Enhancement of ALSA firewire stack

Takashi Sakamoto

2014/06/26

1 Abstract

Many sound devices for IEEE 1394 bus have been produced since 2000. For the devices, ALSA, Linux Sound Subsystem, has Firewire driver stack in kernel-land. But this stack can support a few devices because it has a restriction to just transfers outgoing packets with PCM samples. This is not enough for most of devices for production because these devices require drivers to handle incoming packets for timestamp synchronization, multiplex/demultiplex different types of data to the same packet. Instead of ALSA, FFADO, a project to develop user-land driver, supports many of the device. But there is no way to bridge ALSA applications and FFADO directly, and ALSA applications cannot communicate to the devices directly. In 2014, ALSA firewire stack is extended to support 70-80 devices with new drivers without disturbing user-land drivers to solve this issue.

2 Acknowledgement

There're many persons to whom I should thank, but here I describe two developers. Clemens Ladisch is a maintainer of ALSA Firewire stack. Without his work for ALSA firewire stack and his mockup for Fireworks driver, I might have no interests for this category of devices and might be a developer for USB sound device. Daniel Wagner is a founder of former FreeBoB project. Without his help, I cannot access to BeBoB related resources and have no opportunity to understand BeBoB behaviours.

3 Introduction

Linux is an operating system widely used on various platform. Linux has a sound subsystem, Advanced Linux Sound Architecture ALSA). ALSA is utilized on several platform. On PC/AT-compatible platform, there are peoples who has enthusiasms to use Linux for music production. They require sound devices with unique functionality such like internal mixing and synchronization, and some of

them (not all of them) prefer to sound devices for IEEE1394 bus because the devices satisfies their requirement.

But ALSA, itself, has a weak stack to handle the devices. Instead of ALSA, user-land driver satisfies the users. A project to develop the driver is FFADO. FFADO gives a library and API to communicate to the devices. Although threre're much ALSA applications, there're less FFADO applications. And there's no way for ALSA applications to use FFADO directly.

I[1] realized this issue in 2009. After investigating, I concluded that enhancement of ALSA firewire stack is a better solution. This report is written to explain the conclusion and my work for this stack.

In this report, at first, I review some specifications related to the devices. Next, I describe devices' features. Then I investigate software implementation in user-land and in kernel-land to seek solution. Finally, I explain my work for ALSA firewire stack.

4 Common Specifications

The sound devices on IEEE1394 bus use IEC61883-1/6 for its communication protocol. In this section, I review IEEE1394, IEEE1212, IEC 61883-1/6 and OHCI 1394 to help understanding of devices' features. This report is not for explanations of specifications so I pick up information needed to implement software drivers.

4.1 IEEE 1394 bus

IEEE 1394 is a serial bus. IEEE published its first standard documents in 1995[1]. Between 2000 and 2006, some supplements were added[2, 3, 4]. And in 2008, it was updated[5].

IEEE1394 bus consists of Physical layer, Link layer, Transaction layer and Serial bus management. Two types of communication is available on the bus, isochronous and asynchronous.

4.1.1 Physical layer

IEEE 1394 Physical layer defines two types of environment, cable and backplane. I've never seen the backplane environment so I describe about cable environment. In cable environment, a unique cable and connectors are defined. In this environment, any bus topologies such like tree and star are available. A half duplex communication with arbitration is used. For the communication, analog signal is used for the arbitration. After the arbitration, encoded signal is used for digital data.

[1] Author

4.1.2 Link layer

IEEE 1394 Link layer defines the way to control isochronous cycle, to receive or transmit packets, to fragment a packet.

4.1.3 Transaction layer

IEEE 1394 Transaction layer defines the way to control nodes over the same bus. This layer use IEEE 1212 for addressing of nodes and the types of transaction.

4.1.4 Serial bus management

IEEE 1394 Serial bus management defines the way to manage bus topology, transmission and resources for isochronous communication. This layer works with the other layers.

4.1.5 Isochronous communication

IEEE 1394 isochronous communication guarantees to transfer packets at 8,000 times per second (8.0kHz). One cycle master broadcasts cycle start packets to start a communication, then the other nodes transfer packet with a certain channel. This channel is express within 6bit field of an isochronous packet. Receivers deal with packets identified by the channel.

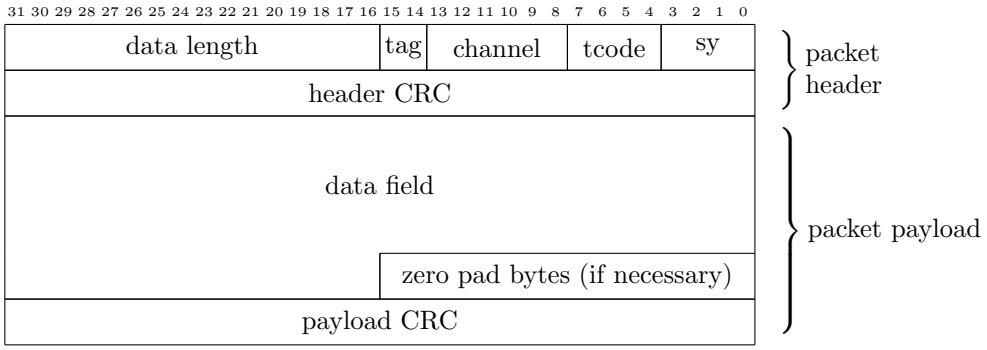

Figure 1: Isochronous packet structure

data length: The length of data field

tag: The data format of isochronous packet, usually b01 for Common Isochronous Packet(CIP)[16]

channel: The channel number for this packet

tcode: The packet format, fixed at b1010

sy: Application specific control field

4.1.6 Asynchronous communication

IEEE 1394 asynchronous communication has no guarantees for delay. The packet can be fragmented. IEEE 1394 transaction layer handles this types of communication. There're two types of transaction, quadlet and block.

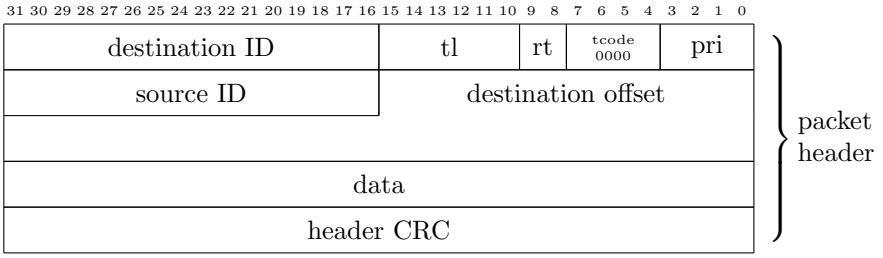

Figure 2: Asynchronous packet structure for quadlet transaction

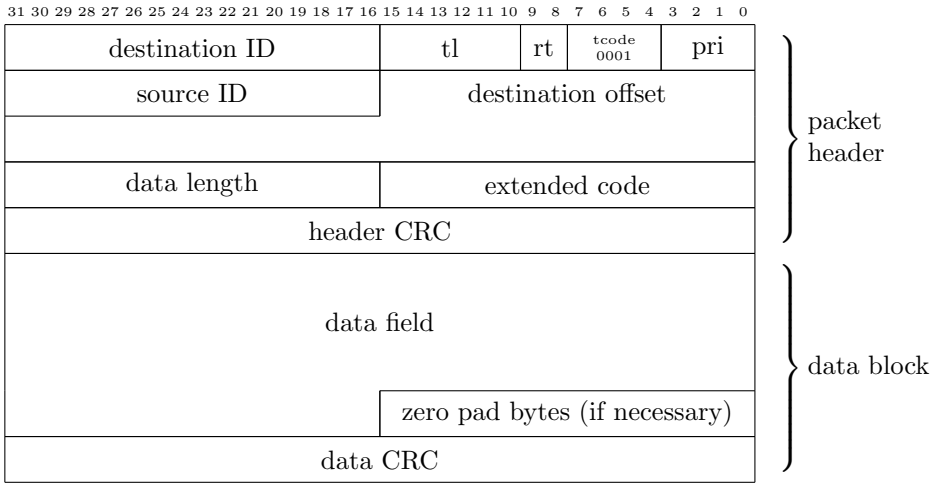

Figure 3: Asynchronous packet structure for block transaction

destination ID: Node ID of destination

tl: Transaction label for a pair of request and response

rt: The way to retry

tcode: The type of transaction, read/write/lock and quadlet/block and request/response

pri: Priority

source ID: Node ID of source

destination offset: An address offset on destination's area

data length: The length of byte in data

extended code: The type of lock transaction

4.2 IEEE 1212

IEEE 1212 is a specification for logical address space for each devices and types of transaction. The first edition was published in 1991[6] and this was standardized as ISO/IEC 13213[7]. The second edition was published in 2001[8] and supervised previous edition.

IEEE1212 defines some entities on the bus. They're module, node and unit. The module includes some nodes and the node includes some units. The node is addressed with 64-bit width value. The node can communicate each other by transactions with the address. IEEE 1212 defines three types of transaction. They're read, write and lock transactions. The address space of each node includes Control and Status Registers (CSR) to control the state of node. The address space also includes Configuration ROM to show information of the node and unit.

4.3 IEC 61883-1

IEC 61883-1 defines a transmission protocol for audio-visual data and control commands which provides for the interconnection of digital audio and video equipment. The first edition was published in 1998[14]. The second edition was published in 2003[15]. The third edition supervised previous editions in 2008[16].

IEC 61883-1 describes Real time data transmission protocol over IEEE 1394 bus, and Common Isochronous Packet (CIP) for a basic format of data in payload of IEEE1394 isochronous packet. The detail of CIP is described in next subsection.

This data transmission is controlled with an idea of 'connection'. The connections is expressed in Plug Control Register (PCR), and Master Plug Register (MPR) has basic information for PCRs. Both of MPR/PCR have 32bit register. The space for them is from 0xFFFF'F000'0900 to 0xFFFF'F000'09FC. A first half of this area is for output plugs and the rest is for input plugs. The first 32bit register of each area is for MPR.

The connection is controlled by Connection Management Procedure (CMP). In CMP, some fields in PCR can be changed by lock transaction to establish/break connections.

The functionality on each node can be controlled by Function Control Protocol (FCP). FCP consists of a pair of transaction for command and response. The address for command is 0xFFFF'F000'0B00 and the address for response is 0xFFFF'F000'D00. The length of data for each transaction is limited by 512 bytes.

IEC 61883 series, itself, came from documents which 1394 Trade Association (1394TA) published. Especially, actual application of FCP is not in public documents and it's in 1394TA documents. The application is called as 'AV/C commands'. There're much variations of AV/C command. A part of documents are applied to the sound devices, including over-specifications[21, 22, 23, 24, 25, 27, 28, 29].

4.4 IEC 61883-6

IEC 61883-6 is a variation of IEC 61883-1 for audio and music data. The first edition was published in 2002[17]. The second edition was published in 2005 with many extensions[18].

IEC 61883-1[14, 15, 16] allows any variations for CIP structure. The CIP header consists of some 32bit fields. The first bit in MSB means End-of-CIP-header (EOH). This bit stands when the 32bit fields is the last CIP header. The second bit in MSB is Form. This bit means the format of followed fields but the specification includes ambiguity of its meaning. AMDTP uses two quadlets CIP header with SYT field. I show CIP structure for AMDTP in Figure 4. Minor fields are with zero.

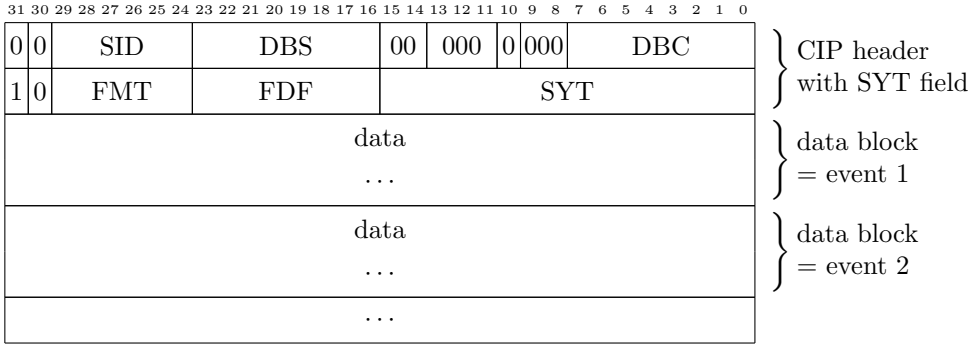

Figure 4: CIP variation for AMDTP

SID: The node ID for transmitter

DBS: The number of quadlets for data blocks

DBC: The number of data blocks which has already transferred. This is to detect discontinuity.

FMT: Format ID. In AMDTP, 0x10.

FDF: Format dependent field. In AMDTP, This field includes 'Event Type' field and 'Sampling Frequency Code' field, described later. Especially, 0xFF means the packet includes no data.

SYT: The offset from isochronous cycle timer. In AMDTP, used to show 'presentation time'. In detail, Section 5.1 to page 8.

After CIP headers, some data blocks follows. In AMDTP, one data block represents a event. IEC 61883-6[17, 18] doesn't define meaning of the event, although the event may be a batch of data accompanied with the same timing. There're some ways to describe the data and AM824 is mostly used.

AM824 represents data by 32bit field that has an 8bit label and 24bit data field[18]. In IEC 61883-6:2002[17], AM824 can describe three types of data, IEC 60958 Conformant, Raw Audio and MIDI Conformant. In IEC 61883-6:2005[18], Raw Audio is renamed as Multi Bit Linear Audio and some data types are newly supported.

As described before, the event is accompanied with timing because the event is real time data for audio and music. As a result, AMDTP is dominated by sampling frequency. The sampling frequency and the type of event is shown in FDF field of CIP header. The 4bit in MSB is EVT field to show the way to describe the data, and the lower 3bit of 4bit in LSB is SFC to show nominal sampling frequency. The rest 1bit is N-flag. This flag is used to show the mode of rate control. When this flag does not stand, it means clock based rate control, described in Section 5.1 to page8. Else, it means command based rate control[18, 29]. For AM824, the value of EVT field is b00. The values of SFC field is shown at Table 1. For most of sound devices, N-flag doesn't stand.

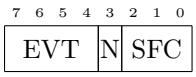

Figure 5: FDF field for AMDTP

Table 1: The value of SFC field in FDF field

Nominal Sampling Frequency	SFC
32.0	b000
44.1	b001
48.0	b010
88.2	b011
96.0	b100
176.4	b101
192.0	b110

For detail to transfer the packet, see Section 5.1 to page 8.

4.5 1394 Open Host Controller Interface

1394 Open Host Controller Interface (OHCI 1394) is an implementation of the link layer protocol of IEEE 1394, with additional features to support the trans-

action and bus management layers. This specification uses Direct Media Access (DMA) engines for high-performance data transfer and a host bus interface. The first edition was published in 1997[19] and revised in 2000[20].

