Kiel University of Applied Sciences

Master Information Engineering Research Project

Technical Report

Media IP Streaming on Embedded Systems

Niklas Wantrupp (928817)

Course of Studies: Information Engineering

Supervisor:
Prof. Dr. Robert Manzke

September 27, 2020

Contents

Abbreviations

ALSA	Advanced Linux Sound Architecture
AVB	Audio Video Bridging
AVDECC	Device Discovery, Enumeration, Connection Management and Control Protocol
BBAI	Beaglebone Ai
BBB	Beaglebone Black
BMCA	Best Master Clock Algorithm
CPU	Central Processing Unit
DAI	Digital Audio Interface
DAPM	Dynamic Audio Power Management
DMA	Direct Memory Access
DSP	Digital Signal Processor
DTO	Device Tree Overlay
LAN	Local Area Network
MAC	Media Access Control
NIC	Network Interface Card
FQTSS	Forwarding and Queuing Enhancements for Time-Sensitive Streams
gPTP	Generalized Precision Time Protocol
GSoC	Google Summer of Code
SoC	System-on-Chip
OSI	Open Systems Interconnection
OSS	Open Sound System
P2P	Peer-to-Peer
PCM	Pulse Code Modulation
QoS	Quality of Service
RT	Realtime
SRP	Stream Reservation Protocol
TDM	Time Division Multiplex

1 Introduction

Embedded systems managed to get applied to many application domains throughout the last two decades. This is especially due to the decreasing cost of system-on-chip development and increasing chip performance on a smaller amount of space over the years. Applications of embedded systems include, but are not limited to, industrial systems, cars, medical care systems, and mobile phones. Due to the large growth of system-on-chip performance, it has to be expected that application areas of embedded systems increase more and more over the coming years.

With the increase in processing power and dedicated digital signal processors getting assembled on embedded hardware, these systems are also extremely well suited to support media applications which are extremely demanding in regards of processing power.

Especially the field of media streaming can benefit from distributed embedded devices. Many broadcasting studios still work with analogue hardware to transmit and mix different signals within their production environment. Analogue connections however have the downside that every source has to be connected to another sink with its own cable. Digital connections on the other side have the advantage that signals can be transmitted and multiplexed over a single digital connection.

The IEEE 802.3 ethernet protocol is of special interest in this context because of its high bandwith capabilities. The problem with standard ethernet connections however is that transmission is package based and packages can be queued in network links. This should be avoided in realtime media transmission because it can lead to frame dropouts in the transmitted audio/video signal.

Because of this the AVB protocol suite has been developed, which introduces several protocols for media transmission in ethernet networks. Those protocols are gathered around the IEEE 802.1 BA-2011 Audio Video Bridging (AVB) Systems protocol. The AVB protocol allows to transmit realtime media signals over several hops within an AVB network and guarantee for in-time delivery of prioritized ethernet packages.

This project brings embedded systems and digital media transmission in ethernet networks together by implementing the AVB protocol stack on both, the Beaglebone Black (BBB) and Beaglebone AI (AI) for the kernel version 5.4-RT. The implementation is achieved by porting an existing implementation of Duraipandian [5] from Linux kernel 4.4. Furthermore, the drivers for the CTAG Face 2|4 card by Langer [15] are ported from kernel 4.4 to the kernel 4.19-RT. The whole project was conducted in the scope of Google Summer of Code (GSoC) 2020, which is a initiative by Google, to motivate students to work in open source projects [1]. The overall project goals can be summarized as follows:

1. Port CTAG Face drivers implemented by Langer [15] for GSoC 2016 to Linux kernel 4.19-rt for BeagleBone AI
2. Port AVB driver stack implemented by Duraipandian [5] for GSoC 2017 to Linux kernel 5.4-rt for BeagleBone AI
3. Refactor and modularize present AVB driver architecture
4. Implement Device Discovery, Enumeration, Connection Management and Control Protocol (AVDECC) protocol part of AVB to allow for device enumeration and control

[1] more information on GSoC can be found at: GSoC about

2 Technical Background

To give an introduction to the background in Linux audio card driver development on system-on-chip (SoC) platforms and the AVB protocol, this chapter briefly describes the basic Linux kernel structure, the ALSA framework including ALSA SoC and the basic structure of the AVB protocol family. For a detailed description of Linux kernel development Liberal de los Ríos [16], Corbet et al. [3] and *The Linux Kernel documentation* [19] are recommended. For a more thorough introduction on ALSA SoC, *ALSA SoC Layer Overview — The Linux Kernel documentation* [1] and Belloni [2] are recommended.

For a more detailed analysis on the AVB protocol family Duraipandian [5] is recommended and for exact description of the protocols, the protocol definitions IEEE 802.1 BA-2011 [12], IEEE 802.1 Qav-2009 [11], IEEE 802.1 Qat-2010 [14], IEEE 802.1 AS-2011 [10], IEEE 1722-2011 [8] and IEEE 1722.1-2013 [7] are recommended.

2.1 Linux Kernel Structure

The Linux kernel is a free and open source operating system, which was introduced by Linus Torvalds. Being open source, the existing codebase can be modified by everybody with sufficient knowledge. The freely available codebase which is open for modification is one of many advantages that helped Linux to get one the most used Operating Systems in the world [17].

Linux is based on a monolithic kernel architecture, implementing all kernel related operations in the kernel space. User processes which are spawned directly by the kernel communicate to system resources via pre-defined system calls. Single task boundaries within the Linux kernel are not always clearly defined but can be split into the following parts according to Corbet et al. [3]:

Process management

1. Managing the creation/destruction of processes
2. Managing process I/O
3. Managing how different processes are being executed on a single central processing unit (CPU)

Memory management

1. Building a virtual address space on top of constrained hardware resources
2. Managing how processes are allowed to allocate memory space

Filesystems

1. The Unix concept is heavily built on top of the concept of files
2. The kernel builds a structured filesystem on top of the hardware and most hardware units are being treated as if they were files

Device controls

1. The kernel needs device drivers for all hardware resources present in the system

2. Device drivers implement a well defined set of callbacks to allow the operating system to communicate to the hardware

Networking

1. The kernel implements routing and address resolution
2. The kernel is in charge to deliver specific data packets to network interfaces or processes
3. Control execution of programs depending on their network activity

2.1.1 Linux Device Driver Model

Linux defines a high level software interface over computer hardware. Computer hardware developers can use this interface to implement the functionality to communicate to the hardware resource using callback functions which are linked to pre-defined *C language strutures*. The kernel uses those structures to communicate to the device. Each device driver is implemented as a module. In the Linux domain those modules can be loaded either at boot time or can be inserted and removed during runtime [3, 16].

All devices which are implemented using the module system can be classified into one of three different classes [3]:

Char devices

- can be accessed as a stream of bytes like a file
- driver is in charge of implementing the correct behavior (*open, close, read, write*)
- can be accessed via standard file system nodes

Block devices

- accessed as file system nodes
- can host file systems
- on most UNIX systems block devices are accessed by reading/writing one ore more blocks of a fixed size but Linux allows to access block devices to be accessed like char devices
- thus the user can access a block device like a char device but the interface from the kernel to the device driver is completely different to a char driver

Network interfaces

- network interfaces are in charge of receiving and sending network packets
- because they are not stream oriented they are not mapped to filesystem nodes
- instead of read and write calls the kernel uses system calls related to packet transmission to communicate to an interface

2.2 ALSA Subsystem

The Advanced Linux Sound Architecture (ALSA) framework is the audio subsystem of the Linux kernel since version 2.6.0. It is the successor of the Open Sound Subsystem (OSS). ALSA allows Linux to provide Audio and MIDI functionality. It has support for several audio interfaces, defines an interface to the predecessor OSS and provides API's for user and kernel space developers to communicate with the ALSA middlelayer [18]. The following sections briefly describe how the kernel space API for writing sound cards is structured. For a more thorough description Dimitrov and Serafin [4] is recommended. Additionally the system-on-chip platform of the ALSA system is presented.

2.2.1 ALSA Driver API

The ALSA interface defines several keywords which are important to distinguish and which need to be understood before developing a soundcard driver. The basic structure addressed by the driver is the *card* or in ALSA terminology the *snd_card* which holds all information about the soundcard. Assuming a physical soundcard which has both in- and output capabilities the next level would address a capture and playback *device* or in ALSA terminology a *snd_device*. If the hypothetical soundcard has two channels for both in- and output it would have two *subdevices* one for the input and one for the output channel. These building blocks are utilized by the programmer to define hierarchical relationships between all components of the system [4].

Other vital structures, which are required by an ALSA driver are *snd_pcm_hardware* which describes the hardware capabilities like sampling rate or channel number, *snd_pcm_ops* which holds references to all implemented ALSA callback functions and lastly a bus type structure which depends on the actual soundcard bus i.e. pci, usb or platform for a virtual device [4].

2.2.2 ALSA SoC platform

Prior to the ALSA SoC platform the Linux kernel had limited support for audio on SoC platforms. This led to high power consumption, code duplication and no standard method to get notified of audio events like mic and headphone insertion. To overcome those shortcomings, the ALSA SoC framework was introduced. ALSA SoC is divided into three main components [1, 2]:

Code Class Driver

- platform and machine independent code to run on any platform
- configures the codec to run on any platform
- must provide: DAI and PCM configurations, Codec control I/O, Mixers and audio controls, codec audio operations, DAPM description and event handlers

Platform Class Driver

- platform dependent code, to configure the audio DMA engine driver, DAI drivers and DSP drivers if there are any
- DMA driver functionality described by *snd_pcm_ops* struct and exported via *snd_soc_component_driver* struct

- DAI drivers describe DAI functionality and operations
- DSP drivers describe DAPM graph, mixer controls and communication with DMA

Machine Class Driver

- links together all platform and codec drivers
- describes relationship between each component
- registers audio subsystem with the kernel as a platform device

2.3 AVB protocol family

The AVB protocol is a set of standards, which allows for synchronized realtime streaming of audio/video data within ethernet networks. Table 1 shows all sub protocols, which form the AVB specification. Some sub protocols have been superseded or incorporated into other protocols, which is marked by an (s). This project utilizes the protocol versions depicted in table 1.

