Synchronized real time audio streaming over ethernet in embedded systems

A thesis submitted for the degree Master of Science
in Information Technology

submitted by

Indumathi Duraipandian 926286
Fachbereich Informatik und Elektrotechnik
Fachhochschule Kiel

January 2018

Declaration of Authorship

I, Indumathi Duraipandian, declare that this thesis titled, 'SYNCHRONIZED REAL TIME AUDIO STREAMING OVER ETHERNET IN EMBEDDED SYSTEMS' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

Abstract

In any complex audio video networks such as professional audio recording studios, automotive or in-flight infotainment systems, concert avenues or even home entertainment systems, the connection between the various audio/ video sources and sinks are mostly analog, point to point and serve a single purpose. This leads to tonnes of confusing cables, each cable serving a specific data exchange. Even the digital solutions such as I^2S, S/PDIF, AES3 for short distance connections, most for automotive applications and Firewire (IEEE 1394), HDMI or audio over USB for high bandwidth applications, still require purpose built cables and proprietary software to work correctly and still they lack interoperability. Ethernet is now ubiquitous and offers high bandwidth for low cost over reasonable distances, so it makes sense to use Ethernet for audio/video streaming. The only limitation of Ethernet networks is that, since it is a packet switched network offering reliable, real time delivery of media is a challenge. To overcome this limitation and use the Ethernet networks for reliable, real time, flexible audio/video streaming, a set of protocols have been developed. These set of protocols required for a synchronized real time streaming is called the Audio Video Bridging(AVB). This is a relatively new standard and currently there are only a handful of implementations that make use of this standard. Our study discusses the implementation of AVB as a part of the open source Linux kernel which can be used and improved by anyone irrespective of applications. The study aims at building AVB stack for an open source hardware Beagle Board and study the various metrics such as latency, synchronization, throughput etc. between devices like Beagle Bone Black and Beagle Board X15 along with documenting the interoperability with off the shelf AVB devices such as MAC OSX laptops.

Acknowledgements

Its my pleasure to take opportunity for preparing master thesis report on 'Synchronized real time audio streaming over Ethernet in embedded systems' application. I thank from the core of our heart to Fachhochschule Kiel for permitting me for carry out the project.

First of all my hearty thanks to Prof.Dr.Robert Manzke who helps me in every possible manner that ensured a proper environment to work and allowed complete freedom and reposed complete faith in my work. He makes me aware about all the aspect of the system & its requirement deeply. I thank him for being a constant source of inspiration and for his advice that will help in my future in this professional field. He ensures the proper completion of project by reviewing my progress at the appropriate stages of the project development. His guidance and inspiration change this project into a faithful and meaningful exercise.

Being a newbie to all the technologies, I had really good hands on experience with these technologies now. After completing this project I came to know that learning technology is much more different than applying technology practically in any application.

I also owe our sincere thanks to all the friends who directly or indirectly helped me with their valuable suggestions. Their warm-hearted guidance acted like a lamp in darkness.
...

Contents

List of Figures

List of Tables

Abbreviations

VoIP	**V**oice **o**ver **I**neternet **P**rotocol
TRS	**T**ip **R**ing **S**leeve
NIC	**N**etwork **I**nterface **C**ard
MRP	**M**edia **R**edundancy **P**rotocol
MMRP	**M**ultiple **M**AC **R**egistration **P**rotocol
AVB	**A**udio **V**ideo **B**ridging
gPTP	**g**eneralized **P**recision **T**ime **P**rotocol
PTP	**P**recision **T**ime **P**rotocol
SRP	**S**tream **R**eservation **P**rotocol
MSRP	**M**aster **S**tream **R**eservation **P**rotocol
FQTSS	**F**orwarding and **Q**ueuing for **T**ime-**S**ensitive **S**ystems
AVDECC	**A**VB **D**iscovery,**E**numeration **C**onnection management and **C**ontrol
CAN	**C**ontrol **A**rea **N**etwork

Chapter 1

Introduction

This chapter introduces the audio/video systems that we use everyday in detail along with their various properties that they exhibit and the various requirements and limitations by which they are defined by. Furthermore down the technical challenges of designing and operating such systems are explained, which is followed by an in depth look into the current solutions available and discuss their merits and demerits. Through these discussions the need for a new improved solution for audio/video streaming is explained. The Audio Video Broadcasting protocol suite is presented as a possible candidate which fits the requirements listed. And finally the objectives required to assess the various features of AVB are listed.

1.1 Audio Video Systems

In the current "Information Age" we are constantly surrounded by all kinds of information processing systems. A most significant part of the information that we produce and consume is in the form of Audio/Video. Although much can be said about the audio/video information itself, here we concentrate more on the systems that are used in the production, distribution and consumption of these audio/video data. More specifically we are concerned in the embedded systems that are involved inside the several audio/video systems anywhere in the production-distribution-consumption chain. Based on the properties of the data and other parameters, these systems can be broadly grouped into two different kind of categories as listed below.

- Infotainment Systems:
  - Any form of audio video data produced and consumed for the purpose of entertainment or information distribution falls in this category.
  - These include movies, television, music albums, news broadcasts, educational programs etc...
  - In these systems large amounts of data are produced and consumed, hence high bandwidth interconnections are required.
  - These systems tend to me more of an one way communication.
  - For a single producer there can be a multitude of consumers.
- Communication Systems:
  - Real time communication via mobile telephone systems, VoIP, video conferencing, virtual presence devices etc... fall in this category.
  - Here synchronized, low latency systems are preferred.
  - These are most often two way communication systems, but there also can be one way communications.
  - Can be between two or also between a group of producers and consumers.

For any audio video system there are several parameters which needs to be considered when designing and building a system. Some of the most important of these parameters are described in detail below. All further discussions regarding audio video systems are discussed in terms of these parameters.

- Bandwidth:
  - Defined as the amount of data which is able to be transferred between two devices at any given time through the connection medium.
  - For systems which handle multi-channel, high resolution, high sample rate audio streaming and high resolution video streaming higher bandwidth is required.
  - The bandwidth required also increases if a single device is connected to several devices through single connection medium.

- Latency
  - Defined as the time required for the propagation of the audio from the source to the destination.
  - In general and specifically for communication systems low latency is preferred.
- Synchronization - All the devices in the network should work as if they are part of a one single large system.
- Cost efficiency - The cost of such an interconnect has to be affordable.
- Scalable - It should be possible to increase capacity without much complexity.
- Modular - Should be possible to be built from simple modules and should work with a minimal set of modules.

1.2 Technical Challenges

To understand the technical challenges in designing and operating the audio video systems, some real world examples of such closed audio/video systems are given below along with the requirements for these systems in terms of the vital parameters discussed in the previous section.

1. Professional Studios:
   - Made up of several input devices (Several musical instruments, vocals, signal/effect generators etc..).
   - When a recording is live, audio data from all these devices should be synchronized and then fed into the mixers/post-processors/encoders etc...
   - The end result from the encoders can be either broadcasted further or stored on a storage medium.
   - The synchronization between various devices is very important here or else we will end up with a recording with out of sync instruments which could not be considered as pleasant music.
2. Theaters/Stadiums/Concert Halls:

- These also contain a multitude of audio inputs connected to mixers/post processing nodes and encoders.
- The output from these mixers can be routed for the amplifier systems for live replay or for broadcasting and also for storage
- But unlike recording studios where faults can be tolerated and any changes to the systems done offline, these systems should be fault tolerant and should be flexible to be reconfigured live.

3. Infotainment:

   - Consists of a large number of receiver devices connected to a central media server, a public addressing system which overrides media consumption and a private communication system for the crew.
   - The main requirement is to handle high volume of data in low priority and a small amount of data in high priority.
   - Making sure the high priority communication systems gets bandwidth over high bandwidth media streams is vital for safety and security
   - Although synchronization is not required between the different receivers, the connections should be able to run parallel to each others.

4. Home media centers:

   - These are made up of all of the consumer electronic devices typically used in a home environment. (Home media servers, media players, television, speakers and everything else which can send or receive media).
   - The main requirements in these systems are ease of use (plug and play), high bandwidth and fault tolerance.

1.3 Established solutions

The previous sections established the different parameters involved in designing an audio video system and the different requirements involved in operating the various real world audio video systems. This section documents the various solutions that are currently available for the interconnections between various devices to transport audio/video data

in the systems. The interconnection systems are currently used are listed below along with their unique characteristics, advantages and disadvantages [7] [8] [9].

- Analogue Connections
  - Uses electrical signals over common electrical cables or twisted pair cables to directly transfer audio video signals and uses Tip Ring Sleeve (TRS) or RCA connectors
  - Audio is transferred as either balanced or unbalanced signals, while videos is transferred as either composite (Color, video, blank, sync), S-video or component (RGB / YCbCr) signals.
  - Advantages of analogue connections are as follows
    * As the signals travel near the speed of light, latency is virtually non-existent for everyday usage cable lengths
    * Simple to setup and operate. High portability as most of the devices support analogue connections
    * As there is no latency, synchronization can be achieved without much further effort and it is cost effective as it uses off the shelf and simple components
  - Dis-advantages of analogue connections are as follows
    * Signal quality deteriorates quickly as the cable or connector quality reduces and with increasing noise
    * No multiplexing possible, a separate connection has to be setup for every single different media source or sink
    * Depending on the cable quality and operating environments signal booster and other post processing equipments are needed additionally where required
- Digital media specific connections:
  - Audio/Video signals are digitized, encoded and transmitted as binary signals organized in a fixed frame structure
  - The signals are usually transmitted over electrical, optical or coaxial cables with RCA, TOSLINK and BNC connectors respectively [8].

  - Protocols such as SPDIF, AES, ADAT, DVI and HDMI are used for the packing and framing of the audio samples along with specification for the electrical or optical signal levels.
  - Advantages of digital audio connections are as follows
    * Provides better bandwidth than analog connections are a single connection can carry multiple channels of audio streams in higher bit and sampling rates
    * More resistant to noise and signal degradation because of cable length and other factors
    * Mature standards with support from several audio video device manufacturers
    * Some kind of synchronization can still be achieved by also transmitting media clocks over separate interface and synchronizing based on the transmitted clock.
  - Dis-advantages of digital audio connections are as follows
    * Because of the additional processing required the overall system latency increases and the exact latency value varies from device to device depending on implementation
    * Portability is little low since different protocols are not inter compatible and additional equipment might be required to transfer signals between different technologies
    * Because of the additional chips required or encoding and decoding along with the special cabling required the cost of the system increases.

- Digital general purpose connections
  - These connections are designed to be general purpose to carry any kind of data in parallel. Because of their general purpose nature they can also carry media data
  - Protocols such as USB, Firewire, Thunderbolt, Ethernet are some examples for such general purpose connections.
  - All these specifications also define their own specific kind of cabling and connectors and all the physical layer specifications.

- Advantages of digital general purpose connections are as follows
  - Provides the best bandwidth of any kind of connections and also provides the possibility to compress the media information and thereby increasing the available bandwidth even more
  - Easy to setup and operate since these protocols are used by everyone and so many of these protocols provide plug and play functionality
- Dis-advantages of digital general purpose connections are as follows
  - As it is not optimized for real time traffic latency can be higher and also jitter is higher
  - Because of the high jitter and a tendency to send data in short bursts instead of regular streams synchronization is very difficult in these connections.

1.4 Audio over Ethernet Protocols

Among the several of the digital audio interconnection mediums listed in the previous section, Ethernet has several inherent advantages. Because of this reason there are several protocols developed under the *"Audio over Ethernet"* model, some noteworthy protocols among them are COBRANET, Ethernet, Ravenna, AES67, Dante and AVB/TSN [8]. And among these protocols Dante is a mature protocol which has been well adopted by the industry and supported by several manufacturers. Some of the advantages and disadvantages [10] [11] [12] of the Dante protocol is listed below:

- Dante has a large adaptation among other protocols in the industry with several devices available in the market with support of the protocol.
- It supports synchronized streaming of multi-channel, high sampling rate audio streams with low latency.
- The dante systems are easily scalable and they support plug and play functionality in most common operating systems such as windows and MAC.
- But Dante is a closed and proprietary system developed by Audinate Inc., which implies royalties, no flexibility in improvements and several other constraints.

1.5 Need for AVB

As discussed above there are several options for interconnections for media devices available. But none of the connections provide a complete better performance over all the parameters. For example the general purpose digital connections provide good bandwidth and ease of operation, but they are very bad for latency and synchronization. On the other hand the digital media connections provide better latency control and synchronization but they provide only a fixed bandwidth and are difficult to setup and reconfigure.

Since the requirements for media streaming grows (high definition and even ultra high definition 4K video streaming, multiple channel surround sound systems, personalized entertainment systems becomes more common, which also increases the demand for content creators to create more content and in some cases high bandwidth real time content), the traditional solutions are stretched to their limits and can no longer support the future streaming requirements in a cost effective way along with keeping the quality of service to a high level. This leads to a conclusion that a new solution is needed to be developed.

Among the several possible solutions proposed, AVB which is based on Ethernet networks has gained traction and is set to replace established media streaming solutions in the near future. Ethernet was chosen as it provides high bandwidth in a cost effective way and already the adaption of Ethernet is high as it is virtually available everywhere. It is also more flexible to develop new specifications that provides the real time synchronized media data transfers required. Because of these various advantages AVB is a good solution for synchronized real time audio streaming in embedded systems.[13]

1.6 Objectives and Thesis outline

As discussed in the previous sections, Audio Video Bridging(AVB) promises to fulfill the requirements for a synchronous real time media transfer for embedded system over Ethernet networks. The goal of this study aims to test the various features of the AVB media streaming in a real embedded system environment and study the various operational parameters. To fulfill this goal the following objectives are set:

- Study of the current available solutions that implement AVB and their features and limitations.
- Implement the various protocols in the AVB stack in the an embedded system platform that is publicly available.
- Evaluation and documentation of the various features of the streaming and the various parameters that define a media streaming such as synchronization, latency, throughput etc...
- Discuss the results and provide a conclusion for the feasibility of AVB as a solution for a synchronized real time audio streaming system on Linux embedded systems which is cost effective, simple to use and provides a high quality of service

Chapter 2

Literature Review

The IEEE 802.1 and IEEE 1722 related specifications for Audio Video Bridging are available as standards since the early 2010s. Since then there have been implementations for the Audio Video Bridging systems. In this section, some of the most prominent of such implementations are reviewed and discussed along with their offered features and limitations along with their advantages and disadvantages.

2.1 Open Solutions

The following are some of the open source solutions for the AVB protocols implementation.

2.1.1 Open Anvu

The Anvu Alliance is an industrial consortium of various companies from different backgrounds such as automotive, consumer electronics and industrial systems manufacturers with common interest is media applications. It was created on 2009 by Broadcom, Cisco, Harmann, Intel and Xlinx. The main objective of this alliance is to work together to establish and certify the systems which implement the Audio Video Bridging systems so that these certifies systems work together seamlessly without the need for any modifications or additional configuration. The Anvu alliance provides an Anvu logo which assures that the device has been certified.

Additional to the certification process the Anvu alliance also sponsors the development of the Open Anvu project. It was first initiated by Intel to encourage the collaboration in development of AVB systems. Collaborating in development ensures standardization, stability and interoperability between systems. The code is released under BSD and GPLv2 Licensing terms and collaborations are always welcomed. The project is still under development to improve and add various new features. The various components and features and in general the advantages of this project are listed below:

- ✔ A Network Interface Card (NIC) driver for Intel devices under Linux that supports the various AVB requirements for a NIC driver.
- ✔ A MRP daemon that supports the set of IEEE802Q-2011 MSRP, MVRP and MMRP protocols which are used to stream reservation and VLAN registrations.
- ✔ A gPTP daemon that implements the IEEE802.1AS-2011 protocol for the time synchronization.
- ✔ Apart from the libraries sample codes for a simple talker and listener are provided as an example how to use the various provided libraries to build a AVB system.
- ✔ As the libraries developed as a collaboration effort interoperability and stability of the implementations are high.

Apart from the various features and obvious advantages of this project there some disadvantages or limitations of this project some of which are listed below.

- ✗ Although several modules required for AVB are implemented it is not yet a complete solution which can be directly used to develop a complete working system
- ✗ The project is still under development and some modules required for a complete AVB system such as AVDECC and AVTP are not yet implemented
- ✗ Not compatible with all software and hardware platforms

2.1.2 XMOS Xcore

XMOS is a fabless semi conductor company which designs voice processing, music processing and control micro controller designs. Most of the chips contain multi-core processors capable or concurrent real time operations with DSP core and control modules. It has several devices based on the XCore technology namely XCore Voice, Voice fusion, for voice applications, XCore Audio for audio processing and XCore 200 and Xcore XS1-L for multi-core applications. The XMOS XCore architecture provide various features of real time operation systems such as scheduling, I/O operations and inter process communications. As all these scheduling happens in the hardware in the end the responses to events can be very fast in nanoseconds precision. The AVB implementation in the XMOS XCore Audio processor [1] is the worlds first to pass the Anvu Alliance certification and to get the Anvu logo. A basic architecture of an XMOS XCore Audio is given the following figure

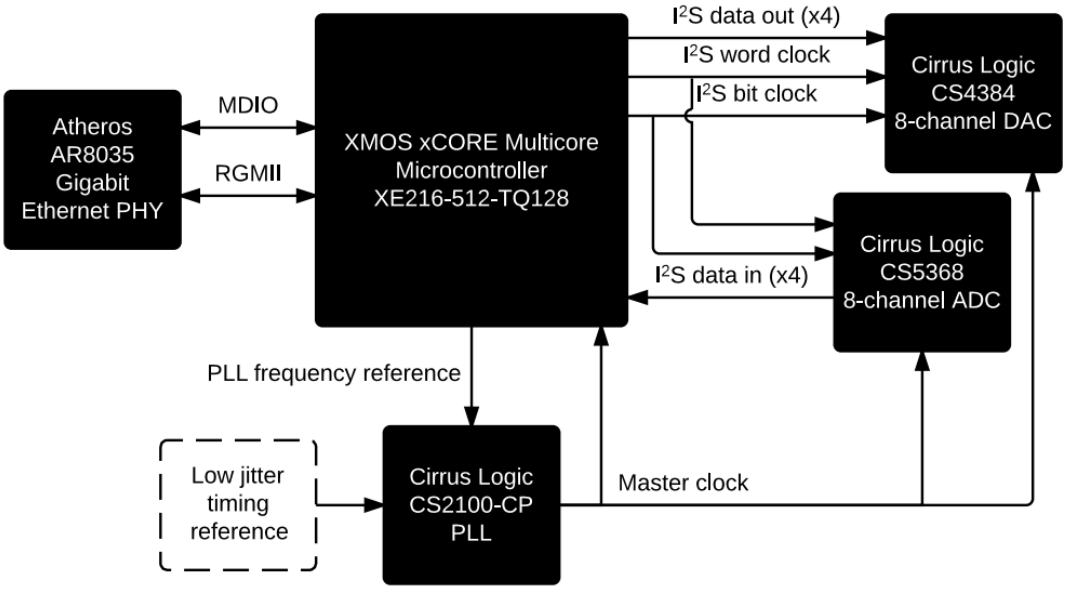

FIGURE 2.1: XMOS XCore 200 with AVB [1]

The software used for the XMOS XCore Audio systems are maintained by XMOS as a open source development [14]. It is the base for the firmware code for the reference implementation of the AVB protocols for XMOS silicon. The features and advantages of the XMOS Xcore Audio AVB solution are as follows

- ✔ Almost complete support for all the related protocols in the AVB protocol as a single ready to use solution
- ✔ The dedicated hardware provides assured performance, reliable and accurate behavior.
- ✔ The first Anvu certified implementation which assures reliable performance and interoperability

The limitations and disadvantages are listed below,

- ✗ Purpose built solution which supports only the XMOS XCore processors in supported models
- ✗ As solutions are externally implemented as a separate and at-least partly with hardware, it increases the hardware bill of materials and it is hard to reconfigure
- ✗ Again as purpose built solution it is not very cost effective

2.1.3 Miscellaneous Solutions

Apart from the major AVB open source solutions described above there are some more open source implementations [15] [16] available for the individual protocols in the AVB protocols. Some of these protocol implementations are listed below

- ➜ Several implementations for the IEEE 1722.1 AVDECC controller, talker and listeners as libraries and or applications [15]
- ➜ Several PTP daemons implementing the IEEE 1588 Precision Time Protocol (PTP) and the IEEE 802.1AS gPTP protocols for the time synchronization [15] [16]

Although these standalone implementations provide reference implementations for several of the protocols in the AVB stack,they do not provide a complete solution for the synchronized media streaming requirements since they are standalone solutions. Even though it is possible to combine several of these implementations to build a complete AVB solution, interoperability could still be an issue.

2.2 Proprietary Solutions

Apart from the open source community driven developments there are several proprietary solutions are available for AVB synchronized media streaming. As in most cases in the proprietary solutions source code and most of the implementation details are not in public domain, we look into the end products and their features. Some of these products with proprietary solutions are listed below.

- Macintosh Operating System
  - The MAC OS has inbuilt support for AVB and synchronized audio streaming over Ethernet since Mac OS version 10.9 Mavericks
  - Although the initial implementation is basic, the features has grown matured in the latest versions
  - But not all models are supported Macbook Pro and iMac are generall supported and Macbook air and similar devices are not supported
  - OS X devices also support AVB Ethernet through the tunderbolt ports using a Thunderbolt to Ethernet adapter
  - When a AVB connection to a AVB supported device is setup all the audio applications in the device can stream audio through AVB
- MOTU Sound Cards
  - MOTU provides a range of external audio devices with Ethernet AVB support and a range of conventional audio connections
  - MOTU also provides a collection of AVB enabled Network switches to build a larger AVB network with many devices in it
  - MOTU devices are also certified by tha Anvu alliance and provide mature devices which can inter-operate seamlessly with other devices
  - Most of the MOTU devices can be used for semi professional to high end consumer audio applications
- AVID VENUE Pro-Audio
  - AVID VENUE range of devices provide professional quality audio recording and distribution systems with inbuilt support for AVB

- ➤ For example the AVID VENUE S3L systems has 3 Gigabit Ehternet port with AVB support enabled
- ➤ These devices are exclusively used for professional audio applications

➜ Excelfore eAVB Automotive Systems

- ➤ Excelfore provides Ethernet AVB solution for Automotive applications ported for several hardware platforms and several operating systems
- ➤ Apart from Audio, it is also provide possibility for video streams along with other regular Ethernet traffic and also acts as bridging network most automotive network traffic such as LUN and CAN etc...

Chapter 3

Technical Background

For evaluation of AVB in an embedded system environment, AVB stack software is developed over Linux operating software in a BeagleBone Black development platform. This chapter elaborates the technical specifications of the various standards involved in the AVB stack, an introduction to the Linux operating system along with the interfaces provided for development of AVB standards and finally the features of the BeagleBone Black platform are described in detail.

3.1 BeagleBone Black

BeagleBone Black is a low cost high performance development platform aimed at developers and hobbyists. It is extensively used to develop applications in the fields of multimedia, connectivity, prototyping, robotics etc... It is mainly built around a AM335x Sitara line of microprocessors from Texas Instruments, which are based on ARM cortex A8 main processor running at a maximum of 1 GHz. They are optimized for image and graphical processing applications with several inbuilt peripherals and a multitude of interface options. Apart from the ARM cortex core, there are dedicated processing core for 3D graphic acceleration, a NEON SIMD co-processor as floating point acceleration and two Programmable Realtime Units (PRU) that run on a separate clocks and has access to a different set of exclusive peripherals. Some key features of the BeagleBone Black can be summarized as below.

- AM335x Siatra Processor. 2000 MIPS @ 1 GHz

- 500 MB DDR3L @ 800 MHz
- 4 GB embedded MMC onboard flash
- 2x USB, 1x UART, 1x mini micro SD card port, 1x HDMI and 1x 10/100 RJ45 Ethernet connector
- 2x 46 pin expansion headers through which up to 4 expansion devices can be connected
- Also available are headers for battery connection for stand alone operation and a 5v power connector along with power switches, power indicators and other indicators.

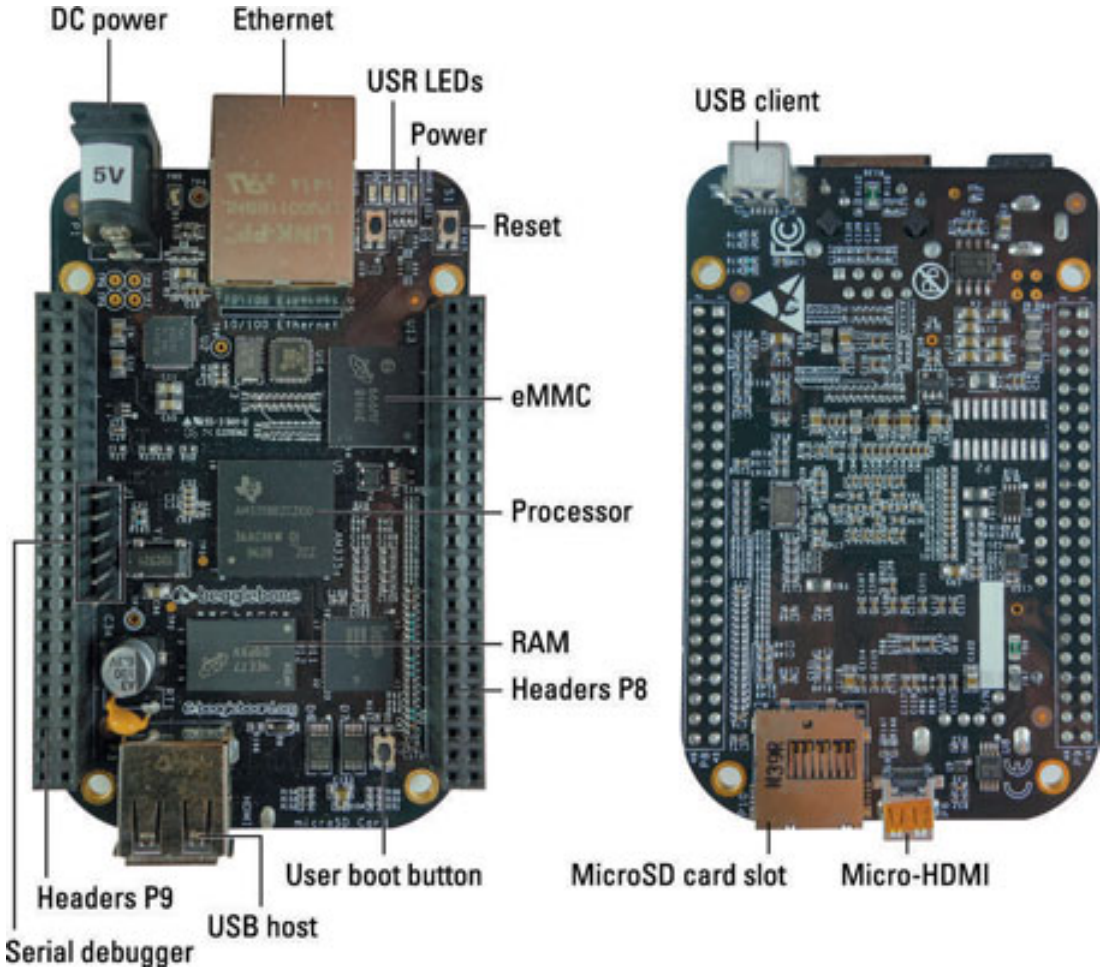

FIGURE 3.1: BeagleBone Black [2]

The BeagleBone Black can be further enhanced via the usage of capes, which are basically expansion boards which directly connects to the 46 pin headers P8 and P9. A special feature of these capes are that the unused pins from the headers are passed

through the capes and so further capes are added on top of other capes (with a maximum limit of four capes). These capes add additional functionality of the BeagleBone Black with some additional hardware interfaced through the expansion headers. Some examples of capes are as follows.

