Media Clock Recovery for IEEE 1722

On the Cyclone V FPGA

Presented by:
Keegan Crankshaw

Prepared for:
Dr. Simon Winberg
Dept. of Electrical and Electronics Engineering
University of Cape Town

Submitted to the Department of Electrical Engineering at the University of Cape Town in fulfilment of a Masters in Science degree.

11 December 2021

Key words:
FPGA, Ethernet, Media clock, Clock synchronisation, IEEE 1722, Clock Reference Format

The financial assistance of Uman Technologies ZA (Pty) Ltd towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at, are those of the author and are not necessarily to be attributed to Uman Technologies ZA (Pty) Ltd.

Declaration

I know the meaning of plagiarism and declare that all the work in the document, save for that which is properly acknowledged, is my own. This thesis/dissertation has been submitted to the Turnitin module (or equivalent similarity and originality checking software) and I confirm that my supervisor has seen my report and any concerns revealed by such have been resolved with my supervisor.

Keegan Crankshaw

Date: 11 December 2021

Acknowledgements

I would like to start by thanking the team at Uman for providing me with the opportunity to undertake this project and learn more about the field and the methods and technologies used.

I would like to thank Dr Simon Winberg for his guidance in forming this dissertation as well as his regular reviews of the content. His mentorship over the years has been invaluable to my personal and academic development.

I would like my friends, all of whom have been of great assistance. There are too many of you to name, but I must make a special thank you to Byron, Delia, Austin and Robin - without whom I likely would not have been able to complete this work. Whether it was through allowing me to talk them through a problem I was having, providing with me encouragement, a casual coffee break, or relaxation through gaming sessions - all of you played a critical role in helping me complete this work.

I would like to thank my family, who have always supported me in my endeavours to the fullest extent of their capabilities, without hesitation. Mom and Dad, you truly are the greatest role-models I could ask for. Thank you for your unending love and support.

And finally I would like to thank my cat, Kitkat "Madeline" Crankshaw, a four-pawed furry friend who kept me going through what was a very difficult year for many of us.

Abstract

The rise of the feature-rich vehicle has created the need for high bandwidth, low cost communication protocols between the components of the system. Traditionally these were implemented with point-to-point copper wiring using specified communication protocols, the wire harnesses of vehicles have become increasingly heavy, complex and expensive. Smart vehicles with features such as driver assistance, radar, cameras and many other sensors further increase the complexity of these systems, all of which are time-sensitive.

In terms of potential replacements to the wire harness, Ethernet has been regularly investigated as an alternative. Ethernet is cheap, easily scalable, has high bandwidth and is a well-established standard, but is traditionally not time-sensitive. In order to resolve this problem, IEEE 1722 presents methods for time-sensitive networking over Layer 2 of the OSI model.

This dissertation presents the work done to implement aspects defined by IEEE 1722, specifically the Clock Reference Format, on a Cyclone V field programmable gate array (FPGA) in order to synchronise two media clocks over Ethernet. This work is presented with the intention that the methods applied and developed can be extended to other time-sensitive applications such as control and sensor data, and in turn reduce the weight, cost and complexity of intra- vehicle communications.

This project was able to synchronise two asynchronous media clocks over Ethernet using the IEEE 1722 CRF protocol to within acceptable bounds. The experiments conducted to investigate the efficacy of the method determined that a phase correction to well within the media playback specification of 5% out of phase can be met, but that this result is dependent on environmental conditions.

Contents

List of Figures

List of Tables

Nomenclature

AFDX Avionics Full DupleX Switched Ethernet

ALM Adaptive Logic Modules

AVB Audio Video Bridging

AVTPDU AVTP Data Unit

AVTP Audio Visual Transfer Protocol

CAN Controller Area Network

Clock domain crossing Traversal of a synchronous signal from one clock domain to another

Clock drift An effect where a clock does not run at the same rate as a reference clock

Clock jitter The deviation of a clock edge from its ideal location

Clock recovery The process of extracting timing information from a serial data stream to allow the receiving circuit to decode the transmitted symbols

Clock skew When a clock signal arrives at components at different times

CRF Clock Reference Format

DCFIFO Dual-clock first-in first-out buffer

ECU Electronic Control Unit

ETH Ethernet

FPGA Field Programmable Gate Array

FQTSS Forwarding and Queing

FSM Finite State Machine

GLK Global clock network

gPTP Generalised Precision Time Protocol

GUI Graphic User Interface

IC Integrated Circuit

Infotainment Any hardware and software system providing information and entertainment to drivers and passengers

LAB Logic Array Block

LIN Local Interconnected Network

Listener An end station that is the destination, receiver, or consumer of an AVTP stream

Megafunction An IP Core provided by Intel for use on Intel FPGAs

Megafunction Intellectual property blocks developed by Intel (previously Altera) that enable functionality on an FPGA

Metastability Where the state of a signal is uncertain due to setup and hold times

MOST Media Oriented System Transport

MR Media Clock Reset. Used by the CRF protocol to indicate that the counter used in packets has been reset.

MSB Most Significant Bit

MTBF Mean Time Between Failures

Multicast Addressing a group of systems on a network simultaneously

OSI Open System Interconnect

PCLK Periphery clock network

PLL Phase-locked loop

RCLK Regional Clock Network

RX Receive

SRP Stream Reservation Protocol

Synchronise To cause two or more events to happen at exactly the same time or same rate, or in a time-coordinated way

Syntonise To adjust two electronic circuits or devices to operate on the same frequency

Talker An end station that is the source or producer of an AVTP stream

TSE Triple Speed Ethernet

TSN Time Sensitive Networking

TTE Time Triggered Ethernet

TTP Time-Triggered Protocol

TU Timestamp Uncertain. Used by the CRF protocol to indicate a timestamp value may be incorrect.

TX Transmit

Unicast Addressing a single system on a network

VHDL VHSIC-HDL, Very High Speed Integrated Circuit Hardware Description Language

Chapter 1

Introduction

1.1 Background

The rise of the feature-rich "smart" vehicle has created the need for high bandwidth, low cost communication protocols between components of the system. The self-driving car, for example, consists of many components such as anti-lock braking system (ABS), proximity sensors, cameras, radars, and the infotainment system. All of these components need to operate and be processed in real time in order to ensure a safe, comfortable experience for the drivers and riders alike. These features are less often seen as luxuries, and are now almost expected with newer models [1], particularly that of infotainment systems, which consist of information systems relaying information to the driver to enhance safety, and entertainment systems, most commonly audio-visual, which make time spent in the vehicle more enjoyable. These systems are often used as the features manufacturers use to advertise their product [2]. This means that infotainment systems not only provide safety and comfort to the users of the vehicles, but also serve a purpose in distinguishing a manufacturer's vehicles from that of their competitors.

Comfort and safety in automotive vehicles have always been a priority for both the manufacturer and consumer. To this end, many features such as parking assist, rear cameras, navigation and other advanced driver assistance features are no longer seen as luxuries in a vehicle, but are somewhat expected as included features [3]. With the increase of autonomous vehicles, additional communications for features such as radar, radio, cameras, infrared sensors and inter-vehicle communications will be required [1]. All these features and functionalities require communication between the given peripheral and a processing node, known as an electronic control unit (ECU). These may exchange up to 6000 messages over different in-vehicle networks such as LIN (Local Interconnect Network), CAN (Controller

Area Network), Flexray and MOST (Media Oriented Systems Transport) [3]. The increased complexity of these in-vehicle networks has lead to heavier, more complex wiring harnesses as shown in Figure 1.1 below.

The method for communication between components in vehicles has often been regular copper connections. These connections, known as wire harnesses, are heavy and messy. Consider the wiring harness of the Bently Bentayga, shown in Figure 1.1 below. This harness is used to connect over 90 computing units consisting of parking sensors, radar systems, stereo speakers, cameras, the suspension, and climate control (to name a few) and weighs around 50kg [4].

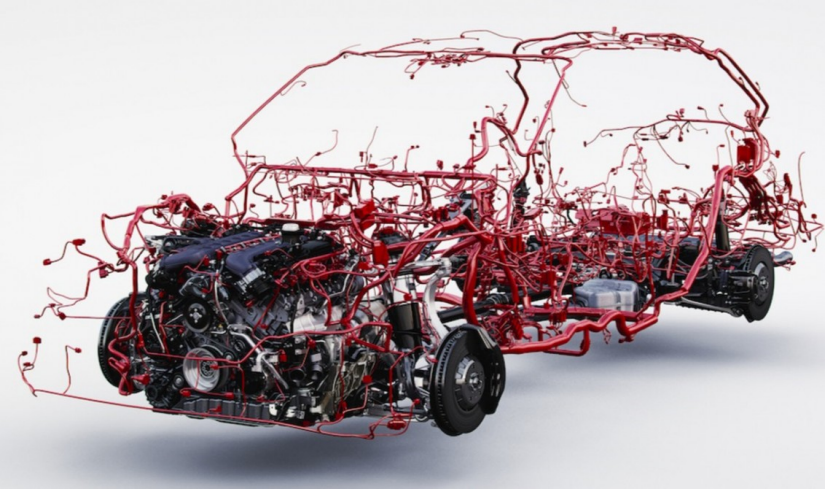


Figure 1.1: The wiring harness of a Bently Bentayga [4].

The complexity and cost of these wiring harnesses has led manufacturers to pursue alternate methods of in-vehicle communications. Ethernet - being cheap, easily scalable and having a well-establish standard - has been used in some instances and is currently being investigated further as a suitable replacement to complex and heavy wiring harnesses. Moving to Ethernet would result in a cheaper, lighter wiring harness which would allow savings to be passed on to the consumer [5]. The big concern with using Ethernet (and hence the low rate of adoption) is mostly due to the fact that Ethernet, unlike the Controller Area Network (CAN), Local Interconnect Network (LIN), and other currently used in-vehicle networking protocols, is neither real-time nor deterministic, but rather a "best effort" service. Vehicles, due to safety concerns, need to have real time constraints on the communications.

Ethernet works best under light loads where the measurement metric is given by average delay. However, in time-sensitive networking (TSN) the metric to measure network performance is worst-case delay. Classes of traffic also need to be distinguished in TSN in order to ensure critical data is transferred first [6]. As can be seen in Table 1.1 below,

in-vehicle communications have various classes of traffic all relating to different aspects of operation and comfort.

Table 1.1: In-vehicle domains and timing criticalness [3].

Domain	**Devices**	**Reliability Requirements**
Powertrain	Engine control, fuel injection, valve control	High
Chassis	Vehicle Dynamics control, braking system, airbag control	High
Driver assistance	Read and front view cameras, parking assistance system	Medium
Body and comfort	Climate control, locks, window and seat control	Low
Telematics and infotainment	Car stereo, GPS, internet	Low

The reliability requirement rating in Table 1.1 above, from high to low, indicates how reliable the communication between nodes in the domain need to be in order to ensure correct and safe operation of the vehicle. "High" indicates that the domain is system-critical - the vehicle cannot operate without it. "Low" indicates that the domain consists of devices which are related to operator experience, such as comfort and entertainment systems.

In summary, due to the weight and complexity of the traditional copper wire harnesses, manufacturers are exploring alternate communication methods for system components. Under heavy consideration is Ethernet. It is cheap and has been widely used in the past 30 years, but the latency, bandwidth and reliability of traditional Ethernet has required additional protocols to be developed. These protocols, broadly categorised under time-sensitive networking (TSN), will be further investigated in Chapter 2.

1.2 Problem Formation

Feature sets in vehicles have drastically increased over time, and integration of new technologies into vehicles has shown no signs of slowing down. Vehicles are now so feature-rich they have the ability to be fully autonomous. Naturally, the introduction of new features means there is an expectation from consumers that older features become more regular and common-place across the range of offerings, not only with newer high-end models.

Prior to the official introduction of the car radio by Motorola in the 1930s [7], consumers were carrying handheld radios into vehicles [8]. The desire for in-vehicle audio has been around since the introduction of the consumer vehicle and has not since subsided. Roughly ten years after the introduction of the first car radio, 30% of vehicles on the road contained one, and 50% of new cars sold had one. Technology has progressed a long way from there, and in recent years the rise of high-fidelity streaming platforms such as Spotify or YouTube Music means consumers have access to an overabundance of entertainment, which they now expect in-vehicle. The audio quality and fidelity expected has also increased. Paramount to high-fidelity digital audio playback is a constant clock which is synchronised across playback nodes.

Synchronisation of clocks can occur in many ways using messages on various communication protocols - some of which are discussed in Chapter 2. As presented in Section 1.1, Ethernet is currently being investigated as a means of transferring messages of various types between nodes in a vehicle - including clock synchronisation aspects. Hardware modules are responsible for generating, sending, receiving and processing these messages. Audio, considered a less timing-critical application [3], can be synchronised in a best-effort approach, giving priority of the network to systems which are considered more time-critical.

The particular research problem posed is to synchronise an audio clock signal across an Ethernet connection using data transmitted in the packet header by implementing hardware modules designed with reference to current standards and literature. The framework developed to solve this problem should be scalable to larger systems - that is, the design of the solution should not be limited solely to the scope of the problem. The smaller test case of synchronising audio clocks provides a relatively rapid method of implementation and testing by using provided hardware. It enables a set of simpler testing procedures in order to ensure the accuracy and precision of the developed framework, which can be applied to and tested on other systems in the future. Development of a synchronised audio clock is not a trivial task, however the level of testing and redundancies required for audio systems as opposed to safety-critical systems does mean that audio applications are easier to prototype and develop for. Once developed, the framework can be extended further, introducing additional testing and verification for safety-critical communication protocols. This enables this research to be used as a stepping-stone of sorts to an implementation suitable for more safety-critical systems.

A range of other fields beyond the scope of the automotive industry stand to benefit from time-sensitive networking implementations. There are many use cases for a protocol that enables transmission of time-sensitive data over an Ethernet network. Some more technical examples include control systems, alarm and security systems, and bistatic or multistatic radar systems. An example which is more directly applicable to the research presented in this

paper is professional audio.

Digital clock construction and correction is applicable in many fields, not just the examples given above. As systems become increasingly heterogeneous, the need for a synchronised distributed clock becomes increasingly important to ensure efficient and effective operation of digital systems. Recovery of the clock signal from factors such as drift, jitter and clock domain crossing are also areas of research significance.

Based on the problem defined above, as well as the additional motivations, some objectives of the project can be determined. These are as follows:

- Use VHDL to construct, send and receive Ethernet frames. These frames should include data required to be able to reconstruct and synchronise the sender's clock at the receiver's end.
- On the receiver, use the data in the frames to correct a digital clock that is in phase with the clock of the sender using appropriate methods.
- Verify the clocks are in phase using an oscilloscope.

1.3 Terms of Reference

The objectives defined in Section 1.2 are loosely defined. In order to better define the scope of the project, certain tasks can be extracted from the problem formation, summarised as follows:

- Develop a means for establishing a distributed clock time to enable synchronisation.
- Generate timestamps from this shared distributed clock on the talker based off the source clock.
- Transmit those timestamps over Ethernet in an established format to the listener.
- Unpack those formats on the listener and extract the relevant information.
- Use the unpacked relevant information to correct the generated clock on the listener to be synchronous to the talker.

After initial research given in Chapter 2, as well as discussion with the industry partner, these broadly defined tasks were further clarified and developed through a design described

in Chapters 4 and 5. The requirements were determined to be the following shown in Table 1.2 below.

Table 1.2: Requirements for the proposed design.

Requirement ID	**Description**
R01	A shared timing reference between both talker and listener
R02	Create a media clock of 48kHz to use as a source clock
R03	Generate timestamps using the timing reference in R01 and source media clock in R02
R04	Package the generated source timestamps in a format suitable for transmission over Ethernet, as described by IEEE 1722
R05	Transmit the packaged timestamps over Gigabit Ethernet in a way that makes it possible to observe them over the wire through an application such as Wireshark
R06	Generate a media clock of 48kHz which can be adjusted in phase in order to match up to the media clock in R02
R07	Generate timestamps of the generated media clock in R06 using the shared timing reference in R01
R08	Unpack the formatted source timestamps received over Ethernet and extract the relevant information
R09	Use the unpacked timestamps and compare them to the generated media clock's timestamps in order to determine a phase difference
R10	Use the phase difference in order to adjust the generated media clock to be in phase with the source media clock
R11	IEEE 1722 CRF standard and requirements should be implemented as much as reasonably possible

1.4 Scope and Limitations

This project intends to develop a set of HDL modules by which audio clock correction and synchronisation can occur over Ethernet. In order to accomplish this, a few modules need to be developed. Firstly (and most obviously) - the ability to transmit and receive packets over Ethernet. The formation and structure of these data packets will need to be investigated through the process of obtaining relevant literature, but it can be assumed that the IEEE 1722 standard will provide adequate information on these matters. A shared timing reference will need to be developed, and timestamps from this reference captured and transmitted when

appropriate. Two media clocks will be required, one source (or reference) clock, and another which needs to be corrected. These clocks will need to be compared using the shared timing reference, and the one to be corrected should be corrected accordingly so that both clocks align in frequency and phase.

Due to the nature of the project, the following limitations have been put in place:

- A shared clock, such as the generalised precision time protocol (gPTP), can be mimicked and does not have to be implemented
- The shared clock is accessible by all nodes on the network directly from shared FPGA logic, and does not require synchronisation over Ethernet
- The complete dissertation should be completed within two years
- The project must, as a minimum requirement, be implemented on the provided hardware with the given task of reconstructing and synchronising an audio clock signal

1.5 Dissertation Outline

This section outlines each chapter in the dissertation alongside a summary of the contents.

Chapter 2 reviews literature that details relevant protocols and investigates technologies available for time-sensitive networking. Context is provided for the platforms and technologies used in the research. The chapter starts by investigating time-sensitive networking as a whole: starting with its formation and working up to the current state of the suite of technologies it encompasses under the broad categories, with further detail into standards relevant to this research. The chapter then investigates other details pertinent to the research to be undertaken, such as other protocols, methods and hardware. Some of these topics include phase-locked loops, the TCP/IP stack and Ethernet, and specifications relevant to the hardware used.

Chapter 3 details the methodology and approach for the research tasks in the dissertation. The stages of the project are covered, from initial planning to completion, as well as the means and methods used to achieve the desired outcomes in each stage. Details are covered in how the project is approached and intended to be completed by looking at a hybrid Spiral-V model design process, as well as presentation of the funnel approach in developing the knowledge required to complete the project.

Chapter 4 starts an investigation into a conceptual design, determining functionality and further evaluating the requirements and specifications. This lays the groundwork for Chapter

5, which covers the design and operation of the project in detail. The detailed design in Chapter 5 covers the implementation details of each module that is found in the design. Information on design decisions and how the module operates is given, providing enough context to understand the nature of the design as well as ensure the requirements are met.

Chapter 6 contains details on multiple sets of experiments, each serving a specific purpose. The chapter starts by detailing the experimental setup and procedures for the experiments undertaken. Details of a simulation used as a proof of concept are given through means of discussion on how the simulation relates back to the modules described in Chapter 5. Subsequently, verification of the modules detailed in Chapter 5 is performed. This is done by investigating the behaviour of each module against its expected behaviour. A second set of experimentation is conducted on the modules at points of integration, to ensure data integrity and expected behaviour when using data passed out from one module into another. There is a considerable amount of experimentation contained in Section 6.3.4 for the `CSGEN` module detailed in Section 5.6. This is because the module is a closed-loop system, mimicking the behaviour of a phase-locked loop (the output of which is affected by an integrated circuit external to the FPGA, the CS2000, which is described in Section 5.5), and closed-loop systems can be complex and behave in unexpected ways. Modelling of the open-loop (the system without feedback) and extensive experimentation on the closed-loop system are required in order to ensure expected operation.

For each of the experiments detailed in Chapter 6, results are given in Chapter 7. These chapters work in-step, both following the same order of progression given above - results from the simulation, followed by results from testing each individual module, integration testing, and then finally additional testing and stress testing results. Each set of results has along with it a discussion, providing analysis of the results obtained and using them to verify requirements determined in the design chapters.

The dissertation ends with a conclusion in Chapter 8. This chapter is reflective, examining the degree to which the requirements in Table 1.2 were met. Future work to be conducted is presented in Section 8.3.

Chapter 2

Literature Review

This chapter consists of four primary categories, shown in Figure 2.1 below. The goal of this chapter is to explore literature relating to the research project, as well as cover aspects pertinent to the application.

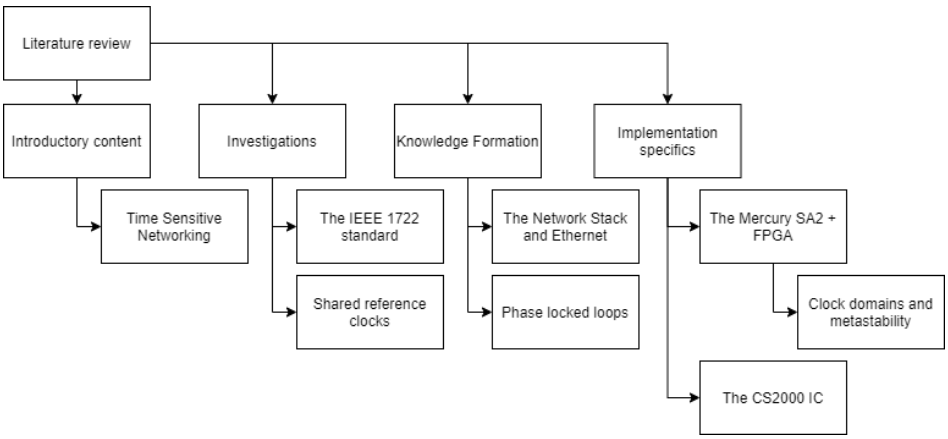


Figure 2.1: The structure of the literature review.

The first category covers introductory content. It starts with Section 2.1, which begins by investigating some current technologies in the automotive industry and how - over the years - these technologies have begun to fall short of the requirements of modern in-vehicle communications. White Rabbit, an alternative approach to time-sensitive networking developed by CERN, is briefly discussed. The section builds upon these discussions to show that the need for a protocol such as IEEE 1722 is well worth developing further and implementing due to its advantages and new features over pre-existing implementations - namely the higher speed and bandwidth enabled through the use of Ethernet.

The second category serves as a set of investigations into the IEEE 1722 standard as well as some information required for correctly implementing the standard. The section relevant to this category is Section 2.2.

The third category covers knowledge formation for the project, and investigates areas pertinent to the research which are required to form a better understanding of the tools and concepts used. These topics were researched as a means to gain the knowledge to implement potential solutions and requirements defined by the standards researched in the investigations category. Sections relevant to this category include Section 2.3 (which covers details regarding the Ethernet and TCP/IP stack) and Section 2.4 (which covers the basic design and operation of a phase-locked loop - a circuit that can adjust a signal's phase and frequency based on an input reference signal).

The final category consists of detailing hardware specific to the implementation as defined by industry partners. This category details and defines the features of the given hardware in order to implement a solution that takes advantage of these tools. This is namely information on how a field programmable gate array (FPGA) is applicable to this project, some considerations to be taken into account when using an FPGA, and the FPGA development board used (the Mercury SA2+), all of which is covered in Section 2.5.

The chapter concludes with a summary in Section 2.6.

2.1 An Overview of In-Vehicle Networking

This section investigates how in-vehicle communications and time-aware networking technologies have evolved over the years, and how the current state of time-sensitive networking emerged. There are many in-vehicle communication methods, and it would be near impossible to investigate them all. As a result a select few, based on prevalence in industry and relevance to this project, will be investigated.

The section begins by briefly investigating in-vehicle communications, namely the Controller Area Network (CAN) bus and FlexRay. Brief mention is made of the MOST (Media Oriented Systems Transport) communication protocol. The evolution of time-sensitive networking is considered, with reference to the Time-Triggered Protocol (TTP), Time-Triggered Ethernet (TTE), and the Avionics Full DupleX Switched Ethernet (AFDX). Mention is also made of White Rabbit, a time-aware networking technology used by CERN, and how this differes from the technologies investigated in this project. The chapter concludes by looking at how the Audio-Visual Bridging standard, a technology intended for professional audiovisual applications, evolved into the TSN standard.

2.1.1 Communication Protocols in Vehicles

Modern vehicles consist of multiple electronic control units (ECUs), which need to relay information between each other regarding the state of the vehicle and its surroundings. Up until the 1990s, ECUs communicated point to point [9]. This caused issues with weight, complexity, cost and reliability of messages passed. The move to a local area network (or LAN) for ECU communications was needed. A modern vehicle can have 25000 signals between up to 70 ECUs for very basic information such as speed or wheel rotation. The Controller Area Network (CAN) bus is likely the most common of communication protocols, selling over 400 million units.

2.1.1.1 The CAN Bus

Originally developed by the Robert Bosch GmbH Company in the early 1980s, the CAN bus is a highly pervasive communications protocol in the automotive industry [10]. It was designed to multiplex communications between ECUs, moving away from point-to-point communications. This helped reduce weight and complexity of the wiring harness and the vehicle as a whole. Peugot implemented the CAN bus in the Peugot 307, resulting in a wire count reduction of 40% in comparison to the 306 [9], proving its efficacy as a cost saving implementation. CAN is event-driven, meaning that messages are created and sent when events occur. This is as opposed to it being time-triggered, which is when messages are sent at specific timing intervals. These messages are 0-8 bytes in size, and are sent at a data rate of up to 1 Mbit/s. The standard supports transmission on a cable up to 500 meters in length, or up to one kilometre in length with appropriate repeaters and bridges. This isn't necessary in consumer vehicles, given the size of a modern vehicle.

2.1.1.2 FlexRay

There have been many efforts on extending CAN, however the limitations of the protocol, including concerns with the ability to work in safety-critical applications [9], have resulted in new standards being developed. One such protocol, primarily designed to overcome the bandwidth limitations of CAN, is FlexRay [11]. Developed in 2001, it is widely used in the automotive industry and serves as a hybrid of event-driven messages and timing driven messages on a shared bus [12].

While FlexRay does offer some improvements over the CAN bus, it is more expensive [13]. Considering the amount of ECUs in modern vehicles, there can be over 2000 wires, with the weight thereof being anywhere between 20-50 kilograms. Moving over to Ethernet would result

in a reduction in both weight and cost [14]. Ethernet provides much higher bandwidth than CAN or FlexRay. New feature sets, which have a large amount of streaming and control data, push these protocols to the limit. Manufacturers do consider Ethernet to be a viable option as a communication infrastructure or network overlay between which ECUs can communicate, as there are protocols and devices for switching between legacy communication protocols and Ethernet. FlexRay has around 70 parameters required for configuration [11], making it much more complex than a simple Ethernet based network.

2.1.1.3 MOST

The Media Oriented Systems Transport (MOST) communication protocol was designed specifically for media and general infotainment due to CAN being too slow to support video streaming [13]. It uses a synchronous, circuit-switched network across all nodes so that all devices run off the same clock.

MOST25, the first implementation, supported up to 25 Mbps. MOST25 was originally considered by some [15] to be sufficient for future audio-visual systems using a few small displays, but with the increase in number of devices as well as quality, this is clearly not the case. Netflix, a popular video streaming service, recommends at least 25 Mbps for "Ultra HD quality" [16]. Given the number of devices present in modern vehicles as well as the increase in expected quality of audio visual streams, much more than 25 Mbps of bandwidth will be required going forward. This works in favour of Ethernet, with 1000BASE-T having a bandwidth of 1 Gigabit per second. While MOST125, a more recent implementation, supports speeds of up to 125 Mbps [13], it is less flexible than standard Ethernet communications, which does not help alleviate the cost and weight concerns, making it less appealing to implement.

2.1.2 Existing Time-Sensitive Networking Technologies over Ethernet

This section seeks to explore some existing time-sensitive technologies that were considered modifications to Ethernet, particularly those that relate to the automotive domain.

2.1.2.1 Time Triggered Protocol

The Time Triggered Protocol (TTP) was developed in the 1980s. Originally considered for use in the automotive industry, the technology also found a way into the aerospace industry - with some notable uses being the Boeing 787, and the pressure control system of the Airbus A380. The protocol is deterministic, fault-tolerant, and has support for data rates of up to

20 Mbps. However the bandwidth use in the protocol is not optimised, especially given the increasing number of nodes in the network. In comparison to aforementioned networks, it is considered to be less flexible than FlexRay due to how data frames are transmitted and how FlexRay supports both time and event triggered messages [9] [17].

2.1.2.2 AFDX

In the avionics industry, the CAN bus is often used for sensors, actuators and cabin communications. For a backbone network as a means of interconnection between systems, the Avionics Full DupleX Switched Ethernet (AFDX) communications protocol is used. In the Airbus A380 and A350, AFDX is used to connect avionics systems. These aircraft still make use of communication protocols (for example, CAN) for networks with low-bandwidth requirements, such as sensor/actuator data and cabin communications [18].

2.1.2.3 Time Triggered Ethernet

Time Triggered Ethernet (TTE) was originally introduced as a time-triggered Switched Ethernet network for safety-critical applications [18] based on TTP [17] and standardised by SAE International (previously the Society of Automotive Engineers). It is built on AFDX and is considered to be an improvement on the technology, and can cater for mixed critical requirements [19]. A notable TTE project is NASA's Orion Multi-Purpose Crew Vehicle. Due to the use of time-triggered communications, TTE decreases modularity and increases complexity of a designed network - meaning the more nodes that are on a network, the more complex the implementation is. This makes implementation and especially reconfiguration difficult [19].

TTE supports strict timing constraints though infrastructure to support time-triggering, rate-constrained traffic, and best-effort traffic. This is done using an offline schedule table based on global synchronisation. It has the advantages of conceptual simplicity and low jitter, and adjusting the network is easier than TTP. However, bandwidth utilisation is not optimal. The system requires clock synchronisation, which adds a level of complexity [17]. The more flexible the network needs to be and the more complex the network is, the more complex the implementation is. TTE is at a disadvantage as it requires a new networking stack [19].

2.1.2.4 White Rabbit - An Alternative Approach

White Rabbit is a relatively recent project stemming from CERN, created out of the effort to renovate the particle accelerator's control and timing systems. It is based on well-known technologies, primarily IEEE 1588-2008 (network synchronisation protocol for sub-nanosecond synchronisation) and is developed to be open source for both hardware and software features. It replaced RS-422 as the control system. It is considered an alternative approach as it is built upon Synchronous Ethernet (also known as SyncE), and adds determinism [20], rather than building on top of standard Ethernet communications. White Rabbit offers some interesting perspectives on time-sensitive Ethernet communications - showing that it certainly is an achievable and implementable task, though this implementation does require special hardware. The White Rabbit implementation is showing adoption in other particle accelerator research facilities, such as the European Synchotron Radiation Facility in France, and the GSI Helmholtz Centre for Heavy Ion Research in Germany. It is being used outside of this application, too, as France, Finland and The Netherlands are currently investigating use of White Rabbit for synchronisation in their national time laboratories [21]. The recently approved draft for IEEE 1588-2019 High Accuracy default profile is heavily based off of the technology [22].

2.1.2.5 Time-Sensitive Networking

Time Sensitive Networking (TSN) consists of a set of standards that have either been approved or are in the process of being approved [17]. This set of standards allows for guaranteed timing behaviour on an Ethernet network using multiple features such as queuing, best effort traffic, and separate streams all controlled using a shaper to determine how much bandwidth each stream should be allocated [19]. It guarantees a low bounded transmission latency, bounded low jitter, and low rates of loss for time critical data [23].

TSN is an extension and improvement on audio video bridging (AVB), which was originally created for transferring time-sensitive audio-visual streams. Where AVB was designed only for the transport of these audio visual streams, TSN extends these standards to include control data traffic classes, enabling the transmission of other classes of time sensitive information. The main improvements of TSN over AVB are optimised synchronisation, the time aware shaper algorithm, frame preemption, and redundancy mechanisms. There are over ten TSN standards and protocols - all of which are considered aspects of time-sensitive networking [19].

TSN offers some advantages over previous protocols such as TTE. The nature of the scheduling algorithm in TSN means it is more suited to adapting to changes in configuration. Most significantly, while TTE requires a new networking stack, TSN can be implemented on

Layer 2 of the OSI model (see Section 2.3). This means there are no required changes to existing infrastructure [19].

TSN is advantageous over other time aware networks because it enables time critical data to be shared between various applications, all with different timing requirements. Not only is TSN suited for time-sensitive applications, but it can be used in many scenarios from 4G and 5G networks in applications such as machine control, and is almost essential for the continual development and increase in feature sets for Industry 4.0 [23].

2.1.2.6 Moving Towards Ethernet-based TSN Solutions

Through this section, various communication protocols were discussed. Each has its place and is, for the most part, well-suited to its application. However, based on the increasing number of integrated features and the high bandwidth requirements of modern systems, the move to Ethernet is becoming increasingly appealing. Ethernet is one of the most cost-effective, scalable solutions enabling high-speed communications [18] resulting in an increase in efficiency (with additional benefits including a reduction in weight of the vehicle [5]) and reduction in cost.

TSN makes the move to Ethernet easier by standardising and consolidating feature sets and requirements in an open standard. While the change will likely be slow and gradual rather than instantaneous [13], the on-the-shelf availability of Ethernet devices at low cost, serving high bandwidth over a network that is easily scalable [19], will almost certainly ensure that Ethernet will become the default in-vehicle communications standard [24].

In the next sections, investigations are made into how TSN standards can be applied to the infotainment aspect of in-vehicle communications, along with the knowledge, procedures and technologies that will make this possible. Specifically, focus is drawn to the IEEE Standard for a Transport Protocol for Time-Sensitive Applications in Bridged Local Area Networks [25], a standard originally developed by the Audio-Visual Bridging (AVB) task group - the group responsible for enabling the transmission of time-sensitive audio visual data over networks.

2.2 TSN over Ethernet and the IEEE 1722 Standard

IEEE 1722, “Standard for a Transport Protocol for Time-Sensitive Applications in Bridged Local Area Networks” [25], is a standard for the transferal of time-sensitive data over networks, worked on by the initial audio-video bridging (AVB) task group. The standard builds on the work done to develop the AVB standards in order to make them applicable to various

industries through supporting not only audio-visual data but time-sensitive control data too. Standardising the method for transmission of time-sensitive data over various mediums reduces costs and enables interoperability between systems. The standard defines various aspects for time-sensitive networking, along with various data formats to use in networks with varying applications, all over networks capable of supporting the technology. Most notably, this includes Ethernet.

In order to transmit data over Layer 2 of the OSI model (investigated in Section 2.3), IEEE 1722 defines data headers for various protocols. All of them are based on the AVTP base protocol. The base protocol will be briefly discussed before covering the standard applicable to this research - the Clock Reference Format (CRF).

For the sake of being concise only information relating to clock recovery will be covered in this section, namely some brief information on the base Audio Visual Transfer Protocol (AVTP) and the Clock Reference Format.

2.2.1 The AVTP Base Protocol

The AVTP base protocol specifies general information and details on how each particular format in the IEEE 1722 standard should be implemented. AVTP takes advantage of a few features (depending on the implementation) to enable audio, video and control data to be transported on a time-sensitive network and reproduced at multiple slaves, which are known as "listeners". There are two primary forms of clock information: clock recovery information - used to reconstruct a defined master (or "talker" clock), and presentation time - the time at which media is to be presented. As presenting media is beyond the scope of this research, this will not be investigated in this literature review.

Data is transmitted in the network using AVTP Data Units (AVTPDUs). The standard defines one common header applicable to all AVTPDUs. From the common header, three classes of headers are expanded upon, and from these three common headers, implementations are defined. This breakdown is shown in Figure 2.2, with the clock reference format highlighted to show how it builds on the common header.

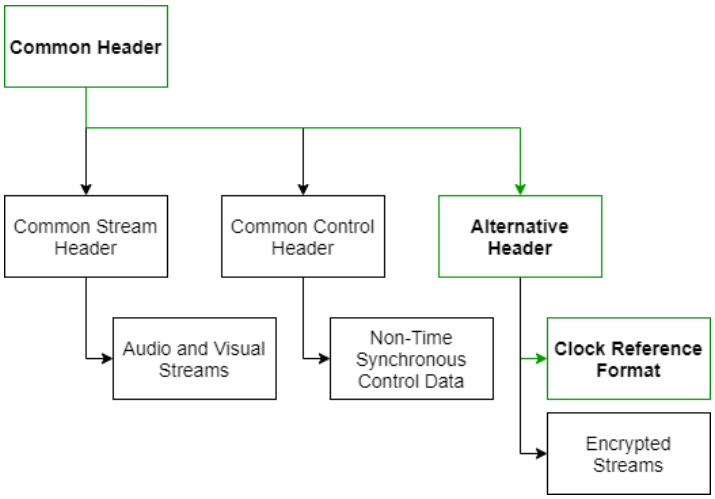


Figure 2.2: A breakdown of the AVTPDU header formats defined by IEEE 1722.

The common stream header is used for formats that require a stream that is time-sensitive. The common control header is used for non-time synchronous control data, and the alternative header is used for encrypted streams as well as the format applicable to this project - the Clock Reference Format (CRF). This header format will be discussed in full alongside the complete CRF implementation in Section 2.2.2.

2.2.2 The Clock Reference Format

The CRF header enables timing information to be distributed across the network and is the standard to be implemented in this project. It is comparatively simple to implement and can be used to minimise complexity in a distributed network, as AVTP end stations that only support data formats that are not time-sensitive are not required to support TSN services such as stream reservation and forwarding and queuing services. It supports formats for audio, video, control and other user-defined formats. It is also more efficient than other formats, as it only includes timing information. It is tolerant of packet loss, and can have data sent at any rate. The standard caters for the creation of multiple clock domains on the network by creating streams which in turn enable multiple clock rates. CRF is a flexible format as it can be used in any system where a synchronised clock is desirable, such as synchronising audio streams (as is the use case in this project), synchronising video playback or synchronising machine cycles.

Sections 2.2.3 and 2.2.4 will further detail aspects of the Clock Reference Format from IEEE 1772-10, bounded by the scope and limitations of the project.

2.2.3 Clock Reference Format Data Encapsulation

The CRF stream uses an AVTPDU alternate header to transmit relevant data over layer 2 of the OSI model. This alternative header is followed by a CRF specific header and then one or more timestamps derived from the shared reference timing source, which in IEEE 1722 is specified to be a generalised precision time protocol (gPTP) timing reference. The style of the header is shown in Figure 2.3. Fields are described in Table 2.1. The numbers running along the top of the image indicate bytes. These bytes (0-4) are further split into bits, of which there are eight per byte. The numbers running down the left hand side of the chart indicate the byte offset of the data in the packet. For example, the second line starts four bytes after the first line, and hence has an offset of "04".

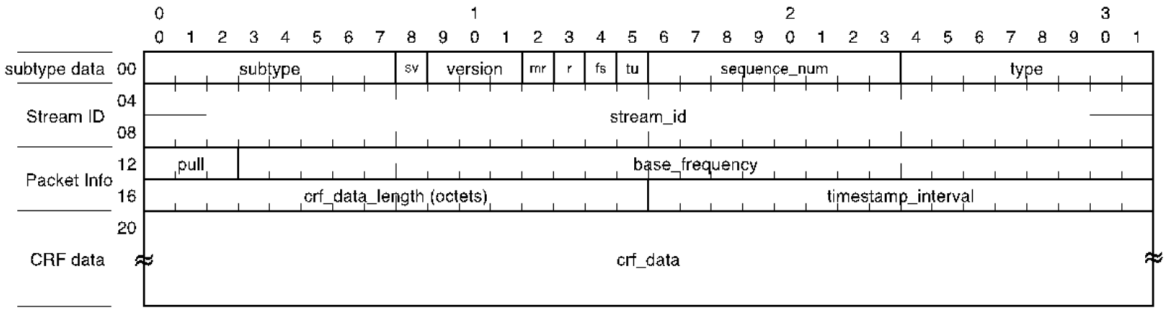


Figure 2.3: The Clock Reference Format Header [25].

Table 2.1: IEEE 1722 CRF header fields [25].

