NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for teknisk kybernetikk

Einar Henriksen

Development of a Sensor Node with Time Sensitive Networking in the Zephyr Operating System

Masteroppgave i Kybernetikk og Robotikk
Veileder: Geir Mathisen
Medveileder: Henrik Austad
August 2021

Einar Henriksen

Development of a Sensor Node with Time Sensitive Networking in the Zephyr Operating System

Masteroppgave i Kybernetikk og Robotikk
Veileder: Geir Mathisen
Medveileder: Henrik Austad
August 2021

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for teknisk kybernetikk

Master's Thesis
TTK4900

Development of a Sensor Node with Time Sensitive Networking in the Zephyr Operating System

Einar Henriksen

Supervisor:
Geir Mathisen

Co-supervisor:
Henrik Austad

Trondheim, August 30th, 2021

NTNU
Norwegian University of Science and Technology

Faculty of Information Technology, Mathematics and Electrical Engineering
DEPARTMENT OF ENGINEERING CYBERNETICS

NTNU
Norwegian University of
Science and Technology

Faculty of Information Technology
and Electrical Engineering
Department of Engineering Cybernetics

MASTER THESIS DESCRIPTION

Candidate:	**Einar Henriksen**
Course:	**TTK4900 Engineering Cybernetics**
Thesis title (Norwegian)	Utvikling av en sensornode med «Time Sensitive Networking» og Zephyr operativsystem
Thesis title (English):	Development of a Sensor Node with Time Sensitive Networking in the Zephyr Operating System

-

Thesis desctiption:

The thesis will be a continuation of the TTK4550 specialization project and will adapt Zephyr on a FRDM-K64F evaluation board to transmit gyro sensor data to a receiver via a TSN stream.

In order to properly fuse sensor data from multiple publishers, it is important that the endpoints are synchronized to the same clock. Furthermore, to accommodate multipe sensor streams in the network configured for TSN CBS, a sender must adhere to strict timeslots and not transmit more data than allocated, nor in "bursts".

The result of this work is planned utilized in a follow-up project where these sensor readings are used to control the position a robotic arms end effector.

The tasks will be:

1. Conduct a literature study for deterministic networks in general and properties and capabilities of embedded real-time systems, in particular Zephyr.

2. Implement a credit-based shaper (CBS) for Zephyr.

3. Implement and evaluate a small microcontroller running Zephyr with TSN/CBS as a sensor node to publish sensor data through a TSN stream.

Start date:	7. April, 2021
Due date:	30. Aug, 2021

Thesis performed at:	Department of Engineering Cybernetics
Supervisor:	Professor Geir Mathisen, Dept. of Eng. Cybernetics
Co-supervisor:	Research Scientist Henrik Austad, SINTEF Digital

Preface

This Master's Thesis was conducted at the Department of Engineering Cybernetics of the Norwegian University of Science and Technology (NTNU) in the final semester of my Master's Programme during the spring and summer of 2021, under the supervision of Professor Geir Mathisen. Henrik Austad, research scientist at SINTEF Digital, functioned as a co-supervisor. I want to thank these supervisors for their encouragement and assistance, and I also want to thank SINTEF for supplying the necessary hardware for this project.

Contents

List of Figures

List of Tables

Nomenclature

AAF	AVTP Audio Format
API	Application Programming Interface
AVB	Audio/Video Bridging
AVTP	Audio/Video Transport Protocol
CBS	Credit Based Shaper
CRF	Clock Reference Format
CVF	Compressed Video Format
DT	Device Tree
EF	Experimental Format
FOSS	Free Open Source Software
GM	Grandmaster
gPTP	generic Precision Time Protocol
IEEE-SA	Institute of Electrical and Electronics Engineers Standards Association
OS	Operating System
PDU	Protocol Data Unit
PTP	Precision Time Protocol
SRP	Stream Reservation Protocol
TAS	Time Aware Shaper
TSN	Time Sensitive Networking

Abstract

This project was a continuation of the term project in TTK4550, and aimed to implement parts of a Time Sensitive Networking (TSN) stack in the Zephyr operating system on an NXP FRDM-K64F board, in order to investigate the potential of using Zephyr to set up a sensor board as a TSN talker node. In this project the main focus was integrating gPTP (generic Precision Time Protocol) functionality and implementing a Credit Based Shaper (CBS).

gPTP functionality was successfully integrated after dealing with a bug in Zephyr's packet socket library in cooperation with the Zephyr developers. CBS functionality was also implemented along with other improvements to the sensor node application. In its current state, the application is able to supply a stream of sensor data with associated timestamps, and its performance looks promising, but there is still work left until a full TSN stack has been realised.

Sammendrag

Dette prosjektet var en fortsettelse av prosjektoppgaven i TTK4550. Målet var å implementere deler av en Tids-Sensitiv Nettverks-stabel (TSN) i operativsystemet Zephyr på en NXP FRDM-K64F utviklingsplattform, for å undersøke potensialet i å bruke Zephyr til å sette opp et sensorkort som en TSN talenode. Hovedfokuset i dette prosjektet var å integrere gPTP-funksjonalitet (generisk Presisjons-Tids-Protokoll) og å implementere en kredittbasert trafikkformer (CBS).

Integrering av gPTP-funksjonalitet var vellykket etter at en programvarefeil i Zephyrs pakkesocketbibliotek ble korrigert i samarbeid med Zephyr-utviklerne. CBS-funksjonalitet ble også implementert, i tillegg til andre forbedringer på sensornodeapplikasjonen. Slik applikasjonen ser ut nå, så er den i stand til å produsere en strøm av sensordata med tilhørende tidsstempel, og den ser ut til å kunne gi god ytelse, men det er fortsatt en del arbeid igjen før en full TSN-stabel har blitt realisert.

1 Introduction

In modern control systems, sensor units in distributed network environments should be small and cheap while also keeping time accurately to ensure precise data acquisition. Time Sensitive Networking (TSN) could possibly provide a solution to this problem, by ensuring a shared timedomain and predictable transmit delays in such a network using standard Ethernet equipment.

Zephyr, the relatively new real-time operating system for embedded devices, offers interesting possibilities for the use of sensor boards with limited hardware as nodes in a Time Sensitive Network. To investigate this potential, a TSN stack should be implemented in Zephyr and its performance should be tested. In particular, the accuracy of gPTP clock synchronization and the sensor board's ability to reliably send data is of interest.

2 Theory

In this chapter the basic features of TSN and its standards gPTP, AVTP and CBS will first be presented. Then an introduction to the Zephyr environment will follow.

2.1 Time Sensitive Networking

This section is partially quoted from the previous project in TTK4550:

Time Sensitive Networking (TSN) is a set of standards that define technical requirements for networks where low transmission latency, high availability and high predictability is necessary. TSN is standardized by the IEEE 802.1 work group, which was first formed in 2004 at IEEE-SA (Institute of Electrical and Electronics Engineers Standards Association). The group was originally working on the AVB standard (Audio Video Bridging) for sending audio and video data through standard Ethernet switches with high speed and reliability. This standard was aimed towards professional AV-purposes such as large scale concerts or TV broadcasts where a precision of 1 μs is required.

However, this proved an effective way of reliably sending time sensitive data through standard networking equipment, thus in 2012 the scope of the project was expanded to encompass any kind of data stream, and not just AV-data specifically. The project was then renamed to TSN, though the term AVB is still used in the context of AV-streams.

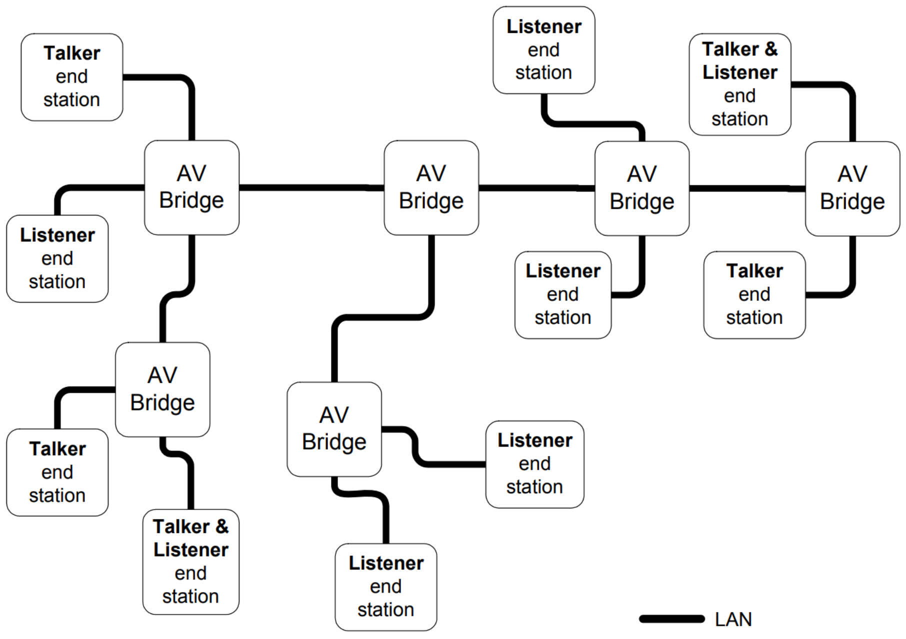


Figure 1: An AVB network [7]

Prior to the AVB standard, Audio/Video transmission was usually done with either complex analog setups or expensive proprietary ones. AVB aims to replace this with an open solution using standard Ethernet equipment. [21] As there is no concept of time in a typical IT network, AVB defines how to make use of this equipment in a way that is satisfactory to real-time applications. This is done by precisely synchronizing the clocks in the network, and by introducing a stream reservation protocol as well as queuing and forwarding rules that ensure data arrives on time.

2.1.1 AVB - IEEE 802.1BA

This section is quoted from the previous project in TTK4550:

"The AVB standard defined in IEEE 802.1BA [7] describes the basic architecture of an AVB network, as well as features, protocols and configurations necessary for the network to be capable of transporting time sensitive data streams.

An AVB network consists of end stations and bridges. End stations are classified as talkers and listeners, where a talker is the source of an AVB stream and a listener is a destination for such a stream. For example a talker can send audio data from a microphone to be played on speakers by one or several listeners. In some cases an end station can be both a talker and a listener. A bridge is a relay device in the network, such as a network switch forwarding AVB streams from talkers to listeners.

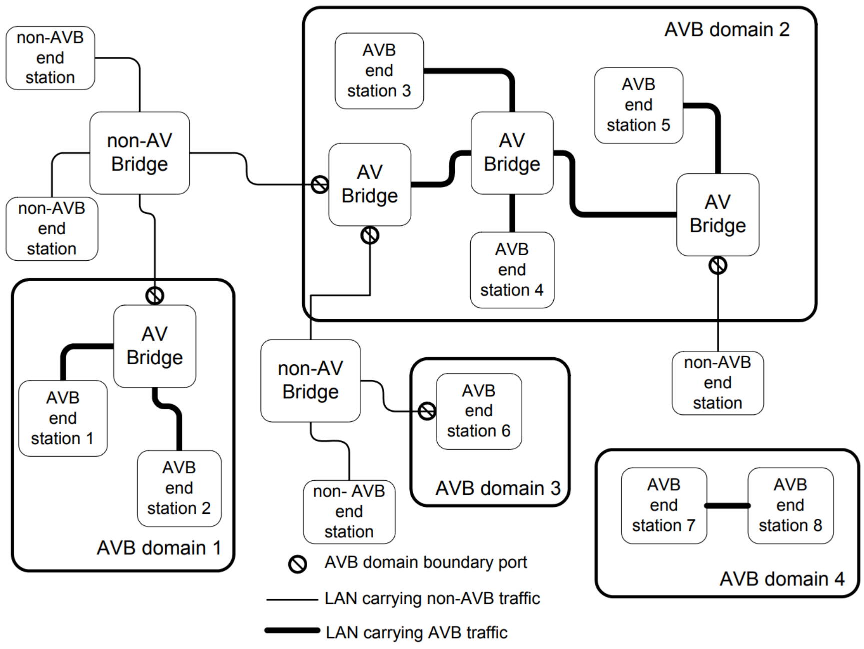


Figure 2: AVB domains in a network with non-compatible nodes [7]

An AVB domain is defined as a section of a network in which any talker would be able to send an AVB stream to any listener, meaning that any bridge between them must be AVB compatible, and that every device in the domain has consistent priority definitions.