IEEE 802.1 BA-2011	Audio Video Bridging (AVB) Systems
IEEE 802.1 Qav-2009 (s)	Forwarding and Queuing Enhancements for Time-Sensitive Streams (FQTSS)
IEEE 802.1 Qat-2010 (s)	Stream Reservation Protocol (SRP)
IEEE 802.1 AS-2011 (s)	Timing and Synchronization for Time-Sensitive Applications (gPTP)
IEEE 1722-2011 (s)	Layer 2 Transport Protocol for Time Sensitive Applications in a Bridged Local Area Network (AVTP)
IEEE 1722.1-2013	Device Discovery, Enumeration, Connection Management and Control Protocol(AVDECC)

Table 1: AVB protocol family

IEEE 802.1 BA-2011 describes the general setup of an AVB network and explains what the minimal implementation of an AVB bridge and end node has to look like, to transmit realtime media data.

IEEE 802.1 Q-2018 [9] incorporates both IEEE 802.1 Qav-2009 and IEEE 802.1 Qat-2010. IEEE 802.1 Qav-2009 is the Forwarding and Queuing Enhancements for Time-Sensitive Streams (FQTSS) protocol, which allows to set priorities for specific packages which are taken into account by the traffic shapers of AVB network bridges. IEEE 802.1 Qat-2010 on the other hand is the Stream Reservation Protocol (SRP), which reserves bandwith and resources between a talker (stream source) and a listener (stream sink) to guarantee the defined quality of service (QoS).

IEEE 802.1 AS-2011 is used to synchronize all media clocks in a given AVB network. The protocol allows for synchronization in the nano seconds range. IEEE 802.1 AS-2011 is superseded by IEEE 802.1 AS-2020 [13].

IEEE 1722-2011 describes the AVTP protocol, which allows for time synchronized media transmission. AVTP uses SRP, gPTP and FQTSS to reserve the required resources, define the media transmission format and to transmit the signals in a synchronized manner throughout the network. IEEE 1722-2011 is superseded by IEEE 1722-2016 [6].

IEEE 1722.1-2013 describes the AVDECC protocol, which allows an AVDECC controller to modify AVDECC entities within an AVB network. Additionally all devices in the network know which device is available and which functionality it provides. The following section give a deeper introduction to all sub protocols, which were implemented during the course of the project.

2.3.1 Generalized Precision Time Protocol (gPTP)

The gPTP protocol synchronizes clock sources within a given time-aware local area network (LAN). The standard defines a time-aware bridged LAN as a combination of multiple time-aware systems which are interconnected and implement the gPTP protocol. There are two types of time-aware systems:

Endstations: Can be either a grandmaster (provide synchronized time information), or a clock client (receive time information)

Bridges: If not a grandmaster, it receives time information, applies delay compensation and retransmits the information to all directly attached links

Multiple connected time-aware systems are defined as a gPTP domain [10].

Within a gPTP domain exists exactly one grandmaster which is chosen by the best master clock algorithm (BMCA). Every time-aware system with clock sourcing capabilities can be selected as a grandmaster [10].

Time synchronization between links in a gPTP domain is done by the grandmaster sending the current synchronized time in regular intervals to all directly attached time-aware systems. All time recipients correct the received time information by a calculated propagation delay and if the recipient is a bridge, the corrected time is sent to all directly attached time-aware systems [10].

To calculate the propagation delay between two links in a gPTP domain which are connected by full-duplex ethernet links the two step peer-to-peer (P2P) path delay algorithm is used. The process of the algorithm is shown in figure 1.

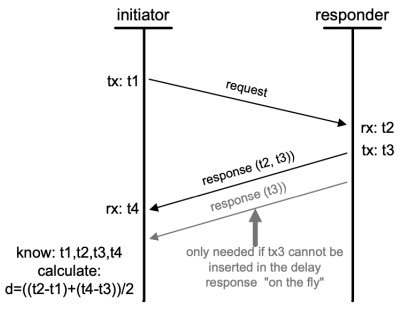

Figure 1: gPTP delay measurement, taken from [10]

The initiator sends a *Pdelay_Req* to the responder and records the egress timestamp (*t1*). Upon reception of the *Pdelay_Req*, the sender records the ingress timestamp (*t2*) and immediatly sends the *Pdelay_Resp* containing the ingress timestamp of *Pdelay_Req* (*t2*) and records the egress timestamp of *Pdelay_Resp* (*t3*). After sending the *Pdelay_Resp* the responder sends a *Pdelay_Resp_Follow_Up*

containing the egress timestamp of *Pdelay_ Resp* (*t3*). This is necessary, because timestamps are measured directly in the network interface card (NIC). When the initiator receives the *Pdelay_ Resp* it records the ingress timestamp (*t4*), after that it waits for the *Pdelay_ Resp_ Follow_ Up*. When the *Pdelay_ Resp_ Follow_ Up* arrived all recorded timestamps are used to calculate the mean path delay with the formula given in figure 1. This process is initiated between all links in a gPTP domain [10].

2.3.2 Forwarding and Queuing Enhancements for Time-Sensitive Streams (FQTSS)

The FQTSS protocol implements forwarding and queueing requirements of time-sensitive streams, so that time-sensitive data can be transmitted reliably. To achieve this FQTSS defines the following mechanisms [11]:

- detection of boundaries between an SRP domain and non-SRP capable bridges
- bandwidth availability parameters to determine the maximum bandwidth and the reserved bandwidth for a given queue in a Bridge
- traffic shaping algorithm, which shapes the traffic in accordance to the reserved bandwidth in a queue
- introduce a relationship between transmission payload size and actual consumed bandwith during transmission
- mapping between frame priority and port traffic class
- behavioral definition of a Talker end-station for time-sensitive data

2.3.3 Stream Reservation Protocol (SRP)

The SRP protocol utilizes different sub-protocols from IEEE 802.1 Q-2011 to reserve stream resources within a bridged network. SRP allows end-stations to reserve network resources that guarantee transmission and reception of data streams with a requested QoS. Those end-stations can be either Talkers (end-stations sending data streams) or Listeners (end-stations receiving data streams). Each stream which is advertised by one Talker can have one or multiple Listeners [14].

Talkers announce their streams by using *Talker Declarations*. Those declarations are sent as long as a stream is available through *Talker Advertise* messages. When a stream is no longer available and a Listener requests the stream a *Talker Failed* declaration is sent [14].

To request attachment to an advertised stream Listeners send *Listener Declarations*, which are classified into [14]:

Listener Ready: one or more Listeners are requesting a stream and resources are available

Listener Ready Failed: two or more Listeners request attachment to a stream and some listeners have sufficient resources but some do not

Listener Asking Failed: none of the requesting Listeners has sufficient resources available

The resulting mode of operation of the SRP protocol following Duraipandian [5] is:

1. Talker advertises stream
2. Along network path every Bridge checks if the required resources are available (required resources are available → propagate advertisement; required resources unavailable → report error back to Talker)
3. When Listener endpoint is reached the Listener can register for the stream using *Listener Declarations*

2.3.4 Audio Video Transfer Protocol (AVTP)

The AVTP protocol specifies the transmission of audio/video data across an AVB network. It specifies methods and transmission formats to transport media streams with accompanying timing information to allow for exact reproduction of the content at Listener devices. The transmission takes place on the media access layer (MAC) of the open systems interconnection model (OSI) [8].

All devices in a network that send, receive or forward AVTP packages have to implement gPTP, SRP and FQTSS to function properly. AVTP uses gPTP to provide a common synchronized time base across all Talker and Listener devices. SRP and FQTSS are used to provide reliable network delivery wihin a bounded latency [8].

2.3.5 Device Discovery, Enumeration, Connection Management and Control Protocol (AVDECC)

The AVDECC protocol is responsiple to allow end-stations in an AVB network to connect and operate between each other. This is achieved, by implementing four independent steps that can be used to form a system of end-stations which are able to interoperate with each other in a standardized way. The four individual steps are taken out by an AVDECC Controller which may exist in a device which is also an AVDECC Talker or Listener or even in a seperate device within the same network [7]. The four steps following "IEEE Standard for Device Discovery, Connection Management, and Control Protocol for IEEE 1722(TM) Based Devices" [7] are:

- **Discovery**: find AVDECC entities in the network
- **Enumeration**: find out about specific capabilities of an AVDECC entity
- **Connection Management**: connect one or more streams between two or more AVDECC entities
- **Control**: ability of the AVDECC Controller to adjust enumerated parameters of an AVDECC entity

All steps except for the connection management are optional, because this step is used to establish bandwidth usage and reservations across the AVB domain [7].

3 Development Environment

To simplify the development process, a Debian based Operating System is recommended because this allows to build the kernel as a Debian Package, which simplifies the installation routine. The machine used for the development process of this project ran Ubuntu 20.04. The following steps have been tested on this environment, so specific commands may be different for other distributions.