Field	Length	Description
version	3 bits	The protocol version
sv	1 bit	Stream Valid. Indicates if the StreamID value in stream_id is valid
mr	1 bit	Media Clock Reset. Used by the talker to indicate a change in clock. Stays at new value until clock changes again.
r	1 bit	reserved
fs	1 bit	Frame Sync. Enables a clock to also convey a frame synchronised clock. Using audio this field must be set to zero.
tu	1 bit	Timing Uncertain. Set by talker if the gPTP clock does not correspond to avtp_timestamp. When a listener receives a packet with this flag set, it can ignore the timing information and let the clock free-wheel.
sequence_num	1 octet	Sequence Number Field. Listeners can use the number to detect lost packets.
type	1 octet	Indicates the type of timestamp received. The type relevant to this research is 01_{16} which indicates audio data.
stream_id	8 octets	Consists of a EUI-48 MAC address and a 8 bit unique identifier.
pull	3 bits	Indicates a multiplier of the base_frequency field. Can have one of a set of specified values
base_frequency	29 bits	Defines the base sampling rate in Hz, between 1Hz to 536879911Hz ($1FFFFFFF_{16}$Hz). Adjusted by the value in the pull field.
crf_data_length	2 octets	Contains the length in octets of the crf_data field. Must be a multiple of 8, as all timestamps are 8 octets long.
timestamp_interval	2 octets	The number of events between each timestamp in the crf_data field, where an event is defined by the type of the clock. It is static and non-zero for the duration of the life of the stream.
crf_data	0 to n octets	The content of this stream depends on the type field.

A crf_data field (shown in Table 2.1) contains the timestamp. The CRF timestamp contains an 8-octet timestamp in nanoseconds and is defined by Equation 2.1.

$$message\_timestamp = (AS\_sec \times 10^9 + AS\_ns) mod 2^{64} \tag{2.1}$$

where: AS_sec = the gPTP seconds field
AS_ns = the gPTP nanoseconds field

The first timestamp in a CRF data frame is required to have a positive offset added, defined as follows:

$$T_{CRF} = T_S + (\lceil \frac{TT_{max}}{p} \rceil \times p) + T_C \tag{2.2}$$

where: T_{CRF} = CRF timestamp placed in the CRF AVTPDU
T_S = Original timestamp, sampled at source
TT_{max} = Max transit Time for the network
p = The period of the clock source
T_C = Amount of time that samples spend accumulating in the talker's transmit buffer

For CRF AVTPDUs the transmission rate depends on specific requirements which requires the talker to be implemented in a particular way. For a listener that supports CRF_AUDIO_SAMPLE (type field set to 1_{16}), any rate can be used, but the suggested rates include 300Hz, 44.1kHz, 48kHz, 88.2kHz, 96kHz, 176.4kHz, and 192kHz. 300Hz is included as it is the greatest common divisor that can be used to recover all sample rates.

If a CRF timestamp is not sent in a reserved stream, latency is then unpredictable and the CRF timestamp may arrive too late. If many timestamps are sent within an AVTPDU, the listener can lock its clock to those values. Once a clock is locked, timestamps can be processed less often to reduce overhead. If timestamps are lost due to network issues, the listener clock can freewheel until CRF timestamps are available.

If a listener is receiving timing information from both a CRF stream and a media stream, the maximum difference between timing points from either stream should not exceed 5% of the media sample period, as shown in Equation 2.3.

$$n \times P_s - \frac{P_s}{20} < T_{offset} < n \times P_s + \frac{P_s}{20} \tag{2.3}$$

where: T_{offset} = Timestamp offset in nanoseconds between the CRF stream and the AVTP Presentation Timestamp
n = Positive integer chosen for implementation
P_s = Sample Period of the CRF timestamp

If the incoming AVTP timestamp value from a media clock exceeds 25% of a sample period difference in alignment between it and the sample period, the listener may interpret the stream as invalid.

2.2.4 Requirements for the CRF Timestamp

Appendix F.12 of IEEE 1722 (included in this document as Appendix A) presents a list of 22 requirements for implementing the CRF standard. While the CRF standard is the one that needs to be implemented for this project, some of these implementation details are either outside of the scope of the project or not relevant to the implementation. As a result they have been removed from the list, with an abbreviated version shown in Table 2.2.

Table 2.2: The abbreviated CRF implementation checklist.

Item	**Requirement**
CRF-1	Does the device use the alternative header?
CRF-2	Is the subtype field set to CRF?
CRF-3	Is the mr bit set as described in section 10.4.3 of IEEE 1722?
CRF-5	Does the mr bit remain in its new state for a minimum of 8 CRF AVTPDUs?
CRF-6	Is the sequence_num field increment by one (1) with wrapping?
CRF-8	Is the base_frequency_field set to a value from 1 to 536 870 911 (1FFFFFFF16)?
CRF-9	Is the crf_data_length field value a multiple of 8 octets?
CRF-10	Is the timestamp_interval nonzero?
CRF-11	Is the fs bit set to zero (0) for CRF AVTPDUs with a type field set to CRF_AUDIO_SAMPLE?
CRF-15	Is a CRF Audio Listener capable of receiving CRF AVTPDUs at the rates given in IEEE 1722 Table 28?
CRF-20	In a CRF AVTPDU that contains multiple CRF timestamps, do the values of the timestamps increase monotonically, and with a constant nominal spacing in time, from the first timestamp to the last timestamp in the CRF AVTPDU when tu equals zero (0)?

CRF-21	In a CRF AVTPDU that contains multiple CRF timestamps, are all timestamps derived from a continuous gPTP clock reference?

Based on Table 2.2 above, a list of requirements can be generated. The items in the checklist can be collated and plans regarding the implementation can be summarised as follows:

- CRF-1, CRF-2, CRF-8, CRF-10, CRF-11
  These values can be hard coded in the module that implements the talker aspect of the CRF standard.
- CRF-3, CRF-5, CRF-6, CRF-9, CRF-20, CRF-21
  Can be implemented through logic in the module that implements the talker aspect of the CRF standard.
- CRF-15
  Can be implemented through logic in the module that implements the listener aspect of the CRF standard.

Further discussion of the timestamp values as well as offsets can be found in Chapters 4 and 5.

2.2.5 Towards Implementation

This section started with a brief overview of how IEEE 1722 came about and why it is a useful standard for implementation. It then delved into details of the standard which relate to the project and what requirements the standard has to consider an implementation correctly and fully implemented. As was discussed, not all of these requirements are pertinent to the given project, so a summarised list of requirements from the standard was presented. Upcoming sections discuss knowledge and aspects applicable to implementing this standard.

2.3 Layer 2 Ethernet Communications

This section covers some details relating to the Ethernet communications which are reffered to in other sections of the literature review. This section provides knowledge for and context to the terminology used in other sections. Details on how Ethernet is implemented in this project can be found in Section 2.5.4.

2.3.1 The OSI Model

The Open System Interconnect (OSI) is described briefly in order to provide context to the project and the references to "Layer 2 of the OSI model", as made in other sections. The OSI model is a means of breaking up methods of communications, allowing for a clear distinction between services, interfaces and protocols [26]. It consists of seven layers, each with a specified function, as shown in Table 2.3 below. Ethernet operates on the lower two layers, and IEEE 1722 operates on layer two - the data link layer - specifically.

Table 2.3: The seven layers of the OSI Model.

Layer	**Name**	**Description**
7	Application Layer	User-oriented. Allows applications access to networked protocols.
6	Presentation Layer	Manages encryption and decryption of data.
5	Session Layer	Establishes and manages connection between two nodes (start, control and stop).
4	Transport Layer	Reliable data flow (flow control, multiplexing, virtual circuit management and error correction and recovery). Most commonly TCP and UDP.
3	Network Layer	Addressing, routing, and network control. Determines the physical path the data will take.
2	Data Link Layer	Defines the format of the data on the network.
1	Physical Later	Deals with physical aspects (voltage, wire gauge, ports, etc). Converts bits to signals.

2.3.2 Frames

A frame is the name given to the data unit transmitted on layer two of the OSI model. Frames encapsulate the AVTP Data Units defined in Section 2.2.2. A frame consists of the fields shown in Table 2.4 below. CRF AVTPDUs fit within the "payload data" section of the frame.

Table 2.4: The ETH-2 basic frame format.

Field	**Length**	**Description**
Preamble	7 octets	Fixed Value of 0x55
Start Frame Delimeter	1 octet	Fixed value of 0xD5 which indicates the start of a frame
Destination Address	6 octets	LSB first
Source Address	6 octets	LSB first
Length/Type	2 octets	2 octet value equal to or greater than 0x600 indicates type field. Otherwise, contains length of payload data.
Payload Data	0..1500/9600 octets	Variable length data
Pad	0..46 octets	Padding
Frame Check Sequence	4 octets	Required only for gigabit Ethernet operating in half duplex, which the Altera Megafunction (used in this project) does not support.

2.3.3 Summary

In Section 2.3 the concepts required for understanding the networking (primarily Ethernet) aspects of the project were briefly introduced. More details on how Ethernet communications are implemented in this project can be found in Section 2.5.4 and Chapter 5.

2.4 Phase-Locked Loops

In this project knowledge of the components and design of a phase-locked loop (PLL) is significant as synchronisation of two signals, a function that can be performed by a PLL, is one of the primary goals of this research.

PLLs are closed-loop control systems that generate a signal that has a fixed relation to the phase of a reference signal [27]. PLLs can lock the output phase and frequency of an output waveform to that of an input waveform. They are commonly used in frequency synthesis (multiplying a reference signal by a constant to produce an output that is either faster or slower than the reference by some constant) [28]. PLLs can also be used as frequency demodulators, tracking generators or in clock recovery circuits in order to generate stable frequencies or recover signals from a noisy communication channel [29]. The performance of

a PLL is primarily measured by a metric known as lock time, which is the time required for the output to adapt and settle to changes to the input [27]. The simpler implementations of a PLL do not lock frequency, but rather phase. To this end, PLLs have a metric known as "centre frequency", which defines the free-running frequency of the output waveform.

2.4.1 PLL Overview and General Design

A generic PLL consists of three primary components [27], as shown in image 2.4:

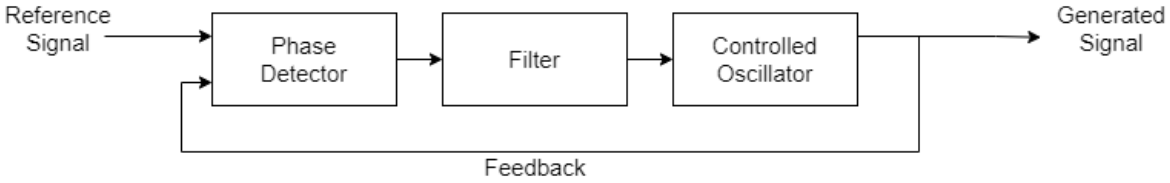


Figure 2.4: Simple components of a phase-locked loop.

These components, along with a brief description, are as follows:

- The phase detector
  The phase detector detects the phase difference between the input (reference) waveform and the output (generated) waveform.
- The filter
  A low-pass filter removes the high frequency components measured between the input and output waves resulting from changes to the controlled oscillator based upon the feedback of the loop.
- The controlled oscillator
  The filtered output from the phase detector adjusts the controlled oscillator to generate an output waveform that is in phase with the input. In an analog system the controlled oscillator is usually a voltage controlled oscillator, whereas in a digital system the easiest implementation is a numerically controlled oscillator. In a digital implementation that only requires a bistable clock, such as the implementation presented in this research, the controlled oscillator design can be further simplified by using the most significant bit of a counter.

2.4.2 Summary

This section investigated the general design and operation of a phase-locked loop. A phase-locked loop is a control system that generates a signal which has a fixed relation to the phase of a reference signal. Understanding of such a system is pertinent to the implementation of this project, as the system to be designed is to fulfil the same functionality.

2.5 Field Programmable Gate Arrays

This section discusses field programmable gate arrays, as well as the issues arising when using them in the context of this project - namely clock domain crossing. Details on the particular FPGA used (the Intel Cyclone V) is covered, as well as how a set of logic blocks, known as Megafunctions, are used to simplify development.

2.5.1 What are FPGAs?

Field programmable gate arrays (FPGAs) are programmable logic devices which can be reconfigured for various functions [30]. Primary manufacturers of FPGAs include Intel (previously Altera) and Xilinx (soon to be owned by Intel's competitor, AMD [31]).

A design is specified using a hardware descriptor language (HDL) such as Verilog or VHDL (VHSIC HDL - Very high speed integrated circuit HDL). These human-readable designs are synthesised to bitstreams which are in turn used to configure the FPGA as the HDL was designed.

FPGAs are often used in places where an application specific integrated circuit (ASIC) is desirable, but the quantity required does not validate the cost of producing one [32]. They are also used in applications where redesigns or "tweaks" may be required throughout the product or device life-cycle. FPGAs are useful in these cases due to their ability to have the logic rapidly reconfigured. FPGAs, due to their inherent parallel and task-specific nature, are considerably faster and more efficient at performing a specific task in hardware over something such as a general purpose CPU implementing the solution in software, despite common processors running at clock speeds multiple times higher than an FPGA. Given that the nature of this project is implementing a design that is time-sensitive, an FPGA is a good fit for a design solution.

2.5.2 Clock Domains and Metastability

A clock domain is a set of circuitry driven by a single clock signal. Multiple clock domain circuits have components of circuitry being driven by clocks of different rates. The nature of this project and the provided hardware require multiple clock domains, and therefore this section calls attention to clock domain crossing and metastability. This area of research is complex and involved. As a result, this section does not seek to explain all these circuit design concerns and considerations. Rather, this section briefly describes the concerns and provide solutions to mitigate the issues raised by using multiple clock domains.

The greatest issue arising when using multiple clock domains is that of metastability. Metastability refers to signals that do not have a stable digital (0 or 1) state at some point during normal operation of a design [33]. This is cause for concern as it means invalid or incorrect data can be propagated through a design. Metastability cannot be avoided in multi-clock designs, but it can be mitigated.

The measure of metastability is defined as mean time between failures (MTBF). This is where a failure is an observed signal is found to be in a metastable state. Larger values for MTBF are desirable as that indicates a longer time period between potential failures.

The primary means of mitigating metastability is through synchronisers. These are circuits which exponentially decrease the likelihood of metastable states [34]. The most commonly implemented of these is a dual flip-flop design. This is enough to synchronise the signal and increase MTBF sufficiently [33]. For signals consisting of multiple bits, the simplest means of transferring data between clock domains is a dual-clock, first-in first-out buffer - also known as a DCFIFO.

2.5.3 The Mercury+ SA2 and the Altera Cyclone V

This project makes use of the Mercury+ SA2. The Mercury+ SA2 is a "high-performance embedded processing solution, combining the flexibility of a CPU system with the raw, real-time parallel processing power of an FPGA system-on-chip (SoC)" [35]. It features an Intel Cyclone V, DDR3L SDRAM, eMMC flash, quad SPI flash, a Gigabit Ethernet PHY, dual Fast Ethernet PHYs and an RTC. Of primary interest to this project are the dual Ethernet PHYs, which will be used to send and receive the CRF packets described in Section 2.2.2. At the heart of the Mercury+ SA2 board is the Altera Cyclone V.

The Cyclone V is an FPGA designed by Altera. It consists of multiple elements, such as Logic Array Blocks, Digital Signal Processing Units, Embedded Memory Blocks, clock

networks, and Input/Output features. It also has interfaces for external devices such as external memory.

Logic Array Blocks (LABs) in the Cyclone V consist of ten Adaptive Logic Modules (ALMs), which is where most of the logic functions, arithmetic functions and data functions of a design are implemented. This is the re-configurable fabric of the FPGA.

The clock network in the Cyclone V consists of three clock regions, namely the Global clock network (GLK), regional clock network (RCLK) and Periphery clock network (PCLK). The GCLK drives the clock through the FPGA, and has low skew for functional blocks such as the aforementioned ALMs and DSP units. The GCLK also drives phase-locked loops on the device. The RCLK is only available in the quadrant they drive into, and provide the lowest possible delay and skew for logic in a quadrant. Periphery clocks have higher clock skew, and are primarily used for driving signals into and out of the Cyclone V.

Phase-locked loops (PLLs, covered in Section 2.4) in the Cyclone V can drive both global (GLK) and regional (RCLK) networks. There are up to 8 PLLs in Cyclone V devices. Multiple clock domains will be required in the research project in order to correctly implement Gigabit Ethernet, a 48kHz media clock, and the logic required. The Intel PLL Megafunction [29] discussed in the next section makes use of the PLL logic and clock domains in order to implement the desired clocks.

2.5.4 Intel Megafunctions

Intel Megafunctions are intellectual property blocks developed by Intel (previously Altera) that enable functionality on the FPGAs. They are pre-made and pre-tested logic blocks that can be easily implemented into a design. By using these functional blocks, design and implementation time can be reduced. This section provides a brief overview to the Megafunctions which are used in this project. The Megafunctions used in this project include the Triple Speed Ethernet (TSE) Megafunction, the Phase-Locked Loop (PLL) Megafunction, and the Dual-Clock First-In-First-Out Buffer (DCFIFO) Megafunction.

The Intel TSE Megafunction [36] provides the means by which Ethernet is enabled in the research project. It handles various aspects of the layer one and layer two implementation specifics. It works by providing OSI Layer 2 information from the frame, and transmits that information. The Megafunction implements the preamble and start frame delimiter, as well as any additional padding and frame checking. It provides an interface for the developer to easily transmit and receive layer two data, resulting in a simpler implementation with faster development times.

The Intel PLL Megafunction [29] allows the creation of clock domains from a singular clock source, routing the new clocks using the clocking fabric in Intel and Altera FPGAs. The Megafunction abstracts components and design of a PLL away from the user, instead allowing creation of new clock domains through the use of a graphic user interface (GUI) tool. This is the method by which the multiple clock domains required for this project were generated.

The Intel FIFO Megafuntion [37] provides a means in which data can be stored and retrieved on the Cyclone V FPGA. This includes methods for dual-clock FIFOs, which is the primary solution used to cross clock domains in this research project. More information on where and how this Megafunction is used can be found in Chapter 4 and specifically Section 4.1.2.

2.6 Summary of the Literature Review

This chapter contains material used in researching the literature relating to the synchronisation of media clocks over Ethernet from the perspective of the automotive use-case. This included review of various important technologies that will be utilised or built on this project. A brief history of communication protocols was investigated and the case for better technologies, namely use of protocols implemented over Ethernet, was made. Details regarding specifics of the implementation of the project were discussed, both through investigation into the relevant standards and detailing of the provided hardware with which the project should be completed. Research was done to learn about technologies required for implementation of the standards using the provided hardware. In closing the research presented in this chapter has formed the basis of knowledge required to undertake the project at hand.

Chapter 3

Methodology

This chapter defines the methodology that was undertaken in order to complete the project. As shown in Figure 3.1, the proposed approach was split into four primary phases, and further decomposed as elaborated on in the sections below. The four main phases were primarily run sequentially, though some moving between sections was done as it facilitated revision and better planning.

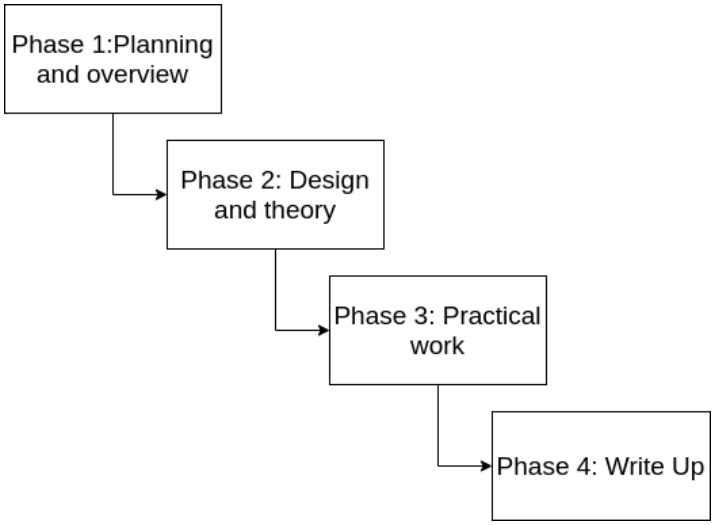


Figure 3.1: The phases of the project.

3.1 Phase 1: Planning and Overview

The planning and overview stage defines the course of research and experimentation, and sets up the methods and means by which research and experimentation was completed. This includes aspects such as getting acquainted with the project and its content, an initial literature review followed by some more in-depth research, and writing up of a project proposal.

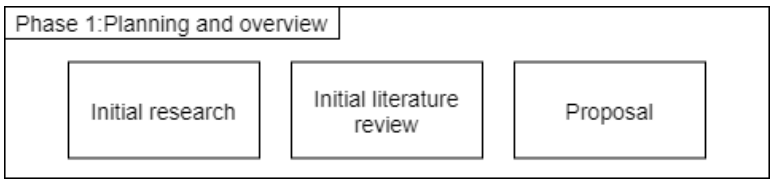


Figure 3.2: A breakdown of Phase 1.

A funnel approach (adapted by a design methodology presented by [38] and shown in Figure 3.3) was used for forming knowledge. First, notes, articles and general information relating to the field were researched. Based on knowledge obtained by reviewing these resources, additional work relating to the project was done to help ideate a design. Finally, research relating to the details of the implementation was done.

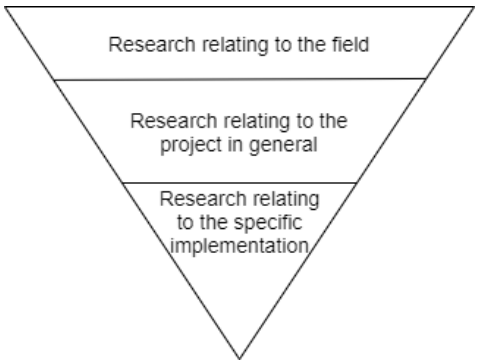


Figure 3.3: The Funnel approach used in researching the project.

3.1.1 Initial Research

Initial research was done into the topic. This process primarily involved gathering of resources which are somewhat related to the proposed topic. The goal of this task was to use gathered information to better formulate investigative questions which guided the research and ultimately the specifics of the implementation.

3.1.2 Initial Literature Review

In conjunction with the initial research, a literature review comprising of relevant standards, papers and investigations was compiled. This was done in an effort to gain knowledge relating to the field and ensure that the research conducted is of value. The funnel approach was applied in order to ensure that research pertinent to the topic received the most attention. The review started with an investigation into the IEEE 1722 standard, which was specified by the industry partner. Articles relating in-vehicle networking were reviewed. A structure of approach was formulated. The approach undertaken was to consolidate the work on in-vehicle networking, investigate how in-vehicle networking has changed over time, and determine the usefulness and applicability of the relevant standard presented in IEEE 1722. Once this connection was made clear, research into the technologies required for implementing a potential solution was conducted. Investigation into the provided hardware and how it applies to the domain was conducted. Considerations relating to the use of the given hardware, such as clock domain crossing and metastability, were investigated.

3.1.3 Research Proposal

The initial research and literature review was used to better define the investigative questions as well as define what the research aimed to achieve. This information was written in to the form of a research proposal, which contained other information such as overviews of literature and methodology, as well as an introduction to the project, the means by which the information and knowledge gained during the course of research will be disseminated, what was required to complete the research project, and the expected timeline for completion. This proposal was presented to and approved by industry stakeholders and the academic supervisor.

3.2 Phase 2: Design and Theory

This section of work involved the steps required to further investigate the specifics of the problem and formulate an initial design based on the information gathered in the initial stages.

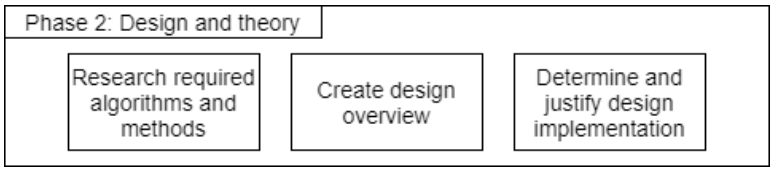


Figure 3.4: A breakdown of Phase 2.

3.2.1 Research Required Algorithms and Methods

Working in parallel with the design overview tasks, this task involved further investigation into components of the system which are critical for operation based upon information gathered in the literature review. From the literature review, it was found that research into the following aspects was required:

- sending and receiving packets over Ethernet
- constructing a clock
- developing a system for ensuring two asynchronous clock sources produce in phase clock signals

3.2.2 Create Design Overview

With knowledge gained from investigation into the subsystems of the project requirements, work began on a design overview. An initial design that meets all requirements and specifications was drafted and approved.

The design will formed from the considerations determined by the hardware available (described in Section 2.5) and the requirements of the CRF standard (described in Section 2.2.2).

The design overview is available in Section 4.2.

3.2.3 Determine and Justify Design Implementation

Once the modules of the design overview were defined, the possible methods of implementation were evaluated. Once decided upon, these methods of implementation were tested and contrasted in work discussed in Section 3.3.

3.3 Phase 3: Practical Work

This section of work relates to the practical tasks undertaken to meet the objectives of the project and conduct research into using IEEE 1722 CRF as a means of correcting two asynchronous clocks by transmitting information through an Ethernet connection.

The overall structure of Phase 3 can be found in Figure 3.5. As discussed later in this section, both Spiral and V-model design paradigms were used in the implementation and testing of the design. In short, the Spiral model was used when developing individual components within the design, but the V-model was used to validate and verify the overall operation of the system.

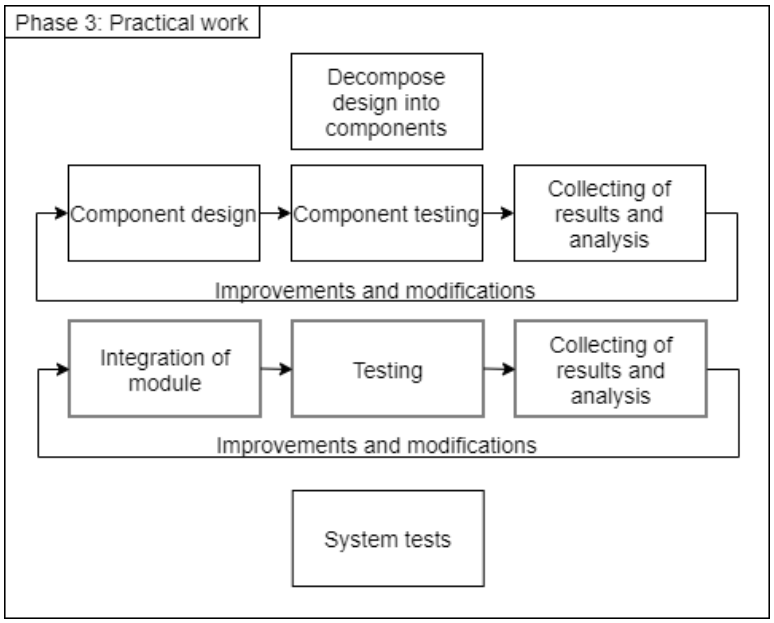


Figure 3.5: A breakdown of Phase 3

3.3.1 Decompose Design in to Components

The component design phase breaks down the larger system described in Section 4.2 to components that can be designed and implemented in VHDL. This work required understanding developed by the investigation into relevant literature and hardware components, as described in Chapter 2. Details of each of these components can be found in Chapter 5.

3.3.2 Component Design and Integration Testing

Once a component-based design was formed, these individual components were implemented and tested individually through applicable frameworks such as test benches or on-chip logic analysers. These components were integrated systematically based on how the system was intended to operate.

Once all relevant combinations of integration tests were run, the components of the design were integrated to the final monolithic design. Testing was conducted to ensure all modules still operated as expected when integrated into the complete system. From this point, final tests were done to ensure operation of the system as a whole, as well as to ensure the system met the requirements described in Sections 1.4, 2.2.4 and Table 1.2. The experimentation conducted is described in Chapter 6, with results given in Chapter 7.

3.3.3 Testing and Improvement Methodology

The approach to the testing, improvement and verification methodology in this project is a hybrid approach consisting of both the V-model and Spiral model.

While the outcomes of the project were defined based upon the list of desired outcomes provided by the industry partner, much investigation and research was required into the means and methods by which the outcomes needed to be achieved. As a result, the overall verification and validation process will use the V-Model. The V-model, shown in Figure 3.6, is well suited for final validation and verification. This means that completion of the industry partner's desired objectives could be achieved using methods and approaches which best satisfy the requirements developed through the research process.

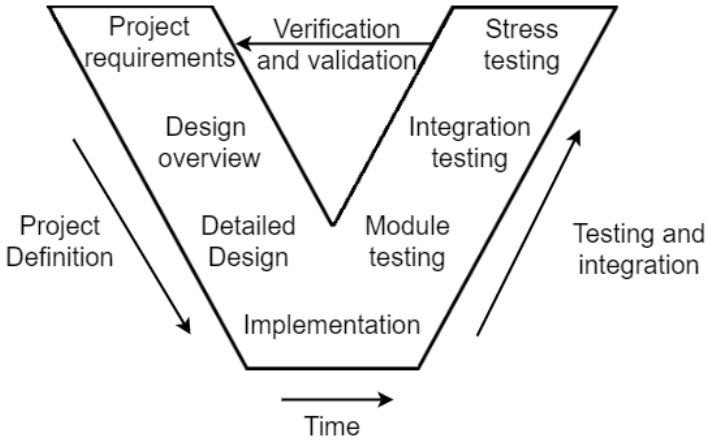


Figure 3.6: The V-Model, adapted to suit this project.

The Spiral model [39] - shown in Figure 3.7 - was used to design and test each individual component. While the overall objectives are defined, the means on how to achieve those goals are not. The Spiral model has the advantages of allowing features to be developed systematically and without changes to the outcomes of the project, rather than needing all requirements of each component in the design to be defined in the initial stages. The Spiral model also has the advantage of iterative testing - if these smaller modules were not attentively tested and improved upon, the complexity of testing and analysis required when the final validation and verification against the objectives provided by the industry partner would result in a much more time-costly process. Because of the nature of the spiral, features can continuously be added and improved upon. This design approach allows for later feature addition, if the design presented were to be incorporated into larger projects.

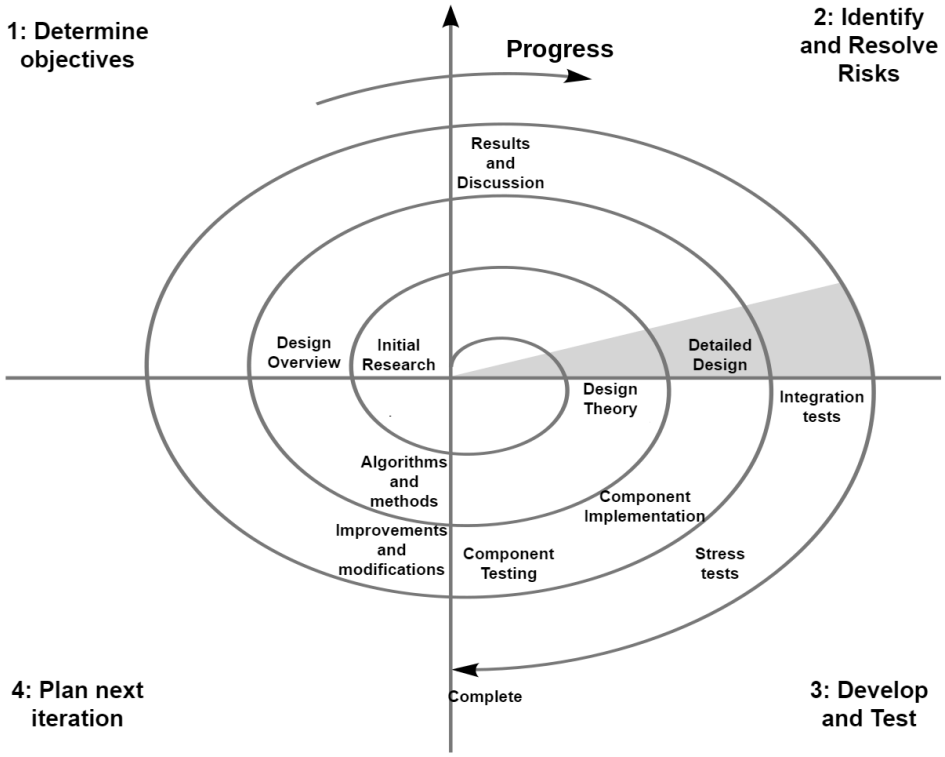


Figure 3.7: The Spiral Model, with adaptations for this project.

Design and implementation of hardware is a time-costly process, and thus being able to implement a smaller design, validate it, and apply new features to meet requirements in an iterative approach can save time which might otherwise be spent debugging monolithic designs.

3.4 Phase 4: Final Write Up

In the final phase of the project, the results of the experiments run in the previous phase were interpreted. These were used to construct a report in which the degree of success of the project is to be recorded. The write up process also includes consolidating the research and design aspects of the undertaken project as well as collating the work done into the write up.

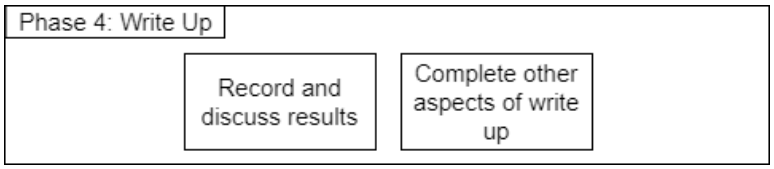


Figure 3.8: A breakdown of Phase 4.

3.4.1 Record and Discuss Results

The final stage allows time for final experimentation and validation not performed in prior stages to be completed. Results recorded through the implementation and experimentation phase were collated into Chapter 7. This stage also enabled the final validation and verification of the project to the initial objectives presented by the industry partner.

3.4.2 Complete Other Aspects

This phase consisted of time set aside for completing other aspects of the report, such as elaborating on work done, editing and corrections.

Chapter 4

High Level Design

This chapter outlines the high-level design of the system, delving into details where information gathered from the literature review as well as various documentation may impact the nature and design of the implementation. The chapter starts by listing design considerations based on the requirements and specifications, as well as any other considerations that occurred during stages of development. These considerations are used to create a high-level design. The high-level design is then investigated on a per-module basis, following the flow of data from master to slave. The requirements and specifications listed in Section 1.4 are expanded upon here into detailed requirements, shown in Table 4.1:

Table 4.1: Requirements for the proposed design.

Requirement ID	**Description**
R01	A shared timing reference between both talker and listener
R02	Create a media clock of 48kHz to use as a source clock
R03	Generate timestamps using the timing reference in R01 and source media clock in R02
R04	Package the generated source timestamps in a format suitable for transmission over Ethernet, as described by IEEE 1722
R05	Transmit the packaged timestamps over Gigabit Ethernet in a way that makes it possible to observe them over the wire through an application such as Wireshark
R06	Generate a media clock of 48kHz which can be adjusted in phase in order to match up to the media clock in R02
R07	Generate timestamps of the generated media clock in R06 using the shared timing reference in R01

R08	Unpack the formatted source timestamps received over Ethernet and extract the relevant information
R09	Use the unpacked timestamps and compare them to the generated media clock's timestamps in order to determine a phase difference
R10	Use the phase difference in order to adjust the generated media clock to be in phase with the source media clock
R11	IEEE1722 standard and requirements should be implemented as much as reasonably possible

4.1 Design Considerations

This section serves as a record of the design considerations which have been developed from the scope and limitations alongside the requirements and specifications. It provides some initial information which informed design decisions.

4.1.1 Considerations for IEEE 1722-CRF

Table 2.2 makes reference to requirements made by IEEE 1722 for the implementation of the CRF. The implementation is to be designed to implement these requirements in a best-effort approach. That is to say requirements R01 through R10 in Table 4.1 are given precedence over CRF1 through CRF22 given in Table 2.2.

4.1.2 Timing Considerations

As the application is time-critical, all delays need to be defined, measured, and accounted for. IEEE 1722-2016 defines a set of delays in the AVTP Application in Figure 6 [25]. The timing considerations listed in the image are as follows:

1. Jack to Jack Latency - Considered out of scope with referral to IEEE 1722.1-2013
2. Application to Application Latency - Considered out of scope with referral to IEEE 1722.1-2013
3. Min, Max and Actual Transmit Times - The transmission time was measured alongside implementation

4.1.2.1 Timing Delays from IP

In order to consider all potential sources of timing delay, the Intel TSE and Intel FIFO IPs were investigated.

In terms of the Intel TSE IP, there are two major considerations: Transmit (TX) latency and receive (RX) latency.

Transmit latency is the number of clock cycles the MAC function takes to transmit the first bit on the network-side interface (MII/GMII/RGMII) after the bit was first available on the Avalon-ST interface. Table 27 in the Intel TSE IP documentation [36], using RGMII in gigabit cut-through, TX requires 40 clock cycles. However, this includes the timing for half duplex support, statistic counters, and magic packet detection.

Receive latency is the number of clock cycles the MAC function takes to present the first bit on the Avalon-ST interface after the bit was received on the network-side interface (MII/GMII/RGMII). According to Table 27 in the Intel documentation [36], using RGMII in gigabit cut-through, RX requires 102 clock cycles. However, this includes the timing for half duplex support, statistic counters, and magic packet detection. In order to minimise the delays half duplex support, statistic counters and magic packet detection will be disabled.

In terms of the Intel FIFO delays, the following points are of consideration, there is time required for the read and write ready signals to be asserted. Table 4.2 below shows a summarised form of Table 7 in [37], showing signals only relevant to the implementation.

Table 4.2: Latency of signals in the Intel DCFIFO.

Signal	**Output latency**
wrreq to rdfull	2 wrclk cycles + following n rdclk
wrreq to wrempty	1 wrclk
wrreq to rdempty	2 wrclk + following n rdclk
rdreq to rdempty	1 rdclk
rdreq to wrempty	1 rdclk + following n wrclk
rdreq to rfull	1 rdclk
rdreq to wrfull	1 rdclk + following n wrclk
rdreq to q	1 rdclk

4.1.2.2 Transmit and Receive FSM Logic

Processing delays are expected on either end of transmit and receive finite state machine (FSM) logic - namely formation and deconstruction of the packets. The total delay depends on the implementation of the transmit and receive logic and hence needs to be determined after implementation.

4.1.3 Clock Domains

Due to the nature of the project, multiple clock domains exist within the design. Some of these include:

- Primary logic oscillator - 25 MHz
  The primary logic oscillator is a crystal oscillator on board the Mercury SA2+. It runs at 25 MHz, and is given as `osc25` in the constraints.
- Secondary logic oscillator - 125 MHz
  This is a 125 MHz oscillator generated by passing `osc25` through a phase-locked loop. It is given the name `clknet` in the top-level module. It is required in order to drive gigabit Ethernet. This clock signal will also be used to drive modules which would benefit from faster operation, such as the mock gPTP implementation.
- Primary media oscillator - 24.576 MHz (from crystal)
  An on-board 24.576 MHz oscillator (denoted `fref`) is divided down to a 48 kHz waveform to generate the source or primary media clock, denoted as `ref_fs_int`.
- Secondary media oscillator - 24.576 MHz (from CS2000)
  A secondary media oscillator is also available on board. This is generated by multiplying a generated waveform (outputted from the FPGA logic, denoted as `fout`) by a given clock - in this case, the same crystal oscillator as the primary media oscillator. This creates a second 24.576 MHz clock domain, which in turn is divided down to create a generated media clock. `fin`. It is by making adjustments to `fout` that `fin` is changed, in order to align `fin` with the generated `ref_fs_int`.

As mentioned in Section 2.5.2, working with multiple clock domains means consideration needs to be made for metastability. In this project, the Intel Dual Clock FIFO Megafuntion was used where necessary in order to ensure data was correctly and accurately moved between domains.

4.1.4 Review of Considerations

In this section various considerations for the design were discussed, namely timing considerations. The nature of the project and the implications of constant and regular correction mean that over long enough a period of time, these minor delays at various stages tend to zero (assuming a large enough sampling of the data). To this end, many of these delays are mitigated by regular correction of timestamps, and use of a large enough DCFIFO buffer able to hold the timestamps used in the correction algorithm. This is especially true when considering the scope of the project to be simply synchronising two clock signals relative to each other, on top of a timing abstraction made through the pseudo-gPTP implementation.

4.2 Design Overview

This section details the design in broad strokes. The design was be approached in a "top-down" method, starting with a conceptual overview, and then splitting the design into two (a "talker" and "listener" - a transmitter and receiver). The specific implementation of these components and their operation will be detailed in Chapter 5.

4.2.1 Conceptual Design

A conceptual design of the system can be seen in Figure 4.1. This figure shows how two asynchronous clock sources, driven from two separate oscillators, are passed into two logically separated circuits built into the FPGA, with information transferred between them over Ethernet.