OHCI 1394 is a document for host controller implementation for IEEE 1394 bus. The implementation includes link and transaction layer. OHCI 1394 describes software interface with register, interrupt and Direct Media Access (DMA). For PC/AT-compatible platform, OHCI 1394 is implemented as card devices on PCI/PCI-Express bus.

OHCI 1394 represents both of asynchronous and isochronous communication as context, which includes a list of descriptor. A descriptor includes information for a packet of each communications. The information includes an address for DMA buffer. The data for the buffer is transferred between system and controller device by DMA. After transferring, interrupts are generated according to information in the descriptor. The minimum number of isochronous context is 4 and the maximum is 32, for a direction.

5 Device features

The sound devices for production have unique features which usual sound devices don't have. In this section, I describe such features to consider software implementation.

5.1 Clock recovery

Clock recovery is a mechanism to regenerate transmitter's clock cycle in receivers' side.

In IEEE 1394[5], isochronous-capable nodes should have 32bit CYCLE_TIMER register. The low-order 12 bits of the register are a modulo 3,072 counter, which increments once every 24.576MHz (or 40.69 nano second). The next 13 higher-order bits are a count of 8.0kHz (or 125us) cycles, and the highest 7 bits count seconds.

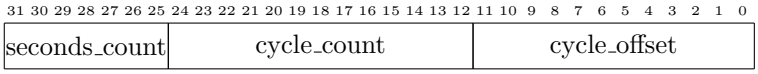

Figure 6: CYCLE_TIMER register

In usual implementation, the register count up according to 24.576MHz clock given by IEEE 1394 Link layer chipset. Furthermore, OHCI 1394 controller devices have some CYCLE_TIMER registers for each isochronous context.

When the cycle_offset field rolls over on IEEE 1394 Link layer chipset, cycle master node broadcasts a cycle-start packet after arbitration. A cycle_time field in this packet includes the value of CYCLE_TIMER register. After receiving cycle-start packet, transmitters and receivers update their CYCLE_TIMER register with the value. As a result, the transmitters and receivers can maintain local time reference to cycle master. Then transmitters start arbitration and

transmit an isochronous packet with a certain channel and the receivers receive isochronous packet with a favorite channel.

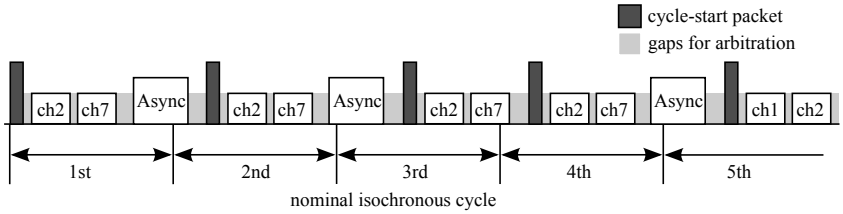

Figure 7: cycle-start packet and nominal isochronous cycle

IEC 61883-1 defines SYT field in CIP header and IEC 61883-6 uses this field to calculate 'presentation time'. The 'presentation time' represents the timestamp for a certain event in a packet. The transmitter calculates offset from local isochronous cycle reference as the timing of sampling clock, then transfers the value in SYT field of a packet. The receivers calculate 'presentation time' with the value of SYT field in the packet and local isochronous cycle reference.

One packet has one SYT field in its CIP header, while one packet can include several events. So all of events cannot have 'presentation time'. The interval between events corresponding to 'presentation time' is defined as SYT_INTERVAL.

Table 2: The value of SYT_INTERVAL

Sample transmission frequency	SYT_INTERVAL
32.0	8
44.1	8
48.0	8
88.2	16
96.0	16
176.4	32
192.0	32

If there's a packet without 'presentation time', the value of SYT is 'No Information' code (0xffff). The way to decide which events corresponding to 'presentation time' in a packet is different depending on transferring method.

In IEC 61883-6[17, 18], there are two methods for transferring packets. They're blocking and non-blocking. The differences between these two methods are transfer delay and the number of events which a packet can includes.

In blocking mode, the number of event in an packet is fixed to the value of SYT_INTERVAL and 'presentation time' is for the first event in a packet. A packet is wait to be transferred until all events in the packet are sampled, then the transfer delay is larger than default value by one packet.

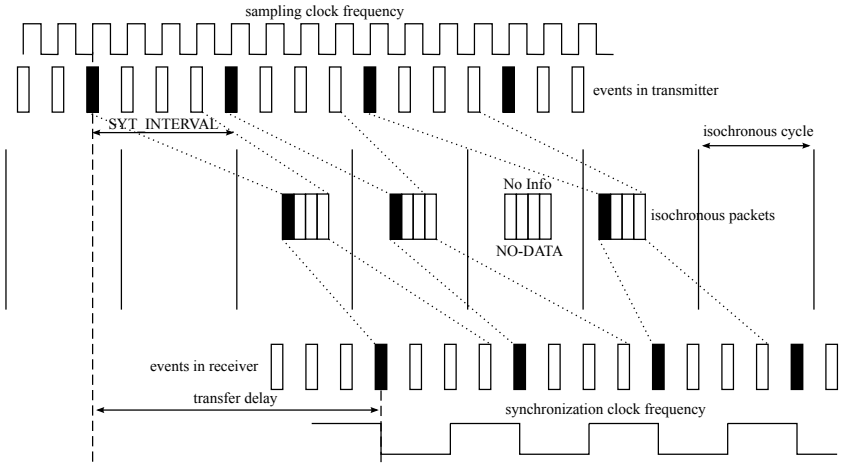

Figure 8: AMDTP in blocking mode

In non-blocking mode, a different way is used because a packet includes different numbers of events. So the transmitter can transfer packets according to nominal isochronous cycle without waiting. The transfer delay equals to default value and is less than the value in blocking mode. But it's more complicated to decide which event is accompanying 'presentation timestamp' in a packet.

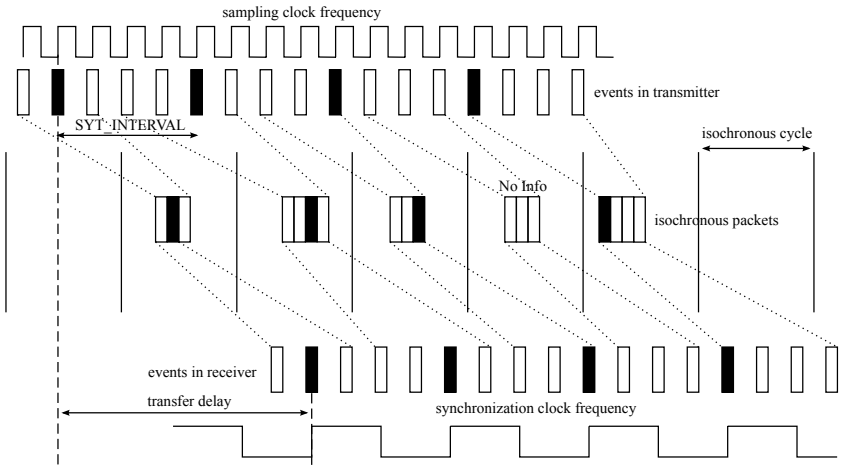

Figure 9: AMDTP in non-blocking mode

A transmitter decide an event corresponding to 'presentation time' in this condition:

$$mod(DBC, SYT\_INTERVAL) = 0 \tag{1}$$

Then receivers calculates the index of events in a packet according to this formula:

$$index = mod((SYT\_INTERVAL - mod(DBC, SYT\_INTERVAL)), SYT\_INTERVAL) \quad (2)$$

This is a mechanism of 'presentation timestamp'.

The receivers can generate 'synchronization clock frequency' to use local time reference maintained by cycle_start packet in each isochronous cycle and the value of SYT field in a packet, according to this formula.

$$Fsync = STF/SYT\_INTERVAL < 8,000 \quad (3)$$

Here Fsync is the synchronization clock frequency. STF is the sampling transmission frequency, equal to sampling clock frequency.

Table 3: Sampling transmission frequency

STF	SYT_INTERVAL	Fsync
32.0	8	4,000
44.1	8	5,512
48.0	8	6,000
88.2	16	5,512
96.0	16	6,000
176.4	32	5,512
192.0	32	6,000

The receivers can use this pulse for PLL reference. This is a mechanism of 'clock recovery'. For an example of actual implementation, see [30].

5.2 Signal processing for internal mixing

Most devices for production use have several ports for analog/digital audio input/output. Such devices have DSP to process several signals from/to the ports for internal mixing. This is 'zero latency hardware monitoring', which each vendor insists.

In the DSP side, outgoing packet is a sink of audio signals and incoming packet is a source of audio signals. The DSP is between IEC 61883-1/6 layer and DAC/ADC for audio ports.

The way to control the DSP behaviour depends on each chipset or model.

5.3 Some options for sampling clock source for DSP

Most devices for production use have several options for source of clock. The source is used for input clock of DSP. The options are:

- Signal from internal clock source

- Signal from input of S/PDIF, ADAT and Word Clock
- Signal with nominal isochronous cycle (8kHz)
- Signal with recovered clock as described in previous subsection (4.000/5,512/6,000Hz)

The way to change or check current source of clock depends on each chipset or mode. When non-internal source is selected, sampling rate should be fixed according to the signal from external devices.

5.4 Duplex streams with timestamp synchronization

As described in former subsection, each AMDTP packet delivers 'presentation time', in the sequence of value in SYT field. When devices support both of transmiting/receiving streams, ideally, the generator of the timestamp should be one, for better synchronization between transmitter and receiver. This is a requirement to pass the value of SYT field in a packets to a packets in an opposite direction.

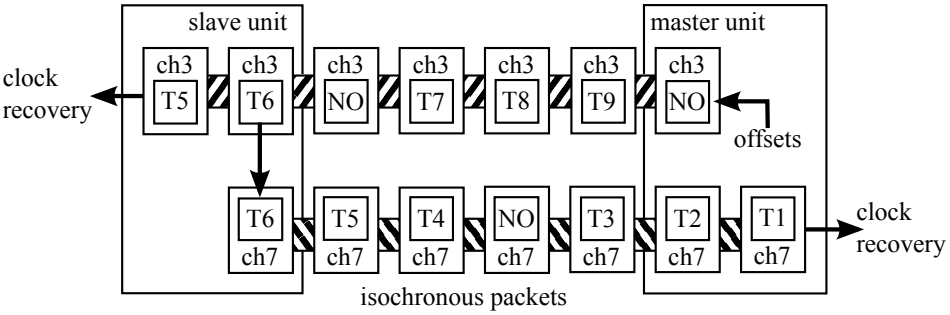

Figure 10: Duplex streams, timing master and slave

Moreover, in previous subsection, the sequence of value in SYT field can generate a pulse and the pulse can be selected as source of clock. This option means that a device can delegate 'timing master' to the generator.

5.5 Different formation of data channel in a data block of packet

In IEC 61883-6[17, 18], a data block of packet can include several types of data channel. Each data is transferred by each data channel in packets. For example, Multi Bit Linear Audio (MBLA) data channel for PCM samples, MIDI Conformant data channel for MIDI messages and IEC 60958 Conformant data channel for IEC 60958 PCM samples.

In IEC 61883-6:2005[18], the position of each data channel is defined:

```
IEC 60958 Conformant Data < Multi-bit Linear Audio <
MIDI Conformant Data < SMPTE Time Code < Sample Count
```

Each model has its own formation of data channels, depending on sampling frequency or a certain modes, even if these models apply the same chipset. The way to retrieve information about the formation depends on each chipset.

5.6 Chipset specific operations and quirks

Although there are some specifications, each device has specific quirks due to chipset or vendor's customization. Especially, the way to control DSP is largely different. Currently quirks of four chipsets are clear.

5.6.1 Fireworks

Fireworks is a board module which Echo Audio produced. This module consists of three chipsets:

- Communication chipset for IEEE1394 PHY/Link and IEC 61883-1/6 (Texas Instruments iceLynx Micro)
- DSP or/and FPGA for signal processing
- Flash Memory to store firmwares

Fireworks uses its own mechanism for transaction. Drivers transmit commands to a specific address on device, and the device transmit responses to a specific address on controller. Therefore Fireworks has unique command sets. Although this transaction is transferred to specific addresses, there's another way to transfer commands and responses as AV/C vendor specific commands.

Fireworks manages streams by CMP and transfers AMDTP packets in blocking mode. But Fireworks streams are not fully compliant to IEC 61883-1. In IEC 61883-1, the value of data block counter means the number of packets which has already transferred. But in Fireworks, it means the number of data blocks which has transferred including current packet. Furthermore, its empty packets are transferred at tag b00 in isochronous packet.

5.6.2 BeBoB

BeBoB is 'BridgeCo enhanced Breakout Box'. This is installed to firewire devices with DM1000/DM1100/DM1500 chipset. It gives common ways for host system to handle BeBoB based devices.

BeBoB uses CMP to maintain AMDTP streams but some BeBoB based devices need both connections to transmit AMDTP stream. BeBoB transfers AMDTP packets in blocking mode.

BeBoB supports AV/C General command [21], AV/C Audio Subunit command [22], AV/C Music Subunit command [23] and AV/C Descriptor Mechanism [26]. Furthermore, BeBoB uses its own extension of AV/C commands [31, 32]. But BeBoB can be customized largely so there're some model specific operations so these commands are not always used.

5.6.3 OXFW970/971

OXFW970/971 was produced by Oxford Semiconductor.

OXFW970/971 uses CMP to maintain AMDTP streams and transfers AMDTP packet in non-blocking mode.

OXFW970/971 support AV/C general command [21] and AV/C Audio subunit command [22].

5.6.4 Dice

Dice is Digital Interface Communications Engine which TC Applied Technologies produces.

Dice doesn't use CMP or any AV/C commands, although uses read/write transactions to specific register for this purpose. Dice has unique mechanism to notify status change. This notification is delivered to a specific address on controller and drivers can indicate the address.

Dice transfers AMDTP packets in blocking mode. But Dice is not fully compliant to IEC 61883-6. At higher sampling rate, Dice transfers AMDTP packets with double count of data blocks at a half of rate than IEC 61883-6. For example, in IEC 61883-6, at 192.0kHz in blocking mode, an AMDTP packet includes 32 data blocks. But Dice transfers AMDTP packets with 64 data blocks at 96.0kHz. This is called as 'dual wire'.

6 Existing drivers

In previous section, I describe device features and chipset quirks. In this section, let's see current implementation for them.

6.1 FFADO

FFADO is a project to develop user-land drivers for firewire sound devices. FreeBoB is a ancestral project of FFADO and Daniel Wagner started this project to develop driver for BridgeCo Enhanced Breakout Box (BeBoB) based devices in 2004. Daniel was in BridgeCo AG and had a contract to disclose relevant resources to the other developers. Later, Peter Palmer joined, The code base of FFADO grew up by them. In 2006, Jonathan Woithe joined in this project. He worked for drivers for MOTU devices. In 2007, to support more devices, FFADO project began.

