CODEN:LUTEDX/(TEIE-3112)/1-53/(2022)

Industrial Electrical Engineering and Automation

Bluetooth low energy mesh logger

- Time synchronization of nodes in a mesh network

Ossian Sandell
Oskar Olsson

Division of Industrial Electrical Engineering and Automation
Faculty of Engineering, Lund University

Abstract

There are many time synchronization systems in large data networks that are in the works around the world to make sure everyone sees the same time down to the granularity of sub seconds. When one dives deeper into smaller networks, like LANs and various networks with embedded systems, a need for higher resolution of granularity arises and resolution around the millisecond is desired. With embedded systems one can develop software for very specific tasks, such as sampling different kinds of entities. When these samplings are done in an environment that calls for quick response times (around the millisecond) there is a need for timestamping each sample. When having a set of data captured at the same time from different devices, for that data to be comparable with each other synchronization is needed. Synchronization can provide a certainty, that each data was captured within desired granularity of time and with respect to a common time line.

This thesis aims to make a prototype that does sampling from gyroscopes, accelerometers and compasses with as little as one millisecond difference between every unit that samples. Longer term tests to gather data of the characteristics of the Bluetooth low energy mesh were made. An algorithm was developed to determine the time it took for signals to travel within a bluetooth mesh network.

Keywords: IoT, Bluetooth Low Energy, Mesh, synchronization, nRF, Zephyr, SoC, 9-DoF sensor

Acknowledgement

The idea and foundation for this thesis was brought up by Adevo Consulting. We want to direct our gratitude and thanks to Mattias Wallinius and Maja Arvehammar at Adevo Consulting for all the support and resources they provided through the thesis work.

We also want to thank Anders Robertsson and Samuel Estenlund at Lunds University for quickly responding to us and providing help when we asked for it.

The thesis has been fun, stimulating and very relevant for our electronics education. As the work progressed there were many moments that showed us that we had a completely different understanding for which use cases the work could prove to be of assistance in. We were delighted to realize what the work could achieve in the industry of internet of things (IoT) and it motivated us to research even more of the possibilities of embedded systems and real time operating system.

Contents

Abbreviations

List of abbreviations and their meaning

API - Application Programming Interface

BLE - Bluetooth Low Energy

CPU - Central Processing Unit

GATT - Generic Attribute Profile

GPIO - General Purpose Input Output

IC- Integrated Circuit.

IDE - Integrated Development Environment

IoT - Internet of Things

I2C - Inter-Integrated Circuit

nRF - Nordic Radio Frequency, a product line from Nordic Semiconductors.

RTC - Real Time Clock

RTOS - Real-Time Operating System

SIG - (Bluetooth)Special Interest Group

SoC - System On a Chip

SDK - Software Development Kit

SPI - Serial Peripheral Interface

TAI - International Atomic Time

TTL - Time To Live

UML - Unified Modelling language

9-DoF - Nine Degrees of Freedom

1 Introduction

This thesis will investigate and develop synchronized logging with BLE mesh communication for the sake of Adevo Consulting AB. The research has been done at Ideon Lund with the help and supervision of Adevo and through the Division of Industrial Electrical Engineering and Automation.

1.1 Background

Adevo are interested in the Nordic Semiconductor nRF series of cards for development and to investigate their capacity to communicate through a BLE mesh. They want to have the communication synchronized and tested to see if it reaches a certain standard. If it is possible to reach the desired standard then base code for synchronizing and logging data should be developed. There are a lot of applications for this if it is proven feasible.

1.2 Project Aims

The main aim of the project would be to investigate and prove that synchronization and collecting of data could be done at a threshold of a millisecond. If done successfully, the secondary aims are developing base code for sensors to log data synchronized in a BLE mesh setup and find an implementation to transfer that data to a cell phone.

1.2.1 Research questions

From the basis of the project aims the following questions should be answered.

1. What are the limits of Bluetooth mesh networks and how to synchronize unit clocks to assert millisecond resolution?
2. How to interface and develop drivers for I2C or SD sensors for Nordic's development kit boards(nRF52840 nRF5340) and Zephyr OS?
3. What are the limitations of logging storage and how should data and time stamps be stored?
4. How to transfer data and timestamps in an efficient way to a mobile phone with full security?

1.3 Outline

The four questions in section 1.2.1 should be done in their started order with respect to time and if the implementation of the synchronization takes longer time than expected then that should be prioritized to complete before moving on to the next step. The project is built so that the next step can not be taken unless the previous one is completed. The software will be programmed in Zephyr OS which is linux based. The software will be developed on a system-on-a-chip (SoC) of model nRF52840 and be written in the Connect SDK development tool for Visual Studio Code. C-guidelines will be followed. All devices are chosen and provided by Adevo Consulting AB. In chapter 2 the theory about Zephyr and nRF is given. This chapter also delves into BLE and its mesh some theory about the components of the project is laid out. Chapter 3 is about experimental setup and describes how the project was approached. The result are then presented in chapter 4 and discussed in chapter 5. Finally section 6 contains the conclusions of the thesis and also contains possible extensions for future work.

2 Theory

In the theory section of this thesis description and functionality of important aspects of this project is given. This section highlights what has been important to understand for completing the project and also makes it easier for the reader to understand dilemmas and suggested solutions.

2.1 Zephyr

Zephyr is an open source and linux-based project. Information about Zephyr is gathered from their official website (Zephyr,2022)[9]. The Zephyr OS is designed for use on resource-constrained and embedded systems. Its kernel supports multiple chip architectures including ARMv8-M (Cortex-M) which the nRF52840 and nRF5340 are built around (see section 2.2.1 and 2.2.2 for more details about each board).

Zephyr is a real-time operating system (RTOS). This RTOS is designed with Bluetooth low energy (BLE) in mind, making Zephyr perfect for the energy constrained devices used in the project. BLE is a "low energy" protocol of communicating through Bluetooth that requires certain features for it to be used and implemented. Zephyr provides what is necessary for BLE to be implemented and uses it as a form of wireless communication. The code in Zephyr is written in the programming language C, Zephyr i primarily written in C (some C++) and supports natively applications written in the C language. To utilize Zephyr's functions it is important to understand the following aspects of embedded systems: flags, devicetree, overlay, interrupts, callback and configurations.

2.1.1 Flags

When creating I/O with Zephyr's GPIO the communication is modifiable through flags. These flags include input/output, different ways on how the communication is triggered and configuring interrupts.

2.1.2 Devicetree

A devicetree in Zephyr is a hierarchical data structure that describes the hardware of supported boards. As the name suggests, devicetree is a data structure with nodes in a tree fashion and is accessible through Zephyr in human-readable text format.

2.1.3 dts

dts stands for devicetree structure and is a name for devicetree bindings, which is the grouping and implementation of devices to a specific file. This file declares each requirement of the belonging device.

2.1.4 Overlay

If changes to the devictree is needed, i.e., for defining new inputs and outputs, an overlay file can be created. In an overlay file you can define new pins for I/O or edit existing ones, as well as making aliases for the pins for easy access from the source files. The overlay file is compiled together with the dts file and can thereafter be flashed onto the board.

2.1.5 Interrupts

An interrupt service routine (ISR) in Zephyr allows you to execute a function asynchronously in response to a hardware or software interrupt. The ISR seizes the control of the current thread in work with very little overhead.

2.1.6 Callback

A callback function is made to be available for other functions. It is executed right after when its associated pin is activated or triggered. With callback you can specify how you want the function to communicate with the chip on its board.

2.1.7 Configuration of GPIO

When declaring the functionality of a certain peripheral, for example a certain I/O-pin and on which flank it should trigger and if it should be an input or an output. The configuration is a function which often returns an integer of 0 if the implementation was successful, and in that way it is possible to detect an error by implementing an *if case*. This is an example of how GPIO can be configured, and configuration is also done for external units and models in similar fashion.

2.2 nRF

In this section, explanation about Nordic and the nRF52/53 series is given. The section attempts to demonstrate the interesting aspects of using two different types of cards and the interest in comparing the results. Information about Nordic and their products are retrieved from their website (Nordic,2022)[1].