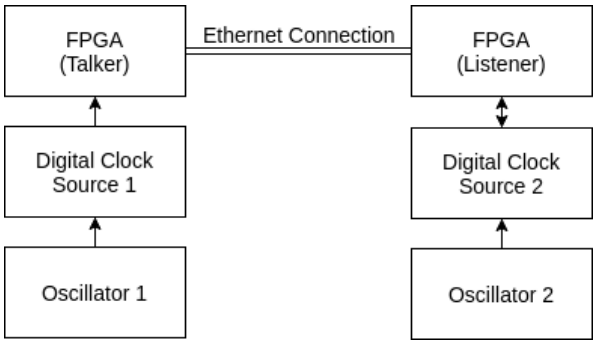


Figure 4.1: A simplistic conceptual design overview.

In the image, the blocks labelled "FPGA" are intended to include any logic to be

programmed in order to facilitate the desired operation of the device. These blocks are expanded upon in the upcoming sections.

This figure conveys an important nuance of the design: That even though there is a singular FPGA in use, it is important to implement the logic of the talker and listener independently of each other in order to imitate a real-world system, and to ensure separation of the source and generated clock domains.

The remainder of this chapter will be used to cover some more nuanced details pertaining to the design on either end of the Ethernet connection, namely the Talker and Listener designs.

4.2.2 Talker (Transmitter) Design

The "Talker" in IEEE 1722 refers to the circuit (or logic) responsible for generating timestamps to be used as the measure against which other timestamps generated by "Listeners" (circuits and logic where a clock is generated to be synchronised with the talker) are compared. The talker is responsible for the following requirements:

- R02 - Create a media clock of 48kHz to use as a source clock
- R03 - Generate timestamps using the timing reference in R01 and source media clock in R02
- R04 - Package the generated source timestamps in a format suitable for transmission over Ethernet, as described by IEEE 1722
- R05 - Transmit the packaged timestamps over Gigabit Ethernet in a way that makes it possible to observe them over the wire through an application such as Wireshark

Special consideration should also be made for R12, which dictates that IEEE 1722 standard and requirements should be implemented as much as reasonably possible.

Based upon the above, the design shown in Figure 4.2 can be conceptualised:

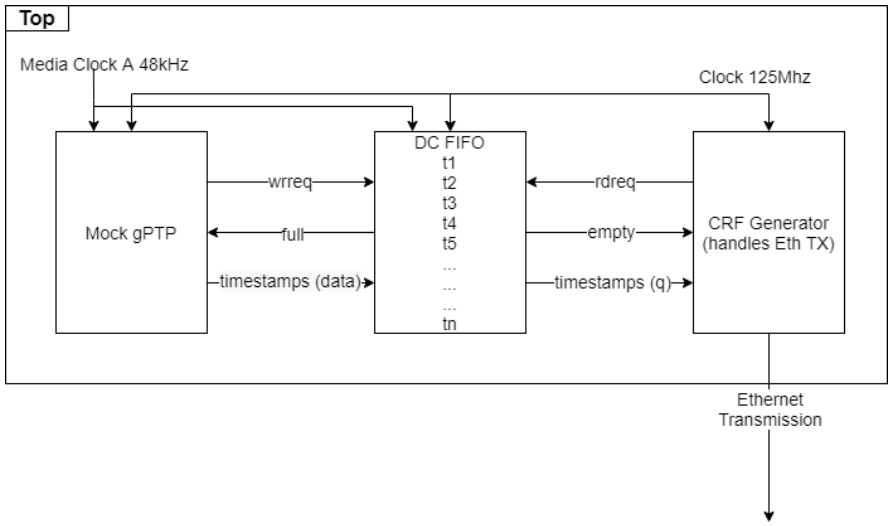


Figure 4.2: The conceptual design for the talker.

In this image the mock GPTP module used to create a timing reference is also shown. This data is used to generate timestamps based upon the source media clock. These timestamps are fed into a dual clock, first-in, first-out buffer. From the buffer, these timestamps move into a CRF Generator module, which will form the required ETH-2 data for transmission. The details of the mock gPTP module and CRF Generator are covered in Sections 5.1 and 5.2 respectively. The dual clock FIFO used is the Intel FIFO Megafunction.

4.2.3 Listener (Receiver) Design

A "Listener" in IEEE1722 "listens" to the talker, and adjusts the generated clock in order to synchronise the local clock with the talker based upon the received timestamps. The listener is responsible for fulfilling the following requirements:

- R06 - Generate a media clock of 48kHz which can be adjusted in phase in order to match up to the media clock in R02
- R07 - Generate timestamps of the generated media clock in R06 using the shared timing reference in R01
- R08 - Unpack the formatted source timestamps received over Ethernet and extract the relevant information
- R09 - Use the unpacked timestamps and compare them to the generated media clock's timestamps in order to determine a phase difference

- R10 - Use the phase difference in order to adjust the generated media clock to be in phase with the source media clock

Special consideration should also be made for R12 - "IEEE 1722 standard and requirements should be implemented as much as reasonably possible". This is to facilitate future conformance with the standard, as opposed to developing an implementation which is only applicable to the use case presented in this research.

Based on the above, a simple design can be conceptualised. Adding in factors for consideration listed in Section 4.1.3 as well as relevant requirements for IEEE 1722-CRF listed in Table 2.2 in Section 2.2.4, Figure 4.3 below presents the components of the design.

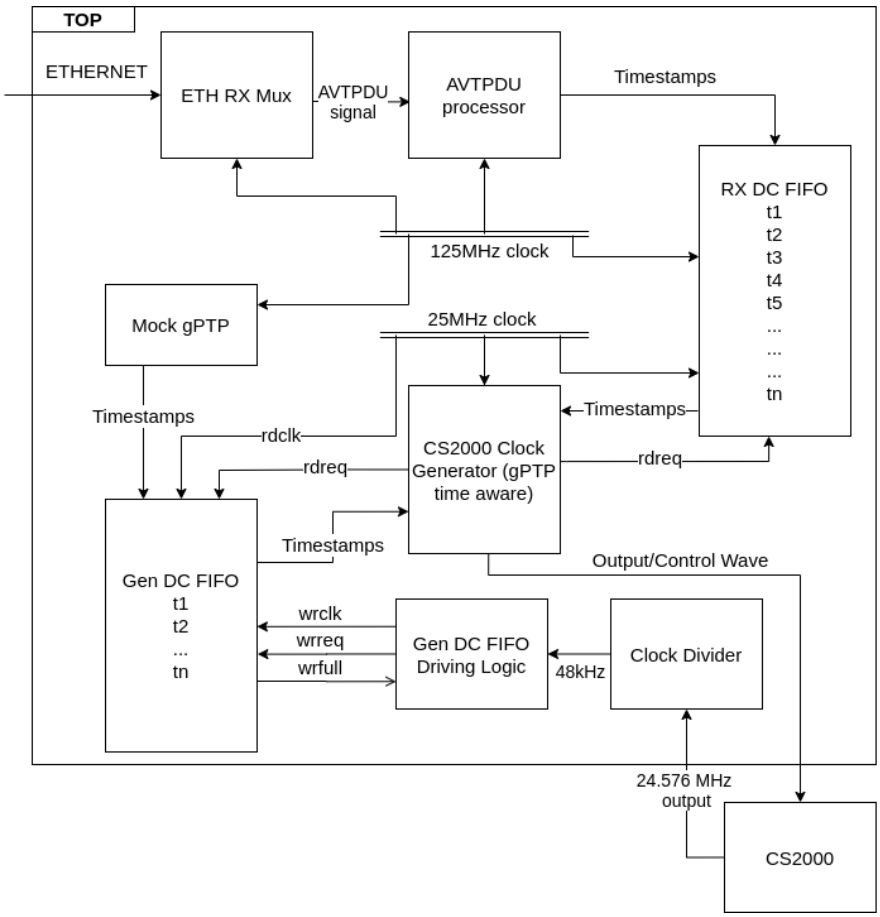


Figure 4.3: The conceptual design for the listener/receiver.

The design consists of three clock domains, namely the 125 MHz clock, a 25 MHz clock and the divided down 24.576 MHz clock from the CS2000 (which in turn is divided down to form the 48 kHz media clock).

It can be seen that the receiver also makes use of a mock gPTP module. This is the same module that is instantiated for the transmitter. To elaborate - there are not two instantiations of the same logic, rather the same circuit is accessed by both transmitter and receiver. This fulfils requirement R01 - a shared timing reference between both talker and listener. In practice the transmitter and receiver would both have their own timing reference, synchronised using a method described by IEEE 1588 [40]. However, synchronisation of timing references is beyond the scope of this project.

Ethernet data is received by the ETH RX Module, which is detailed in Section 5.3. The data is then conditionally passed to the AVTPDU Processor, which is detailed in Section 5.4. The reason for this separation of logic is to ensure compatibility and ease of implementation into pre-existing systems, as well as to easily add additional features to this system. Timestamps, as well as the timestamp uncertain (`tu`) and media clock reset (`mr`) flags are extracted from the frame. Timestamps are passed to a dual-clock FIFO, and the flags are passed directly to the CS2000 Clock Generator. The design of this module is detailed in Section 5.6. This module is responsible for creating an output wave (labelled "output/control wave" in Figure 4.3) which drives the CS2000 module and causes it to produce a 24.576 MHz waveform. This wave is passed back into the FPGA logic, where it is divided down to a 48 kHz clock, which is used to drive a secondary dual clock FIFO. The details regarding the CS2000 and clock divided are discussed in Section 5.6. These timestamps are fed back into the CS2000 Clock Generator circuit, where they are compared in order to determine any changes that are required to be made in order to align the generated media clock (created by the CS2000) with the source media clock (on the transmitter/talker).

4.2.4 Concluding Remarks

This chapter presented a set of design considerations which informed the conception of a high level design. Chapter 5 elaborates further on this high level design, going into detail on each module and component in the system.

Chapter 5

Detailed Design

This chapter builds upon the design overview presented in Chapter 4 and elaborates on the modules used in implementation. Throughout the chapter, repeats of the diagram originally shown in Figure 4.3 will be presented, with relevant modules being highlighted in the image relating to the given functional aspect of the design. The chapter starts by presenting the module responsible for the formation of timestamps on the talker in Section 5.1, as well as formation into frames and transmission of these timestamps over Ethernet in Section 5.2. Section 5.3 details the reception of the Ethernet data, and Section 5.4 discusses how they are processed to extract the relevant information. Section 5.5 details an integrated circuit used to generate a controllable output waveform and how that waveform is divided down to form the listener's media clock, as well as how that media clock is used to generate local timestamps. Finally, Section 5.6 discusses the formation and control of the waveform used to drive the CS2000 IC, as well as the control loop mechanism used to align talker and listener clocks. Code relevant to the modules can be found through a GitHub link given in Appendix D.

5.1 The gPTP Module

In order to satisfy R01 - a shared timing reference between both talker and listener - a timing reference accessible to both talker and listener must be made available. Without a shared frame of reference between both talker and listener, no corrections can be made. As described by [25] and covered in Section 2.2, this timing reference is created by a standard known as the generalised precision time protocol (gPTP). As implementation of gPTP is beyond the scope of this project, the gPTP field (and hence shared timing reference) will be substituted by a monotonically increasing 64-bit counter. The experimentation done to verify operation can be found in Section 6.3.1. The gPTP design is found in both the talker and the listener,

with this module being responsible for the functional blocks highlighted in Figures 5.1 and 5.2 below. In Figure 5.1, the gPTP module is responsible for writing to the DCFIFO. The CRF Frame Generator module, covered in Section 5.2, is responsible for reading from the DCFIFO.

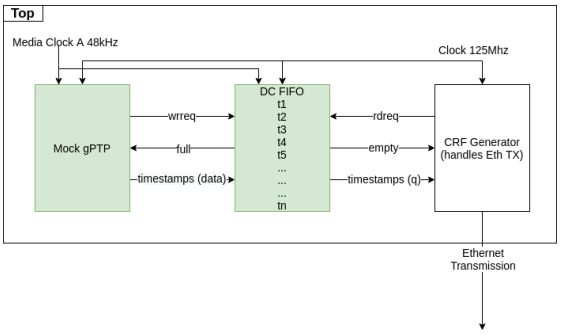


Figure 5.1: The gPTP module application in the talker design.

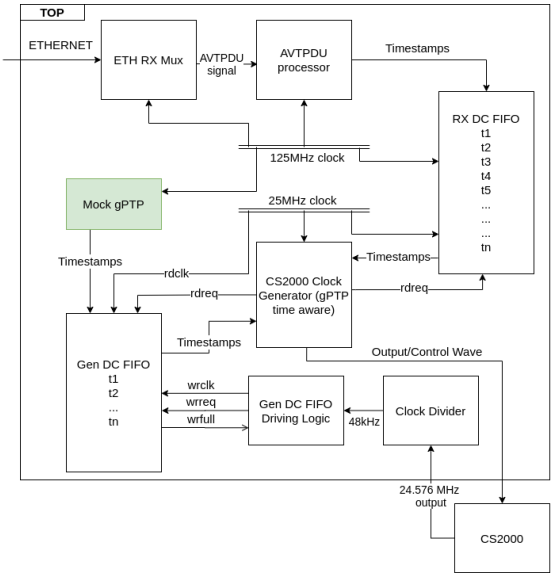


Figure 5.2: The gPTP module application in the listener design.

5.1.1 Requirements

The gPTP module is responsible for the following requirements:

- R01 - Create a shared timing reference between both talker and listener
- R03 - Generate timestamps using the timing reference in R01 and source media clock in R02

R01 is particularly important, as without a shared timing reference, there is no frame of reference by which to measure differences in the phase of the media clocks, and no measurable means of making adjustments. As before mentioned, complete instantiation of gPTP (a standardised means of creating a shared timing reference) is beyond the scope of this project and a "mock" gPTP implementation is instantiated as a 64-bit counter.

The inclusion of R03 in the gPTP module is used as a means of simplifying the design.

Reference is also made to the CRF checklist in Table 2.2 as this module fulfils the following requirements listed there:

- CRF-20 - The values of the timestamps increase monotonically, and with a constant nominal spacing in time, from the first timestamp to the last timestamp in the CRF AVTPDU when `tu` equals zero
- CRF-21 - In a CRF AVTPDU that contains multiple CRF timestamps, all timestamps must be derived from a continuous gPTP clock reference

5.1.2 Design Overview

The design of the gPTP module consists of four inputs and two outputs, as shown in the block diagram in Figure 5.3 below.

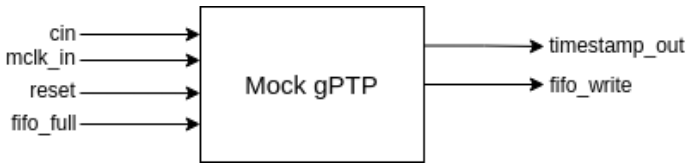


Figure 5.3: A block diagram of the I/O for the mock gPTP module.

The signals shown in Figure 5.3 are elaborated upon in Table 5.1:

Table 5.1: Inputs and outputs for the mock gPTP Module.

Signal	**Direction**	**Width**	**Description**
cin	Input	1	Primary logic clock
mclk_in	Input	1	Source media clock
reset	Input	1	Reset signal. Resets the counter value to 0
fifo_full	Input	1	The write full signal from the FIFO. Used to determine if the fifo_write signal can be asserted
timestamp_out	Output	64	The mock gPTP timestamp
fifo_write	Output	1	The write request signal to the TX FIFO

5.1.3 Operation

The module consists of two processes. The first process, based on `cin`, increases a counter each clock cycle (or resets it to 0 if the reset signal is pulled low). This process also ties the

counter value to the output signal `timestamp_out` which makes it available to be used in the dual clock FIFO which stores the talker timestamps. As the counter is intended to mimic gPTP, which operates in nanoseconds, this counter keeps track of nanoseconds too. The input clock, `cin`, operates at a frequency of 125 MHz. It has a period of 8 ns, therefore, at the rising edge of each clock cycle, 8 ns have passed. Using a base form of time (nanoseconds) as opposed to an arbitrary measurement (such as the number of 125 MHz clock cycles passed) also makes the timestamps easier to compare in a system consisting of multiple clock domains. Using a 64 bit number means there is a counter large enough to progress more than 584 years. An overview of the logic can be seen in the flow diagram in Figure 5.4.

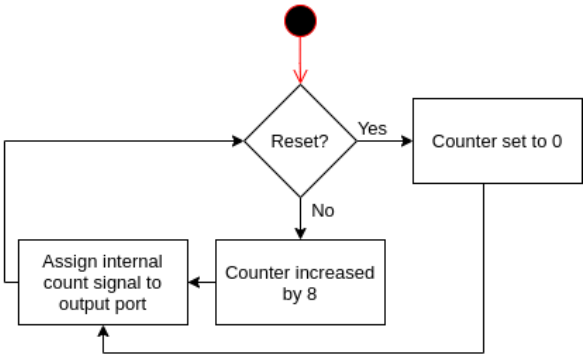


Figure 5.4: A flow diagram highlighting the basic operation of the mock gPTP counting logic.

The second process is run on `mclk_in`. As defined in Table 28 of [25] and as a requirement CRF-15 in Table 2.2, timestamps should be captured every 160th media clock event. In other words, timestamps should only be captured every 160th rising edge of the media clock. This process inside the mock gPTP module fulfils requirement R03 by assigning the write_request on the TX FIFO every 160th media clock. An overview of the logic can be seen in the flow diagram in Figure 5.5.

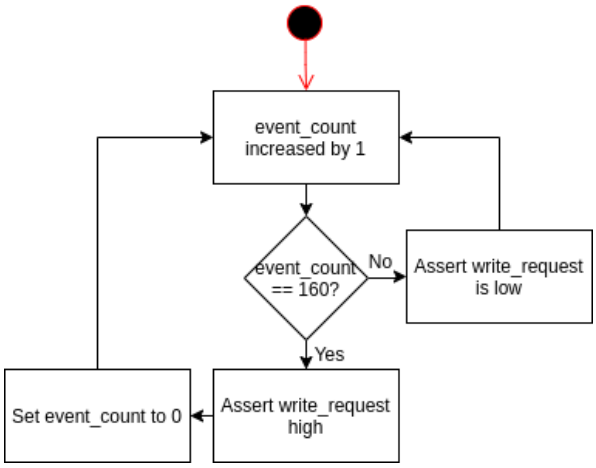


Figure 5.5: A flow diagram highlighting the basic operation of the mock gPTP output logic.

The TX FIFO captures the correct mock gPTP timestamp when the write request signal is asserted by the 48kHz clock process.

5.1.4 Summary

This module creates a mock gPTP implementation and fulfils requirements R01, R03, and aspects of CRF-15, namely the rate at which timestamps on the source clock are captured. This is achieved through the use of two processes running at 125 MHz and 48 kHz, controlling an Intel Dual Clock FIFO [37], ensuring correct values of the timestamps are captured. The experimentation done to verify operation of this module can be found in Section 6.3.1.

5.2 The CRF Frame Generator

The information relating to the format of Clock Reference Format (CRF) frames is drawn from Chapter 10 of IEEE 1722-2016 [25]. A breakdown of these frames has been detailed in Section 2.2.3, with Figure 2.3 providing a visual representation. This section of the design chapter details how CRF frames are generated and encapsulated in the design. Transmission of the frames is handled by Intel TSE IP [36]. The module is responsible for the logical blocks highlighted in Figure 5.6 below. It is responsible for reading from the DCFIFO pictured (whereas the gPTP module described in Section 5.1 is responsible for writing to it) and using those timestamps read to create CRF frames for transmission. Experimentation detailing the verification of this module is given in Section 6.3.2.

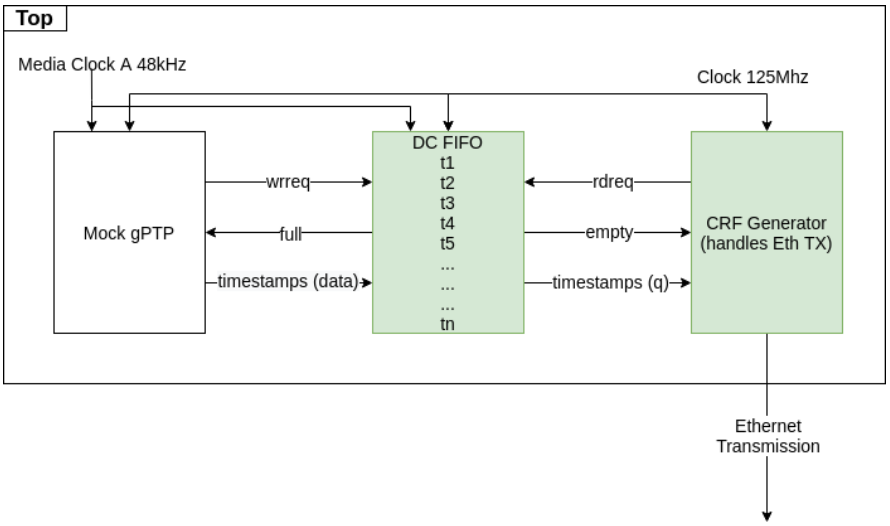


Figure 5.6: CRF Frame Generator module application in the talker design.

5.2.1 Requirements

The module is designed to fulfil the following requirements:

- R04 - Package the generated source timestamps in a format suitable for transmission over Ethernet, as described by IEEE 1722
- R05 - Transmit the packaged timestamps over Gigabit Ethernet in a way that makes it possible to observe them over the wire through an application such as Wireshark

In terms of requirement R12, further reference is made to Table 2.2, for which this module implements the following:

- CRF-1 - The talker uses the alternative header
- CRF-2 - The subtype field is set to CRF
- CRF-3 - The mr bit is set as described in Section 10.4.3 of IEEE 1722
- CRF-5 - Does the mr bit remain in its new state for a minimum of 8 CRF AVTPDUs?
- CRF-6 - Is the sequence_num field increment by one (1) with wrapping?
- CRF-8 - Is the base_frequency_field set to a value from 1 to 536 870 911 (1FFFFFFF16)?
- CRF-9 - Is the crf_data_length field value a multiple of 8 octets?

- CRF-10 - Is the timestamp_interval nonzero?
- CRF-11 - Is the fs bit set to zero (0) for CRF AVTPDUs with a type field set to CRF_AUDIO_SAMPLE?

5.2.2 Design Overview

The overview of the design is shown in Figure 5.7 below. Further elaboration upon the ports is made in Table 5.2. This section will also elaborate upon the implementation of the design in order to convey how the CRF requirements listed above are met.

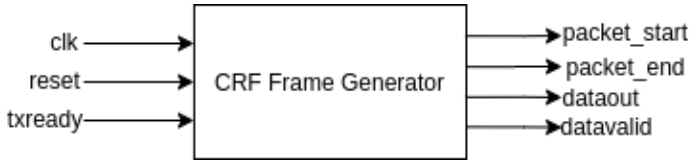


Figure 5.7: A block diagram of the I/O for the CRF Frame Generator module.

Table 5.2: The ports of the CRF Frame Generator.

Port	**Direction**	**Bitwidth**	**Description**
clk	Input	1	Clock input signal
reset	Input	1	Asynchronous reset
txready	Input	1	A signal from the MAC to indicate it is ready to receive data
packet_start	Output	1	Signal to indicate start of packet
packet_end	Output	1	Signal to indicate end of packet
dataout	Output	32	The four octets to be written to the MAC
datavalid	Output	1	Indicates to the MAC if the values on dataout is valid or not

The module uses the rising edge of the clock and the `txready` signals to determine when actions can take place. The input clock of the `CRFFrameGen` module is the same as that of the MAC function (125 Mhz) to prevent any boundary crossing issues when moving between clock domains.

5.2.2.1 Encapsulation

The AVTPDU can be encapsulated in a standard Type-2 Ethernet frame, which can be seen in Figure 5.8 below.

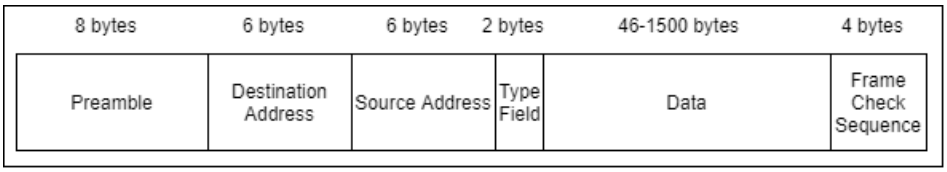


Figure 5.8: The standard Ethernet-2 frame.

The contents of the frame are described in detail in Section 2.2.2, with descriptions of fields in Table 2.1. For the purpose of context, some of the data in Table 2.1 is repeated in Table 5.3 below. While Table 2.1 gives descriptions, Table 5.3 indicates how the field was implemented. A field implemented "as per definition" is implemented per a particular definition given in IEEE 1722 [25]. Additional information or referencing is given where appropriate. In the table, some items are listed as constants or generics. Constants are values that are unchanging, regardless of implementation. Generics are parameters that can essentially be used as constants throughout a module, but can be changed with each new instantiation of the module. Due to the nature of the CRF packet generator and how it regularly re-uses the same data, there are a few generics that are defined. The benefit of generics is that they can be set when instantiated. This means that the module, as it is designed, could be used to instantiate multiple unique talkers.

Table 5.3: Values used in the CRF frames.

Field	**Generic or Constant**	**Value**	**Function**
MAC_DEST	Y	x"FFFFFFFFFFFF"	CRF Packets are broadcast, hence the destination MAC is the broadcast address. Required for the ETH-2 frame.
MAC_SRC	Y	x"123456789123"	An arbitrary but unique MAC address that can be determined as source when inspecting packets. Required for the ETH-2 frame.
ETH_TYPE	Y	x"22F0"	As per definition.
CRF_SUBTYPE	Y	x"04"	As per definition.
Version	Y	x"000"	As per definition.
Stream Valid	Y	'1'	Asserted as such for this implementation.

Media Clock Reset	N	0/1	As per definition (Section 10.4.3 of IEEE 172). Fulfils CRF-3 and CRF-5.
Reserved	Y	'0'	As per definition.
Frame Sync	Y	'0'	As per definition.
Timing Uncertain	Y	'0'	Asserted as such for this implementation.
Sequence Number	N	0-255	As per definition.
CRF Type	Y	x"01"	As per definition.
Stream ID	Y	MAC_SRC & x"0001"	As per definition.
Pull	Y	x"0000"	As 48 kHz fits within the base frequency, this can be set to 0.
Base Frequency	Y	x"000BB80"	48000 Hertz, the frequency of the clock to be corrected.
crf_data_length	Y	x"0030"	6 CRF Timestamps (as per definition) for a total length of 48 bytes.
Timestamp Interval	Y	x"00A0"	160 as events per definition.
CRF Data	N	gPTP Timestamps	6 timestamps, 64 bits each.

5.2.3 Operation

Using the Intel Triple Speed Ethernet Megafunction [36] to handle transmission, a finite state machine (FSM) was implemented. Figure 5.10 shows the transition between states, whereas Figures 5.11 through 5.14 details the process that occurs in each state.

Two variables are used in the process, namely `sel_bits` and `counter`. The `sel_bits` variable is used to select the bits from the `HEADER` signal (a signal containing all the constant data required to form an ETH-2 frame containing the fields given in Table 5.3) that are to be transferred that clock cycle. The `counter` variable is used during the `ST_WAIT` state, to delay for a set amount of clock cycles before moving back to the `ST_START` state.

It is important to note that, while not included in the diagrams for the sake of neatness, the reset signal is checked every clock cycle. When reset is triggered, the process shown in Figure 5.9 occurs.

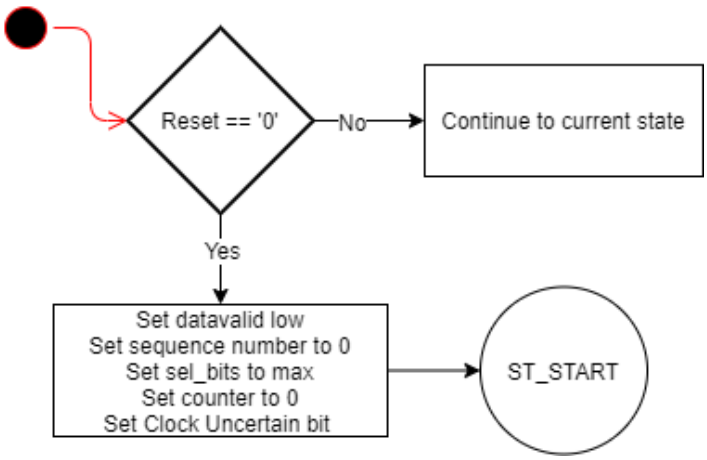


Figure 5.9: What occurs upon reset of the CRF Generator Module

As shown in Figure 5.10 below, the FSM consists of four states, namely `ST_START`, `ST_TRANSMIT`, `ST_END` and `ST_WAIT`. The FSM starts, runs through an initial configuration, and then moves into the first state, `ST_START`.

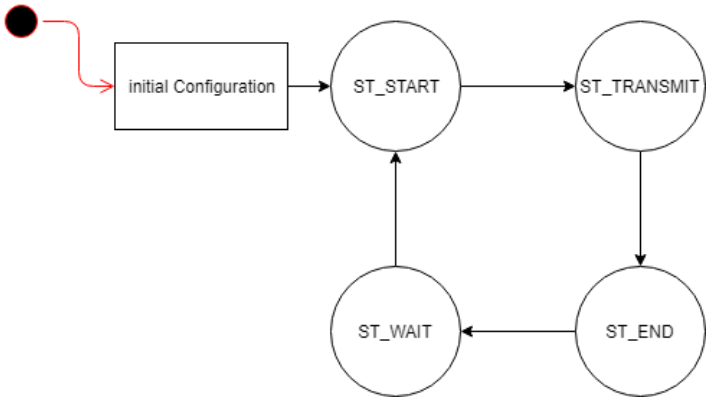


Figure 5.10: An overview of the interaction of states in the CRF Packet Generator.

`ST_START` can be seen in Figure 5.11 below. When the `TX_READY` signal is asserted by the MAC function, the module sets the data valid signal high, asserts the signal number in the `HEADER` signal, sets the start of packet signal high, asserts the first 32 bits for transmission on the `dataout` port, and decreases `sel_bits` for the next state the FSM moves to, `ST_TRANSMIT`.

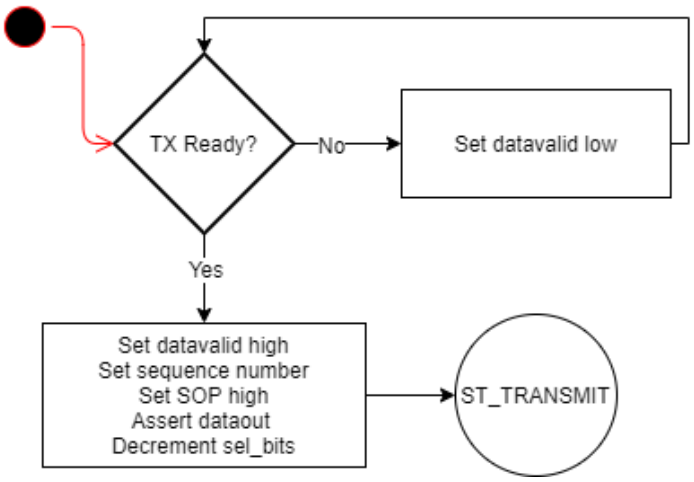


Figure 5.11: START state of the CRF Packet Generator.

In the `ST_TRANSMIT` state (shown in Figure 5.12 below), assuming the `TX_Ready` signal is asserted, the module asserts `datavalid` high, sets the start of packet signal low (as the first packet was transmitted in `ST_START`), asserts the relevant header data based on `sel_bits` on the output data port, and decrements `sel_bits`. This process is repeated until the last 32 bits are to be transmitted, which occurs when `sel_bits` reaches the value of 1. At this point, the FSM moves to state `ST_END`.

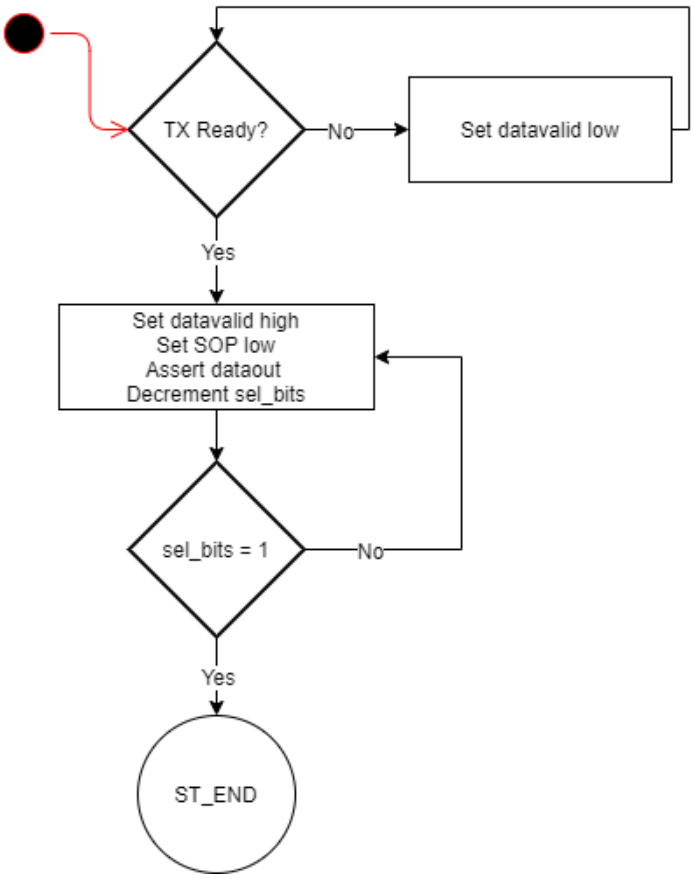


Figure 5.12: TRANSMIT state of the CRF Packet Generator.

Figure 5.13 below shows the `ST_END` state where the last set of data is transmitted. This is done by asserting the end of packet signal. `Datavalid` is set as high, and the final 32 bits are asserted on the output data port. The FSM now moves to `ST_WAIT`.

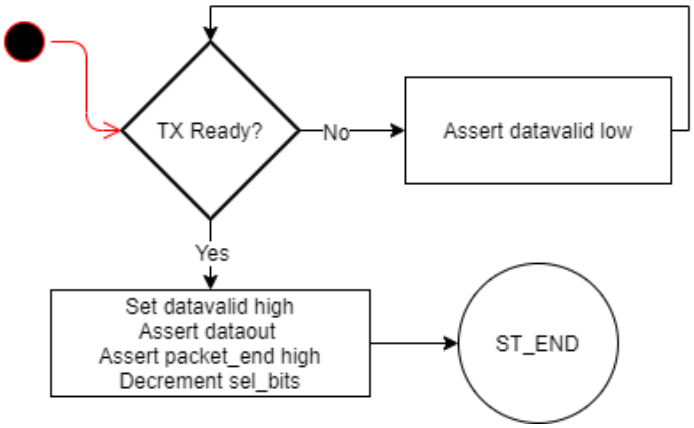


Figure 5.13: END state of the CRF Packet Generator.

In the final state, `ST_WAIT` (shown in Figure 5.14 below), datavalid is set low, the end of packet signal is set low, `sel_bits` is set to the maximum value, and a counter (which starts at 0) is incremented once each clock cycle. This process occurs until the counter value specified is reached, at which point the couter is reset to zero, the sequence number of the CRF packets is increased, and the FSM moves back to `ST_START`. This state is in place to ensure that CRF packets are only sent as often as they are needed, as per IEEE 1722.

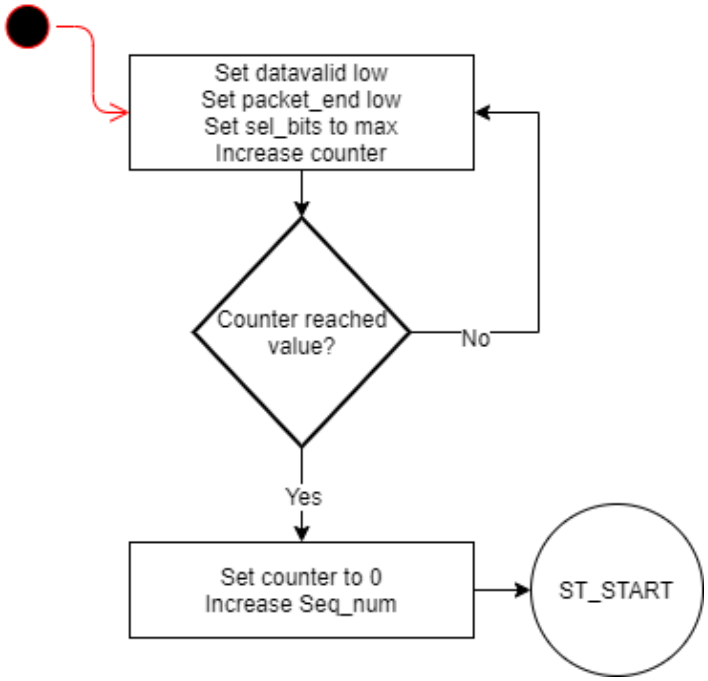


Figure 5.14: WAIT state of the CRF Packet Generator.

5.2.4 Summary

In this section the design and operation of the CRF Frame Generator, the HDL module responsible for formation and transmission of CRF AVTPDUs in Ethernet frames, was detailed. Experimentation detailing the verification of this module is detailed in Section 6.3.2.

5.3 The Ethernet Receiving Module

The Ethernet receiving module acts as a pre-processor for the AVTPDU Processor module (which is detailed further in Section 5.4). While the AVTPDU processor is responsible for processing the data received in order to be used by the listener, the Ethernet receiver module is

responsible for capturing frames and determining if the data being received is suitable for the application. This was designed in such a manner as to make it easy to adapt to receiving and processing other Ethernet Data, allowing for faster incorporation of other systems and traffic into the system, or enabling the modules developed for this project to be easily incorporated into preexisting systems. The Ethernet receiving module fulfils the functional requirements depicted by the highlighted block in Figure 5.15 below. The Ethernet receiver and AVTPDU processor modules were tested simultaneously, with experimentation for both found in Section 6.3.3.

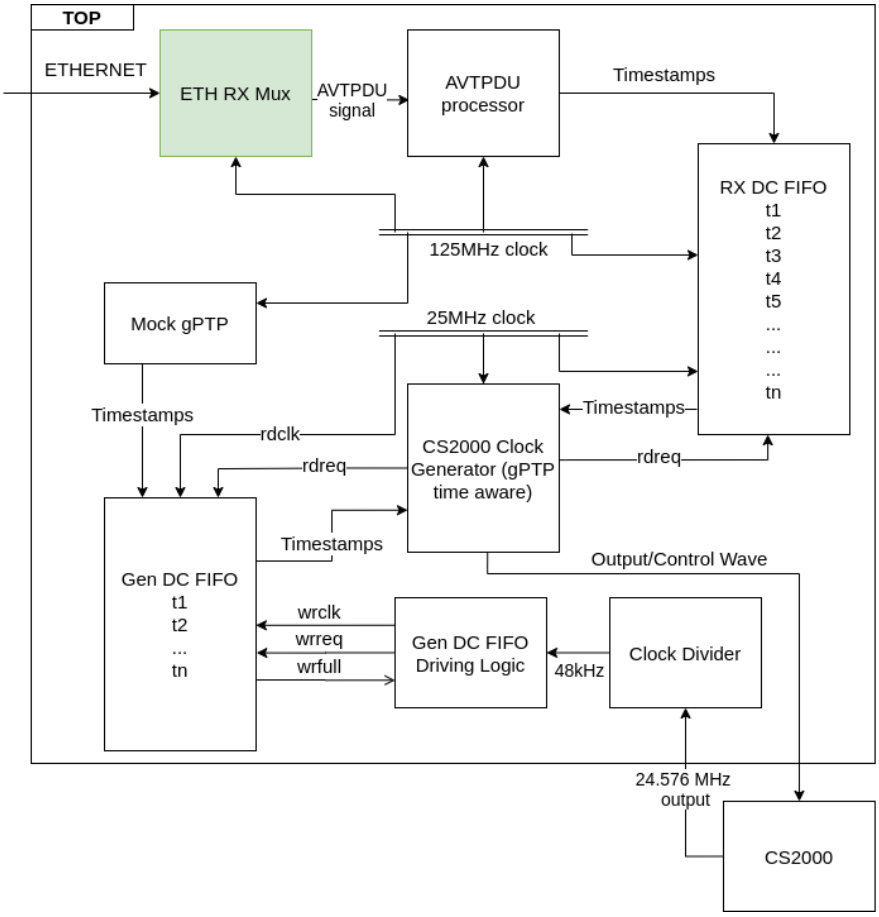


Figure 5.15: Ethernet receiver module application in the listener design.

