the attempted implementation of the driver and describes some of the major stumbling blocks in its development. It also examines future opportunities and possible extensions for this project

Chapter seven is the conclusion.

CHAPTER 2

FIREWIRE AND ALSA

This chapter will slowly ease the reader into the proposed task of streaming audio over *FireWire*, by providing background as to why there is currently a need for a driver of this nature. That is to say: an ALSA (*Advanced Linux Sound Architecture*) compliant sound driver that streams audio to the IEEE 1394 High Performance Serial Bus (HPSB)[*]. A comparison is made between IEEE 1394 and other interconnection technologies; viz. MIDI and USB 2.0. Of all the technologies that are capable of transferring large amounts of data *FireWire* is the strongest candidate for use as a studio interconnection mechanism. The chapter is conceptually divided into two main sections. The first is an overview of *FireWire* and where, why and how it plays a role in audio applications, and why it will continue to do so. The second is an introduction to Linux's ALSA and why there is a need for the driver.

2.1. A BRIEF INTRODUCTION TO FIREWIRE

This section provides a backdrop for the subsequent discussion on where *FireWire* and audio meet. *FireWire* is a serial bus, much like USB or the original RS232 serial port that came with the first PC's. Where it differs from other commonly available busses and what makes it particularly suitable for audio data transfers is outlined in the next section. Basically a demonstration is made as to why it is *the* heavyweight contender for audio environments.

[*] Also referred to as *FireWire*, IEEE 1394, 1394 and the HPSB throughout the remainder of this text.

2.1.1. FireWire's Extensive Capabilities:

FireWire has many features that make it a perfect candidate for music and audio applications. There are a vast number of very good reasons for interfacing *FireWire* to a PC's sound system. To name a few that apply to this problem domain:

- A high speed data transmission capability; 100, 200 and 400 Mbps (Megabits per second) (IEEE 1394-1995 and 1394a supplement), and now up to 3.2 Gbps (Gigabits per second) (IEEE 1394.B[†]).
- Hot plug-and-playability; a bus-reset is simply issued if nodes are added to or removed from the bus while it is in operation, and the topology is reconfigured.
- Support for asynchronous and isochronous (guaranteed bandwidth) transfers.
- Support for up to 1024 buses, in accordance with the Control and Status Register (CSR) Architecture Specification (1023 in reality, the last is reserved for broadcast).
- Support for up to 64 nodes per bus (63 in reality, the last is reserved as broadcast).
- 16 petabytes of address space per bus (theoretically each node has 256TB) [Anderson, 1999].

Some of the above will be discussed in further detail in subsequent chapters.

Yamaha were the first pro-audio manufacturer to realize 1394 as a prospective audio device interconnection technology. They developed their own 1394 implementation, *mLAN* (a local area network for music), which extended 1394 into the realms of audio (the evolution of audio devices is discussed in more detail in section 2.2).

[†] The *IEEE* (*Institute of Electronics and Electrical Engineers*) first defined the specifications for the 1394 High Performance Serial Bus in a document known as *IEEE 1394-1995*, which was further supplemented by the *1394a specification*, and then extended by the *1394B* version to incorporate a higher speed serial bus.

In light of these capabilities, a discussion follows in which *FireWire* is further contrasted with existing technologies. A possible contender for the role of audio interconnection technology is also introduced.

2.1.2. FireWire in Comparison with other Technologies:

The first, most strikingly obvious physical capability of the 1394 HPSB is the speed at which data transfers occur over the bus. Data transfer rates extend from 100 Mbps to 3.2 Gbps. MIDI on the other hand can only provide a data transfer rate of 31.25 kbps (kilobits per second). MIDI also has a limitation in terms of distance between devices (15.0 m) and the relatively few channels it supports (16). Already we can see that *FireWire* boasts better physical performance, and not only regarding the speed issue. The 1394 bus allows a maximum distance of 500 m between devices; a far-cry from the 15 m attributed to MIDI cables. To challenge MIDI's limited channel issue raised earlier, Yamaha's 1394 based *mLAN* can support 256 audio/MIDI streams on a single isochronous channel (isochronous channels will be discussed in further detail in chapter three - this is merely used to provide a comparison between the two technologies) [Shapiro, 2002]. Of course, a discussion of this nature would be incomplete, and perhaps a little unfair, without picking on something in the same "division" as *FireWire*. At this point the latest USB offering, version 2.0, steps into the proverbial ring. USB 2.0 offers data transfer rates of up to 480 Mbps, which is faster than any of the first generation versions of 1394. However, although it contends in terms of speed, USB still uses the master-slave architecture [Shapiro, 2002], where the PC's harbour all intelligence and peripherals are forced to take on a subservient role. In FireWire the opposite is true; all the bus functionality is hosted on the nodes, or devices themselves, with certain nodes taking on "managerial" roles. This distributes the responsibilities amongst nodes in the network. In truth, 1394 and USB 2.0 will have common ground, but *FireWire*, with its faster versions will always dominate the high end of the speed spectrum [Shapiro, 2002].

Another point in favour of 1394 is that, although it has undergone a fair number of changes, the older *FireWire* devices are nevertheless still compatible with the bus's newer specifications.

2.1.3. Transferring Audio over FireWire:

By virtue of its speed *FireWire* is the perfect vehicle for handling real-time audio data transfers. The transfer of audio data is a bandwidth intensive activity, since a large amount of data needs to be transferred rapidly and with negligible loss. This is easily dealt with by the high data transfer speeds mentioned above. The technical aspects of digital audio data transfer will be discussed in the next chapter, where the IEC[‡] [61883] standards pertaining to it are introduced. The technical aspects being: what packet format the nodes on the bus recognize, how the bus deals with synchronisation issues and so forth.

2.2. WHERE FIREWIRE AND AUDIO MEET

2.2.1. Audio Device Evolution and Music Studio Environments:

Since the IEEE 1394 standard made its earliest appearance a large number of manufacturers of pro-audio and home-entertainment equipment have started to provide an IEEE 1394 connection on their music production equipment and consumer audio equipment. Pro-audio equipment refers to synthesizers, mixers, multi-track recording consoles, etc... Consumer audio equipment refers to IEEE 1394 capable receivers/amplifiers, digital video cameras and cameras. As stated previously, Yamaha is producing a wide range of IEEE 1394 compatible pro-audio devices in the form of their own *mLAN* (Local Area Network for music).

The increased production of 1394 capable audio devices is leading to *FireWire* becoming more prevalent in pro-audio applications and environments as an alternative audio device

[‡] IEC stands for the International Electrotechnical Committee - an international standards body.

interconnection technology. It is also being proposed as the chief means of future device interconnection, with respect to home-entertainment devices. Music studios making use of large amounts of digital audio equipment are finding it more and more difficult to cope with the vast amounts of cabling required to simply shunt audio data around the studio between different devices. This is aside from the transport and distribution of the control data to the audio equipment, such as word-clock information and MIDI control data [Laubscher, 1999]. Therefore, music production studios are incorporating it more and more into the complex tangle of legacy environments to reduce the quantities of cabling. These so-called “legacy environments” include the likes of MIDI, ADAT and AES/EBU digital audio interfaces and equipment.

Now it is time to see how all of this fits into the environment of sound in PC’s and in particular the Linux sound environment. The next section introduces the targeted Linux sound architecture for developing the sound driver for IEEE 1394.

2.3. AN INTRODUCTION TO ALSA

ALSA is a recent sound architecture implementation for Linux. It began as a kernel hack by Jaroslev Kysela to unlock the potential of his soundcard. It has since then grown into a fully-fledged sound API, supporting a variety of cards. As it stands, ALSA is on the verge of surpassing the older, custom Linux OSS (*Open Sound System*) in terms of usability and as the *de facto* standard in all Linux distributions. It has in fact surpassed OSS with respect to functionality and structure. It is widely believed that ALSA is more flexible and supports many more features and cards than its counterpart.

The ALSA sound environment in Linux is divided into the *alsa-kernel* (or drivers) and *alsa-lib* (or library), as well as an *alsa-utils* (utilities). The *alsa-kernel* is the core of the architecture, while the libraries provide an easy to understand interface for applications programmers. The utilities section provides some basic user facilities, such as adjusting mixer volumes, playing audio files and recording, as well as an incomplete sequencer. A freely available and fairly advanced GUI sequencer has been written for ALSA, *viz.*

Ardour, which has its own project page on the sourceforge.net pages (http://ardour.sourceforge.net).

2.3.1. The ALSA Kernel/ALSA Drivers:

ALSA has a significantly enhanced driver structure in comparison to the OSS architecture. It uses the common object oriented principle of dividing different functions between separate drivers, or modules[§]. This type of abstraction not only leads to a more flexible sound structure, but also provides an easier to understand audio interface. It also uses the technique of having a great deal of shared code between drivers for different cards, which have similar chipsets. This eliminates the problem of having vast amounts of duplicated code. Another benefit of this is that any bug-fixes or improvements made to the shared code will affect all the drivers which use it. In addition, it allows the drivers to behave similarly and in a deterministic manner, due to their use of the same code. Another obvious benefit, of minor importance, is the downsizing of the driver architecture in terms of code.

2.3.2. The ALSA Libraries:

The ALSA system also provides an enhanced Applications Programming Interface (API). This is a well-written and well-designed library that allows programmers to obtain operating status information. The library helps to improve user space to kernel space inter-communications, and generally benefits ALSA-native applications.

So, in short, ALSA provides an enhanced framework for writing drivers and for developing audio applications, the former being the most pertinent in this case. The latter of course is important, but only at the level of applications programming. "*Only*" is said here in the sense that this is strictly a driver implementation project. It is not meant to disregard the importance of applications programming. Having this facility allows

[§] In Linux drivers are written and compiled into object code; these are termed modules. See chapter four for a more detailed discussion on the matter.

programmers to develop extremely high-quality audio applications that would ultimately be used above the proposed sound driver for *FireWire*.

2.3.3. The ALSA Utilities:

The utilities component of ALSA provides a few tools for use with the API. These include simple mixer, playback and recording facilities.

2.4. WHY THE NEED FOR THE DRIVER?

To answer this one needs to answer the question of who would need the facility of being able to use ALSA-based audio applications, and transport the audio over IEEE 1394. But first let us examine why ALSA is better a better sound platform to use in a studio environment.

2.4.1. Why Choose ALSA over OSS?

The previous section introduced ALSA and comparison was made between it and the OSS. It was established that ALSA was a superior architecture in all respects. It better designed and supports backwards compatibility with OSS, so OSS-native applications are still able to work under ALSA. Aside from the distinct ascendancy of ALSA's driver architecture over OSS's, there is the fact that it has an advanced API. This will inevitably lead to the development of very advanced, studio quality sound engineering software.

Aside from all the above, using the OSS architecture, audio can already be streamed to the IEEE 1394 bus, and very simply too! This can be done through *softlinking* the OSS audio device with the device to transfer audio over *FireWire*[**].

[**] This device will be discussed further in subsequent chapters, *viz.* chapters three and six.

2.4.2. Who Needs the Driver?

Having established the superiority of ALSA, music production studios would be inclined to use *it* for all their PC sound requirements. Users would use ALSA compatible sequencing tools and so forth. Since the audio equipment is interconnected with *FireWire* ALSA requires some form of interface to 1394. Therefore, the first obvious promoters of such an interface would be the audio device manufacturer. The users of the interface would be music producers and studio engineers, as well as broadcast studios. Therefore, using a rule that has governed economics for centuries: “the existence of a market”, it can be seen that there is a need for such a driver.

2.5. SUMMARY

In summary it can be said that *FireWire* is the perfect candidate for music studio device interconnection. This will be expanded upon in the following chapter where *FireWire* is discussed in more detail. It was determined that ALSA would be the most favourable Linux sound framework in which to implement an interface to *FireWire*. It has been determined in this chapter that there is definitely a need for an ALSA sound driver for FireWire. The driver would ultimately find its use in music studios. For that matter it could be used in any environment where IEEE 1394 audio peripherals are connected to a PC. For example, 1394 compliant home entertainment devices, like the Sony STR-LSA1, a receiver/AV amplifier.

The next chapter takes a deeper look into the architecture and specifications, and the protocols required for the transfer of audio over the IEEE 1394 bus.

CHAPTER 3

FIREWIRE AND THE AUDIO AND MUSIC DATA TRANSMISSION PROTOCOL

This chapter gives the reader more insight into the IEEE 1394 bus introduced in chapter two. It also introduces IEC standard 61883; and in particular IEC 61883-6 (part 6 of the standard, also referred to as the *Audio and Music Data Transmission Protocol*), which is used to transfer audio data over the 1394 bus to pro-audio devices.

Chapter three also introduces the IEEE 1394 subsystem in Linux. Following this introduction to the IEEE 1394 subsystem in Linux the reader is also introduced to an Audio and Music Data Transmission Protocol driver. This driver implements IEC 61883-6 for the Linux 1394 subsystem.

3.1. FIREWIRE: ON CLOSER INSPECTION

3.1.1. Node Address Space:

In chapter two, reference was made to the 64-bit node address space of the 1394 bus; this section explains the details of this concept. Each node has 256 Terabytes (TB) of node address space allocated to it in accordance with the CSR architecture specifications. The 1394 standard extends the CSR specification. *Figure 3.1* depicts the manner in which nodes are addressed on the 1394 bus [Anderson, 1998]. Note that the first ten bits refer to one of the 1024 possible target bus ID's (0 - 1023); the next six bits to the 64 possible target node ID numbers (0 - 63) and the last 48 bits to the actual node address space of 256 TB.

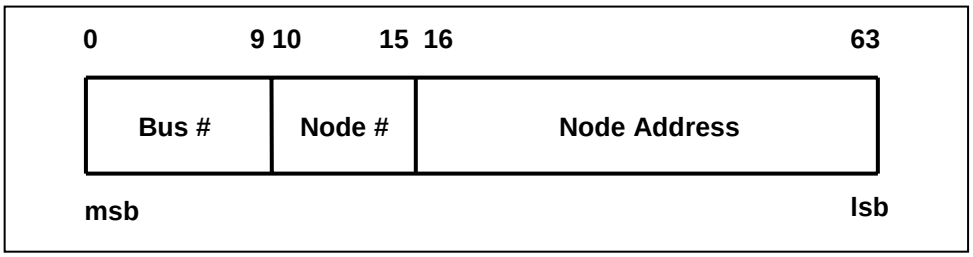

Figure 3.1: IEEE 1394 node addressing.

The 256 TB of each nodes address space is divided into blocks, each with their own function. Part of the space is used to support very necessary serial bus configuration and management [Anderson, 1998].

3.1.2. Asynchronous and Isochronous Data Transfers:

As mentioned, the high performance serial bus has two data transfer mechanisms: asynchronous and isochronous transfers. A major advantage and deciding factor in using *FireWire* as a means of audio interconnection was its support of the isochronous data transfer mechanism. 1394's isochronous data transfer mechanism dictates that data be delivered at constant intervals. The HPSB allocates up to 80% of its bandwidth to isochronous transfers to guarantee the delivery of data. This is important in terms of audio data transmission, since audio transfers are an extremely bandwidth intensive activity.