Nordic Semiconductor is a Norwegian fabless [1] semiconductor company specializing in wireless communication technology that powers the Internet of Things (IoT). Nordic offers a variety of processors with wireless multiprotocol System-on-Chips for development purpose. For this project the work has been on the nRF5340 and nRF52840. These boards are SoC's that pack multiple functionalities, such as Bluetooth low energy, Bluetooth mesh, NFC and multi-threading.

SoC	nRF5340	nRF52840
Bluetooth	5.3	5.3
Thread	Yes	Yes
Matter	Yes	Yes
Zigbee	Yes	Yes
Bluetooth Mesh	Yes	Yes
Flash	1 MB + 256 KB	1 MB
RAM	512 KB + 64 KB	256 KB
CPU	128 MHz Arm Cortex-M33 + 64 MHz Arm Cortex-M33	64 MHz Arm Cortex-M4 with FPU

Figure 1: Comparison of our Bluetooth LE SoCs. The figure is retrieved from (nRF, 2021)[10]

In Figure 1, the prestanda of the two cards can be seen. The two cards CPU performance makes it possible for different configurations of the 3 Bluetooth low energy layers (see Section 2.4.2 for BLE layer explanation), nRF52840 can

[1] Produces schematic and sale of semiconductors, but the production is outsourced.

have single-chip configuration and the nRF5340 can have dual-chip configuration. The two configuration explanations are gathered from (Zephyr,2022)[3] and are described as the following:

Single-chip configuration: Meaning that all layers of BLE and the application are run on the same chip. This means less power consumption since all the layers only need one chip to run but it may affect the processing time.

Dual-chip configuration: The application and host will be run on the same IC while the Controller and Radio Hardware will be run on another meaning there are more stacks to process the communication within a mesh, possibly lowering the processing time.

2.2.1 nRF52840

The nRF52840 has a single Arm Cortex-M4 CPU leading to Single-chip configuration which means host, controller and radio hardware will be running on the same IC. The nRF52 and nRF53 Series are all-flash based SoCs. The information is gathered at Nordic product description of *nRF52840* [10].

2.2.2 nRF5340

The nRF5340 has SoC with two Arm Cortex®-M33 processors allowing Dual-chip configuration meaning two separate ICs, one running the Application and the Host, and a second one with the Controller and the Radio Hardware. The information is gathered at Nordic product description of *nRF5340* [11].

2.3 IDE

The nRF product line has several possibilities for platform development where you can flash, build and debug code through Zephyr's meta-tool, "West". Zephyr give a basic introduction about West on their website in *basics* (Zephyr, 2022) [2]. West allows you to use command lines to work with projects, most commonly Git repositories, under a common workspace directory. Segger Embedded Studio (SES), which is very widely used to interface embedded systems, has an "nordic" edition which provides interface to nRF's SoC devices. Since 2021, Visual Studio Code also has an extension called "nRF connect" which also brings a complete IDE experience for developing code aimed for nRF chips.

2.4 Network

In this project there will be a network for devices to communicate, and here we describe the general theory of networks and then describe the type of network communication that is used.

2.4.1 Synchronization algorithms

The following synchronization algorithms and their description have been retrieved from a master thesis made on the subject from *Time Synchronization in Short Range Wireless Networks* [8]. In any network of nodes that do any type of sampling it is important to have the samples collected from all the nodes synchronized in time and having the data collected without any sort of timestamps makes the data more or less worthless.

In wired networks of master-to-slave or client-to-server there are established time synchronization protocols that are quite accurate and precise, such as the Network Time Protocol (NTP) that was first released in 1984. NTP can synchronize its network packets between nodes down to 200µs from each other and in some cases even as low as tens of µs (in its newest version NTPv4). With these qualities, NTP is widely used to synchronize clocks in PC's. Two examples of algorithms will be briefly explained to give an idea on why synchronization may be needed.

PTP

Precision Time Protocol (PTP), also used over wired networks (proven to work wireless in LAN too), can have synchronization down to submicro second which makes it useful in automated control systems. In a client-to-server network with PTP the client sends out a broadcast containing a time message $tc1$, the server receive that broadcast and timestamp the time $ts1$ when receiving it. Then the client sends out a follow up message including the actual time $tc1$ for broadcasting it. After this the server periodically send out a "delay" message with the transmission time between the client's follow up message and the server's "delay" message $ts2$. Finally, the client responds with a delay response message with the timestamp $tc2$ of receiving the delay message from the server. With these timestamps, the server can now calculate the delay between client and server according to Formulas 1 and 2. Visualization of PTP can be seen in Figure 2.

$$D_{cs} = t_{s1} - t_{c1} \tag{1}$$

$$D_{sc} = t_{c2} - t_{s2} \tag{2}$$

The one-way delay D_w and *Offset* is calculated with:

$$D_w = t_{c1} + \frac{t_{sc}}{2} \tag{3}$$

$$\mathit{Offset} = D_{cs} - D_w \tag{4}$$

The server adjusts its clock according to *Offset*.

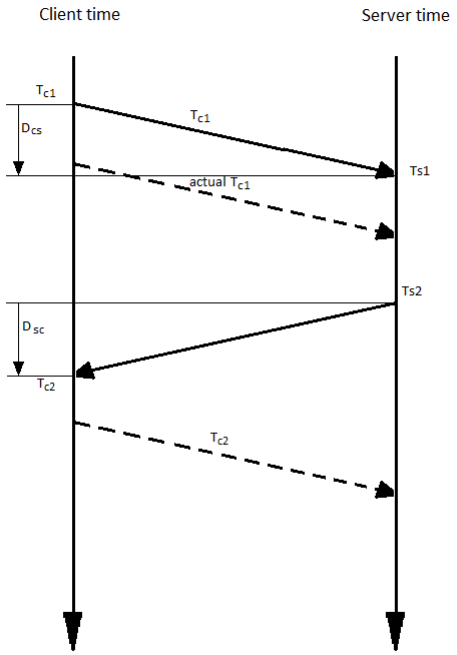


Figure 2: The different timestamps of PTP.

RBS

In wireless networks, such as Wifi and Bluetooth, often with battery driven devices, time synchronization becomes a question of energy consumption. Reference Broadcast Synchronization (RBS) is a protocol for wireless broadcast networks where the receivers estimate each other's clocks. So in the client-server example mentioned in Section 2.4.1, not only is there to be a broadcast message from the client containing a starting point, but also communication between the servers to make sure every server has the same starting point in time. The delays and offsets with the broadcast message caused by the irregularities in time taking to handle the message package in the receivers and senders network stack. With RBS, when timestamps occur at the receivers, resulting in being able to negate the senders time delay and only the receivers package handling needs to be accounted for in the synchronization.

2.4.2 BLE

BLE is a further developed version of the classic Bluetooth, included at version 4.0 developed by Bluetooth Special Interest Group (SIG). It functions as the classic Bluetooth but with more possibilities of implementation and less energy demanding. What makes BLE more energy conserving than regular Bluetooth is that it lets its devices turn the radio controller off to "sleep" and only turns it on when needed. What further makes BLE differ from its classic version is the device communication, where the latter only have point-to-point communication; whereas the former also have broadcasting of messages. Bluetooth official website has an introduction on their *BLE mesh* variant [4].

BLE is implemented by Zephyr to establish a mesh network. Zephyr explains on its website that there are three main layers that together make up the Bluetooth Low Energy protocol stack. The layers are Host, Controller and Radio Hardware and they all need to be implemented for the device to use BLE. Zephyr has many libraries, by configuring and enabling certain libraries with macro APIs they can be included in the build of an application. This grants usage of predefined functions that can be used in said application. This three layer hierarchy can be found in Zephyr's *documentation*[9].

Host

Host sits right below the application layer, meaning that when coding in the application one cannot edit the host layer. The host consists of several network and transport protocols that gives the device ability to communicate with each other.

Controller

The Controller implements the link layer of the OSI-model, and is a low level, real-time protocol that provides standard communication. The link layer guarantee the delivery of packets.

Radio Hardware

Hardware implements the required analog and digital baseband functional blocks that permit the Link Layer firmware to send and receive signals in the 2.4 GHz band of the spectrum.