- The onboard HDMI and eMMC are also considered as virtual capes as they are connected to the interface from the expansion headers.
- Power cape for managing any batteries connected and also monitor, charge and regulate the battery power.
- LCD, RS232, CAN and other interface capes for connecting external display and communication protocol.
- Audio, video expansion capes to extend the audio video interfacing options of the BeagleBone Black.

3.1.1 Ethernet Time stamping

The Ethernet module on board the BeagleBone Black and subsequently the AM335x processor, support hardware time stamping for Ethernet packets, which is required for the implementation of the Precision Time Protocol as described in section 3.3.1. The PTP is the crucial protocol required to synchronize the different devices communicating via the Ethernet network.

There are three different possibilities for generating time stamps for Ethernet frames, they are as listed below.

- Software Time stamping:
  - In this method the time stamping is done completely in the software side in the socket processing.
  - As this is software the the variations in time measurement can be huge as this Ethernet packets can suffer variable delays in the software Ethernet queues.
  - This can be lead to time synchronization with in the such a system to only in the range of several hundred micro seconds to few milliseconds.

  - Although this synchronization is sufficient to some systems it is not sufficient enough synchronization for most real time systems and also this level of synchronization can also be achieved through other existing protocols such as Network Time Protocol (NTP).

- SFD Time stamping:
  - The SFD (Start of frame Detection) time stamping works by time stamping in the hardware at the MAC layer when the Ethernet physical transceiver detects the start of the Ethernet frames.
  - This way of time stamping provides high levels of synchronization enough for the real time media synchronization.

- Hardware Time stamping:
  - In hardware time stamping the time stamps are directly generated by the Ethernet physical transceiver thus ensuring higher levels of accuracy.
  - When low or sub nano second time synchronization accuracy is required then hardware time stamping has to be employed.

The support for hardware time stamping of Ethernet packets in the AM335 processors are handled by the Common Platform Time Sync (CPTS) module inside the processor in the Start of frame Detection (SFD) time stamping method. This module will detect the precise ingress and egress time of the various commands belonging to the PTP are received or transmitted through the Ethernet physical layer.

3.1.2 Audio Cape

As a bare BeagleBone Black development board doesn't contain any on-board audio output (apart from the audio through HDMI which is not analog and could be compressed), if audio output is required the functionality has to be extended via audio capes for BeagleBone Black. One such audio caoe for the beagle bone black is the CTAG face 2—4 device. [17]. This device supports 2 stereo input channels and 4 stereo output channels. Also a full featured audio ALSA driver is available to interface this device with the beagle bone black.

3.2 Linux Operating System

An operating system is a collection of software modules that are responsible for managing a hardware platform (Initialize, start-up and manage communication to and fro the hardware components) and provide an unique interface for user space applications (office applications, media players, browsers and other such utilities). This way an operating system presents an abstract view of hardware for user application developers so that they do not need to be concerned about the underlying hardware.

Linux is a name referring to a family of operating systems that are packaged around the Linux kernel. There are several Linux distributions ("distros" in short) available for every kind of application and every kind of platform it will run on. Linux based operation system are capable of running from a very basic microprocessor with few mega bytes of RAM to state of the art super computer clusters operating at the cutting edge of computing performance. In such way Linux is flexible and modular enough to fit to any hardware platform. These properties come from the philosophy of open source development where any one is able to use the existing Linux source as their starting point and modify it according to their needs or according to their hardware platform. This decentralized way of development means fast adaptation of new features and new hardware platforms.

The various modules inside a Linux operating system are described in detail in the following chapters.

3.2.1 Linux Kernel

A kernel is a small, high performance and highly optimized software component responsible for the overall management of the hardware components, user level software and all the communication between them.

Depending on how they are organized, there are several types of kernels namely

- Monolithic kernels where all the kernel operations are completely implemented in a single kernel processing thread which increases processing speed but are prone to stability issues as the code base grows

- Micro kernel where the kernel is as small as possible holding only the very basic of kernel operations with every other functionalities implemented in the user space. Although this approach leads to slower system performance, it provides much better stability and security as the small code base can be tightly controlled.
- Hybrid kernels which are a combination of monolithic and micro kernel features
- Nano kernels which are similar to micro kernel but still smaller than micro kernels
- Exo kernels where only the hardware management is part of the kernel with no hardware abstraction, thus forcing the application developers to directly handle the hardware.[18]

The three main functionalities of a kernel is as follows.

- Scheduling
  - Multitasking is a major feature for any operating system where the user is able to execute several different operations at any given time
  - This is usually done by time slicing the access to the main processor for the several different applications that are executing at any given time
  - There are several different approaches to how a time slicing organized such as preemptive, fixed priority preemptive, round robin etc...
- Memory management
  - Similar to the main processor, the main memory is also a very important system resource which has to be shared among the several running applications
  - There are several approaches for handling memory management such as virtual addressing, segmentation and paging
- I/O management
  - Apart from the main processor and the main memory the operating systems is also responsible for management and handling of several different internal and external hardware components

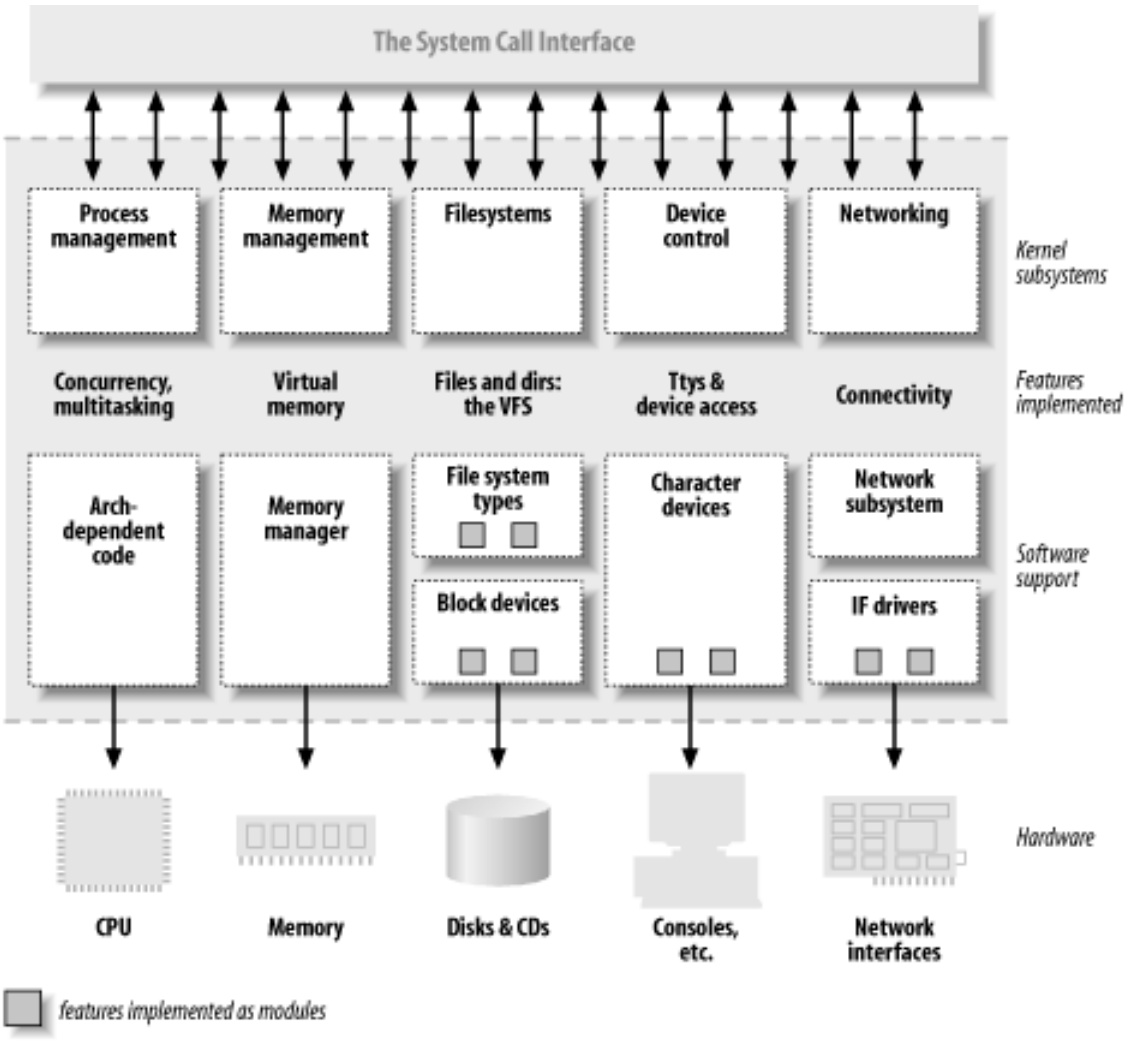

FIGURE 3.2: Linux kernel Architecture [3]

From the discussions above, Linux kernel can be described as a Monolithic, true preemptive multitasking kernel with support for virtual memory. Devices are handled by device drivers completely embedded in the kernel with on demand loading and manual loading of loadable kernel modules. The various modules involved in the Linux kernel are presented in the following figure 3.2.

3.2.2 System calls

Linux system calls are a way for the user space applications to invoke functions in the Linux kernel. When a system call is invoked the kernel functions are not invoked directly but the system call invokes a software interrupt which looks up the system call table and executes the respective kernel function for the system call. The system call in Linux can

be invoked by two ways, one is through the syscall() library function where the system call number has to be passed along with the required arguments. This library function stores all the current registers, loads the arguments and executes the kernel function. Upon completion the syscall() returns either 0 in success or -1 in case of failure with the failure reason is returned in the errno global variable.

3.2.3 Device drivers

Device drivers are software components which resides inside kernel and manage a hardware component. It presents a common interface for the user to operate on that device by abstracting the device into the standard forms which are defined as follows

- Character devices
  - Character devices exposes the device as a file which can be opened and closed as a normal file. The contents of the file can be read or written as a stream of bytes
  - The difference between a normal file and character device is that the former has the ability to seek back and forth the file, while character device driver are not able to seek through the file, just read and write at the end of the file sequentially
  - Many of the character devices are accessible in the /dev/ path of the Linux file system
- Block devices
  - Block devices are similar to character devices, but instead of providing data as a stream of bytes block device provide data always as a fixed block
  - Although block device can only handle a block of data at a time the Linux kernel provides the possibility to read and write arbitrary length of data just like character devices
  - So the only difference between character and block devices are only inside the kernel on how the kernel handles the device drivers
- Network devices

  - Each network device is exposed as a network interface which are capable of sending and receiving data packets
  - As network devices are not stream oriented there are no file system entries for network devices
  - Network devices are nonetheless are identified by a unique identifier in the Linux system which can be used to send or receive data packets to the device.

Although these are the major device driver models inside the Linux system, there are also some minor device driver models available. And also developers can always implement their own device driver model and interfaces for their proprietary devices as required. This is required for some special purpose hardware when having an unique interface provides a more optimized way of using the device than using the standard model of device drivers.

Some important points to note for a Linux device driver development can be summarized as follows,

- Traditionally all the Linux device driver development is implemented using C programming language. Although any other programming language (like C++) could technically be used, it is actively discouraged to try such approaches as they could potentially lead to a cluttered sub-par kernel
- The starting point for the device driver development is to implement some predefined set of device and driver management functions such as init, exit, probe, remove etc... and then register these functions to the kernel along with the information regarding the driver and it's features and limitations
- The kernel will use this information to load, unload the driver as required i.e. when the device is detected or removed etc...
- After the basic functions are implemented based on the device driver mode chosen the required set of function such as open, read, write, ioctl, close etc... has to be implemented and registered with the kernel
- The kernel will use these functions when the associate file is opened by a user mode application and used through the read, write file operations system calls

- Once the development of the source code is completed, the newly developed has to be integrated into the kernel build system by specifying the sources needed to build the kernel in Makefile and loading options such as included, loaded or not enabled in the configuration file Kconfig

3.2.4 Networking stack

The networking stack in the Linux kernel comprises of the following drivers or software components such as network device drivers, kernel core networking module, separate modules for various networking protocols etc... All these software modules interface each other with a common data structure called the socket buffer. This socket buffer contains all the information required to send or receive a packet data (which is also contained inside the socket buffer). For transmitting or receiving any data through a network device a socket has to be created and based on the protocol has to be used and any other special features required the socket options has to be set and then the socket can be used to send packet data or receive packets from the network through the network device. There are several options available that can be set on a socket to enable or disable some special features.

As explained in the section 3.1.1, the Ethernet time stamping is supported by the TI AM335x processor. As this time stamping options is need to used the development for the gPTP protocol in AVB stack there are some software interfaces available through the kernel to operate this feature of the Ethernet module of the microprocessor. This feature is implemented by the CPSW Ethernet driver for the TI AM335x processor. As the user space application doesn't directly use the Ethernet driver, but use the kernel functions related to sockets. So the kernel functions has to provide an interface to use the Ethernet time stamping from the user mode processes. As the Ethernet time stamping options can be different in different implementations (software or hardware) the kernel socket API provides a common interface to handle the Ethernet time stamping. Among the several socket options available to configure the Ethernet time stamping some of the important options are listed below.

- SO_TIMESTAMPING - This socket option enables the generation of timestamping for both transmission and reception sockets. Also supports generation of time stamps for stream scokets.
- SOF_TIMESTAMPING_R(T)X_HARDWARE - This socket option enables the generation of hardware time stamping where supported
- SOF_TIMESTAMPING_RAW_HARDWARE - This socket option enables the reporting of the generated hardware time stamps, this should not be confused with the previous option where only the generation is configured. This option enables the delivery of the generated hardware time stamps to the caller

The generated time stamps are usually returned with the ancillary data feature in the received data packets. For transmitted packets the time stamps are returned along with the transmitted packet in the error queue of the socket.

3.2.5 ALSA Framework

Advanced Linux Sound Architecture (ALSA) is a software framework that is part of the Linux kernel and provides a simple Application Programming Interface (API) for all the audio device drivers. Apart from the audio device drivers ALSA also provides a user space library which further simplifies the exposed API and thus simplifying the design for any high level audio application development. This interface include simple calls such as snd_pcm_open, snd_pcm_close, snd_pcm_read, snd_pcm_write and snd_pcm_ioctl etc... through which the audio PCM data can be transmitted or received via the desired device. ALSA also provides software plugins that provide some extra functionality such as volume control, mixing, re-sampling etc...

3.3 Audio Video Bridging

The AVB standard consists of a set of protocols defined by IEEE 802.1 Audio/Video Bridging Task Group to allow real time media transmission over Ethernet. AVB provides reserved Ethernet bandwidth for deterministic transmission.The purpose of AVB is to provide time synchronized reliable data through IEEE 802(Ethernet) networks.

AVB consists of

- IEEE 802.1BA: Audio Video Bridging (AVB) Systems[19]
- IEEE 802.1AS: Timing and Synchronization for Time-Sensitive Applications (gPTP)[4]
- IEEE 802.1Qat: Stream Reservation Protocol (SRP)[5]
- IEEE 802.1Qav: Forwarding and Queuing for Time-Sensitive Streams (FQTSS).[20]
- IEEE Std 1722.1: AVB Discovery, Enumeration, Connection management and Control (AVDECC)[6]
- IEEE 1722: Audio Video Transfer Protocol (AVTP)[21]

3.3.1 Generalized Precision Time Protocol(gPTP)

The Generalized Precision Time Protocol [4] is used for the synchronization of the media clocks in all the Ethernet ports in a Local or a Virtual Local Area Network (V-LAN).

Although through the existing protocols such as Network Time Protocol (NTP) it is still possible to synchronize the clocks of various devices in the network, because of propagation delays in the network the time advertised by the time server will always be outdated by some nano-seconds to some milli-seconds depending on the position of the receiving node in relation to the time server. This leads to different levels of synchronization to different devices as each device receives the time at a slightly different time and also the setting of the time by the end devices are not standardized and so the devices are free to apply the time whenever and however they want. The end result is a network with the devices that are synchronized so some milli-seconds precision.

This level of precision is sufficient for most normal user applications but for media streaming and most professional industrial systems much more rigorous synchronization is required. For example for a 48 kHz audio streaming, a single sample duration is 20.83μs, so the time synchronization between the transmitter of this audio stream an the receiver of this audio stream should be less than 20μs for the streaming has to result in synchronized playback.

The main drawback of the existing time synchronization protocols is that they do not account for the propagation delay for the time announcement message to propagate through the network from the time server to the individual time slaves. The Precision Time Protocol aims to solve this problem by accurately measuring the propagation delay between every hop in the network and using this value to correct the advertised time at every hop. This way when the advertised time reaches a device the time is correctly synchronized to the time advertised originally as it is corrected to account for the propagation delay. The basic operation and the modules in the gPTP as described below

- Peer delay measurement
  - The peer delay measurement is used for measuring the propagation delay between two participating devices
  - The delay is measured by sending delay measurement request and receiving the response
  - The ingress and egress times of both the request and response are time stamped and returned to the initiator of the delay measurement
  - From these four set of time stamp values the average propagation delay
  - This peer delay measurement is executed on all ports in the system for example the routers in the network the peer delay measurement is executed on all the ports to which supported

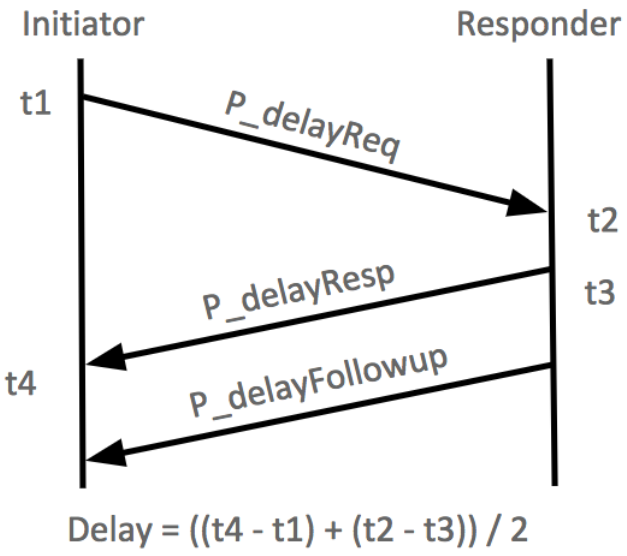

FIGURE 3.3: gPTP - Peer delay measurement [4]

- Best Master Selection
  - The master device which is responsible for announcing the time is selected dynamically based on the advertised capabilities of the devices
  - The device which has the most stable and accurate clock is normally selected as the master device
  - But the master device selection can also be overridden to chose a particular device by assigning user defined ranks for all the system in the devices such that the master device with a higher user set priority is chosen as the master device
  - The best master selection is a distributed protocol where the decision for which device should be master is individually decided by all the devices in the network
  - Each device reports the clock class, accuracy, user set clock priority and other parameters
  - On reception of this announcement message each device compares it with the parameters of the current selected grand master device and if the new parameters are better the current the new device is selected as the master
- Time Synchronization

- Time synchronization is achieved by the master device sending sync messages periodically and every other device correcting their clocks to match the received time
- For routers they update the received time sync messages and add the propagation delay for the port where the sync messages are received and then forward the sync messages to all other ports

An Ethernet frame for the gPTP has a common header as specified in Table 3.1 followed a variable length gPTP body, the contents of which are specified separately for each gPTP message type and the frame ends with an optional suffix as required. All the gPTP messages have an Ethertype value of 0x88F7 and are sent to the broadcast MAC address 01-80-C2-00-00-0E.

Offset	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
0	Transport Specific				Message Type				Reserved				PTP version			
2	Message Length															
4	Domain Number								Reserved							
6	Flags															
8	Correction field															
16	Reserved															
20	Source Port Identity															
30	Sequence ID															
32	Control Field								Log Message Interval							

TABLE 3.1: gPTP Common Message Header

A brief explanation for the various parameters in the gPTP header are specified below

- **Transport specific** parameter is a 4-bit value indication which transport is used to transport the gPTP message. For Ethernet the option L2 is used
- **Message Type** is also a 4 bit value which specifies the current gPTP command. More information and possible values for this parameter is explained in Table 3.2

- **PTP Version** is a 4 bit vlaue indicating the version of the gPTP used
- **Message Length** specifies the length of the gPTP message including the header, data and any optional suffix and exlcuding any padding bytes at the end of the frame. It specifies the length in terms of the number of bytes required
- **Domain Number** refers to the domain in which the gPTP is executing
- **Flags** is a bitfield value indication several parameters of the current messages
- **Correction Field** specified the correction value to be applied in nano seconds
- **Source Port Identity** is an unique identifier which is used to identity the currrent device in the network
- **Sequence ID** is a running count of indicator to find the sequences of the messages. This value increases for all the request messages and the response messages will use the same sequence numbers as the request messages
- **Control Field** is a deprecated value depending on the message type specified in the current command
- **Log Message Interval** specifies the log value of the timeout value for the repetition of the specific message type

The various message types, their numeric value, the PTP body for the message type etc... are documented in the following table.

gPTP Message	Type	Description and parameters
Sync	0x00	➢ The sync messages are sent by the grand master device in periodical intervals ➢ The sync messages contain no more additional data in the gPTP body and is used only as a trigger to synchronize the time ➢ The sync message is always followed up by the Follow_Up message

Follow_Up	0x08	➢ The Follow_Up messages are also sent out by the grand master device immediately after the sync messages are sent ➢ They contain the precise origin time stamp which is the egress time stamp for the sync messages at the port measurement plane of the grand master ➢ These messages are also forwarded by the routers after correcting the time stamp for the measured propagation delay ➢ When the other devices in the network receive the Follow_Up message they use the time stamp in the message and the propagation delay measured to correct it's internal clock
Pdelay_Req	0x02	➢ The Pdelay_Req command initiates the peer delay measurement sequence ➢ The peer delay measurement initiator device, when sending the Pdelay_Req command records the command egress time stamp (t1) ➢ When a device in the network receives this message it records the ingress time stamp (t2) of this message ➢ The Pdelay_Resp command is sent as a response along with the Pdelay_Req ingress time stamp
Pdelay_Resp	0x03	➢ The Pdelay_Resp command is sent in response to a Pdelay_Req command reception ➢ When sending the command the peer delay measurement responder records the egress time stamp (t3) for the command ➢ When the peer delay measurement initiator receives the Pdelay_Resp command, it records the ingress time stamp (t4) ➢ It also parses the Pdelay_Resp command the copies the ingress time of the Pdelay_Req (t2) received by the peer delay measurement responder

Pdelay_Resp_Follow_Up	0x0A	➢ The peer delay measurement responder after sending the Pdelay_Resp command sends the Pdelay_Resp_Follow_Up command ➢ The Pdelay_Resp egress time stamp (t3) recorded is packed into this command and sent ➢ The peer delay measurement initiator upon receiving the command parses the command and extracts the Pdelay_Resp egress time stamp (t3) ➢ At this point the peer delay measurement initiator has all the four time stamp values (t1, t2, t3, t4) ➢ Using the four time stamp values the average propagation delay is calculated as ((t4 - t1) + (t2 - t3)) / 2
Announce	0x0B	➢ Announce messages are sent by every device that are grand master capable ➢ Announce messages are sent only when the device has not decided upon a grand master or when it is the grand master ➢ Announce messages advertise the capabilities of it's clock source and other user configurable parameters that sets the clock priorities ➢ When a grand master capable device receives an announce message with a better clock or a higher priority than it's parameters it stops sending announce messages and records the device with better parameters as grand master

TABLE 3.2: gPTP Message Types

3.3.2 Multiple Stream Reservation Protocol (MSRP)

The Stream Reservation Protocol is used to reserve the required resources in all the devices of the network between the stream source and the stream sink so that the quality of service for the AVB streams are assured. There are three kind of devices involved in the Stream Reservation Protocol as listed below

➢ **Talkers** are end stations that are sources for media stream data

➢ **Listeners** are end stations that are sinks for media streams

➢ **Bridges** are the routing devices that are in the path between the Talkers and Listeners are defined as the bridges in the system

Stream Reservation Protocol operation sequence comprises of the following steps

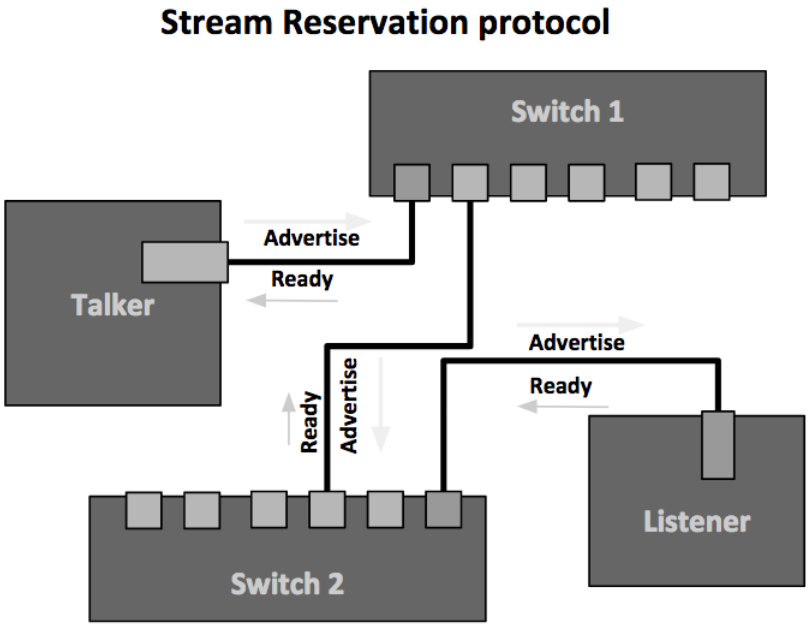

FIGURE 3.4: Stream Reservation Protocol [5]

- The Talkers in the system declare attributes that define the characteristics of the streams they provide.
- These declarations are evaluated by the bridges that receive them and if the bridge can allocate the requested resources for the declared stream the declarations are forwarded to the next device in the network
- If the bridges are unable to allocate sufficient resources for the requested streams they send back a negative response to the talker without further propagating the original talker declarations
- When the talker declarations reach a Listener end point device, the Listener evaluates the declarations and if it is interested in the provided streams, they send a Listener advertisement to register for the stream

- This listener advertisement is also propagated through all the bridges in between the Listener and the Talker in the network
- There is a possibility that there are multiple Listeners interested in a stream by a Talker, in that case the Talker will receive multiple Talker advertisements for the stream
- If the talker receives at least one successful Listener advertisement, then the stream reservation is successful and the Talker can start the media streaming

An Ethernet frame for the SRP has a common header as specified in Table 3.3 followed a variable length body, the contents of which are specified separately for each SRP message type and the frame ends with an 2 byte end marker (which is set to zero). All the SRP messages have an Ethertype value of 0x22EA and are sent to the broadcast MAC address 01-80-C2-00-00-20.

