

# FireWire (Pro-)Audio for Linux

Pieter PALMERS

pieterpalmers@users.sourceforge.net

## Abstract

FireWire audio devices are flooding the (semi-)pro audio interface market. At LAC2005, the FreeBoB project has been presented, aiming to implement support for devices built on top of the BridgeCo BeBoB platform.

This paper discusses the evolution of the FreeBoB project towards a framework generic enough to support a virtually any FireWire based audio device, BeBoB based or not, conforming to the 'official' standards or not. An overview of the design and implementation of the library is presented, and some key issues and their solutions are described.

## Keywords

FireWire, IEEE1394, driver, FreeBoB

## 1 Introduction

Since the presentation of the FreeBoB project's proof-of-concept code at LAC2005, a lot of progress has been made. An extensive number of changes and complete rewrites, followed by a 6 month beta testing stage, resulted in the release of FreeBoB-1.0, mid october 2006.

Meanwhile the interest of developers, users and vendors in FireWire audio on Linux was steadily increasing. This can be illustrated by a meeting held at LAC06 to discuss Linux support for non-BeBoB based devices. It became clear that the FreeBoB project could be a starting point to provide generalized FireWire audio support on Linux.

It was however obvious that the existing codebase was not easily extensible to provide support for the various options that exist for device discovery and data transport. The code had to be rewritten from scratch, marking the birth of FreeBoB-2.0, around may 2006.

Section 2 describes the FireWire system from a FreeBoB point of view. The design and implementation of the FreeBoB-2 library is described in sections 3 and 4. Finally section 5 provides some concluding remarks and briefly summarizes the end-user functionality.

## 2 System overview

This section discusses the different components in a FireWire audio setup on Linux. It starts with a description of the hardware bus topology and the way FireWire works from a FreeBoB perspective<sup>1</sup>, based upon [1]. Next, the current linux1394 driver stack and userspace interface will be discussed [2]. To conclude this section, the FreeBoB-2 library structure will be presented.

---

<sup>1</sup> The FireWire bus is designed for a multitude of functions. Aside from media related transport such as audio and video, common applications are storage access (SBP-2), networking (eth1394) and device control (e.g. Agilent ParBERT 81250). All of these applications use different subsets of the FireWire standard(s).

## 2.1 FireWire from an audio point of view

### 2.1.1 Bus topology

In summary, the FireWire<sup>2</sup> bus is a high-speed, peer-to-peer, self-configuring bus supporting true hot-plugging and plug-and-play. Whenever a device is (dis)connected, the bus is reset and reconfigured. The interfaces and protocols are designed such that a bus reset does not influence the functional behavior of a device, meaning that devices can be added or removed while all others remain fully operational.

A FireWire topology appears as one 64-bit memory mapped space, of which the first 10 bits specify the bus the device is connected to, and the next 6 bits specify the node ID of the device. This makes that a single bus can support 63 nodes (one broadcast ID) and that there can be 1023 busses (there is one 'local bus' ID) in a FireWire topology. Note however that the capability to address multiple busses is not used yet. The case where one computer contains more than one host controller doesn't qualify as one of multiple busses, as the different host controllers don't share the same address space. In such a case the system consists of multiple "topology's"<sup>3</sup>.

The remaining 48 bits of the address provide a 256 terabyte address space inside the device. Parts of this space are standardized (e.g. Configuration ROM), but most of it can be freely used.

### 2.1.2 Data transfer

The FireWire specification defines two main types of data transport: Isochronous and Asynchronous, both being packet based.

Isochronous traffic has the following properties:

- Broadcast one-to-one or one-to-many on a specific 'channel'
- A node can send only one packet per cycle

<sup>2</sup> The FireWire bus was developed by Apple, and is standardized by the IEEE as IEEE1394. It is also called i.Link by Sony.