2.5 Bluetooth mesh

Bluetooth mesh was released officially in July 2017, and was developed and specified by Bluetooth SIG. According to Ericsson's whitepaper about mesh published 2020 it is defined as: *The Bluetooth Mesh Profile standardizes a full stack connectivity solution for mesh networking, extending Bluetooth applicability for IoT use cases*[14]. It is built upon Bluetooth LE 4.x and requires that to work and it broadcasts data over Bluetooth low-energy to specified address.

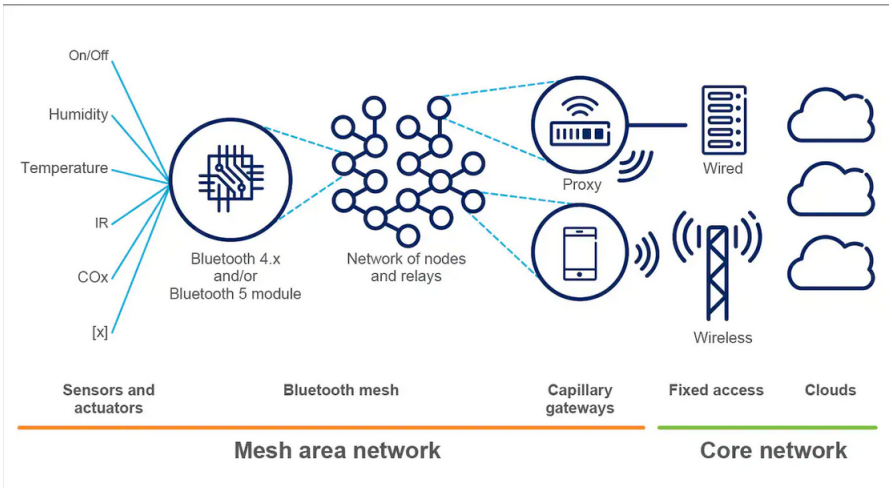


Figure 3: Mesh network

Figures 3 and 4 are retrieved from Ericssons whitepaper [14] about mesh. They describe the Bluetooth mesh concept. It allows for a mesh network to be set up between devices (called nodes once provisioned), where communication is done by relaying. Relaying in a Bluetooth mesh network is based on a managed flooding communication model. This means a message is forwarded by nodes until it reaches its destination. The relaying of a message within a network is known as a hop. A message may hop x amount of times, varying on the Time To Live(TTL) of the message. This form of communication allows a capillary network to be formed between nodes. A capillary network is described as the following by an article from (Ericsson, 2014)[5] : *"... a local network that uses short-range radio-access technologies to provide local connectivity to things and devices"*. This network extends the reach of its devices, allowing communication between nodes that are not within direct radio reach of each other.

Bluetooth mesh standardizes communication and control actions within this network and the communication within the network is through BLE. Nodes in the network can be added at different times and can talk to each other directly. The network does not need any centralized operation and no coordination is required. Addresses to certain nodes or groups of nodes are easily configured through the provisioner. The provisioner is often a smartphone.

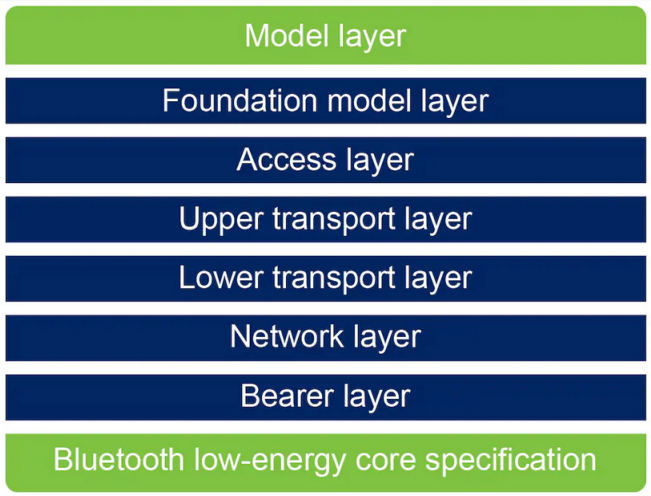


Figure 4: Architecture of the Mesh

In Figure 4 the seven layers are defined. Much like the OSI model[2], there are seven layers where each layer has its own task. In short, they all work together to take data, packet it, authenticate, encrypt and assemble it when transmitting a message. The layers also work together when receiving a message, they decrypt, authenticate, handle segmentation and access the data in the message. There are more functionalities to the mesh model but it is not brought up in this thesis.

[2]The Open System Interconnection (OSI) model consists of seven layers model of data communication

2.5.1 Provisioning a device

The process of provisioning devices is as follows; an unprovisioned device will beacon Bluetooth signals, thus becoming visible to the provisioner. The provisioner is the owner of a network and by provisioning a device the provisioner will add this device to its own network making it a node. The provisioner, which is often a smartphone, is a node that implements the configuration client model. A model in this context means to define the basic functionality of a node, the model context is further explained in Section 2.5.3. The provisioning is done by the provisioner sending an invitation and the device responding by presenting its capabilities. The provisioner and the device exchange public keys, either in-band or Out of Band. (Out of Band control signals are not being sent on the same frequency as the data, with in-band the control signals are on the same bandwidth as the data signals.) The project will provision using OOB for exchange of public keys. Provisioning is described in more detail on Zephyrs website about *provisioning*.[13]

2.5.2 GATT

When it comes to exchanging data between a server and a client, it is easy to just send a simple data packet with the data you want to send. There is however a whole protocol that defines low-level interactions, named Generic Attribute Profile (GATT). In this protocol, every single item of data exchanged between servers and clients must be formatted, packed and sent accordingly. This makes for compatibility, especially with values of sensors. With GATT it's easy to develop applications and firmware to read sensors. GATT's architecture is shown in Figure 5 and is retrieved from O'Reillys article on GATT. [7]

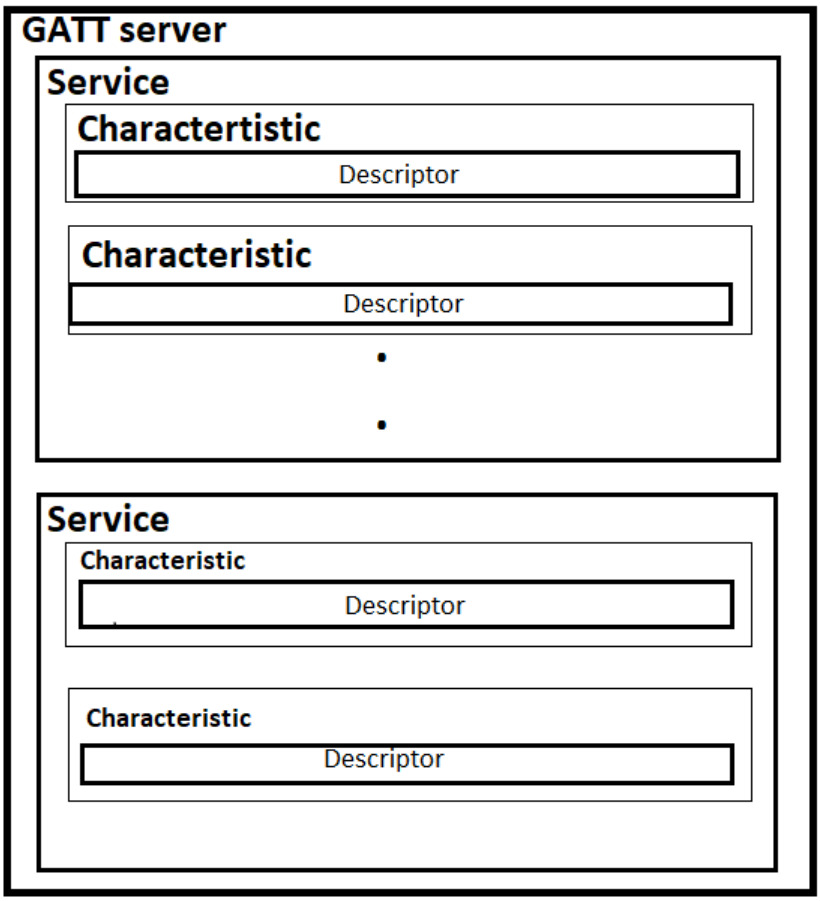


Figure 5: Architecture of Generic Attribute Protcol (GATT).