Asynchronous transfers, on the other hand do not have constant delivery intervals, and are allocated at least 20% of the total bandwidth. Asynchronous transfer occurs via a request specifying the 64-bit destination node address. This type of transfer requires that the receiver acknowledge receipt of data. This type of data transfer is used to distribute control data over the bus. These include messages to set up connections between devices, and to carry out vendor specific functions on devices, such as connection making and connection breaking.

The isochronous transfer mechanism specifies no need for data receipt acknowledgement. This type of transfer uses a 6-bit channel number to specify the target device. “Isochronous listeners” (as devices waiting for isochronous data are termed) use this channel number to access the memory buffer containing the data at the application layer [Anderson, 1998].

Transactions over IEEE 1394 are based on packets. An isochronous packet has a few fields that need to be specified. *Figure 3.2* represents an isochronous packet involved in isochronous transactions. Channel number indicates the isochronous channel on which this data is being streamed. It is used in lieu of an actual physical address. Transaction type indicates that it as an isochronous transaction. The data and CRC (Cyclic Redundancy Check) fields are self-explanatory.

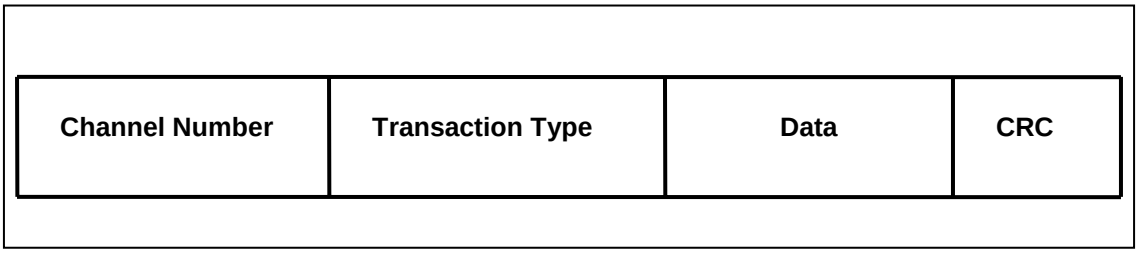

Figure 3.2: A simplified view of the isochronous packet, used during isochronous transactions on the 1394 bus.

3.1.3. Node Capabilities and the Isochronous Cycle:

On each bus there must be nodes that fulfill the role of cycle master (CM), isochronous resource manager (IRM) and bus manager (BM). These roles are determined during a bus reset. The CM broadcasts a cycle start packet, which signals the start of an isochronous cycle. The IRM manages all isochronous resources, such as the available bandwidth and the available channels. During the isochronous cycle isochronous packets can be sent by other nodes on the bus, and they may consume a maximum of 80% of the bandwidth during that cycle - the rest being allocated for asynchronous transactions.

3.1.4. Setting up an Isochronous Transaction:

For a 1394 node to initiate an isochronous transaction it must first obtain a channel number and determine if the there is bandwidth available on the bus. It obtains this information from the isochronous resource manager, contained in the CHANNELS_AVAILABLE and BANDWIDTH_AVAILABLE registers. The target node/s must be configured to respond on a specific channel number associated with the isochronous transaction that it/they should accept.

3.2. THE IEC 61883 STANDARD:

The IEC 61883 standard, (otherwise known as *Digital Interface for Consumer Audio/Video Equipment*[‡‡]) is comprised of six major constituents: 61883-1 through 61883-6. The standard is defined above the IEEE 1394 specification. It standardizes an interface for A/V equipment on IEEE 1394. Part six of the standard is also known as the *Audio and Music Data Transmission Protocol Version 1.0*[§§] [IEC 61883-1/FDIS, 2000]. The 61883, or Digital Interface standard is concerned with real-time (audio and MIDI) data transmission management, connection management and function control. It uses the Common Isochronous Protocol (CIP), Connection Management Protocol (CMP) and Function Control Protocol (FCP), respectively. The A/M protocol is involved solely in transmission synchronisation of Audio and Music (A/M) data, using features offered by 61883 [Laubscher, 1999]. It defines the specifications for the transmission of common digital audio formats and MIDI data over *FireWire*.

Audio/Music data is transferred over *FireWire* using its isochronous transfer mechanism. The packet format used is described in the following sections. There are two packet header formats: the isochronous packet header and the common isochronous packet (CIP) header, which follows the former.

[‡‡] This will be referred to as 61883 from now on in this text.
[§§] This will be referred to as the A/M protocol from now on in this text.

3.2.1. The Packet Header for Audio and Music Data:

3.2.1.1. Isochronous Packet Header Format:

This conforms to the isochronous packet defined by the IEEE 1394-1995 specification. The header of an isochronous packet that forms part of the A/M protocol has the format depicted in *Figure 3.3*.

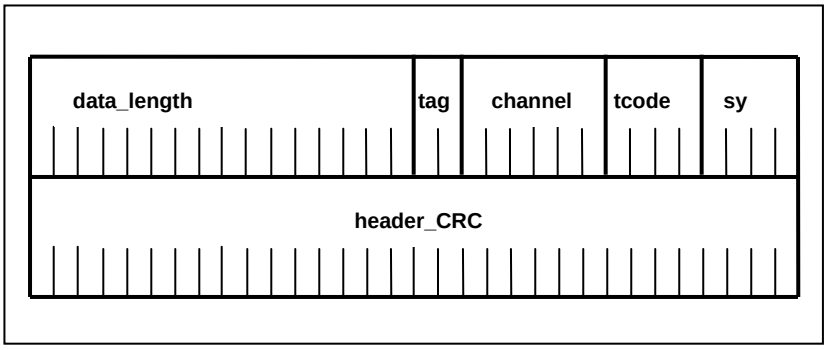

Figure 3.3: The isochronous packet header format.

The isochronous packet header fields have unique values for the A/M protocol, and these are described in more detail *Table 3.1*.

Field	Value	Description
data_length	d.o.p.c	This is the data length in bytes.
tag	01_b	The value given indicates a CIP header is included in the packet.
channel	d.o.p.d	Channel number that the packet is destined for.
tcode	A_{16}	Indicates that the packet is an isochronous one.
sy	Xx	Reserved.
header_CRC	d.o.p.c	CRC check for the header data.

Table 3.1: Field descriptions for the isochronous packet header.

Table Legend
d.o.p.c - depends on packet contents.
d.o.p.d - depends on packet destination.

3.2.2. Common Isochronous Protocol Packet Header Format

3.2.2.1. CIP Header Format

The 61883 standard also defines a two-quadlet CIP header for a fixed length source packet with SYT field (packet timestamp), which forms part of the packet used to transfer A/M data over 1394. The CIP header of an isochronous packet conforming to the A/M protocol has the format depicted in *Figure 3.4* below.

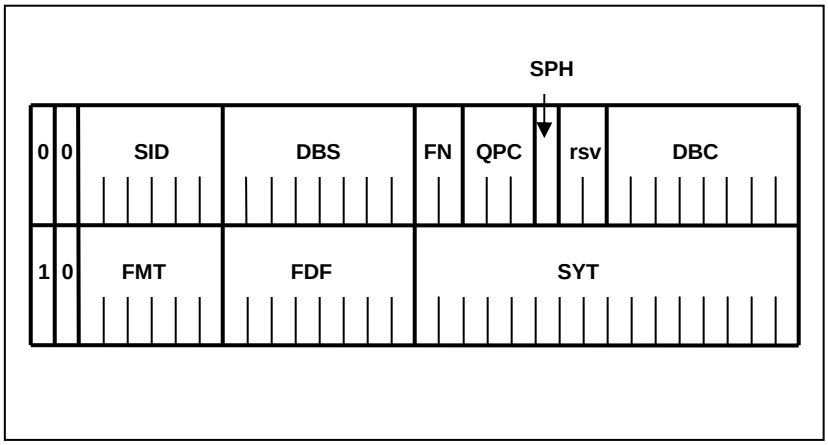

Figure 3.4: The CIP packet header format.

Certain fields have unique values specified by the A/M protocol, which are defined in *Table 3.2*.

Field	Value	Description
SID	d.o.p.s	Node ID of the transmitter.
DBS	d.o.p.c	Data block size in quadlets - max is 256 = max for S100 mode
FN	0_{16}	Fraction number - the number of data blocks that a source packet is divided into.
QPC	0_{16}	Quadlet padding count - number of padding quadlets (0 - 7) to enable division of the packet into equal sized data blocks.
SPH	0_{16}	Source packet header - the value1_b indicates a header.

Table continued overleaf...

Field	Value	Description
rsv	xx	Reserved - all zeroes.
DBC	d.o.p.c	Data block count - detecting packet loss.
FMT	10_{16}	Format ID - this value indicates that the format is for the A/M protocol, i.e. Audio and Music data.
FDF	xx	Format dependent field - indicates attributes of packet data, such as subformat and nominal sampling frequency.

Table 3.2: Field descriptions for the CIP header.

Table Legend
d.o.p.s - depends on packet source
d.o.p.c - depends on packet contents

The timestamp field (SYT) enables a constant delay and reconstruction of the application packet timing on the receiving device [Laubscher, 1999]. Of course there may be more than a single data block in the packets payload. In that case it is necessary to specify to which data block the SYT field refers. The transmitter of the isochronous packet therefore prepares the timestamp for the data block that is a multiple of SYT_INTERVAL, i.e. where **mod** (DBC, SYT_INTERVAL) is zero. SYT_INTERVAL indicates the number of data blocks between two valid, consecutive SYT's.

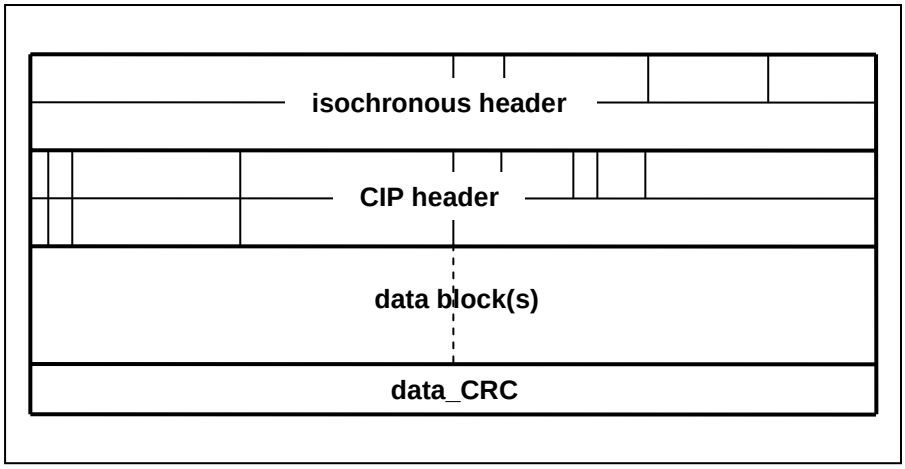

Figure 3.5: An isochronous stream packet - a combined isochronous packet header and CIP header[***].

3.2.2.2. Data Formats within Isochronous Packets:

Digital audio is produced from live sound through the technique of audio sampling. Sound is sampled at regular, frequent intervals and then digitized. This results in a certain number of samples per second, which is termed sample rate. This technique has produced several variant strains of representing audio data.

There are several formats of audio data defined for isochronous packets. Some of the main types that are handled by the Yamaha Map3's[†††] *mLAN-NC1* chip are the AM824 (standing for A/M protocol 8-bit label field and 24-bit data field) and 32-bit floating point data formats [Yamaha Corp., 2001]. The *mLAN-NC1* chip places audio data into and extracts audio data from packets of the form represented in *Figure 3.5* for transmission and reception, respectively. The following formats depicted in *Figures 3.6* to *3.9* constitute quadlets within the **data_block** illustrated in *Figure 3.5*. Other digital audio data types are defined by

[***] Note that the quadlets that make up a packets payload are also known as data blocks.

[†††] Map3 stands for mLAN Audio Port 3, which are Yamaha's experimental 1394 development boards that are A/M protocol compliant (see section 3.2.5).

the A/M protocol, such as 24-bit*4 audio pack data (3 quadlets), however, this is not supported by the *mLAN-NC1* chip.

(a) AM824 data format:

i. *IEC958 conformant* (label = 00_{16} - $3F_{16}$)

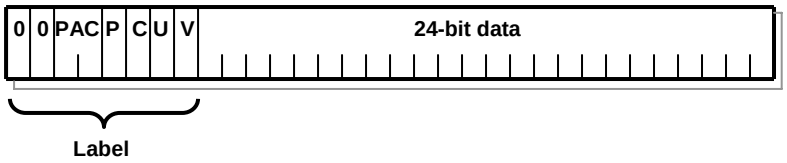

Figure 3.6: Format of IEC958 conformant AM824 data quadlet.

The label bits represent information defined by the IEC 60958 standard. The preamble code (PAC) can represent three values: “M”, “W” or “B” that refer to the start of channels and subframes, respectively, as specified in the IEC958 standard [IEC 60958 spec., 1989].

ii. *Raw audio* (label = 40_{16} - 43_{16})

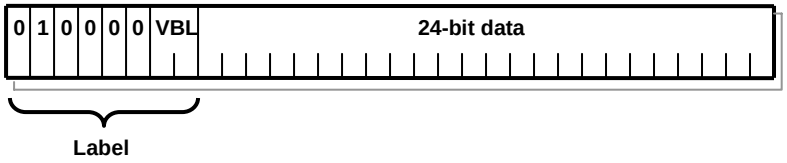

Figure 3.7: Format of the raw audio AM824 data quadlet.

The VBL (Variable Bit Length) field indicates the length of the audio data; 16-bit, 20-bit or 24-bit.

iii. *MIDI conformant* (label = 80_{16} - 83_{16})

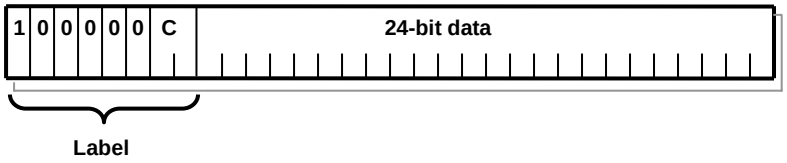

Figure 3.8: Format of the MIDI conformant AM824 data quadlet.

Purely transmitting MIDI data using CIP AM824 over *FireWire* is considered very inefficient. This is due to the number of isochronous transmissions containing no or invalid data and the header size being greater than the actual packet's data [Laubscher, 1999].

(b) 32-bit data format:

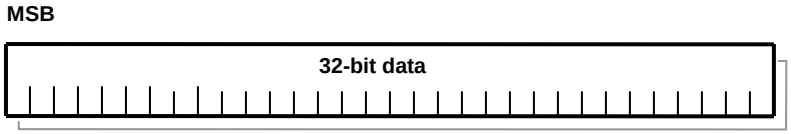

Figure 3.9: Format of 32-bit floating point data.