<sup>3</sup> There is no official terminology for this

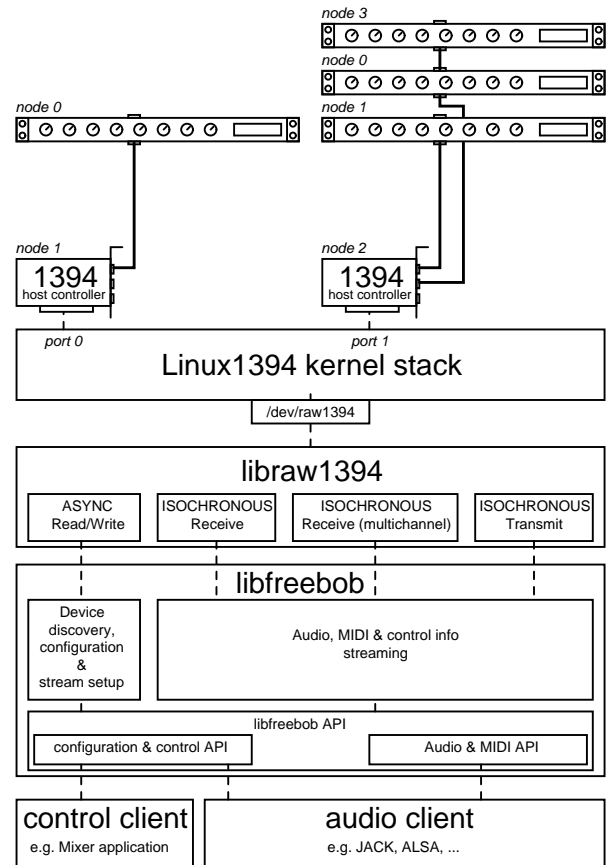


Figure 1: FreeBoB-2 system overview

- Up to 80% of the total bandwidth can be reserved and guaranteed for it
- It occurs at time intervals (cycles) that are regularly spaced in time (125us).
- No error correction or retransmission

The remaining part of the total bandwidth can be used by Asynchronous traffic, exhibiting the following properties:

- Between two specific nodes
- No guaranteed bandwidth
- Nodes can send multiple packets per cycle
- An async packet is acknowledged and optionally responded to by the receiving node, enabling error detection and/or correction

Isochronous traffic should be used for time-critical, error tolerant data transfer, while asynchronous transfers are intended for non error tolerant data or non time-critical transfers.

FireWire audio devices always use isochronous traffic to transport the audio and midi data, and

usually use asynchronous transfers for device configuration and control<sup>4</sup>.

### 2.1.3 Bus management roles

During the configuration phase, some special bus management tasks are assigned to nodes capable of executing them. For this description, the most relevant ones are the isochronous resource manager (IRM) and the cycle master.

The IRM keeps track of the isochronous channels that are in use, and the remaining bandwidth available for isochronous traffic. It provides the bandwidth guarantees that come with isochronous traffic.

The cycle master determines the timing for the isochronous traffic. At the start of every isochronous cycle, being a time slice with a nominal 125us period, the cycle master transmits a 'cycle start' packet. This packet contains the value of the cycle timer register (CTR) of the cycle master at the start of the cycle. This CTR is updated by the cycle master's clock source at 24.576MHz. The other nodes synchronize their CTR to this cycle start packet. The cycle master hence defines the concept of "time" on the bus.

## 2.2 The linux1394 stack

The linux1394 stack consists of a set of kernel modules and a userspace library that provides a clean interface to kernel space. The relevant parts of the stack are the *ohci1394*, *ieee1394* and *raw1394* kernel modules, and the *libraw1394* library.

### 2.2.1 Kernel modules

The linux1394 kernel stack was designed such that a multitude of host controllers could be supported. In order to accomplish this, a separation between the generic IEEE1394 bus functions and the driver for the hardware implementing them was introduced. The generic part is implemented in the *ieee1394* kernel module, while the host-controller

---

<sup>4</sup> Some devices (e.g. MOTU) make use of the isochronous transport channel to transport their status changes, e.g. due to front-panel operations by the user.

specific code<sup>5</sup> is in modules like *ohci1394* and *pcilynx*.

The *raw1394* module together with *libraw1394* provides the kernel-user space interface. The *raw1394* module is not intended to be used separately.

### 2.2.2 Userspace API: libraw1394

When applications want to use the linux1394 subsystem, they should use *libraw1394* instead of directly addressing */dev/raw1394*.

It supports asynchronous traffic through blocking api calls that encapsulate the complete request-ack-respond process. It also enables applications to act as a target for async transactions by having them register handlers for specific FireWire address ranges.