Service:. The service describes certain amount of data contained within characteristics, the amount of characteristics contained within a service may vary. Each service is distinguished through a specific number ID called Universally Unique Identifier (UUID).

Characteristic: Is a container for user data, and includes at least two attributes: the declaration and the value. The declaration provides metadata about the value.

Attribute: This is the metadata itself and attributes contains the data that's supposed to be transmitted

2.5.3 Models in Bluetooth Mesh

Mesh has several predefined models for different purposes. These models standardize the communication between devices from different vendors, the models represent a specific service and define messages that act on a set of states. As Woolley states in the article *Mesh Model Overview*[15], *"... from a network's point of view, models make the device what it is"*. In applications, these models are gathered in one or more elements, acting as a virtual entity in a mesh network. When provisioning a device with a certain model, one can set its own unique unicast address. There are many models available in Zephyr's repository, such as light control, sensor reading, and device configurations, but it is often necessary to define a custom model that handles messages according to one's need.

Model ID

The predefined models use their own reserved vendor model IDs to be included in the network package's opcode. Opcode should be included in the head of the packet that is the message. Opcodes is a number of bytes that tells the hardware what operations should be done with the message it comes with. The different IDs for the models are necessary to identify the origin of the message between the nodes in the Mesh, so one node easily knows how to handle the message.

Predefined models

In this thesis, several models were used. They provide very basic functionality, and are described below.

Configuration model

To configure a mesh network there has to be a provisioner to connect the unprovisioned devices to the mesh. These models provide parameters to the node, encryption keys and any extra features if there are any defined. This model is an important part in making the communication in the Mesh in a secure fashion, and the Mesh becomes resilient against various threats and issues, including replay attacks, man-in-the-middle attacks and trash-can attacks. They are mentioned in Bluetooth's blog abut *provisioning* [13].

Health model

This model looks after the attention state of the devices. As its name indicates, it is primarily used to report errors within the mesh and to provide node diagnostics.

Time model

Time models provide functions for date synchronization with granularity of a 1/256th second. The time measurement is based on the International Atomic Time standard. The time model client role is to send out time synchronization messages to its respective Time model servers whenever it is needed.

Custom model

Custom models need their own ID's, called universally unique identifier (UUID), which is 128 bits. There are macro API's in the Zephyr library that let you define ID's in C code. Depending on what the purpose of the model is, the way the model sends its messages, how it should be received and associated is defined through macro API's.

2.6 Components of the project

The project will have components connected to the development card once the base code is completely developed. One of these components is a RTC which purpose in the project is to detect drift of the internal clocks making it possible to compensate. The other is a 9-DoF sensor which will be collecting data at each server once synchronization within the mesh is done.

2.6.1 The system clocks

The nRF boards each has an internal clock which will start counting from when the system started. These are what the system uses to track time. This project will also use an external hardware clock refered to as RTC and this clock can be battery driven and has a higher precision than the internal clock's. The clock uses the entity *ticks* which depends on frequency of the clock. For example, a clock with the speed of 32,768 Hz will have the value for one tick to be $\frac{1}{32678} = 30517ns$.

The internal clock

The nRF52840 and nRF5340 use a quartz crystal for the internal clock. This type of crystal is prone to drifting due to fluctuations in air pressure and temperature. From the data sheet of the cards nRF52840 and nRF5340 specification of there clock can be found, see Table 1.

Table 1

Data about each cards clock			
nRF 52840	32 MHz crystal	SMD 2520, 32 MHz, +/- 10 ppm	Applies only when HFXO is running
nRF 5340	32 MHz crystal	SMD 2016 32 MHz ftol= ±30 ppm	Only applies when the high frequency crystal oscillator (HFXO) is running
nRF 5340	32 kHz crystal	SMD 2012 32.768 kHz ftol= ±20 ppm	Only applies when the low frequency crystal oscillator (LFXO) is running.

RTC

This project will use an external RTC produced by Maxim Integrated named DS3231. Information is gathered from the *data sheet*.[6]

The DS3231 is a high precision real-time clock applicable at server communicating through I2C with a Temperature Compensated Crystal Oscillator (TCXO)

which compensates for drifts caused by temperature fluctuations.

The crystal in both the internal and the external TXCO, has about 100ppm accuracy, which if a clock based on the crystal is on for 24 hours would produce about 8.6 s error (100 ppm translates to an error of 0.0001 seconds every second).

2.6.2 9-DoF

Synchronization would allow data logged and stored with time stamps and the data should in our case be collected from a 9-DoF sensor. The sensor used was the ICM-20948 produced by TDK InvenSense, see their website for a short description and link to *data sheet* [**9-DoF**].

The IMC-20948 is a low power 9-axis motion tracking device suitable for IoT applications. The device packs a 3-axis gyroscope, 3-axis accelerometer and a 3-axis compass. Communication is done through SPI at 7 MHz.

3 Experimental setup

In experimental setup the tests and their setup will be explained and how each test was approached. One reason for the tests was to investigate the possibility of time synchronizing nodes (see Section 2.5.1 for node description) within a mesh network (see Section 2.5) to a tolerance of one ms. The devices in these tests are the nRF52840 and nRF5340 (Section 2.2). The role of the nodes within the mesh was client and server,respectively. The server may be referred to as a logger since its main purpose is to log data.

Another reason behind these tests, is that with every test, there were useful code that could later be included in the final build. Whenever each test was implemented, the documentation on the subject being implemented was read for understanding and guidelines. The documentation could be from either Nordic or Zephyr, see Sections 2.2 and 2.1 for explanation of important aspects of each documentation.

3.1 Introduction to testing of synchronization

The idea was to find out if BLE could make a mesh hop (see Section 2.5) below one ms. To test this possibility devices would be connected with cables and also through a BLE mesh network. The communication went through both the BLE mesh and through cable. In the mesh network one device would be the client and the other devices connected would be the loggers acting as servers. The client would send a signal to the servers and each server would respond through a cable connected to the client. The signal via the cable would tell the client that the device had received the signal. The client would store the information on a log at what time it got a response from the server. This log would then be analyzed for deciding if the signal was within the tolerance of one ms making it acceptable as a synchronization signal. This would be the setup for investigating signal's transfer time in a BLE mesh, also known as a mesh hop.

The test would have to be going on for a longer time to ensure that there would not occur any errors in code or components. During this longer test data from logs would be collected and stored. This data would serve as background for statistical evidence.

Before setting up the test for investigating mesh hop time and gathering data for statistical evidence, it was necessary to investigate the time it took for signals being transferred and executed. The interesting signals that were being sent and processed was when the client sent a signal to the servers through the BLE mesh and also when the servers responded by sending a signal through cable back to the client. It was critical to see if and how these time consuming actions would affect the time for a signal being sent through BLE mesh. Each signal was isolated and tested on their own before all of them were incorporated in the test bench trial where multiple nRF-cards would be up and running over a longer period of time.

3.1.1 First test: Speed of the interrupt

The purpose of this test was to measure the processing speed of handling interrupts see Section(2.1.5). Starting from a pre-built sample "button" by Nordic Semiconductor and modifying it by configuring GPIO's (2.1.7). One pin was set to work as an output and the pin would go high if the button's callback (2.1.6) was triggered. On the same device an input pin would be set to flag (2.1.1) on the rising edge[3] of the incomming signal.

After completing configuration of I/O's for the pins the next step was connecting the output pin and the input pin with cable and then send a signal from the output pin by pressing the button. To measure the time it took for the signal traveling between the pins. The signal was registered at the input and subtracted by the time it was registered at the output, The times were registered via the callback of each GPIO. The purpose of this implementation was that it would register the time for a signal being sent over cable. This would allow cable communication between devices during the test bench trial, since it could be stated that the time it took for the interrupt handling and travel through cable could be neglected. The result of the test can be seen in the Section 4.1.2.

In Figure 6 is an UML diagram of the code's function calls and handling of interrupts. The cycle of the interrupt is when the output is triggered and set high and the input pin registers the high from the output. The time is captured before setting the output and after registering the input which gives a certification that the time measures have included the two interrupts being handled by the processor.

[3]When going from low to high state.