5.3.1 Requirements

The Ethernet receiver module does not wholly fulfil any requirements, however it does enable the AVTPDU processor to fulfil the requirement R08 - Unpacking the formatted source timestamps received over Ethernet and extract the relevant information, as well as CRF-15 - the ability for a CRF listener to receive CRF AVTPDUs at the given rate.

5.3.2 Design Overview

The module makes use of the Intel Triple Speed Ethernet IP [36] to receive Ethernet frames. The block diagram of the Ethernet receiver module can be seen in Figure 5.16 below, with details of the input and output signals being given in Table 5.4.

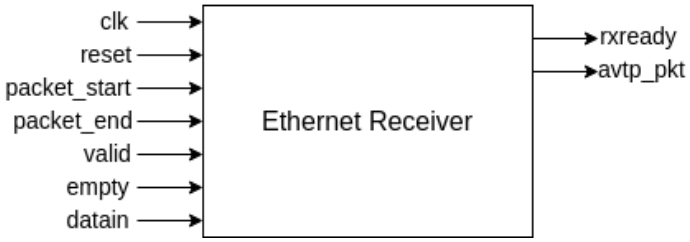


Figure 5.16: A block diagram of the I/O for the Ethernet receiver.

Table 5.4: Inputs and outputs for the Ethernet RX Module.

Port	**Direction**	**Bitwidth**	**Description**
clk	in	1	The input clock driving the module.
reset	in	1	A reset for the system.
packet_start	in	1	A start of packet signal from the MAC IP.
packet_end	in	1	An end of packet signal from the MAC IP.
valid	in	1	From the TSE IP, if the data is valid.
empty	in	2	Signal from TSE IP - “00” if the data is valid.
datain	in	32	The data in from the TSE IP.
rxready	out	1	Sent to the TSE IP to indicate if the receiving function is ready for data to be received.
avtp_pkt	out	1	An output from the module to indicate when timestamps are being processed.

5.3.3 Operation

Using the Intel Triple Speed Ethernet Megafunction [36] to handle receiving the packets, a finite state machine (FSM) was implemented. Figure 5.17 shows the transition between states, whereas Figures 5.18 through 5.21 details the process that occurs in each state.

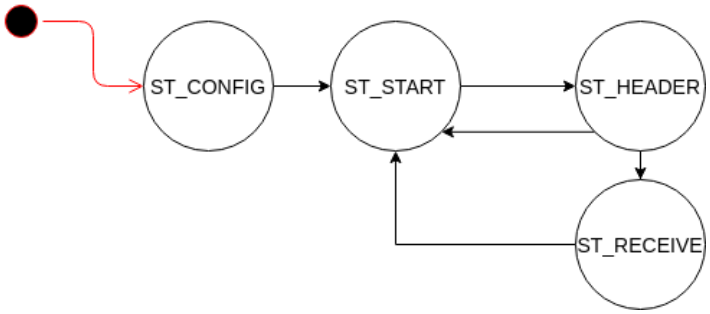


Figure 5.17: Overview of states and transitions for the Ethernet receiver.

The configuration state, shown in Figure 5.18 below, simply sets initial values. The `avtp_pkt` signal (which is used to notify the AVTPDU Processor of a valid CRF frame) is asserted low, and the `rx_ready` signal is sent to the TSE IP to indicate the device is ready to receive frames.

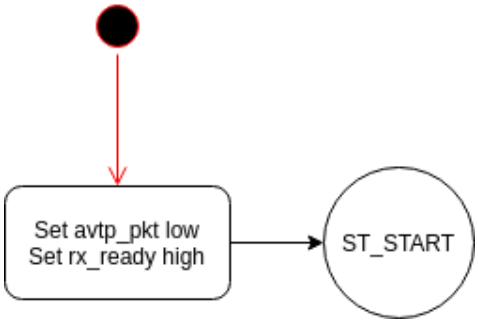


Figure 5.18: CONFIG state for the Ethernet receiver.

The start state, shown in Figure 5.19 below, is simply an idle state. In this state, the FSM waits for a start of packet signal from the TSE IP.

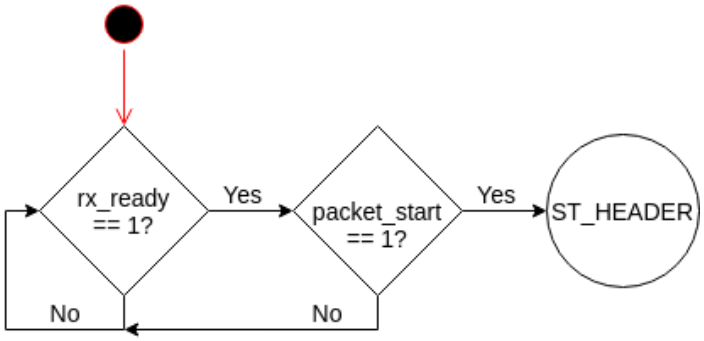


Figure 5.19: START state for the Ethernet receiver.

Upon receiving a valid start of packet signal, the FSM enters into the header state, which is shown in Figure 5.20 below. In this state, the Ethernet type is checked to determine if the packet received is of AVTPDU form. If so, the FSM moves on to the receive state and notifies the AVTPDU processor by asserting the `avtp_pkt` signal. This state can easily be expanded to cater for handling more types of packets by checking the sub-type included in the ETH-2 frame.

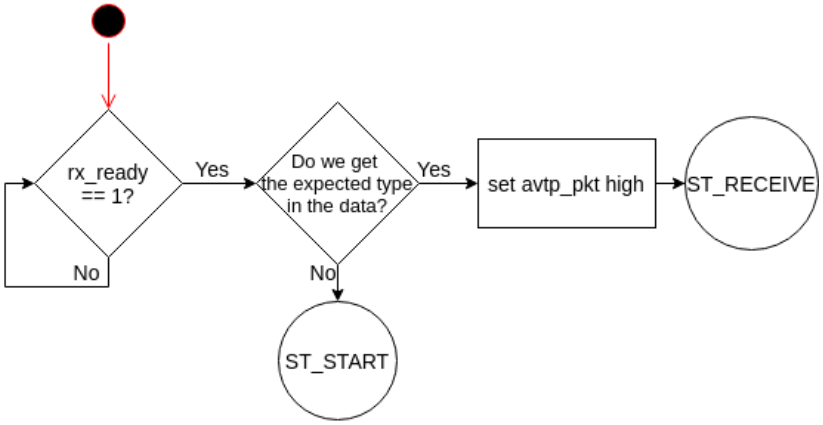


Figure 5.20: HEADER state for the Ethernet receiver.

In the final state for this module, `ST_RECEIVE`, shown in Figure 5.21 below, the FSM simply waits until an end of packet signal is received. When this occurs, the `avtp_pkt` signal is set low so that the AVTPDU processor knows to stop processing data from the Intel TSE IP, and the FSM transitions back to the start state where it waits for a new packet to be received.

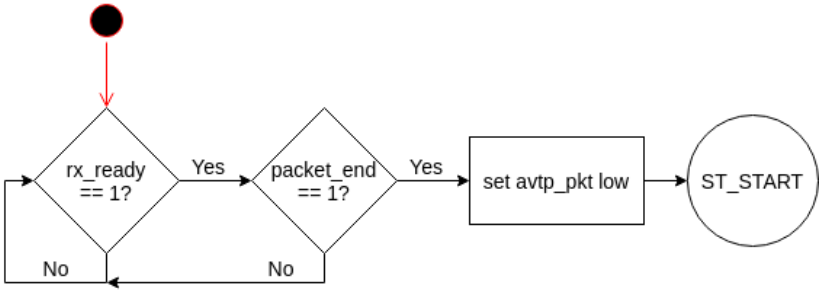


Figure 5.21: RECEIVE state for the Ethernet receiver.

5.3.4 Summary

The Ethernet Receiver module helps facilitate the processing of AVTPDU frames, and hence CRF packets, in fulfilment of R08 and CRF-15. It is designed to be easily expandable to cater

for the handling of other Ethernet traffic. Experimentation used to verify the operation of both the Ethernet receiver and AVTPDU processing modules can be found in Section 6.3.3.

5.4 The AVTPDU Processor

AVTP data units (AVTPDUs) consist of multiple formats, as discussed in Section 2.2.1. The AVTPDU processor module presented in this section ensures that only the CRF sub-type is accepted. The functional application of the AVTPDU processor module is shown in Figure 5.22. It waits upon a ready signal from the Ethernet receiver detailed in Section 5.3 and then pulls the data from the Intel Triple Speed Ethernet (TSE) Megafunction. Part of this information includes the transmitted timestamps, which are sent to a dual clock FIFO (highlighted in the figure below) which acts as a buffer before sending the timestamps to the CS2000 generator module (detailed in Section 5.6). Verification of the AVTPDU processor happened alongside the Ethernet receiver module, with experimentation for these modules found in Section 6.3.3.

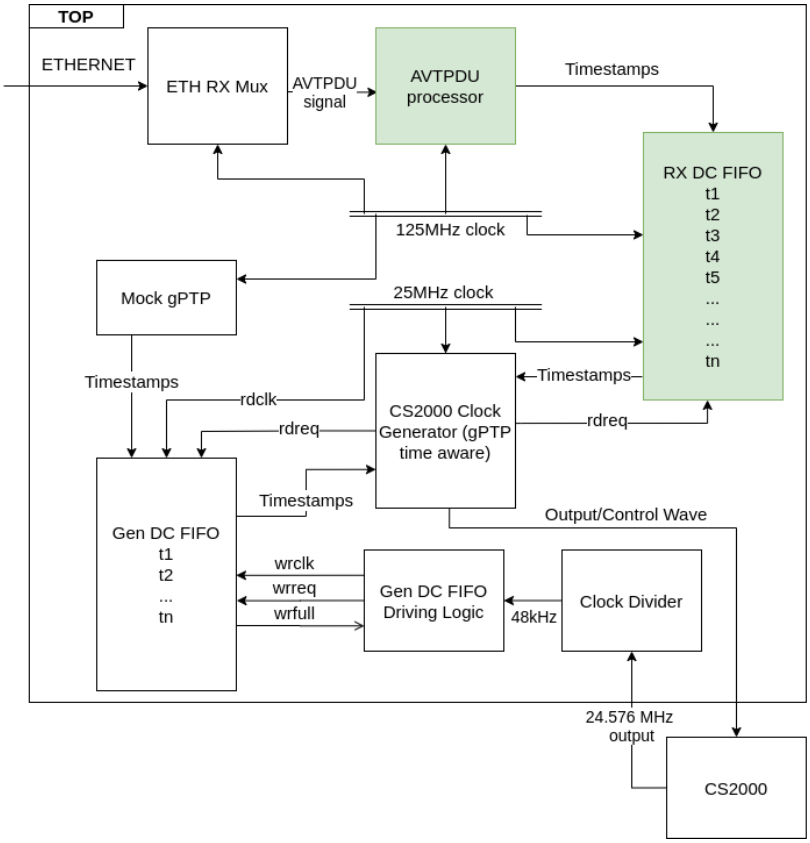


Figure 5.22: AVTPDU Processor Module application in the listener design.

5.4.1 Requirements

This module is used alongside the Ethernet receiver module to fulfil requirement R08 - Unpacking of the data received over Ethernet and extraction of relevant information, as well as requirement CRF-15, in that a listener is capable of receiving CRF AVTPDUs at the defined rates.

5.4.2 Design Overview

Figure 5.23 shows the block diagram for this module, and Table 5.5 elaborates upon the input and output signals.

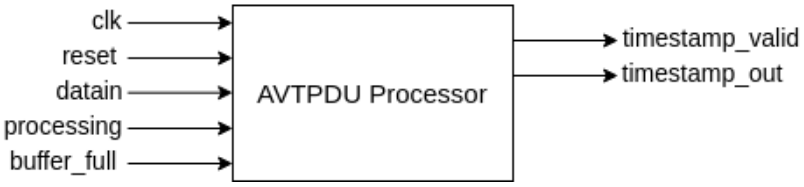


Figure 5.23: A block diagram highlighting IO of the AVTPDU Processor.

Table 5.5: Inputs and outputs for the AVTPDU Processor.

Port	**Direction**	**Bitwidth**	**Description**
clk	in	1	Input clock that drives the module.
reset	in	1	A reset signal for the system.
datain	in	32	Data from the TSE IP.
processing	in	1	A “valid” signal from the ETH RX, set high when working with an AVTPDU.
buffer_full	in	1	A signal from the RX Timestamp buffer indicating if it is full.
timestamp_valid	out	1	The write signal to the RX timestamp buffer.
timestamp_out	out	64	The timestamp out sent to the RX timestamp buffer.

5.4.3 Operation

This section details the operation of the AVTPDU Processor. The purpose of this processor is to correctly extract the information from the received Ethernet data from the Intel TSE

Megafunction. Some of these values include the following, shown in 5.6 below.

Table 5.6: Values extracted from the ETH data by the AVTPDU Processor.

Item	**Description**
MR	When the receiver detects a change in MR (media reset), any and all calculations should be recalculated, as the media clock has undergone a reset.
TU	When the TU (timestamp uncertain) flag is set, the receiver can ignore the frame and let the receiver clock freewheel.
sequence_num	The receiver listens for a few frames to determine the current sequence number. If a sequence number is received that is not an expected value, it is discarded and the frame ignored while the clock freewheels. If three frames with unexpected values are received, any and all calculations need to be performed.
type	The Type field gives the type of timestamp in the frame. In this application, the expected type value is x"04".
stream_id	A combination of the sender's MAC address and a unique ID.
pull	A multiplier for the base frequency. Set to 0 on the sender.
base_frequency	The base frequency of the clock to be recovered - 48 kHz.
CRF_data_length	How many timestamps are contained in a frame.
timestamp_interval	The amount of media clock events between each timestamp.

Due to the nature of the implementation (in that it is in a closed system, and data on the line is known and expected), the following has been done to simplify the implementation:

- `MR` is extracted but currently unused.
- `TU` is extracted but currently unused.
- `sequence_num` is extracted but currently unused.
- `type` is used to ensure the correct CRF sub-type is received.
- `stream_id` is extracted but unused, as only one talker exists on the network.
- `pull` and `base_frequency` are both extracted but left unused, as the clock to be recovered is known to be 48kHz.
- `CRF_data_length` is extracted to be used in the logic of the AVTPDU Processor - ensuring six timestamps are processed.

- `timestamp_interval` is ignored as it is known to be 160 for the single stream used in this implementation.

The overall operation of the module can be seen in Figure 5.24 below. The system starts in a wait state (`ST_WAIT`), moves to a configuration state (`ST_CONFIG`), and ends in a state where timestamps are processed and passed to the RX FIFO (`ST_TIMESTAMPS`, abbreviated as `ST_TStamps` in the figure).

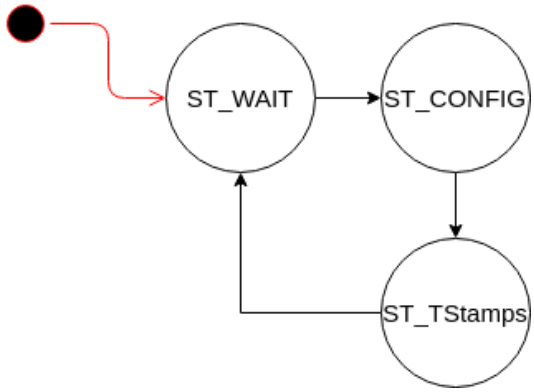


Figure 5.24: A flowchart of the states in the AVTPDU Processor.

The system starts in `ST_WAIT`, shown in Figure 5.25 below. This state starts by asserting `timestamp_valid` low, to complete the write cycle in `ST_TIMESTAMPS` (described later in this section). This state checks if the ETH RX module has received a AVTPDU. If so, it performs another check to ensure the AVTPDU is of the CRF sub-type. If either of those conditions fail, the module stays in this state. If both of these conditions pass, the initial set of data can be extracted from the `data_in` signal, and the module moves to the configuration state.

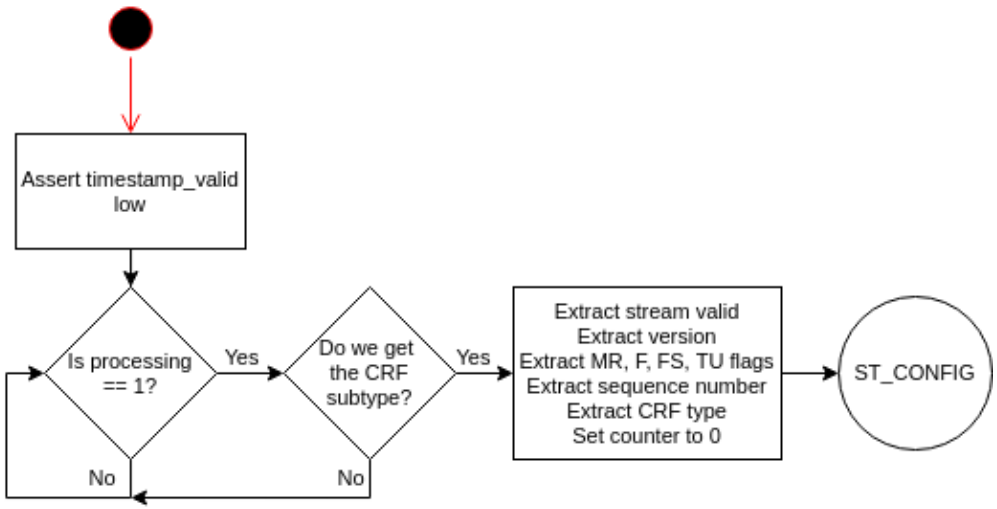


Figure 5.25: WAIT state of the AVTPDU Processor.

In the configuration state (shown in Figure 5.26 below), the module extracts a set of CRF configuration information that is transmitted by the talker and asserts some variables used in `ST_TIMESTAMPS` to ensure safe and correct extraction of the full 64-bit timestamp values. Notably, it sets `CRF_data_len`, which is used to ensure the correct number of timestamps are used. The module then moves to the `ST_TIMESTAMPS` state.

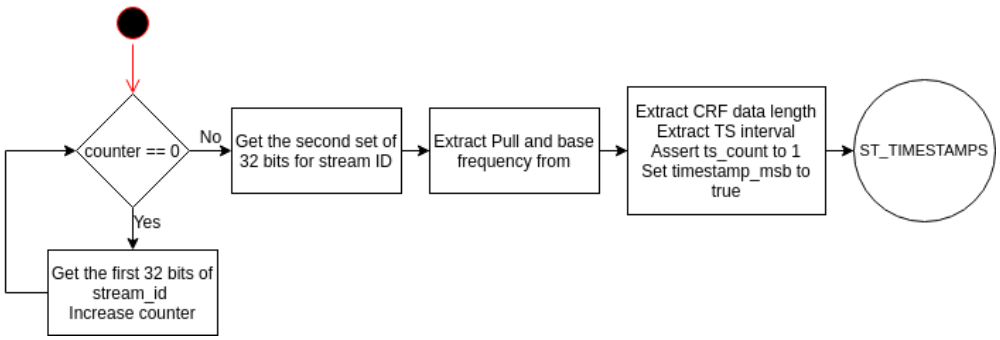


Figure 5.26: CONFIG state of the AVTPDU Processor.

In `ST_TIMESTAMPS`, the system combines two consecutive sets of 32 bits `data_in` to form the complete 64-bit timestamp. If a timestamp is fully formed, it is sent to the RX FIFO for further processing by the CS2000 Control Module, detailed in Section 5.6. Once six timestamps have been processed, the system moves back to the wait state to process the next incoming packet.

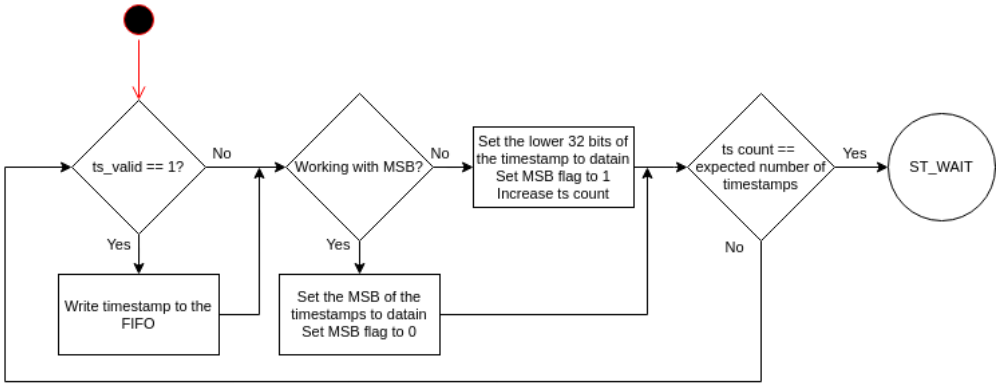


Figure 5.27: TIMESTAMP state of the AVTPDU Processor.

5.4.4 Summary

This section showed the design of the AVTPDU Processor, a module designed to extract information transmitted in CRF packets and prepare it for use by the module designed to correct the generated clock. The AVTPDU Processor module assists in fulfilling requirement R08 and CRF-15. Verification of the module is done in the same process as verification for the Ethernet receiver module, with experimentation detailed in Section 6.3.3.

5.5 CS2000 Generated Signals and Use

In this section, factors relating to the design of the functional blocks highlighted in Figure 5.28 below are discussed. In brief, the `CSGEN` module detailed in Section 5.6 produces a control wave which the CS2000 IC scales up to 24.576 MHz. This is fed into a clock divider, which scales it down to 48 KHz. This in turn is fed into a module which controls the write clock and write request signals to the DCFIFO (depending on the "write full" signal) that stores the timestamps of the locally generated media clock. These are the timestamps which are compared to the talker timestamps, and in turn used to calculate any adjustments to the output/control wave to the CS2000 IC in order to better align the phase of the generated clock on the listener to that of the source clock on the talker.

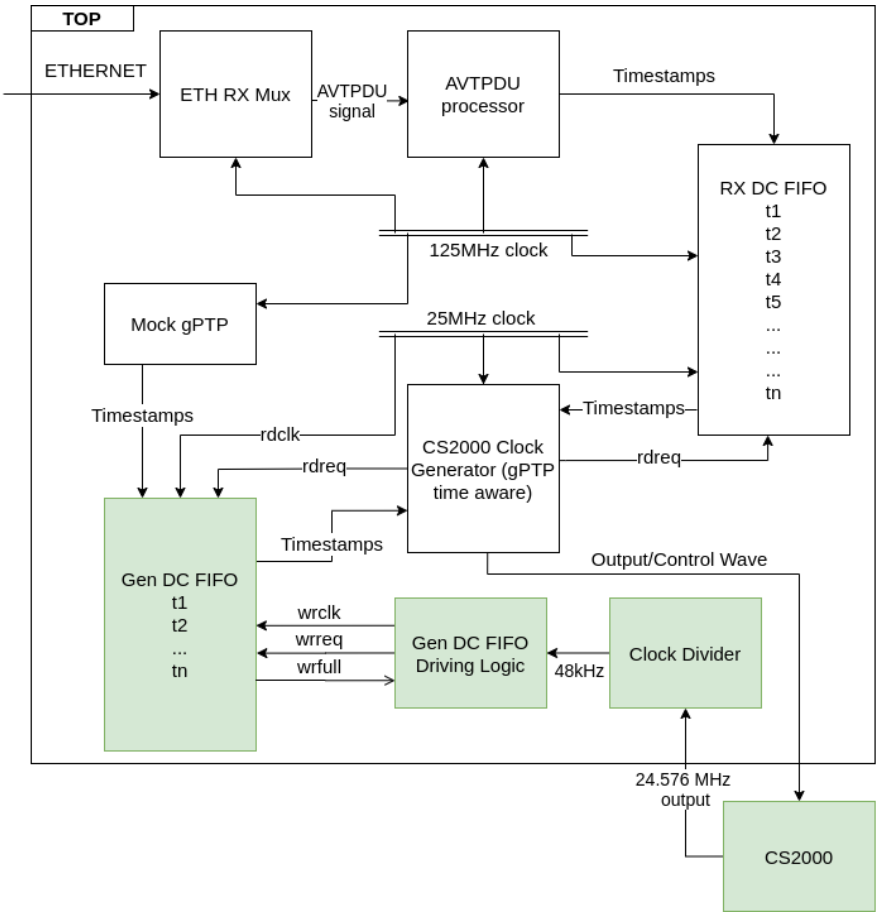


Figure 5.28: Modules that require use of the output signals from the CS2000.

5.5.1 The CS2000 Integrated Circuit

The CS2000-CP is an integrated circuit (IC) designed by Cirrus Technology [41]. It features a programmable phase-locked loop to enable clock generation from a reference clock.

This IC was provided by the industry partner in order to generate the clock signal to be corrected and synchronised against the talker clock.

The chip is connected to the Cyclone V FPGA. It takes an input signal and scales it, allowing control of the input signal (generated on the FPGA in this application) to directly affect the output. The circuit diagram for the IC and how it is connected to the FPGA can be found in Appendix B.1.

5.5.1.1 Configuration

Configuration of the CS2000 is done via a Python script, which on boot of the on-board microcontroller on the Mercury SA2+ development board. More details on configuration, as well as what registers on the CS2000 are configured, can be found in Appendix B.2.

These register values are set in order to configure the CS2000 chip to operate as desired. The device configuration is enabled through writing to registers 3 and 5. The lock signal from the phase-locked loop is assigned to the auxiliary pin and is made available on the development board through test point 1. The dynamic mode of operation (setting the output frequency through control of a digital PLL from an input signal) is set in register 5. Register 22 ensures that when there is no input from the control clock (`CLK_IN`), the output signal remains at its most recently adjusted value.

Register 6 sets the eight most significant bits of the first user-defined ratio between the input timing reference and the output clock. User-defined ratios are stored in 32 bit fixed-point format. In the used configuration, 20 bits are used for the integer component, and 12 for the fractional component. With the defined register values set, the user-defined ratio is equal to 24756. This results in a clock scaling of 1000, as the input timing reference is driven by a 24.576 MHz crystal oscillator.

The configuration set results in the operation shown in Figure 5.29. A 1 kHz wave is fed into the CS2000, and it is scaled to produce a 24.576 MHz output.

5.5.1.2 Operation

In order to understand the role of the CS2000 in the design, how it works must be briefly discussed. Figure 5.29 below shows the three signals of most concern to thr operation of the CS2000 after configuration.

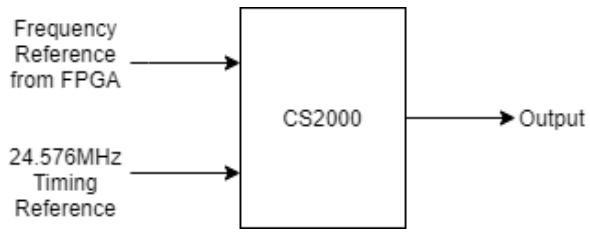


Figure 5.29: A conceptual view of the CS2000 to aid in explanation of its operation.

The 24.576 MHz timing reference is provided by a crystal oscillator on the development

board. The frequency reference from the FPGA controls the output frequency, which is fed back into the FPGA. The user-defined ratio described in the configuration section above along with the frequency reference and timing reference results in an output frequency which can be defined as shown in Equation 5.1 below.

$$Output = F_{ref} * UDR \tag{5.1}$$

where: $Output$ = The output of the CS2000 chip, fed back to the FPGA
F_{ref} = The frequency reference from the FPGA
UDR = The User Defined Ratio, configured to be 24576 in this project

5.5.2 The Clock Divider and Gen DCFIFO Modules

The clock divider highlighted in Figure 5.28 is a simple counter, used to divide down the 24.576 MHz into varying frequencies. Of interest in the application is division by 512, which produces a 48 kHz waveform. This waveform is considered to be the generated media clock signal for the listener, as well as operating as the write clock write request signal for the DCFIFO responsible for storing listener timestamps. The write request signal is dependant on the `wrfull` flag from the DCFIFO.

As the clock divider is a simple module, the simplest way to convey operation of the clock divider is through means of the module code, shown in Listing 5.1 below.

Listing 5.1: The clock divider module.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity aclkdiv is
port (
        clk : in std_logic;
        div2 : out std_logic;
        div4 : out std_logic;
        div8 : out std_logic;
        div256 : out std_logic;
        div512 : out std_logic);
end entity aclkdiv;