Offset	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
0	Protocol Version								Attribute Type							
2	Attribute Length								Attribute List Length							
4	Attribute List Length								Vector Attribute							
6	Vector Attribute															
(n-4)	End Marker								End Marker							

TABLE 3.3: SRP Common Message Format

A brief explanation for the various parameters in the SRP header are specified below

- **Protocol Version** indicates the version of the SRP protocol used
- **Attribute Type** indicates which attribute is being defined in the message
- **Attribute Length** The length of the whole message including the header and body
- **Attribute List Length** specifies the length of the attribute list (body) of the message

- **Vector attribute** defines the attributes for the particular attribute type
- **End Markers** mark the end of the attribute types and the end of the message

The various different attribute types along with their numeric values, their different parameters and uses are described in the table below

SRP Message	Type	Description and parameters
Talker Advertise	0x01	➢ The attribute type Talker Advertise is used to advertise the stream parameters from the talker ➢ The parameters include the stream id, the destination MAC and VLAN address, Maximum number of frames, stream priority and rank
Talker Failed	0x01	➢ The talker failed vector is used to indicate the talker advertisements that are unable to be accepted because of resource restrictions in any of the bridges in the stream path
Listener Advertise	0x02	➢ The listener advertisement attribute type is used by the listeners to indicate interest in the advertised streams ➢ The listener advertisements carry only the stream id as a parameter for the response ➢ Apart from the stream id the listener advertisement also contains the status of the command which can be either one of the following values ➢ Ready indicates that for all the Listeners requested t to attach to the stream sufficient resources are available ➢ Ready failed indicated that there are more than one Listeners requested to attached to the stream, but not all requests can be reserved ➢ Asking failed indicates that none of the Listener advertisements that requested to attach to the stream was successfully because of missing resources
Domain	0x03	➢ The domain vector identifies the domain boundary within which the Stream Reservations are valid

Table 3.4: SRP Attribute Types

3.3.3 Forwarding and Queuing for Time-Sensitive Streams (FQTSS)

The FQTSS [20] protocol is used to enhance the forwarding and queuing behavior of end stations and bridges for time sensitive streams, so that the data for the time sensitive streams are not buffered too long so that they meet their time constraints when transmitted from one end station to another. To meet these requirements the following information are required.

- The stream reservation protocol boundaries are known so that the bridges know which traffic to prioritize and which not.
- The actual bandwidth requirement for the stream and the association of the bandwidth to the size for a single frame in the stream transmission.
- The maximum bandwidth that is available for the outbound queue at each of the ports in the bridge and the maximum bandwidth already reserved in the outbound queue of the port.
- The priority of the audio frame transmitted for each traffic class so that the credit based traffic shaper can order the priorities for the frames in the outbound queue.

During operation the talker end station has to transmit data streams such that the bridges can associate the data frame priorities to the respective traffic classes. This information is used in the credit based traffic shaper in the bridges to prioritize the required traffic so that they meet the time constraints. The only requirement from the listener end station is to be capable of buffering the amount of data required to reduce any jitter in the transmission.

3.3.4 AVB Discovery, Enumeration, Connection management and Control (AVDECC)

The AVDECC [21] protocol is a comprehensive protocol used to manage several aspects of the AVB network and helps devices in the network to know the availability and

capabilities of the other devices and manage them as required. The main operations of the AVDECC protocol are as followed,

- **Device Discovery** functionality enables the devices in the AVB network to know about the availability of other devices in the network, when they are ready, when they are departing etc...
- **Enumeration** functionality enables a device to learn about the various features and capabilities of any other device in the current AVB network
- **Connection management and control** functionality enables a controller device to connect streams between talkers and listeners and then control the various features of the devices to control the streaming

The AVDECC protocol uses the AVTP control command format to define the various command and responses. The format of the common command header format is defined as follows

Offset	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
0	sub type								sv	ver			msg type			
2	status								Data length							
4	Stream Id															

Table 3.5: AVDECC Common Header

All the functionalities in the AVDECC protocols uses the common header above which is derived from the AVTP common control header. The parameters in the AVDECC Common Header are described in details below

- **Subtype** is a 8 bit value which indicates which functionality this command belongs to. The following options are possible
  - 0xFA - AVDECC Discovery Protocol (ADP)
  - 0xFB - AVDECC Enumeration an Control Protocol (AECP)

  - 0xFC - AVDECC Connection Management Protocol (ACMP)

- **Stream Valid** is a single bit value that can be ignored and set to 0
- **Version** is a 3 bits value indicating the version of the protocol used
- **Message Type** is a 4 bit value indicating the type of the message received
- **Status** is a 5 bit value used to report the result of the requested operation
- **Data length** indicates the length of the command excluding the Ethernet header
- **Entity Id** identifies the entity to which this command is directed to

An AVDECC entity could choose to implement four different possible roles of operation. These roles are defined briefly below,

- **AVDECC Controller** is an entity role where the entity initiates commands to other entities to create and manage streams
- **AVDECC Talker** is an entity role where the entity acts as a source for media streams.
- **AVDECC Listener** is an entity role where the entity acts as a sink for media streams.
- **AVDECC responder** is an entity role where the entity has no specific operation, but just responds to any incoming AVDECC commands.

3.3.4.1 AVDECC Discovery Protocol

AVDECC device discovery is a process which follows the AVDECC Discovery Protocol (ADP), through which AVDECC entities in the network learn about the other devices in the network and when they are available and when they are departing. The ADP uses the common header format defined in Table 3.5. The body for the ADP command is specified through the ADP data unit (ADPDU) defined as follows

Offset	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
0	Entity Model Id																															
8	Entity Capabilities																															
12	Talker Stream Sources																Talker Capabilities															
16	Listener Stream Sinks																Listener Capabilities															
20	Controller Capabilities																															
24	Available index																															
28	gPTP Grand master Id																															
36	gPTP Domain No								Reserved																							
40	Identify Control Index																Interface Index															
44	Association Id																															
40	Reserved																															

TABLE 3.6: ADP Data Unit

Via the ADPDU given above an AVDECC entity can advertise it's presence in the network along with a basic set of it's features. It defines the version of the device firmware via the Entity Model Id and the features of the various aspects or roles of the entity via the various capabilities fields. Also it provides the gPTP grand master Id and domain number so that the network to which this entity belongs can be clearly understood. Apart from these important parameters there are also some feature specific parameters are also available. Based on the message type field in the AVTPDU common header defined in Table 3.5 there are three different ADP command which are listed below

- **Entity Available (0x00)** to indicate that the entity is available in the network and ready for configuration and streaming
- **Entity Departing (0x01)** to indicate that the entity is now not in operation and will soon be removed from the system
- **Entity Discover (0x02)** to trigger other entities (all entities when the requested entity id is 0 or the specific entity requested in the entity id) in the network to re-advertise themselves with the Entity Available command

The ADP commands contain the subtype as 0xFA, EtherType type as 0x22F0 and are transmitted to the multicast MAC address 91-e0-f0-01-00-00.

3.3.4.2 AVDECC Enumeration and Control Protocol

The AVDECC Enumeration and Control Protocol (AECP) provides two operations of which the first is to enumerate and discover the various capabilities, formats and controls of the AVDECC entity. The second is to control or modify these capabilities and formats to configure the entity to an operable state.

The AVDECC entity model describes the internal structure of the entity as a hierarchical collection of objects with each object specifying the details of it's parameters and the list of it's children and the hierarchical level where the object belong. An overview of the entity model is described in Figure 3.5 and the major objects are described briefly below.

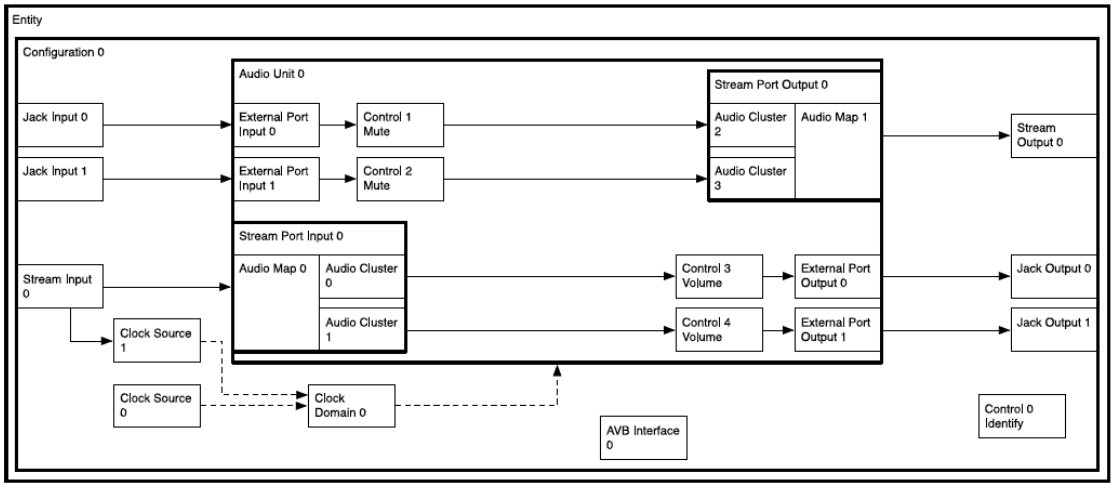

FIGURE 3.5: AVDECC Entity Model [6]

- The entity descriptor is the root level descriptor that defines the basic model and capabilities of the entity
- The configuration descriptor defines the one more configurations the entity supports and it further defines the other descriptors supported in each configuration. For example if an entity is capable of sourcing 8 channels of 48 KHz audio but only 2 channels of 192 KHz audio then it would present 2 different configurations for these two modes of operation.
- The stream input or output descriptors define the characteristics of the streams sourced and sinked in the entity. Here the range of parameter values of the audio

or video streams such as resolution, bit depth, number of channels, sampling rate etc... are defined.

➢ Further descriptors for the AVDECC entity model include the Jack, Clock domain, Sensor etc... descriptors each defined separately for each configuration supported by the entity

The AECP uses several protocol data units based on the common AECPDU fomrat defined below

Offset	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
0	Controller Entity Id																															
8	Sequence Id																															
12 (N-4)	Payload data																															

TABLE 3.7: AECP Data Unit

Here the Controller Entity Id specifies the AVDECC controller which initiates this request and the sequence id is used to track the multitude of requests and to correctly match a command with a response.

The two main commands used for the enumeration of the entity model are AEM_COMMAND and AEM_RESPONSE which are used to query and return the descriptors of individual entity objects one by one.

The AECP commands contain the subtype as 0xFB, EtherType type as 0x22F0 and are transmitted to the unicast MAC address of the entity requested and the responses are sent back unicast to the MAC address of the initiator MAC address.

3.3.4.3 AVDECC Connection Management Protocol

The AVDECC Connection Management Protocol (ACMP) manages the stream connections in the AVB network by connecting and removing stream connections between the stream sources and the stream sinks in the system. The ACMP uses the common ACMP data unit (ACMPDU) for all the commands and responses, the structure of ACMPDU is defined below

Offset	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
0	Controller Entity Id																															
8	Talker Entity Id																															
16	Listener Entity Id																															
24	Talker unique id																Listener unique id															
28	Stream destination MAC																															
32																	Connection count															
36	Sequence Id																Flags															
32	Stream VLAN Id																Reserved															

TABLE 3.8: ACMP Data Unit

- ➢ The controller entity id identifies the controller device that is managing this connection
- ➢ The Talker entity id identifies the talker entity which sources the stream
- ➢ The listener entity id identifies the listener which provides a sink for the stream
- ➢ The talker and listener unique ids identify the particular stream from the several possible stream sources available
- ➢ The stream destination MAC specifies the multicast MAC address to which the streaming data would be broadcast
- ➢ The connection count counts the number of listeners already attached to the stream
- ➢ The sequence id is used to match the responses with the commands
- ➢ The stream VLAN id indicates the VLAN id of the network where the stream is being streamed

The ACMP uses the following set of commands along with the ACMPDU described above to start and stop streaming between the entities.

Value	Message Type	Description
0	CONNECT_TX_COMMAND	Command to connect the talker to the stream

1	CONNECT_TX_RESPONSE	Response from the talker for the connect request
2	DISCONNECT_TX_COMMAND	Command to disconnect the talker from the stream
3	DISCONNECT_TX_RESPONSE	Response from the talker for the disconnect command
4	GET_TX_STATE_COMMAND	Request to get the current state of the talker
5	GET_TX_STATE_RESPONSE	Response from the talker for the current state request
6	CONNECT_RX_COMMAND	Command to connect the listener to a stream
7	CONNECT_RX_RESPONSE	Response from the listener for the connect command
8	DISCONNECT_RX_COMMAND	Command to disconnect the listener from the stream
9	DISCONNECT_RX_RESPONSE	Response from the listener for the disconnect command
10	GET_RX_STATE_COMMAND	Request to get the current state of the listener
11	GET_RX_STATE_RESPONSE	Response for the request to ge the current state of the listener

Table 3.9: ACMP Message Types

The ACMP uses the above commands to setup a streaming matrix and start the streaming. Based on the sequence of commands used there are four different connection modes defined by the ACMP. The four modes of operation are defined in detail below

1. Fast Connect

    - Fast connect mode is used when the Listener entity has a saved state on boot-up.

- The Listener entity tries to connect the last stream source identified by the Entity and unique Id of the Talker entity.
- The Listener sends a CONNECT_TX_COMMAND towards the last saved Talker entity.
- If the Talker is present and sends a CONNECT_TX_RESPONSE, the connection is established and streaming starts.
- If there is no reply the Listener waits for the Talker to be available again in the network by listening to the ADP advertise command.
- When the Talker is available again (identified through ADP), the Listener again sends the CONNECT_TX_COMMAND and after the Talker responds directly proceeds with the MSRP and streaming.
- If the Talker is changed in the network the Controller can delete the saved mode by sending DISCONNECT_RX_COMMAND

2. Fast Disconnect

   - Fast disconnect occurs when the Listener is executing a clean power-off when the stream is still active.
   - Before the power-off the Listener device remembers the connect Talker entity identity and sends a DISCONNECT_TX_COMMAND towards the Talker.

3. Controller Connect

   - Controller connect mode is normally used to setup a new connection for the first time.
   - The controller first sends a CONNECT_RX_COMMAND towards the Listener, then the Listener will send a CONNECT_TX_COMMAND towards the Talker.
   - When the Listener receives the CONNECT_TX_RESPONSE it can start the MSRP and then the streaming.

4. Controller Disconnect

   - Controller Disconnect is the normal mode of operation for terminating a stream for a established connection.

- For the controller disconnect, the controller sends a DISCONNECT_RX_COMMAND towards the talker, which in turn removes the saved state and sends a DISCONNECT_TX_COMMAND towards the Talker.
- When the Talker responds with a DISCONNECT_TX_RESPONSE for the disconnection request the connection terminated.

The ADP commands contain the subtype as 0xFC, EtherType type as 0x22F0 and are transmitted to the multicast MAC address 91-e0-f0-01-00-00.

3.3.5 Audio Video Transfer Protocol (AVTP)

The Audio Video Transfer Protocol (AVTP) [6] enables time synchronized transfer of audio, video and control data in a Audio Video Bridging (AVB) or Time Synchronized Networks (TSN). AVTP uses the AVB protocols such as Generalized Precision Time Protocol (gPTP), Multiple Stream Reservation Protocol (MSRP) and Framing and Queuing Enhancements for Time Sensitive Streams (FQTSS) etc... AVTP specifies the formats for audio video transfer along with the required information from the protocols such as gPTP, MSRP to reproduce the audio video signals exactly with time synchronization in the Listeners. AVTP protocol specifies one Talker device streaming data for one or more Listener devices. AVTP data is usually encapsulated in IEEE 802.3 Ethernet networks and it is also possible to encapsulate it in IEEE 802.11 wifi networks and inside IP protocols.

AVTP uses the concept of presentation time to achieve time synchronization between the Talkers and Listeners. The presentation time is defined as the gPTP time specified in nanoseconds at which point the audio video samples are to be presented at the Listener to the user. Even with the enhancements for a standard quality of service for AVTP streaming, it is still possible for the individual packets in the stream arrive at a variable time points. The presentation time should be chosen such that to smooth out these variations and at the same time low enough to be considered real time.

The AVTP transfers the data as the AVTP data unit (AVTPDU) of which there are three different types. The streaming data uses the AVTPDU stream header, the control applications uses the AVTPDU control format and the rest of the applications use the

AVTPDU alternate header format. The AVTPDU streaming header format is specified below and it's various parameters are discussed in detail

Offset	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
0	Subtype								sv	ver			mr	fsd		tv	seqno								Format specific 1							tu
4	Stream Id																															
12	avtp time stamp																															
16	format specific data 2																															
20	stream data length																format specific data 3															
24	Stream data payload																															

Table 3.10: AVTP Data Unit

- The 8 bit subtype field specifies the actual media data present along with it's encoding type such as IEEE 61883, AVTP Audio Format (AAF), Compressed Video Format (CVF) etc...

- The 1 bit stream valid (sv) field indicates if the stream id present contains a valid stream id or not.

- The version information specifies the protocol version of the AVTP protocol as a 3 bit value.

- The 1 bit media clock restart (mr) field indicates that there is a media source clock change at the Talker

- The 1 bit Time stamp valid (tv) field indicates that if the avtp time stamp field contins a valid time stamp.

- The sequence number field is a single octet filed of a running count of the AVTPDUs. It is incremented by the Talker and used by the Listener to identify lost frames.

- The single bit time stamp uncertain field indicates that the time stamp specified is not accurate and Listener has to handle the uncertainty in time media clocks and prevent any unwanted disturbances.

- The stream id field specifies the unique identifier that identifies the stream. It is composed of the 48 bit MAC address of the stream source entity and a 16 bit unique identifier.

- The AVTP time stamp specifies the nanoseconds value of the gPTP synchronized time. The time stamp is calculated from the gPTP time using the following formula $ts = ((sec * 10^9) + (ns)) mod 2^{32}$ where sec is the seconds part and the ns is the nano second part of the gPTP time.

- Stream data length contains the length of the stream data paylod field which is the size of the entire AVTPDU minus the stream header.

- The Stream data payload field contains the actual media data to be transferred. The format of this filed is subtype specific and it's length is generally limited by the maximum size of a frmae in the underlying transport protocol.

- The three format specific fields are individually defined for every subtype and carry the stream specific parameters

AVTP Audio Format (AAF) encapsulation is a audio only format used to stream audio as an uncompressed stream. The header for the AVTP Audio Format (AAF) encapsulation is as follows

Offset	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
0	Subtype								sv	ver			mr	rsv		tv	seqno								rsv							tu
4	Stream Id																															
12	avtp time stamp																															
16	format								nsr				rsv				channels per frame								bit depth							
20	stream data length																rsv				sp	evt			reserved							
24	Stream data payload																															

TABLE 3.11: AVTP Data Unit - AAF

The AVTPDU for the AAF encapsulation uses the same stream header common for all the AVTPDUs. Further it defines some more parameters specific to the AAF encapsulation which are described below.

- The format field specifies which kind of data format does the audio samples have. Two major kinds are defined namely PCM and AES3.

- The nominal sampling rate filed specifies the sampling rate of the audio stream which possible standard values from 8 kHz to 192 kHz.

- The channels per frame field indicates the number of channel samples present in each frame.

- The bit depth specifies the number of bits used to encode each audio sample.

- The payload data field contains the actual audio samples packed according the parameters specified above.

For example a 16 bit PCM with 8 channel audio frame is packed as follows

Offset	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
0	Sample 1 of Channel 1																Sample 1 of Channel 2															
4	Sample 1 of Channel 3																Sample 1 of Channel 4															
8	Sample 1 of Channel 5																Sample 1 of Channel 6															
12	Sample 1 of Channel 7																Sample 1 of Channel 8															
16	Sample 2 of Channel 1																Sample 2 of Channel 2															
20	Sample 2 of Channel 3																Sample 2 of Channel 4															
24	Sample 2 of Channel 5																Sample 2 of Channel 6															
28	Sample 2 of Channel 7																Sample 2 of Channel 8															
32	Sample 3 of Channel 1																Sample 3 of Channel 2															
36	Sample 3 of Channel 3																Sample 3 of Channel 4															
40	Sample 3 of Channel 5																Sample 3 of Channel 6															
44	Sample 3 of Channel 7																Sample 3 of Channel 8															

TABLE 3.12: AAF 8 channel 16 bit PCM payload

Chapter 4

Implementation

The proposed AVB solution uses the BeagleBone Black or the BeagleBoard x15 as development platforms. No additional hardware is required since the BeagleBoards have an inbuilt Ethernet port with hardware time stamping support from the Texas Instruments AM335x Sitara processors. It also can be expanded by audio capes for analog or any traditional audio in and out. In software front the Debian distribution with a Linux kernel modified for the BeagleBone Black and BeagleBoard x15 is used. The design of the architecture and the details of the implementation are explained in detail in the following sections

4.1 Design

The general software architecture of the purposed AVB stack in the BeagleBoard Debian environment is described in the figure 4.1. It details the structure of the various modules in the implementation along with their interconnections explaining the data and control flow between them. The implementation is based on the Debian distribution based on a Linux kernel where there are two modes of software operation.

- The first is the kernel mode operation where the software executes in a privileged environment with full access to all the resource in the system. But development of software for kernel mode applications is complicated, not always backwards compatible and any issues in the kernel mode software can easily bring down the entire system.

➜ The second is the user mode operation where development is simple and portable but lacks several privileges for access to resources such as i/o blocks system memory etc...

The AVB solution described here has software modules implemented both in the user mode and also the kernel mode. The various modules and their mode of operation along with the arguments for the chosen mode of operation are described in detail. In the figure 4.1 the green coloured blocks are the modules developed under this implementation and the gray coloured modules are software modules already available in the default debian distribution for the BeagleBone Black and BeagleBoard x15 systems.

The individual development modules, developed as part of this project are listed below with a sort summary with the detail explanation of the design and implementation details of these modules are provided in the further sub sections.

1. gPTP Daemon - A Linux Daemon implementing the generalized precision time protocol for time synchronization between nodes.
2. ALSA AVB Driver - A Linux kernel virtual audio driver for the ALSA framework implementing the AVTP, AVDECC and MSRP protocols for the AVB Stack for a real time synchronized audio streaming.
3. A Test Application which uses the ALSA AVB driver to demonstrate a real time synchronized audio streaming.

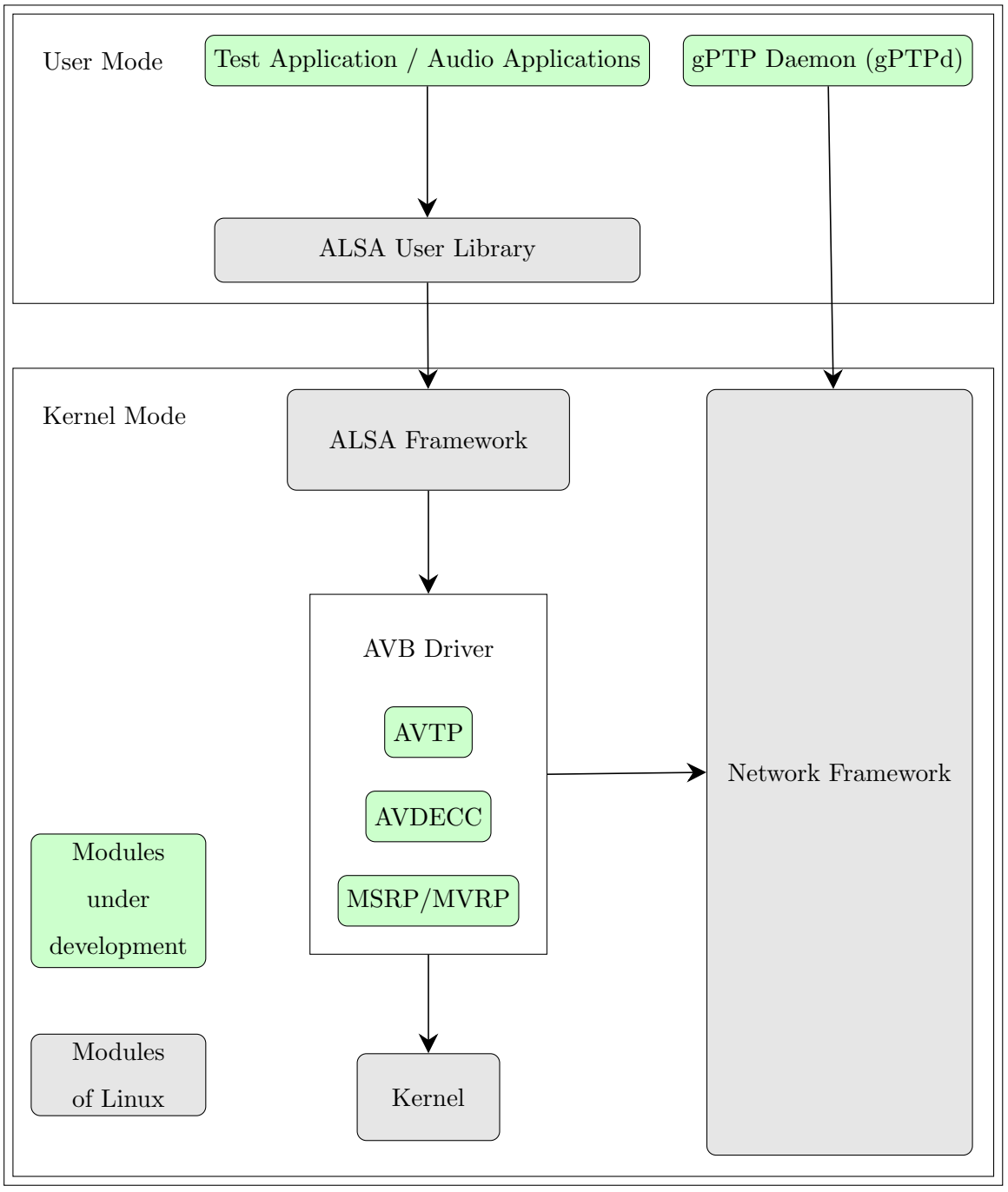

FIGURE 4.1: Software Architecture

4.2 gPTP Daemon

A Daemon is a software program that runs in the background without any possibility for user input and with no feedback to user of its operations. Normally the daemon programs are identified (in Linux systems) by a letter d at the end of the program

name such as httpd, dhcpd, syslogd for responding to HTTP protocol requests, manage DHCP connections and log the system events in a log respectively. Normally these daemon provide a particular service which is required for the entire life time of the system but with no input from the user. The same functionalists are known as system extensions or services in other operating systems. Although the implementation details may differ between different operating systems, the main objective and functionalities of the daemons are the same.