As of 2014, FFADO supports:

- DM1000/1100/DM1500 based devices with BeBoB
- Echo Audio Fireworks based devices
- Oxford Semiconductor OXFW970/971 based devices
- Stanton SC System 1 (PCM only)

- TC Applied Technologies Digital Interface Communications Engine (Dice) based devices
- Some devices which Mark of the Unicorn (MOTU) produces
- Some devices which RME produces

FFADO gives a library, libffado. This library uses 'libraw1394' to execute I/Os to Linux Firewire stack, and use 'libiec61883' to packetize and manage connections. This library also handles model specific operations and AV/C commands including vendor dependent commands. FFADO applications can transfer packets from/to devices, and control the devices.

FFADO also gives a daemon, 'ffado-dbus-server' as a FFADO application. As the name shows, this daemon uses D-Bus interfaces to give a way for the other applications to control device's internal mixer. Currently there're no applications to use this interface than ffado-mixer, which gives GUI with Qt4.

Currently, there are no FFADO applications except for 'ffado-dbus-server' and JACK [2] server daemon (jackd). The jackd uses libffado to packetize and transfer packets to/from devices. The jackd uses UNIX domain socket and System V Shared Memory to gather/distribute data to/from JACK application processes.

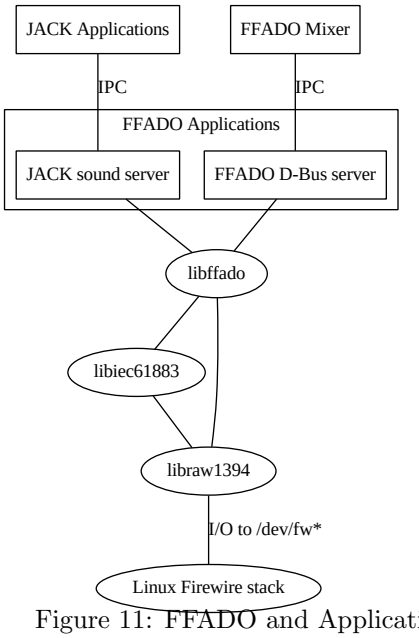

Figure 11: FFADO and Applications

[2] http://jackaudio.org/

6.2 ALSA Firewire stack

There're no actions till 2010 to support firewire sound devices. An ALSA developer, Clemens Ladisch, started his work for ALSA firewire stack in 2010. In 2011, some stuffs were merged to ALSA upstream. This is a beginning of ALSA Firewire stack. In this time, 'snd-firewire-speakers' was committed for two models of OXFW970/971 based devices. In 2012, 'snd-scs1x' was committed as MIDI only driver for Stanton SC System 1[3]. In 2013, 'snd-dice' was committed as playback only driver for Dice based devices.

As of 2013, ALSA firewire stack generally supports PCM playback only, while FFADO supports more functionality. If there're some ways for ALSA applications to use FFADO, or there're some ways in user-land for ALSA applications to access the sound devices, it doesn't matter that ALSA firewire stack is so cheap. In next section, I investigate ALSA implementation in user-land to seek a solution for this issue.

7 Investigation of user-land driver for ALSA

In IEC 61883-6[17, 18], a packet can include several types of data, such like PCM samples and MIDI messages. Additionally, 'presentation time' processing needs a stream from a timestamp generator. In these reasons, when several ALSA applications run for playback and capture PCM samples/MIDI messages, these applications require Interprocess Communication (IPC) between packetization process and application processes.

There're a few possibilities to implement the requirement in user-land implementation of ALSA. They're ALSA PCM plugins, ALSA External PCM plugins and CUSE application. In this section, at first, I describe ALSA implementation in user-land. Next, I investigate the three possibilities.

7.1 ALSA implementation in user-land

Modern CPUs have several modes to restrict software's access to hardware resources. In Linux, two modes are used. One is user-mode and another is kernel-mode. In user-mode, software is forbidden to access to hardware resources. Software executes system-call to switch into kernel-mode and access to hardware resources. After, the process returns to user-mode.

The resources of sound devices are also hardware resources so ALSA applications should execute system-call to communicate to them. In ALSA, the entry points for system-call is character devices and the system-call is file operation to them. In ALSA, most of file operations are ioctl(2) with unique requests. I think ALSA developers requires more specific operations than simple file operations such like read(2)/write(2).

In this reason, ALSA application developers should always consider the combination of ioctl(2) and its requests to handle sound devices. Therefore, it's

[3]This model has its own way to transmit/receive MIDI messages.

natural to use ALSA library to give Application Programing Interface (API) as abstract layer for ALSA specific file operations.

ALSA library consists of several interfaces. Each interface is for each character device. These character devices are under /dev/snd.

Mixer/hcontrol/control interface
: interface for control character device (/dev/snd/controlC%i)

PCM interface
: interface for PCM character device (/dev/snd/pcmC%iD%ip{c,p})

hwdep interface
: interface for hardware dependent character device (/dev/snd/hwdepC%iD%i)

RawMidi interface
: interface for midi character device (/dev/snd/midiC%iD%i)

Sequencer interface
: interface to deliver the MIDI-like events between clients/ports via sequencer character device (/dev/snd/seq)

Timer interface
: interface to use internal timers in hardware/software interrupt timer via timer character device (/dev/snd/timer)

In this list, the format after 'C' shows the number of card and the format after 'D' shows the number of device.

In this section, I describe PCM interface only. ALSA PCM interface is used to transmit/receive PCM samples between application and device. The unit of transfer is called 'frame'. It means that PCM samples with the same timing so the number of PCM samples per frame equals to the number of channels in the PCM substream. Most of functions returns the number of frames, instead of the number of bytes.

The basic design of ALSA PCM interface is based on handle. When starting operations, ALSA application open the handle with PCM nodes. The PCM nodes are added to ALSA runtime by ALSA PCM plugins, to extends functionality and features of PCM node. ALSA runtime configuration has hierarchical relationships between master and slave. ALSA applications access to master, then plugin works and transfer PCM data between master buffer and slave buffer. Finally, data reaches hw PCM plugin and directly communicate to kernel-land drivers. This mechanism allows PCM runtime to have 'plugin-chain' according to its configuration.

The hw plugin executes file operations to PCM character devices. In most case, it execute mmap(2) several times to share some buffers between ALSA driver. Then it executes ioctl(2) to the character device, and drives a sound device.

When ALSA applications call open API of each ALSA interface, this function read configuration files and parse it for runtime. In ALSA default, /usr/share/alsa/alsa.conf

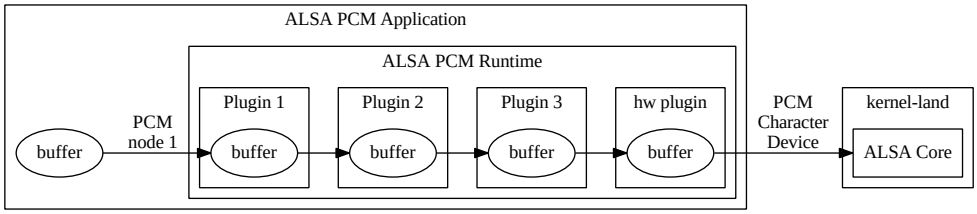

Figure 12: ALSA PCM plugin-chain

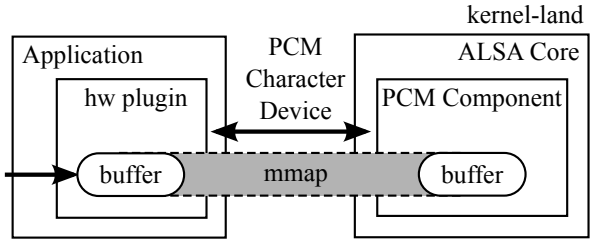

Figure 13: ALSA hw PCM plugin

is loaded at first. This file also loads /usr/share/alsa/alsa.conf.d/*, /etc/asound.conf and /.asoundrc. As a result, for PCM runtime, several PCM nodes are available, for example, hw, plughw(plug plugin with hw slave), dmix, dsnoop and default.

```
$ aplay -L
default:CARD=PCH
    HDA Intel PCH, CX20590 Analog
    Default Audio Device
...
dmix:CARD=Intel,DEV=0
    HDA Intel, ALC889 Analog
    Direct sample mixing device
dsnoop:CARD=Intel,DEV=0
    HDA Intel, ALC889 Analog
    Direct sample snooping device
hw:CARD=Intel,DEV=0
    HDA Intel, ALC889 Analog
    Direct hardware device without any conversions
plughw:CARD=Intel,DEV=0
    HDA Intel, ALC889 Analog
    Hardware device with all software conversions
```

7.2 ALSA PCM plugins

The dmix/dsnoop is a good example for ALSA PCM plugin which utilize IPC mechanism. The dmix plugin multiplexes PCM samples from several applications and transfer the PCM samples to device. And the dsnoop plugin delivers PCM samples from device to several applications. The dmix/dsnoop plugins use System V Shared Memory for IPC, System V Semaphore for mutual exclusive. ALSA PCM direct layer gives helper functions to these plugins.

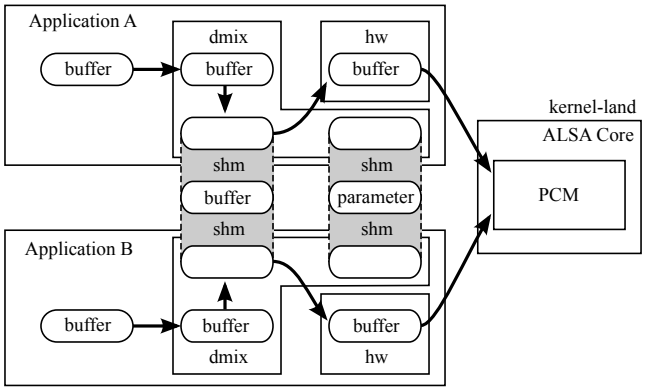

Figure 14: ALSA dmix PCM plugin

When applications open dmix/dsnoop PCM node by calling snd_pcm_open(), the dmix/dsnoop plugins keep or connect System V Shared Memory and System V Semaphore. Then the shared memory is attached to the virtual memory space of each application process. This shared memory is used to share parameters and status of PCM stream for slave. The slave of dmix/dsnoop plugin is fixed to hw plugin. Additionally, dmix plugin keep one shared memory for multiplexing buffer, too.

When calling 'snd_pcm_writei()/snd_pcm_mmap_commit()', the dmix plugin multiplexes PCM samples in application buffer to PCM samples in shared memory, then write into slave buffer. The PCM samples stored to shared memory and used by the other applications for multiplexing. The shared memory is used as 'cache'. This allows any processes to multiplex PCM samples according to their timing and write the sample to buffer for slave, while these processes don't move a buffer pointer for application at slave (hw) by usual way.

The dsnoop PCM plugin is simpler than dmix PCM plugin because there's no need to multiplex PCM samples.

There's two problems of ALSA PCM plugins for my purpose. The framework for ALSA PCM plugins don't allow master without slave. The basic idea of ALSA PCM plugins is to add features hw PCM node. This node is for communication to drivers in kernel-land. This is against my purpose in this section. Additionally, there's an issue that one of processes must do packetization. If an ALSA application perform this role, the application cannot stop as long as

the other applications use the PCM node. Therefore, I realized that a daemon process is required to packetize for ALSA applications.

7.3 ALSA External PCM plugins

ALSA has another way to extend PCM nodes in runtime. It's External PCM Plugin. There're two usages of this, Filter type and I/O type.

Although Filter type plugin is similar to PCM plugins with master/slave, I/O type plugin works as the input or output terminal point. So I/O type is good for my purpose. A good example of I/O type plugin is pulse PCM External plugin for PulseAudio.

PulseAudio[4] is a project for sound server daemon. In modern desktop environment, PulseAudio adds ALSA runtime configuration[5] to overwrite a default PCM node as pulse PCM node. As a result, most of applications in the desktop environment do IPC to PulseAudio via pulse External PCM plugin, then PulseAudio multiplexes/demultiplexes PCM samples for each ALSA applications. Finally, PulseAudio transfer PCM samples from/to each devices.

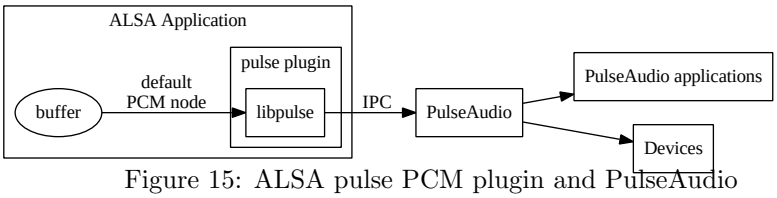

Figure 15: ALSA pulse PCM plugin and PulseAudio

PulseAudio implements several network protocols such like RTP-related protocols, or utilizes communication backends such like Rygel[6]. Furthermore, PulseAudio has its own network protocol to communicate to PulseAudio in the other hosts. This means that PulseAudio has a strong framework to use several types of devices.

There is another example. It's freebob External PCM Plugin for FreeBoB. The source of this plugin is 'alsa-plugin' directory in the top of FFADO subversion trunk[7].

[4] http://www.pulseaudio.org

[5] /usr/share/alsa/pulse-alsa.conf

[6] Rygel is UPnP Device Architecture and Device Control Protocol application for Media Renderer/Server

[7] http://subversion.ffado.org/browser/trunk

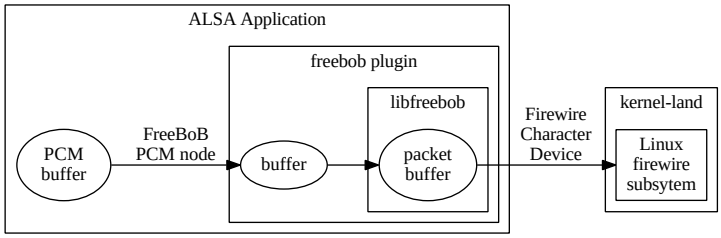

Figure 16: ALSA FreeBoB PCM plugin and Linux firewire subsystem

This plugin creates a thread when starting. This thread polls with libfreebob wait function, and transfers PCM samples/MIDI messages when waking up. There is no mechanism to check avail frames in ALSA PCM buffer and this plugin has a restriction that just one application can transfer streams so either playback or capture is available. This plugin seems to be under working and suspended but shows good perspective for possibility of ALSA/FFADO communication for ALSA applications.

The FreeBoB PCM plugin reinforces the requirement of daemon processes to packetize. And in modern desktop environment, PulseAudio is a candidate for such daemon. If PulseAudio has firewire stack, ALSA applications can transfer data between firewire devices via default PCM node. For MIDI functionality, PulseAudio should use ALSA sequencer interface to gather/deliver MIDI message for ALSA applications, as FreeBoB plugin does.