architecture rtl of aclkdiv is
begin
```

```
        process (clk)
                variable cnt : std_logic_vector(8 downto 0) := (others => '0');
        begin
                if rising_edge(clk) then
                        cnt := std_logic_vector(unsigned(cnt) + 1);
                        div512 <= cnt(8);
                        div256 <= cnt(7);
                        div8 <= cnt(2);
                        div4 <= cnt(1);
                        div2 <= cnt(0);
                end if;
        end process;
end architecture;
```

The "Gen DCFIFO" highlighted in Figure 5.28 also operates in a fairly simple manner. The block diagram of this module showing input and output signals is shown in Figure 5.30 below.

Figure 5.30: The block diagram for the generated timestamps buffer control module.

The signals shown in Figure 5.30 are straightforward, with `48kHz Clock` being the 48kHz signal in from the clock divider used to generate the write request signal to the FIFO, the `gen_fifo_wrfull` signal being the "write full" signal from the DCFIFO, and the `rx_ready` signal being the signal asserted by the Intel TSE IP once it has completed set up. This signal is also used to cater for the start up time of the system so that the timestamp buffer is not filled with timestamps which are not used. This results in a more efficient operation.

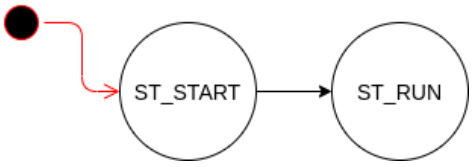


Figure 5.31: A flowchart of the states in the generated timestamps buffer control module.

As the output of this module is purely combinatorial, it is implemented with an and gate in the top-level module consisting of the clock, the `rx_ready` signal from the Intel TSE IP, and the inverse of the `fifo_full` signal from the FIFO as follows:

$$wr\_req = clock \wedge rx\_ready \wedge \neg fifo\_wr full$$

5.5.3 Summary

In this section information relating to the CS2000 and how it is used to generate a media clock on the listener was presented. Furthermore, information pertaining to the creation of timestamps to be used by the `CSGEN` module described in Section 5.6 to adjust the output of the CS2000 IC in order to align talker and listener clocks was also presented.

5.6 The CS2000 Control Module

The CS2000 module controls the waveform sent to the CS2000, an integrated circuit (IC) that results in the generation of the listener's media clock. The IC and derivation of the media clock, as well as the appropriate timestamps, is described in more detail in Section 5.5. This output/control wave is adjusted based on comparison of the source and generated timestamps in order to correctly align the listener clock with the talker clock. This module applies a few of the design aspects of a phase-locked loop, described in Section 2.4. In this section and others, this module is occasionally referred to as the `CSGEN` module, for "CS2000 control waveform generator". The functional blocks controlled by this module are highlighted in Figure 5.32.

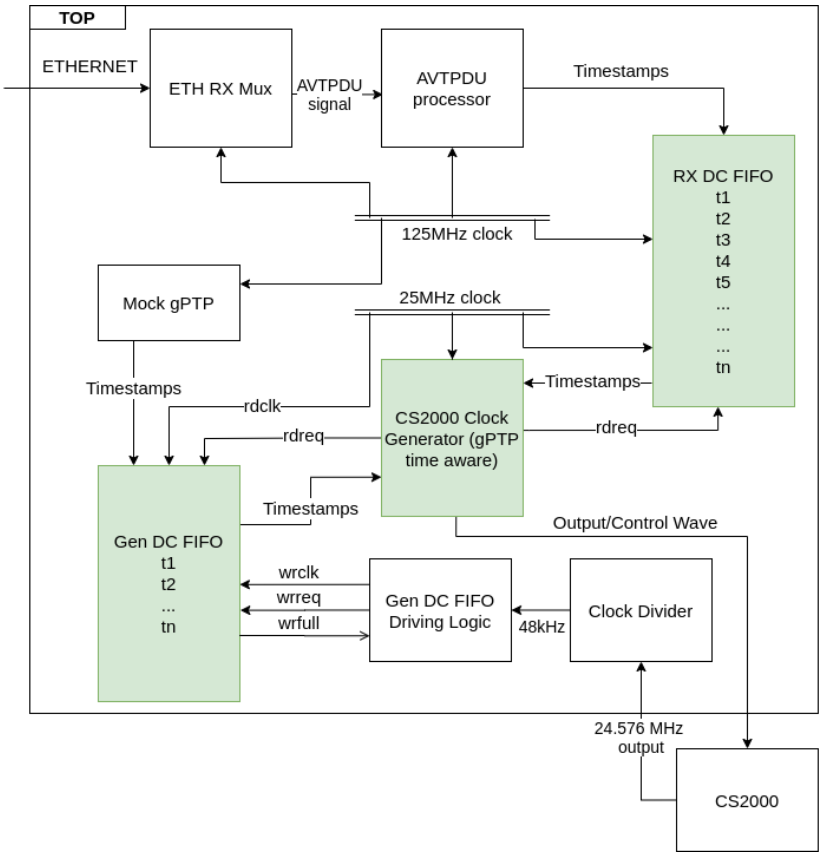


Figure 5.32: CS2000 control module application in the listener design.

The module controls reading from both DCFIFOs, one of which holds timestamps received from the talker, and the other holding timestamps generated by the listener (local) clock. It then compares these timestamps to make adjustments to the control waveform which will, in turn, align the timestamps more precisely. The conceptual operation of the module can be seen in Figure 5.33 below.

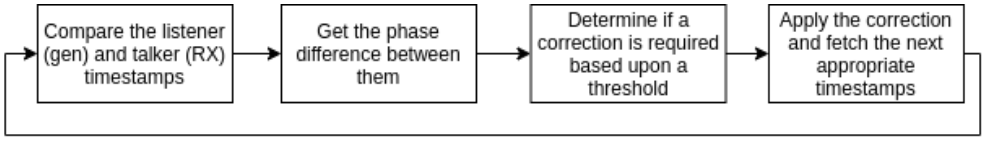


Figure 5.33: The conceptual operation of the CSGEN module.

5.6.1 Requirements

This module is used to fulfil the following requirements:

- R06 - Generate a media clock of 48kHz which can be adjusted in phase in order to match up to the media clock in R02
- R09 - Use the unpacked timestamps and compare them to the generated media clock's timestamps in order to determine a phase difference
- R10 - Use the phase difference in order to adjust the generated media clock to be in phase with the source media clock

Requirements R06 and R09 can be verified through an open-loop implementation, without feedback. Requirement R10 will require integration from other modules to create a closed-loop system.

5.6.2 Design Overview

Figure 5.34 shows a block diagram detailing the inputs and outputs for the module. These ports are further elaborated upon in Table 5.7.

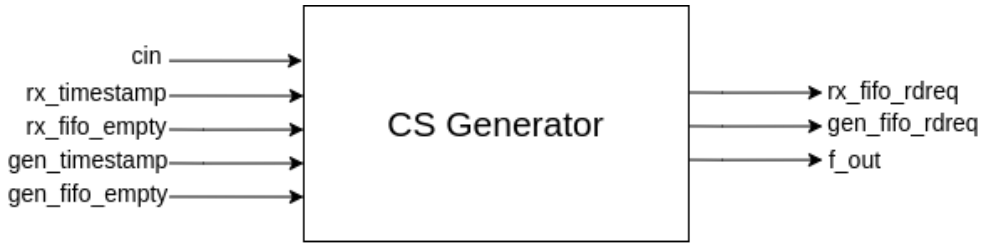


Figure 5.34: A block diagram highlighting IO of the CS2000 Control Module.

Table 5.7: Inputs and outputs for the CSGEN Module.

Port	Direction	Bitwidth	Description
cin	in	1	Input clock that drives the logic.
rx_timestamp	in	64	Timestamps from the RX buffer (received over Ethernet).
rx_fifo_empty	in	1	An empty flag, used to prevent bad read requests.
gen_timestamp	in	64	Timestamps from the local buffer (generated from the local "listener" clock).
gen_fifo_empty	in	1	An empty flag, used to prevent bad read requests.
rx_fifo_rdreq	out	1	A read request to get a new timestamp from the RX buffer.
gen_fifo_rdreq	out	1	A read request to get a new timestamp from the local buffer.
f_out	out	1	The output wave to the CS2000.

5.6.3 Modelling the Open Loop Behaviour of the CS2000

As the `CSGEN` module works with the CS2000 IC to form a closed-loop system, modelling of the open-loop system needs to be done in order to ensure correct operation once the correction algorithm is brought in.

5.6.3.1 Generating the Desired Output

In order to generate a 48 kHz wave output from the CS2000 processing chain (the output of the CS2000 divided down by 512 as described in Section 5.5), an input waveform needs to be fed into the CS2000 IC. The frequency for this input waveform is based on Equation 5.1 derived from the system described in Section 5.5. Substituting the scaling by 512 to achieve 48 kHz into Equation 5.1, the frequency of waveform to be fed into the CS2000 IC by the

CSGEN module can be determined, shown in Equation 5.2 below.

$$\begin{aligned} Output &= F_{ref} * UDR \\ \therefore F_{ref} &= Output/UDR \\ F_{ref} &= (48000 * 512)/24576 \\ F_{ref} &= 1000Hz \end{aligned} \tag{5.2}$$

Given that the CSGEN module runs at 25 MHz, to generate an output wave with a frequency of 1 kHz, a counter can be implemented to toggle the pin, toggling every $25 \times 10^6/1000/2 = 12500$ clock cycles. This is with a counter incrementing by one "step" each clock cycle. However, tweaks can be made to adjust this counter in order to enable greater control over the output waveform. For example, the counter's maximum value and the step can be multiplied by some value n, and as a result the value returned by the correction algorithm can make finer corrections (as the correction value is proportionally smaller than a standard step). Investigation and validation of this value is done in Section 6.3.4.

5.6.4 Operation

The CSGEN module operates in a similar fashion to a phase-locked loop. It determines a phase difference between the source and generated waveform, and adjusts the phase of the generated waveform in order to bring it closer in phase to the source waveform. The module consists of four states.

Every clock cycle, independent of the state of the module, the phase of the output waveform to the CS2000 module is determined, and the value of the waveform is toggled if required. This logic is performed through a simple counter, with three variables of consideration:

- `count_to`
  The value the internal counter counts to before toggling the state of the output waveform which controls the CS2000.
- `cnt_int`
  The current value of the counter. This value gets adjusted by the phase correction algorithm and the phase drift mitigation algorithm.
- `phase_step`
  The value by which `cnt_int` increases by each clock cycle. The combination of `phase_step` and `count_to` produces a 1kHz output to the CS2000.

Every clock cycle, `cnt_int` is increased by `phase_step`. If there is an error calculated in the `ST_CALCULATE` state, it is applied at this point, too. If the value of `cnt_int` is greater than or equal to the value of `count_to`, the output wave to the CS2000 toggles state, resulting in a 1 kHz waveform. The logic of operation for this process is shown in Figure 5.35 below.

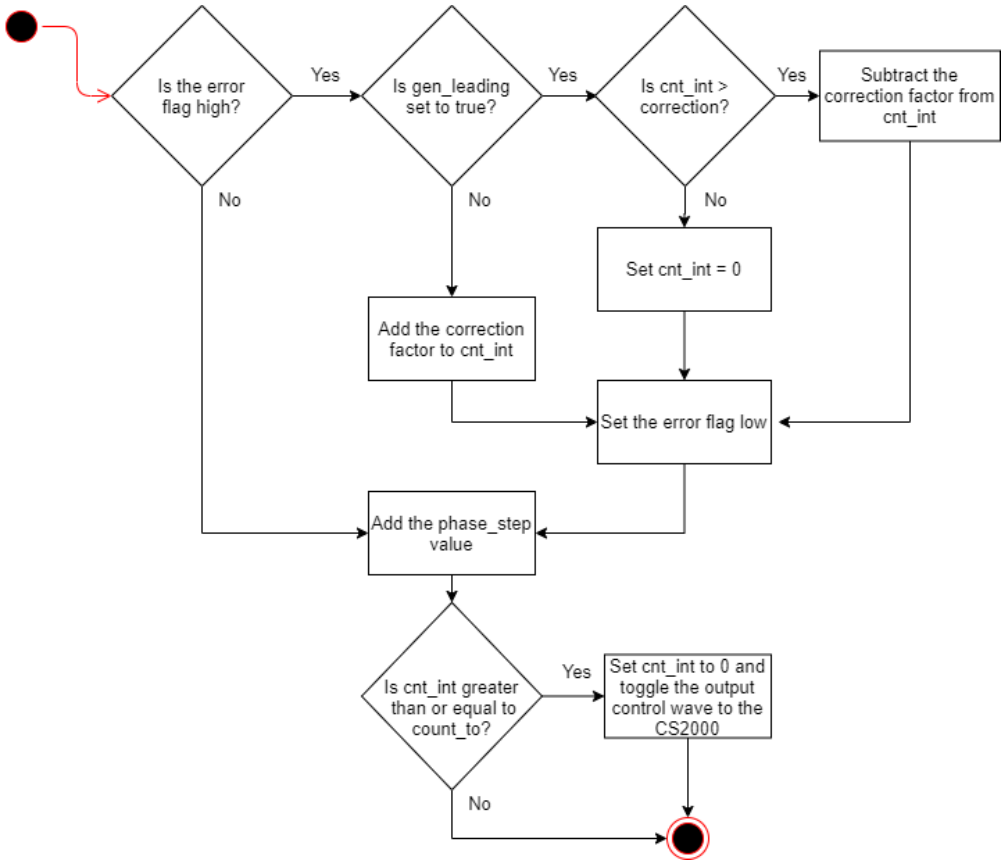


Figure 5.35: The logic controlling the phase of the control wave to the CS2000 IC.

Alongside the logical functions occurring every clock cycle, the process in this module also runs a finite state machine with four states, shown in Figure 5.36 below.

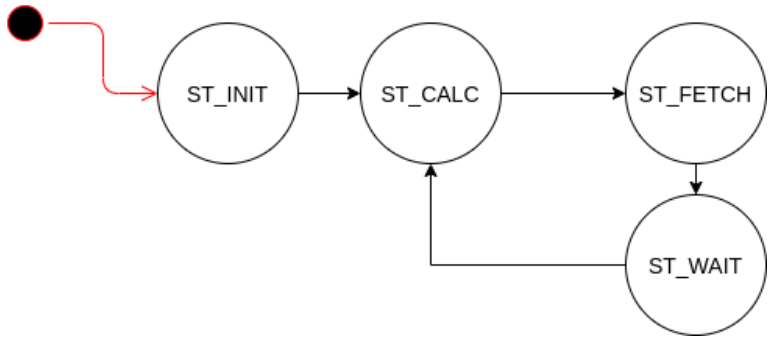


Figure 5.36: A flowchart of the states in the CSGEN module.

The module starts in `ST_INIT`, shown in Figure 5.37 below. The module waits in this state until there is data in both the DCFIFOS, which ensures that the operation of the correction algorithm can proceed. Once it is determined that there are timestamps in both DCFIFOs, the module performs a read request to these FIFOs and moves on to the next state, `ST_CALCULATE`.

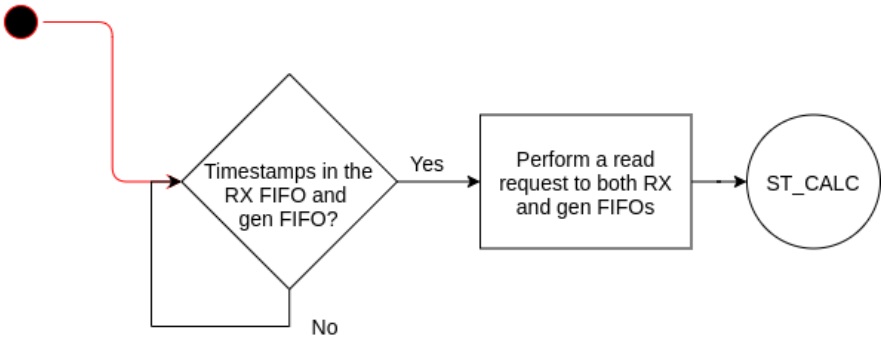


Figure 5.37: The INIT state of the CSGEN module.

In `ST_CALCULATE`, shown in Figure 5.38 below, the module processes the listener (RX) and talker (gen) timestamps to determine the difference between them, and which of the talker or listener is leading or lagging in relation to the other, based upon the timestamp values. By doing this, a reference for the difference in phase between the two can be determined. From this phase difference, the correction factor is determined. The correction factor is the amount by which to adjust `cnt_int`. In order to ensure that two valid timestamps are being compared to determine the correction factor and they are not out of range of each other, the difference between the two must be within a set threshold - namely half the period of the 48 kHz waveform, to cater for both phase leading and phase lagging scenarios. If the difference is within the set threshold, the correction value is calculated and a flag is set in order to indicate that a correction needs to take place. If the difference is not within the threshold,

the correction factor is set to zero and and the correction flag is set low. The module then moves on to `ST_FETCH`.

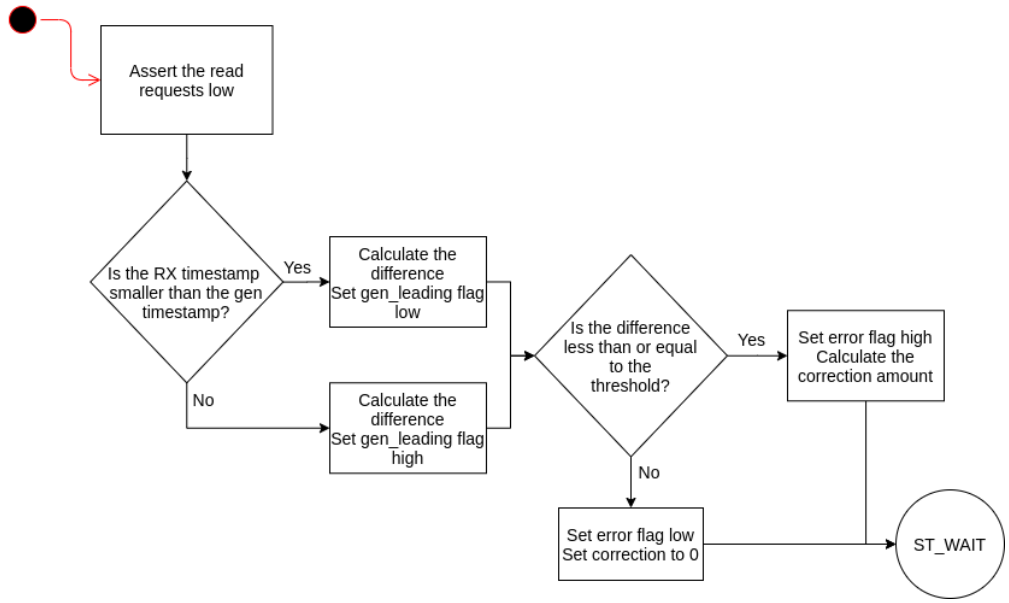


Figure 5.38: The CALCULATE state of the CSGEN module.

The role of the fetch state shown in Figure 5.39 below is to ensure that the correct timestamps are fetched from the correct buffer at the correct points. This is done by setting read request flags based upon values of the `gen_lagging` and `error` flags, with explanations shown in Table 5.8 below.

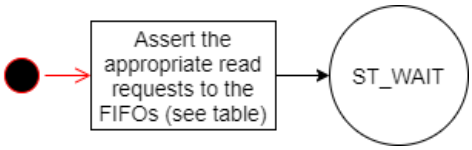


Figure 5.39: The FETCH state of the CSGEN module.

Shown in Figure 5.39, the `FETCH` state is responsible for asserting the correct read requests to the DCFIFOs containing the received timestamps from the talker, and the generated timestamps created on the listener. This is done by observing the `gen_leading` and `error` flags. Combinations of these flags, as well as what each combination implies, is described in Table 5.8 below.

Table 5.8: Process for ST_FETCH based upon the gen_leading and error flags.

gen_leading flag	**error flag**	**Explanation**	**Resulting action**
0	0	The current generated timestamp from the listener is leading the current received timestamp from the talker. However the difference between the two is outside the valid correction range, so a more recent talker timestamp must be fetched.	Assert the rx_fifo read request high
0	1	The current generated timestamp from the listener is leading the current received timestamp from the talker. An error was found, and a correction applied. These timestamps can both be considered processed, and a new talker and listener timestamp must be fetched.	Assert both the gen_fifo read request and rx_fifo read request high
1	0	The current generated timestamp from the listener is lagging behind the current received timestamp from the talker. However the difference between the two is outside the valid correction range, so a more recent talker timestamp must be fetched.	Assert the gen_fifo read request high
1	1	The current generated timestamp from the listener is lagging behind the current received timestamp from the talker. An error was found, and a correction applied. These timestamps can both be considered processed, and a new talker and listener timestamp must be fetched.	Assert both the gen_fifo read request and rx_fifo read request high

Once the correct read request signals have been asserted, the module moves into `ST_WAIT`, as shown in Figure 5.40 below. This module simply serves the purpose of asserting both read request signals low, and allowing time for the timestamp signals from the DCFIFOs to be asserted to the `CSGEN` module.

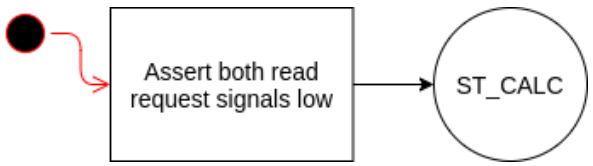


Figure 5.40: The WAIT state of the CSGEN module.

5.6.5 Summary

This section presented the design for the `CSGEN` module. It operates in a similar fashion to a phase-locked loop. It does so by determining the phase difference between the talker and listener media clocks, stored in two DCFIFOs to offset network delay, and updates the phase of the listener media clock by applying a correction value which is calculated based on the phase difference. This enabled the synchronisation of talker and listener media clocks.

5.7 Summary of the Chapter

This chapter presented, in detail, the various design aspects of the system. It followed along the path of data flow, with the exception of Section 5.5 (the CS2000 IC and subsequent logic), which was presented before Section 5.6 in order to provide context to the operation of the CSGEN module. The requirements described in Section 1.3 can be related to the modules presented as shown in Table 5.9 on the following page. From this table it can be seen that consideration for all the requirements have been made in the design. In Chapter 6 the operation of the designs presented in this chapter will be verified in order to ensure that the requirements are met.

Table 5.9: The requirements and where they are designed to be met.

Requirement ID	**Description**	**Module**	**Section reference**
R01	A shared timing reference between both talker and listener	gPTP	5.1
R02	Create a media clock of 48 kHz to use as a source clock	gPTP	5.1
R03	Generate timestamps using the timing reference in R01 and source media clock in R02	gPTP	5.1
R04	Package the generated source timestamps in a format suitable for transmission over Ethernet, as described by IEEE 1722	CRFFrameGen	5.2
R05	Transmit the packaged timestamps over Gigabit Ethernet in a way that makes it possible to observe them over the wire through an application such as Wireshark	CRFFrameGen	5.2
R06	Generate a media clock of 48 kHz which can be adjusted in phase in order to match up to the media clock in R02	CSGEN	5.6
R07	Generate timestamps of the generated media clock in R06 using the shared timing reference in R01	GEN DCFIFO Driver	5.5
R08	Unpack the formatted source timestamps received over Ethernet and extract the relevant information	ETHRX & AVTPDU Processor	5.3 & 5.4
R09	Use the unpacked timestamps and compare them to the generated media clock's timestamps in order to determine a phase difference	CSGEN	5.6
R10	Use the phase difference in order to adjust the generated media clock to be in phase with the source media clock	CSGEN	5.6
R11	IEEE1722 standard and requirements should be implemented as much as reasonably possible	Various	5.1, 5.2, 5.3 & 5.4

Chapter 6

Experiments

This chapter details the experimentation and verification testing performed to ensure the accuracy and correctness of the modules and hence the implemented system.

The contents of this chapter can be considered as two aspects of the development process: experimentation to discover new methods and test hypotheses, and validation to ensure correct and expected behaviours of implemented methods. The design and implementation of the system consists of items specified by relevant standards, as well as aspects which needed to be determined through experimentation - all of which has been described in Chapters 4 and 5.

The chapter starts by describing the experimental set up used in Section 6.1. The equipment and tools used are listed, with further elaborations where required. Section 6.2 details a Python simulation developed in order to model and monitor the behaviours of modules. This section also presents a form of the implemented phase correction algorithm. Section 6.3 describes the testing done to verify each module designed in hardware. Section 6.4 describes the testing to integrate these modules in accordance with the design methodology described in Chapter 3. Finally, in Section 6.5, the system is tested under various conditions to ensure correct operation through longer durations, different environmental conditions, and changing network conditions.

6.1 Experimental Set Up

This section outlines the experimental set up and equipment used in order to test hypotheses and to validate the operation and performance of the implemented design.

A diagram of the hardware setup is shown in Figure 6.1 below. A single FPGA was used, though the design is conceptually split into two parts - a talker, which transmits CRF Frames, and a listener, which receives them. Connected to the FPGA is an oscilloscope and a small breakout board with additional buttons for triggering processes. Connected to the Ethernet ports on the FPGA is a router. This is done so that the frames transmitted by the Talker can be monitored by separate software on a computer. The oscilloscope is also connected to the lab PC through USB for recording the data.

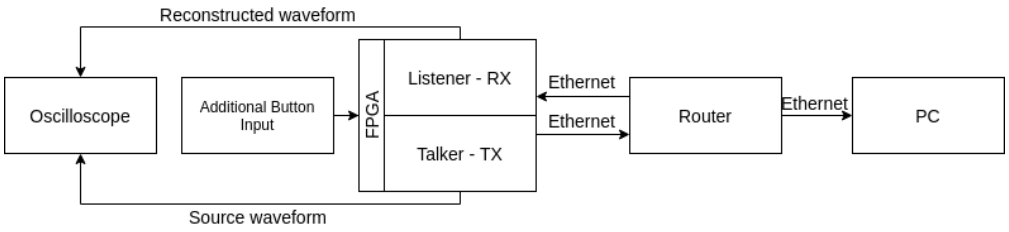


Figure 6.1: A diagram showing how lab equipment is connected.

6.1.1 Hardware and Software Used

Due to the nature of the project and the field, various equipment is needed. Powerful software exists to assist in the testing procedure. Hardware used in the experimentation includes:

- The Mercury SA2+ board and suitable power supply
- The CS2000 breakout board, attached to the Mercury board
- An Altera USB Blaster and mini USB cable to program the FPGA
- An SD card, containing the boot system and CS2000 configuration script
- Various Ethernet cables
- A suitable lab PC

Software included in the experimentation includes:

- Quartus Prime Lite Edition 18.1 - For developing and testing the HDL running on the FPGA
  - Including in-built SignalTap, a logic analyser instantiated in the fabric of the FPGA
- PyCharm 2020.2 - For developing Python simulations

- WireShark - For inspecting and capturing packets on the Ethernet Connection
- BitTwist - For re-transmitting known captured packets at known rates
- Keysight BenchVue - For recording data from the oscilloscope

In order to further facilitate the testing and enable additional tests, the following hardware was also used:

- TP-Link AC1200 Gigabit Router
- Keysight EDUX1002G Oscilloscope
- Additional breakout board for buttons as external input to FPGA
- A hairdryer, used to raise the ambient temperature surrounding the breakout board
- The temperature probe on the UNI-T UT30C multimeter

Owing to widespread practical implications of the COVID-19 pandemic, the machine used for development varied over the course of the research. However, final stages of development were done on a system with the following specifications being relevant to FPGA toolchains:

- AMD Ryzen 5 3600
- 16GB DDR4 Memory at 3600MHz
- Ubuntu 20.04 or Windows 10
- Sabrent 512GB NVMe PCI-E 3.0 SSD

A picture of the original lab setup could not be taken due to changes in working conditions, but the later lab setup is shown in Figure 6.2 below. The router is obscured behind the Samsung screen behind the oscilloscope.

Figure 6.2: A photograph of the lab setup.

6.2 Python Simulation

A Python simulation was developed to replicate the hardware and test the effectiveness of any recovery algorithms that might be developed. Development and testing in Python is considerably faster than development and testing on an FPGA. The simulation was not implemented to comprehensively test the algorithm ultimately implemented in hardware, but rather to act as a means of rapid prototyping a simpler implementation of that algorithm. This was done primarily to inform later design decisions. The findings of the experimentation which informed the design decisions for the hardware can be found in Section 7.1.

In order to mimic the hardware that would need to be developed in HDL, an object-oriented programming (OOP) design was used, with each circuit in hardware equating to an equivalent object. The exception is the gPTP implementation, which is generated by the value in an iterator (a for loop).

A for loop - with each iteration representing one nanosecond - was chosen above a multi-threaded implementation primarily for reasons of simplicity. Threading in Python (an interpreted, not compiled language) is handled differently than one might expect. The interpreter passes the global interpreter lock (GIL) between threads, meaning only one thread executes at any point, resulting in what is essentially a sequentially executed program. This can be bypassed using a multi-processor approach, but this approach increases complexity exponentially. Validation and verification - not to mention debugging - becomes increasingly

challenging. To ensure integrity of the data provided by the simulation, multiple locks and mutexes would be required. As a result, the decision was made to create a single-threaded application. A single threaded program is considerably easier to debug, and the desired data from the simulation easier to obtain and process.

The simulation produces a comprehensive log file, which proved beneficial in the debugging and verification of the operation of the VHDL modules modelled. This log file recorded the timestamp and defined variables every time specified events occurred. This enabled an easy means of verifying the mathematics and following the logic of the correction algorithm implemented to ensure it performed as expected. The data from these log files was used to produce various graphs to better visualise performance and results.

Only the circuits relevant to the clock correction and synchronisation were considered for addition in the Python simulation. After a few iterations, a simple design was completed. A link to the code can be found in Appendix C.

6.2.1 Nuances of Simulating Hardware in Software

Before getting into the details of the experiments conducted, it is important to highlight the difference between these experiments and the implementation of the designs in hardware. Processing in software is distinctly different than processing in hardware. These differences are exaggerated by the fact that, while hardware can usually run in parallel, software - by default - runs sequentially. These items of significance that needed to be taken into consideration can be summarised as follows:

1. Conceptual differences
   Conceptually, software can be thought of as a description to a problem. HDL, on the other hand, is a descriptor of a circuit that, in turn, will solve the problem. The differences between the two are distinct enough that considerable thought must go into how the two implementations complement and contrast each other.

2. Sequential processing
   Logic implemented in software is usually processed sequentially (one line of code executed at a time) - whereas in hardware, multiple signals can be changed in a single clock cycle. To cater for these differences in design approach, the Python simulation iterates over a time period in nanoseconds. This is done by calling clocked devices at the relevant clock rates and unclocked devices (or devices with changing clock rates) each iteration, to see if any updates need to be made.

3. Clock rates

Clock rates in the simulation are managed through the period of the clock to be implemented. An example of how clocks are compared and managed is as follows:

```
if int(current_time % int(self.output_period / 2)) == 0:
    # Perform task here
```

4. Floating point values
   The period of the desired media clock frequency output - 48 kHz - is an irrational number. In order to ensure accuracy when generating timestamps, floats are used before converting to an integer for comparison. This is thought to be the best matching approach to how the hardware will implement timing measurements.

5. Multiple Threads
   A multi-threaded simulation may have made some of the above nuances easier to handle - such as the different clock rates and sequential processing issues. A multi-threaded simulation may also have taken less time to run (for reasons such as only running methods when they were required, less checks in place to see if a method should be run at that time stamp, and making use of multiple threads on hardware), but it was decided that the time cost and design concerns of a multi-threaded implementation outweighed the benefits. Issues such as race conditions and thread locks would make validation and verification of the simulation more difficult to achieve. To this end the method described above - of using a single for loop to iterate over nanoseconds in the simulation - was decided upon.

6.2.2 Simulation Overview

Before fulling delving into creating a simulation, a "proof of concept script" was generated to prove that the logic could be tested in software. The proof of concept script is a simple script used to implement the mathematics of a potential correction algorithm. It does not cater for nuances of hardware, offsets between the oscillators, or anything of the like. The script is available at the GitHub link given in Appendix C.

Once the ability to simulate the implementation in Python was confirmed, work on the design of the simulation began. The simulation itself was designed to mimic the functionality of the following modules, which can be seen in Figures 4.2 and 4.3:

1. The gPTP Module
   As described in the design chapter, Section 5.1, a single, shared time measurement needs to be known between both talker and listener modules in order to perform corrections to the generated clock.

2. The Talker Clock
   This is the talker media clock. It is the source clock to which the phase of the listener clock is meant to be aligned.

3. The CS2000 Driver Module
   This module drives the output wave to the CS2000 and performs any corrections to that wave to align the phase and frequency of the source and generated waveforms.

4. The CS2000 and Clock Divider
   The CS2000 sits off the fabric of the FPGA, but it's logic still needs to be replicated in order to fully test correction algorithms that are developed. The output waveform of the CS2000 is also not the waveform that's used as as the generated clock - this output of the CS2000 is first divided down by 512 to produce the generated clock.

6.2.3 Simulation Modules and Methods

A conceptual overview of the system can be seen in Figure 6.3, with an explanation of the modules and the methods used following the figure. In order to save on execution time and ensure consistency between simulations, the source (talker) timestamps were created outside of the simulation in order to validate their correctness and speed up processing time. This entailed generating one second's worth of nanosecond timestamps for a 48kHz clock, and storing them in a look up table to be used in the simulation, rather than performing the calculations to see if a given nanosecond is a modulo of the 48kHz talker media clock. This look up table is referred to in Figure 6.3 as the "Data File".

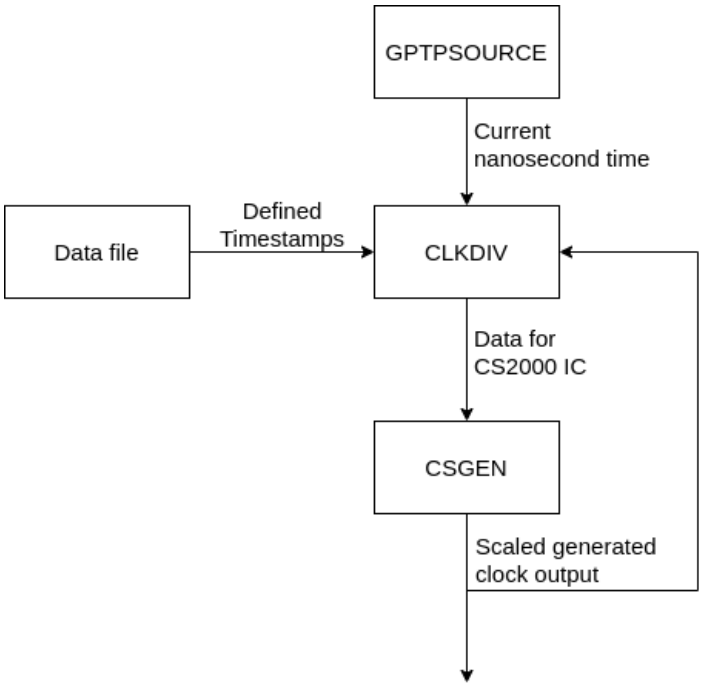


Figure 6.3: Conceptual view of the Python simulation.

The modules implemented are as follows:

1. GPTPTSOURCE
   This is the primary module and it is responsible for instantiating other modules. Its primary function is to iterate over a set time, where each increment in iterator corresponds to an increment in nanoseconds. A few additional methods are available for storing logging results and plotting waveforms. The module contains the following methods:
   (a) `run(self)`
   Runs the simulation.
   (b) `save_log_file(self, log_fields)`
   Writes the given log fields to a file.
   (c) `draw_waveforms(self, start, duration)`
   Draws the talker and receiver output waveforms superimposed for comparison.
   (d) `draw_phase(self, start, duration)`
   Draws the phase difference between the talker and listener waveform in order to visualise how it changes over time.
2. CSGEN

This module is responsible for generating the waveform that drives the CS2000 IC. It is also where the correction algorithm is applied. The module contains the following methods:

(a) `ocw_control(self, log, cs2000)`
Mimics the wave that would be sent to the CS2000, toggling a variable to high or low depending on the current nanosecond time.

(b) `correction_algorithm(self, gptp_time)`
Runs the correction algorithm.

3. CLKDIV
This module handles multiplication of the waveform generated by the CSGEN module, as well as the scaling of the output of the CS2000 to get it back to the desired clock rate. The module contains the following methods:

(a) `trigger_cs2000(self, gptp_time)`
Determines the output period (and hence frequency) of the mimicked CS2000 module.

(b) `generate_clock(self, gptp_time)`
Compares current time and the period of the output of the CS2000 module after scaling to determine what the status (rising edge, falling edge, high or low) of the generated clock signal is.

6.2.4 How the Simulation Operates

This section discusses how the simulation operates. As the CSGEN module is meant to simulate hardware that runs at 25MHz, it is only called when the gPTP time is a multiple of 40 (the period of a 25MHz clock in nanoseconds). This is shown in Listing 6.1:

Listing 6.1: The primary loop of the Python simulation.

```
for i in range(0, self.runtime):  # i is the time in nanoseconds
    # Circuits that need to run constantly or at unknown intervals
    self.cs2000.generate_clock(i)
    # Circuits that run on 25MHz
    if i % 40 == 0:
        self.csgen.ocw_control(i)
        self.csgen.correction_algorithm(i)
```

- First, the CS2000 `generate_clock` method is called. This does a check to see if the generated media clock needs to be toggled or remain the same by determining whether

or not the gPTP time is a modulo of half the desired output frequency of the generated clock, as set by the frequency of the waveform controlling the CS2000 module.

- Every 40 nanoseconds, `ocw_control` and `correction_algorithm` are run
  - `ocw_control` sets the input to the CS2000 module based on a counter. When a specific value in the counter is reached, the wave driving the CS2000 is toggled. If it is a rising edge, the method `trigger_cs2000` is called.
  - `correction_algorithm` calls a separately defined method which uses the source timestamps and the timestamps generated by the `CS2000_module` to update the value that is counted to in the `ocw_control` module. This is the core mechanism by which the clocks are aligned.
- The `trigger_cs2000` method takes the current gPTP time and compares it to the last gPTP time it was triggered. It uses this to get the frequency of the waveform that is driving it. This frequency is then scaled up and divided down to get the period of the output waveform, which is the period that the `generate_clock` method uses to see if the generated media clock needs to be toggled at any given gPTP time.

6.2.5 Testing the Correction Algorithm

The media clock correction algorithm was tested and tweaked in the Python simulation using the iterative Spiral model described in Section 3.

The algorithm starts by getting the difference between the two timestamps. It then determines which bounds this difference is in, and applies a correction to the numerically controlled oscillator appropriately.

Figure 6.4 below shows ranges described for the difference determined by subtracting the listener timestamp from the talker timestamp. The x-axis indicates the magnitude of the difference, with a negative difference indicating that the current listener timestamp is further ahead in time than the current talker timestamp, and vice versa.

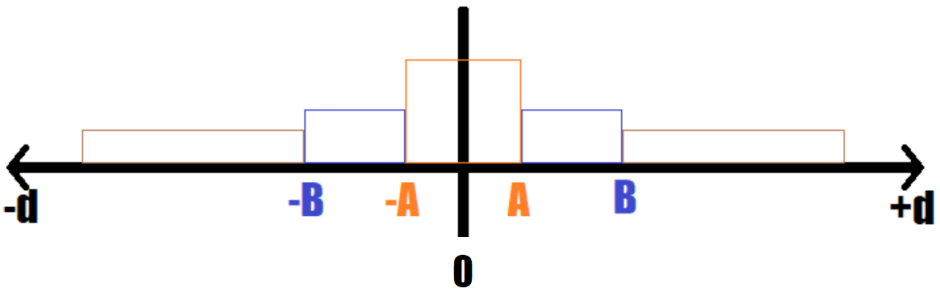


Figure 6.4: The range of differences used to determine the behaviour of the algorithm.

Range `[-A, A]` indicates the range within which no correction is needed, or where the difference is so small the correction algorithm is unable to make a change. Range `(A, B]` and `[-B, A)` indicates a range where a correction needs to be applied. A difference with a magnitude greater than a value of `B` indicate that the two timestamps being compared are outside of the range of applicability. This is to say a subsequent timestamp (from either the talker or listener, depending on whether the difference is positive or negative) is needed to measure a difference. In this scenario, the value of B equates to half the period of the 48kHz waveform. If a difference is greater than half the period of the waveform, unrelated timestamps are being compared. These ranges and how they relate to the implementation are further described in the design of the `CSGEN` module in Section 5.6, Table 5.8.

Using the information developed by subtracting the two timestamps, it can determined by how much the phase of the two timestamps differ at any point in time. This phase difference can be fed into an algorithm which will adjust the current `cnt_int` value (described in Section 5.6) in order to align the phase of the generated clock on the listener. How this algorithm is applied will depend on other values of parameters such as `r_step` and `count_to`, which will be explored experimentally, as described in Section 6.3.4 and 6.4.3.

The results of this experimentation as well as a discussion on these results can be found in Section 7.1.

6.3 Module Testing

This section will iterate over each of the modules detailed in Chapter 5 and validate that requirements are met and the modules operate correctly.

6.3.1 The gPTP Module

In order to validate performance of the gPTP module detailed in Section 5.1, fulfilment of the requirements, listed below, needs to be tested. CRF-21 is partially fulfilled by this module, as the 160 event counter controls the write request signal to the DCFIFO which contains the timestamps to be sent over Ethernet.

- R01 - Create a shared timing reference between both talker and listener
- R02 - Create a media clock of 48 kHz to use as a source clock
- R03 - Generate timestamps using the timing reference in R01 and source media clock in R02
- CRF-20 - The values of the timestamps increase monotonically, and with a constant nominal spacing in time, from the first timestamp to the last timestamp in the CRF AVTPDU when TU equals zero
- CRF-21 - In a CRF AVTPDU that contains multiple CRF timestamps, all timestamps derived from a continuous gPTP clock reference

As per the design, this module runs off of a 125MHz clock. The period of a 125 Mhz clock is 8 ns, and as such this is what the counter will increase by each clock cycle.

In order to validate that the buffer is storing the correct values, a Signal Tap file, with signals listed in Table 6.1 below, was created to capture the input and output values. These values were placed into a spreadsheet and the change from timestamp to timestamp measured.

Table 6.1: Signals in the STP file for testing the mock gPTP Module.

Signal	**Description**	**Expected Behaviour**
mclk_in	The 48kHz source clock	Toggle state periodically.
event_count	Counts the number of media clocks	Increase by 1 on the rising edge of mclk_in. Increase to 159, then wrap back to 0.
fifo_write	A write request to the TX FIFO	On the cycle that event_count wraps to 0, assert 1. Else assert 0.
timestamp_out	The gPTP value sent to the TX FIFO	Increase by either 3333328 or 3333336 each time the FIFO is written to.

The expected result between timestamps is the period of a 48kHz clock multiplied by 160 (to indicate every 160th clock event), then converted to nanoseconds is given in Equation 6.1

below:

$$\frac{1}{48000} \times 160 \times 10^9 = 3333333.333 \tag{6.1}$$

The value calculated is a recurring decimal value, as the period of a 48 kHz clock is recurring decimal value. In this implementation, given that the values are in multiples of 8, values of either $(\frac{1}{48000} \times 160 \times 10^9 \bmod 8) * 8$ or $((\frac{1}{48000} \times 160 \times 10^9 \bmod 8) + 1) * 8$ can be expected. These equate to 3333328 and 3333336 respectively.

6.3.1.1 Summary of gPTP Experimentation

In order to validate the gPTP module and ensure requirements are met, the following experiments need to be run:

1. Ensure the module correctly counts 160 media clock events
2. Ensure the time between each successive media clock event matches the period of a 48 kHz waveform
3. The time between each 160th media clock event matches the expected time passed in nanoseconds

Results of the experiments required to verify the expected behaviour of this module, as well as a short discussion on these results, can be found in Section 7.2.1.

6.3.2 The CRF Packet Generator

In this section, tests are designed in order to ensure the correct formation of packets as well as ensuring that the correct data is transmitted by the module designed in Section 5.2.

The requirements this module is intended to fulfil are as follows:

- R04 - Package the generated source timestamps in a format suitable for transmission over Ethernet, as described by IEEE 1722
- R05 - Transmit the packaged timestamps over Gigabit Ethernet in a way that makes it possible to observe them over the wire through an application such as Wireshark

- Partial fulfilment of R12 - IEEE 1722 standard and requirements should be implemented as much as reasonably possible

In terms of requirement R12, further reference is made to Table 2.2, for which this module implements the following:

- CRF-1 - The talker uses the alternative header
- CRF-2 - The subtype field is set to CRF
- CRF-3 - The mr bit is set as described in Section 10.4.3 of IEEE
- CRF-5 - Does the mr bit remain in its new state for a minimum of 8 CRF AVTPDUs?
- CRF-6 - Is the sequence_num field increment by one (1) with wrapping?
- CRF-8 - Is the base_frequency_field set to a value from 1 to 536 870 911 (1FFFFFFF16)?
- CRF-9 - Is the crf_data_length field value a multiple of 8 octets?
- CRF-10 - Is the timestamp_interval nonzero?
- CRF-11 - Is the fs bit set to zero (0) for CRF AVTPDUs with a type field set to CRF_AUDIO_SAMPLE?

The experiments described in this chapter are used to verify that the above listed requirements are correctly fulfilled. In order to test this module in isolation, test data was used for the timestamps as opposed to data from the gPTP module. For testing of the use of generated timestamps used in packets formed by this module an integration experiment was conducted. The description of this integration experiment can be found in Section 6.4.

6.3.2.1 Verify Packet Formation and Sequence Number

This test ensures that the packets are formed correctly in accordance with the literature discussed in Section 2.3 and in accordance to the CRF standard defined in Section 2.2. For this experiment, constant values were applied to all fields with the exception of the sequence number, and transmitted packets were captured in Wireshark. The captured frames were viewed in Wireshark to verify the correct values were obtained in the fields as per the requirements listed above. Successive frames were inspected to ensure that the sequence number met requirement CRF-6.

6.3.2.2 Verify Packets sent 50 times per second

A requirement of the CRF standard given alluded to by CRF-15 and defined in IEEE 1722 Table 28 is that packets consisting of 6 timestamps need to be sent 50 times per second, for a total of 300 timestamps per second. In order to validate this a filter was applied to Wireshark to only capture the IEEE 1722 CRF packets, and the statistics window used to validate the amount of packets per second.

6.3.2.3 Verify MR Flag

The MR (media clock reset) flag has a number of requirements for correct operation, detailed by CRF-3 and CRF-5 in Table 2.2. This test ensures the MR flag is set and reset correctly. In order to do so, a reset button was added to the board that only triggers the reset of the module in question. For implementation, the reset signal will be added to a system-wide reset signal. Upon triggering of this artificial reset, the MR flag should be toggled, but only if it is has been more than eight packets sent after the previous toggle. At a rate of 50 packets per second, each packet takes 0.02 seconds to transmit, and hence 8 packets will be completed within under 0.2 seconds. In order to verify that this is done correctly, the sequence number is reset to 1 for the next transmitted packet. This enables counting of packets between each reset trigger.

6.3.2.4 Summary on CRF Generator Module Experimentation

The section set out the experiments required to verify the requirements for this module were met by running the following experiments:

- Verify the packet formation and correct sequence number operation
- Verify that packets are sent 50 times per second
- Verify the operation of MR flag

Results and a brief discussion on this module can be found in Section 7.2.2. Experimentation relating to integrating this module into the larger design can be found in Section 6.4.

6.3.3 The Ethernet Receiving Module and AVTPDU Module

This section details the experiments required to verify the correct operation of the modules relating to reception and extraction of Ethernet Data, described in Sections 5.3 and 5.4.

While the functionality of reception and extraction of Ethernet data has been separated into two modules, testing and verification of the AVTPDU module infers correct operation of the Ethernet Reception module. Thus by verifying the design of the AVTPDU processor, the design of the Ethernet module is verified, as the verification of the AVTPDU processor depends on correct operation of the Ethernet receiver.

In order to verify that packets are received correctly, known packets need to be sent at known times. To achieve this, software called BitTwist can used to re-transmit captured packets at known times.

The two modules are designed to fulfil the following requirements:

- R08 - Unpacking of the data received over Ethernet and extraction of relevant information
- CRF-15, in that a listener is capable of receiving CRF AVTPDUs at the defined rates

Requirement R15 is known to be true, as the Ethernet reception module is capable of receiving data at rates of up to 1Gbps, and the amount of data contained in 50 CRF AVPTUs amounts to 24600 bytes per second. This means that the CRF AVTPDUs only utilise 0.01968% of the available bandwidth.

Requirement R08 necessitates a separate round of testing. The test will follow this process:

1. Packets will be captured to a `pcap` file using Wireshark.
2. The Ethernet cable connecting the router and the transmission side of the FPGA will be disconnected.
3. The lab computer will be connected directly to the receiving side of the FPGA to prevent any uncontrolled packets being sent by the router.
4. BitTwist will be used to send known packets from the lab computer to the receiving end of the FPGA using the following command:

```
$ bittwist -i <interface_id> -c <Number_of_packets> <capture_file.pcap>
```

5. These packets will be shown in a Signal Tap file, as well as captured using Wireshark.

6. The Wireshark capture and the Signal Tap file will be compared to ensure the correct values are being stored in the correct signals.

In order to validate these two modules, the signals shown in Table 6.2 can be inspected in Signal Tap.

Table 6.2: The signals under inspection for validation of the AVTPDU processor.

Signal	**Description**	**Expected Behaviour**
datain	The raw data from the TSE IP	Show the most recent 32 bits of the frame
ST_WAIT	The wait state	Active until an AVTPDU is received
ST_CONFIG	The state where initial data is extracted from the data in signal	If a CRF AVTPDU is received, the module enters this state and extracts the specified data
ST_TIMESTAMPS	The state where timestamps are formed and stored to the RX FIFO	After the config state, this state is used to process the data into correctly formed timestamps and write them to the DCFIFO
processing	A flag to indicate a valid AVTPDU packet has been received	Go high when a CRF packet is transmitted from BitTwist
timestamp_out	The 64 bit reformed timestamp	Change after two cycles spent in ST_TIMESTAMPS
timestamp_valid	The write request to the RX FIFO	Assert high when a timestamp is fully formed, otherwise low

As the verification of the values in the transmitted packet occurs in Section 6.3.2, it can be concluded that the data received by the Ethernet reception modules is consistent with the data sent by the Ethernet transmission modules.

6.3.3.1 Summary on Ethernet Receiving Modules Experimentation

This section covered details on experimentation to verify that the correct data is being received by the listener and that it is being correctly extracted from the CRF AVTPDU. This will be done by running two experiments:

1. Observing operations of the modules when a non-CRF AVTPDU is received.

2. Observing operations of the modules when a known CRF AVTPDU is transmitted, and ensuring the AVTPDU processing module extracts the expected data correctly.

The results of these experiments, as well as a short discussion on the results obtained, can be found in Section 7.2.3.

6.3.4 The CS2000 Control Module

This section describes the set of experiments which verify the operation of the `CSGEN` module in an open loop configuration - that is, without any form of correction made to align phases of the listener and talker media clocks. Closed loop tests are run in Section 6.4.3. The design of the module is described in Section 5.6.

The `CSGEN` module is the core of operation in the system, and extra consideration needs to be taken to ensure the open loop configuration and timestamp processing is modelled correctly in order to avoid a "garbage in, garbage out" problem from occurring. In order to achieve this, this section covers aspects of the open loop system, observing the operation of the device, and incrementally catering for problems found.

This section also sets out to verify R06 (Generate a media clock of 48 kHz which can be adjusted in phase in order to match up to the media clock in R02).

The section starts by verifying that the output of the CS2000 IC and frequency divider chain does indeed produce a 48kHz waveform. The open loop behaviours of the `CSGEN` module are then recorded. Notably, this includes a phase drift and an initial offset. Both of these factors are recorded using KeySight BenchVue software running on the lab PC, which allows measurements derived from the recorded media clocks to be monitored over time as opposed to just instantaneously as one does by observing only the oscilloscope.

6.3.4.1 Choosing the Phase Step and Producing a 48 kHz Waveform

As described in the design in Chapter 5, a 1 kHz wave is needed to drive the CS2000 IC to have the desired 48 kHz at the end of the processing chain. As the clock of the `CSGEN` module runs at 25 MHz, the simplest way to generate a 1 kHz wave would be by implementing a counter, counting to a specified value, and toggling the output. As described in Section 5.6, this is the approach that was taken. A counter (with the stored variable being designated `count_int`)is incremented by `phase_step`, up to the specified value `count_to`, at which point the output toggles.

The simplest (conceptual) implementation of these values would be to set `phase_step` to 1, and hence `count_to` to 12500. This would mean that every 12500 cycles of the 25 MHz clock, the output wave would toggle. The total period for this output wave would be $12500 \times 2 \times \frac{1}{25 \times 10^6} = \frac{1}{1000} s$, or a frequency of 1 kHz as desired.

It must be noted that there is an advantage to increasing `phase_step` - namely that greater values of `phase_step` will result in higher levels of accuracy available to the correction algorithm, which in turn results in a more precise output. Given that toggling an output every 12500 clock cycles of a 25 MHz clock results in a 1kHz output, and that a 1 kHz output is required in order for the CS2000 IC to produce a 48 kHz wave, the value of `count_to` must be a multiple of 12500, and the phase step value equal to that multiplying factor.

From the initial open loop configuration used in verifying that a 48 kHz waveform is produced, it was noted that there is considerable phase drift in the listener's media clock relative to that of the talker's. Given that oscillators are highly susceptible to temperature, it is hypothesised that the temperature directly affects this phase drift. In order to confirm this, three scenarios were considered:

1. Cold-start phase drift
   The phase drift measured upon first upload of the bitstream, after the FPGA has been off for a few hours.

2. Running phase drift
   The phase drift measured after the FPGA has been running for 20 minutes and has been determined to reach an unchanging operating temperature.

3. Altered conditions phase drift
   A hairdryer was used to artificially increase the temperature of the oscillator and surrounding circuitry in order to mimic a warmer environment.

At each point in this procedure, a thermocouple used by the UNI-T UT30C multimeter was used to measure the temperature of the heatsink on the FPGA. While not a direct measurement of the temperature of the oscillator generating the waveform, by using the heatsink a regulated temperature more indicative of the ambient conditions of the system can be obtained.

6.3.4.2 Summary of CSGEN Open Loop Experiments

The set of experiments above are designed to verify R6 (generate a media clock of 48 kHz which can be adjusted in phase in order to match up to the media clock in R02) and model

the behaviour of the open loop in order to later compare it to the closed loop implementation. This will be done by running a simple set of experiments, testing the output of the module and ensuring that a 48kHz waveform is produced. The results of all the above experimentation as well as design implications are recorded and discussed in Section 7.2.4.1.

6.3.5 The CS2000 IC and Gen DCFIFO Driver

The CS2000 IC and related modules described in Section 5.5 are responsible for generating a 48kHz wave and correctly creating timestamps. This operation is described by the following requirements:

- R06 - Generate a media clock of 48 kHz which can be adjusted in phase in order to match up to the media clock in R02
- R07 - Generate timestamps of the generated media clock in R06 using the shared timing reference in R01

Section 6.3.4 (with results shown in Section 7.2.4) verified that the output of the CS2000 IC is indeed 48 kHz as required by R06. It also showed a significant phase drift, dependant on environmental conditions. This complicates verification of R07.

In order to verify R07, a Signal Tap file was set up to capture the timestamp values. These will be compared alongside the phase drift as measured by the KeySight BenchVue software.

The rate of change in phase as reported by BenchVue will be used to determine if the rate of change in timestamp values is within range. This is done to cater for the phase drift of the system. The reason for this test is to ensure that the timestamps captured by the `GENDCFIFODriver` module are *reliable*, as opposed to other experimentation which looks for correctness. This is because the phase drift of the system means that the timestamps cannot be accurate to the expected rate of change of a constant 48kHz clock without drift.

The Signal Tap file simply captured the values of the output of the dual clock FIFO which held timestamps from the listener media clock. The timestamps were captured and a rate of drift calculated. This rate of drift was compared to the rate of drift reported by BenchVue. Results of this experiment can be found in Section 7.2.5.

6.4 Integration Testing

Integration testing is the process of combining modules together, and ensuring that under these conditions they continue to operate as expected. The goal of this integration testing and experimentation is to test the functionality of the components of the system as a whole, as opposed to just ensuring correct operation of the modules as Section 6.3 did.

The section breaks down experiments into functional groups, namely:

- Ethernet transmission and reception
- Measuring time considerations
- Closed loop experiments

6.4.1 Ethernet Transmission and Reception

CRF Packet generation was verified in Section 7.2.2. Specifically, the section verified the following aspects of transmission:

- Packets were correctly formed and the sequence number operated as expected
- Packets were sent 50 times per second
- The MR flag works as expected

The gPTP timing module was verified in Section 7.2.1. Specifically, the section verified the following:

- The module produces a timing reference in terms of nanoseconds passed since start
- The module captures the timestamp of every 160th media clock

Ethernet reception and the processing of the AVTPDU packets was verified in Section 6.3.3. Specifically, the experiments verified the following:

- The Ethernet module can handle reception of packets
- The CRF AVTPDU Processor correctly processes CRF AVTPDUs

In this experiment, these modules are combined. The timestamps captured by the gPTP module that represent the talker media clock are inserted into the CRF AVTPDUs and transmitted. These transmitted packets are recorded by Wireshark to verify that the timestamp increments both within and between packets is as expected. A Signal Tap file was also created which captured the following signals shown in Table 6.3 below.

Table 6.3: The signals captured by Signal Tap for the Ethernet TX and RX integration experiment.

Module	**Signal**	**Use**
CRFFrameGen	seq_num[7..0]	The sequence number of the AVTPDU
CRFFrameGen	dataout[31..0]	The data sent to the MAC
CRFFrameGen	state.ST_TIMESTAMPS	Set high when packaging timestamps for the MAC
ETHRX	avtp_pkt	Set high when an AVTPDU is received
CRFListener	processing	Matches the avtp_pkt signal from the ETHRX module
CRFListener	timestamp_out[63..0]	The currently held timestamp value
CRFListener	timestamp_valid	A write request to the DCFIFO, to write the value held in CRFListener.timestamp_out[63..0]

Two tests will be run against the data captured:

1. Consistency between the points of measure, namely:
    - Transmitted AVTPDUs on the `CRFFrameGen` module
    - Received AVTPDUs as per the `ETHRX` and `CRFListener` module
    - AVTPDUs recorded by Wireshark
2. Rate of change of timestamps between AVTPDUs

The results of the above experimentation as well as a discussion on these results can be found in Section 7.3.1.

6.4.2 Delays and Timing Within the System

This section describes experimentation designed to investigate timing and delays within the system. As the system is heavily time sensitive, the time taken for the transfer and use of data within the system needs to be considered. While the use of buffering of timestamps through dual clock FIFOs mitigates the need for certain hard time constraints, the time taken for timestamps to traverse the system needs to be investigated in order to determine

if consideration needs to be made when implementing the correction algorithm. These times then need to be compared to the rate of use of the timestamps in the `CSGEN` module to ensure the system operates within acceptable bounds, and does not under or over produce timestamp data. Figures 6.5 and 6.6 show the sequence of events for the listener and talker timestamps respectively.

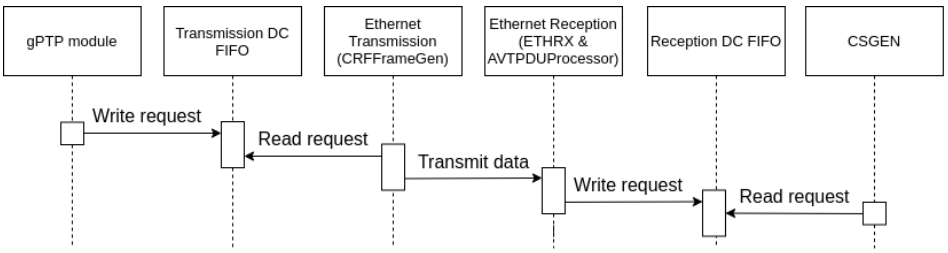


Figure 6.5: A sequence diagram for the formation, transmission, reception and use of talker timestamps.

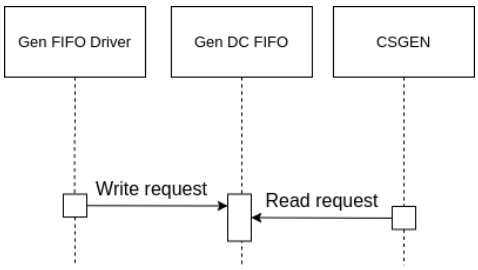


Figure 6.6: A sequence diagram for the formation and use of listener timestamp.

The time between messages in both sequence diagrams was measured. There are two approaches for this method, both involving the use of Signal Tap.

1. Measure the amount of time taken when relevant signals are triggered by using the time passed in the Signal Tap file.
2. Measure the delta between the recorded timestamp and the current gPTP time to use as an indicator of time when a specified event occurs.

Both methods are limited to a sampling rate of 125 MHz. Method 1 is useful as it allows measurements between two immediate points, but has the disadvantage of requiring large amounts of memory to be instantiated for the Signal Tap implementation. Method 2, while simpler (as it does not require large amounts of memory to be instantiated on the FPGA for the greater amount of samples needed to be captured for long periods of time between consecutive events) does have a disadvantage over Method 1. The disadvantage of Method

2 is that the rate of use of timestamps can be expected within a range which depends on multiple factors, such as if timestamps are available in the buffer, whether the `CSGEN` module is in the correct state for usage of the data, or whether or not a correction is due to be made in that given clock cycle. These factors can all affect the amount of time between the timestamp being compared to the gPTP time.

In order to mitigate the variance of these time deltas when using the second method, events at which these deltas need to be unique within the correction process. Using an example of fetching multiple listener timestamps, the concern can be mitigated by only measuring the time delta at the point when a valid error is detected. This is an event that only occurs once for each timestamp. There may be variance in this method, depending on how synchronised the timestamps are, but by choosing a unique event the potential for variance is mitigated as much as possible. The ideal method in this scenario would be to follow a unique timestamp through the process by using the first described measurement method, but these delays would be too large to be reasonably measured with a Signal Tap instantiation.

It can be expected that the time spent in the dual clock FIFOs on the FPGA are fairly consistent, due to constant rates at which timestamps are captured and used. As the only time that is of significance in the design for the timestamps is the period at which a phase error is detected, to calculate the timing delays in the system a Signal tap file can be drawn up with the signals described in Table 6.4 below:

Table 6.4: Signals used for measuring delays within the system.

Designator	**Module**	**Signal**	**Use**
A	gPTPGen	gptp_value	The running gPTP in increments of 8, given in ns
B	CSGEN	rx_timestamp	The talker timestamp, given in ns
C	CSGEN	gen_timestamp	The listener timestamp, given in ns
D	CSGEN	error	Triggered when two timestamps within range have a phase error

The above data can also be used to determine the time at which an error is first detected, which is representative of the time at which the correction algorithm may start. This can be done by using a *Power-Up Trigger*, available in Signal Tap. A Power-Up Trigger captures the first time a trigger is detected and stores the data in memory, to be later accessed and transferred to the Signal Tap instance.

The data obtained by running this Signal Tap file over time may show variance depending on a number of conditions. Most notably is that of the transmission time over Ethernet, shown in Figure 6.6. For this reason, an additional Signal Tap file was created with the signals shown in Table 6.5 below:

Table 6.5: Signals used for measuring Ethernet transmission delays.

Designator	**Module**	**Signal**	**Use**
E	CRFFrameGen	packet_end	Completion of the CRF AVTPDU sent to the MAC
F	ETHRX	packet_start	A new packet received by the listener

From all the above captured data, the following can be determined:

1. The `CSGEN` correction start-up time, given by a rising edge of D when compared with the value stored in A through the use of the Power-Up Trigger.
2. The delays between a timestamp being captured and the time at which it is used, given by comparing the values of B and D to that of A on a rising edge of D.
3. Network delays under varying conditions, given by $F - E$. In this scenario, the period of the Signal Tap clock can be used.

The results of the experiments, as well as a short discussion on the results, can be found in Section 7.3.2.

6.4.3 CSGEN Module Closed Loop testing

This section discusses the experimentation done to verify the closed loop operation of the design. In this experimentation the feedback loop is instantiated in order to correct the phase of the listener media clock to be synchronised with that of the talker media clock. The work of the previous experiments was done to verify that this was achievable in the implemented design, and the results of the previous experimentation indicates that it is. Of note are the results of the integration testing in Sections 6.4.1 and 6.4.2, which show that the generation, fetching and processing of timestamps in order to determine an error value was successful, as well as that the `CSGEN` module itself can process timestamps at the required rates.

At this point in the process the data and design comes into fruition: the timestamp data is used to adjust the `cnt_int` variable in the `CSGEN` module, which will adjust the phase of the waveform driving the CS2000 IC and in turn, the phase of the listener 48 kHz media clock. This is all done in order to align the listener media clock with the phase of the talker media clock. The KeySight EDUX1002G was connected to the BenchVue software in order to monitor changes in phase difference between the talker and listener over time.

The variables in the design that can be modified to affect performance of the phase correction algorithm include:

1. Loop gain coefficient
   By adjusting the loop gain coefficient (`lgcoeff`), the value the listener phase is corrected by can be increased or decreased for a calculated error. Larger values of `lgcoeff` will relate to less drastic corrections, as the relationship between the calculated offset between talker and listener waveforms is described as $correction = error \times 2^{-lgcoeff}$. However larger values of `lgcoeff` will also mean that some phase errors will go by undetected. The maximum phase difference allowed between talker and listener for media presentation as described by IEEE 1722 is 5%. With a 48 kHz clock, 5% in either direction is a phase difference of [-9,9] degrees. This means that `lgcoeff` must at least be able to correct within 9 degrees, or a nanosecond value (timestamp difference) of 520.

2. Phase step
   This value can be tweaked to refine the correction value calculated by the loop gain coefficient. As the nanosecond values remain the same for a given `lgcoeff`, the phase step value can be increased to make a correction less drastic relative to the total phase, or reduced to make a correction have a greater relative impact on the phase. This can be used as a modification in order to avoid floating point division in the design.

Once the tests for adjusting `lgcoeff` and `phase_step` were run, a simple averaging filter was implemented to adjust the correction value. By adding an averaging filter it is expected that the resulting output waveform will be less prone to jitter, which is not desirable in media applications.

6.4.3.1 Summary of Experiments for Closed-Loop Testing

In order to wholly model and verify closed-loop system behaviour, experiments for the following aspects of the system were run:

1. Base correction algorithm to model initial behaviour and verify logic
2. Adjustment of the phase step increment
3. Adjustment of the loop gain coefficient
4. Addition of averaging filter

These results, as well as a discussion on these results, can be found in Section 7.3.3.

6.5 Stress Testing

Stress testing considers what might happen under various irregular or extreme conditions and tests how well the system responds to these scenarios. The experiments are run in order to verify operation of the system as a whole. Given that the system is intended to work in vehicles in order to align media clocks, these tests examine how well the system may operate under those conditions. First, an extended duration test is run in order to examine how the system may operate over the course of a drive, or the length of an average album. Given that vehicles are susceptible to changing conditions environmental conditions, these are also investigated. A heat test examines how the system may operate under varying temperature. Finally, a network test will examine how well the system recovers from lost connectivity. This could be to unfavourable road conditions causing a cable to loosen, or a brown-out from a stalled vehicle.

Three experiments are run to verify behaviour of the system:

1. Running the system and recording the measured phase difference over a period of 30 minutes.
   This is to ensure that the system continues to operate in an expected manner.

2. Running the system under varying environmental conditions.
   This experiment will be run over 10 minutes, with a hairdryer pointing in the direction of the oscillator to create an artificially warmer environment, raised by roughly 20 degrees Celsius to a measurement of 60 degrees Celsius on the heatsink of the FPGA. The goal of this experiment is to determine:

   (a) How well the closed loop operates under extreme conditions.

   (b) If the hypothesis regarding an offset phase value posited in Section 7.3.3.2 is correct. If so, it is expected that a hotter temperature will result in a greater settling phase difference.

3. Running the system and testing how it responds to network connectivity problems.
   This experiment will involve arbitrarily disconnecting and reconnecting network cables, and seeing how long it takes the system to recover.

Results of these tests as well as brief discussions on those results can be found in Section 7.4.

6.6 Summary of Experimentation

This section has aimed to comprehensively define experimentation in order to implement a working implementation of the Clock Reference Format as defined by IEEE 1722. Testing was done to verify that the system operated as intended. A summary of the experimentation is shown in Table 6.6 below. Each aspect of the system is broken down with a relevant section for experimentation and results. Of significance in this project is the integration and closed loop testing. These experiments are covered in Section 6.4, with results given in Section 7.3.

Table 6.6: A summary of the experiments conducted, with reference to the results.

Focus	**Experiment Description**	**Experiment Section**	**Results Section**
Concept Formation	Python Simulation	6.2	7.1
gPTP Module	Ensure the module correctly counts 160 media clock events	6.3.1	7.2.1
gPTP Module	Ensure the time between each successive media clock event matches the period of a 48kHz waveform	6.3.1	7.2.1
gPTP Module	The time between each 160th media clock event matches the expected time passed in nanoseconds	6.3.1	7.2.1
CRF Frame Gen Module	Verify Packet Formation and Sequence Number	6.3.2	7.2.2
CRF Frame Gen Module	Verify Packets sent 50 times per second	6.3.2	7.2.2
CRF Frame Gen Module	Verify MR Flag	6.3.2	7.2.2
Eth RX and AVTPDU Processor Modules	Reception of non CRF AVTPDU	6.3.3	7.2.3
Eth RX and AVTPDU Processor Modules	Reception of CRF AVTPDU	6.3.3	7.2.3
CSGEN Module	Produce a 48kHz output	6.3.4	7.2.4
CSGEN Module	Measure effect of environmental conditions on output	6.3.4	7.2.4
CS2000 IC and processing chain	Difference in consecutive listener timestamps	6.3.5	7.2.5
Ethernet TX and RX Integration	Consistency and correctness of data as it moves through the system	6.4.1	7.3.1
Integration Test - Timing	Delays in the system	6.4.2	7.3.2
Integration Test - Timing	Rate at which data is processed	6.4.2	7.3.2.3
Closed Loop Testing	Adjusting loop gain	6.4.3	7.3.3
Closed Loop Testing	Adjusting phase step	6.4.3	7.3.3
Closed Loop Testing	Filtering of correction signal	6.4.3	7.3.3
Duration Stress Test	Run the system for an extended period	6.5	7.4
Temperature Stress Test	Submit the system to different environmental conditions	6.5	7.4
Network Conditions Stress Test	Simulate an ineffective network	6.5	7.4

Chapter 7

Results

This chapter presents the results of the experimentation presented in Chapter 6. Results and implications of the Python simulation presented in Section 6.2 are discussed in Section 7.1. Section 7.2 verifies the operation of each implemented HDL module. Section 7.3 presents the results of systematically integrating the modules in order to tie together aspects of functionality in the design, namely data transmission, verification of timing and finally forming a closed loop system as the complete design. Finally, Section 7.4 presents the results of operating the system under various conditions. While each section in this chapter discusses the results of the given experiment, an overall conclusion based upon these results can be found in Chapter 8.

7.1 Python Simulation Results

The Python simulation described in Section 6.2 resulted in proving the correction algorithm concept of using the difference in timestamps to determine a phase difference and adjust the output to the CS2000 to better align the talker and listener media clocks. As described in the introduction to that section, the simulation was developed to inform design decisions - not necessarily test the algorithms and methods that would ultimately be implemented. Thus the results in this section cannot be considered as results from testing the algorithm on hardware itself, but were used to inform design decisions and additional testing later in the development cycle as intended.

Many iterations of the simulation were run, each testing various aspects of potential correction algorithms and how they might affect the resulting generated listener media clock. One of the resulting graphs from the simulation is shown in Figure 7.1 below. The graph

shows the results of a variation on the algorithm implemented in the final design, which only performs corrections on positive phase difference values, when the listener timestamp was larger than the talker timestamp. The average value which the algorithm hovers around after the initial offset correction is a timestamp difference of about 500 nanoseconds, which is within 2.4% of the period of a media clock.

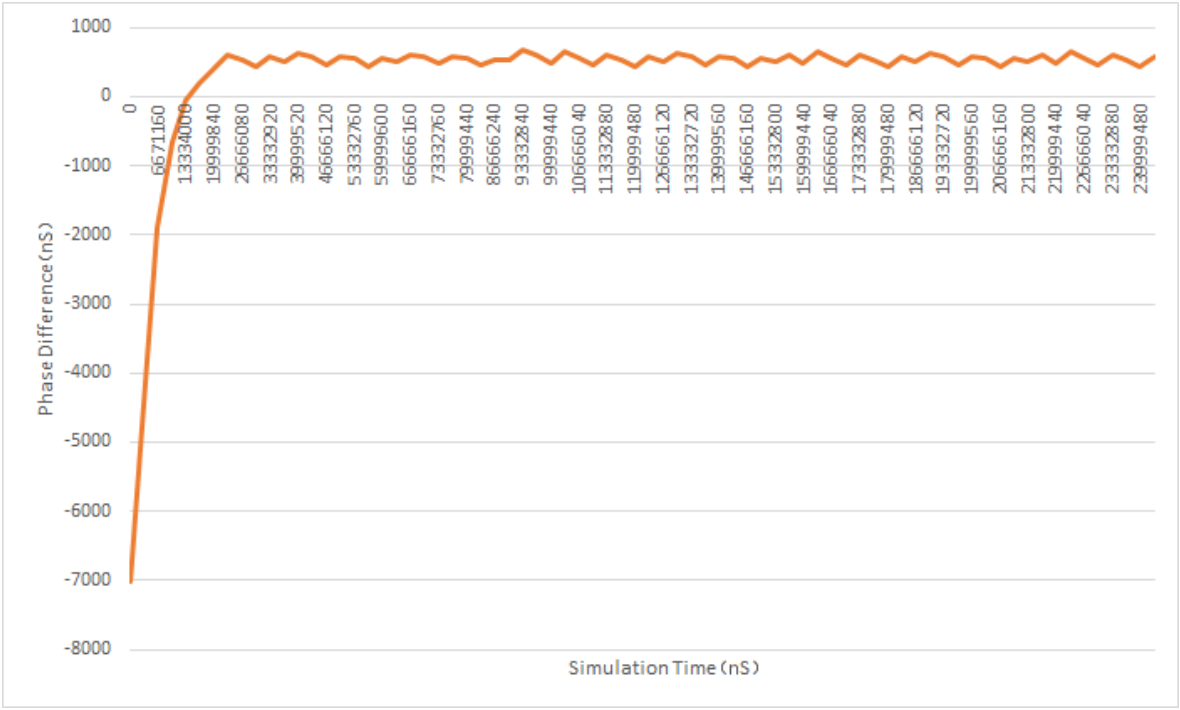


Figure 7.1: A graph showing changes in phase difference over time between talker and listener waveforms.

The information to be taken away from this result and the general findings of the simulations run are as follows:

1. The approach used of determining phase difference and adjusting the control waveform to the CS2000 module (which in turn adjusts the generated media clock) is indeed a valid approach for synchronising the talker and listener clocks.

2. The "jagged" corrections seen in Figure 7.1 may indicate that not enough resolution is available at the given `r_step` and `count_to` values, or that the correction algorithm does not offer fine enough corrections and would require adjustment. This could be mitigated through the use of a low pass filter.

3. The time for the algorithm in this experiment to reach the first "no correction" value (a state where the phases are considered in close enough alignment) is 16666760

nanoseconds, or just under 0.0167 seconds.

The findings from the Python simulation experimentation were used to better inform the design of the `CSGEN` module, and the specifics of the correction algorithm implemented.

7.2 Module Testing Results

This section details results of the experimentation detailed in Section 6.3. In this section, the results of the tests used to verify the performance of each module as well as the ability of each module to meet the requirements listed is discussed. The section starts with results of the gPTP module and works through the design, covering formation of frames, Ethernet transmission and reception, and finally the open loop behaviours of the `CSGEN` module.

7.2.1 gPTP Module

This section details the results of the experiment defined in Section 6.3.1.

First, a test was run to ensure the `event_count` (the value which counts the number of media clocks recorded) operated as expected, and measure the difference between the gPTP values at each `mclk` event. The results of this experiment are shown in Figure 7.2.

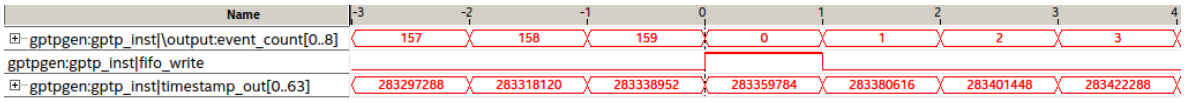


Figure 7.2: The Signal Tap results for the first gPTP test.

The values shown in the image as well as the differences between them is given in Table 7.1 below. The value for the column "Increase from Previous Value" is given by the equation shown in 7.1 below:

$$value[n] = gPTP[n] - gPTP[n-1] \tag{7.1}$$

Table 7.1: Difference in gPTP values between successive media clock events.

gPTP value	Increase from Previous value
283297288	
283318120	20832
283338952	20832
283359784	20832
283380616	20832
283401448	20832
283422288	20840

The second test was run to compare the values written to the TX FIFO when the write request went high. In order to perform this experiment the memory allocated to the Signal Tap instance was drastically increased. Table 7.2 below shows the timestamps recorded.

Table 7.2: Difference in gPTP values between successive captured timestamps.

gPTP value	Increase from Previous value
23970014960	
23973348304	3333344
23976681632	3333328
23980014968	3333336
23983348296	3333328
23986681632	3333336
23990014960	3333328
23993348296	3333336
23996681624	3333328
24000014960	3333336
24003348288	3333328
24006681624	3333336
24010014952	3333328

From these values it is clear to see that the expected timestamp differences consist of the expected values of 3333328 or 3333336. There is one exception to this in the recorded results of 3333344 - greater than an expected value by 8, or one 125 MHz clock cycle.

In order to further investigate the effect this might have on the accuracy of the system, the cumulative difference of the timestamps recorded is compared to the cumulative difference of the calculated values of the timestamps, given by Equation 6.1. The results are shown

in Table 7.3 below. The first column, `gPTP Value`, gives the recorded timestamp value. `Cumulative Difference` gives the difference from the current timestamp in that particular row against the first recorded timestamp given in row 1. `Expected Cumulative Difference` adds the period of the 48 kHz clock to the first gPTP timestamp given in row 1. The final column gives the difference between the measured and calculated differences rounded to 0.001 nanoseconds. This serves as an indication of the accuracy of the system that implements the gPTP timestamps.

Table 7.3: Comparing cumulative differences between recorded and calculated timestamp values.

gPTP Value	**Cumulative Difference**	**Expected Cumulative Difference**	**Difference between Cumulative Differences (rounded)**
23970014960			
23973348304	3333344	3333333.333	10.667
23976681632	6666672	6666666.667	5.333
23980014968	10000008	10000000	8
23983348296	13333336	13333333.33	2.667
23986681632	16666672	16666666.67	5.333
23990014960	20000000	20000000	0
23993348296	23333336	23333333.33	2.667
23996681624	26666664	26666666.67	-2.667
24000014960	30000000	30000000	0
24003348288	33333328	33333333.33	-5.333
24006681624	36666664	36666666.67	-2.667
24010014952	39999992	40000000	-8

7.2.1.1 Summary of Results

Referring to Table 7.1, the difference in media clock values is acceptable. The expected period is 20833.33 ns, so by having 20832 (being short by 1.3) and then having 20840 (over by 6.67) means it can be expected that this behaviour will repeat roughly every five clock cycles (6.67/1.33 = 5). While this is not precise, what matters is value of the timestamps captured over a longer period of time.

While Table 7.2 has reason for concern regarding the difference value of 3333344 which is outside of the expected values of 3333328 or 3333336, Table 7.3 provides evidence that, over

time this difference has no effect on the system. Particularly, it is worth noting that once the system starts running, the cumulative difference between every third measured timestamp and its calculated counterpart is zero. This is mathematically justifiable, as given the value calculated in Equation 6.1 has a factor of $\frac{1}{3}$, every third timestamp would equate to the measured timestamp created by the implemented system, as long as the implemented system handled rounding errors correctly over time. Based on the results shown in Tables 7.2 and 7.3, this module is considered successfully implemented, and fulfils the following requirements:

- R01 - Create a shared timing reference between both talker and listener
- R02 - Create a media clock of 48 kHz to use as a source clock
- R03 - Generate timestamps using the timing reference in R01 and source media clock in R02
- CRF-20 - The values of the timestamps increase monotonically, and with a constant nominal spacing in time, from the first timestamp to the last timestamp in the CRF AVTPDU when TU equals zero
- CRF-21 - In a CRF AVTPDU that contains multiple CRF timestamps, all timestamps derived from a continuous gPTP clock reference

7.2.2 CRF Packet Generator and Ethernet Transmission

This section details the results of experiments defined in Section 6.3.2. The point of this experimentation is to verify that the module fulfils the requirements it was designed for, as described in Section 5.2.

7.2.2.1 Verify Packet Formation and Sequence Number

An example of the frame generated by the CRF generator module as captured by Wireshark can be seen in Image 7.3 below. The intention of this test was to validate the formation of the packet.

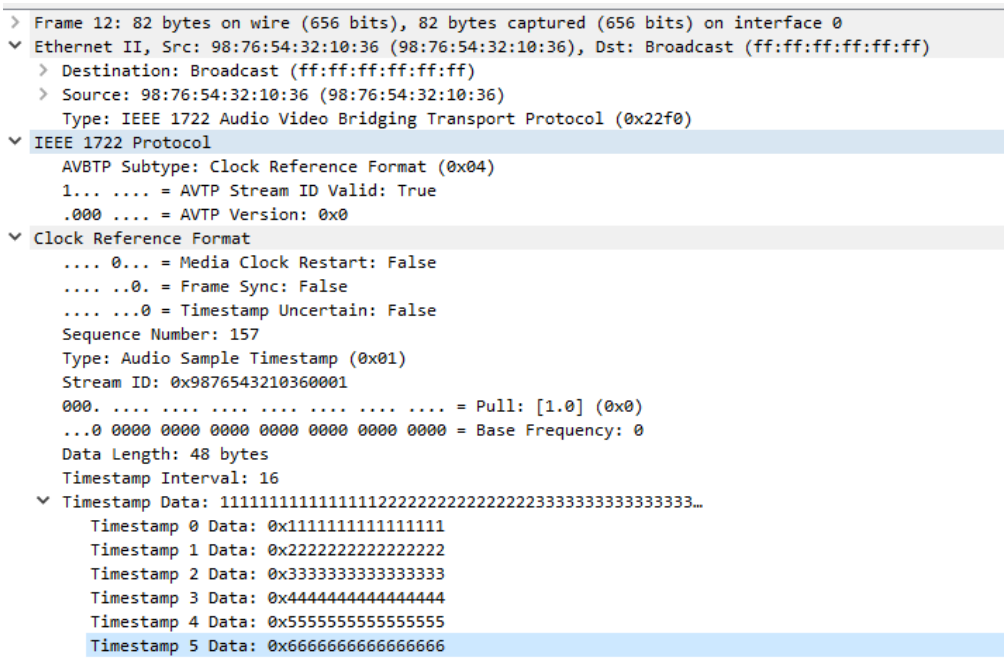


Figure 7.3: Output of the first implementation of the CRF Frame Generator.

Inspection of each field verifies that the module described in Section 5.2 does indeed correctly form the frames. Of primary concern are the fact that it is detected as part of the AVB Transport Protocol and is correctly detected as the Clock Reference Format.

Inspection of the `pcap` file also confirms that the sequence number operates in accordance to the requirements, with values monotonically increasing until 255 and then overflowing to 0.

7.2.2.2 Verify Packets sent 50 times per second

50 packets per second is the requirement for IEEE 1722 CRF. Figure 7.4 shows the Wireshark analysis file with a filter applied to only consider CRF Frames.

Statistics

Measurement	Captured	Displayed	Marked
Packets	5865	5784 (98.6%)	—
Time span, s	115.660	115.660	—
Average pps	50.7	50.0	—
Average packet size, B	85	82	—
Bytes	496499	474288 (95.5%)	0
Average bytes/s	4292	4100	—
Average bits/s	34 k	32 k	—

Figure 7.4: Rate of packets sent by the CRF Packet Generator.

7.2.2.3 Verify MR Flag

Table 7.4 shows the sequence number and MR flag value for the reset trigger experiment as recorded. A sequence number of 1 indicates the module was reset.

Table 7.4: Sequence number vs MR flag.

Seq Number	MR Value	Description
11	0	
12	0	
1	1	Reset triggered, more than 8 frames transmitted so MR toggled
2	1	
3	1	
1	1	Reset triggered, less than 8 frames transmitted so MR not toggled
2	1	
3	1	
1	1	Reset triggered, less than 8 frames transmitted so MR not toggled
2	1	
3	1	
1	0	Reset triggered, more than 8 frames transmitted so MR toggled
2	0	
1	0	

From this experiment it can be seen that the behaviour or the MR flag is as intended. Upon reset, the MR flag is toggled. If another reset is triggered and if there have been less than eight frames transmitted after the flag has been toggled, the MR flag remains unchanged.

7.2.2.4 Summary of Results

From the experiments above, the following requirements are verified as being fulfilled:

- R04 - Package the generated source timestamps in a format suitable for transmission over Ethernet, as described by IEEE 1722
- R05 - Transmit the packaged timestamps over Gigabit Ethernet in a way that makes it possible to observe them over the wire through an application such as Wireshark
- CRF-1 - The talker uses the alternative header
- CRF-2 - The subtype field is set to CRF

- CRF-3 - The mr bit is set as described in Section 10.4.3 of IEEE 1722
- CRF-5 - Does the mr bit remain in its new state for a minimum of 8 CRF AVTPDUs?
- CRF-6 - Is the sequence_num field increment by one (1) with wrapping?
- CRF-8 - Is the base_frequency_field set to a value from 1 to 536 870 911 (1FFFFFFF16)?
- CRF-9 - Is the crf_data_length field value a multiple of 8 octets?
- CRF-10 - Is the timestamp_interval nonzero?
- CRF-11 - Is the fs bit set to zero (0) for CRF AVTPDUs with a type field set to CRF_AUDIO_SAMPLE?

7.2.3 Ethernet Reception and AVTPDU Processor

This section covers the results of experiments described in Section 6.3.3. For verifying the operations of these modules, two sets of experiments are run, namely:

1. Observing operations of the modules when a non-CRF AVTPDU is received.
2. Observing operations of the modules when a known CRF AVTPDU is transmitted, and ensuring the AVTPDU processing module extracts the expected data correctly.

By running these experiments, the operation of these modules can be verified simultaneously.

7.2.3.1 Reception of a non-CRF AVTPDU Frame

Figure 7.5 shows how the AVTPDU module responds when the Ethernet Receiver receives a non-AVTPDU packet.

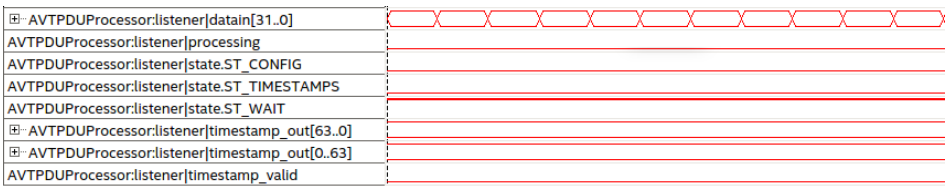


Figure 7.5: Example of a Signal Tap capture for reception of a non-CRF frame.

It can be seen that the data coming from the Ethernet IP changes as the packet is received, but the module never exits the wait state, indicating that the system doesn't erroneously process non-CRF AVTPDU frames.

7.2.3.2 Reception of a CRF AVTPDU Frame

In order to verify correct processing operation, a known frame can be sent to the Ethernet reception module using BitTwist. The CRF AVTPDU transmitted is shown in Figure 7.6. Figure 7.7 shows the Signal Tap capture used to verify the packet was correctly received.