In Linux systems a daemon is created by the following steps programmatically. When a program starts it forks into a different process and then immediately exits the main process. Now the forked process is adapted by the init process of the Linux and now runs in the background with it's standard input output file descriptors closed and so no further input output possible.

The gPTP daemon is such a daemon implemented to execute the various protocols in the gPTP specification such as peer delay measurement, best master selection and time synchronization as described in 3.3.1. The implementation of the gPTP daemon is done completely done with 'C' programming with the individual algorithms executed as state machines. State machine design pattern implements a state machine description of an algorithm. State machines describe the operation of an algorithm by defining a number of states of operation and a list of events which trigger specific operation and optionally might trigger a state transition to a different state.[16]

Before describing the individual state machine and their operations, the main flow of operation of the gPTP daemon is illustrated in the diagram 4.2. The main functions in the gPTP flow diagram are described in detail below.

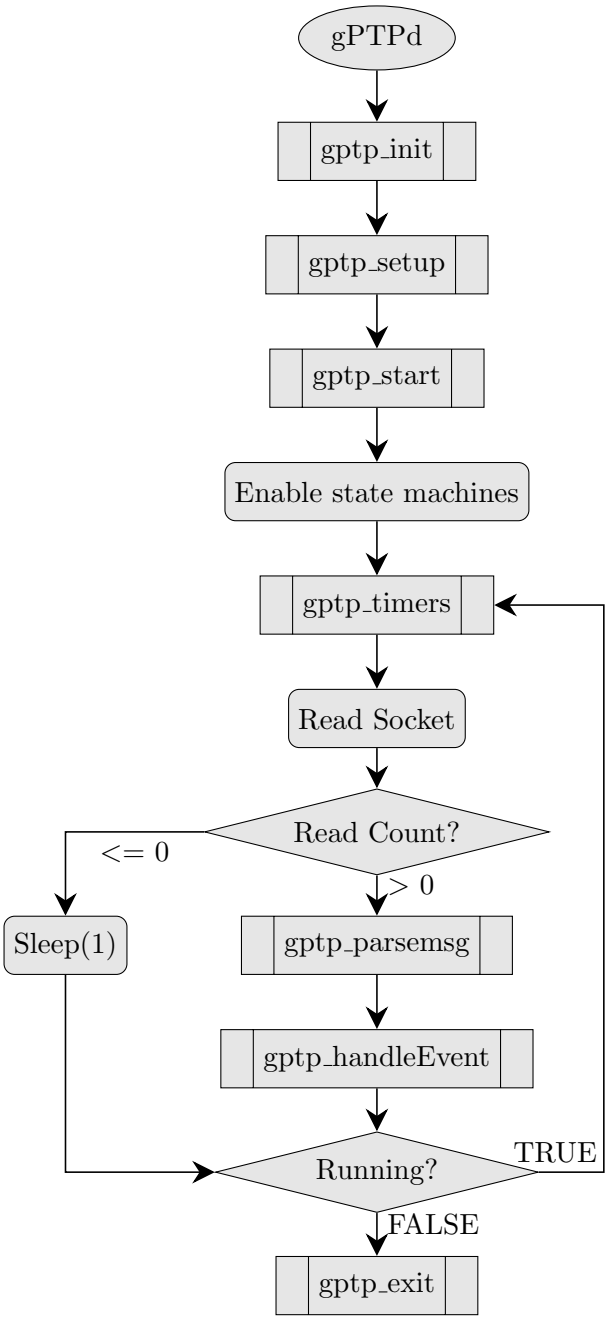

FIGURE 4.2: gPTPd - Main Flow Diagram

- **gptp_init** - This functions handles the initialization of the gPTP daemon. This functions handles the initialization of the variables and state machines. It puts the state machines into initial states. Also it parses the command line arguments provided when the program was invoked and stores the configuration for further use.

- **gptp_setup** - This function handles the process to create the daemon based on the configuration. It first forks the current process, closes the standard file descriptors

and exits the main process thus creating the daemon. The default configuration is to run as a daemon, but if the command line arguments specify to run as a normal application (for debugging or to demo the operation etc...), the steps to create a daemon are skipped and the program continues as a normal operation.

- **gptp_start** - This function mainly handles the initialization of the sockets which will be used for the Ethernet communication for the gPTP commands. First the sockets are opened and the socket options for the hardware time stamping (refer 3.1.1) are set and the socket buffers for transmission and reception are initialized. After this function the process enters a event loop which executes every 1 second and always processes the following functions.

- **gptp_timers** - This function is the first operation in the event loop. It checks if the any of the running timers that are started has now expired. These timers are mainly used to execute the gPTP commands which are to be executed in periodical time intervals. The inbuilt timers of Linux is not used as the timer resolution need not be precise and to keep the single threaded implementation simple. This function checks the time stamps of the last time when the timer expired to the current time stamp to determine which timers are expired.

- **gptp_parsemsg** - This function parses the socket data read from the socket to determine what gPTP command has been received. It then maps the received command as an event for the state machines. The command parameters are also parsed and stored separately for further use in the state machines.

- **gptp_handlemsg** - This functions is the main entry point for the state machines. The current event (either the timer event or the event from the received gPTP message) is passed on to the respective state machine for processing.

The state machines used in the gPTP daemon are described below with their respective state diagrams.

4.2.1 Peer Delay Measurement State Machine

The Peer delay measurement is the process in which a network node measures the propagation delay between it's network port to the network ports of its immediate

neighbor which is described in 3.3.1. The state machine of the peer delay measurement process is illustrated in figure 4.3. The states involved in the peer delay measurement are as follows

- Init - The initial state of the state machine
- Idle - The state where there are no operations i.e. Between the periodic delay measurement period and when there is no pending request from a neighbor device.
- WD_{resp} - The Delay Response Wait State is entered when the periodic timer for the PDelayReq command is expired and the PDelayReq command is sent and awaiting the response.
- WD_{flw} - The Delay Response Followup Wait state, where the PDelayResponse has been received but now waiting for the PDelayReponseFollowUp command.

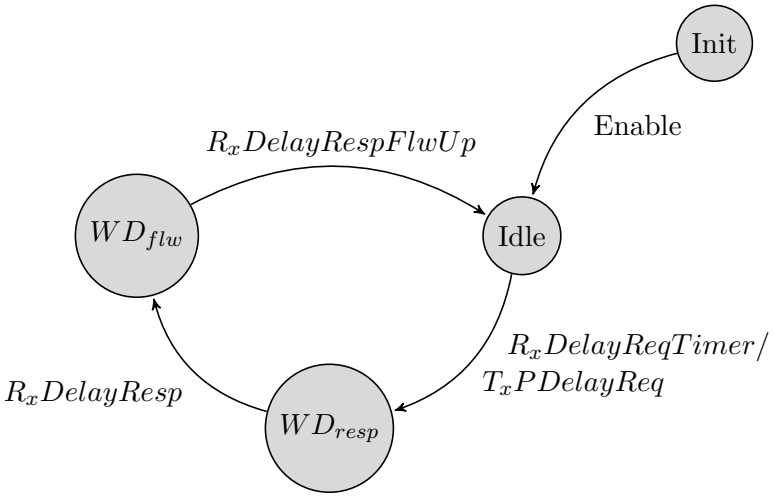

FIGURE 4.3: gPTPd - Peer Delay Measurement State Diagram

When the state machine transfers from the WD_{flw} state all the required time stamps for a delay estimation is available and the delay is calculated and stored. After the next timer event the process is repeated again and the delay is estimated again and updated.

4.2.2 Best Master Selection State Machine

The best master selection is the process in which the network nodes select a device as the grand master device among capable devices in the network as described in 3.3.1.

The state machine of the best master selection process is illustrated in figure 4.4. The states involved in the best master selection are as follows

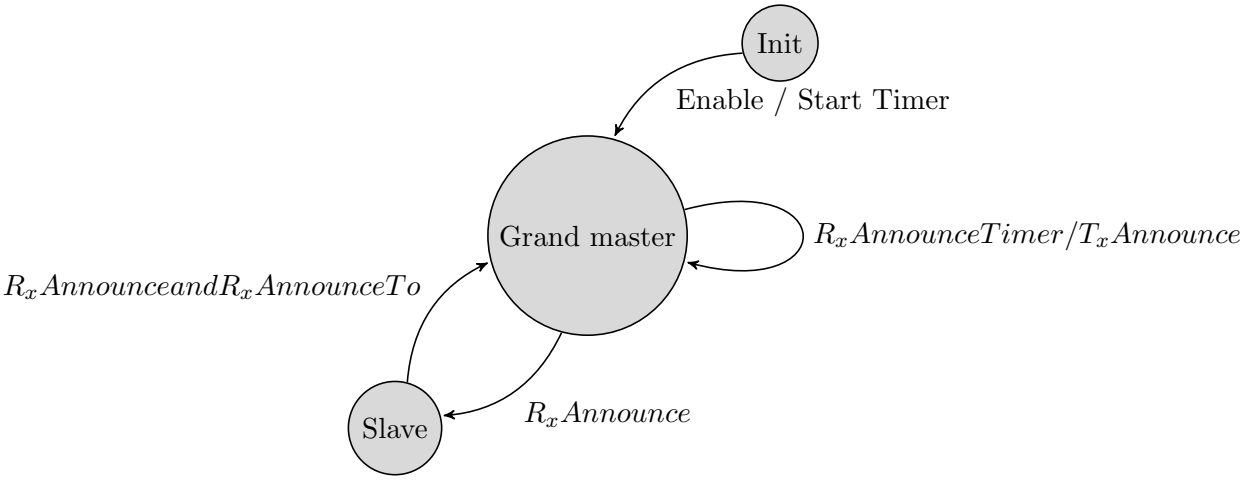

FIGURE 4.4: gPTPd - Best Master Selection State Diagram

- Init - The initial state of the best master selection algorithm.
- Grandmaster - The state where this device is selected as the grand master. This is the initial state when the process is started and it remains in the state until an announce message with a better clock or a high priority message is received at which point we transition to the slave state and remain there until there is a timeout for an announce message.
- Slave - The state when there is an another device in the network with a better or high priority clock. When there is no more high priority clock announce messages for some time then again transition to grand master state.

4.2.3 Time synchronization State Machine

The time synchronization is the process in which the current time is synchronized from the grand master to all the slaves in the network as described in 3.3.1. The state machine of the time synchronizartion process is illustrated in figure 4.5. The states involved in the time synchronization are as follows

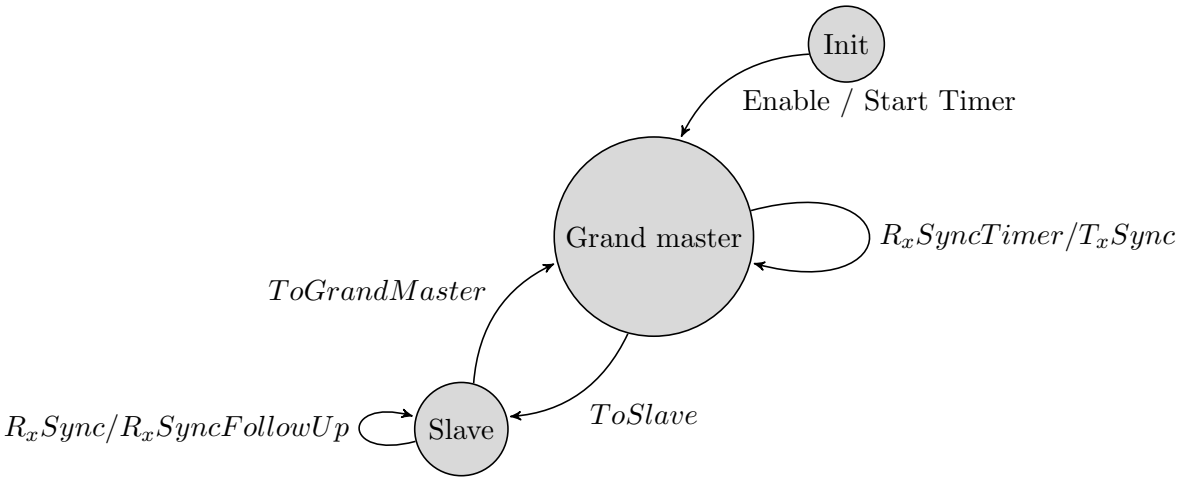

FIGURE 4.5: gPTPd - Time Synchronization State Diagram

- Init - The initial state of the time synchronization algorithm.
- Grandmaster - The state is entered when the best master algorithm selects this device as the grand master. In this state the Sync message with the current time is broad-casted periodically.
- Slave - The state entered when the best master algorithm selects some other device as the grand master. In this state upon reception of the Sync and Sync FollowUp messages the internal clock is adjusted to the grand master time by using the received time and measured propagation delay.

4.3 AVB ALSA Driver

Audio device drivers are software modules which enable the operating system to utilize audio hardware to stream audio in and out of the system. ALSA audio drivers are audio drivers that are developed as part of the ALSA driver framework where the interface to the user level software is the same irrespective of the audio device used. The ALSA framework provides an abstraction of the audio device to the user level applications. The AVB audio driver is termed as virtual audio driver since there is no physical audio hardware device instead the audio data is streamed out of the Ethernet port and received through it.

The general design of the AVB Virtual ALSA Audio driver along with it's internal modules and their sequence of operations are described in detail in the following sub sections.

4.3.1 Loadable Kernel Module

The AVB ALSA driver is developed as a loadable kernel module. Loadable kernel modules are a piece of software which is executed in the kernel space, but still is not part of the initial kernel code loaded at the time of booting. This loadable kernel code is loaded delayed at the time of normal operation when the function of this module is required. This module can be loaded via the command *insmod*, when invoked through this command this module is loaded into the kernel and executes as a kernel mode software. When the function is no more required this can be unloaded via the command *rmmod*.

Loadable kernel modules have to implement two functions, one for initialization at start-up and the other for cleanup before exit. The AVB driver implements the *alsa_avb_init* and *alsa_avb_exit* functions for initialization and cleanup. These functions are then registered with the kernel using the macros module_init() and module_exit() respectively. With these registrations if the program is configured as a loadable kernel module in the Linux kernel build system and compiled the resulting "snd_avb.ko" file becomes a loadable kernel module. When loaded using the insmod function of the kernel the program is loaded into the kernel memory and the initialization function (*alsa_avb_init*) is invoked, where the further initialization of the module can continue leading to normal operation. Once the module is removed using the rmmod function the cleanup function (*alsa_avb_exit*) is invoked where the resources used by the module are freed and processes stopped. When this function returns to the kernel, the module is removed from the kernel memory thus completing the unloading.

4.3.2 Platform Device Driver

The AVB driver follows the "platform device driver model" of Linux kernel device drivers. The platform device driver model is used for unconventional devices which are not connected to the established general purposes buses such as PCI, USB etc... The platform

devices are represented as stand alone devices connected to a pseudo bus inside the Linux driver framework. This driver model is used for legacy devices, system on chips, processors, integrated peripherals and virtual devices. To create a virtual platform device and an associated driver the following steps are executed (refer also the flow chart in figure 4.6) in the module initialization function *alsa_avb_init*:

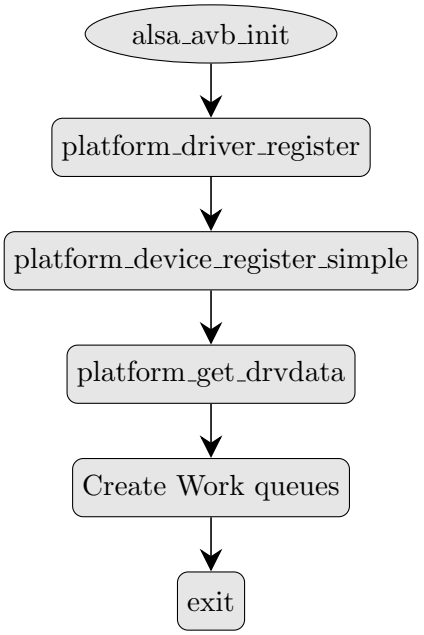

FIGURE 4.6: Platform device-driver initialization

- The driver is registered with the kernel using the function *platform_driver_register*, which passes the *platform_driver* type variable to the kernel. This variable contains the pointers for the probe and remove functions for the driver along with name and the supported features of this driver. The kernel uses the functions registered to probe for the hardware when the device is added and remove the driver when the device is removed.

- Then as this is a virtual device the device would be found or enumerated by the kernel and so the kernel has to be told that the device is present. This is done using the function *platform_device_register_simple* where the name of the device is passed along the index of the device.

- After the device is registered with the kernel, the device is activated by calling the function *platform_get_drvdata*. When this function is called, the probe function of the driver is invoked and thus the driver can start it's operation.

When the platform device has been successfully registered and the driver is loaded the local initialization steps can be executed. The one major initialization carried here is to create the work queues required for the driver operation. The work queues in Linux are way to defer some work for a later time to a kernel process. They differ from other forms of deferred executions such as task queues in the context where the deferred functions are executed. In the work queues the deferred functions run in a kernel process context and not in a kernel interrupt context which enables the deferred functions to execute any function including blocking functions. The following APIs are used to manage the work queues:

- *create_workqueue* is used to create a work queue instance in the kernel. The only parameter required to create a work queue is the name for the work queue.
- *INIT_DELAYED_WORK* is used to initialize a delayed work by specifying the function which has to be executed, the id of the delay work and the data which has to be passed to the deferred function.
- *queue_delayed_work* is then used to actually add the delay work to the work queue in the kernel. The added work is then executed deferred after the time provided has been expired.
- *cancel_delayed_work* is used to remove any queued but not yet executed work items from the work queue. Before the work queue is closed the pending items has to be cancelled using this function.
- *flush_workqueue* is used to clear all the work items that are started but not yet finished. This call makes sure that there are no unfinished work pending in this work queue. This has to executed before closing the work queue.
- *destroy_workqueue* is used to actually close the work queue and release all the resources associated with it.

When the device is to be removed the platform device and driver are de-registered in the clean up function for the module *alsa_avb_exit*. Also all the resources allocated for the driver has to be freed and cleaned up before the module is unloaded.

4.3.3 ALSA Audio Driver Framework

The platform device driver model discussed in the previous section explains the model how the hardware interface of the device is handled (device detection, enumeration, management of resource such as interrupt lines, i/o pins, memory etc... and control of the device) with in the kernel. There is also several models for the driver based on the interface exposed to the user mode programs. Some examples are character drivers, block drivers, network drivers. For audio device drivers in Linux there is one more model of driver namely, "ALSA Audio Device Driver". The AVB driver is based on this ALSA Audio Device Driver model. The ALSA audio framework manages all of these audio device drivers in a common way be providing a common framework to which all the drivers has to be adopted.

The initialization of an ALSA AVB sound driver is illustrated in the figure 4.7, where a new sound card is created and the various interfaces to the sound card are described and the sound card registered. The initialization of the AVB ALSA audio driver is executed in the alsa_avb_probe function where the kernel requests for the hardware detection and initialization. The following steps are involved in the initialization:

- At first a new sound card instance is created using the function *snd_card_new*. This functions allocates the required resources for the sound card operation.
- After a sound card instance has been created the interface to the sound card has to be described. ALSA supports several different interfaces such as PCM, MIDI, control, hardware specific etc...
- A new PCM instance is created using the function *snd_pcm_new* and then the capabilities of the playback and capture functionalists are set using the function *snd_pcm_set_ops* for both playback and capture functions. Here the range of audio parameters (sample rates, number of channels, format etc...) supported by the device are registered.
- A new hardware specific interface is created using the function *snd_hwdep_new*. This hardware specific interface is used to set and get the AVTP time stamps for the audio streaming.

- And finally the newly created sound card is registered with the kernel using the function *snd_card_register*.

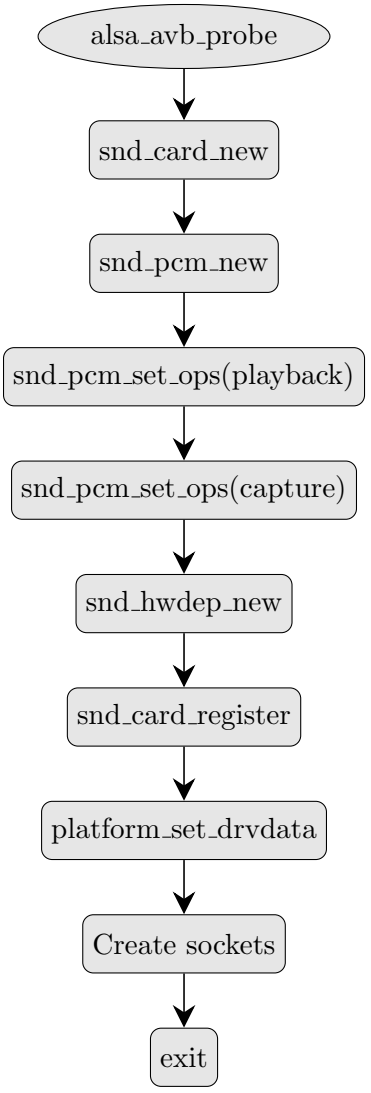

FIGURE 4.7: ALSA AVB audio driver initialization

When the sound card is created and registered, the card instance is set to the driver using the function *platform_set_drvdata*. This data is returned to the driver initialization call of *platform_get_drvdata* completing the driver initialization. After these steps the AVB specific initialization is carried out. One important step in the AVB specific initialization is to create the sockets for the Ethernet communication required for AVB. For AVB a raw socket is created for the first network interface (which is traditionally called "eth0" in Linux systems).

When the sound card is created, two interfaces for the sound card are defined and implemented namely the PCM interface and the hardware specific interface. Here the PCM interface is used for audio streaming in and out of the sound card and the hardware specific interface is used for implementing specific interface functions for this AVB ALSA sound driver. The more detailed information regarding the interfaces are explained in the following sub sections.

4.3.3.1 PCM Interface

The PCM interface for the sound driver is the main interface for the sound driver for streaming audio. Each sound driver can have at max up to four PCM instances. Each of these PCM instances can carry multiple sub streams each of which can be a playback stream or a capture stream. The PCM instance is created using the functions *snd_pcm_new()* and for the playback and capture streams, the reference to the functions that handle the PCM operations are registered via the function *snd_pcm_set_ops()*. More detailed explanation of the PCM interface and the required functions for the interface are explained in the section regarding the AVTP talker and listener in section 4.3.6

4.3.3.2 Hardware dependent Interface

The hardware interface for the sound driver is used to implement an interface for the unique features that are unique for the hardware. In case of the virtual AVB ALSA driver, this interface is used to provide an interface to AVB specific features that are associated with the audio streaming. More specifically this hardware specific interface is mainly used to exchange the synchronized time stamp to and from the user space audio application using this driver. The synchronized time (obtained from the synchronized hardware clock) at which a sample has to be presented at the listener is sent through this interface when the associated sample is sent via the PCM interface. In the other way the received presentation time received from the talker along with the audio samples are returned to the audio application along with the associated audio samples.

The hardware dependent interface is created using the function *snd_hwdep_new()* and then the functions required for the implementation of the hardware dependent interface

are registered. The minimum required functions to be implemented for the hardware dependent interface are listen below along with a short description:

- **_hwdep_open* - This function is used to create and initialize the required resources for the operation of the hardware specific interface functions. As for the AVB ALSA virtual driver there is no hardware no hardware specific initialization is required.
- **_hwdep_ioctl* - This function is used to handle the input output requests to the hardware specific interface. For each feature specific to the hardware, an ioctl id is declared and defined through which the hardware specific feature can be accessed. The following hardware specific ioctl ids are defined for the AVB ALSA driver:
  - *AVB_HWDEP_IOCTL_SET_TS*: This ioctl is used to set the presentation time stamp for the current audio frame that is being transferred. This ioctl has to be invoked when the associated audio frame is transferred from the user space. The time stamp has to be collected from the gPTP hardware clock and then any offset has to be added to generate the presentation time stamp.
  - *AVB_HWDEP_IOCTL_GET_TS*: This ioctl is used to get the presentation time stamp for the current audio frame received from the talker. This ioctl is to be invoked when the audio frame is read in to the user space. This presentation time is to be used to determine when the audio frame is to be rendered.
  - *AVB_HWDEP_IOCTL_SET_GRAND_MASTER*: This ioctl is used to set the grand master id for the AVB network. This ioctl is invoked from the gPTP daemon to set the id of the grand master device when the best master algorithm decides a device as the grand master.
- **_hwdep_release* - This function is used to release all the resources used for the hardware specific interface. Hence the AVB ALSA driver does not use any hardware specific resources no cleanup is required.