Isochronous transfer is implemented by registering a callback function to process incoming packets or provide outgoing packets, together with some stream setup and management functions (init, start, stop, ...).

### 2.2.3 Next-generation FireWire stack

Very recently a new FireWire kernel stack has been proposed to replace the current one. It simplifies the stack by only supporting OHCI cards and eliminating the obese layers. It will also support some more advanced methods to handle isochronous traffic. These should result in lower CPU usage and lower latency due to less intermediate buffering.

The development of this new stack gives is an opportunity to make sure that the specific requirements of pro-audio clients are served. The modular architecture of FreeBoB-2 should allow

---

<sup>5</sup>In reality there is only one type of host controllers available on the general market, being the OHCI1394 compliant ones. OHCI1394 [3] is a specification for a PCI host controller designed mainly by Intel [XX: and Microsoft?]. It enables the use of a unified driver for all compliant host controllers, and is the reason why FireWire extension cards come without a device driver CD.

for an easy transition to the new stack, resulting in early adoption.

### 2.3 LibFreeBoB-2

LibFreeBoB-2 is a C++ library that provides the translation between the FireWire packet based and the audio API's frame/buffer based environments. It can be split into two main parts, being the discovery & configuration part and the streaming part.

The discovery & configuration part is responsible for enumerating the devices that are available and supported. It also provides device-specific configuration functions such as setting the samplerate or controlling the hardware mixer. The streaming part translates the isochronous packet streams into audio and MIDI<sup>6</sup> streams and vice versa.

Although there are standards both for device discovery (AV/C) and audio transport (IEC61883-6), a one-implementation-fits-all approach has few chances to work. The device discovery standards are rather vague, making it possible to implement two different subsets with the same functionality, being equally compliant. On top of that, not all manufacturers care about the standards (e.g. MOTU). This is why support for multiple detection & configuration methods and multiple stream processors is needed. Device support is then achieved by combining a specific detection & configuration method with a specific stream processing method. This allows to re-use the common parts between devices.

One example are the BeBoB and DICE-II devices. Both are standards compliant, but the BeBoB discovery cannot be used for the DICE-II devices due to another interpretation of the standards. On the other hand, the stream processor for both devices can be the same, as the standard doesn't leave any room for interpretation there.

The FreeBoB library provides an external C API that can be used to implement audio clients (JACK

<sup>6</sup> There is also support for 'control data' streams that give feedback on the device status.

backend, ALSA plugin) and control clients (mixer application, device control panel, ...).

### 3 Device Discovery and Configuration Layer

The device discovery and configuration layer performs the following tasks:

- Enumeration of the supported devices present on the bus
- Enumeration of the capabilities of a device
- Configure the device, providing a generalized interface for configuration

This layer has very tight coupling with the device implementation. It is therefore less generalized, and will not be discussed further.

### 4 The Streaming Layer

The core function of the streaming layer is to (de)multiplex the isochronous streams (iso streams) into audio, MIDI and control streams having the appropriate format for the client. It should also recover the timing information.

First, the design of the streaming layer is described, presenting the general concepts. Then the implementation is described in more detail.

#### 4.1 Design

Figure 2 shows a conceptual overview of the FreeBoB library. On the client side it exposes a set of "ports" that represent the different types of streams provided to the client.

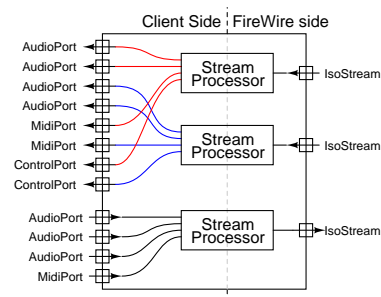


Figure 2: Conceptual overview of the data flow

On the FireWire side there are incoming or outgoing "iso streams". Inbetween these two interfaces there is a "stream processor" doing the actual work.

### 4.1.1 Client side port interface

Ports define the data transport from and to the clients. They have a type (audio, midi or control) that defines the nature of the data flowing through them, and a datatype property that defines the format of this data (float, uint24, ...). A port also has a direction property (playback or capture).

They can support multiple signalling types, depending on the time granularity required by the data. Two signalling types are currently supported: period and packet signalling.