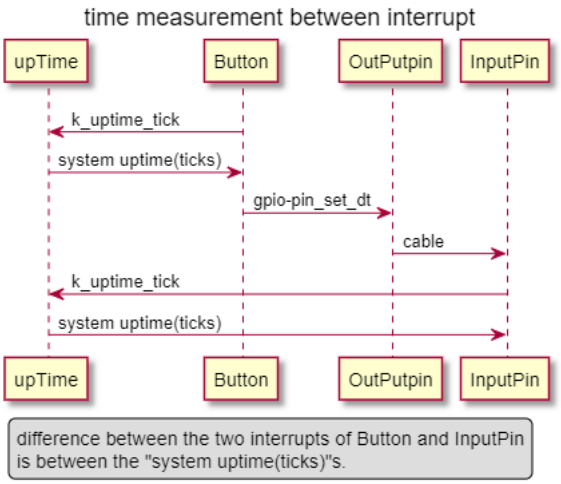


Figure 6: Function calls and interrupts

3.1.2 Second test: Measurement of BLE mesh signal transfer time

Here a mesh network was implemented using the pre-built sample "light" from Nordic Semiconducor and modifying it by adding time model (see Section 2.5.3) to it. The time model would allow for sending and receiving signal and the signals would be time messages referred to as messages. One device would read messages and send signals through an output that would be the server, whereas the other device, i.e., the client, would send messages and read on input. Measuring the time it took for a message being sent by the client and receiving an I/O signal from the server. The time was measured by saving the client's uptime in ticks (see Section 2.6.1) when sending messages and then saving the uptime input was triggered, by subtracting the sending uptime ticks from receiving ticks the amount of ticks it took for signal being sent over mesh was produced.

In Figure 7 is an UML diagram describing the time captured at the client when sending the signal, and capturing the time at which the input pin was triggered.

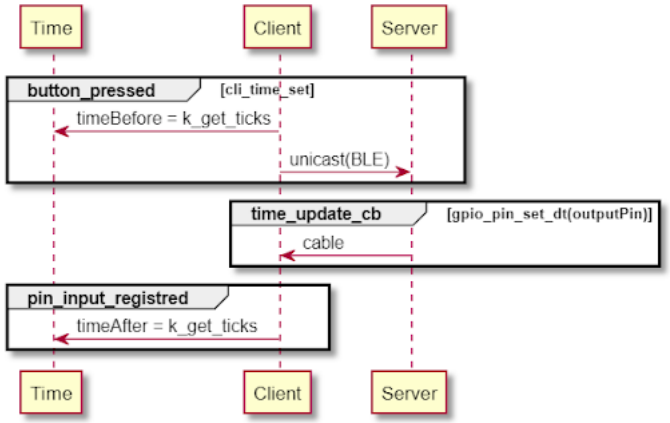


Figure 7: Time between sending the signal to when input on the GPIO was triggered

3.1.3 Final test: Test bench trial

The tests described in Section 3.1.1 and 3.1.2 were the build up for this final test. This test was set up for collecting data to have a background for statistical evidence and was designed as follows:

One device would be the client, sending a BLE mesh signal to the servers and collecting their responses over I/O. There would be five servers receiving the BLE mesh signal of the client and responding by sending a signal with I/O. The client would repeatedly unicast a signal in the form of time message signal over the Bluetooth mesh in an interval of 10 seconds. The response time from the servers for the same Bluetooth mesh signal would be captured as described in Section 3.1.2 and stored in an array The array would then be printed and saved to a csv-file with the program termite[4]. The test would be running over 72 hours to gather data. The purpose of this was to collect data for statistical background, to see if the time difference between the five nodes would be greater than one millisecond.

Multiple tests were run for 0.5 - 2 hours to see if there would occur any errors in the code or the logging program termite by having it collect data for a long time. The setup differed, trying different cards as servers and clients.

[4]Termite is a program working as an external terminal capable to save the printouts in different formats.

The test that ran for 72 hours had a nRF5340 as client and was broadcasting to a group address within the mesh which had six servers/devices connected to it. Three of the cards were nRF5340 but one of these was unsubscribed from the group to fix retransmission problems (see Section 6 bleeding edge). The other three cards connected to the client were nRF52840. The Terminal was logged by termite.

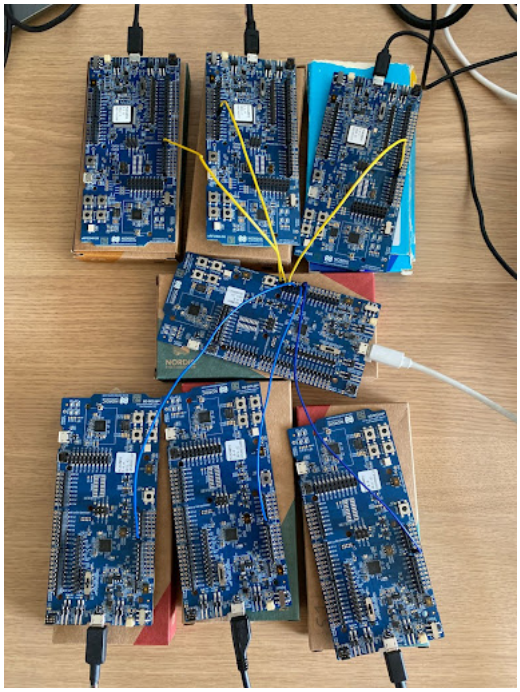


Figure 8: Shown in figure: 3 nRF52840dk boards and 3 nRF5340's all connected through pins and powered with a USB hub. The two boards appearance is very similar.

3.1.4 Synchronization through statistical evidence

The result of the test in Section 3.1.3 will be presented in the result Section 4, but what the result showed was the need for statistical decision making when synchronizing. The decision should be based on statistical background gathered from tests. This section describes the thought behind using gathered data as statistical evidence.

When collecting data from different loggers in a mesh network it is only interesting if the time for when the data was collected is comparable to another sensor in the same mesh network. To synchronize all the nodes having a sensor that is a part of the same mesh-network, without having the nodes talking with each other as with other syncronization algorithms(see 2.4.1), every node needs to have an insurance that the their reference point is synchronized with other nodes reference point. Loggers/servers receive a signal (in the form of a time message) from the client that is below one ms to be the sync pulse. The sync pulse is the pulse that contains the information that will be a reference point for that specific server/logger and the pulse are an certain amount of signals sent within a known interval.

The sync pulse will be a time message containing a **TAI-time** and the server/logger will save the **uptime** at which it received the time message. These two timestamps will be a pair and will be saved as the sync point (the TAI and uptime may differ between the servers/loggers connected within the same mesh but they will have in common that the sync point was set by a sync pulse that was within the time tolerance of one ms).

To ensure that the pulse that will be the sync pulse is within the tolerance, we have to look at the statistical evidence and decide which pulse is guaranteed to be under one millisecond. The statistical evidence says that if 1000 pulses are sent then approximately 850 will be within the tolerance of one millisecond, so the signal of the pulse which would be the sync pulse should be included by the statistical median (the data used here is roughly from the result, see Section for 4.1.3 exact data). By sending 10 pulses, statistical background is generated to be able to determine which one of the pulses arrived withing the one millisecond tolerance. This is the statistical evidence that gives the synchronization its insurance. If the pulses are set to arrive in an interval of one per second one could calculate the deviation of each pulse, by getting the time difference between two pulses and subtracting one second.

Then by sorting the deviations and choosing the median of these to be the sync pulse the “worst” cases are eliminated and a signal is chosen that is sure to be part of the statistical evidence. The reason the median is chosen and not the result closest to zero is that there are uncertainties within the card. The card is told to send a signal after one second but the calculation of one second comes with uncertainties caused by the card’s internal clock drift and execution speed

of the stack. The sync pulse will set the sync point of the server and all data collected by a server/logger will have a time stamp that is referenced to the sync point.

3.2 Including external units

For the final build there needed to be some external devices included and a 9-DoF sensor and a RTC-clock were used. These devices would be connected and implemented to the SoC (nRF53/52) and tested to see if they would handle the intended tolerance.

3.2.1 External RTC clock

The system would need precise clock values for each sample of data. To compensate for the internal clock drift, an external clock would be implemented and used. The clock would be an DS3231 and communicate with the card through the I2C port.