4.3.4 AVDECC Talker and Listener

The first requirement for the AVB device in the network is to discover and enumerate itself and other devices in the network. The AVB Device Discovery Enumeration Command and Control protocol is used 3.3.4.The AVDECC protocol is implemented in the work queue of the AVB driver, since the execution of this protocol has to be in background with no relation to the ALSA interface of the driver. The work queue part of the AVDECC implementation is illustrated in figure 4.8 and described in detail below:

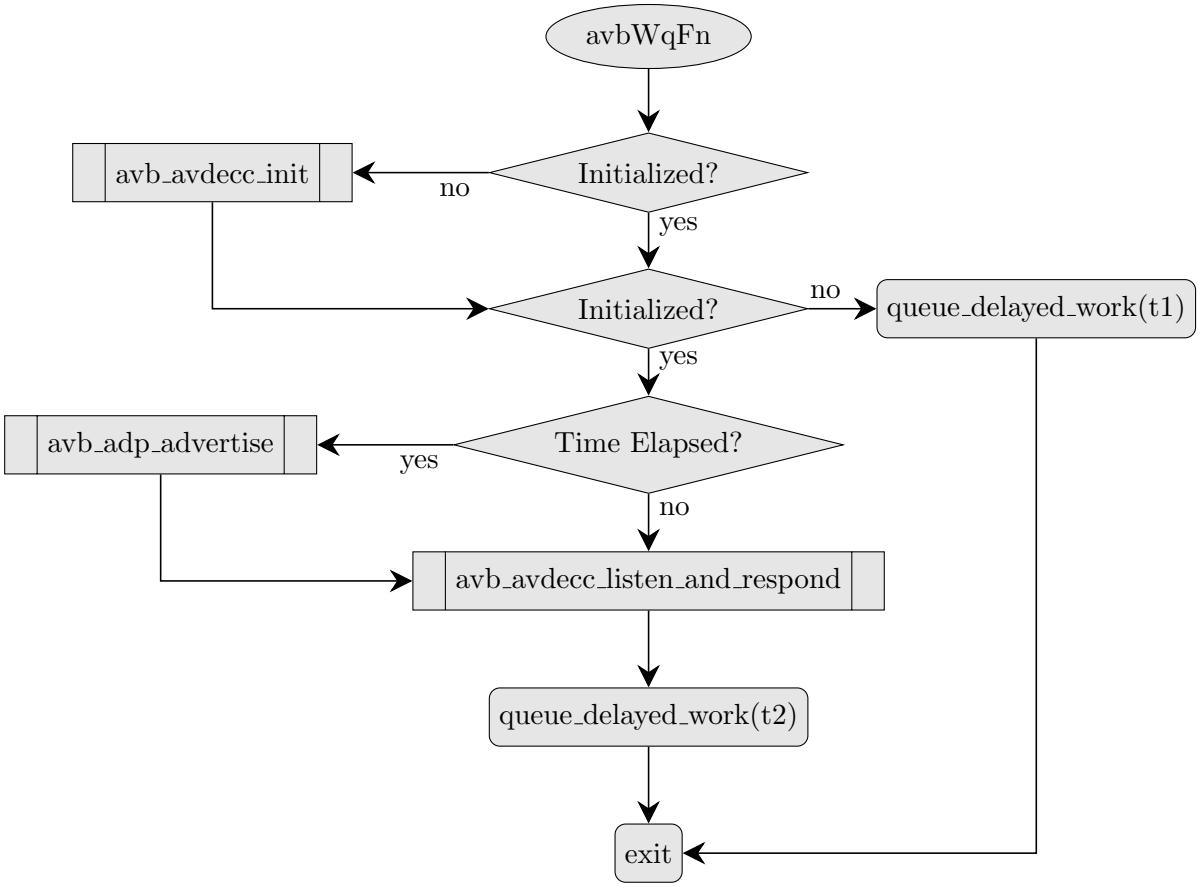

FIGURE 4.8: AVDECC Workqueue Flow Diagram

- When the work queue is invoked for the task of AVDECC, it is first checked if the initialization is completed, if not the initialization function "avb_avdecc_init" is executed, where the socket for the AVDECC communication is initialized.
- If for some reason the initialization failed the AVDECC work task is again queued using queue_delayed_work with a longer timeout (t1)

- For every cycle of the work task, after successful initialization first the transmit opportunity checked. The AVDECC advertise command for the device discovery is transmitted in periodic intervals using the avb_adp_advertise function.
- After optional transmission any incoming AVDECC commands are read and if required responses for the command are sent out. This is handled by the function avb_avdecc_listen_and_respond. The details of this command is described in detail below.
- At the end the work item is again queued in the work queue with a shorter time out t2 and when the work task is invoked again the whole process is repeated again.

The avb_avdecc_listen_and_respond function is comprised of several sub funtions to handle the entire AVDECC protocol. The full details are illustrated in figure 4.9 and described in detail below:

- At start the AVDECC socket is listened to and any pending data is read through the function avb_avdecc_listen. The socket read is read in blocking mode with a short timeout.
- When there is no data read the function returns immediately, but there is some data read the following actions are executed.
- The read data is interpreted as a AVTP common control header format and the AVTP sub-type is extracted. If this not a known valid AVTP sub-type then the function returns immediately. For known and valid sub-types further parsing continues.
- Based on the sub-type the functions avb_avdecc_*_respondToCmd are used to further parse and respond to the commands.

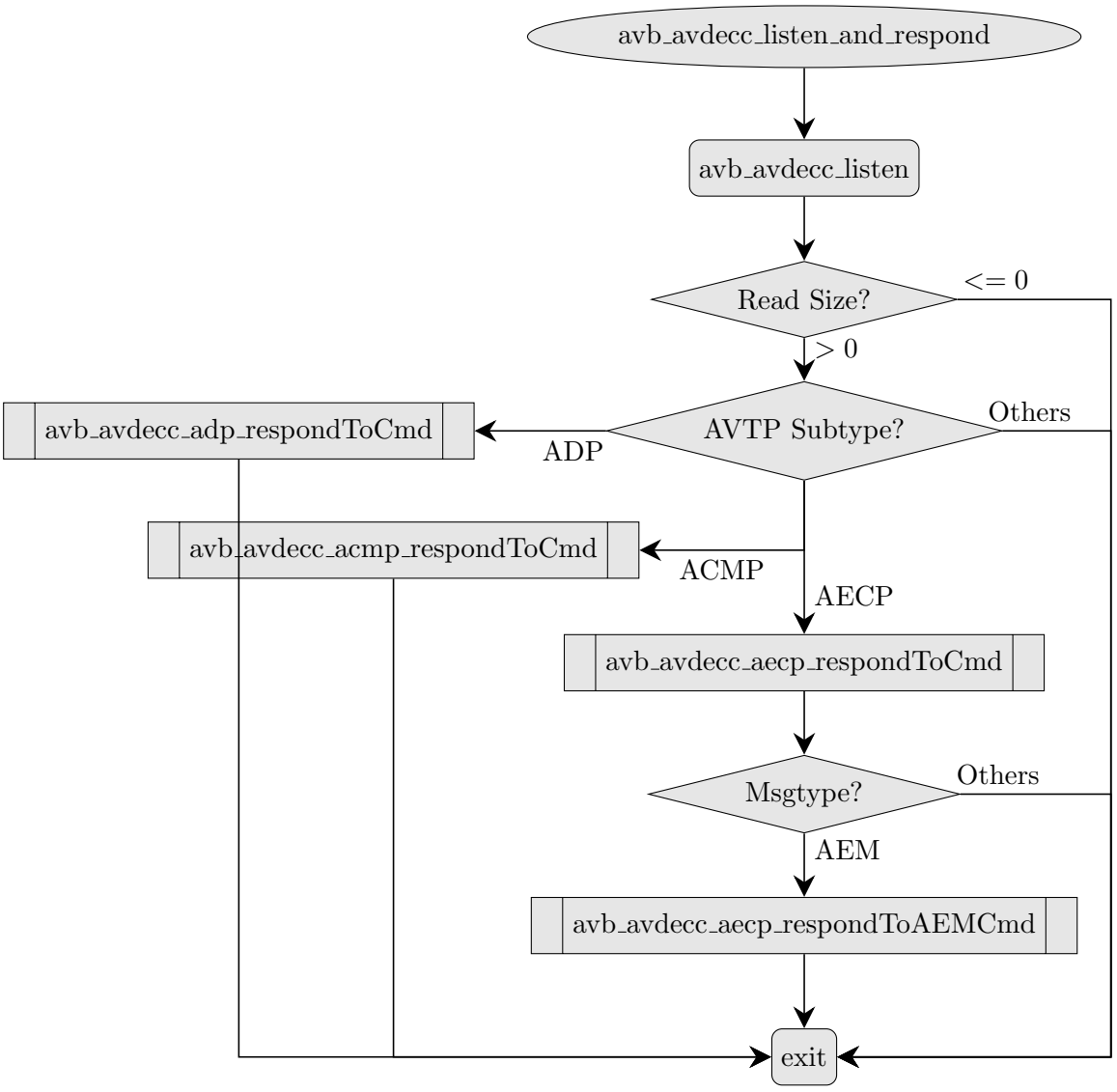

FIGURE 4.9: AVDECC Listen and Respond Flow Diagram

As described above, after every work task for AVDECC another work task is queued and thus the AVDECC listener is always available and responds to all the incoming AVDECC commands.

4.3.5 MSRP

The Multiple Stream Reservation Protocol (MSRP) which is used to reserve for network resources is implemented almost the same way as the AVDECC protocol implementation. It also is executed through work taks in the work queue. The MSRP protocol is initiated when the streaming is initiated by the ACMP protocol. The talker first sends the talker

declarations for the stream to be transmitted, waits for the responses and decides if the stream reservations is succeeded or not. In case of listeners the incoming talker advertisements are evaluated and listener responses are sent out. The work queue part of the MSRP implementation is illustrated in figure 4.10 and explained in detail below:

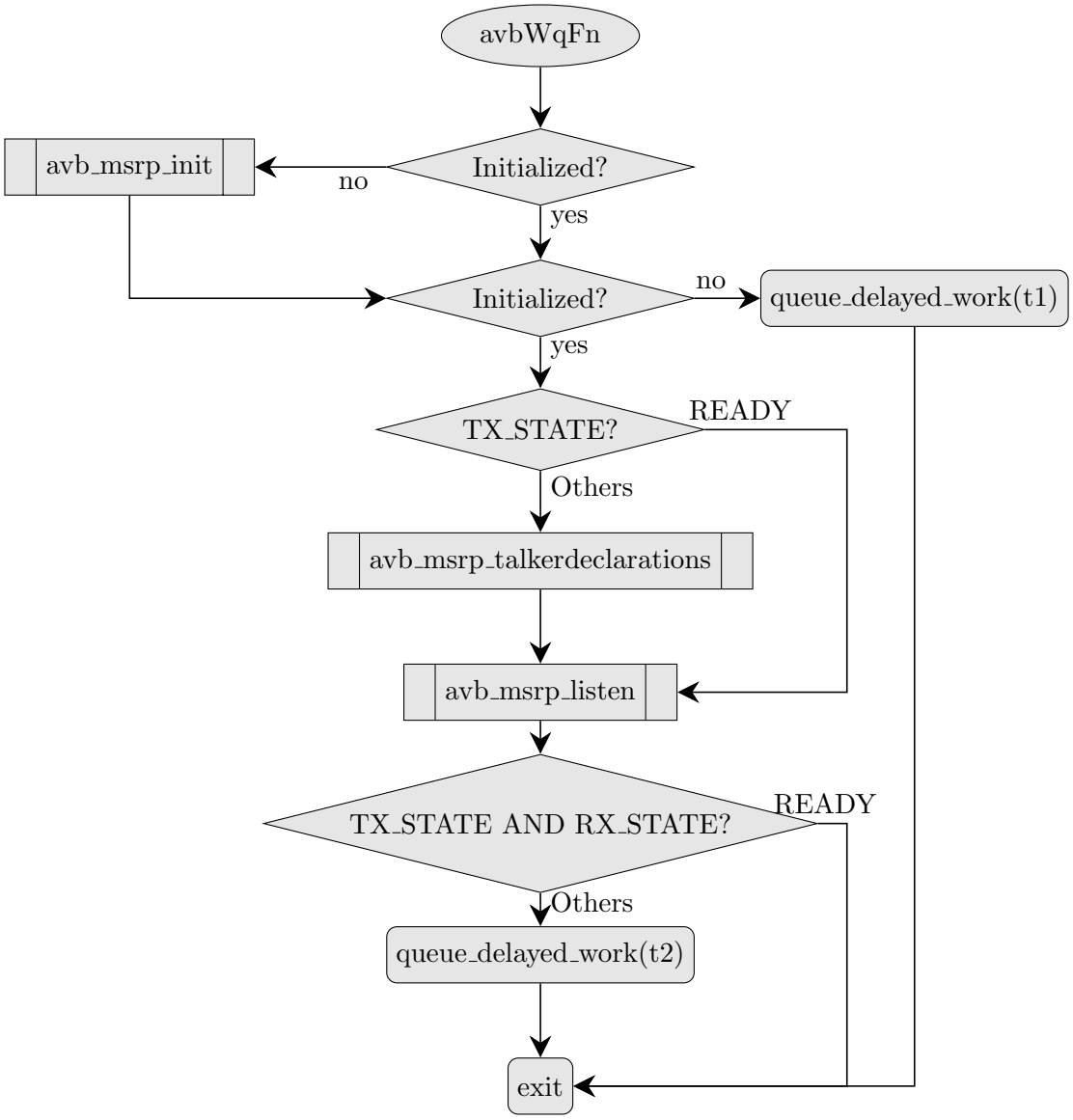

FIGURE 4.10: MSRP Workqueue Flow Diagram

- When the work queue is invoked for the task of MSRP, it is first checked if the initialization is completed, if not the initialization function "avb_msrp_init" is executed, where the socket for the MSRP communication is initialized.
- If for some reason the initialization failed the MSRP work task is again queued using queue_delayed_work with a longer timeout (t1)

- For every cycle of the MSRP work task, after successful initialization, the MSRP TX state is checked. When the state is not ready the MSRP talker advertise command is sent out for the current stream requirements using the command avb_msrp_talkerdeclarations.

- After the optional talker advertisements any incoming MSRP commands are read and if required responses for the command are sent out. This is handled by the function avb_msrp_listen.

- At the end the work item is again queued in the work queue with a shorter time out t2 and when the work task is invoked again the whole process is repeated again.

4.3.6 AVTP Talker and Listener

The Audio Video Transmission Protocol implementation inside the AVB driver is directly tied to the ALSA audio driver model. Hence in-order to describe the implementation of the AVTP talker and listener protocols a good understanding of the ALSA audio driver model and related APIs are required. The ALSA Audio driver model and the APIs it is made up are explained in detail below:

- ****_open*** - This function is invoked by the ALSA middle-ware when the audio driver is loaded. When this function is invoked the hardware device has to be opened and initialized.

- ****_close*** - This function in invoked by the ALSA middle-ware when the audio driver is unloaded. When this function is invoked the hardware device has to be closed and terminated.

- ****_hw_params*** - This functions is invoked by the ALSA middle-ware when the user application opens and configures the device for usage (set parameters such as sample rate, number of channels, bits per sample etc...). Here the actual audio configuration to be used is configured to the device and the resources are allocated and the device is set ready for streaming.

- ***_free** - This functions is invoked by the ALSA middle-ware when the user application stops the streaming and closes the device. Here the resources allocated for the streaming can be freed and the device can be put to idle.

- ***_trigger** - This function is used by the ALSA middle-ware to trigger certain events in the streaming. When the triggers are invoked the the device has to be controlled according to the event.

- ***_hw_pointer** - The ALSA middle-ware uses this function to request the position of the hardware pointer (i.e. the pointer of the audio buffer in the device up-to which the audio samples are processed). Based on this hardware pointer position the ALSA middle-ware either copies more data for playback use-cases and reads more data for recording use-cases.

- ***_copy** - This function is used by the ALSA middle-ware to copy the actual audio data to and from the device. The amount of data copied is configured at the time of configuration by the *_hw_params.

The functions of the driver listed above are not directly invoked by the user space application that is using the device. Instead the user invokes the ALSA middle-ware library which in-turn invokes the functions of the driver for the device currently under use. But in any case the normal usage of playback or capture the following general sequence is illustrated in the images 4.11 and 4.12 and described below:

- When the streaming starts the PCM interface is opened using the *avb_capture_open()* for audio capture or using the *avb_playback_open()* for audio playback. In the open functions the sub stream information is passed as an input argument which is a placeholder for all the information required for the playback or capture of the sub stream.

- Before either playback or capture the stream has to be configured i.e. various parameters such as sample rate, number of channels, frame size etc... are to be configured. For this purpose the either the function *avb_playback_hw_params* for playback or the function for *avb_capture_hw_params* for capture is used. Also after the parameters are configured the other resources required for the streaming are also initialized. For playback the following initialization are carried out:

  - The audio playback buffer for the hardware is allocated and initialized.
  - The AVTP header is initialized with the playback parameters configured for the stream.
  - The time stamp array which holds the presentation time for the audio frames to be sent out is initialized.
  - A timer is allocated according to the sample rate configured in-order to send audio frames in regular intervals. The high resolution timer of the Linux kernel is used for this purpose. The API *hrtimer_init()* is used for this purpose.

  Similarly for the capture function, the following initialization are carried out in the *snd_capture_hw_params()* function:

  - The audio capture buffer for the hardware is allocated and initialized.
  - The time stamp array which holds the received presentation time for the audio frames is initialized.
  - A delayed work is initialized and added to the work queue for the reception of the AVTP audio frames from the talker device. This delay work is receives the AVTP audio frames, decodes them and adds them to the allocated buffer.

- After the PCM is opened and configured, the streaming can be controlled by the functions *avb_playback_trigger* and *avb_capture_trigger* for playback and capture respectively. The commands received in the trigger functions such as

  - *SNDRV_PCM_TRIGGER_START*
  - *SNDRV_PCM_TRIGGER_STOP*

  are used to start and stop respectively the playback or capture of the stream.

- After starting the playback or capture using the trigger function, the ALSA middle-ware starts to fill the hardware buffer or read from the hardware buffer using the functions *avb_playback_copy* and *avb_capture_copy* respectively. After playback of audio frames or capture of audio frames, the driver advances the hardware pointer according to the number of audio samples played back or captured. The ALSA middle-ware uses the functions *avb_playback_pointer* and the *avb_capture_pointer* to know about the position of the playback and capture hardware pointer respectively. Based on this the ALSA middle-ware calculates the free space available in

the buffer (in case of playback) or the number of audio samples available in the buffer (in case of capture) and either copy more audio samples in the buffer (in case of playback) and copy the captured audio samples from the buffer (in case of capture) using the functions *avb_playback_copy* and *avb_capture_copy* respectively. This process is repeated until the streaming is stopped.

Note: As the playback and capture buffer inside the driver are allocated in the kernel memory space, in the functions *avb_playback_copy* and *avb_capture_copy* the audio frames in the kernel buffer can not be directly copied to or copied from to any user space buffers. As doing so would be a potential security violation as it provides access to kernel memory space to user level applications. To overcome this issue the kernel functions *copy_from_user* and *copy_to_user* are used respectively to read from and to write into user space buffers from the kernel buffer. These functions checks the validity of the user space buffer pointers, their access privileges and then setup the processor access rights to copy the data across kernel memory boundary.

- During playback at every invocation of the playback timer, a fixed number of audio samples are packed together and then the AVTP header is added on top. This AVTP frame is then transmitted over the Ethernet.
- Similarly when an AVTP frame is received, the AVTP header is extracted and validated. The total number of audio samples available is extracted from the header and the same number of audio samples are copied from the AVTP frame and then copied into the hardware buffer.
- To stop the playback or capture the ALSA middle-ware first stops the streaming using the trigger functions *avb_playback_trigger* and *avb_capture_trigger* and then frees the hardware resources using the functions *avb_playback_hw_free* and *avb_capture_hw_free*. Here the allocated resources such as the buffer are freed and the running timers, delayed work entries etc... are stopped and cleared. After the PCM interface is closed using the functions *avb_playback_close* and *avb_capture_close*.

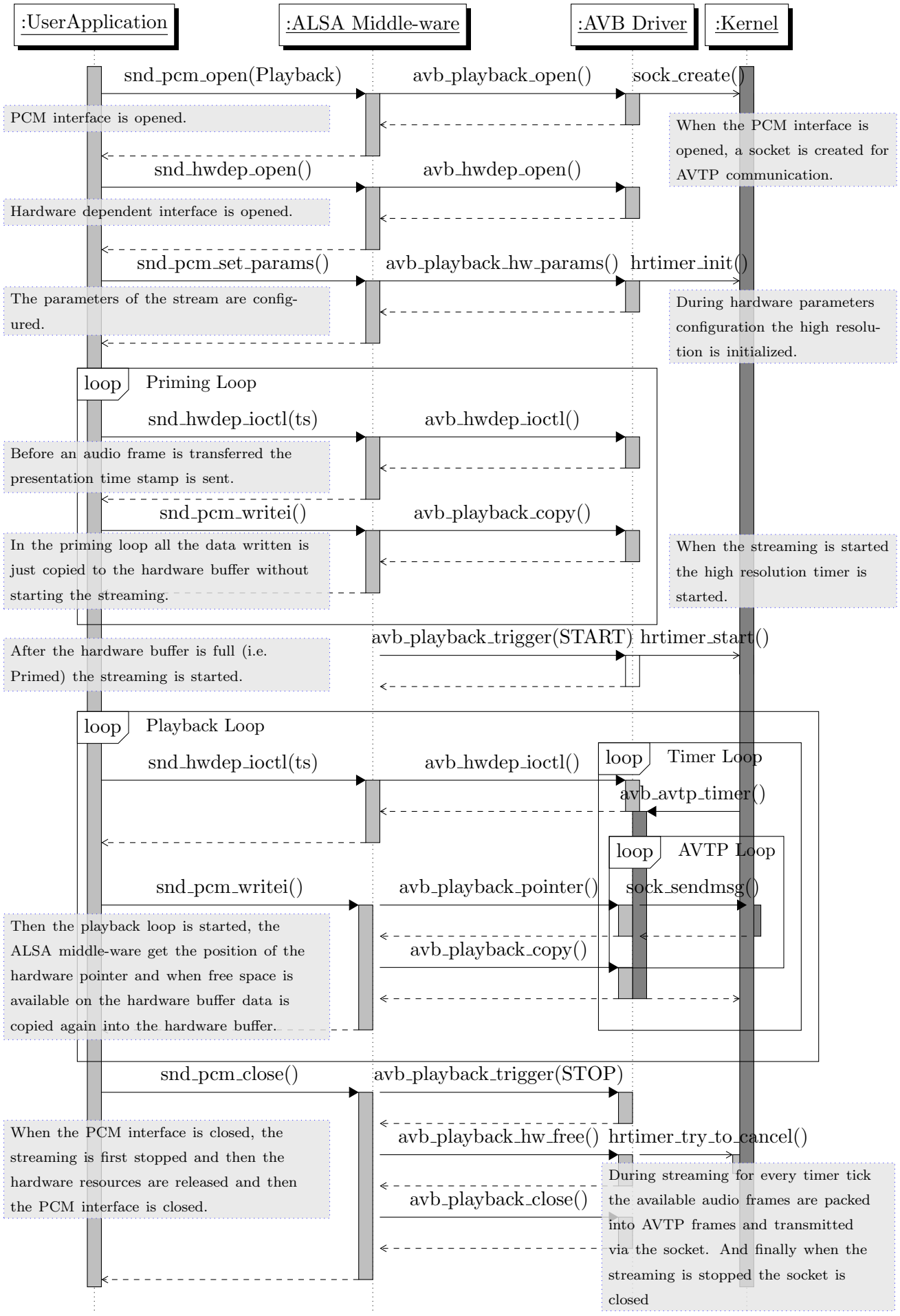

FIGURE 4.11: AVTP AVB Playback

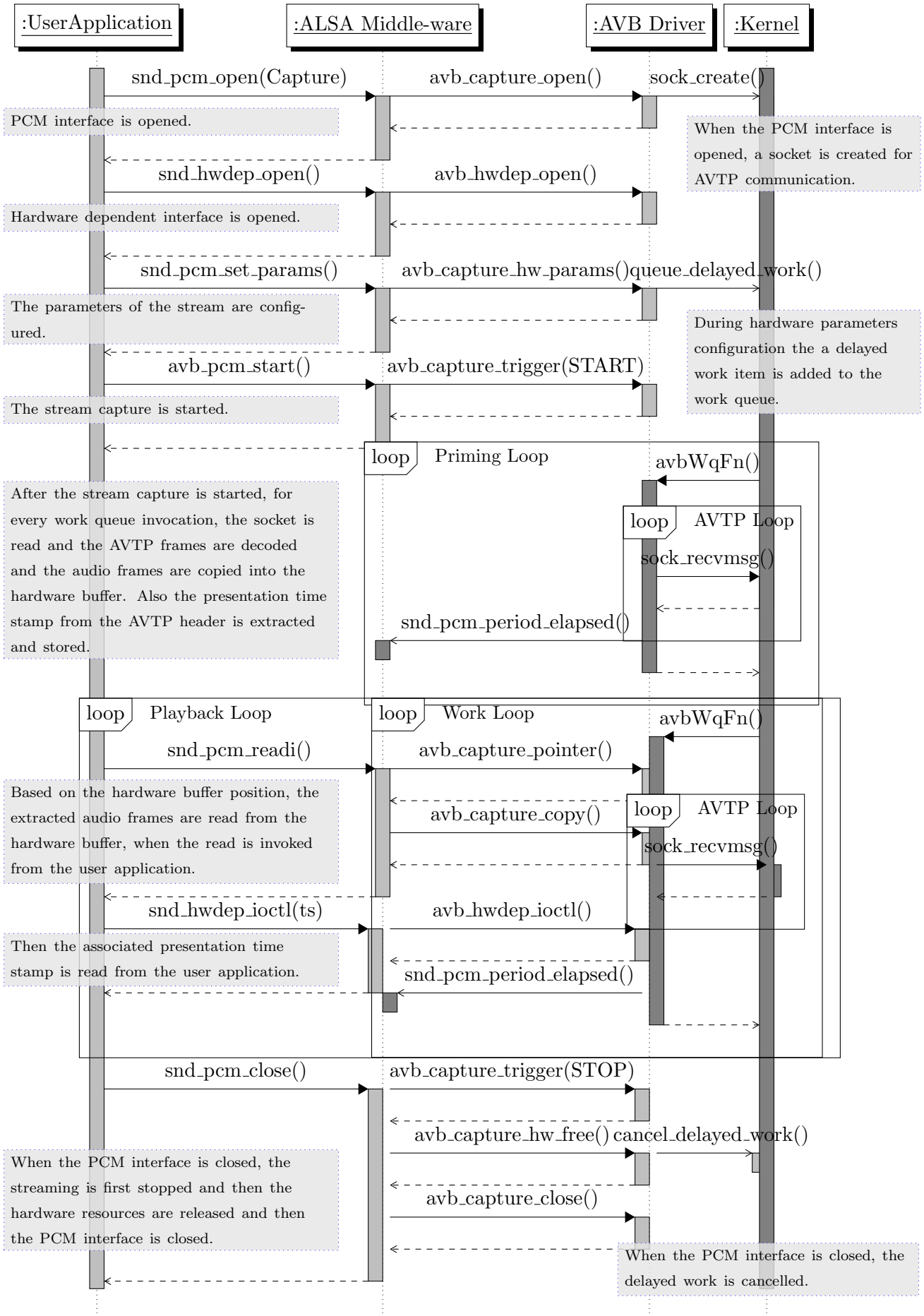

FIGURE 4.12: AVTP AVB Capture

4.4 Test Application

A test application is developed to demonstrate the various features of the AVB Stack developed in the BeagleBone Black and x15 platforms. The test application uses the common ALSA library interface to stream synchronized audio data via the virtual AVB sound card. It also uses the PTP hardware clock to get the time for the synchronized audio streaming. The general architecture, various features and limitations of the provided test application are discussed in detail in the following subsections.

4.4.1 Usage

The command line usage of the AVB test application is as given in the following syntax and it is also explained in detail below.

$avbtest - [p|r|a|b|x|y] - c < num\_channels > - s < sample\_rate > - n < num\_frames > - d < device\_name > - l < log\_level > < input|output\_file >$

- The first option is the operation mode of the test application. The following operation modes are followed:
  - *-p: Playback:* The AVB normal playback mode. The input file is streamed through AVB ALSA virtual device.
  - *-r: Record:* The AVB normal record mode. The audio stream received through the AVB ALSA virtual device is written as a wav file with the name given as the output file.
  - *-a: Demo Mode A:* Device 'A' operation for the demo setup 'AB'. More details in the section 4.4.2.
  - *-b: Demo Mode B:* Device 'B' operation for the demo setup 'AB'. More details in the section 4.4.2.
  - *-x: Demo Mode X:* Device 'X' operation for the demo setup 'XY'. More details in the section 4.4.2.
  - *-y: Demo Mode Y:* Device 'Y' operation for the demo setup 'XY'. More details in the section 4.4.2.