Before starting the installation process, a suitable base image needs to be downloaded for the BBB or BBAI. The base image used for BBB is *AM3358 Debian 10.3 2020-04-06 4GB SD IoT* and for BBAI *AM5729 Debian 10.3 2020-04-06 8GB SD IoT TIDL* is used. Both images can be downloaded at: Beagleboard Images. The downloaded image must be installed to a micro SD card using Balena Etcher. The following code listing describes the installation routine for all required programs to build a custom kernel:

```
sudo apt-get install build-essential git flex bison gcc-arm-linux-gnueabihf lzop
    ncurses-dev
```

When all required programs are installed, the kernel can be cloned from the kernel source:

```
git clone https://github.com/NiklasWan/linux.git
```

After checking out the development branch (*dev_gsoc_face_4.19-rt* for the CTAG Face 2|4 drivers and *submission_gsoc_avb* for the virtual AVB driver), the kernel can be configured and compiled using the following steps:

```
# first load Beagleboard base configuration

make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- bb.org_defconfig

# build the kernel as a debian package

make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- bindeb-pkg -j8

# copy all files to home directory of sd card
cd .. && cp linux-* /media/dev/rootfs/home/debian
```

When the kernel is compiled and copied to the SD card, eject the SD card and start the BBB (BBAI). After boot up, type in the following commands:

```
# install kernel headers
sudo dpkg -i linux-headers*

# install c library
sudo dpkg -i linux-libc*

# install kernel image
sudo dpkg -i linux-image*

# reboot system to load the new kernel version after reboot
sudo reboot
```

When the system is rebooted, the custom kernel is running. For specific compilation and installation instructions for both the CTAG Face 2|4 drivers and the virtual AVB ALSA driver the documentation page is recommended.

4 CTAG Face 2|4

The implementation of the CTAG Face 2|4 driver can be split up into two main parts: The device tree overlay (DTO) and the actual driver code implemented using the C language. The driver code has been implemented for both BBB and BBAI on kernel version 4.19-RT. The implementation, which was merged into the BeagleBoard.org Upstream via a Pull Request can be found on this GitHub branch. A description of the installation routine for the driver on both BBB and BBAI can be found under CTAG Face 2|4 driver documentation and a working image for BBAI can be found via this Gitlab link.

4.1 Device Tree Overlay

Device trees were introduced to the Linux kernel to factor out hardware dependent code of the Linux source. They describe the underlying hardware structure and all its provided interfaces. The device tree is handed over to the kernel at boot time by the bootloader.

The device tree overlay for CTAG Face 2|4 describes the pin configuration of the BBB and BBAI, the basic setup for the hardware codec and the sound card. The device tree overlay for both the BBB and the BBAI can be acquired, by cloning the following repository: Beagleboard DT Overlays.

4.1.1 Device Tree Architecture

Device trees are hardware descriptions of a specific board and are passed during boot into the kernel. To support capes on the BBB and BBAI, the hardware configuration can be overlaid to the base device tree of the board. This process is called device tree overlaying. It allows to change overlays dynamically, without the need to change the source code of the boards device tree. The boot loader just needs to overlay the dtbo over the base dtb, resulting in a device tree blob in memory, which can be passed into the kernel. In the following sections the most important parts of the overlay for BBB are described, with some less relevant sections omitted. The device tree for BBAI just differs in the configuration of the MCASP clock and differing pin values. The device tree for BBAI can be found in this repository

The device tree overlay is organized into fragments, each targeting one specific symbol of the base tree. The first configuration in the device tree for the CTAG Face is the SPI pin configuration. The following listing shows the most important parts of the targeted fragment:

```
...
spi_gpio: spi_gpio {
    ...
    sck-gpios = <&gpio0 11 0>; //P8.32
    mosi-gpios = <&gpio0 9 0>; //P8.33
    miso-gpios = <&gpio0 26 0>; //P8.14
    cs-gpios = <&gpio0 27 0 &gpio0 10 0>; //P8.17 / P8.31
    num-chipselects = <2>;
    ...
    pinctrl-0 = <&audiocard_spi_pins>;
    ...

    ad193x: ad193x@0{
```

```
14          ...
15          pinctrl-0 = <&ad1938_codec_reset_muxing>;
16          ...
17          reset-gpio = <&gpio2 17 0>;
18          ...
19      };
20  };
21  ...
```

Listing 1: SPI + Codec configuration

As you can see in listing 1 the pin headers for P8.32, P8.33, P8.14, P8.17 and P8.31 are configured for SPI usage. Because the MUX configuration of the BBB is not configurable for SPI on those pins, the MUX mode is set to GPIO. This definition requires the kernel to load the SPI bitbang driver. The Face uses SPI for the configuration of the AD1938 audio codec. The section from line 13 to line 19 adds a reset pin, which gets set to LOW and then to HIGH for starting the codec. This is done in the codec driver, which will be shown in section 4.2.

Listing 2 shows the configuration of the MCASP and the sound card.

```
1   ...
2   pinctrl-0 = <&mcasp0_pins>;
3   ...
4   op-mode = <0>;  /* MCASP_IIS_MODE */
5   tdm-slots = <8>;
6   num-serializer = <16>;
7   serial-dir = <  /* 0: INACTIVE, 1: TX, 2: RX */
8       2 0 1 0
9       0 0 0 0
10      0 0 0 0
11      0 0 0 0
12  >;
13  ...
14  sound {
15      compatible = "ctag,face-2-4";
16      ti,model = "CTAG face-2-4 8CH";
17      ti,audio-codec = <&ad193x>;
18      ti,mcasp-controller = <&mcasp0>;
19      ti,audiocard-tdm-slots = <8>;
20      ti,codec-clock-rate = <24576000>;
21      ti,cpu-clock-rate = <24576000>;
22      ...
23  };
24  ...
```

Listing 2: MCASP + sound card configuration

From line 2 to 12 the configuration of mcasp0 is shown. MCASP is an audio transmission peripheral found in TI processors that can operate amongst other protocols in I^2S mode, which is used by the AD1938 codec for audio transmission. Every serializer is configured to receive 8 channels. In the case of the CTAG Face two serializers are used. AXR0 is configured as audio input and AXR2 is configured as audio output. This is done via the matrix shown from line 8 to 11. For audio output all 8 channels

of AXR2 are used and for audio input just 4 channels of AXR0 are used because the CTAG Face card just provides 4 mono audio input channels.

The sound card configuration is done from line 14 to 21. The compatible definition leads the Linux kernel to load the according machine class driver, which will be described in section 4.2. The following lines link the audio codec and the MCASP controller to the sound card, which can be retrieved by the machine class driver at runtime. Also both, the codec and cpu clock rate, are set to 24.576 MHz.

4.2 Driver Code

The driver code for the CTAG Face implements the machine class driver of the ALSA SoC platform. The following sections show the required adjustments to port the driver to kernel version 4.19-RT.

4.2.1 Codec Class Driver Adjustments

To allow the codec to start, the reset pin has to be set to LOW and then to HIGH during startup. The following listing from the file *sound/soc/codecs/ad193x.c* shows the reset process of the codec.

```
1  static int ad193x_reset(struct snd_soc_component *component) { ... }
2
3  int ad193x_probe(struct device *dev, struct regmap *regmap, enum ad193x_type type)
4  {
5      ... if (of_match_device(ad193x_spi_dt_ids, dev))
6          ad193x->reset_gpio = of_get_named_gpio(dev->of_node, "reset-gpio", 0);
7
8      if (gpio_is_valid(ad193x->reset_gpio))
9      {
10         ret = devm_gpio_request(dev, ad193x->reset_gpio, "AD193x Reset");
11         if (ret < 0)
12             return ret;
13     }
14     ...
15 }
```

Listing 3: Codec class driver changes

Line 1 shows the definition of the reset function. This gets called in the *ad193x_component_probe* to allow for proper startup. Line 5 to line 13 show the process of the retrieval of the reset GPIO from the device tree which was described in section 4.1.

4.2.2 Machine Class Driver

The machine class driver glues the codec and component drivers together and defines the operations for the CTAG Face 2|4 card and is implemented in *sound/soc/ctag/davinci-ctag-face-2-4.c*. The machine class driver exposes the functionality of the sound card to the user space via the ALSA API. To achieve this, structs are instantiated, which hold function pointers to functions which implement the ALSA interface and its apropriate data.

First, there is the *snd_soc_ops* struct, which defines the stream operations. In this case, the *hw_params*, *startup* and *shutdown* functions are implemented. The *startup* and *shutdown* functions

prepare or unprepare the master clock respectively. The *hw_ params* function configures the codec and cpu clock for the requested substream.

The *platform_ driver* instance, called *snd_ davinci_ audiocard_ driver*, gets exposed to user space via the macro *module_ platform_ driver*. It holds the complete driver data and calls the *snd_ davinci_ audiocard _ probe* function when the device is attached and the *snd_ davinci_ audiocard_ remove* when the device is removed. The purpose of the probe function is to retrieve all data from the device tree, allocate the required memory in kernel space for the driver data and finally register the sound card to the system. The purpose of the remove function is to unregister the card again. The function which is called before the probe function is the *snd_ davinci_ audiocard_ init* function, which registers the DAPM functionality to the sound card and configures the TDM settings.

4.2.3 Features and Limitations

The CTAG Face 2|4 allows simultaneous playback of 8 audio channels and simultaneous recording of 4 audio channels at a sampling rate of $48\,kHz$. When a sampling rate other than $48\,kHz$ is required, the ALSA plugin module for samplerate conversion can be used. All this functionality was implemented during the course of the project on kernel version 4.19-RT on BBB and BBAI.