AVB is based on the IEEE 802 standard for network architectures, but with added functionality on top. This means that AVB devices can communicate normally with non AVB devices on the same network. The most important additions in the AVB architecture include precise clock synchronization through gPTP, traffic shaping for media streams, admission control through the Stream Reservation Protocol (SRP) and identification of non AVB devices."

2.1.2 gPTP - IEEE 802.1AS

The first two paragraphs of this section are quoted from the previous project in TTK4550:

"gPTP (generic Precision Time Protocol) is a protocol included in the TSN standard that specifies how to synchronize the clocks of the machines in a TSN network. [8] The protocol is based on, and is a profile of, PTP (Precision Time Protocol) defined in IEEE 1588, but with additional specifications for keeping time in a real-time environment. The most important changes are that all communication must be done via link-layer MAC PDUs and that every node in the gPTP domain must be using gPTP (aka be time-aware). In gPTP even bridges that are just forwarding packets are required to be time-aware, which is not the case in PTP.

A single node in the domain is chosen as the grandmaster (GM) through the Best Master Clock Algorithm. The gPTP domain will then use the GM's clock to synchronize all the clocks in the domain. Every talker in the network must be GM capable in case the GM should drop out of the network. The GM regularly shares its clock value with the network, and each node adds its propagation delay (the time it takes for a message from the GM to reach the node) to this value to get the current synchronized time. These propagation delays are also regularly recalculated to account for any changes in the network conditions. In addition to this, the nodes' clocks are all syntonized (frequency locked) to the GM clock by calculating the ratio between the local clock frequency and the GM clock frequency, so that every node operates with the same time base."

When a clock wants to synchronize with the grandmaster clock, it needs to calculate the offset between itself and the master, noted as $\tilde{o}$. This requires two data points to calculate, because there is also a second unknown in the system, the transit delay between the nodes noted as d. The synchronization process starts with the master sending a *Sync* message containing timestamp T_1. The clock receiving this message notes the time it is received as T_1'. It then responds with a *Delay_Req* message containing timestamp T_2. The master responds to this with a *Delay_Resp* message that contains T_2' which is when the master received T_2. These timestamps can express the unknowns in the system like this:

$$T_1' - T_1 = \tilde{o} + d \textit{ and } T_2' - T_2 = -\tilde{o} + d \tag{1}$$

Which means the clock offset can be calculated like this:

$$\tilde{o} = \frac{1}{2}(T_1' - T_1 - T_2' + T_2) \tag{2}$$

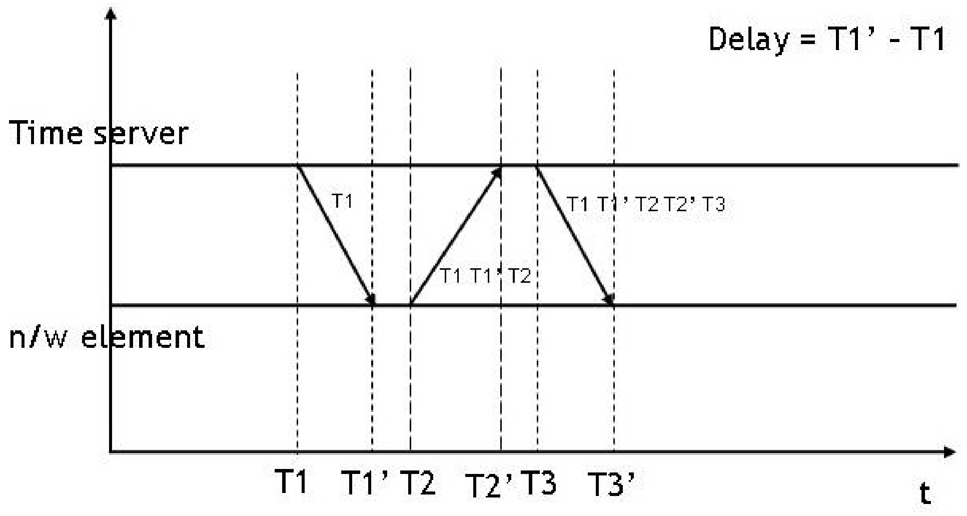


Figure 3: PTP synchronization mechanism [18]

2.1.3 Alternatives to AVB and gPTP

Network Time Protocol (NTP) is the most widely used time synchronization protocol on the internet [13]. NTP is a pure software protocol which works by regularly sending packets to an NTP server and then calculating the round-trip time based on the response time. It is not uncommon that an NTP server used for this purpose is already several layers down the hierarchy from the original clock source. NTP can achieve an accuracy of 1 ms, but propagation errors between servers downwards in the hierarchy make the typical accuracy lower. Due to the low accuracy this protocol is not particularly suitable for real-time IoT applications.

PROFINET, short for Process Field Net, is a real-time Ethernet standard maintained and supported by Profibus & Profinet International in Germany [17]. PROFINET does many of the same things as TSN, but is more focused on covering the needs of communication between technical industry equipment rather than TSN's focus on media streams. PROFINET is also designed as a complete system to be installed at for example a factory as opposed to TSN's approach of only supplying the standard and not necessarily an implementation.

Precision Transparent Clock Protocol (PTCP) is a time synchronization protocol used by the PROFINET standard. Like gPTP this protocol is based on PTP, differing mostly from gPTP in its use of what it calls *transparent clocks*. By this they mean that not every node in the network is synchronized, but only the end nodes and the master. This is made possible by making bridges compensate for their internal forwarding delay by editing the timestamps, which allows for high synchronization accuracy on a network with many hops.

Synchronous Ethernet (SyncE) also known as ITU-T G.8262 is another real-time Ethernet protocol [20]. In this protocol the clock signal is transmitted in the physical layer of the network, and the signal is filtered and regenerated through a phase-locked loop at each node. This can not be done with standard Ethernet equipment and requires custom hardware. SyncE can achieve a frequency accuracy of ±4 ppm, meaning the clock frequency will be synced to within 4 parts per million of the source clock frequency. It does not offer offset correction, meaning that the protocol won't keep the clock's periods in sync. The clock signal in the network should be traceable to a unique external master clock, but how the signal is distributed is up to the implementation and can be done through a tree- or mesh-type network as long as the signal reaches every node and avoids synchronization loops.

2.1.4 AVTP - IEEE 1722

This section is quoted from the previous project in TTK4550:

"AVTP (Audio Video Transport Protocol) defines the transport protocol and data encapsulations used in time sensitive audio, video and control applications that make use of the TSN standard. It is designed to take advantage of the features of TSN, for instance, most AVTP packets contain a gPTP timestamp represented in nanoseconds called the AVTP Presentation Time, which lets the listener know exactly when to use the data it receives. For example it can tell a listener when to start playing audio data, so the audio can be synced across multiple speakers.

AVTP packets are encapsulated in an Ethernet frame and adds an additional header of metadata in the protocol data unit (PDU) before the data payload. There are several different subtypes of AVTP, each with its own defined header format. In general these subtypes are divided into the continuous stream formats and the discrete control formats. Examples of some common subtypes are AVTP Audio Format (AAF), Compressed Video Format (CVF) and Clock Reference Format (CRF). All subtype formats start with a common header that state what subtype is in use, followed by either a common stream header or a common control header with variations for each subtype. [4]

An Ethernet frame (specified in IEEE 802.3) starts with a 14 byte header, where the first six bytes specify the destination MAC address, the next six the source MAC address, and the final two the Ethertype in use. This header is followed by the packet payload, and the frame is ended with a four byte check sequence. Because of a collision detection feature in the Ethernet standard, there must be at least 64 bytes (512 bits) between the start of each packet sent on the network. This has been solved in the standard by defining a packet as being minimum 64 bytes in size, which results in the data payload being at least 46 bytes. If the actual payload happens to be less than 46 bytes, it will be padded with zeroes before it is sent. The maximum payload size in a standard Ethernet frame is 1500 bytes. [2]

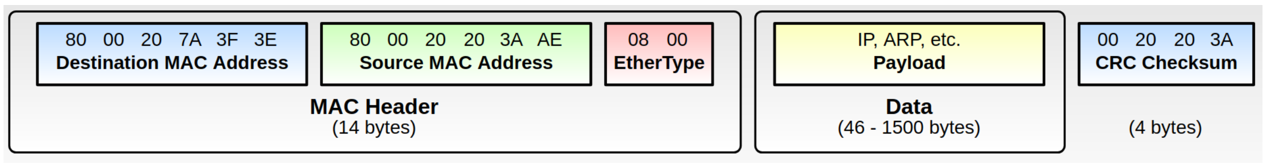


Figure 4: Structure of an Ethernet frame [2]

In the case of AVTP, the Ethernet frame payload consists of an AVTP PDU (AVTPDU). If the AVTP subtype in use is a stream format, this PDU starts with a common stream header (See Figure 5). The common stream header starts with a byte specifying the subtype in use, followed by a byte containing various flags, most notably the *stream_id valid* flag and the *timestamp valid* flag. These two flags signify whether the stream_id and timestamp fields in the header contain valid values. The following byte consists of a sequence number that is incremented by one for each packet in the stream, to let a listener confirm it has received every packet in order. Next is a byte containing a data field that the standard lets each format use as needed, and also the *timestamp uncertain* flag. This flag is set if a discontinuity occurs in gPTP time, for instance if the grandmaster node changes. Then follows an 8 byte *stream_id* field. This ID is used to identify what data stream a packet belongs to, where the first six bytes contain the talker's MAC address and the remaining two bytes is an ID for the specific stream. After this comes the 4 byte *avtp_timestamp*, which should contain the AVTP Presentation Time. Then comes another four bytes of format specific data fields, followed by two bytes specifying the length of the stream data payload (in bytes), and finally another two bytes of format specific data before the data payload itself. In total the common stream header is 24 bytes long, which makes the stream data payload minimum 22 bytes. If the actual payload is less than this it will be padded with zeroes, in which case the listener can use the *stream data length* value in the header to tell the data apart from the padding."

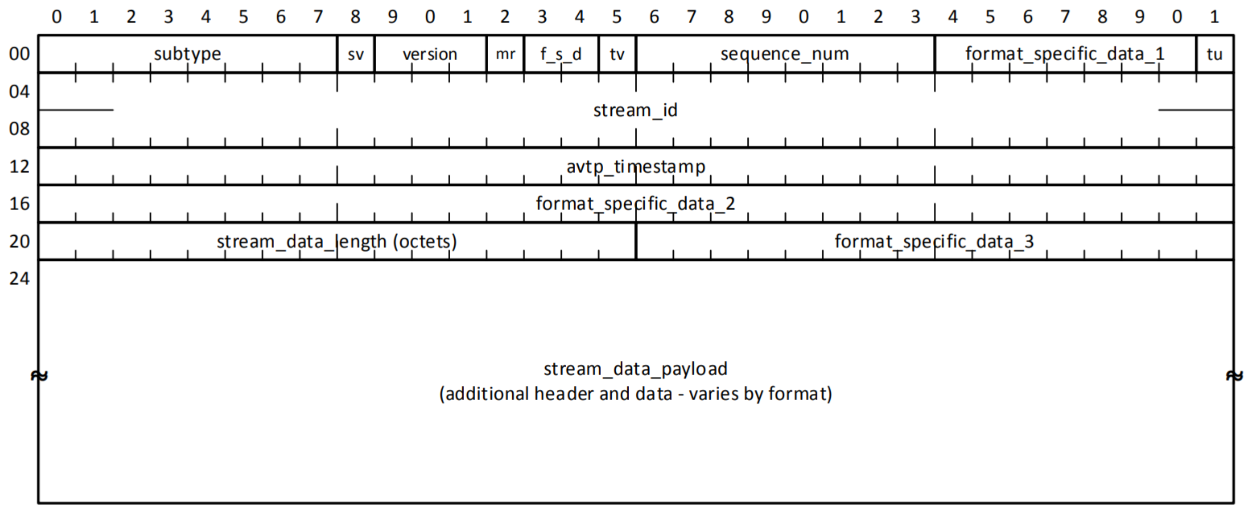


Figure 5: AVTPDU common stream header [4]

2.1.5 CBS - IEEE 802.1Qav

The credit-based shaper (CBS) is an algorithm defined in IEEE 802.1Qav [5], a standard dealing with forwarding and queuing methods in TSN. CBS is what's called a traffic shaping algorithm, meant to distribute high priority network traffic evenly and predictably onto a network. Operating without this type of scheduling in a system with multiple priority classes on the same network will tend to clog up the network with bursts of high priority traffic, making the network availability unpredictable for lower priority traffic. In order to make all priority classes deterministic, CBS makes high priority tasks build up credit before they are allowed to transmit data. How quickly this credit is accumulated is determined by the *idle slope* which is calculated based on how much bandwidth the high priority task needs to operate:

$$\texttt{idleSlope} = \texttt{portTransmitRate} \times \texttt{bandWidthFraction}$$

Here *portTransmitRate* is the data transmit rate that the network interface is able to deliver given in bits per second, and *bandWidthFraction* is the fraction of this bandwidth that TSN is allowed to use, which can be at most 75%. The credit value will increase at this rate as long as the task is idle and the current credit value is negative, or while the task is ready to send but waiting for the network interface to be available. While the high priority task is using bandwidth to transmit data, the credit value shall decrease at a rate determined by the *send slope* value:

```
sendSlope = idleSlope - portTransmitRate
```

As soon as the credit value is non-negative and the network is available, the task is allowed to transmit data.