- *-c<num_channels>:* The number of channels to be recorded. To be used only with operation modes -r, -b, -y (i.e. where recording of AVB stream is required).
- *-s<sample_rate>:* The stream rate of the incoming stream to be recorded. To be used only with operation modes -r, -b, -y (i.e. where recording of AVB stream is required).
- *-n<num_frames>:* The total number of frames to be recorded from the incoming stream to be recorded. To be used only with operation modes -r, -b, -y (i.e. where recording of AVB stream is required).
- *-d<device_name>:* The device to be used for streaming if it is different from the default. Default AVB device: *"hw:CARD=avb,0"*. Default analog multi-channel device *"hw:CARD=C8CH,0"*
- *input—output_file:* The file name of the wave file that is to be played for operating modes involving playback. Or the file name under which the recorded audio data is to be written to the disk.

4.4.2 Features and Limitations

The features of the AVB test application generally revolves around the needed requirements to demonstrate the various features of the AVB ALSA virtual driver. The basic features are to use the AVB ALSA virtual driver as any other audio driver and to playback and record audio streams through it. And then there are other modes of operation which are used to test the time synchronization of the AVB streaming and to test the multi-channel and high sample rate streaming of the AVB ALSA virtual device. The demo modes along with their demo setup are explained in detailed below:

4.4.2.1 Demo Mode 'AB'

This mode is used to demonstrate the synchronized streaming of AVB with stereo streams. Two AVB devices running the AVB ALSA virtual driver is required for this demonstration setup. The basic idea of this setup that the device 'B' streams the given wavw file as a AVB stream with a presentation time starting with a pre-determined time

and the device 'A' will simultaneously plays-back the audio file specified through an analog audio card at the same pre-determined start time and receives the AVB stream and plays-back the AVB stream at the received presentation time through a different analog audio card or through the same audio card (second stereo channel of multi-channel card). With this setup the device 'A' will output two analog stereo streams both should be time synchronized according to the setup. To test the time synchronization between the two analog streams, they are fed into a test PC and recorded and tested. The test setup is illustrated below:

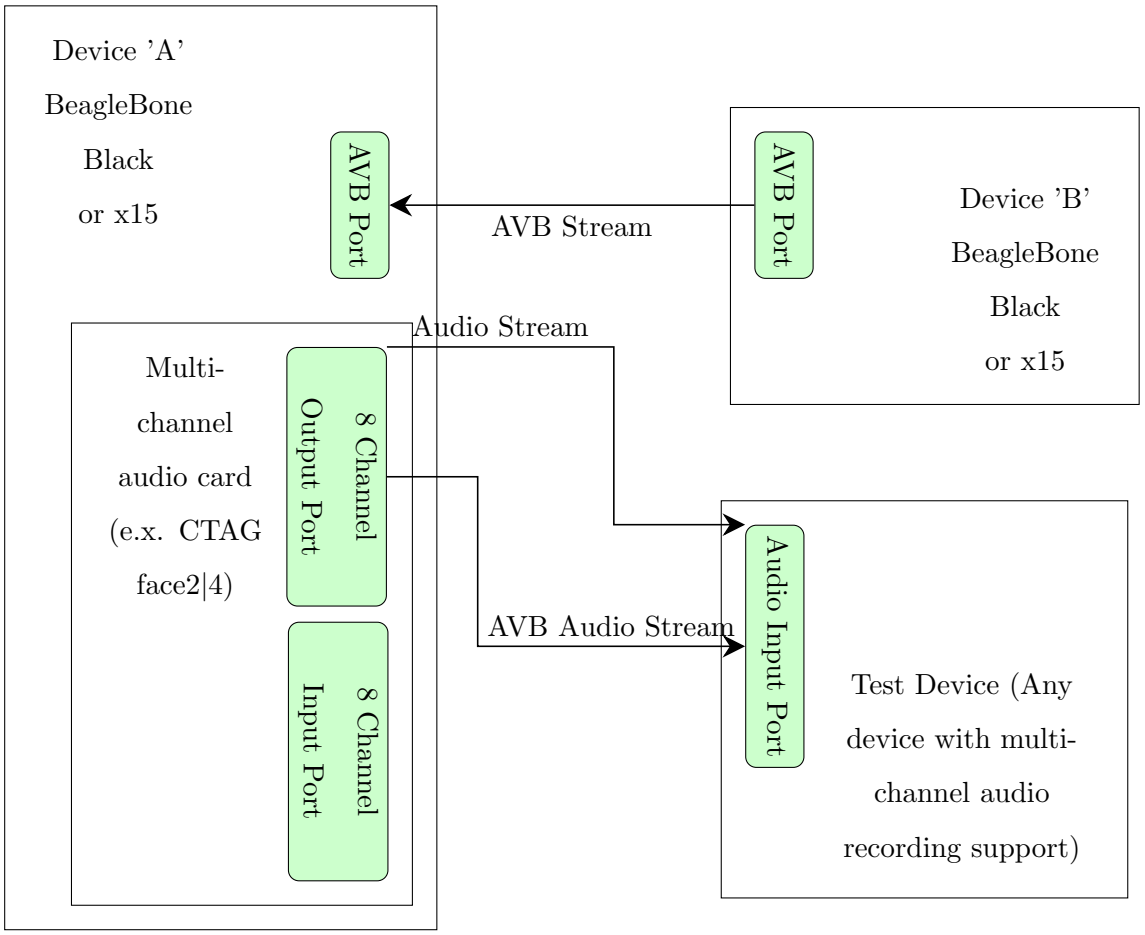

FIGURE 4.13: Demo Setup AB - Variation 1

A slight variation of this test setup is possible where the recording of the output analog streams are not done is a separate PC, but it is done in the same device 'A' through a multi-channel input audio card. Thus the audio analog streams are looped back into the device and recorded simultaneously as multi-channel recording where the synchronization can be tested. This variation is illustrated below:

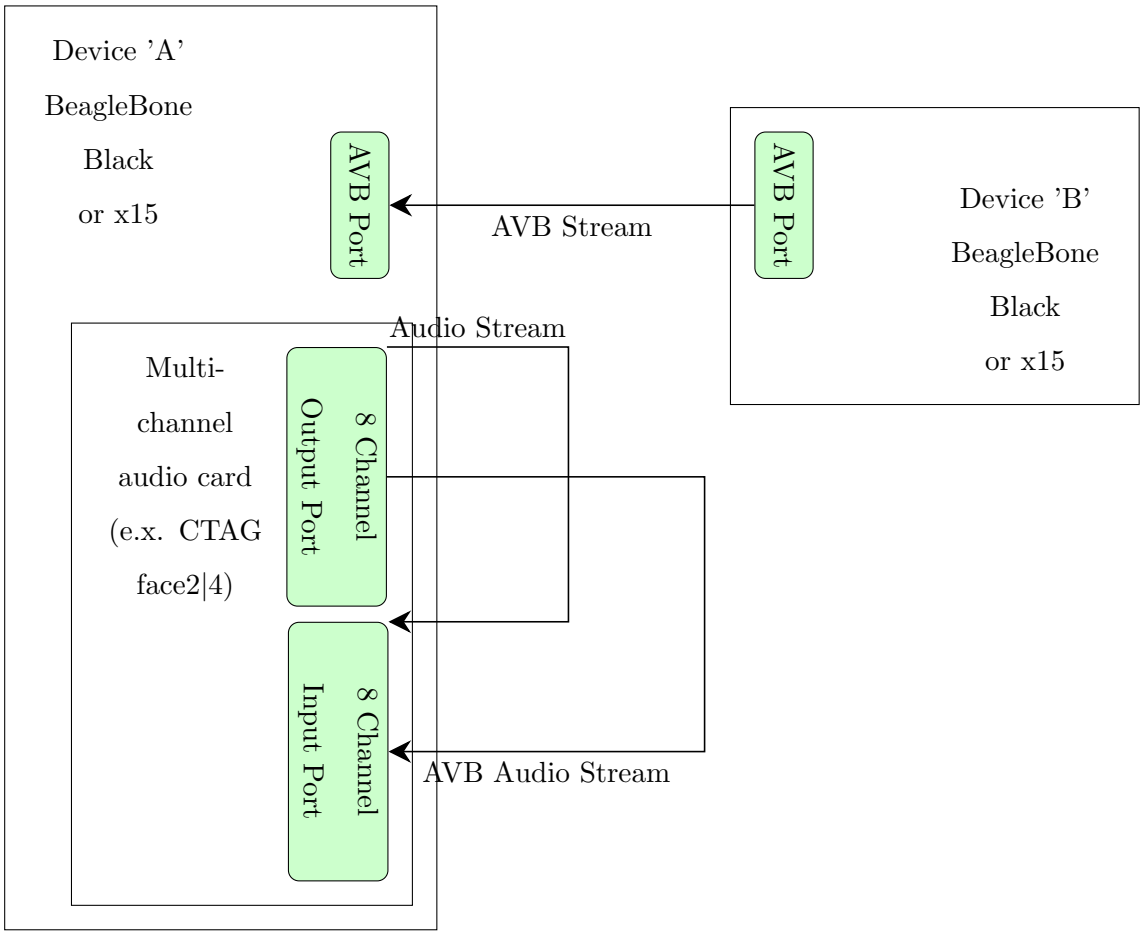

FIGURE 4.14: Demo Setup AB - Variation 2

4.4.2.2 Demo Mode 'XY'

This mode is used to demonstrate the synchronized multi-channel and high sample rate AVB streaming between 2 devices supporting AVB ALSA virtual device. In this setup both the devices have a multi-channel audio sound card. The device 'Y' streams the given wave file through AVB port with a presentation time set from a pre-determined start time. The device 'X receives the AVB stream and loops it back to the multi-channel audio card according to the received presentation time. The device 'Y' also streams the same wave file through it's mutli-channel audio card starting at the same pre-determined time. Both these multi-channel audio streams are fed into a test device where they are recorded and then time synchronization is tested.

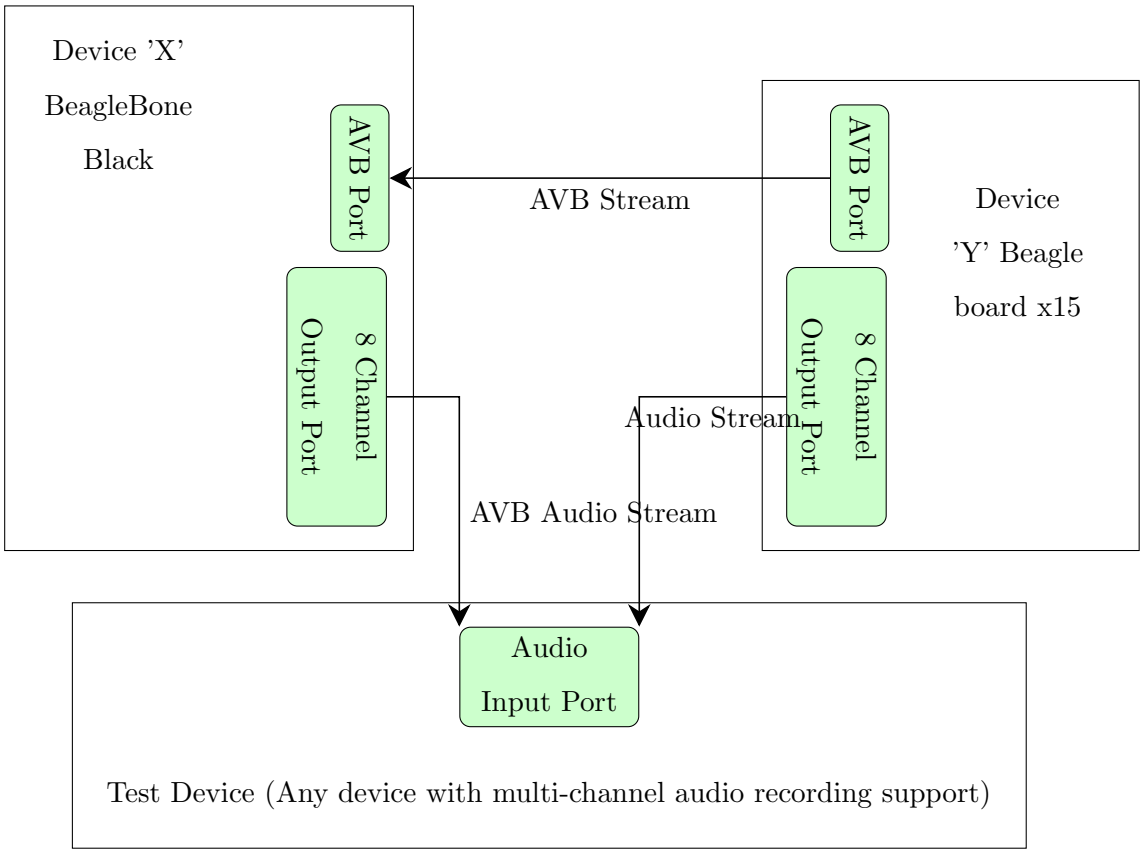

FIGURE 4.15: Demo Setup XY

4.4.3 Design and Implementation

The basic design of the test application is as follows, first the input arguments are parsed to find out about the operational mode and other parameters. In case the input parameters are incomplete or wring the usage instructions for the test applications are printed out and then the test application exits the operation. Based on the input parameters (as explained in section 4.4.1) parsed, a new playback or record instance is configured and then the configured instance execution is started in a new thread.

Based on the operation mode required there may also be a need for multiple playback and recording instances. In which case multiple threads are created and for each thread a playback or record thread is configured. In this case the test application continues operation until all the threads has stopped their execution of the required operation.

This basic design is illustrated as follows:

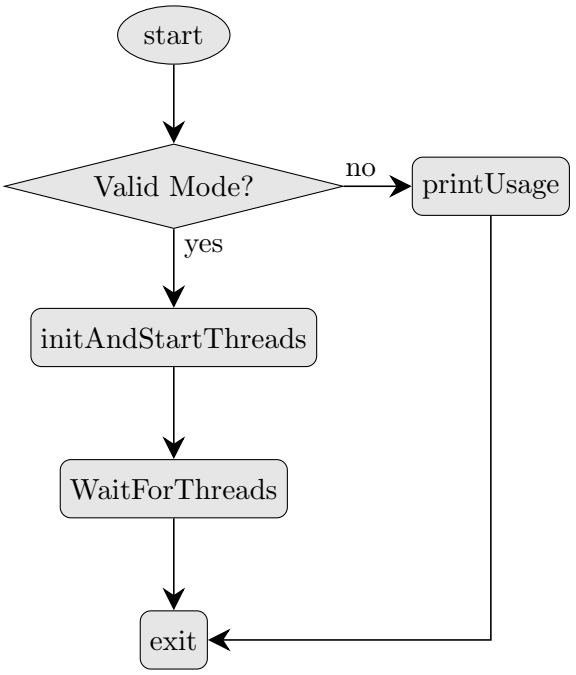

FIGURE 4.16: AVB Test Application

4.4.3.1 Playback

A playback instance operation is described below and illustrated in the figure 4.17

- First the given file is opened as a binary file. If the file cannot be opened for any reason, the playback cannot continue and the playback thread exits.
- When the file is opened the wave header is read, parsed and validated. If the wave header is incorrect the playback thread also exits.
- If the playback is through the AVB device the PCM and hardware dependent interfaces of the AVB AlSA virtual device are opened. Else for normal audio card playback only the PCM interface is opened. Then the PCM interface is configured with the audio parameters from the wave file for playback.
- After the PCM interface is configured, the playback loop is started and the audio samples are read from the wave file and sent to the PCM interface of the driver until all the audio samples in the wave file or until the required amount of audio samples are streamed.
- If the time synchronization is requested, the playback loop is not started until the pre-determined time is reached. For simplicity in the AVB test application this

pre-determined time is defined as the next second which is a full multiple of 3. The playback loop executes a busy-wait in a loop where the PTP hardware time is checked every 100 micro-seconds until we reach the second which is an interger multiple of 3.

- If time stamping option is set, the hardware clock is used to calculate the presentation time and it is set through the hardware dependent interface.
- It is also possible that the playback is not from a wave file, but from an audio buffer where another thread is recording and storing audio samples. For this live playback mode, a wave file is not opened but the audio parameters are configured from the command line and the audio samples are read directly from the audio buffer and streamed through the open PCM interface as described above.
- When the streaming is completed, the PCM interface is closed and the playback thread exits.

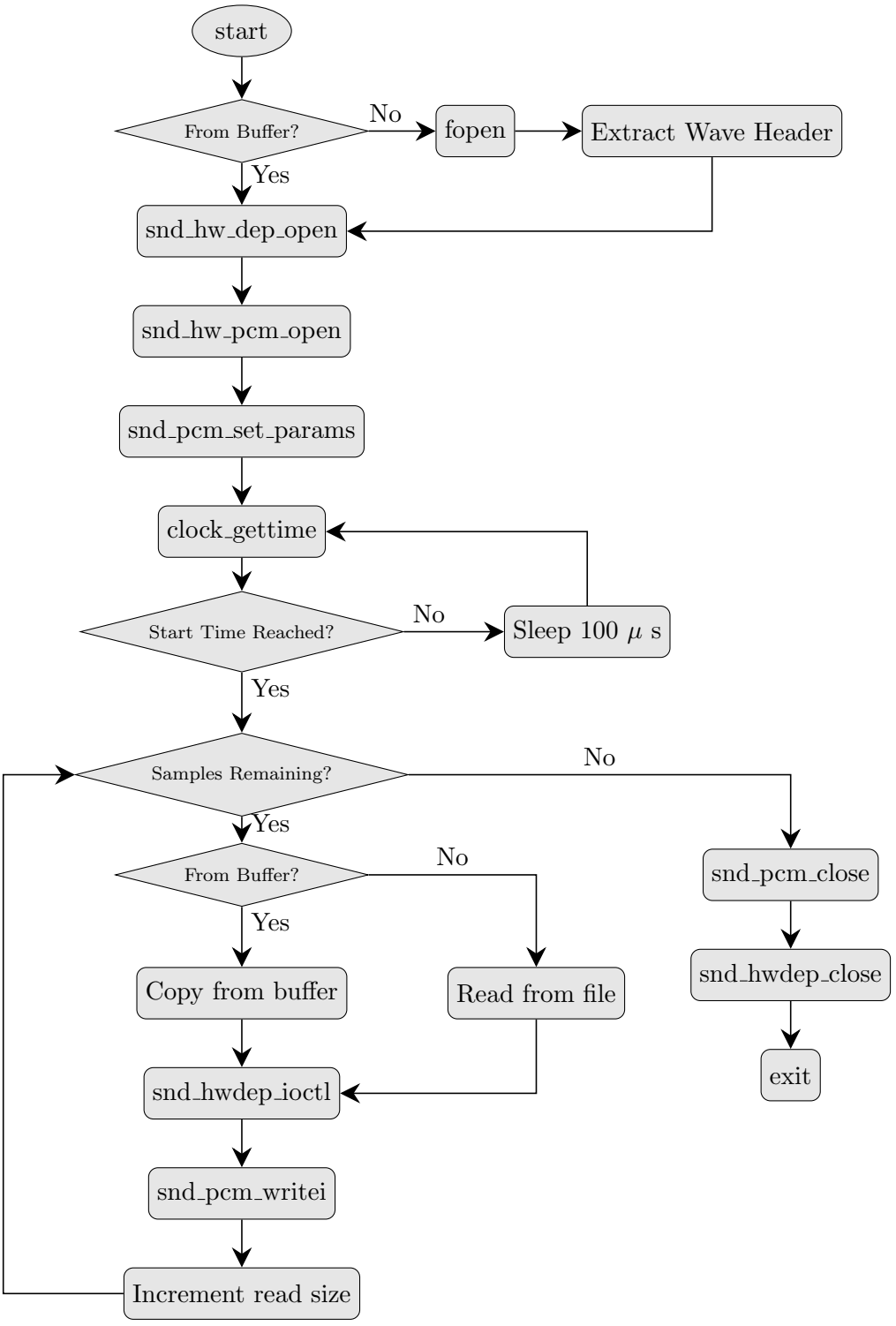

FIGURE 4.17: AVB Test Application - Playback Thread

4.4.3.2 Record

A record instance operation is described below and illustrated in the figure 4.18

- First a wave file with a temporary name is opened as a binary file. If the file cannot be opened for any reason, the record cannot continue and the record thread exits.
- If the record is from the AVB device the PCM and hardware dependent interfaces of the AVB AlSA virtual device are opened. Else for normal audio card playback only the PCM interface is opened. Then the PCM interface is configured with the audio parameters from the wave file for recording.
- After the PCM interface is configured, the record loop is started and the audio samples are read from the driver and then written to the output file until the required amount of audio samples are recorded.
- The time synchronization option is not applicable for the record instance as there is no playback involved.
- If time stamping option is set, the presentation time stamp is read from the hardware dependent interface and stored.
- It is also possible that the record does not store the audio samples to a wave file, but to an audio buffer from which another thread will playback the audio samples. For this live record mode, a wave file is not opened but the audio parameters are configured from the command line and the audio samples are read written to the audio bufferfrom the open PCM interface as described above.
- When the recording is completed, if the wave file audio samples are written to a file and not to a buffer, a new wave file is opened with the given name. First a new wave file header is cretaed using the audio parameters used for streaming and the information regarding the total number of reocrded samples. First this new wave header is written to the file and then the audio samples from the temporary file is copied to the output file.
- When the output file is written the PCM interface is closed and the record thread exits.

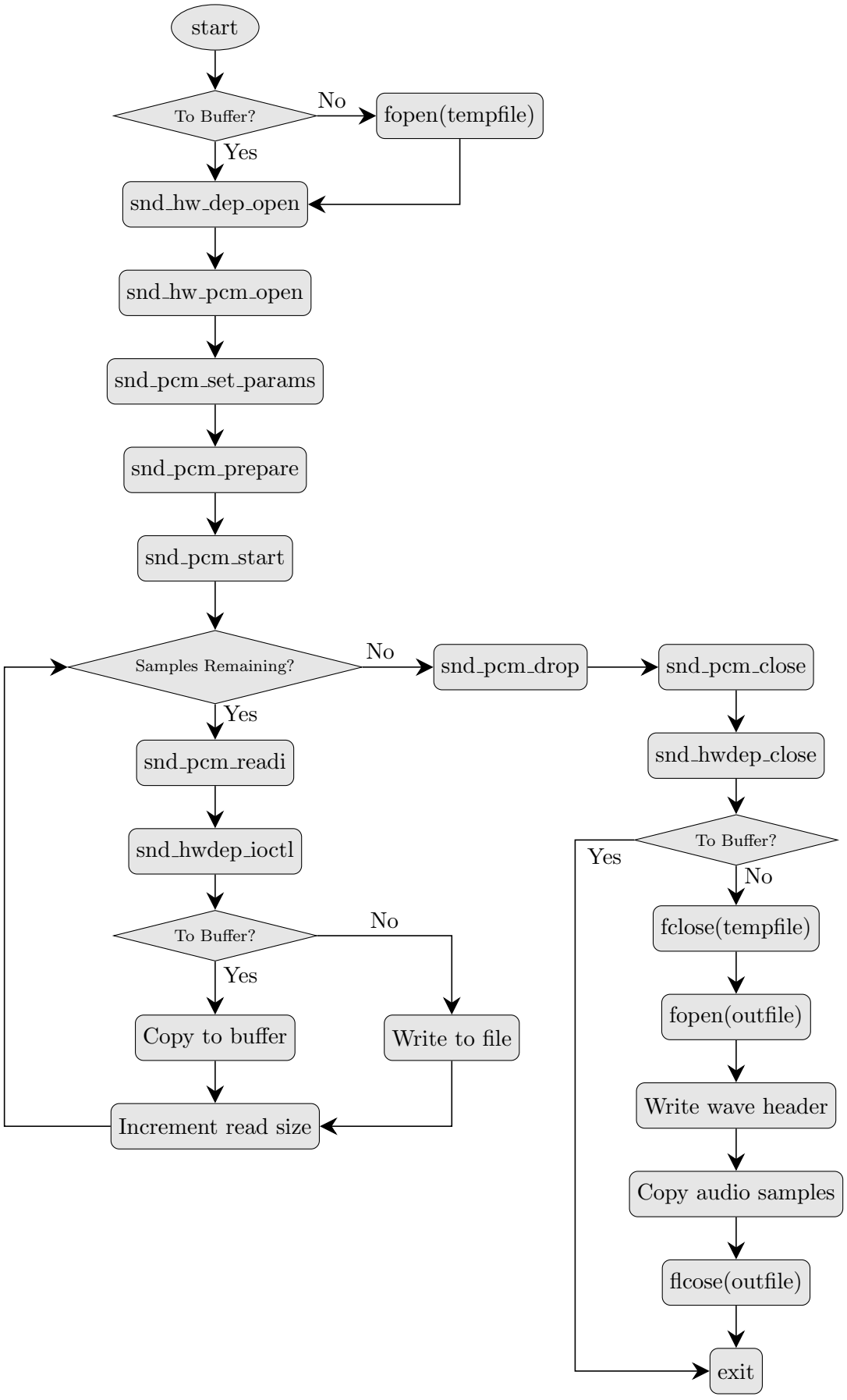

FIGURE 4.18: AVB Test Application - Record Thread

Chapter 5

Development

As the design and the implementation details of the various software modules used for the AVB stack implementation are detailed in the previous chapter, this chapter discusses the development details for these software modules including the various development processes, development tools and development strategies used.

5.1 Google Summer of Code

The main inspiration for the *"Synchronized real time audio streaming over Ethernet in embedded systems"* project came from the involvement in the Google Summer of Code (GSoC). This thesis is an extension of the GSoC project titled, *"BeagleBone AVB Stack"* [22]. This GSoC project was executed as one of the GSoC projects for the Beagle Board organization.

The Google Summer of Code is an international annual program for students, in which Google will provide stipends for successful completion of development of an open source software project task from one of the several participating software development organizations. The main aim of the program is to improve the open source software development projects and inspire students to contribute more to the open source development projects.

All participating organizations first have to be selected by Google and after selection they are assigned a maximum number of projects that they can offer to students. The

organizations then come-up with a list of projects that it wishes to be taken up by students. Students have to create a proposal for any of those list of projects wished by the organization or also they can create a proposal of their own. The proposals are evaluated by the respective organization and the final list of accepted students are published. When a student is accepted by an organization for a project, the student has to complete the project in hand during the summer within the agreed upon milestone. The organization assigns mentors for the students to help the development and to evaluate the results of the project.