Because kernel 4.19-RT still behaves unstable on BBAI at the time of writing a port to kernel version 5.4-RT was attempted. The process however failed, because the SPI Bitbang drivers could not be loaded successfully. For future work the source code, which can be found in this repository can be inspected and modified to get the driver running on kernel 5.4-RT.

5 AVB protocol stack

For the implementation of the AVB protocol stack the source code of Duraipandian [5] was ported from kernel 4.4 to kernel 5.4-RT for both BBAI and BBB. The installation instructions for the driver, the gPTP daemon and the avbtest application can be found on the documentation page. The implementation can be found in this repository and pre-compiled images for both BBB and BBAI can be found on this release page. At the time of writing a Pull Request to the Beagleboard.org Upstream is still open to be merged into the 5.4-RT branch. The underlying implementation implements parts of the AVB stack in user space and parts in kernel space.

The following protocols have been implemented to achieve basic AVB functionality:

1. IEEE 802.1AS-2011 (gPTP) implemented in user space (superseded by IEEE 802.1AS-2020)
   (a) Only one ethernet port per device supported
   (b) No support for IEEE 802.11 (WIFI)
   (c) No seamless change of grand masters (changeover causes abrupt jump in synchronization time)
2. IEEE 802.1 Qat-2010 (SRP) implemented as part of the virtual ALSA AVB driver (incorporated into IEEE 802.1 Q-2018)
   (a) Supports just the reservations of one playback and capture stream in parallel
3. IEEE 1722.1-2013 (AVDECC) implemented as part of the virtual ALSA AVB driver
   (a) Supports just the responder role of the Device Discovery Protocol
   (b) No support for controller role
4. IEEE 1722-2011 (AVTP) implemented as part of the virtual ALSA AVB driver (superseded by IEEE 1722-2016)
   (a) 8 channels are supported for one stream with a maximum sampling rate of 192 KHz.

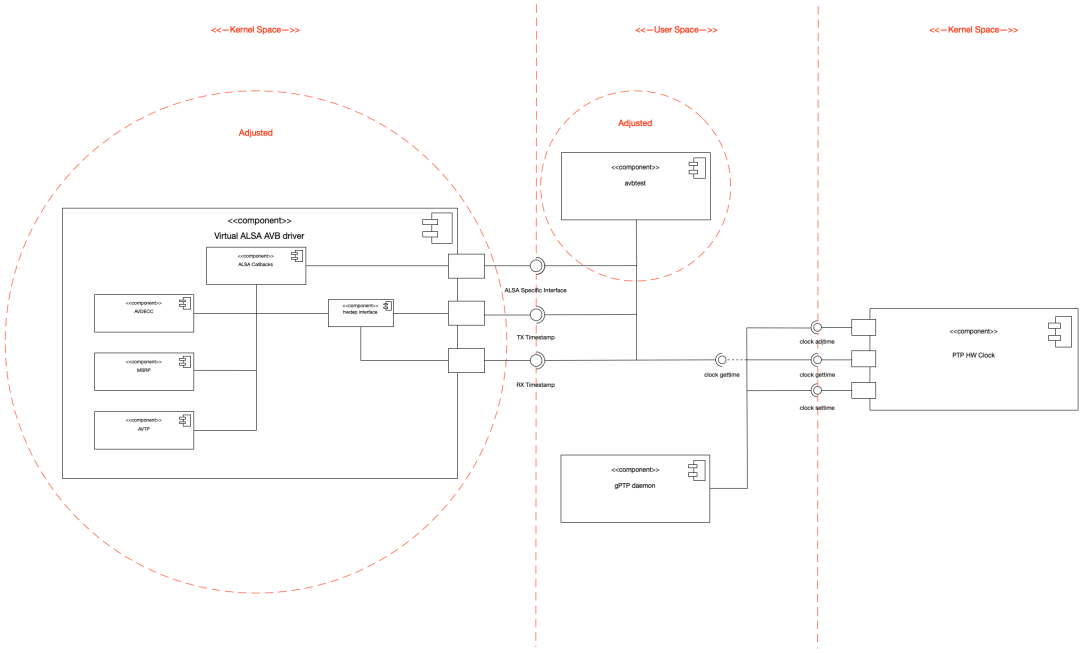

Figure 2: AVB implementation component diagram

Figure 2 shows a component diagram with all different components of the implementation. It can be seen, that besides gPTP the avbtest application is also running in user space. The avbtest app initiates stream transmission, invokes the AVB driver callbacks via the ALSA API and writes the received audio data to disk or plays it back via HDMI. It does so via well defined interfaces between the avbtest app, the gPTP daemon and the virtual AVB driver, both running in kernel space. All components which are marked by red dashed circles have been modified during the course of the project.

To improve the readability of the code base the whole driver code has been modularized into separate C language files. Each protocol, namely AVDECC, SRP and AVTP, now have separate header and source files. Common functionality and macros were exported to a common header and a file for the ALSA core driver code was introduced which implements the ALSA driver API for the virtual AVB driver and registers it with the audio subsystem. To seperate the AVB driver from other ALSA drivers, the implementation directory was changed to */linux/sound/drivers/avb*. Besides modularization, the source code also has been refactored to conform to the Linux kernel coding conventions.

5.1 gPTP User Space Daemon

The gPTP user space daemon implements the synchronization capabilities of the AVB protocol always running as a background process when started. The daemon requests the network adapter of the BBB (BBAI) to apply time stamps to incoming and outgoing ethernet packages and calculates link delays and determines a grand master clock within the AVB network. The source code for the daemon required no changes to run on the 5.4-RT kernel.

To allow for proper synchronization of AVB nodes each BBB (BBAI) must be able to access the

hardware clock of the underlying ethernet controller. The hardware clock is controlled by the PTP driver of the Linux kernel and can be accesed via */dev/ptp0*. All standard time operations exposed by the Linux time API can be applied to the clock.

With this hardware time being available to all devices, the underlying algortihms of gPTP can utilize this time to perform their calculations. The following sections describe the implementation of those underlying gPTP algorithms.

5.1.1 Peer-to-Peer Calculation

The peer-to-peer calculation is the process, where the delay of a node between its direct neighbors is calculated. The algorithm uses a state machine with the states *init*, *idle*, *delay_response* and *delay_followup*. The calculation of delay time is handled in *delaymsr.c*. The following listing shows the function, which implements the basic state machine, with one specific exemplary state workflow being explained.

```
1  void dmHandleEvent(struct gPTPd *gPTPd, int evtId)
2  {
3      ...
4      switch (gPTPd->dm.state)
5      {
6          ...
7          case DM_STATE_DELAY_RESP_WAIT :
8              switch (evtId)
9              {
10                 ...
11                 case GPTP_EVT_DM_PDELAY_REQ :
12                     getRxTS(gPTPd, &gPTPd->ts[4]);
13                     sendDelayResp(gPTPd);
14                     getTxTS(gPTPd, &gPTPd->ts[5]);
15                     sendDelayRespFlwUp(gPTPd);
16                 break;
17                 ...
18             }
19         break;
20         ...
21     }
22     ...
23 }
```

Listing 4: Peer to Peer Delay Algorithm

As one can see, the basic states are implemented using a switch statement, with each state having several substates. The code snippet shows the state machine in the *delay_response* state. When the event *GPTP_EVT_DM_PDELAY_REQ* is received, the algorithm performs the following steps from line 12 to line 15:

1. Get the timestamp of the incoming delay request message and save to gPTP struct.
2. Send the delay response.
3. Get the timestamp of the outgoing message and save to gPTP struct.

4. Send the delay follow up message with the timestamp of the outgoing message.

5. The receiver can then calculate the link delay between both peers.

5.1.2 Best Master Clock Selection

The Best Master Clock selection algorithm is a state machine based algorithm which is executed on all devices within the AVB domain. Every device sends out announce messages, until the grand master is found. The initial behavior of each node is to suspect that it is itself the grand master. This changes when a message with a better master clock or higher priority is received. The possible states of the algorithm are *init*, *grand_master* and *slave* and the algorithm is implemented in *bmc.c*.

The following listing shows the repeating announce messages sent out by the algorithm, when the node is in the *grand_master* state.

```
1  void bmcHandleEvent(struct gPTPd* gPTPd, int evtId)
2  {
3      ...
4    switch(gPTPd->bmc.state) {
5          ...
6      case BMC_STATE_GRAND_MASTER:
7        switch (evtId) {
8          case GPTP_EVT_STATE_ENTRY:
9            gptp_startTimer(gPTPd, GPTP_TIMER_ANNOUNCE_RPT, gPTPd->bmc.announceInterval,
       GPTP_EVT_BMC_ANNOUNCE_RPT);
10           csSetState(gPTPd, TRUE);
11           break;
12         ...
13         case GPTP_EVT_BMC_ANNOUNCE_RPT:
14           sendAnnounce(gPTPd);
15           break;
16                 ...
17       }
18       break;
19   }
20 }
```

Listing 5: Best Master Selection Algorithm

Line 8 to 11 show the entry callback, which is called when the device is selected as a master. This starts a timer, which triggers an *GPTP_EVT_BMC_ANNOUNCE_RP* event repeatedly in a specified interval. This event gets handled upon reception from line 13 to 15, where the algorithm sends out an announce message with its capabilities to all direct neighbors.

5.1.3 Time Synchronization

The time synchronization algorithm is also a state machine based implementation. It has the three states *init*, *grand_master* and *slave*. When in *grand_master* state the device sends out a sync message periodically. When the device is in *slave* mode, it waits for sync messages sent out by the master to adjust the internal clock to the grand master clock. The algorithm is implemented in *sync.c* and is built

in the same structure like the best master clock selection algorithm and the peer-to-peer calculation algorithm.