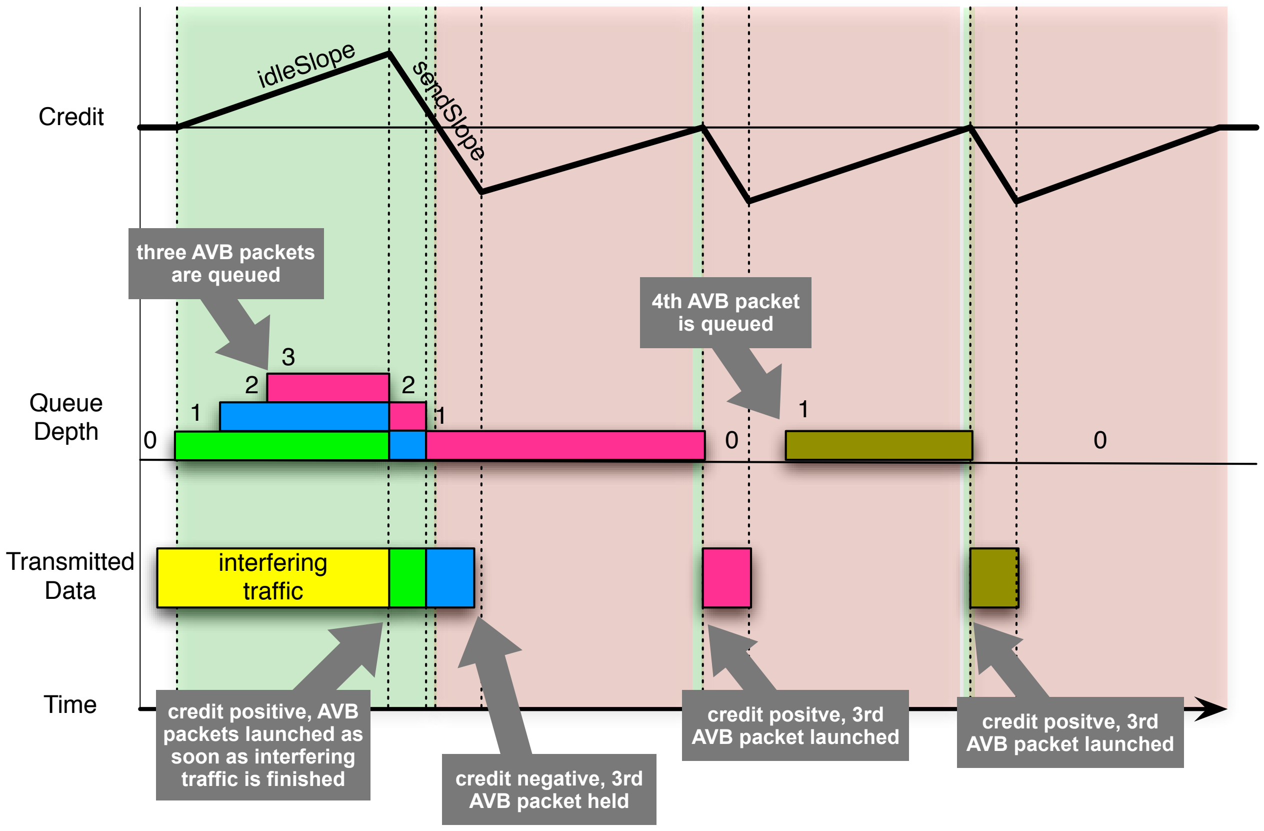

Credit
idleSlope
sendSlope
three AVB packets are queued
4th AVB packet is queued
Queue Depth
0
1
2
3
2
1
0
1
0
Transmitted Data
interfering traffic
Time
credit positive, AVB packets launched as soon as interfering traffic is finished
credit negative, 3rd AVB packet held
credit positve, 3rd AVB packet launched
credit positve, 3rd AVB packet launched

Figure 6: Credit-based shaper operation [22]

When discussing the performance of a traffic shaper like this, one often investigates the bandwidth use in what is called the *measurement interval*, which is the smallest time interval in which TSN should still use less bandwidth than the desired *bandWidthFraction*. This time interval is given by the *idle slope*, the maximum *interference size* and the maximum *frame size* for the TSN stream. The maximum interference size is the largest amount of traffic a high priority task might have to wait for when it is ready to transmit data. For the highest priority task, the largest interference is a maximum size standard Ethernet frame.

For the highest priority class in TSN, the maximum frame size for a packet in the TSN stream is defined to be 75% of the bits that can be transmitted in 125 μs. For 100 Mb/s LAN, this makes the maximum frame size 9368 bits, or 1171 bytes. Using this and the properties of the CBS, the maximum *burst size* could then be calculated, which is the amount of bits the task would be able to send after waiting through a maximum interference, until it runs out of credits. The *measurement interval* would then be this burst size multiplied by 100/75. For the second highest priority task, the largest interference would be the maximum size Ethernet frame plus the highest priority task's maximum burst size, which could then be used to calculate the second highest priority task's maximum burst size, and so on.

There are other traffic shaping algorithms defined in the TSN standards. One of them is the time-aware shaper (TAS) defined in IEEE 802.1Qbv [6]. TAS divides the available network time into repeating cycles of fixed length, where time slices within these cycles can be assigned to one or more specific network priority. This way a specific task can have a periodic time slot with exclusive access to data transmission. This shaper will in most use cases give shorter delays than CBS, but it is also much more complex to implement.

2.2 Zephyr

This section is quoted from the previous project in TTK4550:

"Zephyr is a lightweight open source real-time operating system designed for embedded systems, developed by the Linux Foundation and first released in 2016. Zephyr is marketed towards IoT applications (Internet of Things) and was quickly adopted by industry giants such as Intel, NXP, Texas Instruments and Nordic Semiconductor. [34]

Zephyr aims to provide the necessary tools to develop applications for resource-constrained embedded systems. This includes a small kernel, support for several communication protocol stacks, and a flexible kernel configuration system that ensures only needed resources are included at compile-time. The Zephyr kernel is monolithic, with a single address space, and has support for several scheduling algorithms and both symmetric and asymmetric multiprocessing. [31] The Zephyr source code is written in C, and its kernel configuration systems are inherited from the Linux kernel, but written in Python to make it more portable. Its build system is based on CMake, which means applications can be built on Linux, macOS and Windows."

2.2.1 Zephyr's Build Environment

This section is quoted from the previous project in TTK4550:

"In order to create an application for Zephyr, the source code must be placed in an application directory together with necessary configuration files for the project, and can then be compiled using Zephyr's own command line tool *west*. [25] The simplest Zephyr projects need two of these configuration files, called *CMakeLists.txt* and *prj.conf*. *CMakeLists.txt* links the application directory with the build system, providing basic build options such as board-specific configurations. In *prj.conf*, application-specific kernel options are specified, such as activating the I2C or Ethernet interface, enabling floating point operations, or setting an IP address. It is very important to remember setting the right options here, as applications will still compile even if crucial kernel options are not set, which can lead to undefined behaviour and crashes in run-time."

2.2.2 Zephyr's Devicetree

This section is quoted from the previous project in TTK4550, but the example code has been updated to reflect the currently used Zephyr version:

"Accessing hardware interfaces in a Zephyr application is done through Zephyr's own devicetree structure. [27] In Linux the devicetree is stored as a binary file that is parsed in run-time. However, the memory available on many of the embedded devices supported by Zephyr is very limited. In order to save memory the devicetree is implemented as a C header file abstracted behind a macro API, so that only the relevant parts of the tree are included when the project is compiled.

As long as the devicetree is well defined, it is fairly simple to access the hardware through the devicetree API. For example an I2C interface can be bound to a *device* struct like this:

```
const struct device *i2c_dev;
i2c_dev = device_get_binding(DT_LABEL(DT_NODELABEL(i2c)));
```

Where *DT_NODELABEL( )* returns the *node id* of a node in the devicetree with the given *nodelabel* value, and *DT_LABEL( )* returns the *label* property of the node with the given *node id*. [28] This label property is then used in the function that binds the device."

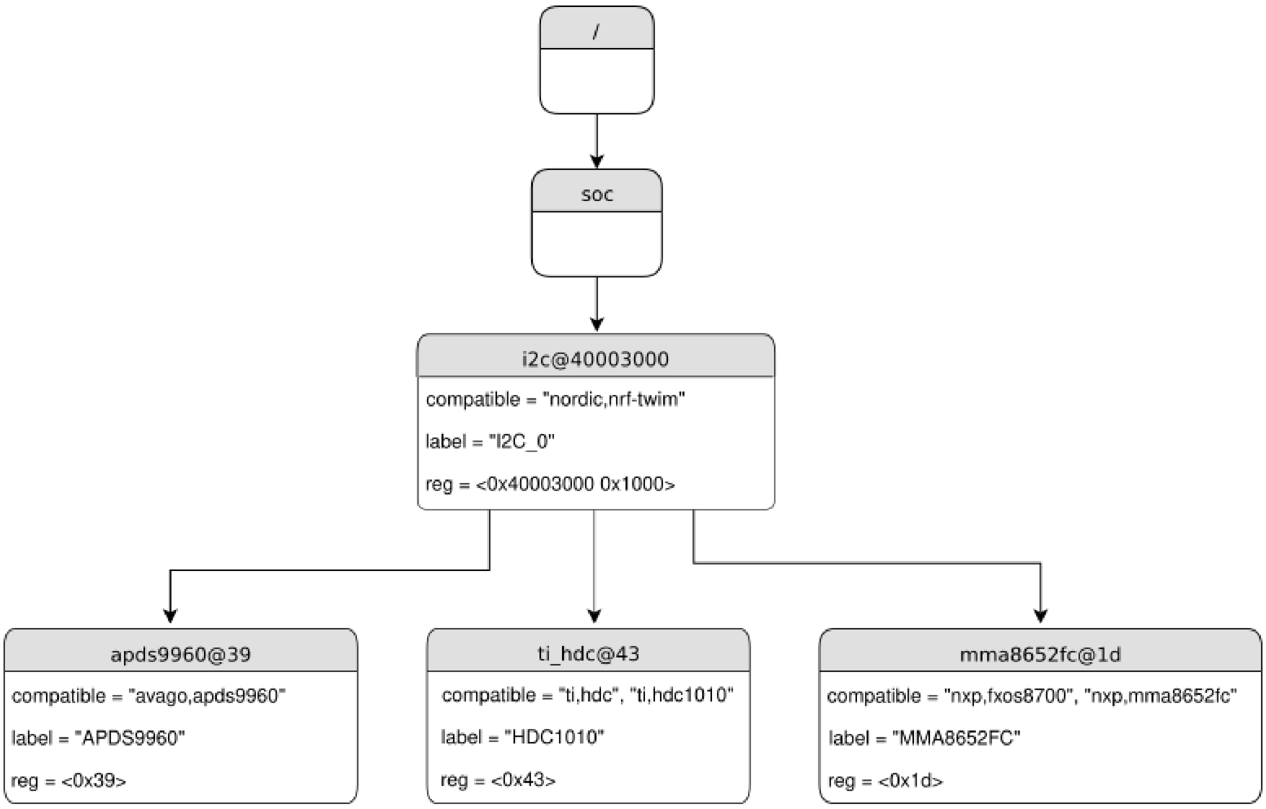

/
soc
i2c@40003000
compatible = "nordic,nrf-twim"
label = "I2C_0"
reg = <0x40003000 0x1000>
apds9960@39
compatible = "avago,apds9960"
label = "APDS9960"
reg = <0x39>
ti_hdc@43
compatible = "ti,hdc", "ti,hdc1010"
label = "HDC1010"
reg = <0x43>
mma8652fc@1d
compatible = "nxp,fxos8700", "nxp,mma8652fc"
label = "MMA8652FC"
reg = <0x1d>

Figure 7: Example of a Zephyr devicetree [27]

2.2.3 Zephyr's Socket Implementation

This section is quoted from the previous project in TTK4550:

"Zephyr contains a partial implementation of the BSD socket API from the POSIX standard. [26] This means that simple network applications can be implemented very similarly to how they would be implemented on a Linux system. The socket implementation is namespaced with a prefix by default to avoid name conflicts with other Zephyr modules, so *bind( )* can be accessed as *zsock_bind( )* and so forth."