The implementation details and the various features of the GSoC Project "Beaglebone AVB Stack" are listed below [23]:

- Implementation of the set of protocols required for a AVB audio streaming between 2 devices.
- Beagle bone black devices are chosen as the development environment.
- Only Stereo channel and low audio bit-rate streams are supported.
- Only the basic functionality of the AVB streaming is evaluated (i.e. no detailed analysis of the stream parameters or no testing of compatibility to other devices)

After the successful completion of the GSoC Project, the same is expanded with new requirements for this Thesis. The new requirements on top the work done for the GSoC are as follows:

- Improvements to also support the Beagle board x15 devices.
- Improvements to also support multi-channel and high bit-rate streams.
- Improvements to add support for inter-operability to any other standard AVB device (Mackbook pro laptops).
- Detailed analysis of the streaming including synchronization accuracy, latency and other parameters and conclude regarding the merits and demerits of the AVB protocol and the implementation of AVB stack in the beagle bone platform.

5.2 Development environment

All the software components are developed on a Linux host computer running Ubuntu operating system. The linaro cross compiler is used to cross compile the source code in the Linux host for the beagle bone devices.

The gPTP daemon is developed platform independent and hence it is possible to also run the gPTP daemon also in any Linux based system. But when executed in normal x86 systems the Linux time-stamping employed is software time-stamping which is less accurate and in these systems, for synchronization normal clocks are used instead of hardware clocks. Irrespective of these limitations executing gPTP daemon in a x86 environment on a Linux host is useful as it supports better debugging tools and hence the development time is shortened. The gPTP daemon source code is version controlled under git and hosted at github here [1]

As the AVB ALSA driver is developed as part of the Linux kernel, a new development branch [2] is created from the Linux kernel repository for beagle bone black version 4.4 long term support branch [3]. The AVB ALSA driver is developed as any other loadable kernel module and tested only in the beagle bone black device. The kernel is also cross compiled in a Linux host. When the kernel is built the resulting loadable kernel module file is transferred to the beagle bone black and loaded into the kernel.

The test application is a normal Linux executable which is also developed in the Linux host machine and then cross compiled to the beagle bone black. It is version controlled similarly in a git reprository and hosted at github here [4]

5.3 Debugging

Debugging of the software is mainly done by analyzing the logs created during execution. Based on the software component various log destinations are used.

- For gPTP daemon the logs are routed to the syslog interface, which can be viewed using the file *"/var/log/syslog"* in the Linux file system.

[1] "https://github.com/induarun9086/gPTPd"
[2] "https://github.com/induarun9086/beagleboard-linux"
[3] "https://github.com/beagleboard/linux"
[4] "https://github.com/induarun9086/avbtest"

- For the AVB ALSA driver, the logs are printed using the kernel version of printf (i.e. printk) to the kernel log buffer. This log buffer contents are also transferred to the syslog. Also the kernel log messages can also be viewed by the dmesg command.

- The test applications logs are directly printed in the console using normal printf.

Chapter 6

Evaluations

This section details the various setups, experiments and their results, leading to the evaluation of the various operational parameters of the AVB software stack.

6.1 Delay Variation

The delay variation is defined as the variance the successive peer to peer delay measurement values 3.3 in the gPTP protocol. Although the delay measurement process follows the same procedure every time, random variations in the time stamping (since the normal crystal clocks used are not perfect) leads to sightly different delay values for every measurement. In ideal case the variance should be zero and the delay should be the same for every measurement, but in normal working conditions this will not be the case. The aim here is that the delay variance should be as small as possible. Since the clock synchronization between the two devices depends on the measured delay, if the delay variation is small the clock synchronization accuracy will be high.

Any setup described in the section 4.4.2 can be used for the delay variation measurement. The general setup is as follows, two devices that support hardware time stamping is connected with a cross Ethernet cable. In this case a beagle bone black and beagle board x15 are connected via a cross Ethernet cable. The gPTP daemon is started in both these devices using the following command

```
> ./gPTPd -n -l6
```

To view the measured delay values the gPTP daemon application is started as a normal application using the *-n* switch (i.e. normal mode as opposed to daemon mode). Also the log level is set to log level 6 using the *-l6* switch to view the measured delay values in the console. The raw log of a such execution is given in appendix A.1, from this log 50 successive measured delay values are extracted and plotted below in the figure 6.1

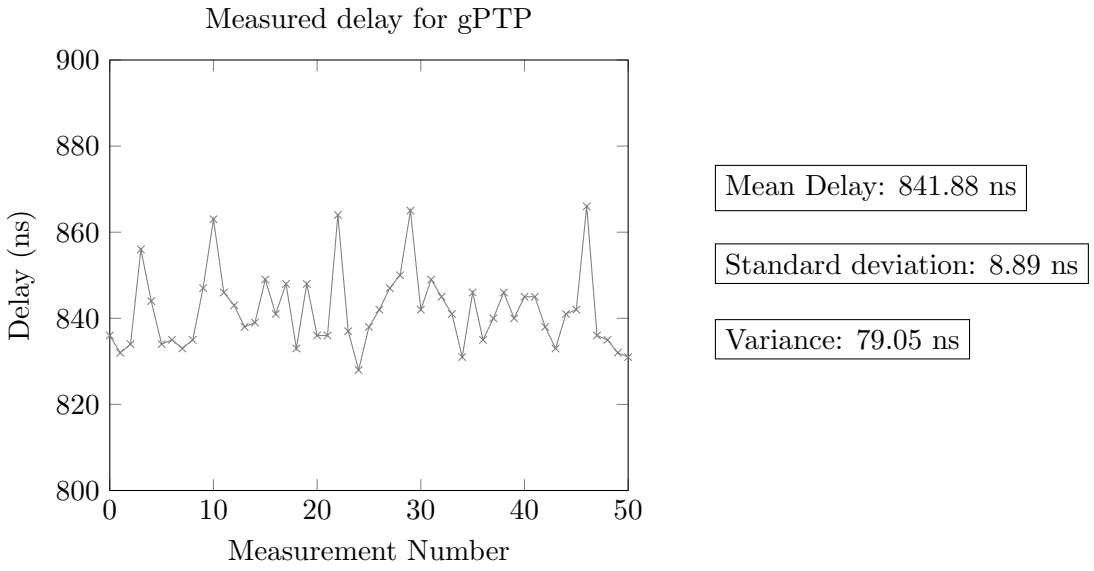

FIGURE 6.1: gPTP Delay Variance

From the above graph the average value for the measured delay value is calculated as *841.88 ns* and the average standard deviation is calculated as *8.89 ns*. Also from the normal distribution 6.2 of the measured delay values we can see that it approximates a bell curve (i.e. normal distribution) indicating that the measured delay values follow a natural normal distribution with no other external influences.

From the above discussions it can be decided that the measured delay values does not influence the clock synchronization too much as the variance is only in the order of 10 ns which is negligible for most audio applications. Also this delay variance is much smaller than other parameters which influence the synchronization.

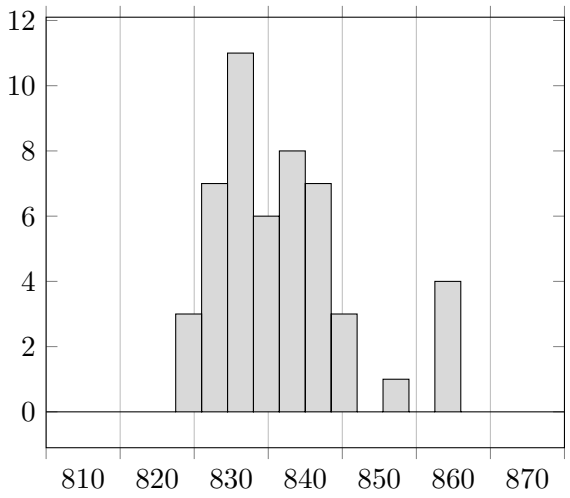

FIGURE 6.2: gPTP Delay Histogram

6.2 Clock Drift

Clock drift is defined as the amount a clock drifts away from another clock, from a defined point in time when the both clocks have the same time value. Since the frequency of the crystal oscillators used in the embedded systems as clock sources are highly unstable, it increases the clock drift. Crystal oscillators are highly influenced by environmental factors such as heat, stress, moisture and inherent factors such as age. Since clock drift affects the clock synchronization it has to be kept low for synchronized audio streaming.

To measure the clock drift any setup described in the section 4.4.2 can be used as same as for the delay variation measurement. The general setup is as follows, two devices that support hardware time stamping is connected with a cross Ethernet cable. In this case a beagle bone black and beagle board x15 are connected via a cross Ethernet cable. The gPTP daemon is started in both these devices using the following command

```
> ./gPTPd -n -l5
```

The command is similar to that of the command required for the delay measurement experiment, but the log level is set a little higher at level 5 with the *-l5* switch to view some more extended logs required to calculate the clock drift. The raw log of clock drift experiment is given in appendix A.2, from this successive measured clock drift values

are extracted and plotted below in the figure 6.3

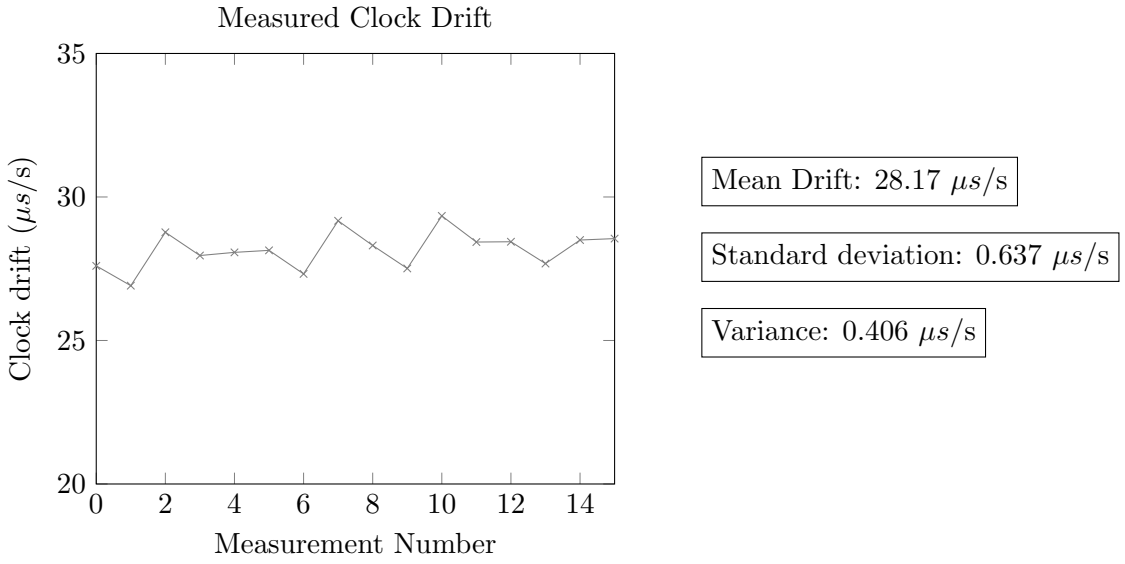

FIGURE 6.3: Hardware Clock Drift

A sample clock drift calculation is explained below with the pair of logs below. These logs are generated when the gPTP slave device receives the sync follow-up command the clock synchronization is calculated. The values are explained as follows.

- *SyncTxTime* - Tx time stamp when the Sync gPTP command was transmitted by the gPTP initiator.
- *SyncRxTime* - Rx time stamp when the Sync gPTP command was received in the gPTP responder.
- *lDelayTime* - Measured gPTP delay.
- *CurrSynOff* - Offset for local clock to the initiator clock.
- *prSyncTime* - Local time before synchronization.
- *poSyncTime* - Local time after synchronization.

```
[000000484481] gPTPd: @@@ SyncTxTime: 1503126212_411757622
[000000484481] gPTPd: @@@ SyncRxTime: 1503126212_410850350
[000000484481] gPTPd: @@@ lDelayTime: 0_000000823
```

```
[000000484481] gPTPd: @@@ CurrSynOff: 0_000908095 (-1)
[000000484481] gPTPd: @@@ prSyncTime: 1503126213_411821643
[000000484481] gPTPd: @@@ poSyncTime: 1503126213_412733008

[000000515481] gPTPd: @@@ SyncTxTime: 1503126244_413223246
[000000515481] gPTPd: @@@ SyncRxTime: 1503126244_412314482
[000000515481] gPTPd: @@@ lDelayTime: 0_000000823
[000000515481] gPTPd: @@@ CurrSynOff: 0_000909587 (-1)
[000000515481] gPTPd: @@@ prSyncTime: 1503126244_412608332
[000000515481] gPTPd: @@@ poSyncTime: 1503126244_413521159
```

The elapsed time between the two synchronizations is calculated as the difference from the two SyncTxTime values. During this elapsed time the difference in the clock progress between the two devices is given as the value CurrSynOff in the second log. So the drift rate is calculated as the ratio between the CurrSynOff and the elapsed time.

From the figure 6.3 the mean clock drift rate is calculated as $28.17\,\mu s/s$ and the standard deviation is $0.637\,\mu s/s$. From these values we can see that although the drift rate is little higher the variance is low i.e. the clock drift is always constant which can be corrected.

As in real embedded system crystal oscillator based clocks the clock drift cannot be avoided, to has to be mitigated by some other means. This is covered by the gPTP protocol synchronization process. By periodically synchronizing the slave devices to the grand master device, the clock drift is reset periodically and the synchronization is re-established. So the maximum drift the slave clock encounters is limited to the amount of drift that is accumulated during the time before the next synchronization.

Based on the current default sync repetition time of 32 seconds, the maximum clock drift for the slave device can be up-to a maximum value of 900 μs. Choosing a smaller sync repetition time will limit this clock drift value to a lower value. Although this value can affect the media synchronization it is limited to the sub milli-second range which is still low enough to be perceived in real world scenario.

6.3 Synchronization Accuracy

The synchronization accuracy is defined as the time delta between the streams output by two devices. To test this the Demo XY setup 4.15 is used. Here both the beagle bone black and the beagle board x15 playbacks the same stream synchronously. These streams are recorded through an external recording device and analyzed for synchronization.

First the demo setup explained in 4.15 is setup and the AVB software stack is started using the following series of commands,

```
> sudo ./gPTPd
> sudo insmod /lib/modules/4.9.50+/kernel/sound/core/snd-hwdep.ko
> sudo insmod snd-avb.ko
```

Where with the first command the gPTP daemon is started with it's default arguments and operation mode. Then the hardware dependent ALSA library is loaded which is required for the hardware dependent interface in the AVB ALSA driver and finally the AVB ALSA driver itself is loaded. After all these are loaded the gPTP processes such as delay measurement and clock synchronization is started and in the AVB ALSA driver the device detection, enumeration and control processes are started.

When the AVB stack is initialized and started the avb test application can be started for testing the media synchronization. For this first in the beagle board x15 device the following command is executed.

```
> sudo ./avbtest -y -c2 -s48000 -l2 rec.wav
```

And in the beagle bone black device the following command is executed

```
> sudo ./avbtest -x -l2 test.wav
```

In parallel the analog audio playback from these devices are recorded in an external device. After the streaming is completed, the recorded file is opened in an audio editing software such as Audacity to check for the synchronization. The result of such as experiment is given in the figure 6.5 with an explanation below it.

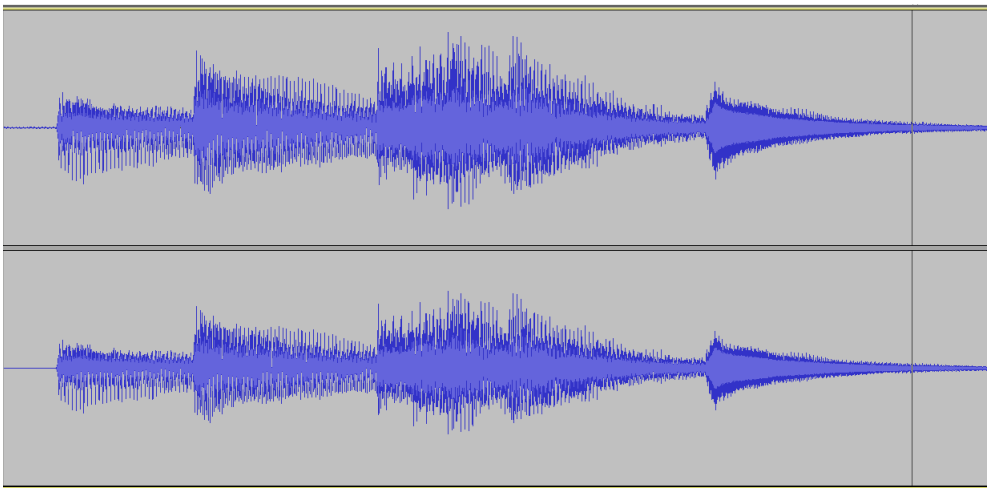

FIGURE 6.4: Full Record of Synchronization test

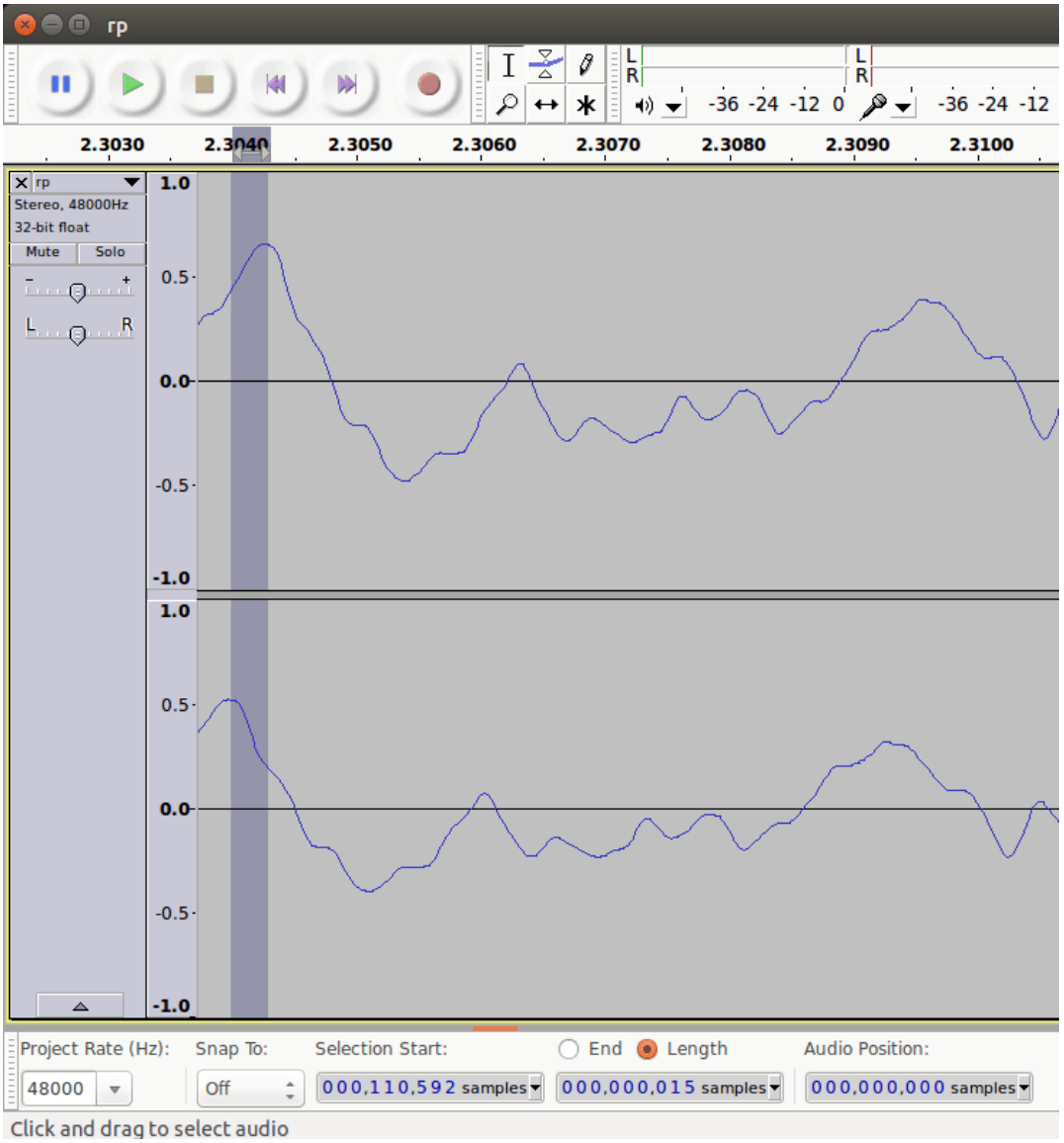

FIGURE 6.5: Audio Synchronization

In the stream in the figure above two channels are recorded each one from a different device. It is done such a way for easy comparison of the stream from both devices. This is evident from the fact that the two wave forms are not of the exact level as they are played back from different device with different audio cards which may be set different volume levels the levels of these streams is not equal. But still we can see the general waveform of these streams which can be used for comparison. For the actual comparison select a crest or a trough in one channel and then select until the same crest or a trough in the other channel and then count the number of samples in the selected portion of the wave form. Then the number of samples and the sampling rate is used to calculate the actual synchronization accuracy. So from the above figure there are 15 samples in the selected portion of the wave form. The recording is at a sampling rate of 48000 Hz. So according to the following calculation the synchronization accuracy is 1.437 ms between these two streams.

`Synchronization Accuracy = (15 / 48000) = 312.5` μ`s`

As it is generally accepted that any audio delay less than 10 ms is not perceivable to the human ear [24] [25], the synchronization accuracy of less than 1 ms is acceptable for most streaming applications.

6.4 Latency

The latency is a measure of the time it takes for the samples to reach the destination from the time they are transmitted. For a real time audio streaming system the latency should be as low as possible.

To measure the latency of the AVB ALSA driver based streaming the following experiment is carried out. For this purpose a AVB stream is played from one device and recorded in the second device. The time the streaming started in the first device is noted down and also the time when the first frames received in the second device is noted down. The difference between these two time stamps is the latency. The two time stamps can be directly compared and subtracted since the hardware clocks of these two devices are synchronized and any drift between them is negligible (refer 6.3).

First the demo setup explained in 4.15 is setup and the AVB software stack is started as explained in the previous section regarding the synchronization accuracy measurement 6.3. When the AVB stack is initialized and started the avb test application can be started for testing the media synchronization. For this first in the beagle board x15 device the following command is executed.

```
> sudo ./avbtest -r -l2 rp.wav
```

And in the beagle bone black device the following command is executed

```
> sudo ./avbtest -p -l2 test.wav
```

```
debian@BeagleBoard-X15:~/avbtest$ sudo ./avbtest -p -l2 piano2.wav
AVB Test Application Oct 10 2017 21:58:03
avbtest: Playing back -1 frames from file piano2.wav (ch: 2 sr: 48000)
t0: Playback params ringbufsize: 1536, periodsize: 384
t0: Playback starting @ 1503125410 s 576817781 ns
t0: First frame transferred @ 1503125410 s 576944541 ns
avbtest: Thread t0 exit with result: 0
avbtest: Operation completed

debian@beaglebone:~/avbtest$ sudo ./avbtest -r -l2 rp.wav
avbtest: Recording -1 frames to file rp.wav (ch: 2 sr: 48000)
t0: Record params ringbufsize: 1536, periodsize: 384
ALSA lib pcm.c:7843:(snd_pcm_recover) overrun occurred
t0: First frame received @ 1503125410 s 595405977 ns
t0: Short read (expected 384, read 48)
avbtest: Thread t0 exit with result: 0
avbtest: Operation completed
```

With the above commands the AVB streams is played back from the beagle bone black device. In the test application when the first frame is transmitted the time stamp is noted and printed to the console. When the first frame is received in the beagle board x15, again the time stamp is noted and printed out by the test application receiving the

stream. The logs of such an execution is given in above logs. The following two lines from the image is of our interest, from these values the latency is calculated as below.

```
t0: First frame transferred @ 1503125410 s 576944541 ns
t0: First frame received @ 1503125410 s 595405977 ns
Latency = (1503125410 s 576944541 ns) - (1503125410 s 595405977 ns)
Latency = 0 s 18,461,436 ns = 18.46 ms
```

Although the measured latency of 18.46 ms is not sufficient for real time audio streaming in live low latency applications, it is just almost sufficient for most of the real time audio streaming for media applications. But if more lower latencies are required the size of the hardware buffer has to be decreased. But decreasing the size of the hardware buffer can result in frequent under-flows in the streaming which can lead to audio breaks. So a fine balance is required to trade-off between low latency and reliable and continuous audio streaming.

6.5 MAC AVB Detection

The AVB ALSA driver running in the Beagle Bone can also be detected as an AVB sound device in MacBook pro computers. To test the detection, the Beagle Bone Black has to be directly connected to the MacBook pro using a ethernet cross cable. In MacBook Pro models where no ethernet port available a thunderbolt ethernet adapter can be used. But a USB ethernet adapter should not be used. The AVB driver can be started using the following commands:

```
> sudo ./gPTPd
> sudo insmod /lib/modules/4.9.50+/kernel/sound/core/snd-hwdep.ko
> sudo insmod snd-avb.ko
```

It takes some seconds for the AVDECC enumeration to take place after which the AVB device can be seen under the "Audio MIDI Setup - Window - Show Network Device Browser". The screen shot of this detection is given in the following image.

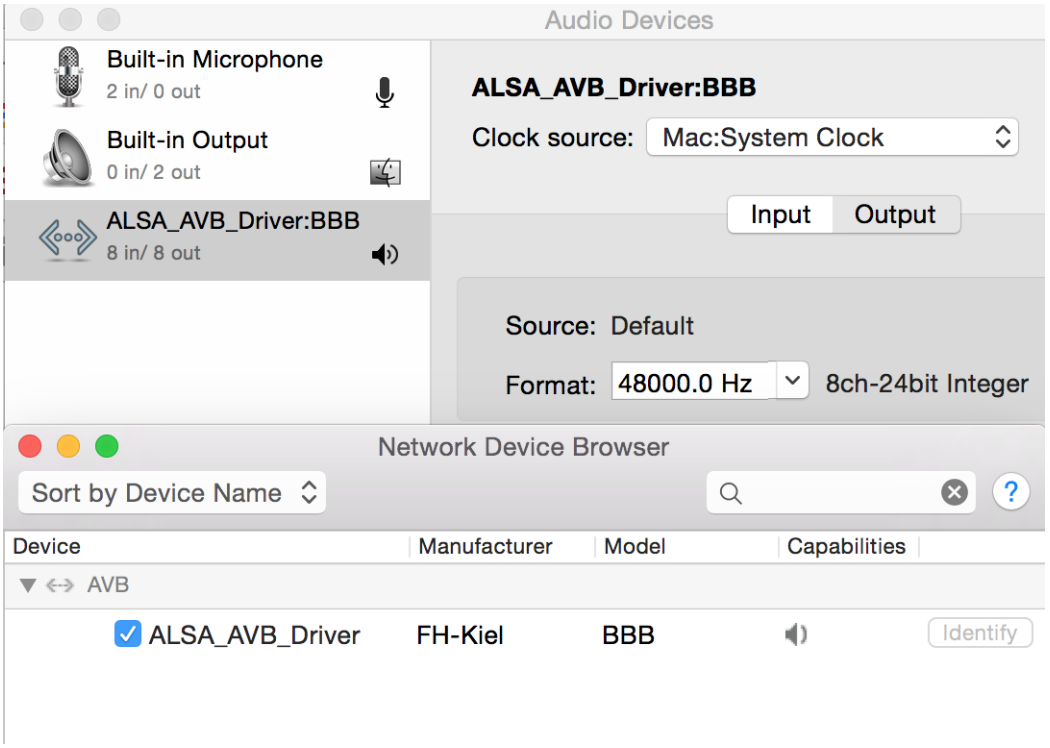

FIGURE 6.6: MAC AVB Detection

Also in the MacBook pro the operational parameters of the AVB connection can be seen using the *"avbdiagnose"* command, which produces a log as given appendix A.3.