5.1.4 gPTP Kernel Space implementation

During the course of the project some time was invested to examine if it is possible to implement the gPTP daemon in kernel space. Implementing gPTP in kernel space would be an advantage over the implementation in user space because this would keep all AVB relevant code in kernel space and thus the implementation could be submitted to the Linux mainline kernel.

The implementation in kernel space however failed, because no possibility was found to access the *CMSGHDR* data, which holds the egress timestamps. This however is a vital part of the implementation, because the socket which requests an egress timestamp receives this information in Linux via the error queue in a *CMSGHDR* structure and this data structure was not filled with information in kernel space. The source code which implements this functionality can be found in this repository. The functions, *void get_rx_ts(struct gptp_instance* gptp, struct timespec64* ts)* and *void get_tx_ts(struct gptp_instance* gptp, struct timespec64* ts)* in the file *gptp_common.c* are causing the issues in accessing the CMSGHDR of a packet. Maybe future research could help in getting the implementation to run in kernel space.

5.2 Virtual ALSA driver

The virtual ALSA AVB driver is implemented using the ALSA driver API. The implementation as a virtual sound card allows the usage of the device in user space application using the ALSA user space API. For the implementation on kernel version 5.4-RT some parts of Duraipandian [5] had to be modified due to changes of the ALSA API.

5.2.1 Changelist

Changes in *snd_pcm_ops* description of playback capture device

Here the *.periods_min* parameter was changed, to adjust the minimum length of pcm interrupts. This was required due to some overrun issues while playing back and recording audio data with the AVB sound card. The following listing shows the resulting structure for the playback hardware, which is redundant to the capture hardware.

```
static struct snd_pcm_hardware avb_playback_hw = {
    .info = (SNDRV_PCM_INFO_INTERLEAVED |
             SNDRV_PCM_INFO_BLOCK_TRANSFER),
    .formats = SNDRV_PCM_FMTBIT_S16_LE,
    .rates = SNDRV_PCM_RATE_8000_192000,
    .rate_min = 8000,
    .rate_max = 192000,
    .channels_min = 1,
    .channels_max = 8,
    .buffer_bytes_max = 131072,
    .period_bytes_min = 16384,
    .period_bytes_max = 131072,
    .periods_min = 2,
```

```
    .periods_max = 4,
};
```

Changes in *snd_pcm_ops* copy function

Because of API changes between kernel version 4.4 and 5.4, the playback and capture copy function needed adjustments. One point is, that in kernel version 5.4 the count of data to copy and the position in the buffer are no longer passed in number of frames but instead the number of bytes are passed. So the parameters needed to be calculated in frames first to allow for expected behavior. The second point was, that the member of *snd_pcm_ops* which handles copying between user and kernel space now was renamed to *.copy_user*.

Utilization of kernel sockets

The socket communication was changed to kernel space sockets. All functions from *linux/net.h* prepended with *kernel_<operation>* are now being used.

5.2.2 Sound Card Architecture

The implementation for the ALSA sound card can be found in the directory *sound/drivers/avb*, with the ALSA API functions being implemented in *sound/drivers/avb/avb.c*. The sound card is made visible to the system as a platform device because the sound card is not interfacing with a real hardware device via e.g. PCI.

To register as a platform device two functions are assigned to the probe and remove fields of the *platform_driver* struct, *avb_probe* and *avb_remove*. When the AVB driver is loaded via *insmod* or *modprobe* the function marked with *__init* is called first. The ALSA AVB driver is initialized via the following function:

```
static int __init alsa_avb_init(void) {...}
```

This function registers the platform device, allocates memory for the AVB device structure, and initializes the working queues for the SRP and AVDECC protocol. This allows to run those protocols in the background and being processed in pre-defined intervals.

Because the AVB device is registered as a virtual platform device, the probe function gets called directly after the init function. The task of the probe function is to create the actual driver data, register the sound devices to the sound subsystem, attach the driver data to the platform driver, and initialize the socket for ethernet communication. The *avb_remove* function gets called when the AVB card is removed and frees the sound driver data. After that the following function gets called:

```
static void __exit alsa_avb_exit(void) {..}
```

This function cancels both, the SRP and the AVDECC queues, releases the acquired kernel memory, and finally unregisters the platform driver from the system.

To allow proper playback and capturing over the network, the AVTP protocol is used. This is implemented directly within the ALSA PCM callbacks, which are called when the playback or capture process is started. The following structure shows the definition of the capture operations, which are inversely correlated with the playback options:

```
static struct snd_pcm_ops avb_capture_ops = {
  .open =   avb_capture_open,
  .close =  avb_capture_close,
  .ioctl =  snd_pcm_lib_ioctl,
  .hw_params =  avb_capture_hw_params,
  .hw_free =  avb_capture_hw_free,
  .prepare =  avb_capture_prepare,
  .trigger =  avb_capture_trigger,
  .pointer =  avb_capture_pointer,
  .copy_user =  avb_capture_copy
};
```

The *avb_capture_open* is called before the playback is started. The function assigns the capture hardware description to the according substream. The *avb_capture_prepare* function is a dummy function and the *avb_capture_trigger* function is called when playback state changes e.g. from START to STOP. The *avb_capture_pointer* function is called by the ALSA middle-layer to retrieve the current position within the audio buffer and the *avb_capture_copy* is used to copy the audio data from kernel to user space. The *avb_capture_hw_params* is the function where the audio buffer for the substream is allocated, the resources for the AVTP protocol are allocated and where the working queue for the AVTP protocol is started. The *avb_capture_hw_free* function frees the allocated memory when playback is finished and cancels the AVTP working queue.

5.2.3 Abstraction of PTP HW clock

Some research was conducted in how to access the PTP hardware clock of the NIC. On TI devices the hardware clock is implemented in the CPTS (Common Platform Time Sync) module. To abstract away access to the hardware clock for other NIC's an abstraction layer was implemented, which can be found in this repository.

The abstraction is done in *avb_hwclock.h* and *avb_hwclock.c*, *ti_hwclock.c* implements this interface for TI devices. At first it was planned to implement synchronization of different audio streams in kernel space but due to the fact that playback of audio via different audio devices is required the implementation is not possible as sound devices are not exposed in kernel space. Because of that the PTP hardware clock in kernel space is not included in the final submission.

5.3 AVB User Space Test Application

The AVB user space test application provides the interface to the virtual ALSA driver and allows for testing the device. The source code which was taken from Duraipandian [5] had to be modified to allow for streaming the audio data across two devices. Because the CTAG Face 2|4 drivers are not working for kernel version 5.4-RT, the avbtest application was simplified to support two use cases:

1. **Use Case: Stream audio data from BBB to BBAI and save file to disk:**
   Steps on receiver device:

   ```
   sudo ./avbtest -r -c<number_of_channels> -l<log_level> -s<sampling_rate> <
       name_of_file_to_be_saved>.wav
   ```

   Steps on playback device:

```
sudo ./avbtest -p -c<number_of_channels> -l<log_level> -s<sampling_rate> <
    name_of_file_to_be_played_back>.wav
```

2. **Use Case: Stream audio data from BBB to BBAI and playback via HDMI**
   Steps on receiver device:

```
sudo ./avbtest -y -c<number_of_channels> -l<log_level> -s<sampling_rate> <
    dummy_name>.wav
```

   Steps on playback device:

```
sudo ./avbtest -x -c<number_of_channels> -l<log_level> -s<sampling_rate> <
    name_of_file_to_be_played_back>.wav
```

The interface definition for audio output via HDMI is hardcoded into the application, so for use case 2 the setup is fixed with the BBB being the Talker device and the BBAI being the Listener device.

Before starting the streaming process the following steps need to be followed on both the Talker and the Listener device.

```
1.)    Start gPTP daemon: sudo ./gptpd
2.)    Load hwdep module: sudo modprobe snd-hwdep
3.)    Load AVB module: sudo modprobe snd-avb
4.)    When using provided images: sudo insmod snd-avb.ko
```

When the Talker device finished streaming, the avbtest app had the problem that driver resources were released before sending the last frame of audio data over the network. This led to the problem, that not the whole audio file was transmitted. To fix this problem, the Talker device now sleeps for one second before releasing all resources. This leads to a flawless transmission of the audio file. Figure 3 shows the sent and received audio files from an AVB transmission via use case 1. It can be seen, that the file which was received (at the bottom of the figure) is exactly the same as the file which was sent (at the top of the figure).

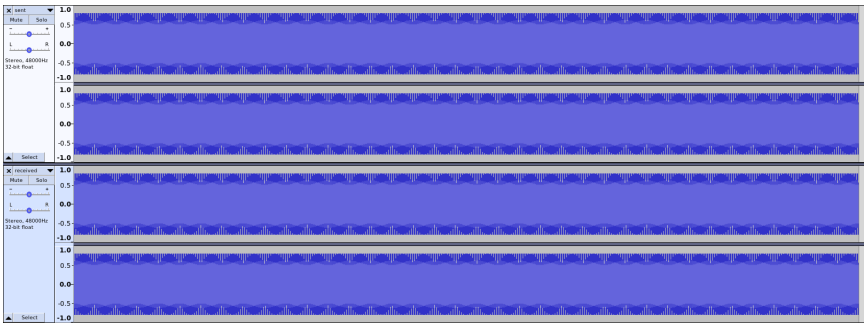

Figure 3: Waveforms of transmitted and received audio data

5.4 Limitations

The AVB implementation, which was successfully ported and refactored from Duraipandian [5] has the following limitations:

- An audio file has to be a multiple of the ALSA period size in length to allow for proper transmission, otherwise the file would not be transmitted completely.
- Transmission of one stream with up to 8 channels is possible.
- Due to a bug transmission of the avbtest application is not properly synchronized.
- Transmission between one Listener (BBB) and one Talker (BBAI) with two use cases is possible.
- AVDECC identification on an Apple MAC with proprietary AVB soulution of Beagleboard Device using ALSA AVB driver is not working properly (However the avbdiagnose application is able to identify the Beagleboard device see Appendix A for an exemplary log).