2.2.4 Threads

One of Zephyr's main appeals is its easily configurable multi-threaded environment for microcontrollers. Zephyr's kernel runs a priority-based scheduler which allows multiple running threads to share the CPU (or CPUs if the platform has multiple cores). The scheduler chooses the highest priority thread to be the *current thread* at any time, if there are multiple threads with the same priority the scheduler chooses the one that has been waiting the longest.

There are several options for implementations of the queue of threads ready to be executed, tailored to different needs and restrictions. For most applications with only a small amount of threads, the simple unordered list called the *dumb* scheduler will be sufficient, but there are more scalable options available such as the *red/black tree* queue that can handle thousands of threads. The only limit to how many threads can be defined in an application is the amount of RAM available on the system. Being an OS, Zephyr will usually run several threads in the background in addition to the threads defined by the application. These threads will spawn automatically based on specifications in the project configuration files. For instance if the project is configured to have basic IP functionality, Zephyr will spawn threads to set this up at boot, and will run threads in the background to do things like answer pings in parallel with the application code. This provides a powerful abstraction for embedded developers, as they can focus on their application while the kernel takes care of the underlying functionality.

2.2.5 Semaphores and mutexes

When dealing with multiple threads in an application, Zephyr has the expected tools to ensure proper handling of information and communication between threads. Among these tools are semaphores and mutexes, which are implemented as *kernel objects* in Zephyr, meaning they can't be accessed directly by user threads but are available through Zephyr's APIs. A semaphore may be given by one thread and taken by another to signal to the thread waiting to take the semaphore that it may go on with its task. A mutex is typically used to ensure only one thread may access a shared variable at a time. [19]

2.3 Other RTOS

FreeRTOS is an open source real-time operating system in development since 2003, which was acquired by Amazon in 2017. [3] It is written mainly in C, and is designed to be small, portable and maintainable. The OS has been ported to 35 different microcontroller platforms, and contains all the basic features expected to be found in a RTOS such as threads, mutexes, semaphores and timers. FreeRTOS is considered to be the most widely deployed RTOS today. [12] Where it differs from Zephyr the most is that FreeRTOS only supplies these basic features and not the more advanced OS features such as drivers, file systems, network stacks etc. This is because FreeRTOS focuses on compactness and speed, so any additional features must be added from other sources.

VxWorks is a proprietary real-time operating system designed for embedded systems, with support for Intel, ARM, POWER and RISC-V architectures. [24] It is based on a simple early RTOS called VRTX (Versatile Real-Time Executive), and has been developed and maintained by Wind River Systems since 1987. VxWorks was among the most popular embedded operating systems for a long time, as it was among the first to include a networking stack in the 90s. It has been widely used in several markets, including aerospace, robotics and consumer electronics. Before Zephyr became a Linux Foundation project in 2016, it was actually being developed by Wind River Systems under the name *Rocket* [34]. The main difference between Rocket and VxWorks was that Rocket aimed to have a much smaller memory footprint, focusing on the growing market of limited hardware sensor boards and single-function IoT devices.

INTEGRITY is a proprietary real-rime operating system for embedded systems, developed and maintained by Green Hills Software. [9] One version of this OS, called INTEGRITY-178 because it adheres to the DO-178B/C aviation guidelines [10], offers hard real-time, meaning its kernel guarantees bounded computing times. This has been done by eliminating some of the more unpredictable OS features, such as dynamic memory allocation. INTEGRITY is widely used in the aviation industry, both in military jets and commercial aircrafts.

Nucleus is a proprietary real-time operating system for embedded systems, developed by Mentor Graphics under Siemens, first released in 1993. [14] It quickly became one of the most widely used RTOSs for embedded systems and stayed among them for several years. Some of its success can be attributed to early support for networking, graphics and file systems, and also that its source code was provided on purchase making it easier to debug than many of its competitors while there still weren't many open source RTOSs to choose from. Nucleus has been used in all types of embedded systems, such as medical, industrial, aerospace and IoT applications.

3 Proposed Solution

The setup from the previous project in TTK4550 was used and expanded upon in this project. The overarching goal of these projects was to investigate the potential of using Zephyr to set up a TSN talker node on a sensor board with limited hardware capabilities.

The NXP FRDM-K64F development board was chosen as the development platform for this project because it appears to be well supported in Zephyr, and is listed as having hardware support for gPTP.

First of all the newest version of Zephyr should be downloaded to investigate whether it is easier to get gPTP and the gpio interrupt operational, because this would majorly impact the project. Then other features of the TSN stack should be implemented (traffic shaping, SRP), and the performance of the talker node should be tested.

3.1 Testing Setup

For a simple testing setup to investigate basic talker performance, the K64F should be connected directly to the desktop computer by an Ethernet cable. With a desktop computer that also supports TSN some basic tests could then be performed:

1. Running a gPTP daemon on the computer to see if gPTP is able to synchronize the clocks of the device and the computer.
2. Sending sensor data from the talker node, and capture it on the desktop computer.
3. Observing timestamps of packets to investigate the performance of a traffic shaper.
4. The same tests as above, but stress testing with a network traffic generator.

At the SINTEF robotics lab, once a TSN stack is set up, the performance of the device could be investigated further in a distributed network of TSN compatible devices, and used to control a robotic arm's end effector.

4 Method

This chapter starts by listing the hardware and software used in the project. Then follows a description of what was done in the project and how solutions were implemented.

4.1 Hardware

4.1.1 NXP FRDM-K64F

The NXP FRDM-K64F is a development board from NXP, featuring a Kinetic K64F MCU with an ARM Cortex-M4 core. [15] The CPU can run at up to 120 MHz, and has 1 MB of flash memory and 256 kB RAM. It is a low-cost development platform with several peripherals and interfaces, including Ethernet and Arduino compatible extension headers. Zephyr compatibility is under active development, making it a good candidate for developing in that environment.

The development kit also came with a FRDM-STBC-AGM01 sensor shield, featuring a FXAS21002C angular rate gyroscope and a FXOS8700C accelerometer/magnetometer. [16]

4.1.2 Intel I210-T1 Ethernet Adapter

This is a network interface controller from Intel that is compatible with gPTP. It was mounted in the desktop computer used in the project, and was used for Ethernet communication with the FRDM-K64F.

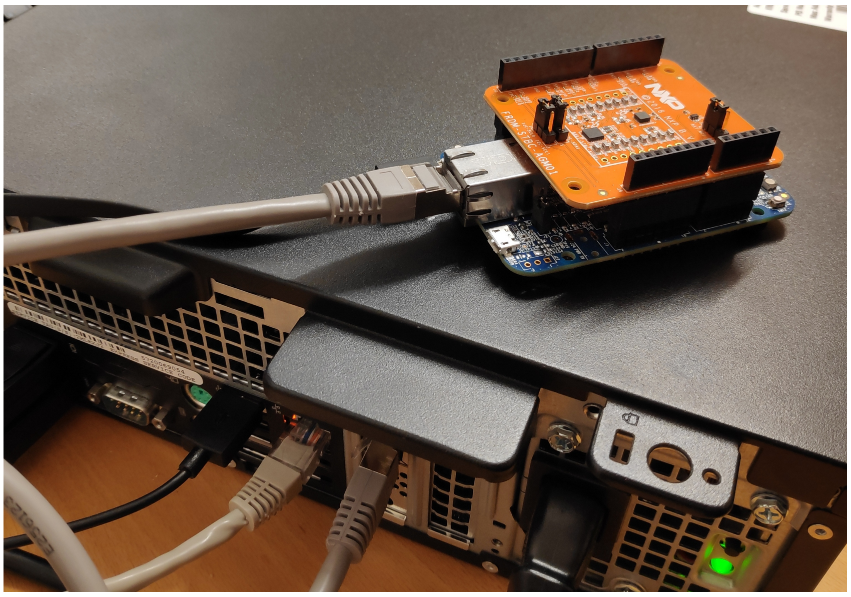


Figure 8: NXP FRDM-K64F connected to the computer via Ethernet

4.2 Software

4.2.1 Wireshark

Wireshark is a network protocol analyzer, used to monitor the network traffic on a computer's network interface. In this project Wireshark has been used to monitor the network traffic between the FRDM-K64F and the desktop computer, capturing packets sent between them.

4.2.2 Ubuntu

The desktop computer used for application development and network analysis was running Ubuntu 20.04.2 LTS. [23]

4.2.3 Zephyr

The Zephyr source code was downloaded using West. The version used in this project was Zephyr 2.5.99. All documentation accessed through Zephyr's websites was related to v.2.5.99, and was accessed in the period from April 2021 to August 2021.

4.2.4 West

West is Zephyr's own multi purpose command line tool, which is used both for downloading the Zephyr source code and for compiling Zephyr applications. [33] The newest version at the time, West v0.8.0, was downloaded through pip, Python's package installer.

4.2.5 Libavtp

Libavtp is an open source implementation of AVTP developed by Intel, now hosted by the AVnu Alliance [1], which is an industry organization that promotes AVB/TSN and provides certification of projects that use the standards. Source code from version 0.1.0 was used in this project, found in the commit posted on GitHub in September 2020.

4.3 Method Description and Implementation

4.3.1 Setup from previous project

Here is a summary of what was set up in the previous project, that is also used in this project:

The Zephyr toolchain was installed on Ubuntu so that projects can be built using west:

```
west build -p -b frdm_k64f samples/hello_world/
```

And then flashed to the NXP FRDM-K64F board connected via USB:

```
west flash
```

The board's output could then be read through a serial terminal using screen on Linux:

```
screen /dev/ttyACM0 115200
```

- Zephyr's implementation of the BSD socket library was used to set up a socket in order to send AVTP packets.
- AVTP packets were created via the open source library libavtp, which was ported to Zephyr as part of the previous project.
- Gyro data from a FRDM-STBC-AGM01 sensor shield mounted on the FRDM-K64F board was accessed through I2C in order to supply the AVTP packets with some real data.
- Zephyr's *timer* API was used to set up a regular timer interrupt, which would trigger reading gyro data and sending a packet.