But there's already a similar PCM External I/O plugin. It's jack PCM plugin. This plugin adds jack PCM node to ALSA runtime and make ALSA applications as JACK client. This way requires users to write ALSA configuration but need no other works.

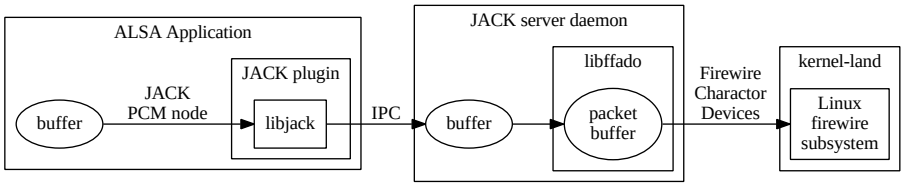

Figure 17: ALSA JACK PCM plugin and JACK server daemon

But there is an disadvantage of this. When adding the configuration for jack plugin, the node always exists in ALSA PCM runtime even if there're no processes for JACK server daemon. This is not good at transparency for users.

7.4 CUSE - Character device in user-space

Linux Filesystem subsystem has CUSE as FUSE extension. CUSE allows to implement drivers for character devices in user-land.

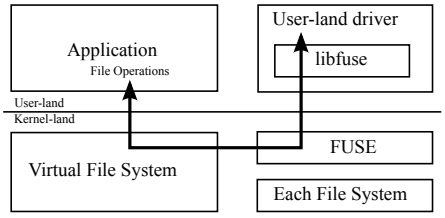

Figure 18: FUSE/CUSE and user-land driver

Open Sound System Proxy Daemon (osspd) is a good example. The osspd is an application of CUSE. When Open Sound System applications access to /dev/{dsp, adsp, mixer}, osspd handles these file operations in user-space with CUSE help, then execute IPC between PulseAudio or ALSA.

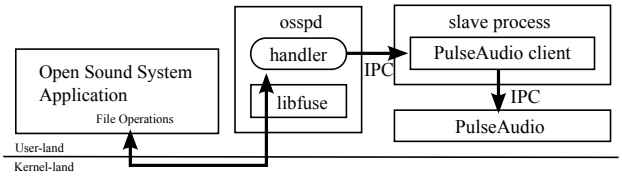

Figure 19: Open Sound System Proxy Daemon

This daemon register Open Sound System devices via FUSE library API. Then udev daemon adds Open Sound System character devices (/dev/dsp, /dev/mixer, /dev/adsp) in system according to rules which osspd adds. The daemon opens a pair of socket for IPC between daemon and slave, then creates a new process for the slave. The slave sleeps to wait IPC from the daemon, then do IPC between PulseAudio or ALSA.

For my purpose, this way is nice because there's no need to touch ALSA user-land implementation or ALSA runtime. But equivalent codes to whole ALSA Core should be implemented in the daemon to handle application's file operations. This needs much works. In this reason, the combination of 'light' ALSA driver in kernel-land and ALSA PCM component in user-land is better instead of implement alternatives of whole ALSA Core in user-land.

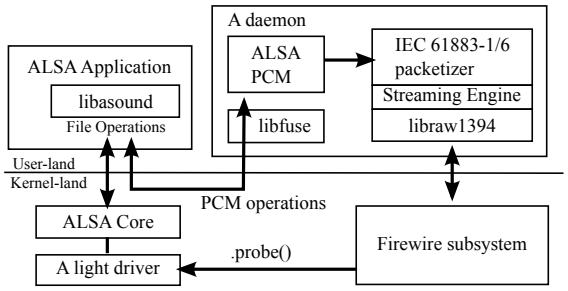

Figure 20: User-land driver for ALSA and Firewire with CUSE

When device is connected, the driver registers ALSA character devices except for PCM. The system detects new ALSA devices for firewire sound devices, then run a daemon. The daemon calls FUSE library to add ALSA PCM character device with appropriate major/minor number. Then new PCM device is available in ALSA runtime. This idea can reduce the amount of codes in the daemon but still needs much works to implement PCM component in the daemon.

In this section, I investigate some software implements in user-land to seek better ways for ALSA applications and firewire driver. As a result, I can conclude that the best way is to implement the drivers in PulseAudio, and realize the work is external to ALSA user-land. In the next section, I investigate ALSA kernel-land to seek another way for my purpose.

8 Investigation of kernel-land driver for ALSA

In this section, I investigate ALSA implementation in kernel-land.

Generally, after executing system-call, a process is running in kernel-mode and can access to hardware resources. If the data from hardware did not arrive yet, the process requests process-scheduler to generate context-switch and sleeps. Then the another process gain CPU and continue to run. Before sleeping, the process adds an event into wait-queue for waking-up. In this case, the event is the data has come. Every time when requested to generate context-switch, process-scheduler checks events in wait-queue and select a process with enough event. This is a part of process-scheduling.

When ALSA application executes system-call, the process in kernel-mode arrives at the codes in ALSA Core. ALSA Core includes some modules for each interface such as PCM. The code in these modules check events and continue the process if the event occurs. Else, it makes the process to sleep. When sound devices generate hardware interrupt, each ALSA driver handles the interrupt and generate a event. This is a basic work of ALSA implementation in kernel-land.

8.1 ALSA Core and drivers

ALSA Core consists of modules which perform important roles in sound subsystem. The sources of ALSA Core locate under sound/core/. ALSA Core manages sound devices as 'card', and manages character devices as 'components' which belong to the card. The component represents each functionality on sound devices such as PCM or MIDI, and the modules include helper functions for the functionalities.

Each driver registers a card instance to ALSA Core and adds component instances to the card. Then ALSA Core request Linux Driver Core to add character devices to system. When Linux Driver Core grants the request and handle it, it registers the character devices and adds corresponding kobject, then notifies kevent. In user-land, udevd receives the kevent and adds special files to system for the character devices. Finally ALSA applications can communicate to components of drivers.

Major modules of ALSA Core are:

snd
: Management of cards and devices, including methods for control interface, helper functions for control component and procfs nodes

snd_pcm
: Including methods for PCM interface and helper functions for PCM component

snd_rawmidi
: Including methods for MIDI interface and helper functions for MIDI component

snd_hwdep
: Including methods for hardware dependent interface and helper functions for hardware dependent component

snd_timer
: Including methods for timer interface and helper functions for timer component

These components include methods of common file operations such as ioctl or mmap, and defines a batch of ALSA-specific operations for drivers. Each driver registers functions as the methods at adding components. In this way, each driver becomes to handle requests from ALSA applications[34].

For communication to actual devices, drivers use each Linux subsystem because the devices are on buses such like PCI-Express bus or platform bus. The drivers should be registered to bus driver and handle callbacks from the bus driver. Additionally, data transmission is mostly based on DMA and related events are notified by hardware interrupt. Each driver should also uses Linux subsystem to handle DMA and hardware interrupt.

In next subsection, I describe interaction between PCM interface/component and application/driver.

8.2 ALSA PCM Interface and PCM Component

ALSA PCM Interface and PCM Component are for transmission of PCM frames between applications and devices.

An instance of PCM Component includes two members for PCM streams. One is for an incoming stream and another is for an outgoing stream. The stream has several PCM substreams, which the driver adds according to device's specification. For the substreams, ALSA Core adds PCM character devices. When applications open PCM character devices by PCM interface, corresponding PCM substream is opened and PCM runtime is attached to the substream.

The PCM runtime has several state, 'open', 'setup', 'prepared', 'running', 'xrun', 'draining', 'paused', 'suspended' and 'disconnected'[33]. Applications changes the state by PCM interface and corresponding method of driver is called by PCM component. The runtime includes several information such like hardware description and hardware/software parameters. When the runtime is created, drivers set the hardware description in the runtime. Next, the application sets the hardware/software parameters by PCM interface.

When the parameters are decided, driver keeps PCM ring buffer. In most case, drivers support mmap file operation of PCM Component. Then PCM interface maps this buffer to process' virtual address area when applications or master plugins set mmap access flags to the hw_params[8].

The state of this buffer is described in the other buffers. They're 'status' buffer and 'control' buffer. These buffers are kept in kernel-space by PCM Component when applications or master plugins open hw plugin. Then, in most case, 'hw_ptr' and 'appl_ptr' in these buffers are mapped to processes' virtual address area[9].

```
struct snd_pcm_mmap_status {
        snd_pcm_state_t state;
        int pad1;
        snd_pcm_uframes_t hw_ptr;
        struct timespec tstamp;
        snd_pcm_state_t suspended_state;
        struct timespec audio_tstamp;
};

struct snd_pcm_mmap_control {
        snd_pcm_uframes_t appl_ptr;
        snd_pcm_uframes_t avail_min;
};
```

The 'appl_ptr' is updated only by applications and there's two cases. One of

[8] SND_PCM_ACCESS_MMAP_INTERLEAVED, SND_PCM_ACCESS_MMAP_NONINTERLEAVED and SND_PCM_ACCESS_MMAP_COMPLEX

[9] When failed to mmap(2), an alternative way is applied. hw plugin keeps a buffer for sync_ptr structure and execute ioctl(2) to update this pointer when any PCM frames are handled.

the cases is that some PCM frames are written or read by non-mmap functions[10]. In this case, PCM component moves the 'appl_ptr' in kernel-space by the number of requested frames. Another case is that some PCM frames are committed by mmap functions[11]. In this case, hw plugin changes the 'appl_ptr' in user-space by the number of frames.

On the other hand, the another pointer, 'hw_ptr' is changed by drivers when handling any hardware interrupts from devices. A page including 'hw_ptr' is mapped as read-only, and no applications can change it. The 'hw_ptr' is updated by the number of frames per 'period'. The 'period' is a part of buffer into which the ring buffer is split. When the devices confirm to finish transferring PCM frames, they call snd_pcm_period_elapsed(). Then the 'hw_ptr' is moved to next beginning of period.

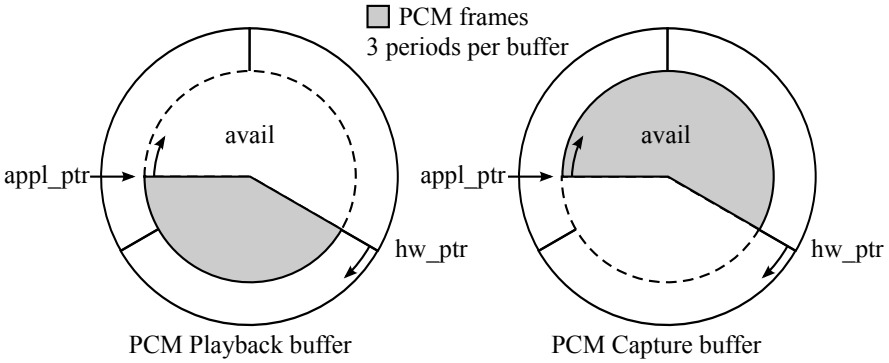

Figure 21: PCM ring buffer

The two pointers, 'hw_ptr' and 'appl_ptr', are used to check 'avail' space for application. The 'avail' space is the number of frames currently available for applications. For playback application, the 'avail' space equals to a space without PCM frames. For capture application, the 'avail' space equals to a space with PCM frames. When hw plugin writes or reads frames with non-blocking operation, functions instantly return if the 'avail' space has less frames than requested frames. When hw plugin writes or reads frames with blocking operation, the application is blocked until timeout or all of requested frames are transferred. As a default, the timeout is 10 sec. The timeout means 'hw_ptr' has never changed during the 10 sec and there may be some troubles about interrupt handler. But this behaviour is changed with 'NO_PERIOD_WAKEUP' hardware flag[12] for PCM substream. If driver supports this functionality, the timeout is expanded to the maximum number which Linux process-scheduler supports[13]. In this mode, hardware interrupts should be generated periodically

[10]The functions are snd_pcm_writei() / snd_pcm_writen() / snd_pcm_readi() / snd_pcm_readn() but be sure to plugin-chain.

[11]The function may be snd_pcm_mmap_commit() but be sure to plugin-chain

[12]SNDRV_PCM_HW_PARAMS_NO_PERIOD_WAKEUP

[13]MAX_SCHEDULE_TIMEOUT in include/linux/sched.h

and drivers are sure to handle them.

When applications executes poll(2) for descriptors gained by snd_pcm_poll_descriptors() with hw plugin, the process is blocked by default till timeout or 'avail' space has a rest. PCM Component has two modes for this behaviour. One is interrupt-base, as described. Another is a combination of both the interrupt-base and timer-base. When using the combination mode, snd_pcm_poll_descriptors() returns two descriptors for both PCM character device and Timer character device. Both of them wake up per period but the source of timer is different. The way to change the mode is to set 'period_event' software parameter for PCM runtime, by snd_pcm_sw_params_set_period_event(). PulseAudio uses a combination of 'NO_PERIOD_WAKEUP' hardware parameter and 'period_event' software parameter for its 'glitch free' operation[14].

The rest of buffer is 'hw_avail', available space for devices. For applications, the 'hw_avail' is the space for rewinding. The applications can reverse 'appl_ptr' by the number of frames in 'hw_avail'.

The actual value for these two pointers doesn't round up in the number of frames in buffer. It's incremented until 'boundary' within UINT_MAX, then round up. The way to calculate the 'boundary' is:

$$boundary = UINT\_MAX - UINT\_MAX / frames\_per\_buffer \qquad (4)$$

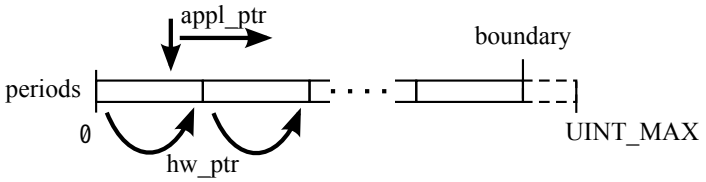

Figure 22: PCM buffer boundary

8.3 Linux firewire subsystem

ALSA firewire stack uses Linux firewire subsystem to communicate to firewire devices. Before investigating ALSA firewire stack, I describe Linux firewire subsystem.

Linux firewire subsystem, so-called Juju[15], is IEEE 1394 and OHCI 1394 application and support these functionalities:

- IEEE 1394 Serial bus management
- IEEE 1394 Transaction layer
- Driver for OHCI 1394 compliant controller device

[14]Features/GlitchFreeAudio http://fedoraproject.org/wiki/Features/GlitchFreeAudio
[15]I don't know the reason.