Although the data format is accepted by *mLAN-NC1*, the scope of this text restricts itself to the discussion of AM824 data types.

3.2.3. A Closer Look at Data Transfer Rates:

The A/M protocol specification sets the stage for some very sophisticated audio transfers over *FireWire*. In terms of the number of real-time audio channels this translates to a large number. If we consider a single "transmitter" and a single "receiver" on the 1394 bus running at 200 Mbps, an 80% guaranteed bandwidth leaves 160 Mbps for isochronous data. There are about 600 (a fair amount of liberty is taken in this estimate, at the expense of the HPSB!) quadlets (32-bit quantities) in an isochronous packet at 200 Mbps, as illustrated in *Table 3.3*. If the nominal sampling rate of real-time audio (48 kHz) is considered, then each packet should carry six audio samples per audio channel

with one quadlet per sample. This will therefore conceptually yield about 100 real-time audio channels; although this figure is bound to be slightly less if we consider other physical overheads. And with a large number of devices this figure is estimated to drop to about 50 [Wiffen, 2001]; calculation courtesy of [Sawada, 2000]. This may seem a large drop, but 50 channels of real-time audio is still a very competitive number, and the higher bandwidth *FireWire* versions will improve on this number. The increase is by more than a factor of two as speed is doubled.

Cable Speed	Data Payload Size (in quadlets)
100 Mbps	256
200 Mbps	512
400 Mbps	1024
800 Mbps	2048
1.6 Gbps	4096
3.2 Gbps	8192

Table 3.3: Maximum data payload for isochronous packets at varying bus speeds.

If further calculations, similar to the one here, are carried out, the following trend as illustrated in *Figure 3.10* appears. Please note these are theoretical figures, and do not reflect the values that might appear in practice.

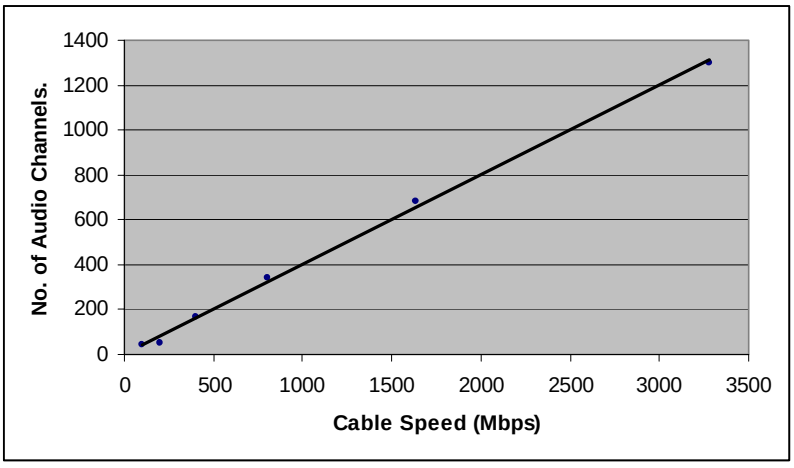

Figure 3.10: Theoretical number of 48 kHz real-time audio channels vs. cable speed.

3.2.4. Synchronisation and Sample Rate Recovery:

The synchronisation of A/M data transmission is an important matter that is controlled by the 1394 bus, and the A/M protocol. A receiver of A/M data must be able to align the samples so they appear in the same time sequence as they were when they were transmitted.

The A/M protocol makes extensive use of the CIP defined by IEC standard 61883 to deal with issues relating to synchronisation. These issues are time alignment and sample rate resolution on reception. The SYT field is utilized to attach a valid timestamp to packets, using the transmitting nodes CYCLE_TIME register [Laubscher, 1999]. Cycle start packets contain clock information based on the CM's 24.576 MHz clock (which is present on every 1394 device). This register is updated with each cycle start packet received. Phase locked looping (PLL) techniques and circuits are used at the hardware level to reproduce the correct timing when packets are received.

The IEEE 1394 bus's synchronisation mechanism is now considered and in particular packetization of audio data samples. Since isochronous cycles occur 8000 times per second (1 every 125 x 10^{-6} seconds), depending on the sample rate of the audio, the number of samples transferred per cycle will vary. Take for example a 44.1 kHz sample rate; if we divide 8 kHz into this we obtain a value of 5.5125 samples per packet. Of course we cannot split samples so one has to alternate between 5 and 6 samples per packet to compensate. Of course this kind of fractional math must be implemented at the driver level, and the fraction type appears in the Linux 1394 IEC 61883-6 driver as [Høgsberg, 2002]:

```
struct fraction {
     int integer;
     int numerator;
     int denominator;
     int counter;
};
```

3.2.5. The Yamaha Implementation of IEC 61883-6:

Previously mention was made of audio device manufacturers building 1394 compliant pro-audio equipment. Yamaha's 1394 implementation *mLAN* in particular is very strong. They have developed a few *mLAN* capable chipsets for their equipment. The particular implementation of *mLAN* to which this section refers is the experimental Map3 development board, which has an *mLAN-NC1* chip on board for IEEE 1394 processing. The *mLAN-NC1* fundamentally provides a solution for physically encapsulating A/M data to be streamed on an isochronous channel over IEEE 1394, and extraction for presentation of the A/M data. The chip has various registers associated with it to control transmission and reception of isochronous packets. The registers can be manipulated via a serial port/terminal window interface. The data types that can be transmitted and received by the boards include MIDI, audio and non-audio. The chip also handles SYT's to ensure timeous presentation of audio. For transmission the word-clock of the audio input is used to create a time stamp. For reception a PLL (phase lock loop) generates the word-clock from the SYT of the isochronous packet [*mLAN-NC1* PH1 block specification, 2001].

The Map3 boards can receive and transmit up to 8 sequences of audio and 2 sequences of non-audio (MIDI), as well as MIDI data streams. Audio sampling rates supported are 32 kHz, 44.1 kHz and 48 kHz. Audio data is received from the 1394 bus processed and then placed in audio FIFOs (First In First Out queues) to be output as digital audio (DA). DA is taken in and packaged and sent onto the 1394 bus. The opposite also occurs.

3.3. THE IEEE 1394 STACK IN LINUX

The previous sections expanded on *FireWire* and the protocol corresponding to the transfer of digital audio data over it. This section describes the currently available IEEE 1394 drivers in Linux which will be used to eventually implement the sound driver.

3.3.1. A Simple 1394 Driver Arrangement:

The most basic 1394 subsystem under Linux consists of the *hardware* or *low-level*, the core *IEEE 1394* and *high-level* drivers.

The hardware driver varies from system to system, depending on which type of IEEE 1394 card is utilized. There are a few available; the OHCI (IEEE 1394 Open Host Controller Interface), PCILynx (*Texas Instruments* PCILynx) and *Adaptec* AIC 5800 (PCI-IEEE 1394 chip) modules. Each module is capable of controlling up to four cards. Above the low-level driver is the core IEEE 1394 driver (ieee1394), which handles all high- and low-level drivers and transactions [Andersen, 1999]. This description refers to the simple setup depicted in *Figure 3.11*. However, for the user to access the 1394 bus the raw1394 module is essential. This module allows *user space* applications to access the bus via a set of library functions called libraw1394 [Bombe].

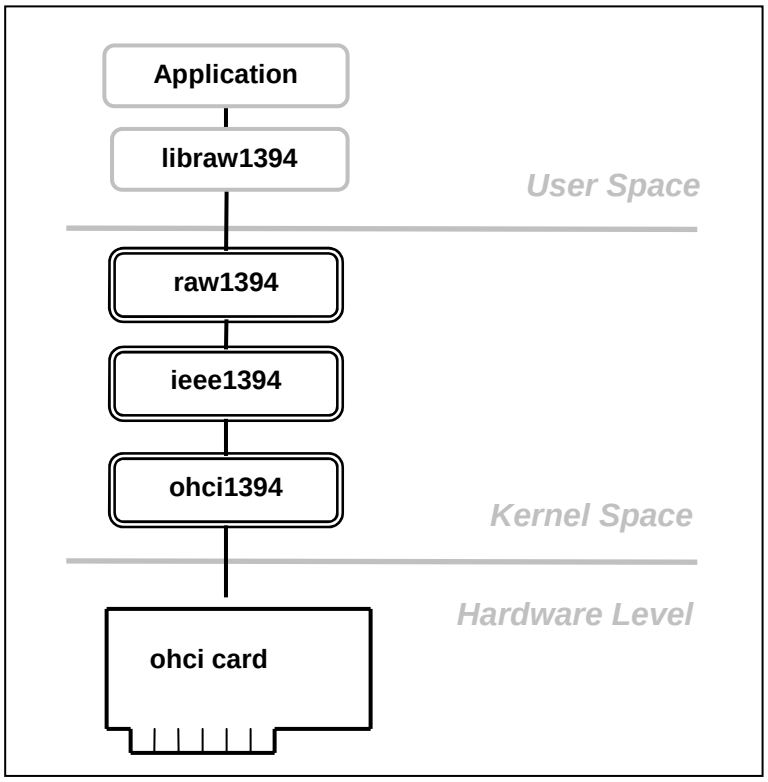

Figure 3.11: An example of a possible, simple 1394 driver hierarchy.

3.3.2. A Linux A/M Protocol Implementation:

Currently there exists in Linux an implementation of the A/M protocol; an Audio and Music Data Transmission Protocol (AMDTP) driver[‡‡‡]. It is a part of the Linux 1394 set of drivers. The A/M driver implements audio data streaming capabilities, in accordance with IEC 61883-6. However, it lacks connection management capabilities. It utilizes the Linux 1394 CMP (Connection Management Protocol) driver for. It also registers itself with the IEEE Core driver. *Figure 3.12* illustrates how the A/M module interacts with the rest of the 1394 environment.

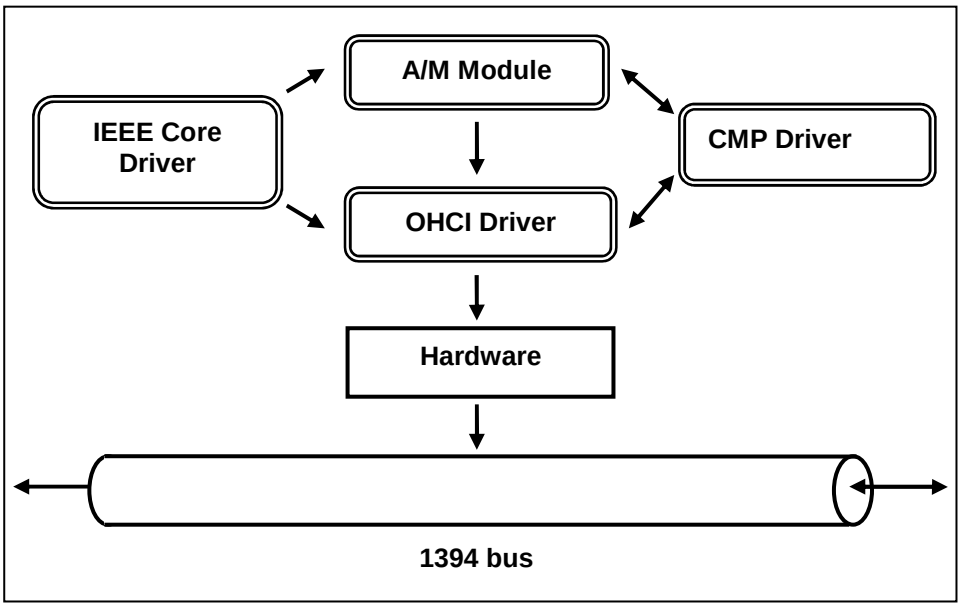

Figure 3.12: How the A/M module interacts with the 1394 subsystem in Linux.

Data can be sent manually to the A/M driver, or via OSS. Sending it via OSS is done by *softlinking* the device files of the OSS and the A/M module. This is relatively simple, compared to writing a driver to open an interface to it and streaming audio. The discussion of device files is beyond the scope of this chapter, and is left till the next.

[‡‡‡] Also referred to as the A/M driver or module in this text.

3.4. SUMMARY

This chapter expanded on the brief introduction to the IEEE 1394 bus given in chapter two. A more detailed investigation was made of IEEE 1394's data transfer mechanisms and how audio makes use of them; in particular the isochronous data transfer. The isochronous mechanism is used since the 1394 bus allocates a large percent of its bandwidth to it. We also introduced the relatively young protocol for the transfer of audio and music (A/M) data over 1394 - the A/M protocol. The protocol specifies certain packet formats that are used within 1394's isochronous mechanism to house audio data on the bus. The protocol handles synchronisation and time alignment of audio data on presentation.

This chapter also introduced the Linux 1394 subsystem, an implementation of IEEE standard 1394 - a high performance serial bus. The reader was also familiarized with the Linux A/M protocol implementation. The A/M driver interfaces to the 1394 subsystem allowing audio to be disseminated to the 1394 bus, according to the IEC 61883-6 standard.

The next chapter introduces driver writing in Linux. It discusses a specific approach to writing drivers and describes with various facilities available to driver writers in Linux and other driver issues.

CHAPTER 4

DRIVERS IN LINUX

This chapter discusses several factors that need to be considered and understood before attempting to design or implement a driver in Linux. In light of the previous chapter, a more detailed inspection is made of the ALSA driver architecture. What follows is a discussion of a few specific issues regarding the ALSA drivers.

The device driver architecture in Linux is simple and yet effective. Drivers are merely there to provide a mechanism for user applications. "*Simple"* and "*merely*" are used rather lightly here. They determine *what* functionality is to be provided, and not *how* that functionality is utilized. How the functionality is utilized is determined by the OS. [Rubini and Corbet, 2001] term this "the distinction between mechanism and policy", which the UNIX (and hence Linux) system implements in its design. This important design decision gives drivers in Linux an immense flexibility [Rubini and Corbet, 2001]. The device driver programmer makes the choice as to how the device appears to applications making use of the facility it provides. Drivers are written as modules. One of the special features of this modularized driver design in Linux is that drivers can be loaded or linked into the kernel at runtime (i.e. while the kernel is operational). The modules are compiled to object code, the linking of which will be discussed further (see section 4.3 Building and Running Linux Modules).

4.1. THE DEVICE DRIVER'S ROLE

The duty of a Linux device driver is to allow user activities, or user-space applications, to communicate with hardware via a set of driver independent, standardized calls. This they do by mapping the calls into device-specific operations to perform hardware-specific functions [Rubini and Corbet, 2001]. Devices in Linux are represented as files in the directory */dev* directory. The device driver must provide device specific `open()`, `close()`, `read()`, `write()`, etc… functions that are called by similarly named low

level operating system (OS) kernel functions when user programs wish to communicate with the device, either to receive or transmit data. All drivers have the following accepted features:

- Performing input/output (I/O) management, providing transparent device management,
- Avoiding low-level programming,
- Increasing I/O speed,
- Including software and hardware error management, and
- Allowing simultaneous access to the hardware by several processes (or concurrency) [Matía, 1998].