4.3.2 Gpio-interrupt

In the previous project, there was an attempt to set up a hardware interrupt from the gyroscope each time new data was ready, to use that as a trigger to read gyro data. However this turned out to be more complicated than expected at the time, because there was no easy way to access the interrupt pin through Zephyr's devicetree. In this project, after getting more familiar with the device-tree structure, the interrupt pin was manually added as its own node in the devicetree. In order to do this, these lines were added to
/boards/arm/frdm_k64f/frdm_k64f.dts:

```
gpio_keys {
    [...]
    ptc12_interrupt: interrupt_pin {
        label = "MYPTC12LABEL";
        gpios = <&gpioc 12 (GPIO_ACTIVE_HIGH | GPIO_PULL_DOWN)>;
    };
};
```

And this line was added to */boards/arm/frdm_k64f/pinmux.c*:

```
pinmux_pin_set(portc, 12, PORT_PCR_MUX(kPORT_MuxAsGpio));
```

This adds the interrupt pin as an active high gpio input with pull-down, very similarly to how the user-switches on the board are set up in the devicetree. With this setup, the interrupt pin could be accessed in the project code, and an interrupt routine was set up using Zephyr's *callback* API:

```
#define PTC12_LABEL     DT_GPIO_LABEL(DT_NODELABEL(ptc12_interrupt),gpios)
#define PTC12_PIN       DT_GPIO_PIN(DT_NODELABEL(ptc12_interrupt),gpios)
#define GYRO_INT_FLAGS  (GPIO_INPUT|DT_GPIO_FLAGS(DT_NODELABEL(ptc12),gpios))

static struct gpio_callback gyro_cb_data;
void gyro_interrupt(const struct device *dev,
                    struct gpio_callback *cb, uint32_t pins)
        {
            k_sem_give(&gyro_data_rdy);
        }

const struct device *int_port;
int_port = device_get_binding(PTC12_LABEL);
gpio_pin_configure(int_port, PTC12_PIN, GYRO_INT_FLAGS);
gpio_pin_interrupt_configure(int_port, PTC12_PIN, GPIO_INT_EDGE_TO_ACTIVE);
gpio_init_callback(&gyro_cb_data, gyro_interrupt, BIT(PTC12_PIN));
gpio_add_callback(int_port, &gyro_cb_data);
```

4.3.3 Integrating gPTP functionality

Since working on the previous project, Zephyr had released a big update, going from version 2.4 to version 2.5. This update included a lot of different bug fixes, a few of them related to gPTP. This seems to have affected gPTP performance on the FRDM_K64F, as gPTP is now able to run in parallel with IP applications on the board with no issue, which was not the case in the previous project. This was verified by running a basic IPv4 application where gPTP functionality was simply included in the prj.conf file. This then communicated with a gPTP daemon running on the Linux host computer, which was set up following Zephyr's guide [30]. The packages being sent between the K64F board and the Linux host were then observed in Wireshark, and verified to function as expected, demonstrated in Chapter 5.1

With gPTP being functional, accurate PTP timestamps could then be added to the packets of gyro data being sent in the main application, following this workflow:

```
uint64_t avtptime;
struct net_ptp_time ptp_ts;
bool gm_present;

gptp_event_capture(&ptp_ts, &gm_present);
avtptime=((ptp_ts.second*NSEC_PER_SEC)+ptp_ts.nanosecond)%(1ULL<<32);
avtp_stream_pdu_set(pdu,AVTP_STREAM_FIELD_TIMESTAMP, avtptime);
```

4.3.4 Debugging Zephyr's packet socket library

However, when attempting to add gPTP functionality to the main application, additional difficulties were encountered. It became apparent that it was not straightforward to make the gPTP library cooperate with the *NET_SOCKETS_PACKET* library, which is used in the application to set up a simple packet socket. The issue was isolated into a test setup where the main file was empty, so only Zephyr's background processes were running. With a simple setup that only initialized IP and gPTP functionality, everything seemed to work as expected and the system replied to both gPTP and ping requests. Then, simply by adding the lines below to the prj.conf file, the system would no longer reply to either type of request:

```
CONFIG_NET_SOCKETS=y
CONFIG_NET_SOCKETS_PACKET=y
```

After contacting the Zephyr developers about this on GitHub, it was discovered that while this packet socket module was activated every incoming packet would be sent to this module, and would never arrive at the module they were meant for, which explains the observed behaviour. The developers then provided a branch to work on where Zephyr would look for other L2 network handlers such as gPTP in the system before dropping a packet.

This solved the isolated case described above, but when returning to the main application other issues appeared. After a socket had been created, the board would stop responding to gPTP and ping requests after a few seconds, and this error message would be printed for each incoming packet afterwards:

```
net_conn: pkt cloning failed, pkt <packet pointer> dropped
```

This means that the board is running out of packet buffers, and increasing the size of the available buffer only increased the number of seconds it took until the error occurred, pointing to a probable buffer leak. The buffer leak was verified by observing the buffer usage in Zephyr's net shell using the *net mem* and *net allocs* commands, which revealed that as soon as there was an initialized packet socket present, all incoming packets would stay in the packet buffer indefinitely. No packets seemed to get dereferenced even though all requests were handled correctly until this buffer was full. The explanation for this turned out to be that Zephyr would send every incoming packet to this initialized socket, and then *copy* them to other available network handlers where they would be handled properly. The packets arriving at the socket would then have to be handled manually to keep the buffer from overflowing, for instance by calling the *recv( )* function. As of right now, there is no way in Zephyr to create a "send-only" packet socket or a packet socket that filters out received packets by protocol, even though protocol is specified when the socket is created. This is unfortunate, as it should not be necessary for an application to spend processing power and time on clearing a buffer that is not meant to receive anything. In the project application this issue was dealt with by creating a thread that would regularly call the *recv( )* function in order to keep the rx buffer clean. The thread was implemented in Zephyr like this:

```
void buffer_cleaner(void){
    char packet_drain[128];
    while(1){
        zsock_recv(s,packet_drain,128,ZSOCK_MSG_DONTWAIT);
        k_sleep(K_MSEC(25));
    }
}

K_THREAD_DEFINE(BUFFER_CLEANER,1024,buffer_cleaner,NULL,NULL,NULL,7,0,8000);
```

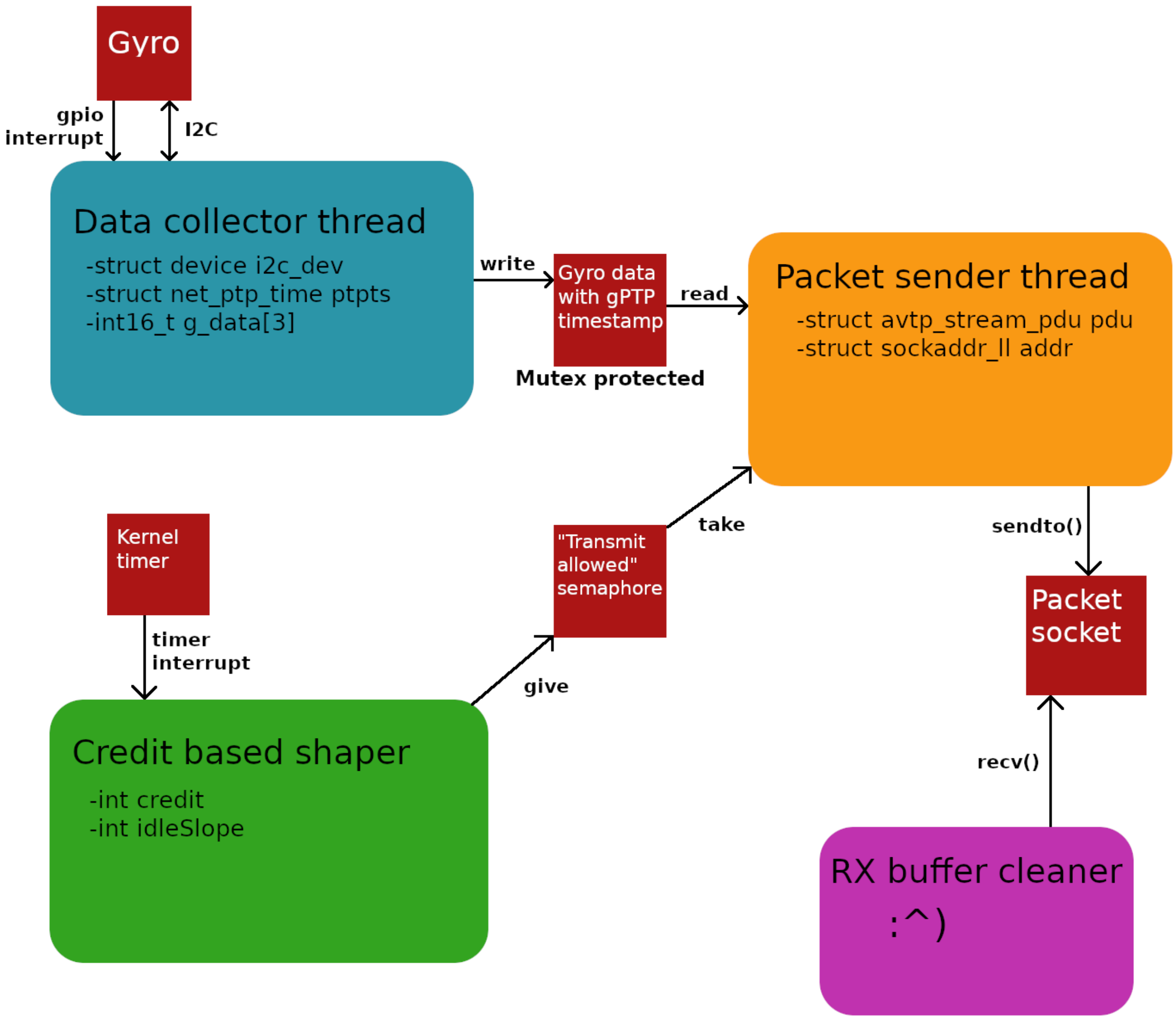


Figure 9: Diagram showing the threads in the application and how they interact

4.3.5 CBS, threading and Zephyr timers

Up until this point, collecting and sending data was triggered by the same interrupt. But with the hardware interrupt from the gyro being available these two processes could be split up into two threads and handled separately. Now one thread could wait for the gyro interrupt and collect its data together with an associated gPTP timestamp, while a different thread would get the responsibility of sending AVTP packets with gyro data at a regular pace. This pace would be set by a traffic shaper, which would run in an additional thread of its own.

One of the goals of this project was to implement a Credit Based Shaper following the definition in IEEE Std. 802.1Qav [5]. This was done by creating a thread that increments a credit value based on regular timer interrupts, that will signal the packet sending thread once the credit value gets high enough. For such a thread to function, a source for a reliable timer interrupt on the FRDM-K64F board in Zephyr is required. Zephyr has a few different APIs for generating timer interrupts, the most relevant ones being *timer* and *counter*. The *counter* API takes a hardware clock from the devicetree as a clock source, which is probably the best option for this project, but there aren't a lot of clocks available in the devicetree as is. One of the few that's there is the RTC, but this clock only gives one tick per second. There are other HW-clocks on the device that could be added to the devicetree manually, but doing this would require a lot of knowledge about the NXP timer structure and the Zephyr devicetree structure, which in a way defeats the purpose of using an environment like Zephyr. The *timer* API uses the kernel clock as its clock source. A lot of time was spent digging through Zephyr's source code and both NXP and Zephyr documentation in order to try to figure out how to set what clock to use as the kernel clock source. NXP microcontrollers have several options for this documented in their data sheets, but it was not obvious how to apply this information in the Zephyr environment. In the end the *clock_init* function was located in *soc/arm/nxp_kinetis/k6x/soc.c*, where the clock is set to run in PLL Engaged External (PEE) mode to generate the maximum 120 MHz system clock by default, which was also the desired clock configuration.