Period signalled ports transfer the data in 'periods' as they are defined by the client. This allows for period-at-once (de)multiplexing of the isochronous streams, increasing efficiency. This signalling type is used for audio buffers.

For packet signalled ports the demultiplexing is done at the moment a packet is received, making the data is available at the port as soon as it arrives<sup>7</sup>. For playback ports, the data is multiplexed into the stream as soon as it is available at the port. This allows for the high time-granularity as needed by MIDI data.

Another property of a port is the data transfer method. Currently we provide two data transfer methods: blocking read from/write to a ringbuffer, and a direct to memory decode method into client supplied buffers.

### 4.1.2 FireWire side streaming interface

An 'iso stream' is an abstraction for a sequence of isochronous packets over time. It is linked to a specific isochronous channel in a specific FireWire 'topology' (or port in the *libraw1394* nomenclature). Its type can be 'receive' or 'transmit'.

The actual packet handling for an iso stream is performed by an 'iso handler' that interfaces with *libraw1394*, as indicated in figure 3. The reason for this intermediate iso handler is that one iso handler can serve multiple iso streams. *libraw1394*

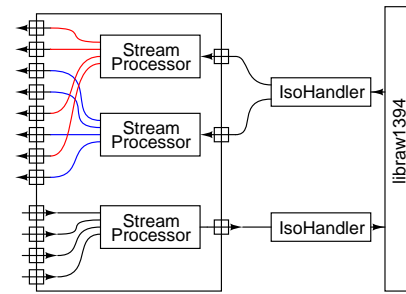


Figure 3: iso streams are fed or consumed by iso handlers

supports a multichannel isochronous receive mode, that allows to receive multiple channels at once. This mode can become important when a lot of devices are present, because it needs less hardware resources (only DMA engine, instead of one per channel).

### 4.1.3 The Stream Processor

A 'stream processor' is responsible for the (de)multiplexing of the ports into/from iso streams, i.e. for the conversion between audio/midi frames and isochronous packets. Every different data transport protocol requires a different stream processor.

The information provided by the detection process is used to construct a set of ports corresponding to the composition of the expected iso stream. It is also used to configure the (de)multiplexing code.

The last important function of the stream processor is recovering timing information (when to signal that a buffer is ready) from the incoming streams, as well as encoding this timing information into the outgoing streams.

### 4.1.4 Making it work

The presented concepts (ports, stream processors, iso streams and iso handlers) are not 'active' entities. In order to have them perform their task, two 'active' entities are introduced: the 'processor manager' and the 'handler manager'.

The processor manager is responsible for managing and executing the stream processors in the correct manner. This encompasses their

<sup>7</sup> « as soon as it is available » in this context should be regarded as « as soon as the timestamp of the data expires ».

initialization, the detection of period boundaries, the initiation of data transfers, etc...

The handler manager iterates the iso handlers, meaning that it will make sure that the isochronous packets are transferred from the kernel to the iso handlers and vice versa.

## 4.2 Implementation

The FreeBoB-2.0 streaming layer is written in C++, in order to facilitate abstraction and code reuse. This subsection will present the classes used to implement the design described previously.

### 4.2.1 The client side port interface

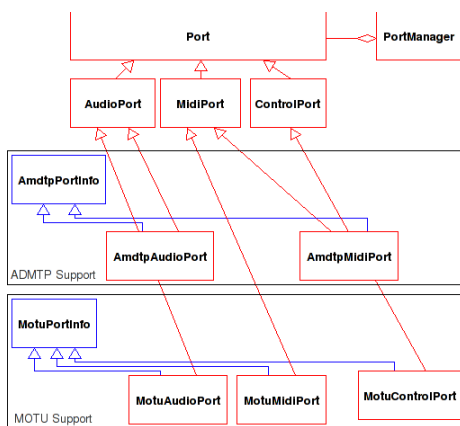


Figure 4: Class diagram for the client side Port interface, showing support for AMDTP (IEC61883-6) and MOTU streaming

Figure 4 shows a class hierarchy for the client side port interface. It starts with a *Port* base class, containing all functionality needed to implement the operations and properties described in the previous section. This *Port* class is subclassed for every port type.