Driver codes for the DS3231 existed in the Zephyr's library for use. When the drivers were added to our code it turned out to be incompatible with the nRF52840dk internal clock, which led the stack to believe that the frequency on the internal clock was lower than it was configured to be. Consequently the DS3231 drivers were modified.

When the code detects a drift with the internal clock, it should notify the client by sending a message to it. It was initially planned to use the GATT protocol to send a message and therefore its architecture was studied in Section 2.5.2. This idea was eventually scrapped in favor for using a custom mesh model for messaging. The implication of using custom messages (see Section 2.5.3) includes sending messages to specific addresses with custom data within the mesh network, and also handling these messages by extracting the data from the payload custom message package. This would allow a server that has drifted to send a message at the time it drifted to the client. The client could act upon this message and restart a time synchronization within the mesh network.

3.2.2 9-DoF sensor

Zephyr uses a devicetree to hold all the description of hardware. Since the driver code for our 9-DoF sensors did not exist in the Zephyr project code it had to be implemented. A driver code for an already implemented 6-DoF sensor was used as a reference. There are two ways of implementing new driver codes for devices to the Zephyr devicetree: either do the changes "Out of Tree" or "In Tree". "In Tree"-changes are the most straightforward process, where the driver code files are simply added to Zephyr project's driver folder. Since the project is updated with new versions regularly, the "In-Tree" option becomes unreliable with time;

any external changes to the project once downloaded would be overwritten with every new version downloaded.
Additions to the devicetree in an “Out of Tree” manner is a bit more complicated. “Out of Tree” means that the driver code would be added independently of the Zephyr project’s repository, which means that the driver codes would not be overwritten with new updated versions of Zephyr.
The driver code was modified to communicate with the board through SPI and in an ”Out of Tree” manner.

4 Result

In this chapter we present the results of the tests described earlier in the report and also the resulting synchronization structure developed from results of the tests.

4.1 Result of the millisecond time sync test

This part will report the result of the tests described in Section 3.1

4.1.1 Result of first test

Presented in Appendix 7.1 is the time differences (in µs) between 100 interrupts. In 41 of the 101 outputs the time difference was 30 517 ns. Since the internal clock speed is set at 32 768 hz, the minimum time difference can be 1/32678 = 30 517 ns. This could also be converted to one tick (the smallest possible unit for the processor to count in).

4.1.2 Result of second test

Presented in Appendix 7.2 is the time differences (in µs) between 100 interrupts. Summary of the result is that the time can vary between $\sim$ 8000 μs and $\sim$ 60000 μ, showing that the BLE signal may differ with each signal sent. Results from the first test in Section 4.1.1 shows that the I/O is responsible for 0-30517 ns of the time of the signal, which is at most approximately 0.3% of the total transfer time of the signal.

4.1.3 Result of final test

In this section the results of the test in Section 3.1.3 are displayed. The result of the final test was gathered over 72 hours and can be seen in Table 2:

Table 2

Time differences between mesh hops measured by the client				
Nbr of cards	Card type	Max time difference between cards	Median time difference between cards	Percent of time difference under 1ms
2	nRF53	156555 µs	458 µs	$\sim$ 85.8%
3	nRF52	157318 µs	580 µs	$\sim$ 88.4%
5	nRF52 & nRF53	159790 µs	3143 µs	$\sim$ 0.00004%

To clarify the Table 2, where it says two of nRF53 the data being displayed is between two cards of the type nRF53. Where it says five of nRF52 & nRF53 the data being displayed is between five cards, two being nRF53 and three being

nRF52. The amount of data samples gathered during the 72 hours are 24029. During the test the amount of transmissions within the mesh was logged; in 99.8% of the sinals there were no package loss within the mesh resulting in no retransmissions. The result of the three day test is used as statistical evidence, which is then used for further implementation of the code.

4.1.4 Calculation of the statistical evidence

In this section the statistical probability calculation is presented, see Section 3.1.4 where statistical evidence is explained. The result of Section 4.1.3 says that $\sim 15\%$ of transmissions will have a signal that differ more than one millisecond, compared to the other signals within the same transmission[5]. By choosing the median out of 10 signals as a *sync pulse*, means that for the *sync pulse* to not be within tolerance, 5 or more signals have to "fail", the "failures" need to have the same polarity Z^+ or Z^- in order to affect the median negatively. When calculating the possibility of error during synchronization, the "failures" are assumed to be of the same polarity. By using the binomial theorem the following can be summarized in Table 3.

Table 3

Using Binomial Theorem to calculate the probability of five signals out of ten arriving later than one millisecond at one server	
Probability of failure on a single trial	0.15%
Number of trials	10
Number of successes (x)	5
Cumulative probability P(X > x)	0.009874091

This means that even though including a margin on failure chance and disregarding that the error needs to have same polarity to effect the median, the chance of one server choosing a sync pulse that is not within 1 ms is less than 1%.

4.2 The code resulting from tests

The goal of the base code is to ensure that servers connected in the same mesh network have a synchronization within the tolerance of 1 ms. The synchronization for this project means that each server has a "sync-point" that is related to the client (time authority). The sync-point is guaranteed to have been sent from the client and received at the server within the tolerance. This section will mention synchronization pulses; a synchronization pulse is a time message sent from the client via the pre-defined model *Time_cli*. It is sent through the mesh network. This section will show the resulting base code, implemented after the

[5]The signal is the message from the client directed at one server. The transmission is the client broadcasting to all the servers

test result shown in Section 4.1. The plant UML diagram in Figure 9 will have the function name corresponding to each arrow.

4.2.1 Synchronization pulses

Pulses of 10 (this is how it is implemented, it is possible to increase/decrease the amount of pulses) are sent from the client to each server in the network. These pulses are time messages sent in an interval of 1 second via the *Bt_mesh_model_time_cli* and the time messages is containing a time stamp in TAI. Each server saves the time stamp and the up-time at which it arrives to an paired array[6]. See plantUML in Figure 9 for visualization of how this work:

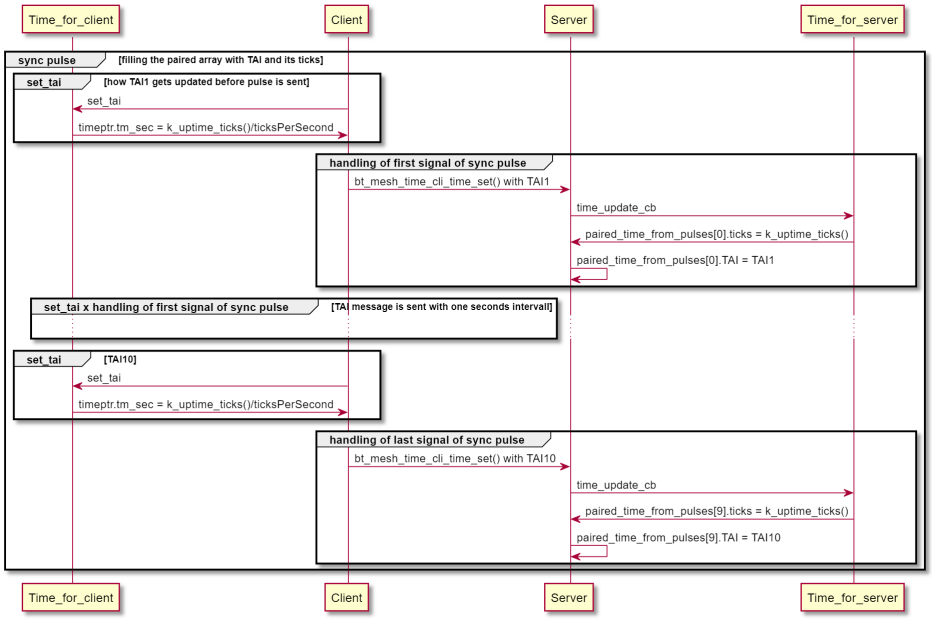


Figure 9: An UML diagram over the synchronization process

Set_tai

Set_tai is a function at the client that when called on, gathers the uptime of device in tick and converts that to seconds, then updates the TAI pointer at the client

[6] An array of pair objects. A pair object is defined in object-oriented programming languages as an object that holds two values. In this project a pair object was made to hold two integers

Handling of sync pulse

This section illustrates the client sending time messages and the server handling those messages and storing the information retrieved from them.