Since data was now written to and read from by different threads, mutexes were introduced to the project to ensure thread safe data handling. In Zephyr a mutex can be defined and used like this:

```
K_MUTEX_DEFINE(data_access);

k_mutex_lock(&data_access,K_FOREVER);
<read or write here>
k_mutex_unlock(&data_access);
```

Semaphores were also introduced to the project at this point, to communicate between threads. A semaphore can be created and used in Zephyr like this:

```
K_SEM_DEFINE(transmitAllowed,0,1);

//thread 1 waiting for signal from thread 2:
k_sem_take(&transmitAllowed, K_FOREVER);

//thread 2 signalling to thread 1:
k_sem_give(&transmitAllowed);
```

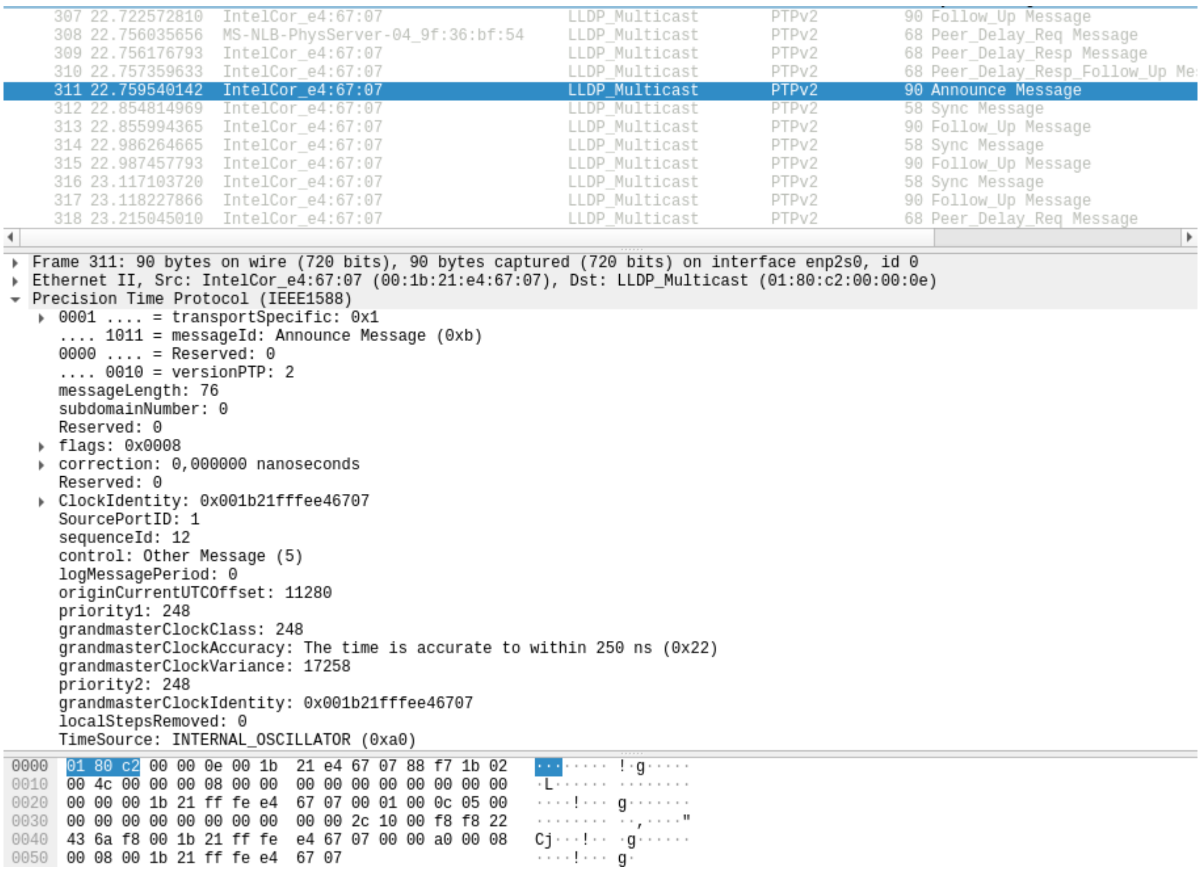
5 Results and observations

In this chapter it is shown that gPTP was able to synchronize the clocks of the K64F board and the desktop computer, and that the application is able to send real gyro data in AVTP packets. Then the timing performance of the system with a traffic shaper is observed.

5.1 gPTP packets captured in Wireshark

Figure 10 shows a few gPTP packets captured in Wireshark, and the contents of an announce message. These announce messages are sent from the GM each second once a gPTP domain has been established, and contain information about the current synchronization status.

Also visible in this screenshot are the packets calculating the propagation delay between the K64F and the desktop computer, as described in 2.1.2. Specifically the packet labeled 308 is a Delay Request message from the K64F to the GM, and 309 is the GM answering this request with a Delay Response message.



```
307 22.722572810  IntelCor_e4:67:07                 LLDP_Multicast     PTPv2       90 Follow_Up Message
308 22.756035656  MS-NLB-PhysServer-04_9f:36:bf:54  LLDP_Multicast     PTPv2       68 Peer_Delay_Req Message
309 22.756176793  IntelCor_e4:67:07                 LLDP_Multicast     PTPv2       68 Peer_Delay_Resp Message
310 22.757359633  IntelCor_e4:67:07                 LLDP_Multicast     PTPv2       68 Peer_Delay_Resp_Follow_Up Me
311 22.759540142  IntelCor_e4:67:07                 LLDP_Multicast     PTPv2       90 Announce Message
312 22.854814969  IntelCor_e4:67:07                 LLDP_Multicast     PTPv2       58 Sync Message
313 22.855994365  IntelCor_e4:67:07                 LLDP_Multicast     PTPv2       90 Follow_Up Message
314 22.986264665  IntelCor_e4:67:07                 LLDP_Multicast     PTPv2       58 Sync Message
315 22.987457793  IntelCor_e4:67:07                 LLDP_Multicast     PTPv2       90 Follow_Up Message
316 23.117103720  IntelCor_e4:67:07                 LLDP_Multicast     PTPv2       58 Sync Message
317 23.118227866  IntelCor_e4:67:07                 LLDP_Multicast     PTPv2       90 Follow_Up Message
318 23.215045010  IntelCor_e4:67:07                 LLDP_Multicast     PTPv2       68 Peer_Delay_Req Message
```

```
▸ Frame 311: 90 bytes on wire (720 bits), 90 bytes captured (720 bits) on interface enp2s0, id 0
▸ Ethernet II, Src: IntelCor_e4:67:07 (00:1b:21:e4:67:07), Dst: LLDP_Multicast (01:80:c2:00:00:0e)
▾ Precision Time Protocol (IEEE1588)
  ▸ 0001 .... = transportSpecific: 0x1
    .... 1011 = messageId: Announce Message (0xb)
    0000 .... = Reserved: 0
    .... 0010 = versionPTP: 2
    messageLength: 76
    subdomainNumber: 0
    Reserved: 0
  ▸ flags: 0x0008
  ▸ correction: 0,000000 nanoseconds
    Reserved: 0
  ▸ ClockIdentity: 0x001b21fffee46707
    SourcePortID: 1
    sequenceId: 12
    control: Other Message (5)
    logMessagePeriod: 0
    originCurrentUTCOffset: 11280
    priority1: 248
    grandmasterClockClass: 248
    grandmasterClockAccuracy: The time is accurate to within 250 ns (0x22)
    grandmasterClockVariance: 17258
    priority2: 248
    grandmasterClockIdentity: 0x001b21fffee46707
    localStepsRemoved: 0
    TimeSource: INTERNAL_OSCILLATOR (0xa0)
```

```
0000  01 80 c2 00 00 0e 00 1b  21 e4 67 07 88 f7 1b 02   ········ !·g·····
0010  00 4c 00 00 00 08 00 00  00 00 00 00 00 00 00 00   ·L······ ········
0020  00 00 00 1b 21 ff fe e4  67 07 00 01 00 0c 05 00   ····!··· g·······
0030  00 00 00 00 00 00 00 00  00 00 2c 10 00 f8 f8 22   ········ ··,····"
0040  43 6a f8 00 1b 21 ff fe  e4 67 07 00 00 a0 00 08   Cj···!·· ·g······
0050  00 08 00 1b 21 ff fe e4  67 07                     ····!··· g·
```

Figure 10: Wireshark screenshot of a gPTP Announce Message

5.2 Plot of gyro data from AVTP packets

Figure 11 shows a plot of raw gyro data extracted from AVTP packets sent from the K64F board. This demonstrates that the application in its current form is able to supply a stream of sensor data.
While collecting this data set, the board was first rotated around the x-, y- and z-axis individually for a few seconds each, then it was rotated around every axis chaotically before it was placed back on the table.

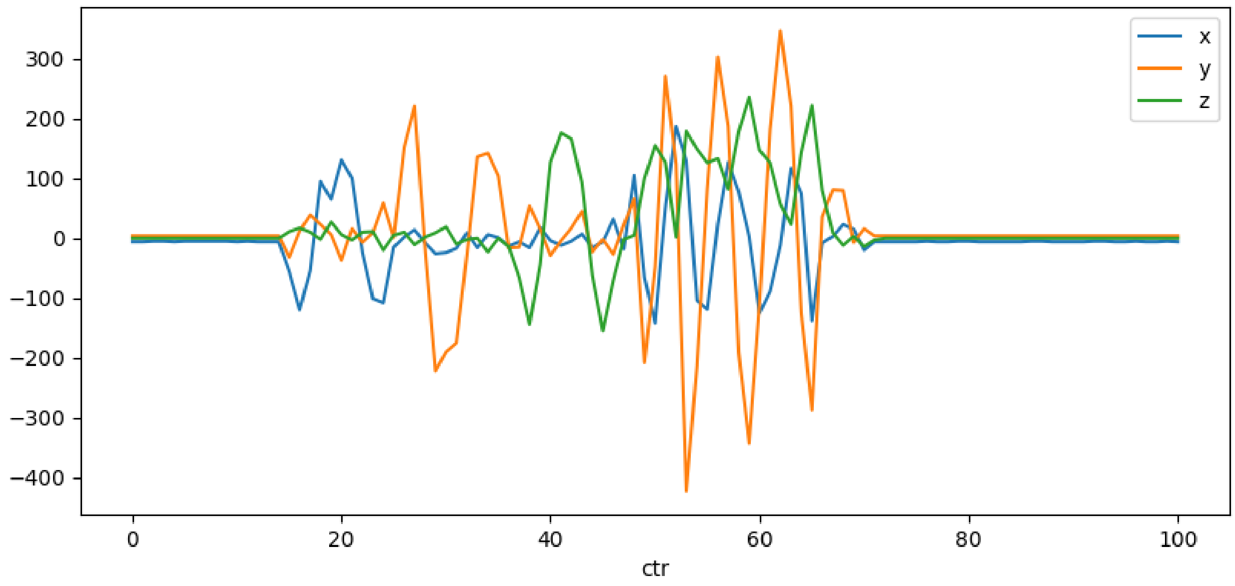


Figure 11: Plot of gyro data extracted from captured AVTP packets

5.3 Timing performance of traffic shaper

Some tests were run to see how well the implemented traffic shaper was able to queue packet transmissions at regular intervals. For these tests the goal was to send an AVTP packet every 250 ms. Below, plots of two different data sets are presented, created by collecting timestamps of incoming AVTP packets in Wireshark. After collecting series 1 it was observed that the average interval between packets was very slightly above 250 ms, so the send slope of the traffic shaper was made a tiny bit less steep in an attempt to fine-tune the shaper. Other than this, the code was identical while creating the two data sets. Figures 12 and 14 show the time since the previous AVTP packet arrived for each packet in these series, which was within ±1 ms of 250 ms for 99.8% of the packets. Series 1 had a mean value of 250.028 ms and a standard deviation of 0.22 ms, while series 2 had a mean value of 250.004 ms and a standard deviation of 0.18 ms.

Figures 13 and 15 show how precisely each packet arrives compared to if one packet had arrived at exactly every 250 ms. What becomes apparent from these plots is that in a few short time intervals, bursts of slightly delayed packets cause the packet arrival times to gradually drift away from the ideal behaviour.