6 Conclusion

The implemented AVB protocol stack allows for transmission of audio data between a BBB and a BBAI. Both devices are synchronized within the nanoseconds range via the gPTP protocol, however synchronized transmission is currently not possible due to a bug in the avbtest application. The CTAG Face driver is working on BBB and BBAI and audio in-/output is possible via the ALSA user space API.

Howether, due to some difficulties in existing documentation and lack of time not all project goals were achieved. The major issue is that there was not enough time to implement a properly working version of the AVDECC protocol. This would allow communication between an Apple MAC using a proprietary AVB solution with a BBB or BBAI running the implemented AVB stack and control the operation from there. Possible Future Directions to improve the AVB implementation therefore could be:

- Implement gPTP in kernel space to allow for Mainline kernel submission
- Implement CTAG Face 2|4 drivers on Linux kernel 5.4-RT
- Implement a properly working version of the AVDECC protocol
- Implement MSRP to allow for reservation of multiple streams
- Fix avbtest bugs to allow for synchronized transmission of streams
- Implement functionality to allow BBB and BBAI to be used in a complex AVB domain with multiple devices
- Implement the new protocol versions for gPTP and AVTP

References

[1] *ALSA SoC Layer Overview — The Linux Kernel documentation*. URL: https://www.kernel.org/doc/html/v4.10/sound/soc/overview.html (visited on 07/05/2020).

[2] A. Belloni. *ASoC: Supporting Audio on an Embedded Board*. Berlin, 2016. URL: https://elinux.org/images/b/b5/Belloni-alsa-asoc.pdf.

[3] J. Corbet et al. *Linux device drivers*. 3rd ed. Beijing ; Sebastopol, CA: O'Reilly, 2005. ISBN: 978-0-596-00590-0.

[4] S. Dimitrov and S. Serafin. "Minivosc - a minimal virtual oscillator driver for ALSA (Advanced Linux Sound Architecture)". English. In: *Proceedings of the Linux Audio Conference 2012*. Linux Audio Conference 2012 ; Conference date: 12-04-2012 Through 15-04-2012. CCRMA, Stanford University, Apr. 2012, pp. 175–182. ISBN: 978-1-105-62546-6.

[5] I. Duraipandian. "Synchronized real time audio streaming over ethernet in embedded systems". MA thesis. Kiel: Kiel University of Applied Sciences, Jan. 2018. URL: https://www.creative-technologies.de/wp-content/uploads/2018/01/Synchronized_real_time_audio_streaming_over_Ethernet_in_embedded_systems-33.pdf.

[6] "IEEE Standard for a Transport Protocol for Time-Sensitive Applications in Bridged Local Area Networks". In: *IEEE Std 1722-2016 (Revision of IEEE Std 1722-2011)* (2016), pp. 1–233.

[7] "IEEE Standard for Device Discovery, Connection Management, and Control Protocol for IEEE 1722(TM) Based Devices". In: *IEEE Std 1722.1-2013* (2013), pp. 1–366.

[8] "IEEE Standard for Layer 2 Transport Protocol for Time Sensitive Applications in a Bridged Local Area Network". In: *IEEE Std 1722-2011* (2011), pp. 1–65.

[9] "IEEE Standard for Local and Metropolitan Area Network–Bridges and Bridged Networks". In: *IEEE Std 802.1Q-2018 (Revision of IEEE Std 802.1Q-2014)* (2018), pp. 1–1993.

[10] "IEEE Standard for Local and Metropolitan Area Networks - Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks". In: *IEEE Std 802.1AS-2011* (2011), pp. 1–292.

[11] "IEEE Standard for Local and metropolitan area networks– Virtual Bridged Local Area Networks Amendment 12: Forwarding and Queuing Enhancements for Time-Sensitive Streams". In: *IEEE Std 802.1Qav-2009 (Amendment to IEEE Std 802.1Q-2005)* (2010), pp. 1–72.

[12] "IEEE Standard for Local and metropolitan area networks–Audio Video Bridging (AVB) Systems". In: *IEEE Std 802.1BA-2011* (2011), pp. 1–45.

[13] "IEEE Standard for Local and Metropolitan Area Networks–Timing and Synchronization for Time-Sensitive Applications". In: *IEEE Std 802.1AS-2020 (Revision of IEEE Std 802.1AS-2011)* (2020), pp. 1–421.

[14] "IEEE Standard for Local and metropolitan area networks–Virtual Bridged Local Area Networks Amendment 14: Stream Reservation Protocol (SRP)". In: *IEEE Std 802.1Qat-2010 (Revision of IEEE Std 802.1Q-2005)* (2010), pp. 1–119.

[15] H. Langer. “Linuxbasiertes Mehrkanal-Audiosystem mit niedriger Latenz”. German. Bachelor’s Thesis. Kiel: Kiel University of Applied Sciences, Dec. 2015. URL: https://www.creative-technologies.de/wp-content/uploads/2016/02/Linuxbasiertes_Mehrkanal-Audiosystem_mit_niedriger_Latenz.pdf.

[16] A. Liberal de los Ríos. *Linux driver development for embedded processors: Learn to develop Linux embedded drivers with kernel 4.9 LTS*. eng. Second edition. ISBN: 978-1-72932-182-9.

[17] R. Love. *Linux kernel development*. 3rd ed. Developer’s library : essential references for programming professionals. OCLC: ocn268788260. Upper Saddle River, NJ: Addison-Wesley, 2010. ISBN: 978-0-672-32946-3.

[18] D. Phillips. *A User’s Guide to ALSA | Linux Journal*. Jan. 2012. URL: https://web.archive.org/web/20120109014951/http://www.linuxjournal.com/node/8234/print (visited on 09/25/2020).

[19] *The Linux Kernel documentation*. URL: https://www.kernel.org/doc/html/latest/index.html (visited on 03/26/2020).