Linux drivers are implemented as modules, which are compiled into object code. These modules can be loaded into the kernel at run-time. They can therefore also be seen modular as extensions to the kernel's functionality.

4.2. CLASSES OF DEVICES AND MODULES IN LINUX

There are three major class of device in Linux, namely:

- *Char devices*,
- *Block devices* and
- *network devices* [Rubini and Corbet, 2001]:

4.2.1. Character Devices:

Character devices are accessed as a stream of bytes, for example the memory (*/dev/mem*) and printer (*/dev/lp*). These allow sequential access to the device. Audio devices are character devices. Character drivers therefore allow information transmission between user and device on byte-by-byte basis. A special type of char driver; the terminal driver controls the console or terminal for user communication. These devices require certain

functions (e.g. tab press/auto-finish and up/down arrow press), which the kernel provides for in the way of special routines, and for which the driver itself has special procedures [Matía, 1998].

4.2.2. Block Devices:

Block devices on the other hand are usually data storage hardware, which includes the likes of IDE hard disks (*/dev/hda*) SCSI disks (*/dev/sd1*) and so on. Data is transmitted block-by-block, and hence block drivers make use of a buffer to store data in the interim (between transmission and/or reception).

4.2.3. Network Devices:

The third and final device type is the *network device*, which handles the sending and receiving of data packets. The drivers that control these devices have to handle high speed data flows, and are termed stream drivers. They operate by having several protocol layers at the kernel level, as well as the driver level to handle user space application communication with the network device (e.g. */dev/eth0*) [Matía, 1998].

4.2.4. Major and Minor Numbers:

Each device in */dev* is identified by both a major and a minor number. The major number is used by the *kernel* to bind the driver to that particular device. A driver is capable of controlling several devices, and so the minor number is used locally by the driver to differentiate between the devices to which it may be bound [Rubini and Corbet, 2001]. For example consider the following output from the command `ls`[‡‡‡] `–l /dev/am*`:

```
crw-rw-rw- 1 root   root 171, 240  May 12 2002 amdtp
```

[‡‡‡] The `ls` command lists files in the directory specified after, and the `-l` option lists all file attributes as well.

The *amdtp* device (see chapter 3, section 3.3.2 - *A Linux A/M Protocol Implementation*) is associated with driver number 171, which represents all 1394 drivers in the Linux system. Within that set of drivers the amdtp device is bound to minor number 240, representing the A/M protocol driver. A device file is created initially making use of the following command:

```
mknod –m 666 /dev/amdtp c 171 240
```

This command *mknod* makes the *amdtp* device file in */dev*. The "`c`" refers to it being a *char* device. If the Linux system does not use the *devfs* (or the device filesystem) then the device major and minor numbers must be assigned manually using the *char* device register function defined in `<linux/fs.h>`:

```
int register_chrdev(unsigned int major, const char
     *name, struct file_operations *fops);
```

Where `major` is the major number being requested, name is device name in */dev* and `fops` is a pointer to the array of function pointers, which define the drivers entry points [Rubini and Corbet, 2001]. This call will fail if that major number has already been assigned to another device.

4.3. BUILDING AND RUNNING LINUX MODULES

4.3.1. The Linux Module Interface:

One of the greatest features of Linux is its driver modularization. Drivers in Linux are written as modules, which are vastly different from normal applications. Applications merely run from beginning to end and perform a single, certain task that they were designed to do. On the other hand a driver in Linux will be loaded into the *kernel* and then be capable of serving future requests.

4.3.1.1. Module Initialization:

Linux modules require explicit initialization within their code. To do this they utilize the *module interface*. To make use of the *module interface* one must `#include <linux/module.h>` and `#define MODULE`. Linux drivers, or modules, have two entry points that must be defined within the modules object code:

```
int init_module (void)
void cleanup_module (void)
```

To register these functions modules make use of the following two functions that are defined in the Linux system include file `<linux/init.h>`:

```
module_init (init_module)
module_exit (cleanup_module)
```

The `init_module` and `cleanup_module` functions register and unregister the module with the kernel, basically announcing the presence or departure of the module. This implies the availability of it's functionality to the kernel and other modules.

4.3.1.2. Linking the Module into the Kernel:

The *insmod* command loads a module into the kernel at kernel-runtime and the rmmod command removes it from the kernel (for example):

```
root# insmod ./my_module.o..........(4.3.1)
root# rmmod my_module..............(4.3.2)
```

Line 4.3.1 will load or link `my_module` (from the current directory) into the kernel, while line 4.3.2 will unload `my_module` and any reference to it from the kernel. `insmod` will register the modules services to the kernel and rmmod will unregister them.

This is perfect for testing/debugging and helps to limit the development time of the driver, since one doesn't need to reboot the machine each time a change is made to a driver-in-progress and one has to recompile the code. One could also use the `modprobe` command to load a module into the kernel. However, this loads the module from the *path/to/kernel/source/linux/modules* directory and loads any other modules on which it depends (see section 4.3.2 about driver stacking).

Another difference between a normal application and a module is that a module is linked to the kernel and can therefore only make use of functions that are exported to the kernel. An application on the other hand can make use of functions it doesn't define by simply making use of the correct library (i.e. linking it at compile time). Modules do not have the luxury of linked libraries [Rubini and Corbet, 2001]. The modules described up till this point, run in what is known as the *kernel space*, as opposed to applications that run in the *user space*. These refer to the CPU execution modes, of which there are at least two levels:

- High (sometimes called *supervisor mode*) and
- Low level (also known as *user mode*).

The modularized driver will extend the functionality of the kernel by running in the *kernel space* [Rubini and Corbet, 2001].

4.3.1.3. Kernel Tracking of Modules:

The kernel tracks modules that are in use by making use of a `MOD_IN_USE` count variable. Each module has a `MOD_IN_USE` counter variable associated with it. For each module is incremented each time another module or application makes use of it. Only if this variable is zero for a particular module, can that module be unloaded via *rmmod*.

4.3.2. The Kernel Symbol Table and Driver Stacking:

Within the kernel there is a public symbol table which `insmod` uses to resolve undefined symbols against. This public table contains all the addresses of global functions and variables that are required. Symbols can be exported by a module when it is loaded, and these symbols can be used by other modules or code with the correct privileges. This is how one stacks drivers or modules in Linux. There are numerous examples of it in Linux, even within the mainstream kernel sources. The availability of this feature is particularly useful when implementing large or complex projects, if you want to place new levels of abstraction into a device driver. For example: the IEEE 1394 subsystem in Linux makes use of this stacking mechanism, an example of which is illustrated in *Figure 4.1* [courtesy of Andersen, 1999]. Like the previous mechanism of driver modularization this module layering technique also helps in reducing the development time of projects [Rubini and Corbet, 2001]. For the most part, modules generally implement their own functionality. They do not need to export any symbols to the kernel. A module not exporting symbols must place the following macro in its code:

```
EXPORT_NO_SYMBOLS;
```

If modules need to export symbols to the publicly accessible kernel symbol table, then they must make use of the following macros:

```
EXPORT_SYMBOL (symbol_name);
EXPORT_SYMBOL_NOVERS (symbol_name);
```

where `symbol_name` is the name of the function or variable being exported. Any module that makes use of these exported functions or variables to enhance their own functionality is said to be ‘dependent’ on the module which did the exporting.

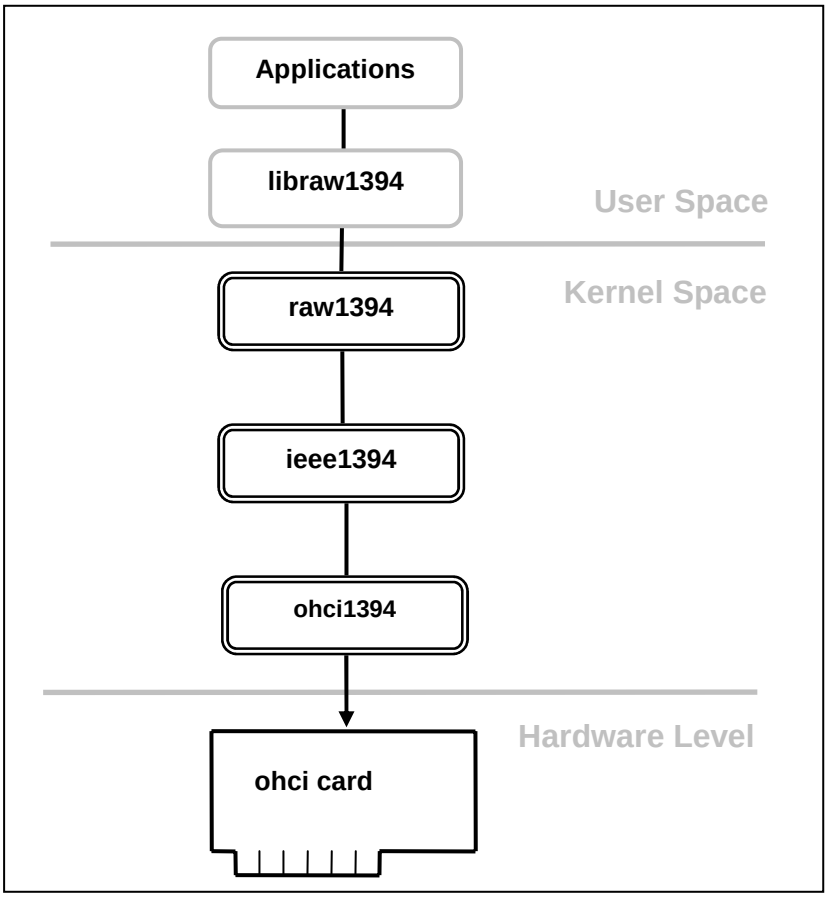

Figure 4.1: An example of a simple IEEE 1394 driver stacking.

4.3.3. Important Kernel Structures:

4.3.3.1. Kernel files and file ops:

This section describes some of the most important structures used in kernel code, and drivers in particular. A driver can carry out a range of actions on the device/s to which it is bound. The kernel uses the `file` struct and the already mentioned `file_operations` struct. A `file` is the OS's internal representation of an open device, and the kernel makes use of the array of pointers contained in `file_operations` to gain access to the driver's file (and hence device)

operations [Rubini and Corbet, 2001]. The operations are named `open`, `read`, `write`, `ioctl` and `close` (to name a few - there are others), which are a driver's implementation of system calls. Each entry in the `file_operations` struct must represent an implemented function in that driver, otherwise `NULL` to implement the default kernel behavior for that particular operation.

The `ioctl` function performs device specific configuration, the `read`, `write` and `open` and `close` functions are fairly self-explanatory. The operations take a `file` as an argument at the very least, and other parameters depending on the function. For instance:

- The `open` will take only a `file` argument,
- The `close` will also take only a `file`,
- The `write` might take a `buffer` to extract data from,
- The `read` a `buffer` to read into as well, and
- The `ioctl` will take a `command` to perform on the device.

These are the more essential operations that a driver should implement. Other operations exist that are used to implement other behaviour, like mapping a device's memory into machine memory.

4.3.3.2. Useful data structures:

Aside from the two very important kernel structures just discussed, Linux offers a great array of tools and aids. These are in the form of data structures for driver writing. These include linked-lists and doubly linked-lists, for simple handling of lists of data or structures.

4.3.4. Exchanging Information with the User Space:

For device drivers, or modules it is often necessary to exchange data with the *user space*. This is done by implementing `read` and `write` functions. Of course there is little more to this! Underlying the read and writes are two simple constructs:

```
copy_to_user(*to, *from, count);
copy_from_user(*to, *from, count);
```

The nature of these functions is simple. The first copies data **from** an address (pointer) in *user space* **into** a data buffer of some sort contained within the module (*kernel space*). The second copies data **from** a data buffer contained within the module **to** an address in *user space*.

4.4. WHEN THINGS GO AWRY

A famous pessimist once made his name with the following sentiment regarding life in general: *"if you don't want it to happen, it will"*. And with driver writing this is no exception. Kernel code is extremely tricky to debug, and the next section provides some techniques designed and used for that purpose.

4.4.1. Debugging Kernel Code:

When writing a driver, or at least a kernel space driver, one needs to monitor the code. However, kernel code is notoriously difficult to debug, since the errors that are generated are difficult to reproduce and can often bring down the entire system! Aside from this traditional debuggers are quite useless when dealing with kernel code [Rubini and Corbet, 2001]. Techniques have been developed to tackle this problem. The most prevalent being that of debugging by using `print` statements. The difference here is of course that you cannot use the `printf`, but instead use has to be made of the `printk`

statement, which logs messages in */var/log/messages*. There are other techniques of debugging such as:

- Allowing your driver to export or write pertinent information to the */proc* directory and reading from that,
- Using a system call known as *ioctl* and constructing some `ioctl` commands in your driver specifically for debugging, which copy useful information to the *user space* to make diagnosis of the problem a little simpler, and finally
- There are also unofficial kernel debuggers available [Rubini and Corbet, 2001].

There is also a tool called the Linux Trace Toolkit (LTT) that allows a user to trace actions occurring in the kernel. It comes as a kernel patch (http://www.oper-sys.comLTT) [Rubini and Corbet, 2001].

Failing all this, there is always the big red button! It might be added, however that this is not the recommended technique.

4.5. THE ALSA DRIVERS IN MORE DETAIL

This section expands on the information provided in chapter two about ALSA. It discusses the devices and modules of ALSA sound kernel. It attempts to offer a detailed explanation of how the ALSA kernel presents a digital audio interface to the Linux kernel. It divides the discussion into a brief explanation of digital audio interfaces, and then, ALSA's manner of representing this.

4.5.1. Digital Audio Interfaces:

A digital audio interface is conceptually a simple scheme. It provides a means of receiving and sending audio from the PC. It is there to transform received audio into a stream of bits and to transform a stream of bits into something meaningful. At the driver level, sending audio entails filling a circular buffer with samples. These are then written into a hardware buffer on a sound card [Davis, 2002] to be turned from digital (PCM)

format to analog. This is of course where we see the sound driver for *FireWire* fitting in. The *FireWire* sound driver must somehow redirect these audio buffers to 1394 bus.

4.5.2. ALSA Devices:

Once the ALSA drivers have been compiled and installed the device files appear in */dev/snd*. These device files represent three types of device: global devices, card dependent devices and card dependent devices allowing multiple device files of the same type [Bartels, 1999]. Examples of ALSA devices appear in *Table 4.1* below

Device Name	Type	Function
/dev/snd/timer	Global dev.	Provide timing information for ALSA sound drivers.
/dev/snd/controlCn	Card dep. dev.	A controlling device for card n.
/dev/snd/pcmCnDm	Card dep. dev. with multiple file types	PCM (audio) device m on card n. Can have multiple audio "subchannels" provided hardware supports it.
/dev/snd/mixerCnDm	Card dep. dev. with multiple file types	Mixer m for card n. Allow volume control of different channels.

Table 4.1: Examples of ALSA devices.