A generic *Port* does not contain any information on the location of its data in the isochronous packets, as this information is specific to the streaming protocol used for the isochronous traffic. We provide this information by subclassing from a *[\*]PortInfo* class. The details of these *[\*]PortInfo*'s are protocol dependant, therefore there is no common base class for them. The *StreamProcessor* for a protocol is aware of the details of its corresponding *[\*]PortInfo* class.

In order to facilitate the management of a collection of *Ports*, a *PortManager* class has been implemented.

### 4.2.2 FireWire side streaming interface

The basic entities on the FireWire side, as shown in figure 5, are the *IsoStream* and *IsoHandler* classes. The *IsoStream* class implement the operations necessary for processing a single isochronous stream. This can be either consuming a sequence of packets provided by an *IsoHandler* of the receive type (*IsoRecvHandler* or *IsoMultiRecvHandler*), or producing a sequence for the *IsoXmitHandler*.

To manage *IsoHandler*'s, an

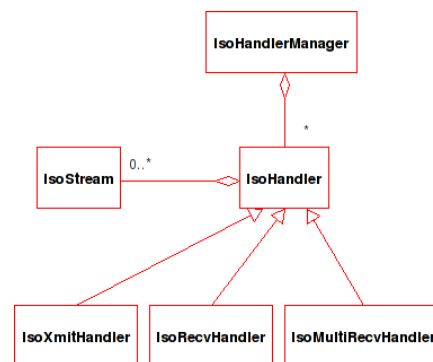


Figure 5: Class diagram for the FireWire side streaming interface

*IsoHandlerManager* class is implemented. This class creates and destroys *IsoHandler*s whenever they are needed. In order to link an *IsoStream* with an *IsoHandler*, the *IsoStream* has to be registered with the *IsoHandlerManager*. This class will decide whether to create a new *IsoHandler* or re-use an existing one (in case of multichannel receive). It will also determine the type of handler the *IsoStream* needs (receive or transmit), and its optimal settings. Finally it allows unregistering *IsoStreams*, destroying *IsoHandlers* that are not used anymore.

### 4.2.3 The StreamProcessor

The *StreamProcessor* classes are the workhorses in the FreeBoB library. They perform the actual operations to translate the client side data to/from isochronous streams.

Figure 7 shows that at one side, the

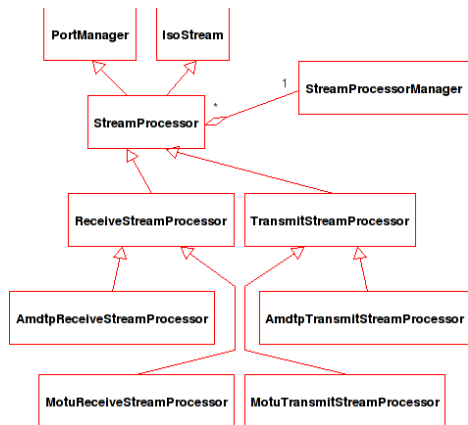


Figure 6: Class diagram for the *StreamProcessor* class

*StreamProcessor* is a *PortManager*, as it presents a set of *Ports* to the client side. At the other side, it is an *IsoStream*, as it processes a sequence of isochronous packets.

Leveraging the power of C++, we leave all common functionality in the *StreamProcessor* base class, and only implement the protocol-specific parts in the child classes.

In order to manage and 'activate' the collection of *StreamProcessors*, a *StreamProcessorManager* class is implemented. This class will be discussed in more detail in the next subsection.

#### 4.2.4 Making it all happen

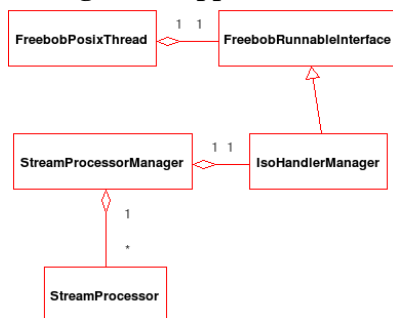


Figure 7: Class diagram for the active part of *FreeBoB*

The two basic 'active' operations of the library are the reception/transmission of packets, and the translation of these packets into audio/midi/control frames. These operations are implemented by the classes shown in Figure 7.

Isochronous reception and transmission is executed by the *IsoHandlerManager*'s

workfunction, that keeps running while the library is started. The *StreamProcessors* should decide if they want to process incoming packets and should always be able to generate valid packets (albeit no-data or empty packets).