Appendices

A Avbdiagnose Log

```
avbdiagnose 850.1

Captured at 2020-09-26-00-02-55-CEST [2020-09-25 22:02:55 +0000]

OS: Version 10.15.5 (Build 19F101)

Installed Versions
IOEthernetAVBController: 1.1.0 (1.1.0) [Loaded UUID 87D8C7D0-186F-3688-A295-89
    AD0A3E24BE]
AppleBCM5701Ethernet: 10.3.5 (10.3.5) [Loaded UUID AB48227C-65FD-3F48-B730-
    B29C0704CBEA]
IOTimeSyncFamily: 840.3 (840.3) [Loaded UUID 9F78DA80-945A-35B8-8230-8E3FE182C80A]
IOgPTPPlugin: 840.3 (840.3) [Loaded UUID 2EF0E705-399C-3DA6-AD8F-843994699648]
TimeSync: 840.3 [On Disk UUID D485273B-FE76-3AFB-8778-C6DB67B5E23C]
IOAVBFamily: 850.1 (850.1) [Loaded UUID AC8A33C1-6C74-37C5-94E7-A00EEA55A8ED]
IOAVBDiscoveryPlugin: 850.1 (850.1) [Loaded UUID F2F370D5-DB92-3D34-B3F8-28F7B4D0E006]
IOAVBStreamingPlugin: 850.1 ((null)) [Loaded UUID
    00000000-0000-0000-0000-000000000000]
IOAVBControlPlugin: 850.1 (850.1) [Loaded UUID F16C8790-F871-3961-A1DD-7B840CB6BB24]
IOMRPPlugin: 850.1 ((null)) [Loaded UUID 00000000-0000-0000-0000-000000000000]
AudioVideoBridging: 850.1 [On Disk UUID B4F298A9-4290-33B1-9DCE-41782B32A1C7]
AudioServerDriver: 540.1 [On Disk UUID 4F746345-AE43-3001-A0B8-D7AB5DCCE668]
AppleAVBAudio: 850.1 [On Disk UUID 3924C893-9240-30EC-984E-920A65478C66]

|+AirPort_Brcm4360
|    gPTP Present: Could not read property
|    Link Valid: YES
|    Link Active: YES
|    MAC Address: XX:XX:XX:XX:XX:XX
|    Link Speed: 217000000 bps
|    |+AirPort_Brcm4360_Interface
|    |    Interface is not AVB Capable
|    |    Interface: en1
|    |    Wants MSRP: Could not read property
|    |    Wants MVRP: Could not read property
|    |    Wants ADP: Could not read property
|    |    Wants ACMP: Could not read property
|    |    Wants AECP: Could not read property
|    |    Wants MAAP: Could not read property

|+BCM5701Enet
|    gPTP Present: YES
|    Link Valid: YES
|    Link Active: YES
|    MAC Address: XX:XX:XX:XX:XX:XX
|    Link Speed: 1000000000 bps
|    |+IOEthernetAVBInterface
```

```
|    |    Interface has AVB Enabled
|    |    Interface: en0
|    |    Wants MSRP: Could not read property
|    |    Wants MVRP: Could not read property
|    |    Wants ADP: YES
|    |    Wants ACMP: Could not read property
|    |    Wants AECP: Could not read property
|    |    Wants MAAP: Could not read property
|    |    |+IOAVB17221EntityDiscovery
|    |    |    Interface: en0
|    |    |    Entities:
|    |    |        |+IOAVB17221RemoteEntity
|    |    |        |    Time To Live: 57
|    |    |        |    Entity ID: 0x5051a9fffe9bd26a
|    |    |        |    Entity Model ID: 0x5051a99bd26a0001
|    |    |        |    Entity Capabilities: 0x00008508
|    |    |        |    Talker Stream Sources: 1
|    |    |        |    Talker Capabilities: 0x4001
|    |    |        |    Listener Stream Sinks: 1
|    |    |        |    Listener Capabilities: 0x4001
|    |    |        |    Controller Capabilities: 0x00000000
|    |    |        |    Available Index: 8
|    |    |        |    gPTP Grandmaster Identity: 0xa860b6fffe14a63f
|    |    |        |    gPTP Domain Number: 0
|    |    |        |    Association ID: 0x0000000000000000
|    |    |        |    MAC Addresses:
|    |    |        |        XX:XX:XX:XX:XX:XX
|    |    |        |    Identify Control Index: 0x0000
|    |    |        |    Interface Index: 0x0000
|    |    |+IOTimeSyncEthernetAVBInterfaceAdapter
|    |    |    Interface: en0
|    |    |    |+IOTimeSyncEthernetPort
|    |    |    |    Port Number: 1
|    |    |    |    Port Role: Could not read property
|    |    |    |    Interface: en0
|    |    |    |    Announce Clock Priority 1: 250
|    |    |    |    Announce Clock Class: 248
|    |    |    |    Announce Clock Accuracy: 33
|    |    |    |    Announce Offset Scaled Log Variance: 17258
|    |    |    |    Announce Clock Priority 2: 236
|    |    |    |    Announce Grandmaster Identity: 11ddb1c4731a0000
|    |    |    |    Announce Steps Removed: 0
|    |    |    |    Announce Time Source: 160
|    |    |    |    Received Clock Priority 1: 250
|    |    |    |    Received Clock Class: 248
|    |    |    |    Received Clock Accuracy: 254
|    |    |    |    Received Offset Scaled Log Variance: 16640
|    |    |    |    Received Clock Priority 2: 250
|    |    |    |    Received Grandmaster Identity: 5051a9fffe9bd26a
|    |    |    |    Received Steps Removed: 1
|    |    |    |    Received Time Source: 0
|    |    |    |    Local Link Type: AVB Ethernet
```

```
|    |    |    |    Remote Link Type: Unknown
|    |    |    |    Local Timestamping Mode: Hardware
|    |    |    |    Remote Timestamping Mode: Unknown
|    |    |    |    Local Frequency Tolerance: Could not read property
|    |    |    |    Remote Frequency Tolerance: Could not read property
|    |    |    |    Local Frequency Stability: Could not read property
|    |    |    |    Remote Frequency Stability: Could not read property
|    |    |    |    PTP Port Enabled: YES
|    |    |    |    ASCapable: NO
|    |    |    |    Propagation Delay Limit: 1000
|    |    |    |    Propagation Delay: 0
|    |    |    |    Maximum Propagation Delay: 0
|    |    |    |    Minimum Propagation Delay: 0
|    |    |    |    Maximum Raw Delay: 0
|    |    |    |    Minimum Raw Delay: 0
|    |    |    |    Remote Clock Identity: 0x0000000000000000
|    |    |    |    Remote Port Number: 0
|    |    |    |    Remote Is Same Device: NO
|    |    |    |    Multiple Remotes: NO
|    |    |    |    Measuring PDelay: NO
|    |    |    |    Local Announce Log Mean Interval: 0
|    |    |    |    Remote Announce Log Mean Interval: 1
|    |    |    |    Local Sync Log Mean Interval: -3
|    |    |    |    Remote Sync Log Mean Interval: 0
|    |    |    |    Local PDelay Log Mean Interval: 0
|    |    |    |    Remote PDelay Log Mean Interval: 1
|    |    |    |    Received PDelay Request Count:              3
|    |    |    |    Transmitted PDelay Response Count:          3
|    |    |    |    Transmitted PDelay Response Followup Count: 3
|    |    |    |    Attempted PDelay Response Count:          3
|    |    |    |    Attempted PDelay Response Followup Count: 3
|    |    |    |    Transmitted PDelay Request Count:        57
|    |    |    |    Attempted PDelay Request Count:        57
|    |    |    |    Received PDelay Response Count:          27
|    |    |    |    Received PDelay Response Followup Count: 26
|    |    |    |    Received Sync Count: 27
|    |    |    |    Received Followup Count: 27
|    |    |    |    Received Announce Count: 27
|    |    |    |    Received Signal Count: 0
|    |    |    |    Received Packet Discard Count: 54
|    |    |    |    Sync Receipt Timeout Count: 0
|    |    |    |    Announce Receipt Timeout Count: 0
|    |    |    |    Allowed Lost Responses Exceeded Count: 0
|    |    |    |    Transmitted Sync Count: 0
|    |    |    |    Transmitted Followup Count: 0
|    |    |    |    Transmitted Announce Count: 0
|    |    |    |    Transmitted Signal Count: 0
|    |    |    |    Attempted Sync Count: 0
|    |    |    |    Attempted Followup Count: 0
|    |    |    |    Attempted Announce Count: 0
|    |    |    |    Attempted Signal Count: 0
|    |    |    |    Raw Delay Exceeded Count: 0
```

```
|    |    |    |    Raw Delay Measurement Count: 0
|    |+IOTimeSyncEthernetAVBControllerAdapter
|    |    Wants Time Sync: YES
|    |    Time Sync Required: YES
|    |    |+IOTimeSyncEthernetNICClock
|    |    |    Clock Identifier: 0x11ddb1c4731a0006
|    |    |    Lock State: Locked

|+AppleThunderboltIPPort
|    gPTP Present: Could not read property
|    Link Valid: YES
|    Link Active: NO
|    MAC Address: XX:XX:XX:XX:XX:XX
|    Link Speed: 10000000000 bps
|    |+IOEthernetInterface
|    |    Interface is not AVB Capable
|    |    Interface: en2
|    |    Wants MSRP: Could not read property
|    |    Wants MVRP: Could not read property
|    |    Wants ADP: Could not read property
|    |    Wants ACMP: Could not read property
|    |    Wants AECP: Could not read property
|    |    Wants MAAP: Could not read property

|+IOTimeSyncClockManager
|    Mach Clock Identifier: 0x10ddb1c4731a0000
|    Mach CoreAudio Clock Domain: 0x63690000
|    |+IOTimeSyncgPTPManager
|    |    System Domain Identifier: 0x11ddb1c4731a0005
|    |    Domains:
|    |        |+IOTimeSyncDomain
|    |        |    Clock Identifier: 0x11ddb1c4731a0005
|    |        |    Lock State: Locked
|    |        |    Grandmaster Clock Identity: 0x11ddb1c4731a0005
|    |        |    gPTP Path:
|    |        |        0x11ddb1c4731a0005
|    |        |    Ports:
|    |        |        |+IOTimeSyncMachNanoPort
|    |        |        |    Port Number: 0
|    |        |        |    Port Role: Slave
|    |        |        |    Announce Clock Priority 1: 250
|    |        |        |    Announce Clock Class: 248
|    |        |        |    Announce Clock Accuracy: 33
|    |        |        |    Announce Offset Scaled Log Variance: 17258
|    |        |        |    Announce Clock Priority 2: 236
|    |        |        |    Announce Grandmaster Identity: 11ddb1c4731a0005
|    |        |        |    Announce Steps Removed: 0
|    |        |        |    Announce Time Source: 160
|    |        |        |    Received Clock Priority 1: 250
|    |        |        |    Received Clock Class: 248
|    |        |        |    Received Clock Accuracy: 33
|    |        |        |    Received Offset Scaled Log Variance: 17258
```

```
|   |       |       |   Received Clock Priority 2: 236
|   |       |       |   Received Grandmaster Identity: 11ddb1c4731a0005
|   |       |       |   Received Steps Removed: 0
|   |       |       |   Received Time Source: 160
|   |       |       |   Battery Powered: YES
|   |       |       |   External Power Connected: NO
|   |       |       |   Has Wired Ethernet Link Active: YES
|   |       |       |   Has Ethernet Hardware Timestamping: YES
|   |       |       |   Has WiFi Hardware Timestamping: NO
|   |       |+IOTimeSyncDomain
|   |       |   Clock Identifier: 0x11ddb1c4731a0000
|   |       |   Lock State: Locked
|   |       |   Grandmaster Clock Identity: 0x11ddb1c4731a0000
|   |       |   gPTP Path:
|   |       |       0x11ddb1c4731a0000