- Driver for fw character device

This subsystem locates under drivers/firewire/ and consists of several parts. It includes snoop mode driver for a certain 1394 controller and protocol drivers for RFC 1734, RFC 2855, RFC 3146 and SBP-2, but here I describe core functionality and OHCI 1394 related stuff only:

drivers/firewire/core-cdev.c
Methods for file operations of fw character devices

drivers/firewire/core-device.c
Helper functions for management of fw character devices

drivers/firewire/core-topoligy.c
Management of bus topology by reaction to events on the bus

drivers/firewire/core-iso.c
Helper functions to handle isochronous context

drivers/firewire/core-transaction.c
Helper functions to handle asynchronous context

drivers/firewire/core-card.c
Abstract layer for host controller devices

drivers/firewire/ohci.c
Driver for OHCI 1394 compliant host controller

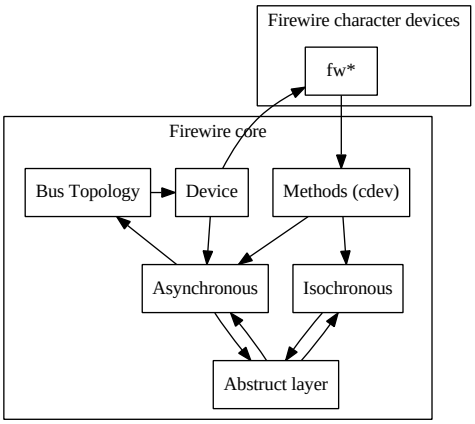

Figure 23: Linux firewire subsystem (a higher part)

This subsystem performs Cycle Master and Bus Manager. This subsystem adds fw character devices for applications. The fw character devices are corresponding to each node on IEEE 1394 bus. The applications can communicate

to the nodes by file operations to the fw character devices, mainly ioctl(2). As ALSA does, this subsystem also gives a library (libraw1394) for API as a abstract layer for the specific file operations. Additionally, this subsystem also gives API in kernel-land.

The functionalities of Asynchronous Communication and Isochronous Communication are based on OHCI 1394. So they're represented as context (section 4.5 to page 7).

Linux firewire stack has an abstract layer for the controller device. Currently two drivers are under this abstract layer. The dummy driver is mainly used to shutdown actual cards, so usually OHCI 1394 driver is used. This driver receives/transmits data of packet from/to controller by DMA, and handle hardware interrupts the controller generates. The driver controls the controller via registers.

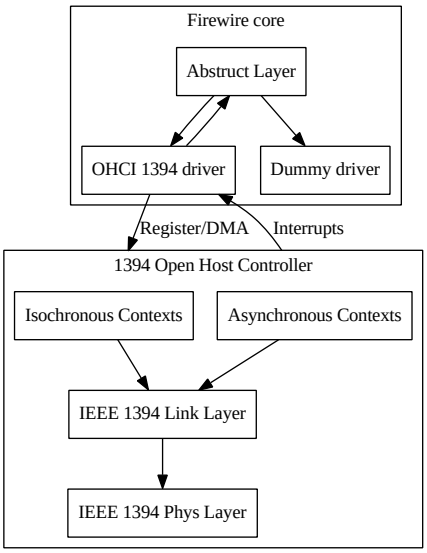

Figure 24: Linux firewire subsystem (a lower part) and host controller

8.4 ALSA firewire stack

ALSA firewire stack is designed for each driver to communicate to firewire devices via firewire subsystem. ALSA firewire stack locates under sound/firewire/. This stack includes helper functions for IEC 61883-1/6:

sound/firewire/iso-resources.c
: Helper functions to manage isochronous resources

sound/firewire/cmp.c
: For Connection Management Procedure (CMP) in IEC 61883-1

sound/firewire/lib.c
Helper functions for asynchronous transaction

sound/firewire/fcp.c
For Function Control Protocol (FCP) in IEC 61883-1

sound/firewire/packets-buffer.c
Helper functions to manage buffer for continuous packets

sound/firewire/amdtp.c
For AMDTP packet streaming in IEC 61883-6

All of them are used to built for 'snd-firewire-lib' module. While firewire drivers use the helper functions to communicate to sound devices, drivers also use ALSA Core to communicate to ALSA applications.

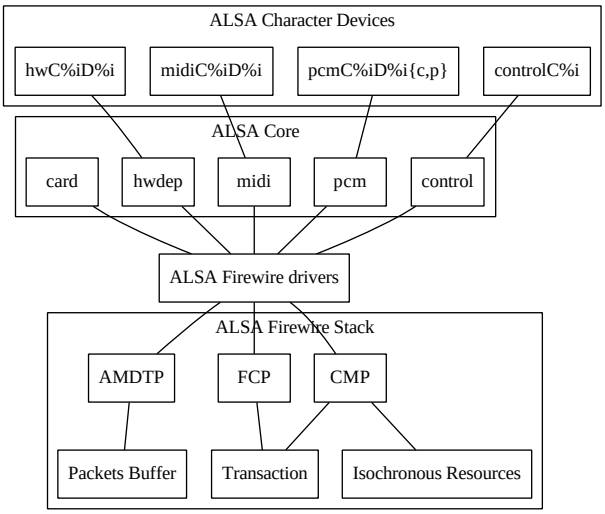

Figure 25: ALSA Core and firewire stack

8.4.1 Isochronous resources functionality

This module gives a structure for isochronous resources such as channels and bandwidth. Drivers initialize an instance of the structure by fw_iso_resources_init() with firewire unit. After initializing, the drivers can allocate/free isochronous resources by fw_iso_resources_allocate()/fw_iso_resources_free(). When bus reset occurs, fw_iso_resources_update() should be called. To release the instance, fw_iso_resources_destroy() is used.

8.4.2 CMP functionality

This module gives a structure for CMP operation. This functionality uses the isochronous resources functionality internally. Drivers initialize an instance of the structure by cmp_connection_init(). To keep bandwidth/channels and establish a connection, cmp_connection_establish() is used with maximum number of bytes for payload of a packet, and cmp_connection_break() is used to break the connection. To release the instance, cmp_connection_destroy() is used after breaking the connection. When bus reset occurs, cmp_connection_update() is used to update the connection.

As of 2013, this functionality supports operations for iMPR/iPCR so as input connections.

8.4.3 FCP and transaction functionality

FCP functionality gives a function, fcp_avc_transaction() for FCP and AV/C commands. The drivers can transmit FCP request and wait for FCP response for 125 millisecond. All of FCP requests should be retried when bus reset occurs, so fcp_bus_reset() is used for this purpose.

FCP functionality internally uses snd_fw_transaction(). This is a wrapper function of fw_run_transaction() and supports retrying and generation constraint.

8.4.4 Continuous packets buffer functionality

This module gives a structure for continuous packet buffer. Drivers initialize an instance of the structure by iso_packets_buffer_init() with the number of packets and the number of bytes in an packet. After initialized, the instance has pages and the pages can be used to queue packets. To releasing the instance, iso_packets_buffer_destroy() is used.

8.4.5 AMDTP functionality

This module gives a structure for AMDTP stream and some operations to handle an instance of the structure. This functionality uses the continuous packets buffer functionality internally. Drivers initialize an instance of the structure by amdtp_stream_init(). After initializing, amstp_stream_set_parameters() is used to set streaming parameters such as sampling rate, the number of Multi Bit Linear Audio (MBLA) data channel and MIDI Conformant data channel for the stream. After setting parameters, amdtp_stream_get_max_payload() returns the maximum number of bytes of payload for a packet with current parameters.

Drivers can start the stream by amdtp_stream_start(). Then amdtp_stream_stop() is used to stop the stream. To release the stream, amdtp_destroy_stream() is used. When bus reset occurs, amdtp_stream_update() is used to prepare for changing node ID.

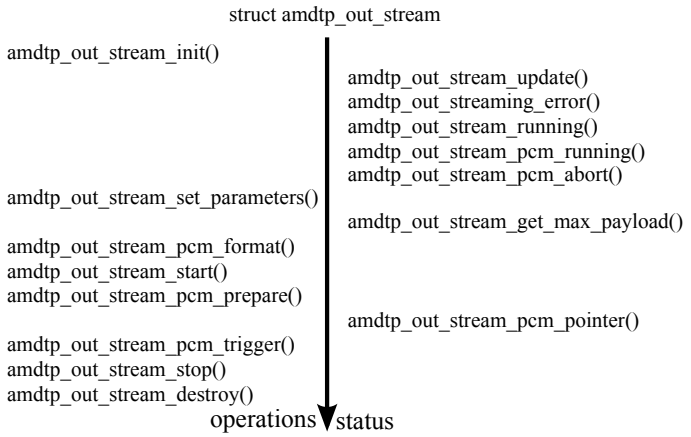

Figure 26: Life cycle of an instance for old AMDTP stream

For PCM component, firewire-lib gives helper functions. During streaming, To set PCM format, amdtp_stream_set_pcm_format() is used. To initialize PCM-related members of a stream, amdtp_stream_pcm_prepare() is used. When starting/stopping PCM substream, amdtp_stream_pcm_trigger() is used. To get the numbers of frames in PCM buffer, amdtp_stream_pcm_pointer() is used. When killing PCM substreams, amdtp_stream_pcm_abort() is used. To check PCM substream is running or not over a stream, amdtp_stream_pcm_running() is used.

Let's see the detail to generate AMDTP outgoing packets in Figure 27. The amdtp_stream_start() function queues a series of packets with a 'skip' flag, then firewire-ohci adds the packets into descriptor list. To the end of the series, the function sets 'interrupt' flag. When the 'interrupt' packet is queued, firewire-ohci sets special flag to a descriptor corresponding to the packet. After queueing, the amdtp_stream_start() function starts an isochronous context with a callback function. Then firewire-ohci starts to transfer descriptors to OHCI 1394 controller via DMA, and OHCI 1394 controller transmits packets corresponding to descriptors. When OHCI 1394 controller finish transmitting the 'interrupt' packet, it generates hardware interrupt. The firewire-ohci handles the hardware interrupt and schedules tasklet. When the tasklet is executed in software interrupt context, the callback function is executed. Then AMDTP functionality generates new series of packets with PCM samples, and queueing them. When queueing the end of a series of packets, the functionality sets the 'interrupt' flag again. When OHCI 1394 controller transmits this 'interrupt' packet, it generates hardware interrupt and snd-firewire-lib generates new series of packets and queues them.

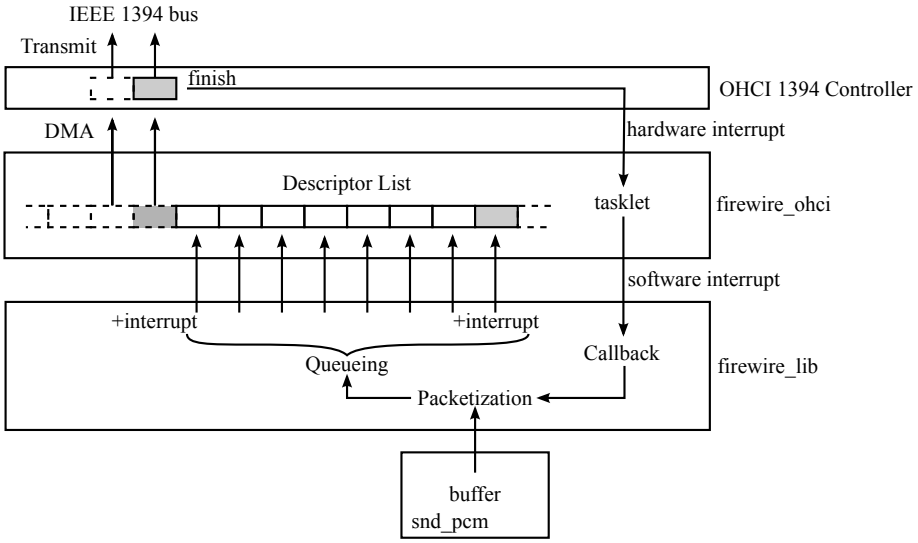

Figure 27: Generating packets for out stream

As of 2013, AMDTP functionality uses 48 queues [16] and 16 packets for interrupt interval [17]. So hardware interrupts nominally occur every 2 millisecond but this really depends on bandwidth for asynchronous communication. See Figure 7. Roughly, it takes 6 millisecond to transmit a first packet since staring isochronous context [18].

There is a case that firewire-lib fails to continue streaming. It's to fail queueing. To check whether a stream is running or not, amdtp_stream_running() is used, and amdtp_streaming_error() is used to detect error. Once streaming becomes error state, the stream should be stopped by amdtp_stream_stop(), then start by amdtp_stream_start() again.

As of 2013, this functionality don't allows drivers to perform as a timing slave because incoming packets are not supported. Therefore, this functionality includes a generator for the sequence of value to SYT field. In non-blocking mode, the sequence should be corresponding to the number of events in each packet. So this functionality also includes a generator for the sequence of number of events. Generally, there's no common way to generate these sequence so they're one of implementing practices.

8.5 Requirements for ALSA firewire stack

As of 2013, this stack is only for outgoing stream (received stream in device side) in both blocking/non-blocking mode. In detail,

[16] QUEUE_LENGTH in sound/firewire/amdtp.c
[17] INTERRUPT_INTERVAL in sound/firewire/amdtp.c
[18] In IEEE 1394, 8,000 packets are surely transferred in a second

- CMP layer handles iMPR/iPCR only
- AMDTP layer handles isochronous outgoing context only
- AMDTP packet can transfer PCM samples only
- The value of SYT field in packets from device is not available for outgoing packet

To support more devices, this stack requires these functionalities:

- Handling oMPR/oPCR
- Handling isochronous incoming context
- Handling several data in an AMDTP packet, like MIDI
- Handling duplex streams with timestamp synchronization

Additionally, Each driver require these functionalities:

- Retrieving device capabilities and parsing them to make PCM runtime rules
- Maintaining duplex streams
- PCM component for both of playback and capture
- MIDI component for both of playback and capture
- Handling model specific operations and quirks

Furthermore, each driver should prevent from disturbing user-land drivers. Dice driver already gives these functionalities via hwdep interface:

- Giving Firewire node information corresponding to ALSA card
- Notifying to start/stop streaming
- Locking/Unlocking streaming

Comparing to an idea to implement drivers in user-land, an idea to implement it in kernel-land has less issues of communication to ALSA application because this idea can use ALSA Core. This idea gives good transparency to ALSA applications. Moreover, there's already ALSA firewire stack for firewire drivers. The stack has a lack of some requirements but an enhancement of the stack is less work than implementation in user-land.

9 Enhancement of ALSA firewire stack

I started this work in the beginning of 2013, and it's merged to ALSA upstream May 2014. It takes one and a half years. June 2014, at Linux 3.16 merge window, ALSA maintainer pushed this work into Linux upstream. The patchset for this work consists of 63 patches but 14 patches of them are for correction.

The first 18 patches are for firewire-lib, for AMDTP, CMP and FCP. The next 14 patches are for fireworks driver. The next 16 patches are for bebob driver.

9.1 Enhancement for AMDTP

The firewire-lib newly supports these functionalities for AMDTP:

- Handling incoming AMDTP stream
- PCM capturing support
- MIDI capturing/playbacking support
- AMDTP data channel mapping
- duplex streams with timestamp synchronization

This work renames prefix of 'amdtp_out_stream' to 'amdtp_stream' for all of functions and members of AMDTP stream structure, to use the same structure for both directions.