ALSA distinguishes between audio playback and capture (or record) [Davis, 2002], and this is reflected in the cards pcm device (*/dev/snd/pcmCnDm*) by the suffix *"p"* or *"c"*. These are termed an audio device's subchannels. An audio streaming driver for *FireWire* would only implement a playback interface, since a stream is a unidirectional entity (towards the 1394 bus).

4.5.3. ALSA Kernel Modules:

This section describes a few of the important core ALSA kernel modules. The main ALSA sound kernel module is *snd.o*[§§§]. This module is the father of all ALSA modules,

[§§§] Any module that appears as part of ALSA is prefixed by snd-.

with which all other snd modules must register and unregister. It contains functions to allocate and deallocate the memory for a sound card struct, which represents an actual card [Andersen, 1999]. The digital audio (PCM) abstraction module *snd-pcm.o* contains code to provide functionality for PCM devices on a sound card. Finally, the *snd-timer.o* module provides ALSA with timer callbacks for sound card drivers to use to dispatch irq's (interrupt requests) to the ALSA kernel. There is also an extensive armoury of card drivers that were not all supported in OSS.

The ALSA kernel has also been designed such that it is backwardly compatible with OSS by supplying an OSS emulation layer [Andersen, 1999]. The compatibility is provided by linking the OSS device to ALSA's emulation device. Thus applications supporting OSS device types can be mapped into ALSA.

4.6. SUMMARY

In this chapter the reader was alerted to some of the mechanisms and techniques for implementing and helping to implement Linux device drivers. The concept of device files was explained. Devices or peripherals to the system are represented as files in Linux. Drivers need to implement a means to configure these devices. This is done via *ioctl* commands. Also they must implement a means to get data to and from the device; i.e. *reads* and *writes* to the device. This is important design background for the final two chapters, which go onto discuss design considerations and implementation, respectively.

Also discussed in the above was the ALSA device and module structure. ALSA is a very powerful sound framework with many enhanced features in comparison to OSS. If music production studios are going to have Linux running on their PC's then ALSA is definitely going to be the sound architecture of choice.

With this in mind we move onto discussing the design considerations for the sound driver in the next chapter.

CHAPTER 5

DESIGN CONSIDERATIONS AND DECISIONS

This chapter illustrates the decisions that were made regarding the design of the driver. It begins by simply outlining the common issues concerning the placement of a driver within Linux. The key question in this regard is whether it should be a user space "application" type driver, or a kernel space driver? At the outset this was really the first decision that needed to be made.

Further design issues are also discussed in the chapter. These include issues such as:

- The choice of implementation language,
- How the driver will be integrated into the incumbent Linux 1394 subsystem, and
- How to interface it to the A/M protocol driver.

5.1. LANGUAGE OF IMPLEMENTATION

The choice of the language of implementation for the driver was made simpler by the design of the Linux OS. All Linux modules are written in *C programming language*, and to move away from this would probably be a bad decision. In keeping with this "tradition" the driver's implementation would remain the *C programming language*. One of the many strengths of *C* is that it has many low-level programming features, while retaining many of the advantages of a high-level programming language. Some may argue that it is indeed a low-level language! This leads to faster code in general. The feature that makes *C* so desirable as a driver programming language is its pointer mechanisms, which emulate low-level code. Not entirely appropriate here, but more pertinent when programming actual device drivers, is that pointers allow machine-code like access to hardware registers.

5.2. USER SPACE OR KERNEL SPACE DRIVER?

There are two options when considering the placement of a driver in Linux: (1) it can be positioned in the *user space* or (2) it can be deployed in the *kernel space* as part of the 1394 stack of drivers. This concept is depicted in *Figure 5.1*. At the outset this was the first decision that had to be made regarding the drivers' implementation. There are both arguments for and against the *user space* implementation.

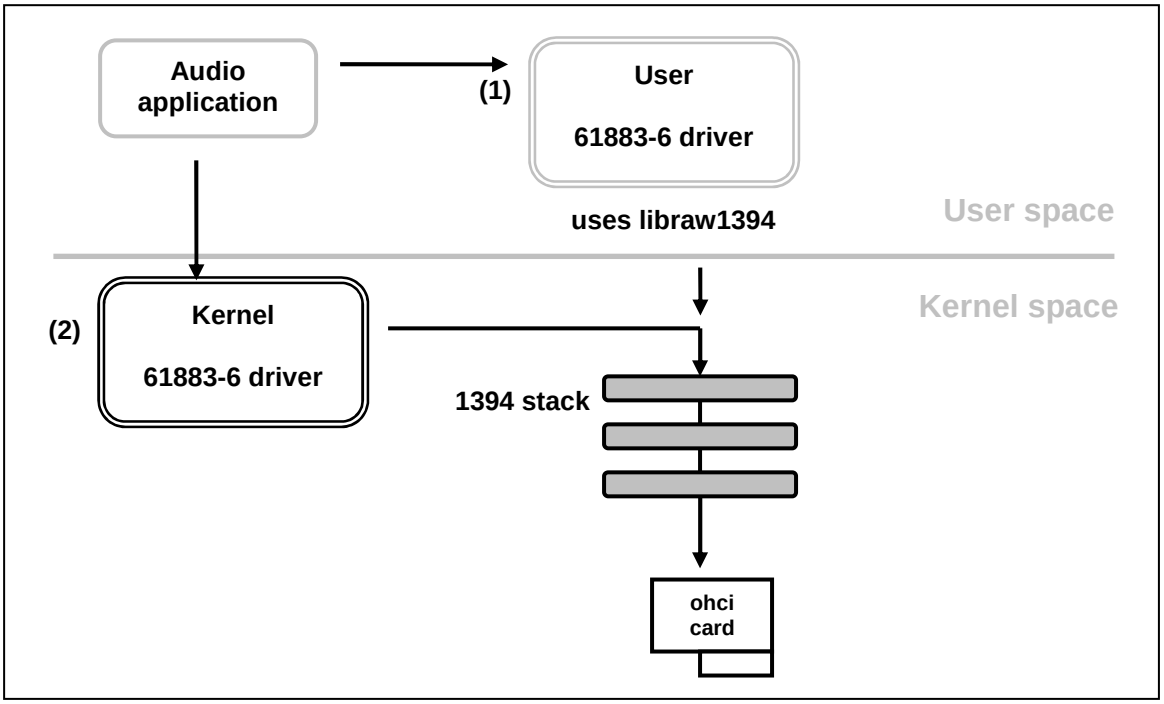

Figure 5.1: Two possible placements of a 61883-6/sound for 1394 driver - (1) the user space and (2) as a part of the kernel.

5.2.1. A User Space Driver:

First, a user space sound driver for the A/M protocol, as illustrated in Figure 5.1 (1) above, is considered. As with any course of action, there are of course pros and cons. The advantages and disadvantages of a driver based in *user space* are outlined in the following two sections.

5.2.1.1. Advantages of a User Space Driver:

Some of the advantages of a *user space* driver, or "application style" include:

- The full C library can be linked to the "application".
- Conventional debuggers or debugging techniques can be run or used.
- If the driver hangs it can be killed with worrying about hanging the entire system.
- User memory is swappable, so if the driver is not used often then it won't be occupying RAM (unlike kernel memory – which is not swappable) [Rubini and Corbet, 2001].

5.2.1.2. Disadvantages of a User Space Driver:

On the other hand there is a downside to writing a *user space* driver:

- Interrupts are not available in the user space.
- DMA (Direct Memory Access) to system memory is only available through `mmap`ping */dev/mem*, which requires privileged user status.
- Response time is impinged on, due to expensive context switching.
- Response time is further deteriorated by swapping of the driver to disk [Rubini and Corbet, 2001].

If a user space driver is written then it will be an application type driver. Hence, it would have to make use of some sort of library of functions (API) to access the kernel drivers, *viz. libraw1394* [Bombe, 2002]. This would be unnecessarily complicated. In that a library would have to be implemented, or an existing library is used. Using an API to access kernel drivers would result in seriously detrimental response times.

5.2.2. A Kernel Space ALSA Driver:

Streaming of audio is a delay-sensitive undertaking. Therefore, driver response times are particularly critical. The fact that a user space driver's response times are so high suggests that a kernel space driver would be the better implementation route to attempt. It has already been determined that ALSA will be the vehicle of implementation for the driver. The ALSA kernel drivers are inherently kernel modules. This essentially *makes* the decision regarding the position. The decision was to make it part of the ALSA kernel drivers by modifying a sound card driver. This is illustrated in *Figure 5.2*. What is depicted in the figure is conceptually in the OS *kernel space*.

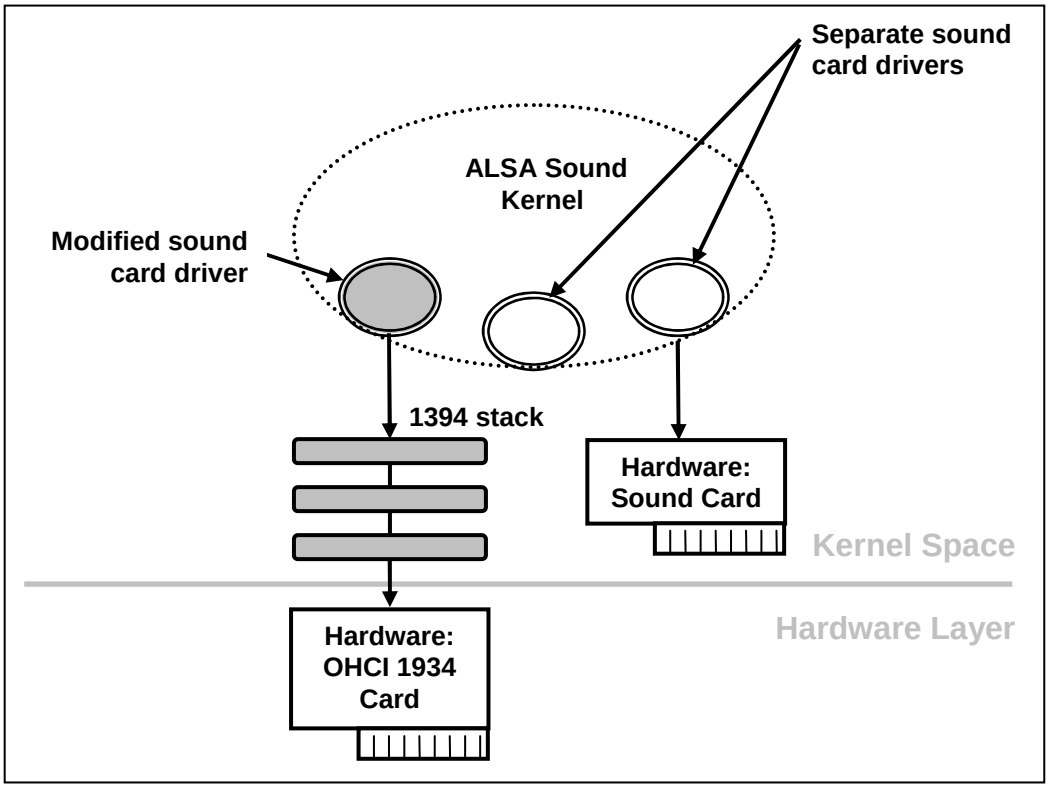

Figure 5.2: The ALSA sound modules positioning in the kernel.

5.3. DRIVER MODULARIZATION

This aspect was less of a design decision than a clear-cut and forgone conclusion: all drivers in Linux are written as modules. All modules have the same pre-defined and carefully structured interface that needs to be followed by driver implementers. This modular interface includes the implementation of specific file operations to access the device, as pointed out in detail in the previous chapter (section *4.3.3. Important Kernel Structures*).

5.4. THE SOUND DRIVER AS PART OF THE LINUX 1394 STACK

This section discusses the positioning of the new sound driver in relation to the Linux 1394 subsystem. In particular it illustrates how the driver will make use of the Linux A/M protocol implementation introduced in chapter three.

5.4.1. The Linux A/M Protocol Driver Revisited:

In chapter three the existence of a Linux implementation of the A/M protocol was alluded to. This implementation is in the form of a driver that constitutes part of the Linux 1394 stack. The driver, as mentioned, provides basic IEC 61883-6 data streaming capabilities. This A/M protocol functionality can be shared with other kernel modules to use. Furthermore, function exporting is made relatively simple by the module interface in Linux described in the previous chapter.

5.4.2. Integrating the Sound Driver into Linux 1394:

The positioning of the proposed ALSA compliant sound driver is illustrated in *Figure 5.3* on page 5. The ALSA sound kernel encapsulates the driver. Furthermore, the driver is located directly above the A/M protocol driver. After careful consideration this approach was deemed the simplest and best to implement. The options here were either to use this method, or to convert the A/M protocol driver into an ALSA module. The reasoning

goes back to the concept of modularization. At all levels of programming it is preferred that dissimilar functionalities are separated from one another into different modules. The differing functionalities here are those of a sound driver, and those of the A/M protocol module. This seems in keeping with ALSA's manner of operation, i.e. a large degree of code compartmentalisation. Also, converting the A/M module would be a more complicated task than simply modifying an existing sound driver. The porting of the A/M module from being a kernel module to an ALSA sound card module is not trivial.

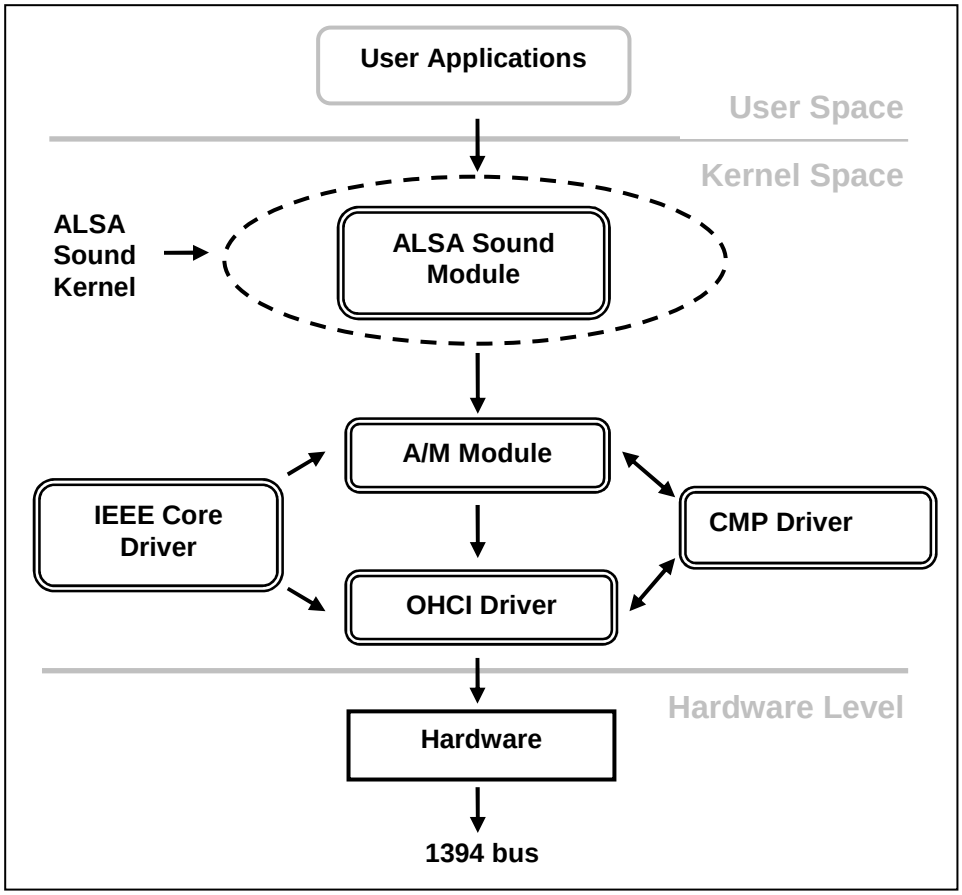

Figure 5.3: Location of the ALSA compliant sound driver for Firewire.