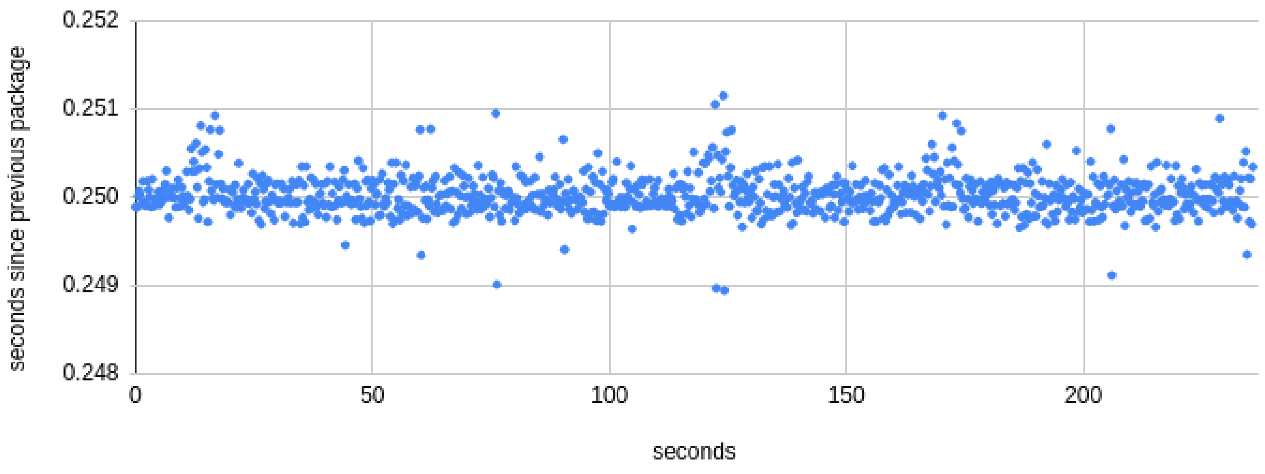


Figure 12: Plot of packet arrival intervals in series 1

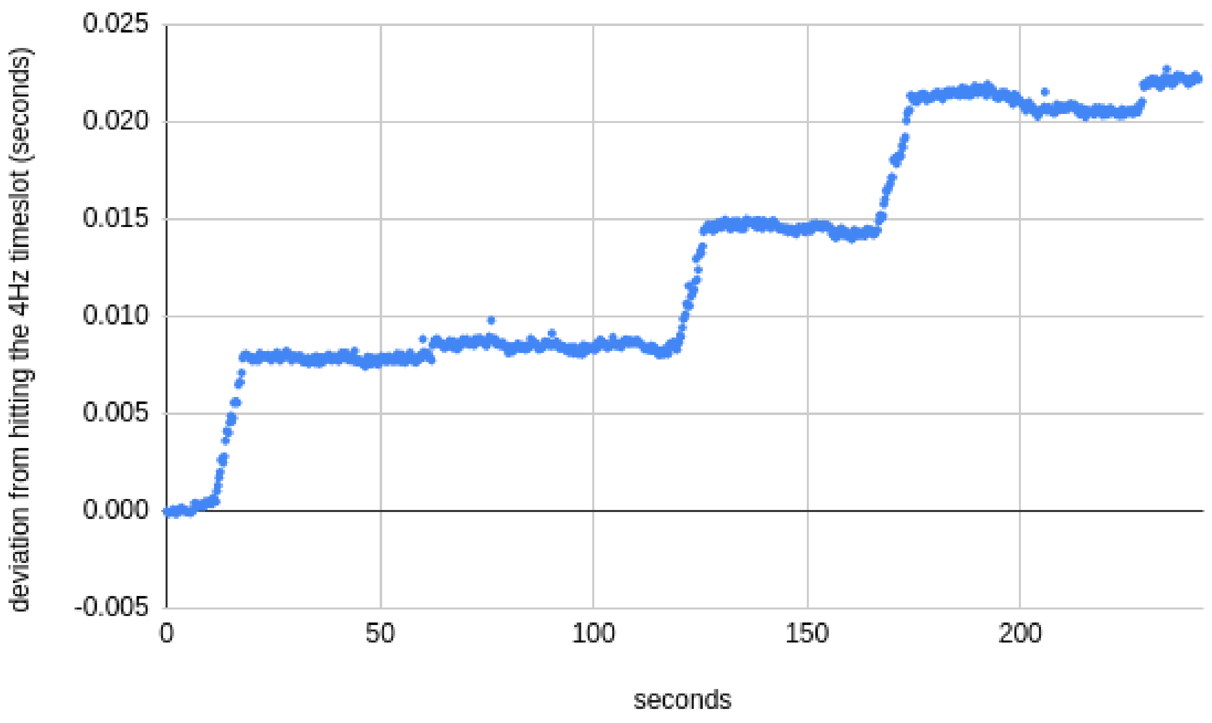


Figure 13: Plot of deviation from ideal packet arrival interval over time in series 1

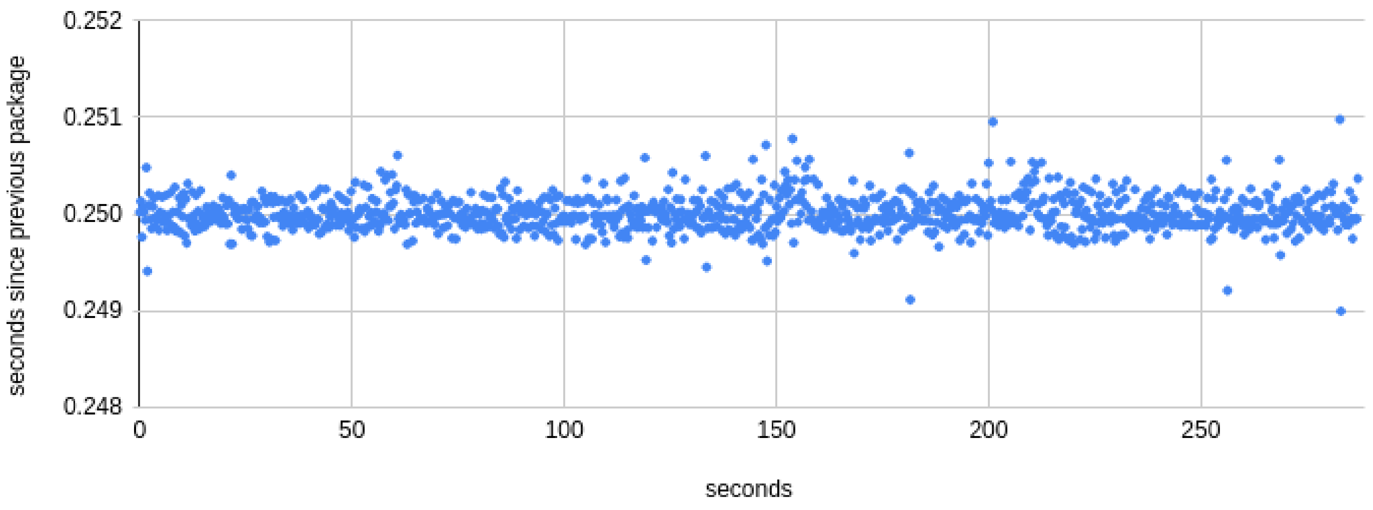


Figure 14: Plot of packet arrival intervals in series 2

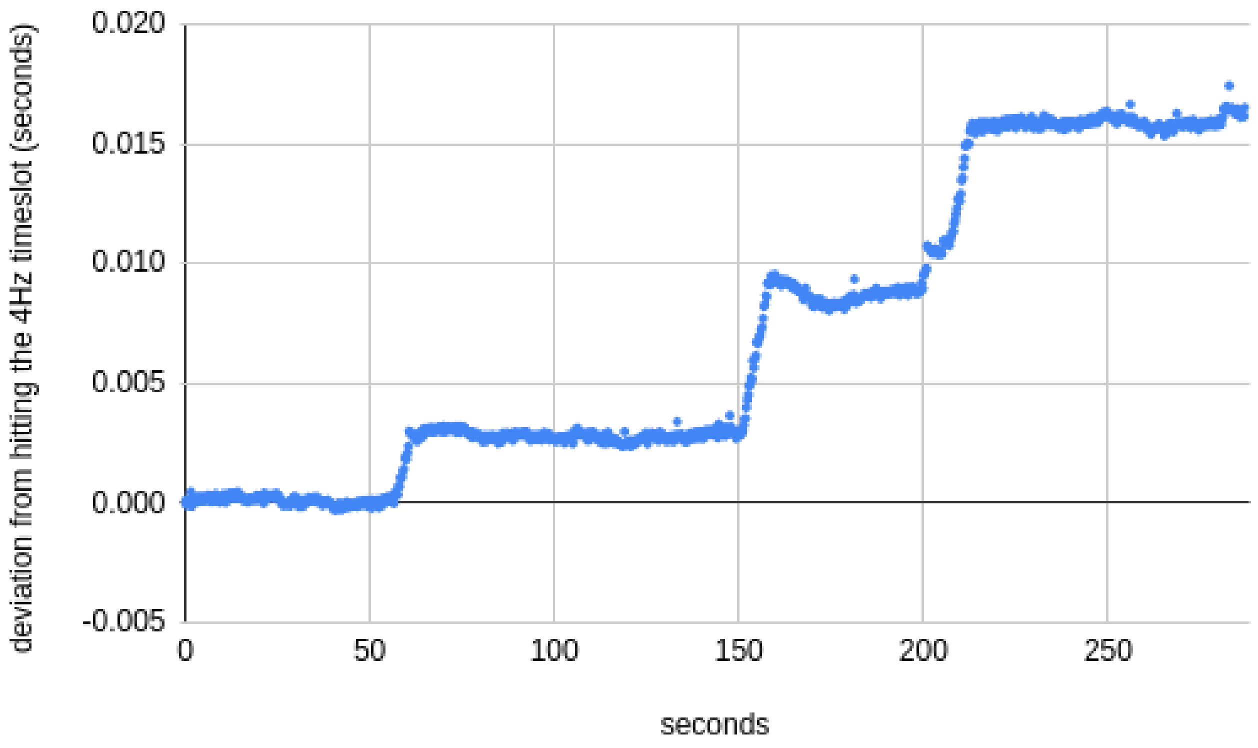


Figure 15: Plot of deviation from ideal packet arrival interval over time in series 2

It was also attempted to send packets at a 1kHz frequency, to see if the system would behave the same way at higher frequencies. In this case the mean interval was 1.008 ms and the standard deviation was 0.104 ms. See Figure 16 for a plot of this data set.

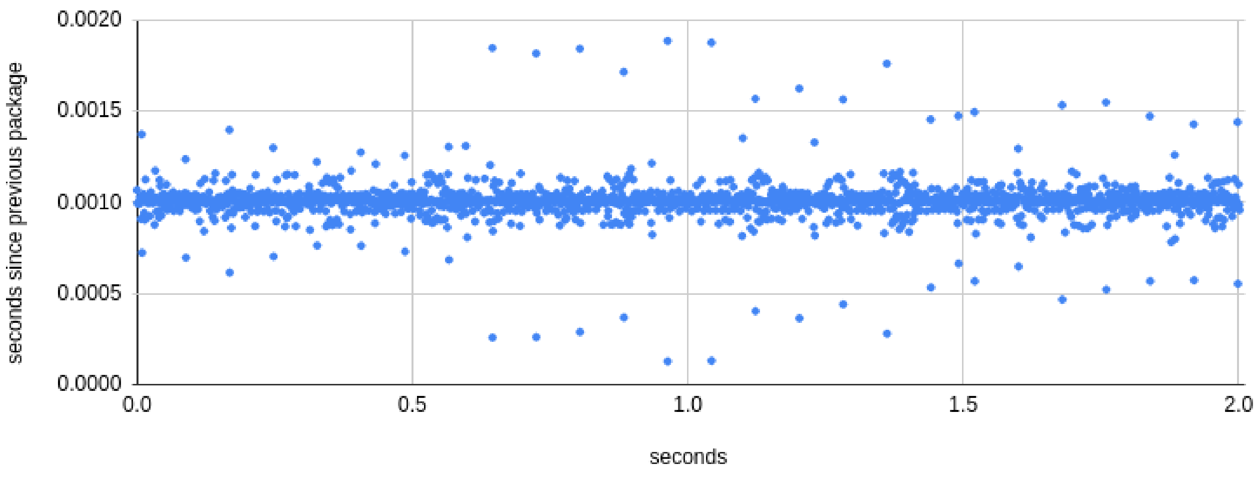


Figure 16: Plot of packet arrival intervals in series 3

	mean	max	min	std dev
4 Hz, #1	250.028 ms	251.149 ms	248.946 ms	0.22 ms
4 Hz, #2	250.004 ms	250.979 ms	248.992 ms	0.18 ms
1 kHz	1.008 ms	1.888 ms	0.127 ms	0.104 ms

Table 1: Table showing characteristics of plotted data sets

6 Discussion

6.1 Observations on the performance of the traffic shaper

The performance of a traffic shaper is usually investigated by observing the *measurement interval*, and whether the TSN stream stays below its permitted bandwidth use within this interval. This involves having a closer look at the stream's bandwidth use and potential burst size. In order to send a TSN stream of 1 kHz with the current frame size of 60 bytes, the needed bandwidth would be 480 kb/s. This is approximately 0.5% of the available data transmit rate on the network interface, which is 100 Mb/s, so even if packets were sent at for instance 4 kHz that would still only take up 2% of the available bandwidth. With an *idleSlope* of 2 Mb/s (2% of the bandwidth), the maximum interference of the biggest possible Ethernet frame would let the traffic shaper build up enough credit to send just a single packet, meaning the maximum burst would be a single packet of 60 bytes. It would then follow that the measurement interval would be the period of the target frequency, because practically what the measurement interval describes is how big packet bursts are allowed in a TSN stream. Based on this information, it might seem like the Time Aware Shaper could be a better fit for low bandwidth streams, since the flexibility of the Credit Based Shaper might not matter as much for such a stream.