```
▸ Frame 2: 82 bytes on wire (656 bits), 82 bytes captured (656 bits) on interface enp3s0, id 0
▸ Ethernet II, Src: 98:76:54:32:10:36 (98:76:54:32:10:36), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
▸ IEEE 1722 Protocol
▾ Clock Reference Format
    .... 0... = Media Clock Restart: False
    .... ..0. = Frame Sync: False
    .... ...0 = Timestamp Uncertain: False
    Sequence Number: 163
    Type: Audio Sample Timestamp (0x01)
    Stream ID: 0x9876543210360001
    000. .... .... .... .... .... .... .... = Pull: [1.0] (0x0)
    ...0 0000 0000 0000 1011 1011 1000 0000 = Base Frequency: 48000
    Data Length: 48 bytes
    Timestamp Interval: 160
  ▾ Timestamp Data: 00000000b3cf101000000000b401ece800000000b434c9b8…
      Timestamp 0 Data: 0x00000000b3cf1010
      Timestamp 1 Data: 0x00000000b401ece8
      Timestamp 2 Data: 0x00000000b434c9b8
      Timestamp 3 Data: 0x00000000b467a690
      Timestamp 4 Data: 0x00000000b49a8360
      Timestamp 5 Data: 0x00000000b4cd6038
```

Figure 7.6: The frame sent to test Ethernet reception operation, viewed in Wireshark.

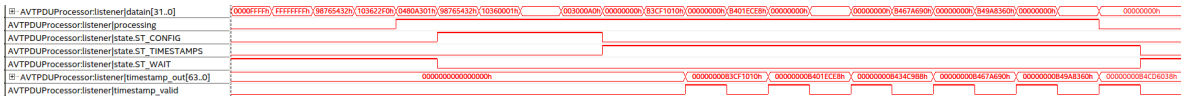


Figure 7.7: An example of the Signal Tap output of the recorded signals when a CRF AVTPDU is received.

Of most importance in the implemented design is the correctness of the stored timestamps. Comparison of the frame as displayed by Wireshark and the captured timestamps show that they match.

7.2.3.3 Summary of Results

Based on the results in Section 7.2.3 it can be seen that the Ethernet receiver and AVTPDU Processor act as intended. As a consequence of these results, the following requirements are met:

- R08 - Unpacking of the data received over Ethernet and extraction of relevant information
- CRF-15, in that a listener is capable of receiving CRF AVTPDUs at the defined rates

Most importantly to the implementation, the correct timestamp values are written out at the appropriate times to the RX FIFO to be used by `CSGEN`, the CS2000 driver module.

7.2.4 The CS2000 Control Module

This section describes testing and modelling of the open loop configuration of the `CSGEN` module. The design of this module is described in Section 5.6, with experimentation described in Section 6.3.4.

7.2.4.1 Ensuring a 48 kHz generated wave

As discussed in Section 6.3.4.1, a counter was implemented to generate a 1 kHz waveform at the output of the `CSGEN` module, which would result in a 48 kHz waveform. Shown in Figure 7.8 is a capture from the oscilloscope. The talker waveform is Channel 1, shown in yellow, with the generated waveform for the listener recorded as Channel 2, shown in green.

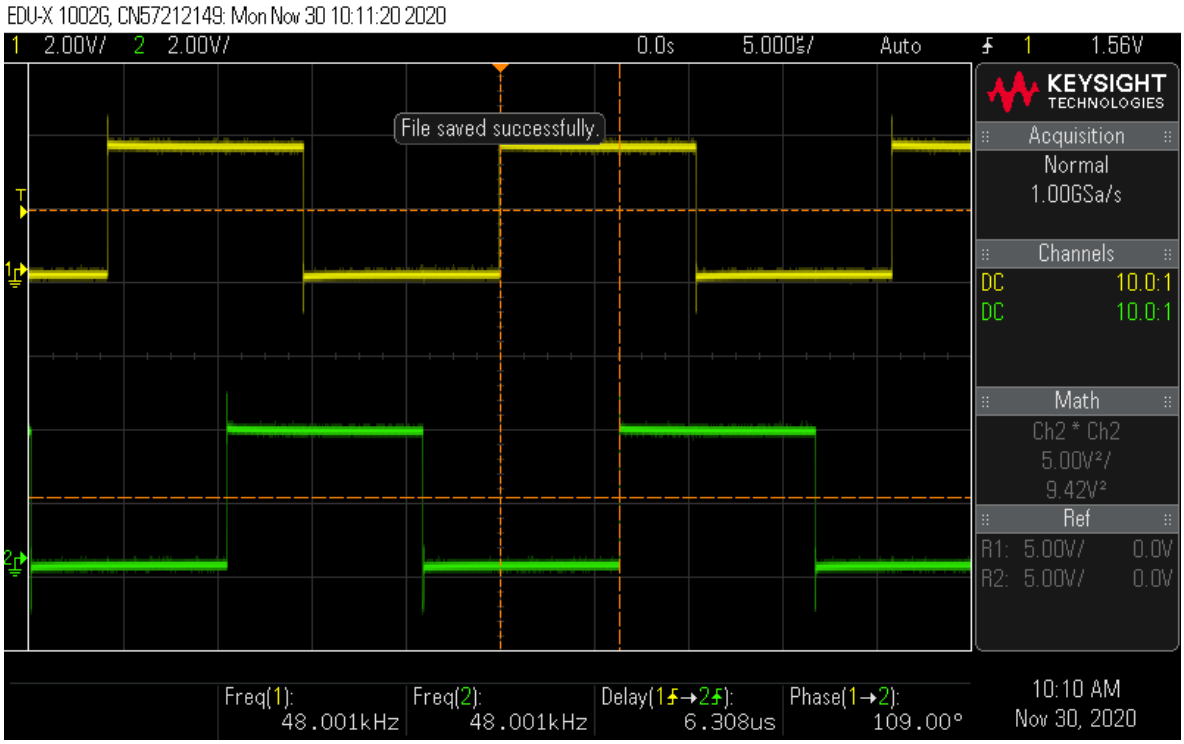


Figure 7.8: A screen-shot of the oscilloscope showing both talker and listener at 48 kHz.

While initial inspection of the oscilloscope suggests that both listener and receiver media clocks are at the same frequency, further inspection suggests there are more factors at play.

From conducting multiple runs of uploading the bitstream, the following was noted:

- There is always an initial offset between the two waves.
- There is a phase drift of the generated wave when compared to the talker waveform.

Point 1 is to be expected given that the two waves are not designed to be in phase from the start. Point 2, regarding phase drift, is cause for concern. Both waveforms above use the same source oscillator and both of those waveforms are divided down using an instantiation of the frequency divider module (the counter described in Section 5.5), but the generated waveform uses the 1 kHz waveform and the CS2000 IC as opposed to a division straight from a 24.576 MHz oscillator on the board. Plotting this phase difference over time using KeySight's *BenchVue* software (sampling every 500mS) results in the plot shown in Figure 7.9 below[1].

[1]The "bumps" in the figure are from when screenshots of the oscilloscope were taken.

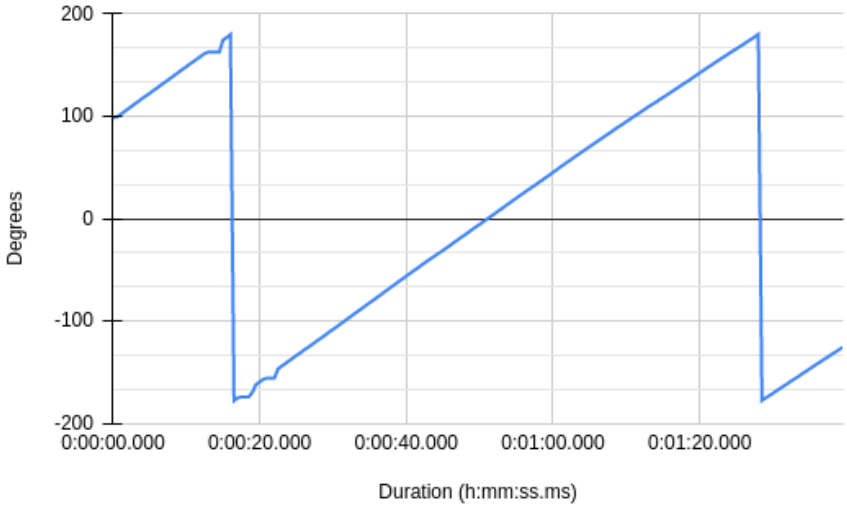


Figure 7.9: A plot of relative phase between talker and listener media clocks over time.

It is expected that this phase drift could be variable based on the conditions of the oscillator on the day (such as temperature), and is not constant. In order to show this, the data shown in Table 7.5 below was recorded.

Table 7.5: The time taken for a 360 degree phase drift.

Experiment	**Temperature (°C)**	**Net duration**
Cold Start	24	0:01:11.999
After 20 minutes	42	0:01:28.501
After 50 minutes	42	0:01:28.001
Hairdryer	61	0:00:07.000

These results indicate that the phase drift is indeed dependant upon environmental conditions.

7.2.4.2 Summary of Results

The module is successful in producing a 1 kHz output waveform, which, in turn, successfully produces the required 48 kHz waveform. As noted, there is a phase drift relative to the talker's 48 kHz media clock dependant on environmental conditions.

As a result of the experimentation, R06 - "Generate a media clock of 48 kHz which can be adjusted in phase in order to match up to the media clock in R02" has been fulfilled.

7.2.5 The CS2000 IC and GEN DCFIFO Driver Module

This section reports on the results of the experiment described in Section 6.3.5, which is designed to verify the operation of the `GENDCFIFODriver` module and Requirement R07 (Generate timestamps of the generated media clock in R06 using the shared timing reference in R01) is met.

A total of 74 timestamps were captured by the Signal Tap instantiation. In order to ensure that these timestamps were representative of a 48 kHz clock, consider the plot of their successive differences shown in Figure 7.10 below:

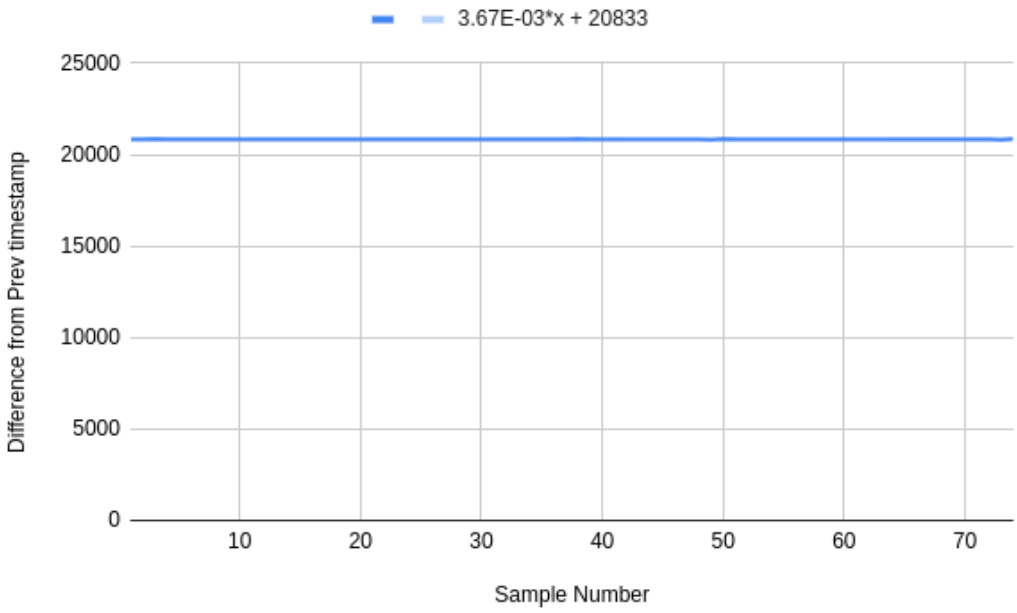


Figure 7.10: A chart showing the difference between successive captured timestamps.

As shown in the figure, the line of best fit on this chart is characterised by the equation $y = 3.67 \times 10^{-3} \times x + 20833$. Given that the period of a 48 kHz clock is 20833.33 ns, this result confirms that the `CSGEN` module is indeed producing the 1 kHz wave required to produce the 48 kHz waveform at the end of the CS2000 chain.

In terms of the phase drift, this too can be modelled by adding the consecutive differences between the received timestamp value, and the expected timestamp value. A plot of the difference between the captured timestamp value and the "actual" (theoretical) timestamp value is shown in Figure 7.11 below.

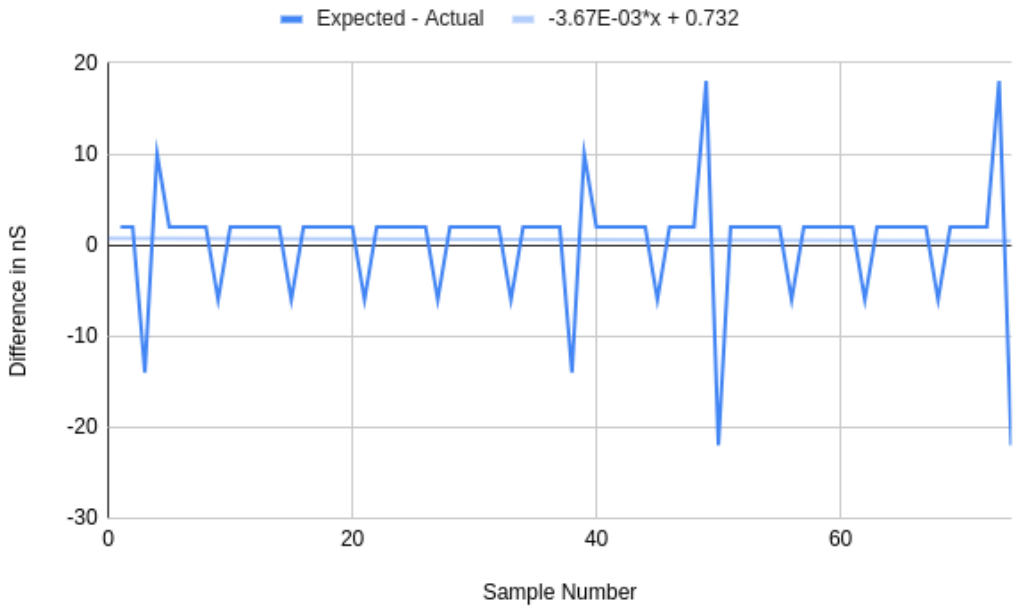


Figure 7.11: A chart showing the difference between a measured timestamp and the calculated value.

As shown in the figure, the line of best fit modelling this equation is given by the formula in 7.2:

$$y = -3.67 \times 10^{-3}x + 0.732 \tag{7.2}$$

This is the difference between successive timestamps, which are 20833.33 ns apart. If this difference were to be calculated over the course of 500 ms (the period used by BenchVue for capturing data off the oscilloscope), the total difference over the course of 500 ms based upon the data measured would be 1.51 degrees. This is 0.5 degrees off from the average change in degrees reported by the BenchVue software over the course of 90 samples, which was calculated to be 2.01 degrees. The difference may be due to not enough samples captured by the test bench file, as only 74 timestamps were recorded, which is a total duration of less than 2 milliseconds.

7.2.5.1 Summary on Results

The results shown above confirm the findings of the results discussed in Section 7.2.4, namely that the output of the CS2000 IC processing chain does indeed produce a 48 kHz output, and that there is a phase drift. The testing also shows that the timestamp values captured by the dual clock FIFO by `GenFIFODriver` module are reliable, verifying that requirement R07

- generate timestamps of the generated media clock in R06 using the shared timing reference in R01 - is met.

7.3 Integration Testing Results

This section reports on and discusses results of the experimentation presented in Section 6.4. The objective of this experimentation was to systematically integrate modules together and ensure the modules operated as intended.

7.3.1 Ethernet Transmission and Reception

This section reports on the experiments described in Section 6.4.1. The two experiments conducted were:

1. Consistency between the points of measure
2. Rate of change of timestamps between AVTPDUs

7.3.1.1 Consistency Between Points of Measure

In this experiment, talker timestamps generated by the gPTP module were transmitted in CRF AVTPDUs and captured by Wireshark to verify that the correct Timestamp intervals were used. A Signal Tap file was used to capture the signals from the FPGA. Data transmitted byt the `CRFFrameGen` was compared to the next immediately received packet on `ETHRX`.

Shown in Figure 7.12 is the frame as captured by Wireshark.

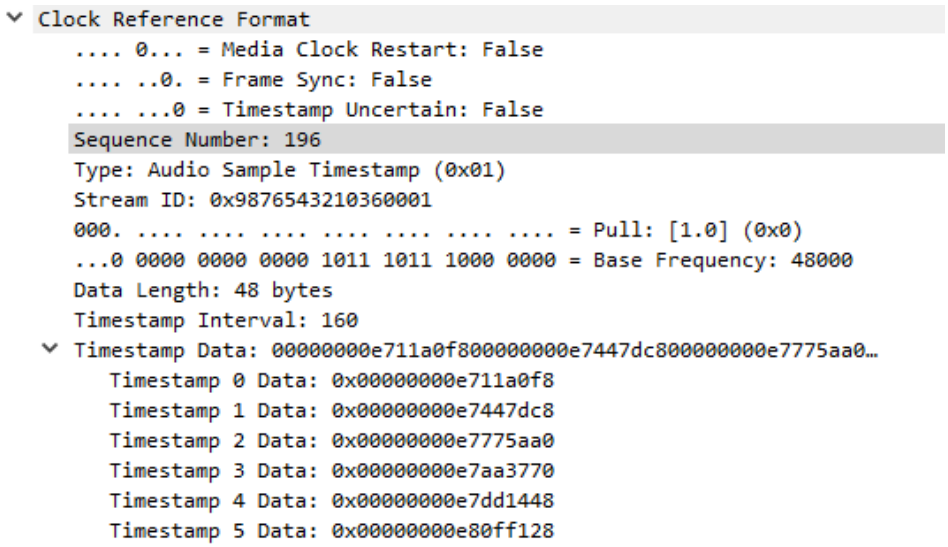


Figure 7.12: The packet as captured by Wireshark.

Comparing the above with the signals captured by Signal Tap with the values captured by Wireshark, Table 7.6 is constructed:

Table 7.6: Comparison between TX, Wireshark, and RX timestamp values.

TX		**Wireshark**		**RX**	
Seq Num	**TS (Hex)**	**Seq Num**	**TS (Hex)**	**TS_Valid**	**TS (Hex)**
196	00000000E711A0F8	196	00000000E711A0F8	1	00000000E711A0F8
196	00000000E7447DC8	196	00000000E7447DC8	1	00000000E7447DC8
196	00000000E7775AA0	196	00000000E7775AA0	1	00000000E7775AA0
196	00000000E7AA3770	196	00000000E7AA3770	1	00000000E7AA3770
196	00000000E7DD1448	196	00000000E7DD1448	1	00000000E7DD1448
196	00000000E80FF128	196	00000000E80FF128	1	00000000E80FF128

It can be seen from the above table that timestamps between transmission and reception is consistent, and that Wireshark correctly reports on these values, too.

7.3.1.2 Rate of Change of Timestamps in AVTPDUs

Given that the above experiment shows that data is consistent between Wireshark and the transmitted and received frames on the FPGA, Wireshark alone can be used here to test the rate of change between timestamps transmitted in order to verify correct operation of the gPTP module over time. 25 packets will be inspected, which, at a rate of 50 packets per

second as per the IEEE specification, should result in a 0.5 s test. The first five packets will have all timestamps reported, thereafter only the first timestamp from the AVTPDU will be captured. The captured timestamps are recorded in Appendix E, Table E.1.

In the table, the magnitude of the difference between consecutive intra-AVTPDU timestamps differs slightly. This is expected as the period of a 48 kHz clock is a non-terminating repeating floating point value, which cannot be captured by the gPTP module, which is limited to 8 ns accuracy. The calculated theoretical value between successive timestamps is given as $160 * \frac{1}{48000}$ seconds, or 3333333.333 nanoseconds. Taking the average of the values over the five timestamps, the calculated average is 3333332.414 ns, off by the theoretical value by 0.9193 ns.

When considering the difference in the first timestamp between successive AVTPDUs, the average over 25 AVTPDUs is found to be 19999995.2 ns. The theoretical value is six times larger than the theoretical value for intra-AVTPDU timestamps, for a value of 20000000. The difference is a value of 4.8 ns, well within the 8 ns precision of the system.

7.3.1.3 Summary of Results

This section showed that the talker timestamps are consistent along the transmission line from talker to receiver. It also showed that the values of the timestamps to be used by the `CSGEN` module are accurate enough to be used by the system. Values that are off from the theoretical value can be explained through drift of the oscillator and the limited precision of the system.

7.3.2 Delays within the System

This section presents the results of the testing described in Section 6.4.2. The objective of this set of testing was to measure delays within the system to determine if any compensation need be made when applying the result of correction algorithm to the generated media clock in order to correctly synchronise it with the talker media clock. The results of the experimentation also provided insight into how the system may operate under different networking conditions. The results of the network timing are presented first as they impact the start-up time of the `CSGEN` module. For the `CSGEN` module start-up, two sets of experiments were run, with each experiment capturing 10 samples.

For the rate of use of timestamps, a large amount of memory was instantiated on the FPGA for the Signal Tap file. This allowed multiple samples to be compared over time.

7.3.2.1 Network Timing

For network timing, three experiments were run using the Signal Tap file described in Table 6.5. The first made use of a 30 cm Ethernet cable directly between the TX and RX ports on the FPGA, and the second experiment replaced the 30 centimetre cable with a cable 1 metre in length. The third experiment measured the transmission times when a router was positioned between the RX and TX ports. The router, the TP-Link AC1200, was connected to the FPGA RX and TX with two Ethernet cables both 30 cm in length. Ten captures for each experiment resulted in the average times shown in Table 7.7 below. The time taken for a packet to form on transmission is determined by the design of the module (described in Section 5.2) and is constant, taking 0.0002 milliseconds. This time is not included in the times below.

Table 7.7: The average time in ms from the end of packet signal on the talker to the start of packet signal on the listener.

Experiment	**Average time (ms)**
30 cm direct connection	0.001611
1 m direct connection	0.001614
30 cm both ways through Router	0.004169

It can be seen that inclusion of a router drastically increases the delay by up to 2.5 times. However, all three experiments still allow for the 50 packets per second requirement as per Requirement CRF-15.

7.3.2.2 Start-up Time

The start up time was measured by using a Power-Up Trigger in Signal Tap in order to detect the first time at which an error was detected by the `CSGEN` module. The conditions for an error signal are described in Section 5.6. The given talker and listener timestamps at the point of first error detection were also recorded. The results for a direct 30 cm connection is shown in Table 7.8, while Table 7.9 shows the results of using the 30 cm - router - 30 cm setup described in the network timing results above. All values are given in nanoseconds.

Table 7.8: The average system start times when using a 30 cm Ethernet cable connected directly between the listener and talker Ethernet ports.

Run	gPTP Time	Talker (RX) Value	Listener (Gen) Value	RX-Gen (Error Value)	gPTP-RX	gPTP-Gen
1	2463402752	2426692512	2426683000	9512	36710240	36719752
2	2463402752	2426692528	2426682904	9624	36710224	36719848
3	2543402752	2506692520	2506682976	9544	36710232	36719776
4	2543402712	2506692448	2506682808	9640	36710264	36719904
5	2383402704	2346692536	2346682872	9664	36710168	36719832
6	2383402672	2346692504	2346682968	9536	36710168	36719704
7	2383402672	2346692488	2346682960	9528	36710184	36719712
8	2563402784	2526692482	2526682984	9498	36710302	36719800
9	2383402712	2346692512	2346682944	9568	36710200	36719768
10	2383402672	2346692456	2346682872	9584	36710216	36719800
Average	**2449402718**	**2412692499**	**2412682929**	**9569.8**	**36710219.8**	**36719789.6**

Table 7.9: The average system start times when using a router and two 30 cm Ethernet cables.

Run	gPTP Time	Talker (RX) Value	Listener (Gen) Value	RX-Gen (Error Value)	gPTP-RX	gPTP-Gen
1	3223405544	3186692328	3186682928	9400	36713216	36722616
2	3043405552	3006692360	3006683032	9328	36713192	36722520
3	3103405512	3066692376	3066682952	9424	36713136	36722560
4	3183405592	3146692320	3146682912	9408	36713272	36722680
5	3103405472	3066692328	3066683040	9288	36713144	36722432
6	3163405512	3126692344	3126683032	9312	36713168	36722480
7	3203405552	3166692344	3166682992	9352	36713208	36722560
8	3183405592	3146692344	3146682888	9456	36713248	36722704
9	3103405592	3066692336	3066682960	9376	36713256	36722632
10	3243405512	3206692280	3206682880	9400	36713232	36722632
Average	**3155405543**	**3118692336**	**3118682962**	**9374.4**	**36713207.2**	**36722581.6**

As shown by Table 7.8, the average system start time without using a router is given as roughly 2.4 seconds. With a router, roughly 3.1 seconds. The timestamps are, as expected, also roughly 0.7s apart, the same as the difference in start up times between the two network configurations. The differences of the averages of the error value, the gPTP value less the talker timestamp and the gPTP value less the listener timestamps are negligible, which suggests consistent operation of the system independent of the network conditions after catering for the network delays.

7.3.2.3 Rate of use of Timestamps

Over the course of a single run (i.e. no reboots or re-uploads of the bitstream to the FPGA), the talker and listener timestamps were captured alongside the gPTP time at the rising edge of the error flag. A final Signal Tap capture was taken after roughly two hours to see if there was any significant change in the relationships of the data. Some of the captured data was corrupted due to the requirement to have a large amount of data samples in instantiated memory, so only valid and correctly formed samples were used. From these valid samples, the difference between the listener timestamps and the gPTP time, as well as the difference between the talker timestamp and the gPTP time was calculated.

The summary of nine Signal Tap captures can be found in Table E.2 in Appendix E.2. Notable from the table is the "Duration into capture" column, which indicates how long into a Signal Tap capture that given sample was recorded at. It can be seen that all captures are at times 10280, 29480, 48680 or 67880 nanoseconds (however not all captures have samples at these times due to sampling errors). These are consistently spaced at 19200 nanoseconds apart (or 480 clock cycles of the 25 MHz clock), which indicates a constant rate of processing. Given that the listener timestamps update every three clock cycles on average as the module moves through the states described in Section 5.6, this suggests that every 160th listener timestamp is being used for comparison, which aligns with how talker timestamps occur every 160 media clock events. As the average rate of processing at 19200 nanoseconds is less than that of the period of the media clock, the `CSGEN` module is capable of detecting errors and correcting for them in a timely fashion, in accordance with requirement CRF-15 (A CRF Audio Listener is capable of receiving CRF AVTPDUs at the rates given in IEEE 1722 Table 28).

By plotting the calculated differences between the gPTP value and the talker timestamp, as well as the gPTP value and the listener timestamp, both as a function of the gPTP time, a correction drift rate can be determined. This is not the phase drift of the generated wave, but instead acts as a performance metric for the `CSGEN` module, as a measure of how well it "keeps up" with the rate at which media clocks are produced. Figure 7.13 shows changes in the difference between gPTP time and the talker timestamps as well as changes in the difference between gPTP time and the listener timestamps. The values have been adjusted from nanoseconds to seconds, for the sake of readability.

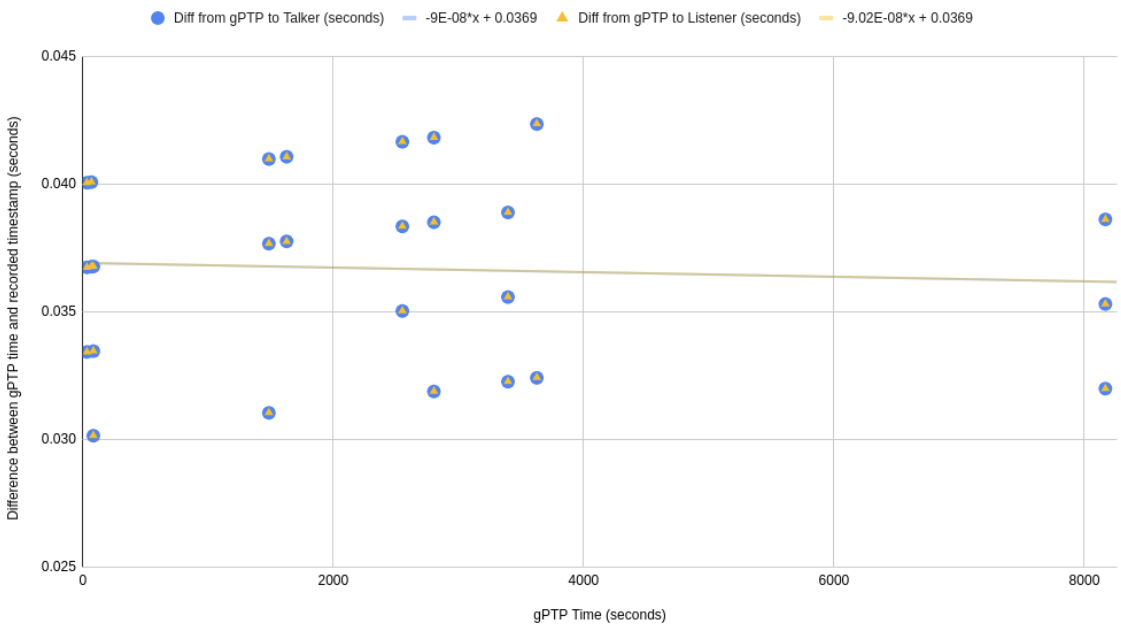


Figure 7.13: The difference between timestamps and gPTP time with lines of best fit.

As seen in the figure above, these two differences are essentially identical. Plotting the lines of best fit (shown in light blue and yellow) for these recorded differences shows that too, with the only difference between the two being the value of m, which differs by 0.02×10^{-8}. Using the line of best fit suggests that the difference between timestamps and the gPTP time may eventually be zero. This is not realistic as there are network delays and delays where timestamps are spent in the DCFIFOs, and the difference cannot be zero. It does however confirm that the difference over time would not drift too significantly, and that the `CSGEN` module will continue to operate as expected over a longer period of time.

7.3.2.4 Summary of Timing and Delays

This section presented results from to a suite of experiments designed to test the timing, delays and effectiveness of operation of the design, and particularly the `CSGEN` module. The results showed that while networking conditions have an effect on transmission time between talker and listener, the design of the system is left largely unaffected, with network delays resulting in a equally larger delay to the time at which an error is first detected by the `CSGEN` module. The results also showed that the `CSGEN` module is able to regularly detect an error in phase between the talker and listener in a time period less than that of a 48 kHz clock.

These integration test results, along with those presented in Section 7.3.1 confirm that

the design of the modules in the system are sound, and that a closed-loop system can be implemented.

7.3.3 Closed Loop Results

This section presents and discusses the results of the experimentation described in Section 6.4.3. These experiments relate to modelling and refining the closed loop feedback system, which is designed to align the phase of the listener and talker media clocks. The section starts by investigating a scaling of a loop gain value. Taking the best performing loop gain value, adjustments are then made to the `phase_step` value in order to adjust the ratio between the calculated correction value and the degree to which it affects the phase of the listener media clock. Finally, an averaging filter is implemented to investigate how filtering the correction value may impact performance.

7.3.3.1 Adjustment of the Loop Gain

By adjusting the value of the loop gain (a scaling value applied to the calculated difference between related timestamps), the detectable differences of errors can be adjusted. The `lgocoeff` variable is used to multiply the error as follows:

$$correction = difference \times 2^{-lgcoeff} \tag{7.3}$$

where: $correction$ = The amount to adjust the internal phase reference, cnt_int, by
$difference$ = The difference between two related timestamps
$lgcoeff$ = The loop gain coefficient

By measuring the values of the minimum and maximum corrections (a measure of how drastically the correction algorithm adjusts the phase of the listener media clock), as well as the average value between the differences between successive phase differences (which gives the phase correction applied), the performance of each `lgcoeff` value can be measured. Table 7.10 below shows that by not adjusting the error, the listener waveform is corrected by smaller values. This is better, as smaller corrections would result in a waveform with less jitter. The average phase after settling, which for all results was roughly 16 degrees. This is investigated further in the next section, which looks into adjusting the phase step value.

Table 7.10: Results from changing the loop gain coefficient value.

lgceff	phase_step	Ave. phase diff. (settled)	Ave. correction	Max correction	Min correction
5	200	16.23904899	-0.003746397695	3.56	-5.83
4	400	16.07116147	-0.004079320113	3.35	-5.98
3	800	16.06367232	-0.0008757062147	3.51	-5.68
2	1600	16.02239316	0.001339031339	3.34	-5.57
1	3200	16.02544944	0.0008707865169	3.48	-5.74
0	6400	16.0304298	0.004613180516	3.44	-5.54

Figure 7.14 below shows the resulting phase difference for a loop gain coefficient of 4. While this result shows a "stepping" before settling, this does not occur with smaller values of `lgcoeff`. This is due to smaller correction values being filtered out by being divided down, and as a result when an error is eventually detected, it is larger in proportion to the phase step value, resulting in a greater phase correction to the point where the output is corrected out of phase. This further promotes not scaling the detected error value.

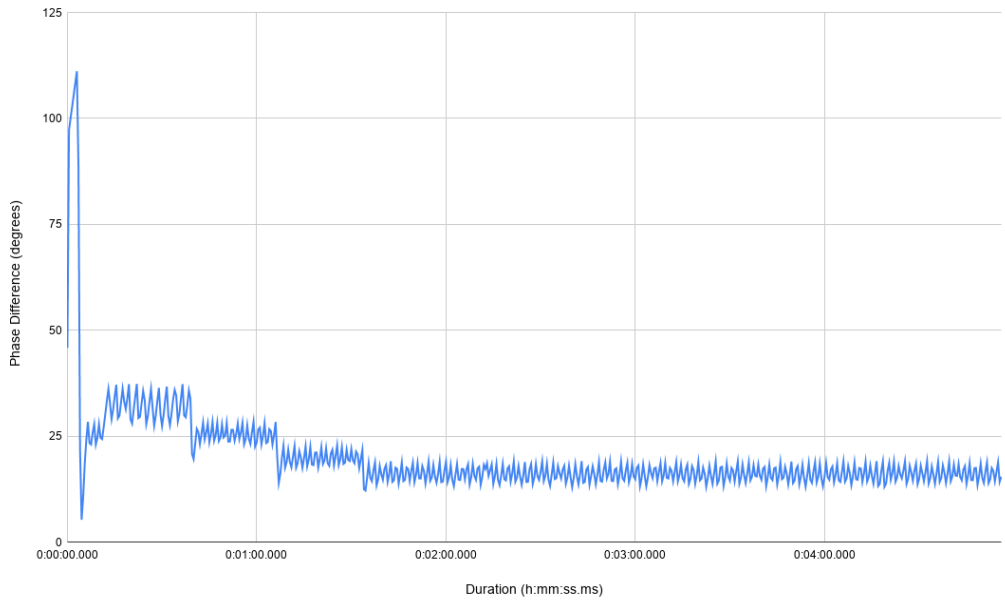


Figure 7.14: The phase difference between the talker and listener waveforms with lgcoeff=4 and phase_step=400.

7.3.3.2 Adjustment of the Phase Step Value

By adjusting the phase step value it is expected that the phase difference to which the correction algorithm corrects to can be adjusted. For any given correction factor calculated,

the relative phase degree of correction can be smaller or larger, depending on the value of count required to step one degree (i.e. larger phase step values will mean that calculated correction factors are relatively less impactful). By recording the average phase difference once the phase difference can be considered settled, the effectiveness of the phase step value as it relates to the loop gain coefficient can be determined. The results from these experiments are shown in Table 7.11 below. A phase step value of 1400 resulted in the system not settling.

Table 7.11: Results from changing the phase step value.

phase_step	**Ave phase diff. (settled)**	**Ave correction**	**Max corr**	**Min Corr**
6400	16.0304298	0.004613180516	3.44	-5.54
5240	12.60522727	-0.01198863636	2.99	-5.81
4080	9.366327684	0.01285310734	3.13	-5.41
3000	6.228228571	-0.01277142857	3.76	-5.75
2800	5.652988827	-0.0137150838	3.11	-5.53
2600	5.101085714	-0.009371428571	4.06	-6.17
2400	4.51877551	0.00915451895	3.27	-5.67
2200	3.920294118	0.002411764706	4.51	-6.5
1920	3.320478873	0.01335211268	8.99	-8.78
1400	Did not settle (DNS)	DNS	DNS	DNS
1600	3.162165242	-0.01333333333	15.89	-15.77

From the results recorded in Table 7.11 it can be seen that adjusting the `phase_step` value does affect the settling phase, as expected. The trade off to be made for settling values closer to zero is that of larger corrections, which implies more jitter in the generated media clock. A phase step value of 2800 was chosen for implementation due to it having the smallest range of maximum and minimum phase corrections at 3.11-(-5.53) = 8.64 degrees. The settling phase for this value is 5.65 degrees, well within the 9 degrees (or 5%) requirement of IEEE 1722. A plot showing the comparison between the original phase step value used in determining the value for the loop gain and the new phase step determined from this experimentation is shown in Figure 7.15 below.

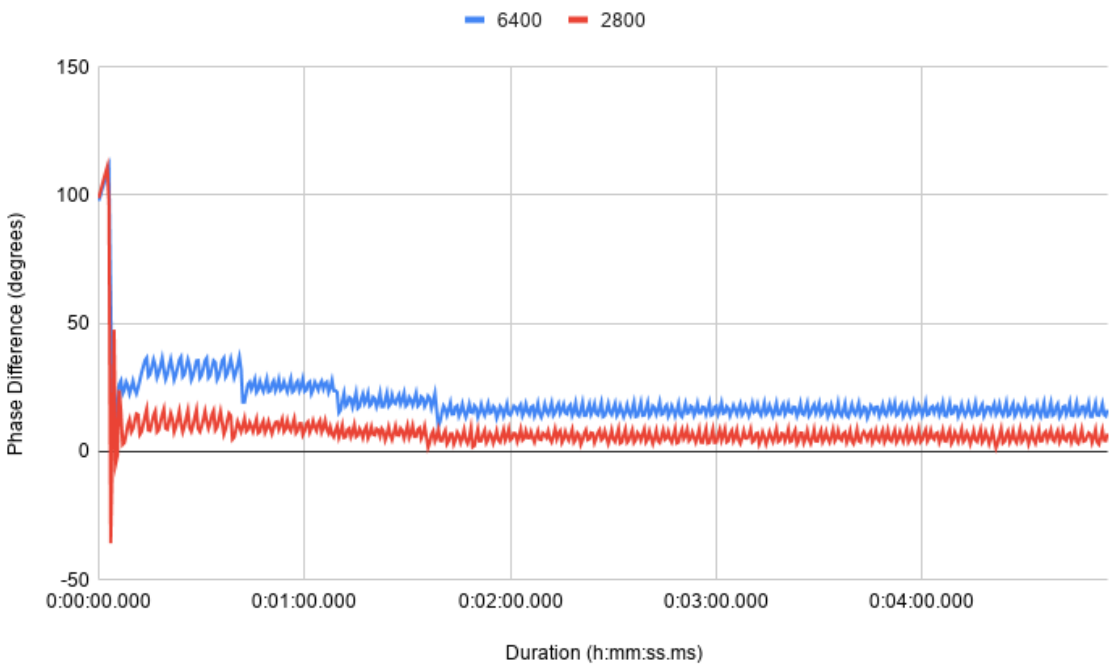


Figure 7.15: Resulting phase difference between talker and listener for phase_step values of 6400 and 2400.

A possible reason for the correction algorithm not correcting closer to a phase difference of zero is the phase drift in the system (which is directly tied to the environmental conditions of the oscillator), which results in a positive phase difference between the talker and listener. The experimentation to test this hypothesis is described in Section 6.5.

7.3.3.3 Addition of Averaging Filter

The averaging filter works as a low pass filter to prevent drastic changes in phase caused by large correction values. The result of adding an averaging filter is shown in Figure 7.16 below.

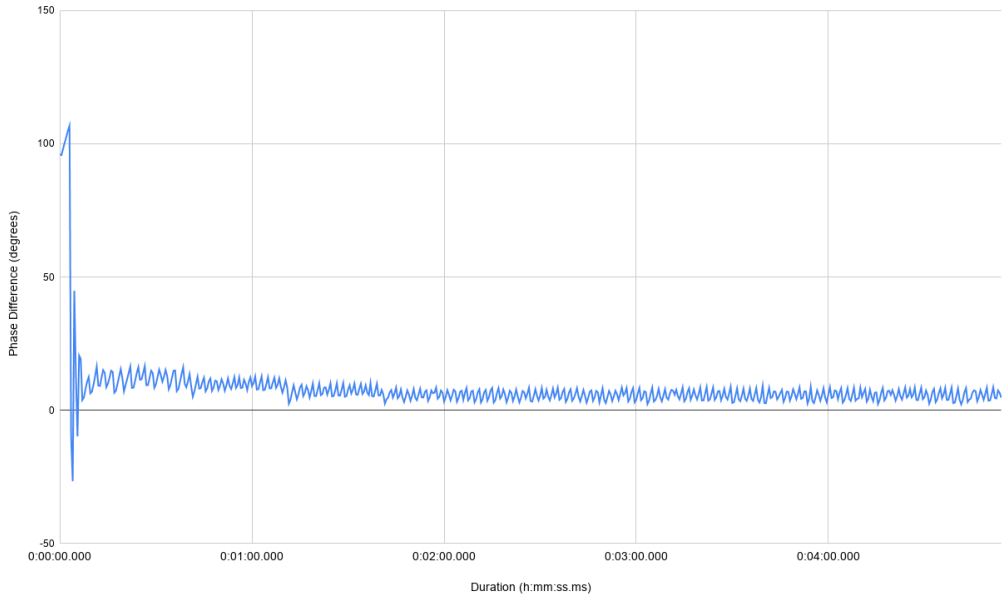


Figure 7.16: The output of the closed loop system with the addition of an averaging filter.

The goal of the averaging filter is to reduce the value of the average correction applied, reducing jitter. The average value of a phase correction made once settled is -0.005 degrees, less than half of the value without a filter, which justifies implementation of the averaging filter. However it can be seen by inspection that the resulting phase difference is still not smooth. This could be due to the sampling period of the BenchVue software (500 milliseconds). A better performing filter and better measurement equipment would be needed to be implemented to test this hypotheses.

With the addition of an averaging filter, the phase difference settles around 5.59 degrees out pf phase, with the listener waveform leading the talker waveform. The magnitude of the largest correction in this settled state is 6.4 degrees. Both these values are within the 5% phase difference required by IEEE 1722.

7.3.3.4 Summary of Results

Figure 7.17 shows a comparison to the phase difference over time without a correction, versus the phase difference over time with the final closed loop implementation.

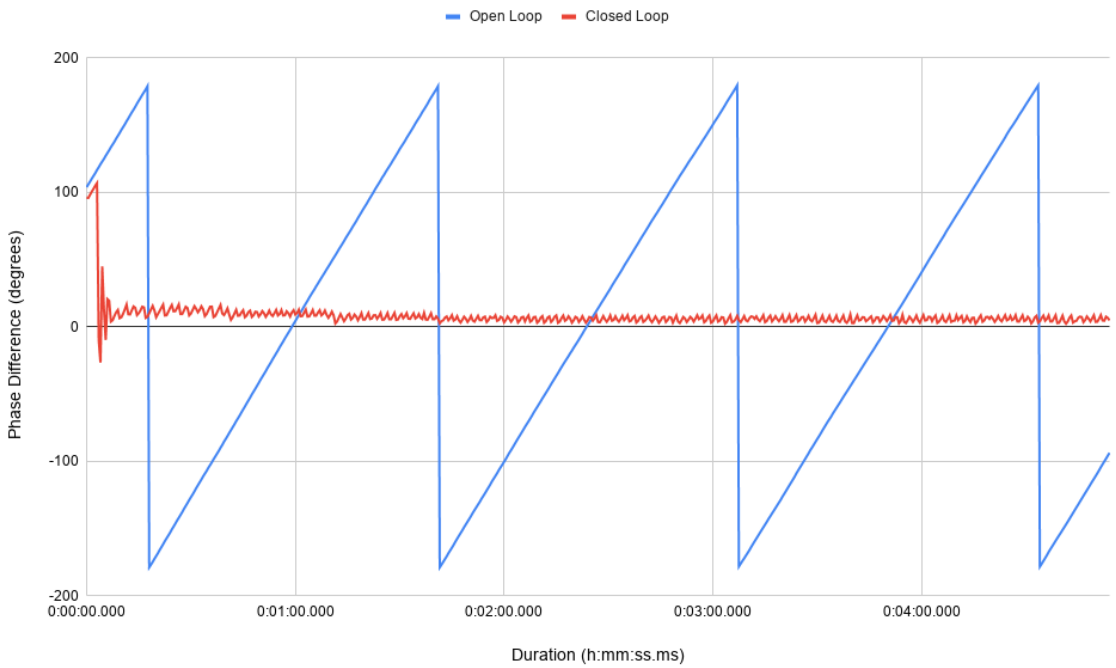


Figure 7.17: Comparison of system with and without the closed loop feedback system.

It can be seen that the performance of the closed loop system implemented does have the desired effect of maintaining a reasonably stable phase difference. The closed loop feedback system implemented in the `CSGEN` module is able to correct the listener media clock within the 5% range required for media presentation as described in IEEE 1722. There is some offset as to where the phase difference settles around, which is hypothesised to be due to the phase drift inherent within the system. The inclusion of an averaging filter does improve the performance of the system, however it comes at the cost of increasing the offset from the zero phase difference. This difference is marginal, and still within bounds. As a result, the averaging filter is included in the final implementation.

7.4 Stress Testing Results

This section covers the results of the experimentation discussed in Section 6.5, which tests how the system responds over time, and to test the system under irregular conditions.

7.4.1 Duration Test

The duration experiment tests how the system responds over a longer period of time than in the initial testing. Figure 7.18 shows the phase difference over a 30 minute run time.

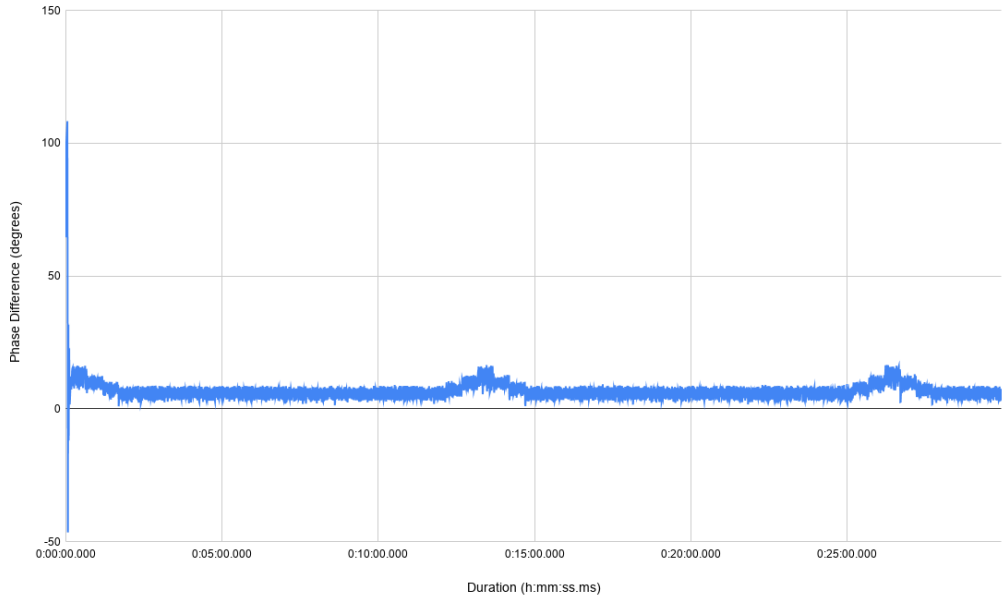


Figure 7.18: Phase difference between talker and listener over a duration of 30 minutes.

It is apparent from this graph that the system has periods when it does not perform valid corrections. The periodicity of these "stepped corrections" at roughly 13 minute intervals suggests a undetermined design flaw or logic error in the correction algorithm.

7.4.2 Heat Test

Figure 7.19 below shows the results of artificially increasing ambient temperature around the system using a hairdryer. The temperature measured on the heatsink of the FPGA, which is a good indication of the ambient temperature, was raised by roughly 20 degrees Celsius, from 42°C to 61°C.

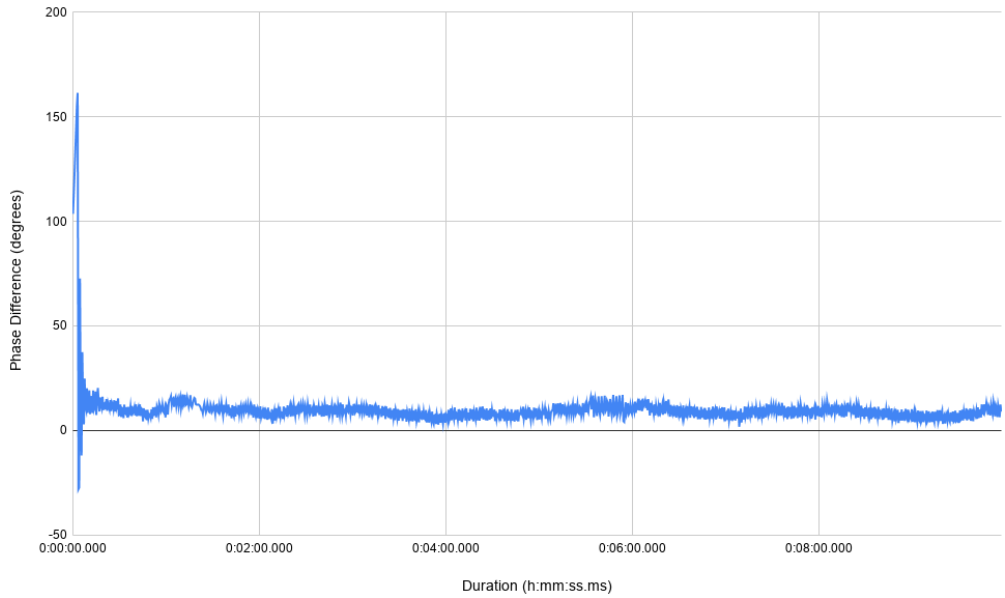


Figure 7.19: Phase difference between talker and listener in a warmer environment.

Given that the waveform does not settle as well as the results shown in Figure 7.16, the settling phase difference of this result has greater variation. Taking the time period at which the results in 7.16 and calculating the average phase from that point for the duration results in an average phase error of 8.77 degrees, 3.18 degrees greater than those shown in Figure 7.16. This confirms the hypothesis that temperature, which is directly correlated with the rate at which the phase of the listener drifts relative to that of the talker, is responsible for the settling phase difference value.

7.4.3 Network Conditions Test

After running the system for two to three minutes without failure, the network cable was removed. The cable was left disconnected for arbitrary periods of time, ranging from 30 seconds to five minutes. After reconnecting the cable, the time for recovery is under three seconds, regardless of duration of lack of network connectivity. This is in-line with the system start-up time recorded in Tables 7.8 and 7.9 in Section 7.3.2.

7.4.4 Summary of Stress Testing Results

The results of the three tests indicate that operation of the system holds up to changes in network conditions, but that there is an undetermined design flaw or logic error causing offset correction factors. These offset correction factors and the observed periodic "stepping" are also observed to be sensitive to environmental conditions such as temperature. Considerations of these factors are discussed further in Section 8.2.

Chapter 8

Conclusion

The work presented in this dissertation was designed to implement and test a CRF talker and listener pair using design aspects drawn from IEEE 1722, to perform a media clock correction and to synchronise two media clocks over Ethernet. The objective of this work was to present a design that could be used in the automotive industry, particularly relating to the category of infotainment, discussed in Section 1.1 (and elaborated upon in Table 1.1), with a focus on audio applications. It is intended that this work could be further extended to implement clock correction for time-critical data using Ethernet connectivity as a means of cost, weight and complexity reduction in various systems.

The next section proceeds with a summary of the project, recapping the main objectives, as well as the development and experimentation tasks performed. Next, the extent to which the requirements that were specified in Section 1.3 are discussed, along with a discussion of the effectiveness of the implementation in achieving these requirements. The chapter ends with recommendations for future work that were identified during the process of development and testing of the project.

8.1 Summary of the Dissertation

The project presented in this dissertation involved development and testing of a system to align the phase of two asynchronous 48 kHz media clocks using timestamps transmitted over Ethernet. Literature relating the relevant standards as well as the fields where this application may be considered useful is investigated in Chapter 2. This research was used to develop the requirements presented in Section 1.3. A design methodology and the approaches used in the dissertation are presented in Chapter 3. The design implemented is presented in Chapter 4,

with details of the design as well as operation presented in Chapter 5. Chapter 6 presents a set of experiments used in verifying the design. The results of this experimentation are presented in Chapter 7. These results demonstrate that the system design succeeded in aligning the phases of the clock within an acceptable margin, though this is impacted by environmental conditions, most notably of which is temperature. Section 8.3 makes the recommendation to mitigate for these effects and retest the implementation.

8.2 Review of Requirements

This section discusses the degree to which the requirements of the project specified in Section 1.3 were met, and elaborates upon the discussion points presented in the Chapter 7, which presented the results from testing the implementation.

Table 8.1 below summarises where the requirements presented in Section 1.3 were verified as being correctly implemented, as well as the degree to which they were met.

Table 8.1: The requirements of the project and the results to verify that they were met.

Requirement	**Description**	**Module**	**Results**	**Requirement Met**
R01	A shared timing reference between both talker and listener	gPTP	7.2.1	Yes
R02	Create a media clock of 48kHz to use as a source clock	gPTP	7.2.1	Yes
R03	Generate timestamps using the timing reference in R01 and source media clock in R02	gPTP	7.2.1	Yes
R04	Package the generated source timestamps in a format suitable for transmission over Ethernet, as described by IEEE 1722	CRFFrameGen	7.2.2	Yes
R05	Transmit the packaged timestamps over Gigabit Ethernet in a way that makes it possible to observe them over the wire through an application such as Wireshark	CRFFrameGen	7.2.2	Yes
R06	Generate a media clock of 48kHz which can be adjusted in phase in order to match up to the media clock in R02	CSGEN (Open Loop)	7.2.4	Yes
R07	Generate timestamps of the generated media clock in R06 using the shared timing reference in R01	GEN DCFIFO Driver	7.2.5	Yes
R08	Unpack the formatted source timestamps received over Ethernet and extract the relevant information	ETHRX and the AVTPDU Processor	7.2.3	Yes
R09	Use the unpacked timestamps and compare them to the generated media clock's timestamps in order to determine a phase difference	CSGEN (Closed Loop)	7.3.3	Yes
R10	Use the phase difference in order to adjust the generated media clock to be in phase with the source media clock	CSGEN (Closed Loop)	7.3.3	Partial
R11	IEEE 1722 standard and requirements should be implemented as much as reasonably possible	See table 8.2 below		

Table 8.2 below shows the aspects of CRF requirements that were chosen to be implemented in this project, and whether or not the implementation was successful.

Table 8.2: The requirements of IEEE 1722 and the results to verify that they were met.

Item	Requirement	Results	Requirement Met
CRF-1	Does the device use the alternative header?	7.2.2	Yes
CRF-2	Is the subtype field set to CRF?	7.2.2	Yes
CRF-3	Is the mr bit set as described in Section 10.4.3 of IEEE 1722?	7.2.2	Yes
CRF-5	Does the mr bit remain in its new state for a minimum of 8 CRF AVTPDUs?	7.2.2	Yes
CRF-6	Is the sequence_num field increment by one (1) with wrapping?	7.2.2	Yes
CRF-8	Is the base_frequency_field set to a value from 1 to 536 870 911 (1FFFFFFF16)?	7.2.2	Yes
CRF-9	Is the crf_data_length field value a multiple of 8 octets?	7.2.2	Yes
CRF-10	Is the timestamp_interval nonzero?	7.2.2	Yes
CRF-11	Is the fs bit set to zero (0) for CRF AVTPDUs with a type field set to CRF_AUDIO_SAMPLE?	7.2.2	Yes
CRF-15	Is a CRF Audio Listener capable of receiving CRF AVTPDUs at the rates given in IEEE 1722 Table 28?	7.3.2.3	Yes
CRF-20	In a CRF AVTPDU that contains multiple CRF timestamps, do the values of the timestamps increase monotonically, and with a constant nominal spacing in time, from the first timestamp to the last timestamp in the CRF AVTPDU when tu equals zero (0)?	7.2.1	Yes
CRF-21	In a CRF AVTPDU that contains multiple CRF timestamps, are all timestamps derived from a continuous gPTP clock reference?	7.2.1	Yes

8.2.1 Discussion

As shown in Tables 8.1 and 8.2, the implemented design was able to fulfil the requirements given in Section 1.3. Discussion stemming from the results of each experiment is done in each relevant results section, but this section will consolidate those and focus on the implementation as a whole, as well as implications of the findings resulting from the experimentation.

The results of the Python simulation presented in Section 7.1 indicated that the devised approach for capturing and comparing timestamps in order to correct the listener waveform was well-suited to the application. The modules developed for capturing, transmitting and comparing timestamps were shown to be reliable and sound in design due to regularly and reliably captured timestamps being able to be processed by the `CSGEN` module at periodic, predetermined intervals. The system does display unpredicted behaviour when running for longer periods of time. The "phase stepping", shown clearly in Figure 7.18 in Section 7.4, indicates a problem in the design of the closed feedback loop in the `CSGEN` module, as all

module and integration testing up to that point showed results indicative of a robust, reliable design. This stepping effect could be the result of not catering for a phase drift in the system, which testing indicated heavily impacted the efficacy of the closed loop phase correction algorithm. As a result of this, Requirement R10 is considered to only be partially met, in the sense that it is able to calculate a phase difference and correct the listener media clock, but it is not able to do so with complete efficacy to the point that a zero-degree phase error remains.

8.3 Future Work

This section discusses additional future work and experimentation which can be done to improve the design presented in this dissertation, as well as additional testing to determine the efficacy of the system and to determine where the root of the problem found in Section 7.4 lie.

8.3.1 Additional Experimentation for Modelling

Equipment limitations meant that a holistic modelling and measurement of the system could not be done. For example, there was no reliable way of measuring jitter on either talker or listener waveform, and the software used to capture data from the oscilloscope was limited to a sampling period of 500 ms, which did not work reliably at times.

8.3.2 Design Improvements

There are a lot of features of the CRF standard presented by IEEE 1722 that go unused or unimplemented in the presented design. While not all of them are relevant to the audio application presented as a use case in this dissertation, there are some flags which would serve a purpose in systems which implements multiple talkers and listeners. Notably there are also some features which are implemented on the talker side by the module responsible for generating the CRF AVTPDUs which currently go unused on the listener. These are as follows:

- `MR` is extracted but unused.
- `TU` is extracted but unused.
- `sequence_num` is extracted but unused.

- `stream_id` is extracted but unused.
- `pull` and `base_frequency` are both extracted but unused.
- `timestamp_interval` is unused.

Many of these features are only useful when implementing multiple listeners on the network, or when developing a listener which may be able to process a range of talker and listener frequencies. `MR` and `TU` would be useful to implement as a way of notifying the listener when a talker was unable to transmit timestamp data or the network connection failed, a response to which would be to cause the listener to go into a holdover mode to maintain the last well known relative phase difference. Given that there was no means of compensating for the phase drift and enabling holdover functionality, a test beyond verifying that these flags were correctly extracted would serve no purpose in this design.

Another design improvement would be to process timestamps less often once a lock between talker and listener was obtained. This is suggested in IEEE 1722 and mentioned briefly in Section 2.2.3 of this dissertation, but as shown in the results, a zero phase error lock was not obtained - presumably due to the phase drift in the system. The logic could be implemented when a threshold considered to be "close enough" to zero phase error is reached, such as the 5% phase difference suggested by IEEE 1722.

Finally, additional filtering techniques can be applied to the closed loop system, with regards to both the phase correction algorithm and the low pass filter applied to the correction values. The methods should be compared and contrasted, with the best techniques applied to the system.

8.3.3 Implementation of Holdover and Phase Drift Compensation

An algorithm for compensating against the phase drift inherent to the system would prove highly beneficial. It was shown that the system is sensitive to temperature, which adjusts the phase drift of the system and affects performance. Attempts were made at implementing control logic to measure phase drift over time and compensate for it when appropriate, but these algorithms were proven ineffective. Implementation of an effective phase drift compensation algorithm would result in better performance of the closed loop system, and enable the system to maintain the last best known position when the system fails to perform corrections due to problems such as network connectivity errors.

Bibliography

[1] N. Ilyadis, "Challenges of Autonomous Vehicles : How Ethernet in Automobiles Can Overcome Bandwidth Issues in Self-Driving Vehicles," 4 2017, Date accessed: 26/10/2020. [Online]. Available: https://web.archive.org/web/20180507094139/http://blogs.marvell.com/2017/04/challenges-of-autonomous-vehicles-how-ethernet-in-automobiles-can-overcome-bandwidth-issues-in-self-driving-vehicles/

[2] J. Guo, B. Song, Y. He, F. R. Yu, and M. Sookhak, "A survey on compressed sensing in vehicular infotainment systems," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 4, pp. 2662–2680, 2017. [Online]. Available: https://doi.org/10.1109/comst.2017.2705027

[3] F. L. Soares, D. R. Campelo, Y. Yan, S. Ruepp, L. Dittmann, and L. Ellegard, "Reliability in automotive ethernet networks," in *2015 11th International Conference on the Design of Reliable Communication Networks (DRCN)*. IEEE, Mar. 2015. [Online]. Available: https://doi.org/10.1109/drcn.2015.7148990

[4] A. Sarkar, "Bentley Bentayga Electrical System is Simply Amazing," pp. 1–4, 2016. [Online]. Available: http://www.driversmagazine.com/bentley-bentayga-electrical-system-is-simply-amazing/

[5] B. Kraemer, "Automotive ethernet," *IEEE Communications Magazine*, vol. 54, no. 12, pp. 4–4, Dec. 2016. [Online]. Available: https://doi.org/10.1109/m-com.2016.7785879

[6] M. J. Teener, "A Time-Sensitive Networking Primer: Putting It All Together," *International IEEE Symposium on Precision Clock Synchronization for Measurement, Control and Communication*, pp. 1–49, 2015.

[7] T. R. Shapiro, "Under CEO's guidance, Motorola made history," *The Washington Post*, Oct. 2011.

[8] B. Karin, C. Eefje, K. Stefan, and M. Gijs, *Sound and Safe : A History of Listening Behind the Wheel.* Oxford University Press, 2013. [Online]. Available: https://doi.org/10.1093/acprof:oso/9780199925698.001.0001

[9] N. Navet, Y. Song, F. Simonot-Lion, and C. Wilwert, "Trends in automotive communication systems," *Proceedings of the IEEE*, vol. 93, no. 6, pp. 1204–1223, Jun. 2005. [Online]. Available: https://doi.org/10.1109/jproc.2005.849725

[10] F. P. Rezha, O. D. Saputra, and S. Y. Shin, "Extending CAN bus with ISA100.11a wireless network," *Computer Standards & Interfaces*, vol. 42, pp. 32–41, Nov. 2015. [Online]. Available: https://doi.org/10.1016/j.csi.2015.04.002

[11] I. Park and M. Sunwoo, "FlexRay network parameter optimization method for automotive applications," *IEEE Transactions on Industrial Electronics*, vol. 58, no. 4, pp. 1449–1459, Apr. 2011. [Online]. Available: https://doi.org/10.1109/tie.2010.2049713

[12] T. Pop, P. Pop, P. Eles, Z. Peng, and A. Andrei, "Timing analysis of the FlexRay communication protocol," *Real-Time Systems*, vol. 39, no. 1-3, pp. 205–235, Oct. 2007. [Online]. Available: https://doi.org/10.1007/s11241-007-9040-3

[13] S. Tuohy, M. Glavin, C. Hughes, E. Jones, M. Trivedi, and L. Kilmartin, "Intra-vehicle networks: A review," *IEEE Transactions on Intelligent Transportation Systems*, vol. 16, no. 2, pp. 534–545, Apr. 2015. [Online]. Available: https://doi.org/10.1109/tits.2014.2320605

[14] Z. Sheng, D. Tian, V. C. M. Leung, and G. Bansal, "Delay analysis and time-critical protocol design for in-vehicle power line communication systems," *IEEE Transactions on Vehicular Technology*, vol. 67, no. 1, pp. 3–16, Jan. 2018. [Online]. Available: https://doi.org/10.1109/tvt.2017.2770182

[15] H. Muyshondt, "Media oriented systems transport (most): a high-speed multimedia and control bus," *ECN (Radnor, Pa.)*, vol. 50, no. 9, pp. 25–, 2006.

[16] "Internet connection speed recommendations," Date Accessed: 26/10/2020. [Online]. Available: https://help.netflix.com/en/node/306

[17] N. Navet, "A journey into time-triggered communication protocols with a focus on ethernet tsn," 2018. [Online]. Available: https://orbilu.uni.lu/handle/10993/37456

[18] A. Amari and A. Mifdaoui, "Specification and performance indicators of AeroRing—a multiple-ring ethernet network for avionics embedded systems," *Sensors*, vol. 18, no. 11, p. 3871, Nov. 2018. [Online]. Available: https://doi.org/10.3390/s18113871

[19] L. Zhao, F. He, E. Li, and J. Lu, "Comparison of time sensitive networking (TSN) and TTEthernet," in *2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC)*. IEEE, Sep. 2018. [Online]. Available: https://doi.org/10.1109/dasc.2018.8569454

[20] *THE WHITE RABBIT PROJECT*. 2nd International Beam Instrumentation Conference, 2013. [Online]. Available: https://cds.cern.ch/record/1743073

[21] M. Rizzi, M. Lipinski, P. Ferrari, S. Rinaldi, and A. Flammini, "White rabbit clock synchronization: Ultimate limits on close-in phase noise and short-term stability due to FPGA implementation," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, vol. 65, no. 9, pp. 1726–1737, Sep. 2018. [Online]. Available: https://doi.org/10.1109/tuffc.2018.2851842

[22] M. Jimenez-Lopez, F. Girela-Lopez, J. Lopez-Jimenez, E. Marin-Lopez, R. Rodriguez, and J. Diaz, "10 gigabit white rabbit: Sub-nanosecond timing and data distribution," *IEEE Access*, vol. 8, pp. 92 999–93 010, 2020. [Online]. Available: https://doi.org/10.1109/access.2020.2995179

[23] J. Farkas, L. L. Bello, and C. Gunther, "Time-sensitive networking standards," *IEEE Communications Standards Magazine*, vol. 2, no. 2, pp. 20–21, Jun. 2018. [Online]. Available: https://doi.org/10.1109/mcomstd.2018.8412457

[24] D. Thiele, J. Schlatow, P. Axer, and R. Ernst, "Formal timing analysis of CAN-to-ethernet gateway strategies in automotive networks," *Real-Time Systems*, vol. 52, no. 1, pp. 88–112, Oct. 2015. [Online]. Available: https://doi.org/10.1007/s11241-015-9243-y

[25] IEEE, *IEEE Standard for a Transport Protocol for Time-Sensitive Applications in Bridged Local Area Networks*, 2016th ed. IEEE, 2016, vol. 2016, no. June. [Online]. Available: https://doi.org/10.1109/ieeestd.2016.7782716

[26] Y. Li, D. Li, W. Cui, and R. Zhang, "Research based on OSI model," in *2011 IEEE 3rd International Conference on Communication Software and Networks*. IEEE, May 2011. [Online]. Available: https://doi.org/10.1109/iccsn.2011.6014631

[27] A. Dalal, N. Digrase, and R. Lanjewar, "Radio frequency design of fast locking digital phase locked loop," in *2015 International Conference on Computing Communication Control and Automation*. IEEE, Feb. 2015. [Online]. Available: https://doi.org/10.1109/iccubea.2015.187

[28] J. Dunning, G. Garcia, J. Lundberg, and E. Nuckolls, "An all-digital phase-locked loop with 50-cycle lock time suitable for high-performance microprocessors," *IEEE Journal of Solid-State Circuits*, vol. 30, no. 4, pp. 412–422, Apr. 1995. [Online]. Available: https://doi.org/10.1109/4.375961

[29] Altera, *ALTPLL (Phase-Locked Loop) IP Core User Guide ug-altpll*. Altera, 2017. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/altera_pll.pdf

[30] D. R. Martinez, M. M. Vai, and R. A. Bond, *High performance embedded computing handbook: A systems perspective*. CRC Press, 2008.

[31] M. A. Cherney, "AMD Agrees to Buy Xilinx for $35 Billion in Latest Effort to Take on Rival Intel. Its Stock Is Dropping." Oct 2020, Date Accessed: 29/10/2020. [Online]. Available: https://www.barrons.com/articles/amd-agrees-to-buy-xilinx-for-35-billion-in-effort-to-take-on-rival-intel-51603794660

[32] N. Campregher, P. Y. K. Cheung, G. A. Constantinides, and M. Vasilko, "Yield enhancements of design-specific FPGAs," in *Proceedings of the internation symposium on Field programmable gate arrays - FPGA '06*. ACM Press, 2006. [Online]. Available: https://doi.org/10.1145/1117201.1117215

[33] C. E. Cummings, "Clock domain crossing (cdc) design & verification techniques using system verilog," *SNUG-2008, Boston*, 2008.

[34] S. Friedrichs, M. Fugger, and C. Lenzen, "Metastability-containing circuits," *IEEE Transactions on Computers*, vol. 67, no. 8, pp. 1167–1183, Aug. 2018. [Online]. Available: https://doi.org/10.1109/tc.2018.2808185

[35] "Mercury+ SA2," 2017, Date Accessed=28/09/2020. [Online]. Available: https://web.archive.org/web/20170121174104/https://www.enclustra.com/en/products/system-on-chip-modules/mercury-sa2/

[36] Intel, *Triple-Speed Ethernet Intel FPGA IP User Guide*, 17th ed. Intel, 2019. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_ethernet.pdf

[37] ——, *FIFO Intel® FPGA IP User Guide*, 18th ed. Intel, 2018. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_fifo.pdf

[38] R. Martin, "Design thinking: achieving insights via the "knowledge funnel"," *Strategy & Leadership*, vol. 38, no. 2, pp. 37–41, Mar. 2010. [Online]. Available: https://doi.org/10.1108/10878571011029046

[39] B. W. Boehm, "A spiral model of software development and enhancement," *Computer*, vol. 21, no. 5, pp. 61–72, May 1988. [Online]. Available: https://doi.org/10.1109/2.59

[40] IEEE, *IEEE Std 1588-2008, IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*. IEEE, 2008, vol. 2008, no. July. [Online]. Available: https://doi.org/10.1109/ieeestd.2020.9120376

[41] *Fractional-N Clock Synthesizer & Clock Multiplier*, Cirrus Logic, 9 201, dS761F3. [Online]. Available: https://statics.cirrus.com/pubs/proDatasheet/CS2000-CP_F3.pdf

Appendix A

CRF Protocol Implementation Conformance

The full list of requirements for IEEE 1722 CRF is shown in Figure A.1 on the following page.

F.12 Clock Reference Format

Table F.16—Clock Reference Format

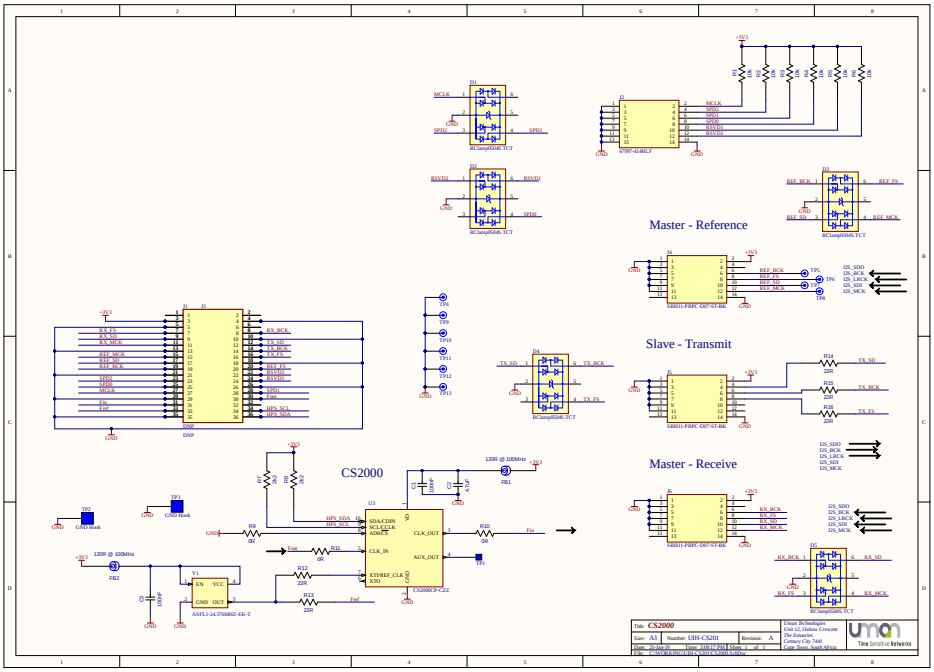
Item	Feature	Status	References	Support
CRF-1	Does the device use the alternative header?	CRF:M	4.4.6	Yes [ ]
CRF-2	Is the **subtype** field set to CRF?	CRF:M	4.4.3.2	Yes [ ]
CRF-3	Is the **mr** bit set as described in 10.4.3?	CRF:M	10.4.3	Yes [ ]
CRF-4	Does a CRF Listener that is also a media stream Talker toggle the **mr** bit in outgoing streams as described in 10.4.3?	CRF:M	10.4.3	Yes [ ]
CRF-5	Does the **mr** bit remain in its new state for a minimum of 8 CRF AVTPDUs?	CRF:M	10.4.3	Yes [ ]
CRF-6	Is the **sequence_num** field increment by one (1) with wrapping?	CRF:M	10.4.6	Yes [ ]
CRF-7	Is a Listener able to tolerate a starting **sequence_num** of any value?	CRF:M	10.4.6	Yes [ ]
CRF-8	Is the **base_frequency_field** set to a value from 1 to 536 870 911 ($1FFFFFFF_{16}$)?	CRF:M	10.4.10	Yes [ ]
CRF-9	Is the **crf_data_length** field value a multiple of 8 octets?	CRF:M	10.4.11	Yes [ ]
CRF-10	Is the **timestamp_interval** nonzero?	CRF:M	10.4.12	Yes [ ]
CRF-11	Is the **fs** bit set to zero (0) for CRF AVTPDUs with a **type** field set to CRF_AUDIO_SAMPLE?	CRF:M	10.4.13.3	Yes [ ]
CRF-12	Is the **fs** bit set to zero (0) for CRF AVTPDUs with a **type** field set to CRF_VIDEO_SAMPLE?	CRF:M	10.4.13.4	Yes [ ]
CRF-13	Is the **fs** set to one (1) for CRF AVTPDUs with a **type** field set to CRF_VIDEO_LINE and the timestamp represents the first line of a video frame?	CRF:M	10.4.13.5	Yes [ ]
CRF-14	Is the **fs** set to zero (0) for CRF AVTPDUs with a **type** field set to CRF_VIDEO_LINE and the timestamp does not represent the first line of a video frame?	CRF:M	10.4.13.5	Yes [ ]
CRF-15	Is a CRF Audio Listener capable of receiving CRF AVTPDUs at the rates given in Table 28?	CRF:O	10.5	Yes [ ] No [ ]
CRF-16	Is a CRF Video Frame Listener capable of receiving CRF AVTPDUs at the rates given in Table 29?	CRF:O	10.5	Yes [ ] No [ ]
CRF-17	Is a CRF Video Line Listener capable of receiving CRF AVTPDUs at the rates given in Table 30?	CRF:O	10.5	Yes [ ] No [ ]
CRF-18	Is a CRF Video Line Listener capable of receiving CRF AVTPDUs at the rates given in Table 31?	CRF:O	10.5	Yes [ ] No [ ]
CRF-19	Is the CRF timestamp contained in the **crf_data** field set to a value which is offset, in positive nanoseconds, from the transmit reference plane at the Talker by the Max Transit Time of the network, rounded up to the nearest multiple of the media clock as shown in Equation (14)?	CRF:M	10.7	Yes [ ]
CRF-20	In a CRF AVTPDU that contains multiple CRF timestamps, do the values of the timestamps increase monotonically, and with a constant nominal spacing in time, from the first timestamp to the last timestamp in the CRF AVTPDU when **tu** equals zero (0)?	CRF:M	10.7	Yes [ ]
CRF-21	In a CRF AVTPDU that contains multiple CRF timestamps, are all timestamps derived from a continuous gPTP clock reference?	CRF:M	10.7	Yes [ ]
CRF-22	Are all **avtp_timestamps** sent by CRF Talkers within +/− 5.0% of the media sample period of the received CRF stream?	CRF:M	10.8 Equation (15)	Yes [ ]

Figure A.1: Full list of requirements for IEEE 1722 CRF

Appendix B

The CS2000 Integrated Circuit

B.1 CS2000 Circuit Diagram



B.2 CS2000 Configuration

The CS2000 is configured over I2C upon boot of the on-board microcontroller on the Mercury SA2+ development board. The configuration is done by the following Python script shown in listing B.1:

Listing B.1: The CS2000 configuration script