It was thus decided to modify the Linux A/M module such that it provided an interface for another module to stream data to the IEEE 1394 bus. "*Another module*" in this case is the proposed ALSA sound driver. It followed then that the necessary changes would be

required to be made to a sound card driver to allow it to communicate with the A/M protocol driver, and send audio data to it.

5.5. SUMMARY

We can see from the preceding chapter that the design choices when writing the driver really began with choosing the placement of the driver in the OS. The penultimate decision in terms of this was to implement it as a *kernel space* driver. Furthermore it was decided, for various reasons, that it would be best to implement it as an ALSA compliant sound driver. Not least of all these reasons was the delay-sensitive nature of audio transfers. It was decided to then let the IEEE 1394 bus appear as a sound card to other audio drivers and audio applications above it. This sound card module would then interface to the extant A/M protocol module, and thus audio data would be streamed to the 1394 bus.

The next chapter discusses the attempted implementation of this audio streaming driver, the difficulties, the failures and the triumphs.

CHAPTER 6

DRIVER IMPLEMENTATION

The final chapter in this thesis deals with the actual attempt at implementing the sound driver to stream audio over *FireWire*. We begin with a discussion of the early testing of the experimental Map3 development boards and *FireWire* in general. The experiments included testing that the A/M driver *could* stream audio to the boards. It then moves onto the actual attempts at implementation of an ALSA compliant sound driver for *FireWire*; the successes, failures and possible extensions.

6.1. PLAYING WITH FIREWIRE

6.1.1. Experimenting with the Map3 Boards

The learning for this project began in the shallow end of the pool of knowledge, with simple explorations of the IEEE 1394 bus. These *"explorations"* involved fiddling with Yamaha's experimental Map3 development boards. They were tested to see if they were capable of processing raw audio. This was done by:

- Supplying audio at the audio input of board #1
- Allowing it to pass over the IEEE 1394 bus between the two boards, and
- Having it reproduced at the audio output on board #2.

This was eventually successful and after much frustration, with a cryptic hyper-terminal RS232 port being the only thread of communication with the boards. The hyper-terminal can be used to manipulate the 8-bit info. registers of the *mLAN-NC1* chip on the boards. For example the boards have an audio `receive_enable` register and a set of eight registers that are used to monitor the isochronous and CIP header of received audio. *Figure 6.1* illustrates the set of eight registers.

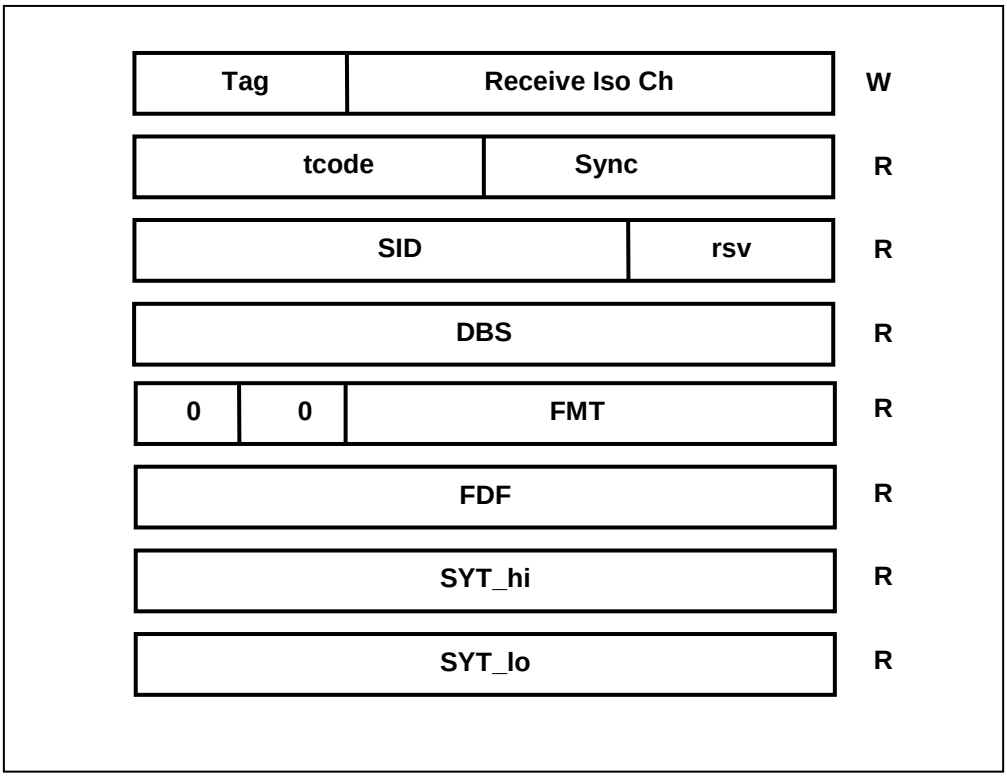

Figure 6.1: The 8-bit register set used by Map3 to monitor isochronous packet headers.

These registers are used to monitor isochronous packets, and hence for the most part they are read-only registers. However, the six isochronous channel receive bits are write, and therefore the channel for reception can be adjusted *via* the RS232 serial interface. Each [of the eight] audio FIFOs on the boards has its own isochronous channel receive registers, as well as an 8-bit LABEL register which can be written to, to specify the AM824 label of the data destined for that FIFO.

A 1394 bus analyzer called *FireSpy400* was used to analyze the packets on the 1394 bus to make sure they conformed to the A/M protocol's specifications.

6.1.2. Testing the A/M Module with the Map3 boards.

The next part in terms of experimenting was to test the 1394 capabilities of the boards using the Linux A/M driver. This process was difficult to eventually get come into fruition. A simple transmission application was developed to stream an audio file (e.g. *.mp3) over *FireWire* to the Map3. This was achieved after a little frustration. The basic concept is depicted in *Figure 6.2*. The program made use of the interface provided by the A/M driver. The interface is a simple one - various *ioctl* commands are understood by the A/M driver, which are used to configure the *amdtp* device. Data writes can then be performed on the device and the audio data is streamed to the 1394 bus via the *ohci1394* hardware driver. All connection management is set up by and managed by the *cmp* module. The A/M driver's *ioctl's* include setting up a channel or a connection to an IEC 61883-1 defined plug mechanism using functions exported by *cmp*. Other *ioctl's* that are taken care of by the *amdtp* module are: setting the format, the sample rate and dimension (i.e. whether it is stereo or mono, etc...) of the audio data being transmitted. Channel can be between 0 and 63, as defined by the IEEE 1394 specifications. The formats supported by the A/M driver are `AMDTP_FORMAT_RAW` (raw AM824 audio), `AMDTP_FORMAT_IEC958_PCM` (normal AM824 audio; 16, 20 or 24-bits per sample) and `AMDTP_FORMAT_IEC958_AC3` (compressed AM824 audio; 16-bits per sample). The sample rates supported are 32, 44.1, 48, 88.2, 96 176.4 and 192 kHz.

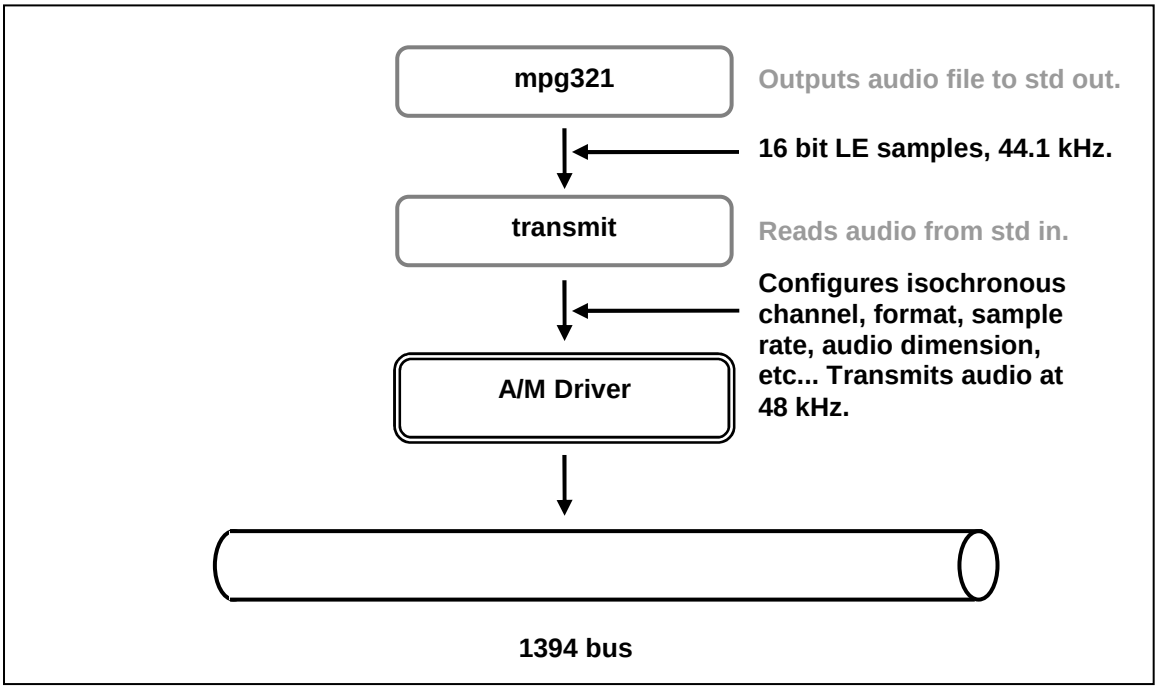

Figure 6.2: The transmission of audio to the A/M driver and subsequently the 1394 bus.

For example: a data packet captured off the *FireWire* network when transmitting using the A/M driver is depicted in *Figure 6.3*. This was done using a 1394 bus analyzer. Note the Isochronous header and CIP header. The bus analyzer cannot recognize CIP headers. If the CIP header is scrutinized more closely it can be seen that it conforms to the A/M protocol discussed in chapter three.

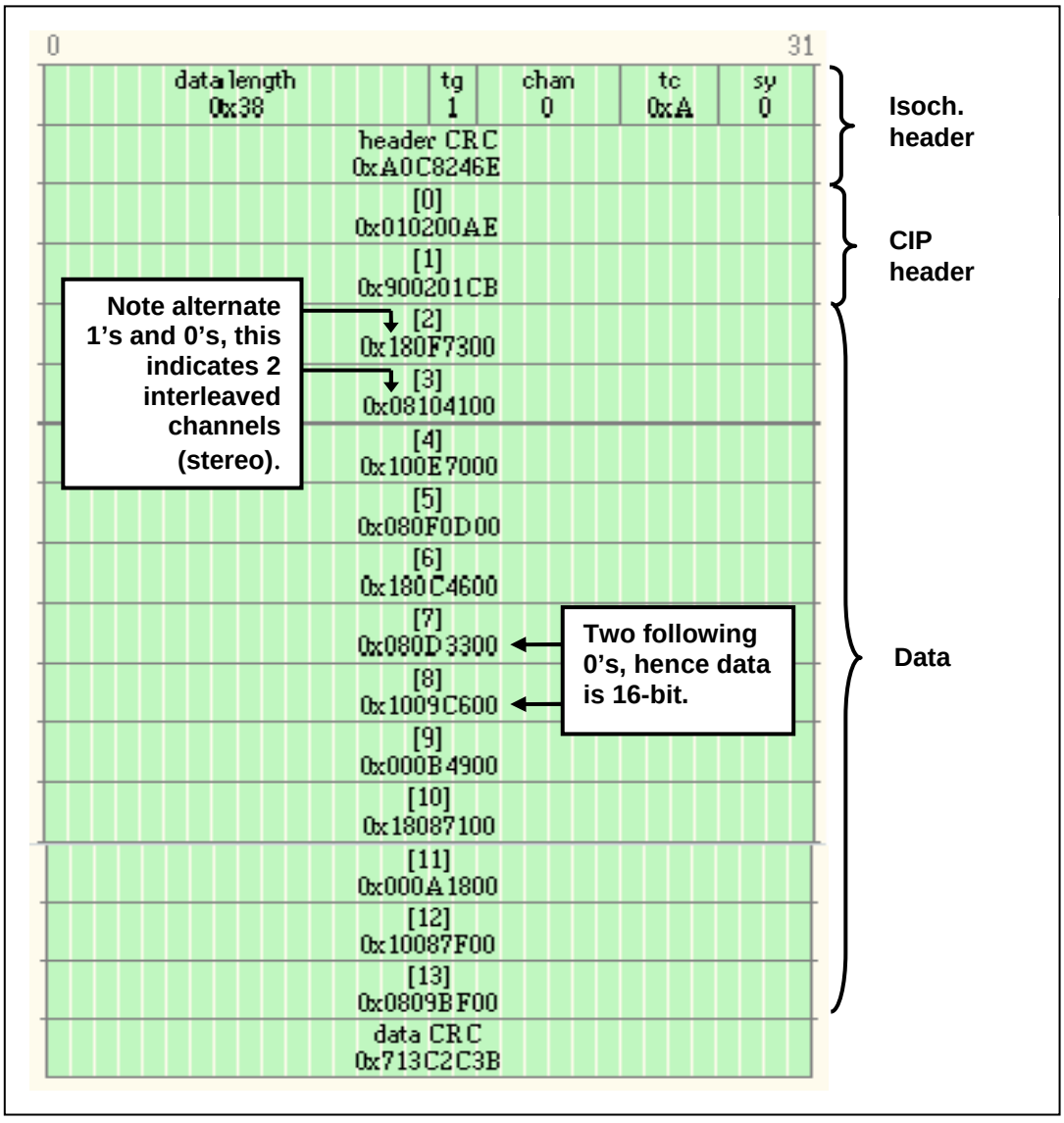

Figure 6.3: An isochronous packet containing A/M data.

The format was selected as `AMDTP_FORMAT_IEC958_PCM` (16-bit samples), the sample rate 48 kHz and the dimension set at two (stereo). The data quadlet type in this example is therefore AM824; IEC958 conformant data (format depicted in chapter three, section 3.2.2.2 - *Data Formats within Isochronous Packets*). As noted in the illustration the data conforms on all levels: having an 8-bit label field and a 24-bit data field

(although only 16-bits are used). The channels are interleaved, i.e. the samples alternate from one quadlet to the next.