- *Bt_mesh_time_cli_time_set()* with TAI: time message being sent from the client over BLE mesh containing TAI value as data.
- *Time_update_cb*: callback function at the server which will be triggered to execute when a time message is received
- *Paired_time_from_pulse[x].tick*: a paired array that will be updated when the callback function is triggered. The tick part of the array stores the current uptime when time message was received and the TAI part stores the corresponding TAI-value sent by the client.

A time message is sent via *Time model* from the server containing the TAI pointer described above. The TAI value and the uptime in ticks since the system start is saved to an array pairing the two objects together. 10 sync pulses will arrive with a one second interval, after which the server count the amount of pulses to 10 a K_work [7] function to set a synchronization point is scheduled.

4.2.2 Selection of synchronization point

After 10 sync pulses have been saved to a pair array the next step is to calculate the difference between two adjacent times at which they arrived at the server, and then subtract the one second interval. The calculations are saved to the paired array *Time_diff_pulses* which is sorted based on the uptime, the median values of this array become the global sync-point for that server.

Cal_median_time

Function to start the process of calculating and selecting the server's sync point and is called upon right after the server has received the 10 pulses described in Section 4.2.1 *Paired_time_from_pulses* and *Time_diff_pulses* are paired arrays.

Collecting time differences

The array gathered from the sync pulse function explained in Section 4.2.1 is calculated in preparation for sorting.

- *Time_diff_pulses[x].tai = paired_time_from_pulses[x+1].tai*: transferring TAI value from previous array to the new array that will be calculated on.
- *Diff*: Calculating difference of 2 adjacent tick-values

[7] *K_work* is used when scheduling of events in the thread and is a functionality provided by Zephyr

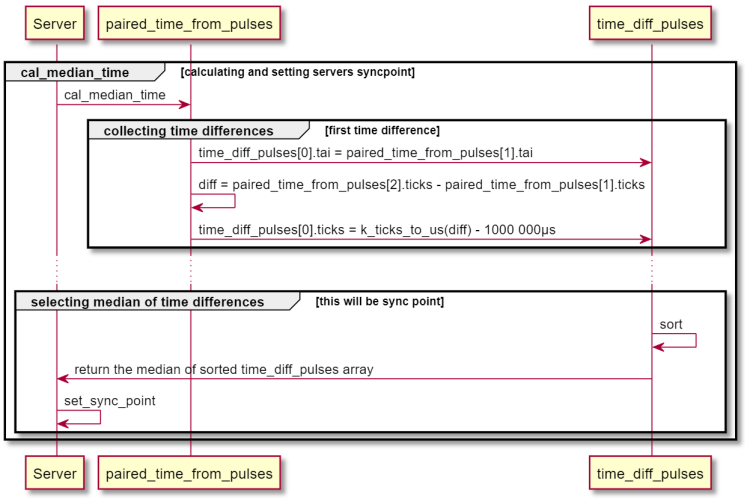


Figure 10: a figure how the synchronization pulse is processed

- *Time_diff_pulse[x].ticks=k_ticks_to_us(diff) - 1s*: taking the time diff and converting it to microseconds and subtracting 1 second, then storing the value in the array with a paired value of TAI.

The *Time_diff_pulses* array has a third paired category called *uptime*, here the raw value of the ticks are saved to make it easier when selecting the syncpoint.

Selecting median of time differences

Sorting of time diff pulses and choosing the servers global sync point.

- *Sort*: function that sorts the array *Time_diff_pulses* based on the calculated ticks.
- *Set_sync_point*: selecting the median values of sorted array *Time_diff_pulses*, TAI and uptime values will be the global sync point of the server.

The sync point will be a reference when collecting data with timestamps.

4.2.3 Compensation for internal drift of clock

Once synchronization is set at a server the validity of the syncpoint is as long as the internal clock doesn't drift further than 1 millisecond. By having an external clock of higher precision and comparing it to the internal clock it is possible to tell if drift of a magnitude greater than 1 ms has occurred and if so the mesh network would be resynchronized by the client. In the UML diagram in Figure 11 the code checking for drift is described.

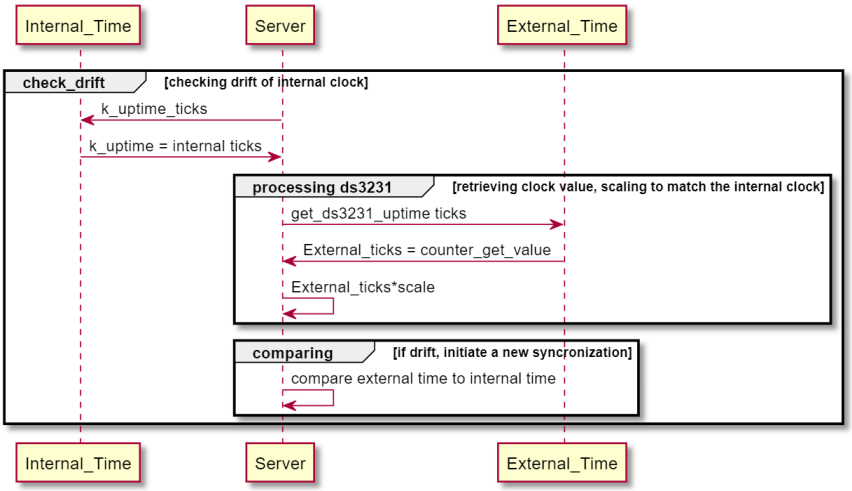


Figure 11: How the code is checking drift with the TCXO

Check drift

A function that is able to be scheduled to be performed at a chosen interval.

- *Internal_ticks = K_up_time_ticks*: store the internal clocks value for comparing

Processing the DS3231

Function to gather external clock value and storing for comparing

- *Get_DS3231_uptime_ticks*: function that communicates with external component RTC, that retrieves its clock value in ticks. Stores the amount of tick to variable *External_ticks*
- *External_ticks*scale*: since the RTC clock speed is different to the internal clock speed a scaling is performed for comparing.

Comparing

Compare the values of internal and external clock and if the difference is greater than one millisecond a message is sent via *custom model* to the client to start a new synchronization.

5 Discussion

In this section the results from Section 4 are analyzed, how the result matters and plausible reasons for how the results turned out. It will also briefly touch some use cases in which the results could play a role.

5.1 Analyzing the final test

The result from this test (see Section 4.1.3) met our expectations: 86% - 88% of the broadcast signals were received at the servers within 1 millisecond from each other on cards of the same type. This result meant that the project continued focusing on cards of the same type when establishing a mesh. The median time retrieved from the final test can be mainly attributed to the time it take for a mesh hop, and not the time it took for the interrupt handling and cable traveling, since the result given in Section 4.1.2 shows that it can be neglected.

Focusing on the result of signal transmission time for cards of same type meant that the code was in need of implementation to be repeatable, meaning that instead of the synchronization trial being accurate 85% of the time it would be a certainty that the synchronization was a success. The synchronization would be repeatable and have the same outcome each time. The implementation of this synchronization model would not be possible without the the statistical evidence from the final test.

5.1.1 The result variation of the cards

From the result of the final test (see Section 4.1.3) it became clear that a mesh network including two different card types was not possible, since only 1 out of 24029 trails were within the tolerance of 1 ms resulting in a success of $\sim$ 0.00004%. The result of this could be explained by the fact that the two card types had different chip-configuration, see Section 2.2. The different chip-configurations can affect the network packet handling, resulting in different processing speeds of message sent over the mesh. There could be a limitation on the synchronization depending on chip-configuration of the nodes in the mesh seeking synchronization, but more tests and research are needed in order to conclude the chip-configuration's implications.