At a few time intervals in the low frequency data sets, specifically around the 10, 120 and 170 second marks in series 1 and the 60, 150 and 210 second marks in series 2, there are grouped bursts of delayed packets. As can be seen in Figures 13 and 15, these bursts cause the packet arrival times to drift by a few milliseconds. With a credit based traffic shaper some variation in packet arrival can be expected, but there would be expected to be an equal amount of late and early packets as the credit system evens out the packet delays. What is observed instead is a periodic burst-wise increasing delay in packet arrival. This should not be caused by interfering traffic on the network device, as the traffic shaper as it is implemented will keep amassing credit based on interrupts from the kernel timer. It could be that this behaviour is caused by irregularities in the timer module, in which case it would advised to set up the *counter* API instead. Either way it would be interesting to investigate the cause of this behaviour further.

In Figure 16 the effect of the CBS can be seen clearly, in that delayed packets are followed by early packets, since the shaper has accumulated credit while waiting for the delayed packets to be sent. This effect is visible in the plot as mirrored bands in the upper and lower reaches of the plot. What's unexpected about this plot is how big some of the delays of the outliers are, up to almost 0.9 ms. In theory, the maximum interference on the network device should cause a delay of around 0.15 ms. It's possible that gPTP traffic is assigned a higher priority than the AVTP packets, which could force the application to wait for a burst of gPTP packets to be sent. Another possible explanation is that the packet sender thread has to wait for the gyro data mutex before it can load gyro data into a packet and send it. If this is the case, then it would be a good idea to load the data into the packet beforehand. This would require some restructuring of the code, but could be worth it.

6.2 Thoughts on developing in Zephyr

It is clear after working on this project that Zephyr is still a work in progress. Many features are yet to be implemented or need major reworks to really function properly. This was most noticeable when dealing with the packet socket library in this project, which didn't behave as expected when combined with other libraries. In the developers' own words, "the packet socket code is a bit convoluted and would need an overhaul in order to avoid these weird issues". Zephyr is under constant development though, and issues like this are being dealt with all the time. As the developers also mentioned, not very long ago enabling the packet socket library meant IP functionality wouldn't be available at all. Zephyr already released two big updates this year, v2.5 in February and v2.6 in June, which both added and changed hundreds of features. This is in no way an unusual development process for free open source software (FOSS), especially when it supports several different platforms like Zephyr does. So clearly there are some compromises to be made between the reliability of proprietary software and the freedom of open source software, with regards to ease of development and variety of features supported.

Another issue that often came up was the varying degree of documentation available for Zephyr's APIs and features. In particular it is often unclear what has to be included in an application's prj.conf file in order for it to function properly. If this file isn't configured correctly, the project will still build successfully, but the application might crash or behave unexpectedly in run-time, which leads to a lot of confusion while debugging.

That said, when Zephyr works it's proven to be an indispensable tool in developing multi threaded applications for embedded systems. It provides an incredibly useful abstraction layer for the developer in these kinds of projects, as long as the individual pieces are cooperating properly.

6.3 Progress towards a full TSN stack

With gPTP functional and running on the board, the packets being sent by the FRDM K64F are one step closer to being full fledged AVTP packets, meaning the packet stream is also closer to being a valid TSN stream. The packets now contain both real sensor data and a precise timestamp indicating when the data was collected. This is all the packets really need to contain in order to be useful. What can still be done in order to make better use of the AVTP format is to send additional information about the data in the format specific data fields in the AVTP header. For example, the header could give information about what type of data the packet contains, or how the data is formatted.

Another significant milestone for this project was having the gpio interrupt operational, which meant the functionality for collecting data and sending packets could be separated into different threads operating independently. This is important for several reasons. Measuring the gyro on a gpio interrupt from the gyro itself means that the timestamp associated with the data is guaranteed to be as accurate as possible, as opposed to when the data was measured on a timer interrupt. Keeping the sending of packets separate from the collection of data enables the application to time the packet sending much more accurately, making the traffic shaper a lot more effective.

There are still some pieces missing before a full TSN stack has been implemented. The CBS implementation in this project is not a general CBS implementation, but only implements traffic shaping for this application in particular. It would also be interesting to implement a Time Aware Shaper to compare the performance of the two shapers. The stream reservation protocol has not been implemented either, and will be necessary once the system will be tested in an actual TSN network.

7 Conclusion

In this project several steps were made towards implementing a TSN talker node in Zephyr. gPTP functionality was integrated into the project, allowing accurate synchronization of clocks in the network and timestamped data. The gpio interrupt from the gyro was set up, allowing the application to be split into independent threads. Traffic shaping was introduced to the application by implementing a credit based shaper. In its current form the application is able to supply a stream of sensor data with associated gPTP timestamps.

While working on integrating gPTP into the project, a bug was discovered that made gPTP nonfunctional while Zephyr's packet socket library was activated. This was successfully dealt with in cooperation with the Zephyr developers.

There is still a good amount of work left until a complete TSN stack is achieved. The CBS implementation in this project only implements traffic shaping functionality for this specific application, and isn't a general CBS implementation, which is definitely something that could be of interest. It could also be of interest to implement a Time Aware Shaper, and compare the performance of the different traffic shapers. The stream reservation protocol (SRP) has not been implemented either, and will be necessary once the system is ready to be tested in an actual network, which does not seem so far away with the progress that has been made in this project. Overall Zephyr seems like a promising environment for developing this kind of real-time embedded application.

8 Suggestions for future work

Topic 1 Add HW timers to the devicetree to use with Zephyr's counter API

Topic 2 Complete the Credit Based Shaper implementation

Topic 3 Implement the Time Aware Shaper (TAS)

Topic 4 Implement the Stream Reservation Protocol (SRP)

Topic 5 Utilize the format specific data fields in the AVTP header

Topic 6 Investigate timing performance further

Topic 7 Test a full TSN stack in a distributed network

References

[1] *AVnu FAQ*. URL: https://avnu.org/faqs/.

[2] *Ethernet Frame - Wikipedia*. URL: https://en.wikipedia.org/wiki/Ethernet_frame.

[3] *FreeRTOS - Wikipedia*. URL: https://en.wikipedia.org/wiki/FreeRTOS.

[4] "IEEE Standard for a Transport Protocol for Time-Sensitive Applications in Bridged Local Area Networks". In: *IEEE Std. 1722* (2016).

[5] "IEEE Standard for Local and Metropolitan Area Networks - Amendment 12: Forwarding and Queuing Enhancements for Time-Sensitive Streams". In: *IEEE Std. 802.1Qav* (2010).

[6] "IEEE Standard for Local and Metropolitan Area Networks - Amendment 25: Enhancements for Scheduled Traffic". In: *IEEE Std. 802.1Qbv* (2015).

[7] "IEEE Standard for Local and metropolitan area networks - Audio Video Bridging (AVB) Systems". In: *IEEE Std. 802.1BA* (2011).

[8] "IEEE Standard for Local and Metropolitan Area Networks - Timing and Synchronization for Time-Sensitive Applications". In: *IEEE Std. 802.1AS* (2020).

[9] *INTEGRITY - Wikipedia*. URL: https://en.wikipedia.org/wiki/Integrity_(operating_system).

[10] *INTEGRITY-178 - Green Hills*. URL: https://www.ghs.com/products/safety_critical/integrity_178_tump.html.

[11] *Libavtp*. URL: https://github.com/Avnu/libavtp.

[12] *Most Popular RTOSs - Lynx Software Technologies*. URL: https://www.lynx.com/embedded-systems-learning-center/most-popular-real-time-operating-systems-rtos.

[13] *Network Time Protocol - Wikipedia*. URL: https://en.wikipedia.org/wiki/Network_Time_Protocol.

[14] *Nucleus RTOS - Wikipedia*. URL: https://en.wikipedia.org/wiki/Nucleus_RTOS.

[15] *NXP FRDM-K64F*. URL: https://www.nxp.com/design/development-boards/freedom-development-boards/mcu-boards/freedom-development-platform-for-kinetis-k64-k63-and-k24-mcus:FRDM-K64F.

[16] *NXP FRDM-STBC-AGM01*. URL: https://www.nxp.com/design/development-boards/freedom-development-boards/sensors/sensor-toolbox-development-boards-for-a-9-axis-solution-using-fxas21002c-and-fxos8700cq:FRDM-STBC-AGM01.

[17] *Profinet - Wikipedia*. URL: https://en.wikipedia.org/wiki/Profinet.

[18] *PTP - Wikipedia*. URL: https://en.wikipedia.org/wiki/Precision_Time_Protocol.

[19] William Stallings. *Operating Systems: Internals and Design Principles*. Sixth Edition. Pearson Education International, 2009.

[20] *Synchronous Ethernet - Wikipedia*. URL: https://en.wikipedia.org/wiki/Synchronous_Ethernet.

[21] Michael Johas Teener. *No-excuses Audio/Video Networking: the Technology Behind AVnu*. 2009. URL: https://avnu.org/wp-content/uploads/2014/05/No-excuses-Audio-Video-Networking-v2.pdf.

[22] *Traffic Shaping - Wikipedia*. URL: https://en.wikipedia.org/wiki/Time-Sensitive_Networking#IEEE_802.1Qav_Forwarding_and_Queuing_Enhancements_for_Time-Sensitive_Streams.

[23] *Ubuntu*. URL: https://releases.ubuntu.com/focal/.

[24] *VxWorks - Wikipedia*. URL: https://en.wikipedia.org/wiki/VxWorks.

[25] *Zephyr - Application Development*. URL: https://docs.zephyrproject.org/latest/application/.

[26] *Zephyr - BSD Sockets*. URL: https://docs.zephyrproject.org/latest/reference/networking/sockets.html.

[27] *Zephyr - Devicetree*. URL: https://docs.zephyrproject.org/latest/guides/dts/intro.html.

[28] *Zephyr - Devicetree API*. URL: https://docs.zephyrproject.org/latest/reference/devicetree/api.html.

[29] *Zephyr - Getting Started*. URL: https://docs.zephyrproject.org/latest/getting_started/.

[30] *Zephyr - gPTP setup*. URL: https://docs.zephyrproject.org/latest/samples/net/gptp/README.html.

[31] *Zephyr - Introduction*. URL: https://docs.zephyrproject.org/latest/introduction/index.html.

[32] *Zephyr - NXP FRDM-K64F*. URL: https://docs.zephyrproject.org/latest/boards/arm/frdm_k64f/doc/.

[33] *Zephyr - West*. URL: https://docs.zephyrproject.org/latest/guides/west/.

[34] *Zephyr - Wikipedia*. URL: https://en.wikipedia.org/wiki/Zephyr_(operating_system).

Appendix A Source code

Here is a list of the files included in the appendix, and the functions that are found within them:

- **tsn_stream**
  - CMakeLists.txt
  - prj.conf
  - **src**
    - main.c
      - void cbs_timeout(struct k_timer *timer_id)
      - void gyro_interrupt(const struct device *dev, struct gpio_callback *cb, uint32_t pins)
      - static int init_pdu(struct avtp_stream_pdu *pdu)
      - void packet_sender_thread(void)
      - void credit_based_shaper(void)
      - void data_collector_thread(void)
      - void buffer_cleaner(void)
    - avtp.c/h
      - int avtp_pdu_get(const struct avtp_common_pdu *pdu, enum avtp_field field, uint32_t *val)
      - int avtp_pdu_set(struct avtp_common_pdu *pdu, enum avtp_field field, uint32_t val)
    - avtp_stream.c/h
      - int avtp_stream_pdu_get(const struct avtp_stream_pdu *pdu, enum avtp_stream_field field, uint64_t *val)
      - int avtp_stream_pdu_set(struct avtp_stream_pdu *pdu, enum avtp_stream_field field, uint64_t val)
      - int avtp_stream_pdu_init(struct avtp_stream_pdu *pdu)
    - gyro.c/h
      - void gyro_init(const struct device *i2c_dev)
      - void gyro_read(const struct device *i2c_dev, int16_t * g_data)
    - util.h
      - static inline uint32_t get_unaligned_be32(const void *p)
      - static inline void put_unaligned_be32(uint32_t val, void *p)

- **gptp_test**
    - CMakeLists.txt
    - prj.conf
    - **src**
        - main.c

            void main(void)