Having ascertained that the devices being streamed to were in perfect working order, the implementation now needs to be considered.

6.2. INFRASTRUCTURE PROVIDED FOR IMPLEMENTATION

The implementation began by attempting to allow the A/M driver to open an interface for a sound driver to utilise. Before this is considered, a few of the important structures belonging to the A/M module and the Sound Kernel need to be elucidated. This section provides a more detailed inventory of the available infrastructure.

6.2.1. Important A/M Module Data Structures:

This section describes the main data structures crucial to the implementation of the driver. The structures are:

- `struct amdtp_host`,
- `struct stream`,
- `struct buffer`,
- `struct packet`, and
- `struct fraction`.

6.2.1.1. A/M Host Struct:

This structure defines hardware types, such as a pointer to an ohci card type field, which represents the ohci compliant card.

6.2.1.2. Isochronous Stream Struct:

The most important structure defined by the A/M driver is an isochronous `stream` data structure. It is defined in Appendix A (*A.1.1.*). However a few things need to be mentioned. The `stream` struct defines the isochronous channel number, the format, the sample rate, the dimension and the format dependent field (fdf) fields. These are all attributes of the isochronous packet discussed in previous chapters. The stream also defines various fields relating to timestamping of CIP's. It contains isochronous plug information as defined by the 61883 protocol as well. Aside from this it houses the necessary fields to address the data's actual location in the physical memory of the PC.

6.2.1.3. Sample Buffer Struct:

Another important data structure defined by the A/M module is a circular `buffer` utilised by the driver to store incoming samples from the *user space*. Its basic structure is outlined in Appendix A (*A.1.2*).

6.2.1.4. Other Structures:

Other structs include:

- A `packet` struct that defines header and payload (data or quadlets) fields in terms of the 32-bit quantity (quadlet).
- A `fraction` struct (already mentioned see chapter three, section 3.2.4 - *Synchronisation and Sample Rate Recovery*), which is utilised to determine the number of samples per packet based on sample rate [Høgsberg, 2002].

6.2.2. Allocating the OHCI for the A/M Module:

The A/M module provides two functions that add and remove the *ohci1394* low-level driver (see chapter three, section 3.3.1 - *A Simple 1394 Driver Arrangement*). This is

what provides the A/M module with a connection to the *FireWire* hardware. The two functions main tasks are registering a hardware IRQ (interrupt request) handler and unregistering the IRQ handler. Prototypes of the functions are:

```
static void amdtp_add_host(struct hpsb_host *host);
static void amdtp_remove_host(struct hpsb_host *host);
```

6.2.3. Important ALSA (snd) Structures:

Two important structures used by the ALSA *pcm* sound modules to define digital audio streams are described here are:

- `struct substream` and within that,
- `struct runtime`.

6.2.3.1. The Digital Audio Substream Structure:

The *pcm* module, or digital audio interface (sometimes also called digital audio abstraction layer) in ALSA, offers a digital audio `substream` struct shown in Appendix A (A.2.1.) with various fields defined of importance to digital audio storage. The structure includes a `stream` field that defines the direction of an audio stream; either playback or capture (record). The sound driver for *FireWire* implementation will embellish only the playback stream, since it is streaming data to the IEEE 1394 bus. The struct also defines a `runtime` field. The `runtime` field holds runtime information on the actual digital audio data, as well as hardware and software information about the digital audio data. This information includes the sample rate, the DA format and timing information for synchronisation purposes.

6.3. IMPLEMENTING THE DRIVER

The implementation of the driver began with choosing a suitable sound driver to modify and convert into an ALSA compliant driver capable of streaming audio to the *FireWire* bus. The driver that was chosen for this comes as part of the ALSA sound kernel drivers and is named the Sound dummy driver. It represents no sound-card; most drivers represent a particular card, like the SoundBlaster Live or Gravis Ultra-Sound cards. The dummy driver, like all the sound drivers in ALSA, registers a mixer for itself with which to adjust volumes of various channels[****] (however, there are no *actual* channels to adjust, since there is no card being supported!).

After the choice was made as to what sound driver to begin modifying, the A/M driver was modified to allow other modules to utilize it.

6.3.1. Modifying the A/M Module to Provide a Usable Interface

This process involved two tasks. Firstly, modifying the module's file *open* and *close* operations, and secondly the *ioctl* file operation. These are important operations that the driver carries out on the device file to which it is bound.

6.3.1.1. Opening and Closing an Isochronous Stream:

The *open* file operation of the A/M module opens an isochronous stream (as described earlier; section 6.2.1.1 - *Isochronous Stream Struct*). The *close* file operation simply does the opposite; it frees the stream. These are vital operations in terms of streaming audio. Without the isochronous stream open there is no channel of communication opened for A/M data transfer to the IEEE 1394 stack of drivers.

These two functions were re-implemented to allow another module to utilize them. However, the old file operations were retained so that the A/M module lost none of its

[****] Channel here refers to either playback or capture (record) - utilizing synthesizer, wave or PCM.

previous functionality available to applications and modules. The re-implemented versions were exported to the kernel symbol table (chapter four, section 4.3.2 - *The Kernel Symbol Table and Driver Stacking*).

At first the way in which the implementation was to occur was through having a global stream pointer within the A/M module that gets assigned in the *open file op (f_op)*. Then have a function to simply return this pointer that can be used by another module, e.g. the proposed ALSA sound module. This was scrapped in favour of a better technique. The technique settled on was implementing alternate versions of *f_ops* based on the actual *f_ops*, but which manipulated pointers to isochronous streams opened in the A/M module.

(1.a) Original amdtp_open f_op:

open (file)
register an A/M host.
if (host not available)
return error
allocate a stream to the file passed as param
if (stream not allocated)
return error
return success
end fn

(1.b) Alternate amdtp_open f_op:

open (void)
register an A/M host
if (host not available)
return error
allocate a stream to a pointer
if (stream not allocated)

return error
return pointer
end fn

The above functions are very similar, except the alternate version returns the isoch stream opened, upon success. The *close* (or *release*) *f_op* was an even simpler modification. The following describes the change:

(2.a) Original amdtp_release f_op:

close (file)
obtain stream handle on file
deallocate stream
return success
end fn

(2.b) Alternate amdtp_release f_op:

close (stream)
deallocate stream
return success
end fn

Again these are relatively similar, except that the alternate *close f_op* manipulates a stream pointer passed. The reason that the two functions use pointers to refer to isochronous streams is so that the ALSA sound driver can reference it (the implementation of which is described in section 6.3.2 - *The Sound dummy Driver*).

These changes were implemented successfully. The next function modification to consider is the *ioctl f_op*.

6.3.1.2. The ioctl File Operation:

The *ioctl f_op* for the A/M module is no different from any other device driver *ioctl f_op*. It is basically a switch statement, which uses the command parameter passed to it. The A/M module provides four possible *ioctl* commands (*cmds*): `AMDTP_IOC_CHANNEL`, `AMDTP_IOC_PLUG`, `AMDTP_IOC_PING` and `AMDTP_IOC_ZAP`. Some of the commands (viz. plug and channel) have ioctl information associated with them. The A/M modules ioctl information is stored in the `amdtp_ioctl` struct, which has a field structure, something like the following (in very pseudocode!):

1. **format** - to specify data format to write,
2. **rate** - to specify the sampling rate of data to write,
3. **dimension** - to specify stereo, etc..., and
4. **isoch connection type** - to specify 61883 defined connection type
    - ***channel***, or
    - ***plug*** [Høgsberg, 2002].

Here is a brief description of how the *ioctl* commands operate:

- The ***channel cmd*** means to configure the stream using ioctl information passed as a parameter,
- The ***plug cmd*** simply means to use the CMP (see chapter three, section 3.3 - *The IEEE 1394 Stack in Linux*) modules isochronous plug connection mechanism (IEC 61883 defined[††††]) to set up a channel (also uses ioctl information),
- The ***ping cmd*** offsets the isochronous stream's cycle count, and
- The ***zap cmd*** is a hard-reset command for the module (i.e. drop `MOD_IN_USE` to 0; see chapter four, section 4.3.1.3 - *Kernel Tracking of Modules*), meaning the module will no longer be in use.

[††††] Conceptual input and output plugs define an isochronous connection between two IEEE 1394 A/V devices.

Another *ioctl* needed to be implemented for debug purposes: the ***zip cmd***. This *ioctl* command increases `MOD_IN_USE`. This was due to weird errors occurring: the `MOD_IN_USE` variable sometimes went to -1! This essentially means that the module was in use by -1 other modules! [a bizarre situation, requiring bizarre measures].

The alternate version of this *f_op* merely uses an argument to define the *ioctl* information instead of copying it from the *user space*, using the copy_from_user function (see chapter four, section 4.3.4 - *Exchanging Information with the User Space*).

Having discussed the interfacing requirements for the A/M module, the discussion now tends towards the implementation of the sound driver proper.

6.3.2. The Sound Dummy Driver:

With the modifications described in the previous section in place, the set of events depicted in *Figure 6.4* is what should occur; that being interaction with the A/M module.

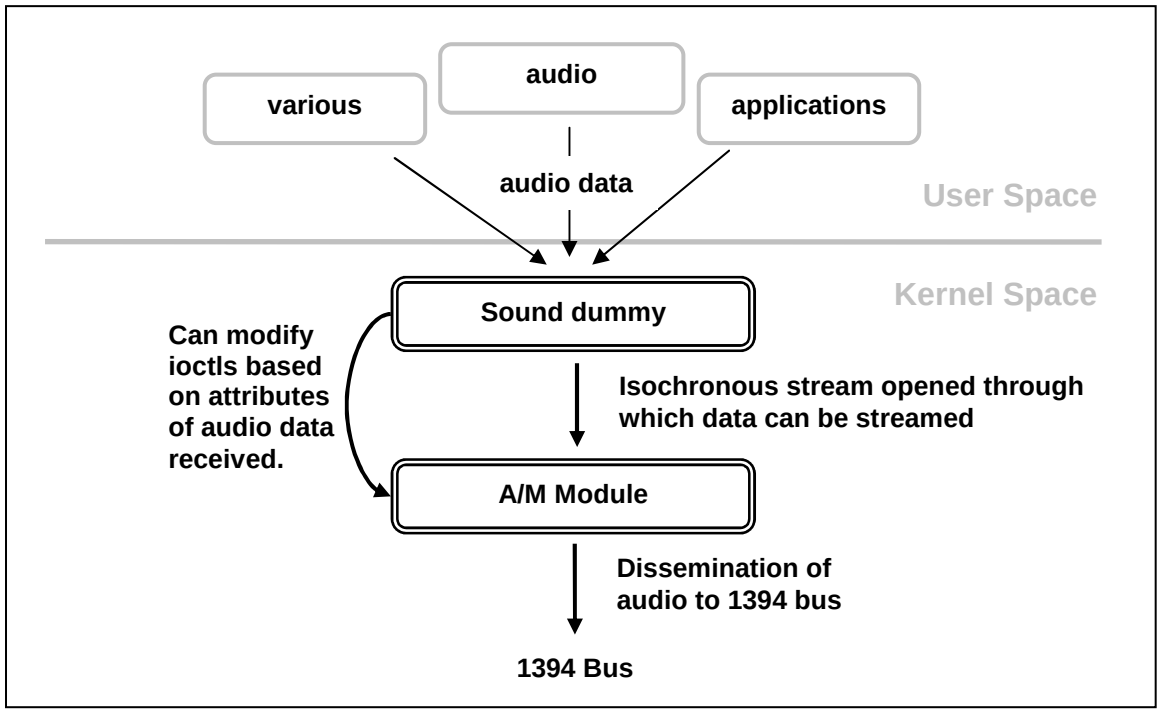

Figure 6.4: Proposed interaction between the snd-dummy module and A/M module.

The sound dummy module was furnished with a global isochronous stream pointer, since the stream struct is defined in the A/M module. This global stream is what should provide the sound dummy with a channel of communication to the IEEE 1394 bus.
All sound card drivers in ALSA, as already mentioned, support playback and capture subchannels. Underlying these subchannels are, of course, *f_ops*. The ALSA sound dummy module playback file operations were of greatest interest with regard to the implementation of this driver, since this is an audio *streaming* application[‡‡‡‡]. Those that came under scrutiny were the *open*, *close* and *ioctl f_ops*. Another issue that is discussed below is extraction of audio buffers from the sound driver.

6.3.2.1. Modifying the open, close and ioctl File Operations:

The playback *open* and *close f_ops* were modified to call their alternative A/M module counterparts, which came under discussion in the previous section. The *open f_op* assigned the global stream pointer to an isochronous stream in the A/M module. This it did by calling the alternately defined `amdtp_open` function. The *close f_op* released the stream pointer by calling the alternate `amdtp_release` function.

The playback *ioctl f_op* was also modified. The `pcm_substream` type (see section 6.2.2.1 - *The Digital Audio Substream Structure*) passed to the *ioctl* was used to obtain runtime information about the audio data. This information is stored in the `runtime` struct field of the audio substream (see Appendix A, section A.2.1 - *Digital Audio Substream Struct*).

The information was then used to disseminate *ioctl* commands to the A/M module, to allow setting up the attributes of the isochronous stream, i.e. isochronous channel number, data format, sampling rate, audio dimension, etc...

[‡‡‡‡] The word “application” used here is not in reference to the meaning normally associated with it in Computer Science.

The following pseudocode describes the transformation of information into A/M module *ioctl* commands:

```
ioctl (pcm_substream)
        evaluate pcm_substream-> audio attributes
        switch (format)
                case 1: ioctl (cmd, format_type1)
                        ...
                        ...
        switch (rate)
                 case 1: ioctl (cmd, rate_type1)
                        ...
                        ...
        switch (dimension)...
                case 1: ioctl (cmd, dimension_type1)
                        ...
                        ...
end fn
```

6.3.2.2. Extracting the PCM Runtime Data Buffer:

Each sound driver implements playback prepare and trigger operations to prepare and trigger the hardware for audio transmit and receive. This sort of functionality requires modification if the audio data is to be streamed to the A/M module, instead of the actual card driver. Several attempts were made to extract the data buffer and attach it to the isochronous stream opened. In attempts to understand the nature of the playback function hierarchy within the ALSA sound kernel the following was done:

- Audio was played on an ordinary mp3 player,
- Function calls by various sound modules were traced, and
- A map of the calls created.

This provided some insight, but was not entirely able to help reduce the problem. The following section discusses the problems encountered further.

6.4. STUMBLING BLOCKS

This section provides a description of the difficulties encountered during implementation. It also discusses how the driver could be improved and possible extensions to the Linux driver for *FireWire* project.

6.4.1. Major Obstacles:

This section is perhaps erroneously titled. There was, in retrospect, only one major obstacle, accompanied by a host of lesser problems. The obstacle was the extraction of audio data from the sound dummy module. The minor obstacles encountered were:

- Poor driver documentation for ALSA, and
- Taciturnity of the Linux 1394 development folks.