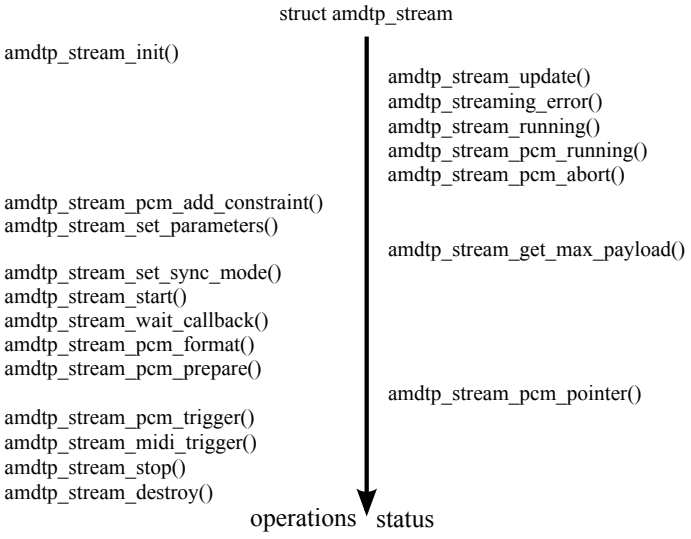

Figure 28: Life cycle of an instance for new AMDTP stream

This work newly adds a support for AMDTP in stream, and adds an argument for amdtp_stream_init(), to indicate the direction of AMDTP stream. After initialized, the instance of stream can be operated by the same way for both direction.

Let's see the detail to handle AMDTP incoming stream in Figure 29. The mechanism is almost the same as generating outgoing packets, while there are two big differences.

One is dequeueing packets instead of generating packets. When OHCI 1394 controller fills an 'interrupt' descriptor with an incoming packet, it generates a hardware interrupt. The firewire-ohci handles this hardware interrupt and schedules tasklet. When the tasklet is executed in software interrupt context, a callback function in snd-firewire-lib is executed. Then AMDTP functionality perform dequeueing and queueing packets.

In current implementation, the length of queue and the interval of interrupt is the same for incoming/outgoing AMDTP stream. As a result, it nominally takes 2 millisecond to handle a first packet since the packet reached.

Another is a functionality to detect discontinuity of the sequence of incoming AMDTP packets. Each AMDTP packet has DBC field for this purpose. In a case of the detection, the stream is stopped and corresponding PCM substream is aborted. In this case, the stream should be stopped and started again.

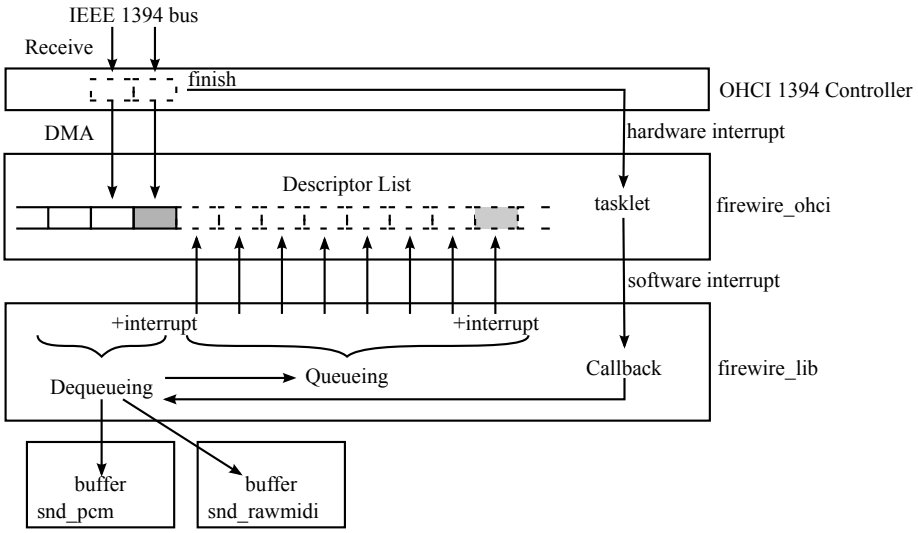

Figure 29: Handling packets for in stream

This work newly adds amdtp_stream_wait_callback() to wait for a first packet until given timeout. This is for detection of the case that drivers set wrong parameters for packets, i.e. tag for IEEE1394 isochronous packet. In this case, OHCI 1394 driver doesn't execute callback function in isochronous context. Then no packets are handled, and PCM component don't waken a process of

PCM substream because 'avail' space has never been updated (section 8.2 to page 25). This function is expected to be used just after staring a stream, to prevent from this case. Drivers should handle error state when receive false.

This work adds some flags for AMDTP stream structure, to handle device's quirks of stream. Between initializing and starting a stream, these flags can be set to an instance of the stream. Fireworks and BeBoB drivers use these flag.

As a default, any AMDTP streams are not associated. For duplex streams with timestamp synchronization (section 5.4 to page 12), this work adds amdtp_stream_set_sync(). This function associates two streams as a timing master and a timing slave with given synchronization mode, and this function should be called before starting streams. Currently CIP_SYNC_TO_DEVICE is effective for this parameter and for AMDTP stream in blocking mode. The case to use this option is that a timing master is incoming stream and a timing slave is outgoing stream. In this case, the value of SYT field in incoming packets are passed to outgoing packets. The packets in both direction are handled in interrupt context for an incoming isochronous context. So a callback in isochronous context for outgoing stream is ignored. In the other cases, there's no requirement to pass the value of SYT field between two streams.

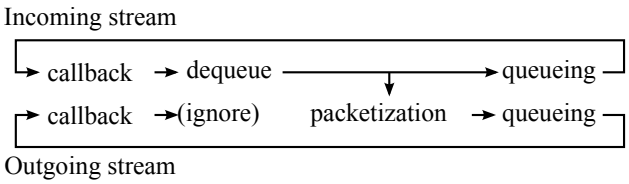

Figure 30: Handling associated duplex streams

This work newly adds a support of MIDI substream to AMDTP stream. During streaming, each drivers can trigger MIDI messages with amdtp_stream_midi_trigger(). Some models ignore MIDI messages in more than 8 data blocks in an packet. This may be due to a gap of data rate between MIDI bus and IEEE 1394 bus. To handle this quirk, struct amdtp_stream.rx_blocks_for_midi is used.

For PCM component, this work newly adds amdtp_stream_add_hw_constraints() to add PCM constraint according to parameters of AMDTP stream. This function should be called after all of needed flags are set to an instance of AMDTP stream structure.

This work also enables drivers to call amdtp_stream_set_pcm_format() during a stream is running. This is for the case that PCM substream is going to join in AMDTP stream which MIDI substream already starts.

9.2 Enhancement for CMP

This work adds an argument to cmp_connection_init(), to indicate direction. After initializing, the instance of CMP is handled as what it was.

Each driver can read iPCR/oPCR by transaction in any time. To confirm no drivers handle the device, cmp_connection_check_used() is used. This is a consideration about user-land driver.

I note that the value of payload field in iPCR is not used to keep bandwidth because some device chip reports a wrong value. Instead of this, snd-firewire-lib uses the value of data_block_quadlets member in struct amdtp_stream.

9.3 Enhancement for FCP

This work enables firewire-lib to handle deferred transaction. According to AV/C General specification[21], receivers of AV/C command should respond to transmitter within 100 millisecond, while the receivers send an intermediate response within 100 millisecond, and send a final response later. This is deferred transaction. The interval between the intermediate and final response is undefined. But to promise to return to caller, firewire-lib uses 125 millisecond interval.

Additionally, this work adds three AV/C commands, which seems to be used in some drivers. They're INPUT/OUTPUT SIGNAL FORMAT command and PLUG INFO command in AV/C General specification[21].

10 Driver implementation

These new functionalities help to develop new drivers. Furthermore, existed drivers can be updated for both PCM/MIDI playback/capture.

10.1 Common functionality

PCM functionality of each driver has SND_PCM_INFO_BATCH flag because the driver uses a part of packet queues for each PCM period. PCM functionality also has SND_PCM_INFO_BLOCK_TRANSFER flag because PCM frames are transferred at each interrupt by the number of data blocks in some packets [19]. PCM samples are copied to each data blocks in its order and the position of PCM samples in a data block is linear, so SNDRV_PCM_INFO_INTERLEAVED should be used. If devices supports both streams, in most case, incoming/outgoing AMDTP streams should be the same sampling rate, so SNDRV_PCM_INFO_JOINT_DUPLEX should be used. Each drivers should restrict sampling rate for both playback/capture as the same. If current source of clock is not internal or recovered clock (section 5.1 to page 8), the sampling rate should not be changed.

MIDI functionality is supported. Most devices have a pair of input/output port but a few devices have several ports.

Hwdep functionality is used to the same purpose as Dice driver uses. Additionally some drivers implement its own operation into hwdep interface.

[19] QUEUE_LENGTH in /sound/firewire/amdtp.c

To help debug, the drivers add proc nodes under the card's firewire subdirectory. When reporting bugs, users should also reports output from these nodes.

For most devices, the drivers has no control functionality. The usage of this functionality is to control device's DSP behaviour (section 5.2 to page 11). But this can be achieved by user-land implementation. For this purpose, currently, ffado-dbus-server/ffado-mixer are one of recommendations.

10.2 Fireworks driver

Fireworks driver locates in sound/firewire/fireworks/ and consists of 9 files. When initializing, the module register a handler into a certain address to receive responses of Fireworks transaction. The structure of Fireworks transaction is in include/uapi/sound/firewire.h.

```
#define SND_EFW_TRANSACTION_USER_SEQNUM_MAX ((__u32)((__u16)~0) - 1)
/* each field should be in big endian */
struct snd_efw_transaction {
        __be32 length;
        __be32 version;
        __be32 seqnum;
        __be32 category;
        __be32 command;
        __be32 status;
        __be32 params[0];
};
```

ALSA applications can transmit request by writing hwdep interface. Each instance of fireworks driver has a queue to store the responses and ALSA applications can read them via hwdep interface, too. There is a rule for request that ALSA applications must use the correct value for 'seqnum' member between 0 to SND_EFW_TRANSACTION_USER_SEQNUM_MAX because value of out of the range is used by the driver itself. The driver doesn't implement whole fireworks commands, implements needed commands to maintain streams or help debug.

To support much quirks of AMDTP packets, the driver apply several flags to amdtp structure, but CMP is normal and separately from each direction.

10.3 BeBoB driver

BeBoB driver locates in sound/firewire/bebob/ and consists of 12 files. To handle vendor specific operation, the driver has several files for each vendor. Especially, M-Audio largely customized and two models [20] have many specific operations. These devices also has write-only controls. Because of these controls, the driver adds Control interface.

[20] Firewire 1814 and ProjectMix I/O

BeBoB has a quirk at bus reset. It transmits packets with disorder just after bus reset. To recover from this discontinuity, the driver uses Linux completion and serialize PCM component .prepare() call and firewire subsystem .update() call.

10.4 Dice driver

10.5 OXFW driver

11 Rest of issue

Currently I focus on enabling ALSA to handle devices. So there are some issues which don't cause big problems.

11.1 OXFW and Dice enhancement

I already posted patches for these drivers [21] [22]. The previous revision of OXFW driver [23] has a regression for a device. The device has several entries for stream formation at a certain sampling rate and parser of driver brought this regression. Dice enhancement adds supports for duplex streams with timestamp synchronization and playback/capture of PCM/MIDI.

I don't have the device confirmed with regression and any Dice based devices. So I really need testers.

11.2 A lack of packet sorting

Some Fireworks based devices sometimes transfer packets with disorder. Such sequence of packet is detected as discontinuity and firewire-lib stop streaming. I've already posted a candidate for packet sorting but this is rejected by maintainer in three points:

- AMDTP is nearly real-time protocol so it should be processed as is.
- Currently this quirk is confirmed only on Fireworks so no need to apply general solution.
- A pattern is confirmed for the discontinue sequence of packets. It's possible to add a few codes for this quirk.

I confirm the disorder does not always occurs. Further investigation is needed to judge better solution.

[21] [alsa-devel] [RFC][PATCH 00/15 v4] OXFW driver, a successor to firewire-speakers http://mailman.alsa-project.org/pipermail/alsa-devel/2014-May/076581.html

[22] [alsa-devel] [RFC][PATCH 00/13 v1] Enhancement for Dice driver http://mailman.alsa-project.org/pipermail/alsa-devel/2014-May/076730.html

[23] It's a part of [alsa-devel] [RFC v3] [PATCH 00/52] Enhancement for support of firewire devices http://mailman.alsa-project.org/pipermail/alsa-devel/2014-January/071820.html

11.3 A lack of throttles for MIDI messages in outgoing stream

In MIDI specification, its phy layer can transfer 31,250 bit per seccond. This equals to 3,906 Byte per second. But in IEC 61883-6[17, 18] and MMA/AMEI RP-027[13], the stream can transfer MIDI messages by one eighth of sampling transmit frequency (STF). This is really much than MIDI specification so devices must buffer the gap between these data rate.

Table 4: Data rate for MIDI message over IEEE 1394 bus

STF	Byte per second
32,000	4,000.0
44,100	5,512.5
48,000	6,000.0
88,200	11,025.0
96,000	12,000.0
176,400	22,050.0
192,000	24,000.0

A few devices have buffer-over-flow when receiving much MIDI messages. For example, M-Audio Firewire 1814 transfer discontinuous packets when receiving much MIDI messages. There're some devices which ignores MIDI messages in more than first 8 data blocks in a packet. In this reason, ALSA Firewire stack should have a throttle for MIDI messages in outgoing stream.

11.4 A lack of arrangement for severe latency

Currently, the minimum number of frames per period is not less than 5 millisecond. PCM component of each driver has a flag, SNDRV_PCM_INFO_BATCH and the minimum number of periods in buffer is 2. As a result, the minimum latency is over 10 millisecond. For severe latency, the size of period should be as smaller as possible.

Ideally, the number of PCM frames in a period should be the number of data blocks in packets of one interrupt. But in both of blocking and non-blocking mode, the number of data blocks in packets of one interrupt is different. See Figure 31. In this figure, the interval of SYT is 8 data blocks, the number of packets in an interrupt is 4, the number of PCM frames in an period is 16. In blocking mode, PCM period is elapsed two times in 3rd interrupt. In non-blocking mode, PCM period is elapsed two times in 5th interrupt. These don't follow an idea of PCM period (section 8.2 to page 25).