On the client side, a *wait()* function is provided by the *StreamProcessorManager* to have the client wait for a period boundary. Once a period boundary is detected, the client *wait()* function returns. It should then call a *transfer()* function to have the *StreamProcessorManager* instruct its *StreamProcessors* to demultiplex and decode the queued packets (in bulk), and to transfer their contents to the *Ports*.

For the playback direction the inverse operations are performed, i.e. *Port* contents are encoded, packets are dequeued.

## 5 Issues and gory details

This section will talk about some implementation issues that were met when implementing *libFreeBoB-2*.

### 5.1 Embedding and using time information

The IEC61883-6 standard defines a method to embed the timing information of the frames in the packet. Most other streaming protocols (i.e. *Motu*) use variants of this technique. Therefore it is interesting to discuss this technique.

The idea behind the method is that every packet contains a timestamp that indicates the time at which the sample is to be 'presented'. The sender thus determines the time instant at which the receiver should process the sample. Calculation of the timestamp should be done by recording the *FireWire* cycle timer at the instant the sample is captured, adding the senders processing delay to this, and then adding some extra time to allow for transfer delays. Receivers are allowed to add some extra time to the received timestamp, as long as this extra time is constant.

The *FreeBoB* library handles timestamp synchronization by introducing a timestamp aware buffer. The basic idea is that every time one or

more frames are added to the buffer, the corresponding timestamp is passed along. The buffer uses these timestamps to calculate the timestamp of any other frame in the buffer. In order to cope with jitter issues it uses a delay locked loop as described in [4].

Since the timestamped buffer is able to predict the timestamp of any frame in the buffer, it is also able to predict the time instant at which one period of frames will be 'presentable'. This prediction is used by the processor manager to determine the time to wait for the next period. At initialization, one stream processor is elected as synchronization source, and this stream processor will be used to predict the buffer transfer time instant.

## 5.2 Synchronization across topologies

Having the timestamp expressed with respect to the FireWire cycle timer poses a significant synchronization challenge. It is not necessarily true that this cycle timer is related to the sample clock of the device. It can also happen that there are multiple cycle timers in a system, e.g. when two devices are connected to different host controllers (hence different busses). This scenario is currently unsupported, but will be in the future. Supporting it will require that one global time reference is elected, and all other time domains get synchronized to it. This will probably come down to electing the system timer as the global time reference, and implementing a cycle-timer to system time mapping for all domains.

## 5.3 IPC and multiplexing

There are some unresolved issues regarding IPC and the fact that different data is multiplexed upon the same stream.

The first issue is that capture isn't easily decoupled from playback due to them having to be synchronized. This prevents the use of playback and capture by different applications.

The fact that midi is transmitted along with the audio frames in the same stream poses some issues when interfacing to clients that don't have a midi

API. Providing midi to a different application as the audio is also a problem.

In some cases, feedback regarding the device's status is multiplexed into the isochronous streams. This can be for example a notification of a mixer volume change due to a front panel control. Usually the application needing this kind of information (e.g. a mixer application) is different from the one consuming the remainder of the stream (e.g. jack).

## 5.4 FireWire based processing units

The FireWire bus is not only used for audio interfaces, but also for processing units (e.g. TC PowerCore, SSL Duende). As these devices use the same concepts and protocols as audio interfaces, they fit the FreeBoB framework. It is not clear how they fit in the 'bigger' picture on Linux. In the Mac/Windows world, these devices present themselves e.g. as VST plugins. It is currently unclear to what extent the Linux alternatives provide support for this.

## 6 Conclusion

This paper presents the FreeBoB-2 library as (part of) a solution for FireWire Audio on Linux. It described the design and implementation of the library, along with some solved and some remaining issues.

## References

- [1] J. Canosa, Fundamentals of FireWire, Questra Consulting, <http://public.rz.fh-wolfenbuettel.de/~bermbach/research/FireWire/files/basics.pdf>
- [2] [www.linux1394.org](http://www.linux1394.org)
- [3] The OCHI1394 specification [http://developer.intel.com/technology/1394/download/ohci\\_11.htm](http://developer.intel.com/technology/1394/download/ohci_11.htm)
- [4] "Using a DLL to filter time", Fons Adriaensen