```
1  import i2c
2
3  CHIP = 0x4e  # The I2C address
4
5  # Define the registers used and the values
6  regs = (
7      (3, 1),  # Device configuration 1
8      (3, 7),  # Aux_out (un)lock status indicator
9      (4, 1),  # Dynamic Ratio
10     (5, 1),  # EnDevCfg2
11     (6, 6),  # Ratio MSB
12     (0x16, 0x80) # Refclk div
13 )
14
15 # Write the registers
16 for s in regs:
17     i2c.writecs(CHIP, s[0], s[1])
```

Table B.1 below describes each register in the above script and the relevant bits used in configuration:

Table B.1: The configuration details of the CS2000 chip

Register	Bits	Description	Value Set	Effect
3	[0]	Enable Device Config 1	1	Enables device for the control port mode
3	[2:1]	Auxiliary Output Source Selection	3	Assigns the PLL Lock Signal to the auxilliary output, accessible on the board through TP1
4	[0]	Fractional-N Source for Frequency Synthesizer	1	Dynamic Ratio from Digital PLL for Hybrid (analog-digital) PLL Mode (see 4.2 on pg 12)
5	[0]	Enable Device Config 2	1	Works with EnDevCfg to configure the device for control port mode
6	[7:0]	Ratio 0, [31:24]	6	Sets the user defined ratio to be 24576
22	[7]	Clock Skip Enable	1	Enables clock skipping mode for the PLL and allows the PLL to maintain lock even when the CLK_IN has missing pulses

These register values are set in order to configure the CS2000 chip to operate as desired. The device configuration is enabled through writing to registers 3 and 5. The lock signal from the phase-locked loop is assigned to the auxiliary pin, and is made available on the development board through test point 1. The dynamic mode of operation (setting the output frequency through control of a digital PLL from an input signal) is set in register 5. Register 22 ensures that when there is no input from the control clock (`CLK_IN`), the output signal remains at it's most recently adjusted value.

Register 6 sets the eight most significant bits of the first user defined ratio between the input timing reference and the output clock. User defined ratios are stored in 32 bit fixed-point format. In the used configuration, 20 bits are used for the integer component, and 12 for the fractional component. With the defined register values set, the user defined ratio is equal to 24756. This results in a clock scaling of 1000, as the input timing reference is driven by a 24.576 MHz crystal oscillator.

Appendix C

Python Simulation

The third attempt used in testing the Python implementation can be found at https://github.com/kcranky/CRFSim/blob/master/CRF.py.

Appendix D

VHDL Code

The code for the modules descirbed in Chapter 5 can be found at this GitHub link: https://github.com/kcranky/IEEE1722CRFImplementation

Appendix E

Recorded Data

E.1 Consecutive Timestamps Captured

Table E.1: Successive timestamps captured from Wireshark.

Timestamp	Value (HEX)	ns Value (DEC)
196.0	00000000E711A0F8	3876692216
196.1	00000000E7447DC8	3880025544
196.2	00000000E7775AA0	3883358880
196.3	00000000E7AA3770	3886692208
196.4	00000000E7DD1448	3890025544
196.5	00000000E80FF128	3893358888
197.0	00000000e842cdf0	3896692208
197.1	00000000e875aac8	3900025544
197.2	00000000e8a88798	3903358872
197.3	00000000e8db6470	3906692208
197.4	00000000e90e4140	3910025536
197.5	00000000e9411e18	3913358872
198.0	00000000e973faf8	3916692216
198.1	00000000e9a6d7c0	3920025536
198.2	00000000e9d9b498	3923358872
198.3	00000000ea0c9168	3926692200
198.4	00000000ea3f6e40	3930025536
198.5	00000000ea724b10	3933358864
199.0	00000000eaa527e8	3936692200

199.1	00000000ead804c8	3940025544
199.2	00000000eb0ae190	3943358864
199.3	00000000eb3dbe68	3946692200
199.4	00000000eb709b38	3950025528
199.5	00000000eba37810	3953358864
200.0	00000000ebd654e0	3956692192
200.1	00000000ec0931b8	3960025528
200.2	00000000ec3c0e90	3963358864
200.3	00000000ec6eeb60	3966692192
200.4	00000000eca1c838	3970025528
200.5	00000000ecd4a508	3973358856
200	00000000ebd654e0	3956692192
201	00000000ed0781e0	3976692192
202	00000000ee38aed8	3996692184
203	00000000ef69dbd8	4016692184
204	00000000f09b08d0	4036692176
205	00000000f1cc35d8	4056692184
206	00000000f2fd62c8	4076692168
207	00000000f42e8fc0	4096692160
208	00000000f55fbcc0	4116692160
209	00000000f690e9b8	4136692152
210	00000000f7c216b0	4156692144
211	00000000f8f343b0	4176692144
212	00000000fa2470a8	4196692136
213	00000000fb559da8	4216692136
214	00000000fc86caa0	4236692128
215	00000000fdb7f7a0	4256692128
216	00000000fee92498	4276692120
217	00000001001a5198	4296692120
218	00000001014b7e90	4316692112
219	00000001027cab88	4336692104
220	0000000103add888	4356692104
221	0000000104df0580	4376692096

E.2 Rate of Use of Timestamps

Table E.2: Comparing the talker and listener timestamps to the gPTP time when a phase error is detected.

Capture	Duration into capture (nS)	Talker Timestamp	Listener Timestamp	gPTP Time	gPTP - Talker	gPTP - Listener
1	10280	32483351680	32483349600	32523396056	40044376	40046456
1	29480	32486685008	32486682936	32523415256	36730248	36732320
1	48680	32490018344	32490016264	32523434456	33416112	33418192
2	10280	67943342952	67943328776	67983410256	40067304	40081480
2	29480	67946676288	67946662104	67983429456	36753168	36767352
3	29480	83566672480	83566662104	83603435696	36763216	36773592
3	48680	83570005816	83569995440	83603454896	33449080	33459456
3	67880	83573339144	83573328776	83603474096	30134952	30145320
4	10280	1485803005152	1485802995448	1485843977376	40972224	40981928
4	29480	1485806338488	1485806328768	1485843996576	37658088	37667808
4	67880	1485813005152	1485812995448	1485844034976	31029824	31039528
5	10280	1627682971976	1627682974600	1627724034256	41062280	41059656
5	29480	1627686305312	1627686307936	1627724053456	37748144	37745520
6	10280	2552082755768	2552082745464	2552124403896	41648128	41658432
6	29480	2552086089112	2552086078768	2552124423096	38333984	38344328
6	48680	2552089422432	2552089412104	2552124442296	35019864	35030192
7	10280	2803942694384	2803942682936	2803984504656	41810272	41821720
7	29480	2803946027720	2803946016272	2803984523856	38496136	38507584
7	67880	2803952694384	2803952682936	2803984562256	31867872	31879320
8	29480	3395685880528	3395685870432	3395724760536	38880008	38890104
8	48680	3395689213864	3395689203768	3395724779736	35565872	35575968
8	67880	3395692547224	3395692537104	3395724798936	32251712	32261832
9	10280	3626062490064	3626062474608	3626104833496	42343432	42358888
9	67880	3626072490064	3626072474608	3626104891096	32401032	32416488
10	29480	8168308061504	8168308057936	8168346669576	38608072	38611640
10	48680	8168311394840	8168311391272	8168346688776	35293936	35297504
10	67880	8168314728168	8168314724608	8168346707976	31979808	31983368