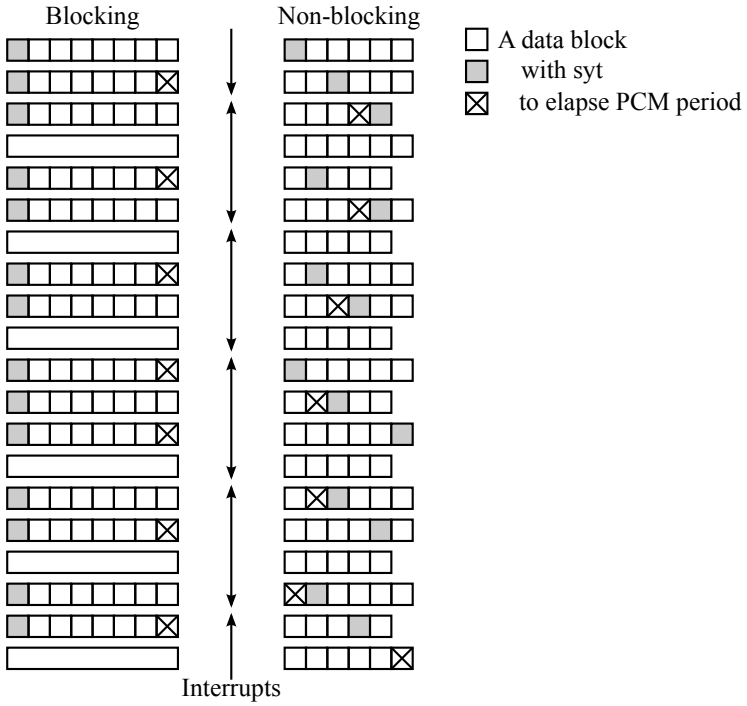

Figure 31: Packets, interrupt and PCM period

To apply severe latency, the interrupt interval should be changed dynamically according to sampling transmission frequency and the number of frames in a period. But this needs correct calcularion and expectation of the number of packets till next PCM period. But they are a bit complicated. So current implementation uses 5 miliseconds to ensure that at least one interrupt occurs between PCM periods.

11.5 A few devices freeze when frequent establishing/breaking connections

'PreSonus FP10' can freeze when frequent establishing/breaking connections. This brings a disadvantage to PulseAudio. When PulseAudio detects the card, it starts/stops 4 streams at least. Currently PulseAudio cannot detect devices with non-surround channels. But when making PulseAudio detects the device with dbus rule and PulseAudio profiles, it may take the device freezing.

11.6 Various results at bus reset

Various results are confirmed with Fireworks/BeBoB at bus reset. They're depending on cases:

Case 1. Software bus reset

When generating bus reset with ffado-test or jujuutils[24], streaming stopped suddenly. When generating bus reset, the drivers update connections according to IEC 61883-1. But after a few hundred millisecond, the device break connections by itself. I don't know the reason exactly but I guess the synchronizing mechanism I described has some relations.

Case 2. Connecting devices or removing devices on the bus

This case depends on which devices are connected or removed. If they're Fireworks/BeBoB based devices, streaming sometimes continues and sometimes stops. If they're not Fireworks/BeBoB based device, streaming continues. I don't know the reason exactly but I guess the synchronizing mechanism I describe have some relations.

11.7 A lack of Control interface

To reduce maintaining effort and source code base, ALSA firewire stack and firewire drivers basically have no Control interface. But the sound devices needs software implementation to control DSP behaviour (section 5.2 to page 11).

Currently, ffado-dbus-server/ffado-mixer is one of recommendation for this purpose but these implementations are isolated from ALSA applications.

Like ALSA PCM interface, ALSA Control interface has External Control plugins in alsa-lib. So it may be possible to give ways to control DSP for ALSA applications.

But there is an issue that a few devices have write-only control so there is a need to prepare for permanent storage.

11.8 A lack of synchronizing several devices on the same bus

As long as reading manual of PreSonus/Focusrite/PrismSound for BeBoB based devices, there is a way to synchronize devices on the same bus.

This mechanism is driven by handling the value of SYT field in CIP packets:

- Driver select a device as 'Sample clock source' then the other devices are 'Sample clock destination'.
- Driver handle packets from 'Sample clock source' device.
- Driver read SYT and calculate presentation timestamp for each devices, then transfer packets to the other devices.

IEC 61883-6:2005[18]describes actual examples in Annex D.1.3 Embedded sample-clock jitter.

Currently ALSA Firewire stack doesn't support this mechanism. The reason is a bad balance between actual cost to implement its requirements and actual advantage to implement them. The requirements are:

[24]https://code.google.com/p/jujuutils/

- Requirement to manage states of several devices simultaneously
- Requirement to handle several streams from/to all devices simultaneously

Instead of this type of synchronization, ALSA Firewire stack handle packets from a device and read SYT, then transfer packets with the value to the device.

References

[1] IEEE 1394-1995 IEEE Standard for a High Performance Serial Bus

[2] IEEE 1394a-2000 IEEE Standard for a High Performance Serial Bus - Amendment 1

[3] IEEE 1394b-2002 IEEE Standard for a High Performance Serial Bus - Amendment 2

[4] IEEE 1394c-2006 IEEE Standard for a High Performance Serial Bus - Amendment 3

[5] IEEE 1394-2008 IEEE Standard for a High Performance Serial Bus

[6] IEEE 1212-1991: IEEE Standard Control and Status Register (CSR) Architecture for Microcomputer Buses

[7] ISO/IEC 13213:1994 Information technology – Microprocessor systems – Control and Status Registers (CSR) Architecture for microcomputer buses

[8] IEEE 1212-2001: IEEE Standard Control and Status Register (CSR) Architecture for Microcomputer Buses

[9] TA Document 1997001 Audio and Music Data Transmission Protocol Version 1.0

[10] TA Document 1999014 Enhancement to Audio and Music Transmission Protocol Version 1.0

[11] TA Document 2001003 Audio and Music Transmission Protocol 2.0

[12] TA Document 2001024 Audio and Music Transmission Protocol 2.1

[13] MMA/AMEI RP-027 MIDI Media Adaptation Layer for IEEE-1394 Version 1.0 (Nov 30 2000)

[14] IEC 61883-1:1998 Consumer audio/video equipment - Digital interface - Part 1: General, Edition 1.0

[15] IEC 61883-1:2003 Consumer audio/video equipment - Digital interface - Part 1: General, Edition 2.0

[16] IEC 61883-1:2008 Consumer audio/video equipment - Digital interface - Part 1: General, Edition 3.0

[17] IEC 61883-6:2002 Consumer audio/video equipment - Digital interface - Part 6: Audio and Music Data Transmission Protocol, Edition 1.0

[18] IEC 61883-6:2005 Consumer audio/video equipment - Digital interface - Part 6: Audio and Music Data Transmission Protocol, Edition 2.0

[19] 1394 Open Host Controller Interface Specification Release 1.0 (Oct 20 1997)

[20] 1394 Open Host Controller Interface Specification Release 1.1 (Jan 06 2000)

[21] TA Document 2004006, AV/C Digital Interface Command Set General Specification Version 4.2, September 1, 2004

[22] TA Document 1999008, AV/C Audio Subunit 1.0, August 21, 2000

[23] TA Document 2001007, AV/C Music Subunit 1.0, April 8, 2001

[24] TA Document 1999025, AV/C Descriptor Mechanism Specification Version 1.0, July 23, 2001

[25] TA Document 1999045, AV/C Information Block Types Specification Version 1.0, July 23, 2001

[26] TA Document 2000004, Enhancements to the AV/C General Specification 3.0 Version 1.1, Oct 24, 2000

[27] AV/C Stream Format Information Specification V1.0

[28] TA Document 2004008, AV/C Stream Format Information Specification 1.1 (working draft), April 15, 2005

[29] TA Document 1999015, AV/C Command Set for Rate Control of Isochronous Data Flow 1.0

[30] Using TSB43Cx43A: S/PDIF Over 1394 (Dec 2003, Texas Instruments Incorrporated)

[31] Additional AVC commands - AV/C Unit and Subunit - SDD Products Draft 1.7-00 (Nov 17 2003, BridgeCo)

[32] AV/C Connection Management - Implementation Recommendation - SDD Products Draft 4.0-01 (Jul 16 2004, BridgeCo AG)

[33] ALSA Library API
http://www.alsa-project.org/main/index.php/ALSA_Library_API

[34] Writing an ALSA Driver (2005, Takashi Iwai)
http://www.alsa-project.org/main/index.php/ALSA_Driver_Documentation

A Functions of Linux firewire stack

ALSA firewire stack utilize functions exported by Linux firewire stack. In this appendix, I describe the functions.

A.1 Serial Bus Management

```
void fw_iso_resource_manage(struct fw_card *card, int generation,
                            u64 channels_mask, int *channel,
                            int *bandwidth, bool allocate);
```

This function allocates or deallocates a channel and/or bandwidth for resources of isochronous communication, to run transactions to CSR_CHANNELS_AVAILABLE and CSR_BANDWIDTH_AVAILABLE in Isochronous Resource Manager.

```
void fw_schedule_bus_reset(struct fw_card *card, bool delayed,
                           bool short_reset);
```

This function schedules bus reset on OHCI 1394 controller.

A.2 Firewire subsystem transaction services

```
struct fw_address_handler {
        u64 offset;
        u64 length;
        fw_address_callback_t address_callback;
        void *callback_data;
        struct list_head link;
};
struct fw_address_region {
        u64 start;
        u64 end;
};
int fw_core_add_address_handler(struct fw_address_handler *handler,
                                const struct fw_address_region *region);
void fw_core_remove_address_handler(struct fw_address_handler *handler);
```

The 'fw_core_add_address_handler' allocates an address region to the handler. If the address space is not already allocated or the address space is for FCP, the allocation is success. The callback in the handler is called when receiving transactions to the address space.

```
void fw_send_response(struct fw_card *card,
                      struct fw_request *request, int rcode);
int fw_cancel_transaction(struct fw_card *card,
                          struct fw_transaction *transaction);
```

The 'fw_send_response()' and 'fw_cancel_transaction()' are used in the callback of the handler to send response or cancel the transaction.

```
int fw_run_transaction(struct fw_card *card, int tcode, int destination_id,
                       int generation, int speed, unsigned long long offset,
                       void *payload, size_t length);
```

The 'fw_run_transaction()' is used to send asynchronous transaction to the node.

A.3 OHCI 1394 isochronous contexts

```
struct fw_iso_buffer {
        enum dma_data_direction direction;
        struct page **pages;
        int page_count;
        int page_count_mapped;
};
int fw_iso_buffer_init(struct fw_iso_buffer *buffer, struct fw_card *card,
                       int page_count, enum dma_data_direction direction);
void fw_iso_buffer_destroy(struct fw_iso_buffer *buffer, struct fw_card *card);
```

This structure describes DMA mapped pages for OHCI 1394 isochronous context. The payload of each IEEE 1394 isochronous packet is stored in this buffer.

```
struct fw_iso_context *fw_iso_context_create(struct fw_card *card,
                        int type, int channel, int speed, size\_t header_size,
                        fw_iso_callback_t callback, void *callback_data);
```

This function keeps isochronous context on OHCI 1394 host controller, then allocate DMA buffers for OHCI 1394 packet descriptor and IEEE 1394 isochronous packet header. The caller must allocate channel and bandwidth in advance.

```
struct fw_iso_packet {
        u16 payload_length;
        u32 interrupt:1;
        u32 skip:1;
        u32 tag:2;
        u32 sy:4;
        u32 header_length:8;
        u32 header[0];
};
```

This structure basically describes IEEE 1394 Isochronous Packet. An 'interrupt' member is to generate software interrupt (tasklet) from hardware interrupt of OHCI 1394. If the value of 'interrupt' member is 1 in every 16 packets, it means that the software interrupt is generated every 16 packets handled by OHCI 1394 controller.

```
int fw_iso_context_queue(struct fw_iso_context *ctx,
                         struct fw_iso_packet *packet,
                         struct fw_iso_buffer *buffer,
                         unsigned long payload);
```

This function queueing given packet for isochronous context. The members of 'struct fw_iso_packet' are copied to DMA buffer for OHCI 1394 descriptor.

```
void fw_iso_context_queue_flush(struct fw_iso_context *ctx);
```

This function wakes up isochronous context with queued packets.

```
int fw_iso_context_flush_completions(struct fw_iso_context *ctx);
```

This function generates software interrupt (tasklet) for isochronous context.

```
int fw_iso_context_start(struct fw_iso_context *ctx,
                         int cycle, int sync, int tags);
```

This function takes isochronous context to start.

```
int fw_iso_context_stop(struct fw_iso_context *ctx);
```

This function takes isochronous context to stop and disable software interrupt (tasklet).

```
void fw_iso_context_destroy(struct fw_iso_context *ctx);
```

This function takes isochronous context to stop and disable software interrupt, then release isochronous context and deallocate buffer.

B Functions of ALSA firewire stack

In this appendix, I describe functions of ALSA firewire stack in detail.

B.1 Continuous packets helper functions

These functions are in sound/firewire/iso-packets.{c,h}.

```
struct iso_packets_buffer {
        struct fw_iso_buffer iso_buffer;
        struct {
                void *buffer;
                unsigned int offset;
        } *packets;
};
int iso_packets_buffer_init(struct iso_packets_buffer *b, struct fw_unit *unit,
                            unsigned int count, unsigned int packet_size,
                            enum dma\_data_direction direction);
void iso_packets_buffer_destroy(struct iso_packets_buffer *b,
                                struct fw_unit *unit);
```

This structure includes 'struct fw_iso_buffer' and non-tag structure. The non-tag structure consists of pointer and offset to access the content of memory pages in packet unit. The 'iso_packets_buffer_init()' function calculates the number of memory page frames with 'packet_size' argument, 'count' argument, L1_CACHE_ALIGN() and PAGE_SIZE and keep the pages for DMA with 'fw_iso_buffer_init()'. The 'iso_packets_buffer_destroy()' releases these.

B.2 Isochronous Rresource Management helper functions

These functions are in sound/firewire/iso-resources.{c,h}.

```
struct fw_iso_resources {
        u64 channels_mask;
        struct fw\_unit *unit;
        struct mutex mutex;
        unsigned int channel;
        unsigned int bandwidth;
        unsigned int bandwidth\_overhead;
        int generation;
        bool allocated;
};
int fw_iso_resources_init(struct fw_iso_resources *r,
  struct fw_unit *unit);
void fw_iso_resources_destroy(struct fw_iso_resources *r);
int fw_iso_resources_allocate(struct fw_iso_resources *r,
                              unsigned int max_payload_bytes, int speed);
```

```
int fw_iso_resources_update(struct fw_iso_resources *r);
void fw_iso_resources_free(struct fw_iso_resources *r);
```

The structure 'fw_iso_resources' represents an manager for isochronous resource such like channel and bandwidth. The 'fw_iso_resources_init()' initializes it and 'fw_iso_resources_destroy()' release it. The 'fw_iso_resources_allocate()' keeps isochronous resources with given 'max_payload_bytes' and 'speed'. The 'fw_iso_resources_update()' reallocate isochronous resources to handle bus reset. The 'fw_iso_resources_free()' releases isochronous resources. Most operations utilizes 'fw_iso_resource_manage()' in Linux Firewire Subsystem.