Chapter 7

Limitations

Apart from the various features of the AVB software stack discussed in the previous sections for the synchronized real time audio streaming, there are also some limitations. These limitations are listed and described briefly below:

- In gPTP daemon, only one port per device is supported, for example although beagle board x15 contains two Ethernet ports the gPTP can communicate through only one Ethernet port at a time. This is different from the gPTP specification where multiple ports per device can be supported.
- Seamless changeover from between grand masters is not supported. Hence when a grand master changes there will be an abrupt jump in the synchronized time.
- Only Ethernet physical medium (IEEE 802.3) is supported although the specification provides support for IEEE 802.11 (Wi-fi) and IEEE 802.3 Passive optical links.
- The Forwarding and Queuing for Time-Sensitive Streams (FQTSS) improvement for the networking queues are not implemented.
- Only the AVDECC responder role is implemented, but does not support the AVDECC controller role.
- Regarding streaming a maximum of 8 channels and maximum sampling rate of up-to 192kHz is supported.
- Only one playback and capture stream is possible in parallel.

Chapter 8

Conclusion

The previous sections elaborate on the objectives laid out in the introduction of this thesis 1.6.

In the Literature review section 2 the various current implementations of the AVB stack are listed and described in detail. It was concluded there that although several implementations for the AVB stack are available, either they are not complete turn key solutions or in case of complete solutions they are not available in public domain as open source software. So there is a need for a complete open source AVB stack solution.

The technical review section 3 describes the various technical details of the AVB standards, Linux operating system and the beagle board hardware. These details are required to be clearly understood before a AVB solution is designed and implemented.

In the next sections of implementation 4 and development 5 the implementation details of the various modules involved in the AVB software stack are described in detail explaining the various features of the software modules and their detailed design. The development section describes the various development strategies, tools and environments used in the development of the software modules for the AVB software stack.

And finally in the evaluation section 6, several experiments are carried out in order to calculate the various operational parameters regarding the synchronization and latency of the audio streaming. The experiments and their results are described in detail and then the results are interpreted and discussed in detail regarding their impact in the

synchronized, real time streams. Here the various results pointed out that the AVB software stack implemented is indeed sufficient for a synchronous, real time audio streaming. Finally the limitations of the AVB software solution are listed in 7.

To summarize, AVB was proposed as a solution for a synchronized, real time audio streaming in embedded systems. The proposed solution is researched, it's technical details are studied and a software solution is designed and implemented. Beagle bone devices, BeagleBone black and BeagleBoard x15 are chosen as a hardware platform for evaluating the solution. The implemented AVB solution is evaluated on these beagle board devices. By analyzing the results of the evaluations it is concluded that synchronized, real time audio streaming is possible in embedded systems using the proposed AVB software solution and the beagle board devices are sufficient in providing an environment for a stable and synchronized hardware clock through gPTP and sufficient bandwidth through the integrated Ethernet ports. And because of their small form factor, power and support for a range of Capes that extend their functionality, the combination of the beagle board devices plus the proposed AVB solution can provide a platform for synchronized real time audio streaming applications.

Some future enhancements for the proposed solution includes removing the limitations found, expand for other platforms, make it more configurable and improving the overall performance figures by even more strict implementation of the protocols and through more testing and evaluations.

Appendix A

Evaluation Logs

A.1 Delay variance measurement logs

```
[000000000000] gPTPd: --------------------------------------
[000000000001] gPTPd: gPTP Init bbb Jan  3 2018 20:38:01
[000000000001] gPTPd: --------------------------------------

[000000000029] gPTPd: ---> Assuming grandmaster role
[000000010030] gPTPd: ---> gPTP msrdDelay: 836
[000000019030] gPTPd: ---> gPTP msrdDelay: 832
[000000028030] gPTPd: ---> gPTP msrdDelay: 834
[000000037077] gPTPd: ---> gPTP msrdDelay: 856
[000000046078] gPTPd: ---> gPTP msrdDelay: 844
[000000055078] gPTPd: ---> gPTP msrdDelay: 834
[000000064078] gPTPd: ---> gPTP msrdDelay: 835
[000000073086] gPTPd: ---> gPTP msrdDelay: 833
[000000082086] gPTPd: ---> gPTP msrdDelay: 835
[000000091086] gPTPd: ---> gPTP msrdDelay: 847
[000000100119] gPTPd: ---> gPTP msrdDelay: 863
[000000109118] gPTPd: ---> gPTP msrdDelay: 846
[000000118118] gPTPd: ---> gPTP msrdDelay: 843
[000000127118] gPTPd: ---> gPTP msrdDelay: 838
[000000136118] gPTPd: ---> gPTP msrdDelay: 839
[000000145118] gPTPd: ---> gPTP msrdDelay: 849
[000000154118] gPTPd: ---> gPTP msrdDelay: 841
[000000163118] gPTPd: ---> gPTP msrdDelay: 848
[000000172118] gPTPd: ---> gPTP msrdDelay: 833
[000000181118] gPTPd: ---> gPTP msrdDelay: 848
[000000190118] gPTPd: ---> gPTP msrdDelay: 836
[000000199162] gPTPd: ---> gPTP msrdDelay: 864
```

```
[000000208162] gPTPd: ---> gPTP msrdDelay: 837
[000000217162] gPTPd: ---> gPTP msrdDelay: 828
[000000226162] gPTPd: ---> gPTP msrdDelay: 838
[000000235170] gPTPd: ---> gPTP msrdDelay: 842
[000000244170] gPTPd: ---> gPTP msrdDelay: 847
[000000253170] gPTPd: ---> gPTP msrdDelay: 850
[000000262203] gPTPd: ---> gPTP msrdDelay: 865
[000000271202] gPTPd: ---> gPTP msrdDelay: 842
[000000280202] gPTPd: ---> gPTP msrdDelay: 849
[000000289202] gPTPd: ---> gPTP msrdDelay: 845
[000000298202] gPTPd: ---> gPTP msrdDelay: 841
[000000307202] gPTPd: ---> gPTP msrdDelay: 831
[000000316202] gPTPd: ---> gPTP msrdDelay: 846
[000000325202] gPTPd: ---> gPTP msrdDelay: 835
[000000334202] gPTPd: ---> gPTP msrdDelay: 840
[000000343202] gPTPd: ---> gPTP msrdDelay: 846
[000000352202] gPTPd: ---> gPTP msrdDelay: 840
[000000370246] gPTPd: ---> gPTP msrdDelay: 845
[000000379246] gPTPd: ---> gPTP msrdDelay: 845
[000000388246] gPTPd: ---> gPTP msrdDelay: 838
[000000397258] gPTPd: ---> gPTP msrdDelay: 833
[000000406258] gPTPd: ---> gPTP msrdDelay: 841
[000000415258] gPTPd: ---> gPTP msrdDelay: 842
[000000424288] gPTPd: ---> gPTP msrdDelay: 866
[000000433286] gPTPd: ---> gPTP msrdDelay: 836
[000000442286] gPTPd: ---> gPTP msrdDelay: 835
[000000451286] gPTPd: ---> gPTP msrdDelay: 832
[000000460286] gPTPd: ---> gPTP msrdDelay: 831
```

A.2 Clock drift measurement logs

```
[000000094291] gPTPd: @@@ SyncTxTime: 1503125823_211258798
[000000094291] gPTPd: @@@ SyncRxTime: 1503125823_210380945
[000000094291] gPTPd: @@@ lDelayTime: 0_000000828
[000000094291] gPTPd: @@@ CurrSynOff: 0_000878681 (-1)
[000000094291] gPTPd: @@@ prSyncTime: 1503125823_210724283
[000000094291] gPTPd: @@@ poSyncTime: 1503125823_211606219

[000000127322] gPTPd: @@@ SyncTxTime: 1503125856_243282526
[000000127322] gPTPd: @@@ SyncRxTime: 1503125856_242371615
[000000127322] gPTPd: @@@ lDelayTime: 0_000000828
[000000127322] gPTPd: @@@ CurrSynOff: 0_000911739 (-1)
[000000127322] gPTPd: @@@ prSyncTime: 1503125856_242685968
[000000127322] gPTPd: @@@ poSyncTime: 1503125856_243600955
```

```
[000000160322] gPTPd: @@@ SyncTxTime: 1503125888_243720926
[000000160322] gPTPd: @@@ SyncRxTime: 1503125888_242833794
[000000160322] gPTPd: @@@ lDelayTime: 0_000000828
[000000160322] gPTPd: @@@ CurrSynOff: 0_000887960 (-1)
[000000160322] gPTPd: @@@ prSyncTime: 1503125889_243800335
[000000160322] gPTPd: @@@ poSyncTime: 1503125889_244691539

[000000191322] gPTPd: @@@ SyncTxTime: 1503125920_245272246
[000000191322] gPTPd: @@@ SyncRxTime: 1503125920_244381174
[000000191322] gPTPd: @@@ lDelayTime: 0_000000828
[000000191322] gPTPd: @@@ CurrSynOff: 0_000891900 (-1)
[000000191322] gPTPd: @@@ prSyncTime: 1503125920_244687753
[000000191322] gPTPd: @@@ poSyncTime: 1503125920_245582743

[000000224363] gPTPd: @@@ SyncTxTime: 1503125953_287256314
[000000224363] gPTPd: @@@ SyncRxTime: 1503125953_286333308
[000000224363] gPTPd: @@@ lDelayTime: 0_000000828
[000000224363] gPTPd: @@@ CurrSynOff: 0_000923834 (-1)
[000000224363] gPTPd: @@@ prSyncTime: 1503125953_286640473
[000000224363] gPTPd: @@@ poSyncTime: 1503125953_287567562

[000000256370] gPTPd: @@@ SyncTxTime: 1503125985_295276674
[000000256371] gPTPd: @@@ SyncRxTime: 1503125985_294379142
[000000256371] gPTPd: @@@ lDelayTime: 0_000000830
[000000256371] gPTPd: @@@ CurrSynOff: 0_000898362 (-1)
[000000256371] gPTPd: @@@ prSyncTime: 1503125985_294758845
[000000256371] gPTPd: @@@ poSyncTime: 1503125985_295660567

[000000289402] gPTPd: @@@ SyncTxTime: 1503126018_327288266
[000000289402] gPTPd: @@@ SyncRxTime: 1503126018_326359459
[000000289402] gPTPd: @@@ lDelayTime: 0_000000830
[000000289402] gPTPd: @@@ CurrSynOff: 0_000929637 (-1)
[000000289402] gPTPd: @@@ prSyncTime: 1503126018_326667245
[000000289402] gPTPd: @@@ poSyncTime: 1503126018_327599968

[000000322402] gPTPd: @@@ SyncTxTime: 1503126050_327723734
[000000322402] gPTPd: @@@ SyncRxTime: 1503126050_326822826
[000000322402] gPTPd: @@@ lDelayTime: 0_000000830
[000000322402] gPTPd: @@@ CurrSynOff: 0_000901738 (-1)
[000000322402] gPTPd: @@@ prSyncTime: 1503126051_327771070
[000000322402] gPTPd: @@@ poSyncTime: 1503126051_328676014

[000000386443] gPTPd: @@@ SyncTxTime: 1503126115_371288442
[000000386443] gPTPd: @@@ SyncRxTime: 1503126115_370355256
[000000386443] gPTPd: @@@ lDelayTime: 0_000000830
[000000386443] gPTPd: @@@ CurrSynOff: 0_000934016 (-1)
[000000386443] gPTPd: @@@ prSyncTime: 1503126115_370665034
[000000386443] gPTPd: @@@ poSyncTime: 1503126115_371602192
```

```
[000000418454] gPTPd: @@@ SyncTxTime: 1503126147_383288510
[000000418454] gPTPd: @@@ SyncRxTime: 1503126147_382383076
[000000418454] gPTPd: @@@ lDelayTime: 0_000000823
[000000418454] gPTPd: @@@ CurrSynOff: 0_000906257 (-1)
[000000418454] gPTPd: @@@ prSyncTime: 1503126147_382688614
[000000418454] gPTPd: @@@ poSyncTime: 1503126147_383597976

[000000484481] gPTPd: @@@ SyncTxTime: 1503126212_411757622
[000000484481] gPTPd: @@@ SyncRxTime: 1503126212_410850350
[000000484481] gPTPd: @@@ lDelayTime: 0_000000823
[000000484481] gPTPd: @@@ CurrSynOff: 0_000908095 (-1)
[000000484481] gPTPd: @@@ prSyncTime: 1503126213_411821643
[000000484481] gPTPd: @@@ poSyncTime: 1503126213_412733008

[000000515481] gPTPd: @@@ SyncTxTime: 1503126244_413223246
[000000515481] gPTPd: @@@ SyncRxTime: 1503126244_412314482
[000000515481] gPTPd: @@@ lDelayTime: 0_000000823
[000000515481] gPTPd: @@@ CurrSynOff: 0_000909587 (-1)
[000000515481] gPTPd: @@@ prSyncTime: 1503126244_412608332
[000000515481] gPTPd: @@@ poSyncTime: 1503126244_413521159

[000000548522] gPTPd: @@@ SyncTxTime: 1503126277_455258890
[000000548522] gPTPd: @@@ SyncRxTime: 1503126277_454320299
[000000548522] gPTPd: @@@ lDelayTime: 0_000000823
[000000548522] gPTPd: @@@ CurrSynOff: 0_000939414 (-1)
[000000548522] gPTPd: @@@ prSyncTime: 1503126277_454629596
[000000548522] gPTPd: @@@ poSyncTime: 1503126277_455572122

[000000580533] gPTPd: @@@ SyncTxTime: 1503126309_467404830
[000000580533] gPTPd: @@@ SyncRxTime: 1503126309_466494533
[000000580533] gPTPd: @@@ lDelayTime: 0_000000820
[000000580533] gPTPd: @@@ CurrSynOff: 0_000911117 (-1)
[000000580533] gPTPd: @@@ prSyncTime: 1503126309_466808070
[000000580533] gPTPd: @@@ poSyncTime: 1503126309_467722243

[000000067065] gPTPd: @@@ SyncTxTime: 1503126390_087735838
[000000067065] gPTPd: @@@ SyncRxTime: 1503126390_086816229
[000000067065] gPTPd: @@@ lDelayTime: 0_000000816
[000000067065] gPTPd: @@@ CurrSynOff: 0_000920425 (-1)
[000000067065] gPTPd: @@@ prSyncTime: 1503126390_087275478
[000000067065] gPTPd: @@@ poSyncTime: 1503126390_088199109

[000000099106] gPTPd: @@@ SyncTxTime: 1503126422_130197434
[000000099106] gPTPd: @@@ SyncRxTime: 1503126422_129284587
[000000099106] gPTPd: @@@ lDelayTime: 0_000000822
[000000099106] gPTPd: @@@ CurrSynOff: 0_000913669 (-1)
[000000099106] gPTPd: @@@ prSyncTime: 1503126422_129720851
```

```
[000000099106] gPTPd: @@@ poSyncTime: 1503126422_130637674

[000000132115] gPTPd: @@@ SyncTxTime: 1503126455_139586978
[000000132115] gPTPd: @@@ SyncRxTime: 1503126455_138646329
[000000132115] gPTPd: @@@ lDelayTime: 0_000000822
[000000132115] gPTPd: @@@ CurrSynOff: 0_000941471 (-1)
[000000132115] gPTPd: @@@ prSyncTime: 1503126455_139080660
[000000132115] gPTPd: @@@ poSyncTime: 1503126455_140025266

[000000165143] gPTPd: @@@ SyncTxTime: 1503126487_168284090
[000000165143] gPTPd: @@@ SyncRxTime: 1503126487_167371203
[000000165143] gPTPd: @@@ lDelayTime: 0_000000812
[000000165143] gPTPd: @@@ CurrSynOff: 0_000913699 (-1)
[000000165143] gPTPd: @@@ prSyncTime: 1503126488_167969880
[000000165143] gPTPd: @@@ poSyncTime: 1503126488_168886875

[000000197141] gPTPd: @@@ SyncTxTime: 1503126520_167590854
[000000197141] gPTPd: @@@ SyncRxTime: 1503126520_166650038
[000000197141] gPTPd: @@@ lDelayTime: 0_000000812
[000000197141] gPTPd: @@@ CurrSynOff: 0_000941628 (-1)
[000000197141] gPTPd: @@@ prSyncTime: 1503126520_167107372
[000000197141] gPTPd: @@@ poSyncTime: 1503126520_168052176

[000000229143] gPTPd: @@@ SyncTxTime: 1503126552_171112454
[000000229143] gPTPd: @@@ SyncRxTime: 1503126552_170199857
[000000229143] gPTPd: @@@ lDelayTime: 0_000000812
[000000229143] gPTPd: @@@ CurrSynOff: 0_000913409 (-1)
[000000229143] gPTPd: @@@ prSyncTime: 1503126552_170500402
[000000229143] gPTPd: @@@ poSyncTime: 1503126552_171416995
```

A.3 MAC AVB Diagnosis log

```
Ethernet Interface "en6"
    Interface has AVB enabled.
    BCM5701Enet
        gPTP Present: YES
        Link Valid: YES
        Link Active: YES
        MAC: a8:60:b6:14:a6:3f
        Wants Time Sync Service: YES
        Time Sync Required: NO
        Wants Streaming Service: YES
        Wants MSRP: YES
        Wants MVRP: YES
        Wants ADP: YES
        Wants ACMP: YES
```

```
    Wants AECP: YES
    Wants MAAP: YES
IOAVB17221RemoteEntity
    Time To Live: 62
    Entity ID: 0x04a316fffead4156
    Entity Model ID: 0x04a316ad41560001
    Entity Capabilities: 0x00008508
    Talker Stream Sources: 1
    Talker Capabilites: 0x4001
    Listener Stream Sinks: 1
    Listener Capabilities: 0x4001
    Controller Capabilities: 0x00000000
    Available Index: 92
    gPTP Grandmaster ID: 0xa860b6fffe14a63f
    Association ID: 0x0000000000000000
    MAC Addresses:
        04:a3:16:ad:41:56
    Wrote /tmp/avbdiagnose-2018-01-10-16-58-26-GMT+1/ALSA_AVB_Driver-0x04a316fffead4156.aemxml
IOAVB17221LocalEntity
    Time To Live: 12
    Entity ID: 0x35363bca4b08001d
    Entity Model ID: 0x35363bca4b080011
    Entity Capabilities: 0x00010008
    Talker Stream Sources: 1
    Talker Capabilites: 0x4801
    Listener Stream Sinks: 1
    Listener Capabilities: 0x4001
    Controller Capabilities: 0x00000000
    Available Index: 46
    gPTP Grandmaster ID: 0xa860b6fffe14a63f
    Association ID: 0x0000000000000000
    MAC Addresses:
        a8:60:b6:14:a6:3f
    Wrote /tmp/avbdiagnose-2018-01-10-16-58-26-GMT+1/ALSA_AVB_Driver-0x35363bca4b08001d.aemxml
IOMVRP has local attributes.
    VLAN ID: 2
        Registrar State: 0
        Applicant State: 5
IOMVRP has remote attributes.
IOMSRPListener has local attributes.
    Stream ID: 0x04a316ad41560001
        Four Pack: Ignore
        Registrar State: 0
        Applicant State: 5
IOMSRPListener has remote attributes.
    Stream ID: 0xa860b614a63f0000
        Four Pack: Ready
        Registrar State: 0
```

```
        Applicant State: 0
IOMSRPDomain has properties.
    MSRP Capable: YES
    Class A PCP: 3
    Class A VLAN ID: 2
    Class B PCP: 2
    Class B VLAN ID: 2
IOMSRPDomain has local attributes.
    Traffic Class: 6
        PCP: 3
        VLAN ID: 2
        Registrar State: 0
        Applicant State: 5
    Traffic Class: 5
        PCP: 2
        VLAN ID: 2
        Registrar State: 0
        Applicant State: 5
IOMSRPDomain has remote attributes.
IOMSRPTalker has local attributes.
    Stream ID: 0xa860b614a63f0000
        Destination MAC: 91:e0:f0:00:ca:2a
        VLAN ID: 2
        Max Frame Size: 832
        Max Interval Frames: 1
        Priority: 3
        Rank: 1
        Accumulated Latency: 123860
        Failure Bridge ID: 0x0000000000000000
        Failure Code: 0
        Registrar State: 0
        Applicant State: 5
IOMSRPTalker has remote attributes.
    Stream ID: 0x04a316ad41560001
        Destination MAC: 91:e0:f0:00:33:4b
        VLAN ID: 2
        Max Frame Size: 80
        Max Interval Frames: 1
        Priority: 3
        Rank: 1
        Accumulated Latency: 1000000
        Failure Bridge ID: 0x0000000000000000
        Failure Code: 0
        Registrar State: 0
        Applicant State: 0
IO8021AS has properties.
    AS Capable: YES
    Clock Identity: 0xa860b6fffe14a63f
```

```
    Grandmaster ID: 0xa860b6fffe14a63f
    Remote Clock Identity: 0x04a316fffead4156
    Remote Port Number: 1
    Link Propagation Delay: 576ns
    Remote is on this machine: NO
    Propagation Delay Request Log Mean Interval: 0
    Sync Log Mean Interval: 253
    Announce Log Mean Interval: 0
    Clock Priority 1: 248
    Clock Class: 248
    Clock Accuracy: 254
    Clock Priority 2: 248
IOAVBInputUserSpaceStream
    Stream ID: 0x04a316ad41560001
    PCP: 3
    VLAN ID: 2
    Destination MAC: 91:e0:f0:00:33:4b
    PID: 218
IOAVBOutputUserSpaceStream
    Stream ID: 0xa860b614a63f0000
    EtherType: 0x22f0
    PCP: 3
    VLAN ID: 2
    Source MAC: a8:60:b6:14:a6:3f
    Destination MAC: 91:e0:f0:00:ca:2a
    PID: 218
    Output Frames: YES
```

Bibliography

[1] xmos.com, "AN00202 AVB Endpoint," 2017. [Online]. Available: http://www.xmos.com/published/an00202-gigabit-ethernet-avb-endpoint-example-using-i2s-master?version=latest

[2] beagleboard.org, "Beagle Bone." [Online]. Available: https://beagleboard.org/

[3] A. Rubini and J. Corbet, *Linux Device Drivers, 2nd Edition.* O'Reilly&Associates Inc, June 2001.

[4] IEEE, "IEEE standard for a precision clock synchronization protocol for networked measurement and control systems," *IEEE Std 1588-2008*, 2008.

[5] ——, "IEEE standard for local and metropolitan area networks-virtual bridged local area networks–amendment 9: Stream reservation protocol (srp)," *IEEE Std 802.1Qat*, 2010.

[6] ——, "IEEE standard for device discovery, connection management, and control protocol for ieee 1722(tm) based devices," *IEEE Std 1722.1-2013*, 2013.

[7] J. Watkinson, *Art of Digital Audio.* Taylor & Francis, 2013. [Online]. Available: https://books.google.de/books?id=WI4gJGEb_i0C

[8] B. McCarthy, *Sound Systems: Design and Optimization: Modern Techniques and Tools for Sound System Design and Alignment.* Taylor & Francis, 2016. [Online]. Available: https://books.google.de/books?id=FMejCwAAQBAJ

[9] M. Walker, "Choosing an audio interface," 2008. [Online]. Available: https://www.soundonsound.com/sound-advice/choosing-audio-interface

[10] J. Huntington, "AVB and Audinate's Dante: An Audio Networking Update After Infocomm 2016," 2016. [Online]. Available: http://controlgeek.net/blog/2016/6/25/avb-and-audinates-dante-an-update-after-infocomm-2016

[11] T. Shuttleworth, "The Nominees are AVB or Dante; And the Winner is?" 2015. [Online]. Available: https://www.linkedin.com/pulse/nominees-avb-dante-winner-tim-shuttleworth/

[12] A. Inc., "Dante Overview," 2018. [Online]. Available: https://www.audinate.com/solutions/dante-overview

[13] A. Diarra, T. Hogenmueller, A. Zimmermann, A. Grzemba, and U. A. Khan, "Improved clock synchronization start-up time for ethernet avb-based in-vehicle networks," *Proceedings of the IEEE 20th Conference on Emerging Technologies And Factory Automation (ETFA)*, 2015.

[14] xmos.com, "sw_avb," 2017. [Online]. Available: https://github.com/xcore/sw_avb

[15] J. Koftinoff, "AVB Statusbar," 2017. [Online]. Available: https://avb.statusbar.com/

[16] Several, "PTP daemon," 2017. [Online]. Available: https://github.com/ptpd/ptpd

[17] R. Manzke and H. Langer, "Embedded multichannel linux audiosystem for musical applications," in *Proceedings of the 12th International Audio Mostly Conference*, ACM, Ed., 2017.

[18] R. Love, *Linux Kernel Development (3rd Edition)*. Pearson publications Inc, 2010.

[19] IEEE, "IEEE standard for audio video bridging (avb) systems," *IEEE Std 802.1BA*, 2011.

[20] ——, "IEEE standard for local and metropolitan area networks-virtual bridged local area networks - amendment: Forwarding and queuing enhancements for time-sensitive streams," *IEEE Std 802.1Qav*, 2009.

[21] ——, "IEEE standard for a transport protocol for time-sensitive applications in bridged local area networks," *IEEE Std 1722-2016*, 2017.

[22] Beagleboard, "GSoC - Beaglebone AVB Stack," 2017. [Online]. Available: https://elinux.org/BeagleBoard/GSoC/2017_Projects#Project:_BeagleBone_AVB_Stack

[23] ——, "GSoC - Beaglebone AVB Stack Wiki," 2017. [Online]. Available: https://elinux.org/BeagleBoard/GSoC/BeagleBoneAVB

[24] C. E. Ralf Steinmetz, "Human perception of media synchronization," *Technical Report 43.9310, IBM European Networking Center Heidelberg, Germany*, 1993.

[25] J. Deber, R. Jota, C. Forlines, and D. Wigdor, "How much faster is fast enough?" *33rd Annual ACM Conference*, pp. 1827–1836, 04 2015.