|   |       |   Ports:
|   |       |       |+IOTimeSyncMachNanoPort
|   |       |       |   Port Number: 0
|   |       |       |   Port Role: Slave
|   |       |       |   Announce Clock Priority 1: 250
|   |       |       |   Announce Clock Class: 248
|   |       |       |   Announce Clock Accuracy: 33
|   |       |       |   Announce Offset Scaled Log Variance: 17258
|   |       |       |   Announce Clock Priority 2: 236
|   |       |       |   Announce Grandmaster Identity: 11ddb1c4731a0000
|   |       |       |   Announce Steps Removed: 0
|   |       |       |   Announce Time Source: 160
|   |       |       |   Received Clock Priority 1: 250
|   |       |       |   Received Clock Class: 248
|   |       |       |   Received Clock Accuracy: 33
|   |       |       |   Received Offset Scaled Log Variance: 17258
|   |       |       |   Received Clock Priority 2: 236
|   |       |       |   Received Grandmaster Identity: 11ddb1c4731a0000
|   |       |       |   Received Steps Removed: 0
|   |       |       |   Received Time Source: 160
|   |       |       |   Battery Powered: YES
|   |       |       |   External Power Connected: NO
|   |       |       |   Has Wired Ethernet Link Active: YES
|   |       |       |   Has Ethernet Hardware Timestamping: YES
|   |       |       |   Has WiFi Hardware Timestamping: NO
|   |       |       |+IOTimeSyncEthernetPort
|   |       |       |   Port Number: 1
|   |       |       |   Port Role: Could not read property
|   |       |       |   Interface: en0
|   |       |       |   Announce Clock Priority 1: 250
|   |       |       |   Announce Clock Class: 248
|   |       |       |   Announce Clock Accuracy: 33
|   |       |       |   Announce Offset Scaled Log Variance: 17258
|   |       |       |   Announce Clock Priority 2: 236
|   |       |       |   Announce Grandmaster Identity: 11ddb1c4731a0000
|   |       |       |   Announce Steps Removed: 0
|   |       |       |   Announce Time Source: 160
```

```
|    |        |        |    Received Clock Priority 1: 250
|    |        |        |    Received Clock Class: 248
|    |        |        |    Received Clock Accuracy: 254
|    |        |        |    Received Offset Scaled Log Variance: 16640
|    |        |        |    Received Clock Priority 2: 250
|    |        |        |    Received Grandmaster Identity: 5051a9fffe9bd26a
|    |        |        |    Received Steps Removed: 1
|    |        |        |    Received Time Source: 0
|    |        |        |    Local Link Type: AVB Ethernet
|    |        |        |    Remote Link Type: Unknown
|    |        |        |    Local Timestamping Mode: Hardware
|    |        |        |    Remote Timestamping Mode: Unknown
|    |        |        |    Local Frequency Tolerance: Could not read property
|    |        |        |    Remote Frequency Tolerance: Could not read property
|    |        |        |    Local Frequency Stability: Could not read property
|    |        |        |    Remote Frequency Stability: Could not read property
|    |        |        |    PTP Port Enabled: YES
|    |        |        |    ASCapable: NO
|    |        |        |    Propagation Delay Limit: 1000
|    |        |        |    Propagation Delay: 0
|    |        |        |    Maximum Propagation Delay: 0
|    |        |        |    Minimum Propagation Delay: 0
|    |        |        |    Maximum Raw Delay: 0
|    |        |        |    Minimum Raw Delay: 0
|    |        |        |    Remote Clock Identity: 0x0000000000000000
|    |        |        |    Remote Port Number: 0
|    |        |        |    Remote Is Same Device: NO
|    |        |        |    Multiple Remotes: NO
|    |        |        |    Measuring PDelay: NO
|    |        |        |    Local Announce Log Mean Interval: 0
|    |        |        |    Remote Announce Log Mean Interval: 1
|    |        |        |    Local Sync Log Mean Interval: -3
|    |        |        |    Remote Sync Log Mean Interval: 0
|    |        |        |    Local PDelay Log Mean Interval: 0
|    |        |        |    Remote PDelay Log Mean Interval: 1
|    |        |        |    Received PDelay Request Count:              3
|    |        |        |    Transmitted PDelay Response Count:          3
|    |        |        |    Transmitted PDelay Response Followup Count: 3
|    |        |        |    Attempted PDelay Response Count:          3
|    |        |        |    Attempted PDelay Response Followup Count: 3
|    |        |        |    Transmitted PDelay Request Count:        57
|    |        |        |    Attempted PDelay Request Count:        57
|    |        |        |    Received PDelay Response Count:          27
|    |        |        |    Received PDelay Response Followup Count: 26
|    |        |        |    Received Sync Count: 27
|    |        |        |    Received Followup Count: 27
|    |        |        |    Received Announce Count: 27
|    |        |        |    Received Signal Count: 0
|    |        |        |    Received Packet Discard Count: 54
|    |        |        |    Sync Receipt Timeout Count: 0
|    |        |        |    Announce Receipt Timeout Count: 0
|    |        |        |    Allowed Lost Responses Exceeded Count: 0
```

```
|    |        |        |    Transmitted Sync Count: 0
|    |        |        |    Transmitted Followup Count: 0
|    |        |        |    Transmitted Announce Count: 0
|    |        |        |    Transmitted Signal Count: 0
|    |        |        |    Attempted Sync Count: 0
|    |        |        |    Attempted Followup Count: 0
|    |        |        |    Attempted Announce Count: 0
|    |        |        |    Attempted Signal Count: 0
|    |        |        |    Raw Delay Exceeded Count: 0
|    |        |        |    Raw Delay Measurement Count: 0

--------------------------------------------------------------------------------

IOAVBNub
    EntityID: 0x10ddb1c4731a0000


--------------------------------------------------------------------------------

Discovered Entities
    Entity 0x<Mac-addr>:
        Wrote /tmp/avbdiagnose-2020-09-26-00-02-55-CEST/ALSA_AVB_Driver-0x<Mac-addr>.
    aemxml



--------------------------------------------------------------------------------

Audio Driver State

AppleAVBAudio
+AudioPlugin
|    Object ID: 34
|    Base Class: aobj
|    Object Class: aplg
|    Owner ID: 1
|    Bundle ID: com.apple.audio.AppleAVBAudio
|    Resource Bundle URL: (null)
|    Manufacturer: (null)


AppleTimeSyncAudioClock
+AudioPlugin
|    Object ID: 35
|    Base Class: aobj
|    Object Class: aplg
|    Owner ID: 1
|    Bundle ID: com.apple.audio.AppleTimeSyncAudioClock
|    Resource Bundle URL: (null)
|    Manufacturer: (null)
|    Clock Device Objects:
|        +AudioClockDevice
```

```
|         |    Object ID: 41
|         |    Base Class: aobj
|         |    Object Class: aclk
|         |    Owner ID: 1
|         |    Name: TimeSync Clock 0x11ddb1c4731a0005
|         |    Manufacturer: Apple Inc.
|         |    Device UID: ATSAC:11ddb1c4731a0005
|         |    Transport Type: atac
|         |    Clock Domain: 0x63690001
|         |    Is Active: YES
|         |    Is Running: NO
|         |    Nominal Sample Rate: 48000.000000
|         |    Available Nominal Sample Rates:
|         |        Rate[0]: 44100.000000
|         |        Rate[1]: 44100.000000
|         |        Rate[2]: 48000.000000
|         |        Rate[3]: 48000.000000
|         |        Rate[4]: 88200.000000
|         |        Rate[5]: 88200.000000
|         |        Rate[6]: 96000.000000
|         |        Rate[7]: 96000.000000
|         |        Rate[8]: 176400.000000
|         |        Rate[9]: 176400.000000
|         |        Rate[10]: 192000.000000
|         |        Rate[11]: 192000.000000
|         |    Is Hidden: NO
|         |    Zero Timestamp Period: 0
|         |    Controls:
|         +AudioClockDevice
|         |    Object ID: 42
|         |    Base Class: aobj
|         |    Object Class: aclk
|         |    Owner ID: 1
|         |    Name: TimeSync Clock 0x11ddb1c4731a0000
|         |    Manufacturer: Apple Inc.
|         |    Device UID: ATSAC:11ddb1c4731a0000
|         |    Transport Type: atac
|         |    Clock Domain: 0x63690002
|         |    Is Active: YES
|         |    Is Running: NO
|         |    Nominal Sample Rate: 48000.000000
|         |    Available Nominal Sample Rates:
|         |        Rate[0]: 44100.000000
|         |        Rate[1]: 44100.000000
|         |        Rate[2]: 48000.000000
|         |        Rate[3]: 48000.000000
|         |        Rate[4]: 88200.000000
|         |        Rate[5]: 88200.000000
|         |        Rate[6]: 96000.000000
|         |        Rate[7]: 96000.000000
|         |        Rate[8]: 176400.000000
|         |        Rate[9]: 176400.000000
```

```
|         |         Rate[10]: 192000.000000
|         |         Rate[11]: 192000.000000
|         |     Is Hidden: NO
|         |     Zero Timestamp Period: 0
|         |     Controls:
|         +AudioClockDevice
|         |     Object ID: 43
|         |     Base Class: aobj
|         |     Object Class: aclk
|         |     Owner ID: 1
|         |     Name: TimeSync Clock 0x11ddb1c4731a0006
|         |     Manufacturer: Apple Inc.
|         |     Device UID: ATSAC:11ddb1c4731a0006
|         |     Transport Type: atac
|         |     Clock Domain: 0x63690003
|         |     Is Active: YES
|         |     Is Running: NO
|         |     Nominal Sample Rate: 48000.000000
|         |     Available Nominal Sample Rates:
|         |         Rate[0]: 44100.000000
|         |         Rate[1]: 44100.000000
|         |         Rate[2]: 48000.000000
|         |         Rate[3]: 48000.000000
|         |         Rate[4]: 88200.000000
|         |         Rate[5]: 88200.000000
|         |         Rate[6]: 96000.000000
|         |         Rate[7]: 96000.000000
|         |         Rate[8]: 176400.000000
|         |         Rate[9]: 176400.000000
|         |         Rate[10]: 192000.000000
|         |         Rate[11]: 192000.000000
|         |     Is Hidden: NO
|         |     Zero Timestamp Period: 0
|         |     Controls:



--------------------------------------------------------------------------------
```