B.3 CMP helper functions

These functions are written in sound/firewire/cmp.{c,h}.

```
enum cmp_direction {
        CMP_INPUT = 0,
        CMP_OUTPUT,
};
struct cmp_connection {
        int speed;
        bool connected;
        struct mutex mutex;
        struct fw_iso_resources resources;
        __be32 last_pcr_value;
        unsigned int pcr_index;
        unsigned int max_speed;
        enum cmp_direction direction;
};
int cmp_connection_init(struct cmp_connection *connection,
                        struct fw_unit *unit,
                        enum cmp_direction direction,
                        unsigned int pcr_index);
int cmp_connection_check_used(struct cmp_connection *connection, bool *used);
void cmp_connection_destroy(struct cmp_connection *connection);
int cmp_connection_establish(struct cmp_connection *connection,
     unsigned int max_payload);
int cmp_connection_update(struct cmp_connection *connection);
void cmp_connection_break(struct cmp_connection *connection);
```

The 'struct cmp_connection' includes parameters related to CMP such like PCR index. The 'cmp_connection_init()' check Output/Input MPR by a READ transaction according to given 'direction' and 'pcr_index', then initialize 'resources' member. The 'cmp_connection_destroy()' release these structures. The 'cmp_connection_establish()' keeps isochronous resources and establishes a connection by a LOCK transaction. The 'cmp_connection_update()' updates a connection by a LOCK transaction to handle bus reset. The 'cmp_connection_break()'

breaks an established connection by a LOCK transaction. The connection should be broken before destroyed.

The 'cmp_connection_check_used()' is a bit special. This is added for relationships to drivers in user-space. When the drivers handle devices, the caller of this function can check the driver's connection as long as the devices are compliant to CMP.

B.4 AMDTP stream helper functions

These functions are in sound/firewire/amdtp.{c,h}.

```
enum amdtp_stream_direction {
        AMDTP\_OUT_STREAM = 0,
        AMDTP\_IN_STREAM
};
enum cip_flags {
        CIP_NONBLOCKING         = 0x00,
        CIP_BLOCKING            = 0x01,
        CIP_SYNC_TO_DEVICE      = 0x02,
        CIP_EMPTY_WITH_TAG0     = 0x04,
        CIP_DBC_IS_END_EVENT    = 0x08,
        CIP_WRONG_DBS           = 0x10,
        CIP_SKIP_DBC_ZERO_CHECK = 0x20,
        CIP_SKIP_INIT_DBC_CHECK = 0x40,
        CIP_EMPTY_HAS_WRONG_DBC = 0x80,
};
struct amdtp_stream {
        struct fw_unit *unit;
        enum cip_flags flags;
        enum amdtp_stream_direction direction;
        struct fw_iso_context *context;
        struct mutex mutex;

        struct iso_packets_buffer buffer;
        int packet_index;

        enum cip_sfc sfc;
        unsigned int syt_interval;

        unsigned int source_node_id_field;
        unsigned int data_block_quadlets;
        unsigned int data_block_counter;
        unsigned int pcm_channels;
        unsigned int midi_ports;
        u8 pcm_positions[AMDTP_MAX_CHANNELS_FOR_PCM];
        u8 midi_position;
```

```
        unsigned int tx_dbc_interval;

        unsigned int transfer_delay;
        unsigned int data_block_state;
        unsigned int last_syt_offset;
        unsigned int syt_offset_state;

        struct amdtp_stream *sync_slave;

        bool callbacked;
        wait_queue_head_t callback_wait;

        struct snd_pcm_substream *pcm;
        void (*transfer_samples)(struct amdtp_stream *s,
                                 struct snd_pcm_substream *pcm,
                                 __be32 *buffer, unsigned int frames);
        unsigned int pcm_buffer_pointer;
        unsigned int pcm_period_pointer;
        bool pointer_flush;
        struct tasklet_struct period_tasklet;

        struct snd_rawmidi_substream *midi[AMDTP_MAX_CHANNELS_FOR_MIDI * 8];
        unsigned int rx_blocks_for_midi;
};
```

This structure represents an AMDTP stream. An AMDTP stream can transfer one PCM substream and 8 MIDI substreams. Currently this structure is for AM824 data. The 'buffer' member is for continuous packet buffer, and the 'index' is for handling the continuous packets. The 'sfc' and 'syt_interval' are dominant parameters. These parameters are decided by stream's nominal sampling rate:

```
enum cip_sfc \{
        CIP_SFC_32000  = 0,
        CIP_SFC_44100  = 1,
        CIP_SFC_48000  = 2,
        CIP_SFC_88200  = 3,
        CIP_SFC_96000  = 4,
        CIP_SFC_176400 = 5,
        CIP_SFC_192000 = 6,
        CIP_SFC_COUNT
\};
extern const unsigned int amdtp_syt_intervals[CIP_SFC_COUNT] = {
        [CIP_SFC_32000]  =  8,
        [CIP_SFC_44100]  =  8,
```

```
        [CIP_SFC_48000]  =  8,
        [CIP_SFC_88200]  = 16,
        [CIP_SFC_96000]  = 16,
        [CIP_SFC_176400] = 32,
        [CIP_SFC_192000] = 32,
};
```

The 'source_node_id_field', 'data_block_quadlets', 'data_block_counter' are for CIP header. The 'tx_dbc_interval' is for a device-specific quirk which has fixed interval for the value of data block counter, not compliant to IEC 61883-1.

The 'transfer_delay', 'data_block_state', 'last_syt_offset', 'syt_offset_state' are used to generate the value of SYT field in non-blocking mode in simplex streams without timestamp synchronization. The 'syc_slave' is used for blocking mode with duplex streams with timestamp synchronization to handle timestamps in incoming packets to outgoing packets.

The 'callbacked' and 'callback_wait' are used to wait till a first packet is available.

The 'pcm' and 'transfer_samples' members are used to multiplex/demultiplex PCM samples into an AMDTP packet. The 'pcm_buffer_pointer', 'pcm_period_pointer', 'pointer_flush', 'period_tasklet' are used for 'snd_pcm_period_elapsed()'.

The 'midi' member is used to multiplex/demultiplex MIDI message into an AMDTP packet. The 'rx_blocks_for_midi' is for a model-specific quirk to ignore MIDI messages in data blocks more than a certain numbers in a packet.

```
bool cip_sfc_is_base_44100(enum cip_sfc sfc);
```

The 'cip_sfc_is_base_44100()' returns given 'sfc' is 44100, 88200, 176400 or not.

```
int amdtp_stream_init(struct amdtp_stream *s, struct fw_unit *unit,
                      enum amdtp_stream_direction dir,
                      enum cip_flags flags);
```

The 'amdtp_stream_init()' initializes members in the structure. This function gets reference counter from given 'struct fw_unit'. The 'dir' and 'flags' are important for an AMDTP stream.

void amdtp_stream_destroy(struct amdtp_stream *s);

The 'amdtp_stream_destroy()' clear some members in the structure. This function put reference counter to given 'struct fw_unit'. The caller is sure to stop the stream in advance.

```
void amdtp_stream_set_parameters(struct amdtp_stream *s,
                                 unsigned int rate,
                                 unsigned int pcm_channels,
                                 unsigned int midi\_ports);
```

The 'amdtp_stream_set_parameters()' set parameters to given stream structure. As a result, the stream structure has 'sfc', 'syt_interval', 'pcm_channels', 'midi_ports', 'data_block_quadlets', 'transfer_delay'. This function also initializes 'pcm_positions' and 'midi_positions' according to IEC 61883-6:2005.

```
unsigned int amdtp_stream_get_max_payload(struct amdtp_stream *s);
```

The 'amdtp_stream_get_max_payload()' returns the number of bytes as maximum size of payload of CIP for an AMDTP packet. The caller must call 'amdtp_stream_set_parameters()' in advance.

```
void amdtp_stream_set_sync(enum cip_flags sync_mode,
                           struct amdtp_stream *master,
                           struct amdtp_stream *slave);
```

As a default, the synchronization mode is simplex stream without timestamp synchronization. The 'amdtp_stream_set_sync()' set synchronization mode between master and slave streams for duplex streams with timestamp synchronization.

```
int amdtp_stream_start(struct amdtp_stream *s, int channel, int speed);
bool amdtp_stream_wait_callback(struct amdtp_stream *s, unsigned int timeout);
```

The 'amdtp_stream_start()' starts an amdtp stream over given channel and speed. Then Linux Firewire subsystem start isochronous context and to handle related interrupts. The 'amdtp_stream_wait_callback()' is to wait till a first packet can be available or timeout with given value.

```
bool amdtp_stream_running(struct amdtp_stream *s);
bool amdtp_streaming_error(struct amdtp_stream *s);
```

The 'amdtp_stream_running()' is to check whether the stream is running or not. The 'amdtp_streaming_error()' is to check whether the stream is stopped with errors. The reasons of error is described later in this chapter.

```
void amdtp_stream_update(struct amdtp_stream *s);
```

The 'amdtp_stream_update' is used to update the 'source_node_id_field' member, basically to handle bus reset. Internally, amdtp_stream_start() also calls this function.

```
void amdtp_stream_stop(struct amdtp_stream *s);
```

The 'amdtp_stream_stop' stops and destroys isochronous context, to stop an AMDTP stream. Then this function releases the buffer for continuous packets.

B.5 PCM substream helper functions for AMDTP stream

```
#define AMDTP_IN_PCM_FORMAT_BITS        SNDRV_PCM_FMTBIT_S32
#define AMDTP_OUT_PCM_FORMAT_BITS       (SNDRV_PCM_FMTBIT_S16 | \
                                         SNDRV_PCM_FMTBIT_S32)
```

These macro is for 'struct snd_pcm_hardware.formats'. This represents current firewire-lib capability.

```
#define AMDTP_MAX_CHANNELS_FOR_PCM      64
```

This macro represents the maximum number of PCM channels which current firewire-lib can handle.

```
int amdtp_stream_add_pcm_hw_constraints(struct amdtp_stream *s,
                                        struct snd_pcm_runtime *runtime);
```

This function adds constraints to given PCM substream runtime according to given stream.

```
void amdtp_stream_set_pcm_format(struct amdtp_stream *s,
                                 snd_pcm_format_t format);
```

The 'amdtp_stream_set_pcm_format()' sets 'transfer_samples' member into stream according to given format. This is expected to be called in 'struct snd_pcm_ops.hw_params()'.

```
void amdtp_stream_pcm_prepare(struct amdtp_stream *s);
```

The 'amdtp_stream_pcm_prepare()' initializes members related to handling 'snd_pcm_period_elapsed()'. This is expected to be called in 'struct snd_pcm_ops.prepare()'.

```
unsigned long amdtp_stream_pcm_pointer(struct amdtp_stream *s);
```

The 'amdtp_stream_pcm_pointer()' returns the number of frames in an PCM ring buffer.

```
void amdtp_stream_pcm_trigger(struct amdtp_stream *s,
                              struct snd_pcm_substream *pcm);
```

The 'amdtp_stream_pcm_trigger()' sets 'pcm' member to given AMDTP stream. This is expected to be called in 'struct snd_pcm_ops.trigger'.

```
void amdtp_stream_pcm_abort(struct amdtp_stream *s);
```

The 'amdtp_stream_pcm_abort()' stops PCM substream with XRUN state. This is expected to be used before stopping AMDTP stream.

```
bool amdtp_stream_pcm_running(struct amdtp_stream *s);
```

The 'amdtp_stream_pcm_running()' is to check whether PCM substreams is running on the AMDTP stream or not.

B.6 MIDI substream helper functions for AMDTP stream

```
#define AMDTP_MAX_CHANNELS_FOR_MIDI     1
```

This describes the maximum number of MIDI Conformant data channels in an AMDTP packet which firewire-lib can handle. According to IEC 61883-6:2005, 8 MPX-MIDI data streams are multiplexed into one MIDI Conformant data channel. So the number of maximum MIDI ports which firewire-lib can support is 8.

```
void amdtp_stream_midi_trigger(struct amdtp_stream *s,
                               unsigned int port,
                               struct snd_rawmidi_substream *midi)
```

The 'amdtp_stream_midi_trigger()' set MIDI substream to given AMDTP stream. This is expected to be called in 'struct snd_rawmidi_ops.trigger'.

B.7 Transaction service helper function

These functions are in sound/firewire/lib.{c,h}.

```
#define FW_GENERATION_MASK      0x00ff
#define FW_FIXED_GENERATION     0x0100
#define FW_QUIET                0x0200
int snd_fw_transaction(struct fw_unit *unit, int tcode,
                       u64 offset, void *buffer, size_t length,
                       unsigned int flags);
```

This function is a helper for an asynchronous transaction. Usually this function retries three times if the transaction is failed because of error or bus_reset. The flag 'FW_FIXED_GENERATION' returns error if the transaction is failed because of bus reset. The flag 'FW_QUIET' suppress error message.

B.8 FCP helper functions

These functions are in sound/firewire/fcp.{c,h}.

```
int fcp_avc_transaction(struct fw_unit *unit,
                        const void *command, unsigned int command_size,
                        void *response, unsigned int response_size,
                        unsigned int response_match_bytes);
```

The 'fcp_avc_transaction()' sends an AV/C command frame to the target and wait for the corresponding response frame. The 'response_match_bytes' is a bitmap to match frames of command/response in bytes unit. This function supports deferred transaction in AV/C Digital Interface Command Set General Specification.

```
void fcp_bus_reset(struct fw_unit *unit);
```

The 'fcp_bus_reset()' is used to handle bus reset. All of pending AV/C transactions are terminated and retried.

B.9 AV/C General commands

These functions are in sound/firewire/fcp.{c,h}.

```
enum avc_general_plug_dir {
        AVC_GENERAL_PLUG_DIR_IN         = 0,
        AVC_GENERAL_PLUG_DIR_OUT        = 1,
        AVC_GENERAL_PLUG_DIR_COUNT
};
int avc_general_get_plug_info(struct fw_unit *unit, unsigned int subunit_type,
                              unsigned int subunit_id, unsigned int subfunction,
                              u8 info[AVC_PLUG_INFO_BUF_BYTES]);
```

The 'avc_general_plug_dir()' is used to get plug information according to AV/C Digital Interface Command Set General Specification. Extended addressing is not supported.

```
int avc_general_set_sig_fmt(struct fw_unit *unit, unsigned int rate,
                            enum avc_general_plug_dir dir,
                            unsigned short plug);
int avc_general_get_sig_fmt(struct fw_unit *unit, unsigned int *rate,
                            enum avc_general_plug_dir dir,
                            unsigned short plug);
```

The 'avc_general_set_sig_fmt()' and 'avc_general_get_sig_fmt()' are used to set/get current sampling rate according to AV/C Digital Interface Command Set General Specification. Extended addressing is not supported.