6.4.2. Improvements

For possible improvement, the driver could have been implemented by converting the A/M module into an ALSA sound module. All of the A/M protocol functionality would be contained in the sound driver. As outlined in the design and considerations chapter (see chapter five, section 5.4.2 - *Integrating the Sound Driver into Linux 1394*) this route was not explored further than its concept. This was due to the complicated nature of porting the A/M module into ALSA as well as the desire to maintain a modularized structure within ALSA.

6.4.3. Possible Extensions:

Extensions to this project would be to get the audio data buffers out of the sound dummy and into an A/M isochronous stream. Once this is working other extensions might include supporting MIDI transfers with the sound driver. This would mean having to understand a completely new representation of data in ALSA, that of MIDI. This functionality may actually be better left to another module, bearing in mind the modular structure of ALSA. Another possible extension could be providing the facility to handle multiple sequences of audio.

6.5. SUMMARY

The implementation of the sound streaming driver for *FireWire* began with testing of the devices being streamed to. The devices were the Map3 development boards. This was successful, and proved the A/M module worked satisfactorily.

Data structures belonging to both the A/M module and the PCM interface of ALSA were outlined in this chapter. The PCM substream structure was important in implementing a method to configure the isochronous stream for A/M data. This was done by passing *ioctl cmds* to the A/M module based on the attributes of the audio data. Audio attributes were gleaned from the PCM substream structure. The PCM substream carried the DA data that needed to be transferred to the IEEE 1394 bus. Attempts were made to extract the audio data buffers contained within the substream.

CHAPTER 7

CONCLUSION

FireWire is becoming the studio interconnectivity technology of the future. With its use in studio environments becoming widespread there is a need for a digital audio interface to *FireWire* within a PC's framework. Linux has a high-quality sound architecture on which to base an implementation of the interface to *FireWire*. Furthermore, there is a Linux 1394 subsystem in place to provide additional infrastructure and platform for this interface. The powerful sound architecture that provides the framework for this project is the *Advanced Linux Sound Architecture* (ALSA). This thesis introduced a novel ALSA sound driver to stream audio over *FireWire*.

The IEEE 1394 bus is the perfect audio transportation mechanism as a result of the extremely high data transfer speeds of which it is capable. The IEEE 1394 bus also has the ideal transfer mechanism for audio because of its isochronous capability. Above this there is the IEC 61883-6 standard or the A/M protocol, which defines how real-time audio data is to be transferred over *FireWire*. This protocol defined the methods of transmission of audio over *FireWire* for the ALSA sound driver being developed.

A design for the driver was arrived at after careful consideration of the Linux environment and driver architecture. In particular Linux's ALSA sound framework guided many of the decisions regarding design and implementation. The specific methods of implementing a driver in Linux dictated much of what was done. There are very rigid and standardized routines to follow when constructing Linux kernel modules that have been discussed to a certain degree of depth in this thesis. This aided in the development of the driver by providing a solid framework within which to work.

The sound driver was designed to communicate with the Linux 1394 subsystem *via* an implementation of the IEC 61883-6 standard. The 61883-6 driver provided the functionality for packetizing audio data and placing it into an isochronous stream for

delivery to the 1394 bus. Connection management between sending and listening devices is handled by a connection management protocol driver.

The sound driver for *FireWire* was able to open an isochronous stream through the A/M module. It was also able to configure the stream for A/M data, utilizing the attributes of the digital audio (DA) data contained within the sound driver. However, difficulties were encountered when attempting to extract the data and send it through the opened isochronous stream. Difficulties included:

- Firstly, discerning where the DA data was being channelled out, and then
- Obtaining a handle on the DA streams in the sound card.

At this point traces were made through the ALSA sound kernel to discern what hierarchy of function calls were being made as audio data was being played to the actual sound card. This proved to be little help, since it elucidated the calls to functions driving the sound hardware, and these were too device specific.

Future work in this area would be first to achieve actual audio streaming with the driver. Other extensions to this theme would be providing facility for handling multiple sequences of audio. An additional extension, slightly out of theme with the topic, could be handling of MIDI streaming.

REFERENCES

Andersen, S., *An overview of the GNU/Linux IEEE-1394 Subsystem*, 1999

Anderson, D., *FireWire System Architecture*, Addison-Wesley, 2nd Edition, 1998

Bartels, S., *ALSA 0.5.0. Developer Documentation*, ALSA, 1999
on-line: http://www.math.tu-berlin.de/%7Esbartels/alsa/

Høgsberg, K., *Audio and Music Data Transmission Protocol Driver*, Free Software Foundation, 2001

IEC *IEC 61883 Digital Interface for Consumer Electronic Audio/Video Equipment* International Electrotechnical Commission, 1996

IEC *IEC 61883-6 Audio and Music Data Transmission Protocol* International Electrotechnical Commission, 2000

Laubscher, R., *An Investigation into the use of IEEE 1394 for Audio and Control Data Distribution in Music Studio Environments*, M.Sc. Thesis, Rhodes University, Grahamstown, 1999

Matía, F., *Kernel Korner: Writing a Linux Driver*, Linux Journal, 1998
on-line: http://www.linuxjournal.com/article.php?sid=2476

Rubini, A., Corbet, J., *Linux Device Drivers*, O'Reilly Publishers, 2001

Sawada, Y., *Interviewed by Paul Wiffen*, 2000

Shapiro, L. *FireWire - The Consumer Electronics Connection*, ExtremeTech, 2002

BIBLIOGRAPHY

Pomerantz, O., *Linux Kernel Module Programming Guide*, The Linux Documentation Project, 1999
on-line: http://www.tldp.org/LDP/lkmpg/mpg.html

Rusling, D., *The Linux Kernel*, Version 0.8-3, The Linux Documentation Project, 1999
on-line: http://www.tldp.org/LDP/tlk/tlk.html

The 1394 Trade Association Web-Site: http://www.1394ta.org

The ALSA Home-Page: http://www.alsa-project.org

The Kernel Hackers Guide (a.k.a. KHG), The Linux Documentation Project
on-line: http://www.tldp.org/LDP/khg/HyperNews/get/khg.html

APPENDIX A

DATA STRUCTURES

A.1. A/M DRIVER DATA STRUCTURES

A.1.1. A/M Host Struct

The A/M host struct contains fields that describe the 1394 bus [Høgsberg, 2002].

```
struct amdtp_host {
      struct hpsb_host *host; /* high performance serial bus */
      struct ti_ohci *ohci; /* ohci card type */
      struct list_head stream_list; /* list of isoch streams */
      spinlock_t stream_list_lock;
      struct list_head link;
};
```

A.1.2. Isochronous Stream Struct:

The following is the definition of the isochronous stream structure appears in the A/M module [Høgsberg, 2002]. It is utilised by the A/M driver to package audio data into isochronous packets. The structure houses information DMA (*Direct Memory Access*) information regarding where the packet data is stored in physical memory. The stream contains timestamp information as well.

```
struct stream {
      int iso_channel; /* The isochronous channel */
      int format; /* The data format */
      int rate; /* The sample rate */
      int dimension; /* The dimension; stereo or mono */
      int fdf; /* Format Dependent Field */
      struct cmp_pcr *opcr;

      /* Input samples are copied here. */
      /* See section A.1.2. of this Appendix. */
```

```
        struct buffer *input;

        /* ISO Packer state */
        unsigned char dbc;
        struct packet_list *current_packet_list;
        int current_packet;
        struct fraction packet_size_fraction;

        /* We use these to generate control bits when we are
         * packing iec958 data.
         */
        int iec958_frame_count;
        int iec958_rate_code;

        /* The cycle_count and cycle_offset fields are used
         * for the synchronization timestamps (syt) in the cip
         * header.  They are incremented by at least a cycle
         * every time we put a time stamp in a packet.  As we
         * dont time stamp all packages, cycle_count isn't
         * updated in every cycle, and sometimes it's
         * incremented by 2.  Thus, we have cycle_count2,
         * which is simply incremented by one with each
         * packet, so we can compare it to the transmission
         * time written back in the dma programs.
         */
        atomic_t cycle_count, cycle_count2;
        int cycle_offset;
        struct fraction syt_fraction;
        int syt_interval;
        int stale_count;

        /* These fields control the sample output to the DMA
         * engine. The dma_packet_lists list holds packet
         * lists currently queued for dma; the head of the
         * list is currently being processed.  The last rogram
         * in a packet list generates an interrupt, which
         * removes the head from dma_packet_lists and
         * puts it back on the free list.
         */
        struct list_head dma_packet_lists;
        struct list_head free_packet_lists;
        wait_queue_head_t packet_list_wait;
        spinlock_t packet_list_lock;
        int iso_context;
        struct pci_pool *descriptor_pool, *packet_pool;

        /* Streams at a host controller are chained through
         * this field. */
        struct list_head link;
        struct amdtp_host *host;
};
```

A.1.3. Sample Buffer Struct:

This structure is also defined by the A/M module [Høgsberg, 2002]. It is utilised by the A/M driver to copy input samples from the *user space*. It is a circular buffer to hold the audio samples.

```
struct buffer {
     int head, tail, length, size;
     unsigned char data[0];
};
```

A.2. SOUND KERNEL STRUCTURES

A.2.1. Digital Audio Substream Struct:

The pcm module in the ALSA sound kernel defines a structure: the digital audio substream struct. In ALSA, as mentioned already, pcm is synonymous with digital audio. Here is the structure defined in *pcm.h* header file of the ALSA sound kernel [Kysela, Bagnara]:

```
struct _snd_pcm_substream {
     snd_pcm_t *pcm;
     snd_pcm_str_t *pstr;
     int number;
     char name[32];             /* substream name */
     int stream;                /* stream (direction) */
     size_t buffer_bytes_max;   /* limit ring buffer size */
     int dma_type;
     void *dma_area; /* DIRECT MEMORY ACCESS STUFF */
     dma_addr_t dma_addr;
     size_t dma_bytes;
     size_t dma_max;
     void *dma_private;
     /* -- hardware operations -- */
     snd_pcm_ops_t *ops;
     /* -- runtime information -- */
     snd_pcm_runtime_t *runtime; /* RUNTIME INFORMATION */
    /* -- timer section -- */
     snd_timer_t *timer;        /* timer */
     int timer_running;         /* timer is running */
     spinlock_t timer_lock;
```

```
      /* -- next substream -- */
      snd_pcm_substream_t *next;
      /* -- linked substreams -- */
      snd_pcm_substream_t *link_next;
      snd_pcm_substream_t *link_prev;
      snd_pcm_file_t *file;
      struct file *ffile;
#if defined(CONFIG_SND_PCM_OSS) ||
      defined(CONFIG_SND_PCM_OSS_MODULE)
      /* -- OSS things -- */
      snd_pcm_oss_substream_t oss;
#endif
      snd_info_entry_t *proc_root;
      snd_info_entry_t *proc_info_entry;
      snd_info_entry_t *proc_hw_params_entry;
      snd_info_entry_t *proc_sw_params_entry;
      snd_info_entry_t *proc_status_entry;
      snd_info_entry_t *proc_prealloc_entry;
};
```

APPENDIX B

GLOSSARY OF TERMS AND ABBREVIATIONS

ALSA..... The *Advanced Linux Sound Architecture*; a sound framework for Linux developed under the Gnu GPL of the FSF.

A/M Data..... Audio and music data, as defined by the A/M protocol (see below).

A/M Module..... Name given to the Linux A/M Data Transmission Protocol driver in this text.

A/M Protocol..... The Audio and Music Data Transmission Protocol - Part six of IEC standard 61883.

ADAT..... Alesis Digital Audio Transmission format - Alesis corp.'s proprietary digital audio format.

AES/EBU..... Audio Engineering Society/EBU defined digital format.

Asynchronous..... Type of serial bus data transfer where delivery is guaranteed; acknowledgement is returned on success and retry's attempted on failure.

CYCLE_TIME

Register..... This is an IEEE 1394 register that

DA..... Digital audio, synonymous with PCM in ALSA.

Devices..... Devices (*a-la-Linux*) refer to files in */dev* which represent physical peripherals, as well as virtual devices (*see Chapter 4, Section 4.2*).

FireWire..... Apple trademark of the IEEE 1394 high performance serial bus. Essentially referring to the same entity.

FSF..... *Free Software Foundation.*

Gnu GPL..... The Gnu *General Public License*; open source software license.

IEC..... *International Electrotechnical Commission*; an international standards body.

IEEE..... *Institute of Electronics and Electrical Engineers*; an international standards body.

Isochronous..... Type of serial bus data transfer where the rate of delivery is critical; there is no re-sending of packets or acknowledgement.

Isochronous..... Channel There are 64 available isochronous channels on the 1394 bus, one of which is a broadcast node.

Kernel Space..... Privileged execution mode or area of an operating system; in Linux often referred to as *supervisor mode*.

Kernel..... The central module of an operating system responsible for memory management, process and task management, and disk management.

Map3..... Stands for *mLAN* Audio Port 3 development boards. Yamaha prototype boards to interface audio, MIDI and *FireWire*.

MIDI..... Musical Instrument Digital Interface. A specification for interconnection of digital music processors (esp. keyboards, DSP's and synthesizers).

Module..... Drivers in Linux are referred to as modules, which run in the Linux *kernel space*, and extend the kernels functionality.

Mp3..... Mpeg Layer 3; compressed audio data.

OS..... Operating System. In the context of this text it refers to Linux.

OSS..... *The Open Sound System*; another set of sound drivers in Linux - it is the standard, shipped with almost all distributions.

PCM Device..... In ALSA the digital audio device on a sound card.

PCM..... Pulse code modulation, which is the sampling mechanism for representing analog signals in digital; esp. audio, in this case. In ALSA PCM is synonymous with Digital Audio.

PLL..... Phase lock loops are clock recovery circuits. Receiving devices use PLL's to generate a synchronised clock signal to a transmitter, based on some timing information encoded in the received data. E.g. Ethernet uses timing info inherent in manchester encoding; A/M protocol uses the SYT field.

Quadlet..... A 32-bit data quantity; all data packets and fields are defined in terms of them.

Sample rate..... The number of samples per second that make up a digital-audio sound recording.

Sampling..... Taking periodic snapshots of continuous phenomena; esp. sound in this case. Digital audio is produced by sampling live sound at frequent, regular intervals, and then digitizing each sample.

Stream..... A unidirectional data transmission; in this case audio.

SYT_INTERVAL The of data blocks in an isochronous payload between two valid, consecutive SYT's (time stamps).

Time stamp..... A quantized timing of events based on a reference clock. In this text all time stamps are generated with reference to CYCLE_TIME.

User Space..... The *mode* in which applications run in an operating system.

Word-Clock..... A word-clock is a synchronisation signal sent from one digital audio device to another, to align samples with one another.

APPENDIX C

NOTE TO EXAMINERS

The code that comes with this project is on the Compact Disc accompanying this thesis.

APPENDIX D

POSTER PRESENTATION

At the back of this thesis a copy of the poster for this project. It was presented on the 22 of August 2002.