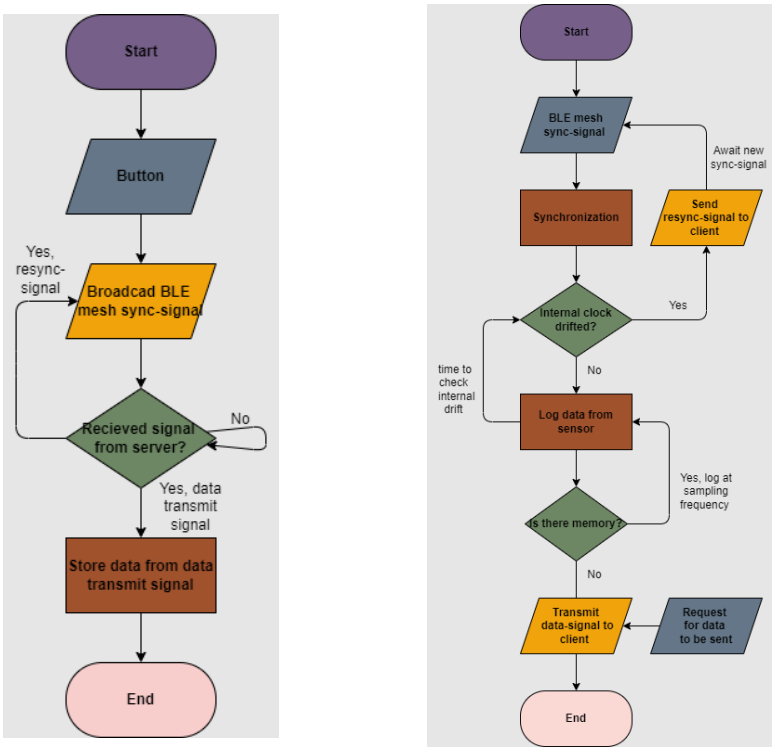
There was also a difference between the two different boards. The two nRF5340 (servers) had between themself a median time of 468 µs and the three nRF52840 (servers) had between themselves a median time of 580 µs. This difference is probably caused by the fact that the more servers in a network, the more time it took for the client's stack to handle all incoming GPIO interrupts. One possibility explaining the increased delay is that when the client sends out a broadcast message, the more server nodes there are in the mesh, the higher the risk is that one server misses the broadcast and must wait until it gets the message by the flood-principle.

5.2 Implemented code for synchronization

The result of the implemented code that is mentioned in Section 5.1 gave the synchronization within the mesh a repeatability, setting a sync point could be set within the tolerance. Meaning this project would be able to be built on further to hopefully one day be a working prototype. In section 4.1.4 calculation is done on what the percentage outcome of setting a sync-point from 10 pulses would be. It should be noted that these calculations are approximated and balanced in favor of failure. The result of these calculations is that the chance of a sync-point being set that is not within the tolerance is less than 1%. This calculation depends on the number of cards withing the mesh, the probability of one synchronization failing becomes greater the more cards there are in the mesh. This could be combated by raising the amount of pulses that are used in setting the sync point. More trials and tests would be needed to find the optimal amount of nodes and sync pulses for this code implementation.

5.2.1 Visualising the complete project

All code that has been or is planned on being implemented has followed a vision for the final project build. In this section that vision is tried to be visualised by depicting a flowchart for the goal of the final build, see flowchart in Figure12.



(a) Flowchart for client

(b) Flowchart for server

Figure 12: Flowcharts

Seen in Figure 12a is the flowchart for the client. The input of a button pressed will start an output of sync-signals being sent to servers. The client will then be in a passive state until a signal from a server will start the process of re-syncing or receiving data from a server.

Seen in Figure 12b is the flowchart of the server. The server will start the synchronization process when an input from a sync-signal is detected. Once synchronization is completed it will be in a stage where it is logging data at the sampling frequency. The drift of the internal clock will be checked regularly and if it has drifted then an output in the form of resync-signal will be sent to the client. Before storing a logged value the memory space will determine if it is possible to store another value else the collected data will be sent to the client. There is also an input, to trigger data transfer to server.

5.3 Possible use cases for the results

The results indicate it would be possible to synchronize servers within a mesh to a tolerance of 1 ms, which would allow data that is collected within this mesh to be comparable to each other knowing that the difference between two timestamps at max differ one ms. A field in which such a thing may be applicable is the field of robotics. The following is a citation from article OH, (Y.T,2019) [12] *"Geometric errors in robots, i.e., errors in joint angles and datum location error, are considered to be the main sources of position error"*. So a possible application for the research of this project is to equip sensors that are wireless to each axis of a robotic arm and collect data so the errors may be documented and possibly compensated for. These errors cause end effector inaccuracy and eventually the robot needs to be calibrated. This is often done offline, and causes downtime in production. Having a 9-DoF sensor on each axis of the robot arm and having sampling done synchronized in time can provide an ability to do the calibration in real time.

6 Conclusion

In this section a conclusion to the project is given, answering the research questions from Section 1.2.1 that were asked before the project started and looking at how this project might be developed further with future works.

The project has been run with the aim to produce and develop a base code, meaning that priority has been to produce something of value for the company rather than researching the specifics over each choice. There have been certain parts of the project that have been challenging and the problem of the challenge has been explained to Adevo. Adevo has then taken responsibility to address these problems and to solve them so that the development of the project may continue.

The project has been developed at the bleeding edge which has been noticed at times with certain bugs or uncertainties.

6.1 Answers to research questions

See Section 1.2.1 for the questions that the project set out to answer. These questions were formed by Adevo AB. Although they have not been explicitly addressed, trying to produce and develop the code for a synchronized BLE mesh logger has helped to get these questions partly answered.

6.1.1 Synchronization

What are the limits of mesh networks and how to synchronize unit clocks to assert ms resolution?

The limits of mesh network found through the 72 hours test was that messages would arrive within the tolerance ca 85% of the time. With the knowledge of the limitations the code was developed to establish synchronization through statistical measures.

6.1.2 Interface device

How to interface and develop drivers for I2C or SD sensors for Nordic's development kit boards(nRF52840 nRF5340) and Zephyr OS?

Implementing a driver makes it possible for external devices, such as the RTC and the 9-DoF, to communicate with the operating system Zephyr. Drivers are interfaced differently if they are included in Zephyr libraries or not. If a device has its driver in Zephyr's library the only need is to include the header file in the source files, enable configuration of the device in the *prj.conf* file and bind the device's ID in its *.yaml* file. If the device (in this project the 9-DoF sensor) is not included in Zephyr's libraries the device needs to be added to the

device-tree manually. Adding devices to the device-tree build can be done in- or out-of-tree. If adding the device in-tree it will be removed once Zephyr releases a new version if not performing a pull-request to Zephyr adding it permanently to Zephyr libraries. The benefit of adding out-of-tree is that the implementation is not threatened by getting deleted by any updates to Zephyr versions. Once a driver is added to the devicetree in an out-of-tree manner, the same procedure that is done for devices with drivers already included in Zephyr's library should be performed.

Adding communication between a sensor and the board is done by looking at the board's pin layout and how that layout is configured for certain communication, for example I2C or SPI. In the *dts* file an overlay is then added that specifies which communication form and pin the device will be using and that it is compatible with the driver of that device.

6.1.3 Data storage

What are the limitations of logging storage and how should data and time stamps be stored?

Data and time stamps should be stored in a paired array similar to what has been used collecting time pulses when performing synchronization. Using this type of array to store data makes it possible to connect a certain positional values with a certain point in time. When using this type of array to store data when logging, the length of the array is unknown and this will affect storage and memory. there was not enough time to investigate this and it will need to be looked at further in future works (see Section 6.2).

6.1.4 Data transfer

How to transfer data and timestamps in an efficient way to a mobile phone with full security?

When implementing the external clock a model was implemented to send custom messages. This custom model allowed for modification of the payload of messages sent and extracting the information of the message when received. The custom model also allowed being sent to any address within the same mesh network as sender, meaning that hopefully this custom model could be customized to send data to a mobile phone that has entered the same network as the sender. However, there was not enough time in this project to further develop the custom model and test if it works or the security of it, will have to be done as future work (see Section 6.2).

6.2 Future work

Some of the future work are the loose ends that were not able to be completed within the designated time frame. These have already been mentioned in Section 6.1 and in this section they will be given a brief description.

6.2.1 Implementation of external device

The RTC was included in the project's base code, and was communicating with the main card. However, the preexisting implementation of the RTC's driver was so that when enabled to communicate over I2C it would lower the master (main card) internal clock speed. Adevo will look at this and rework the implementation of the driver.

The 9-DoF sensor did not have a driver preexisting so an out-of-tree driver was implemented, but there was a pathing error that needed to be worked through with the help of Adevo and time ran out.

6.2.2 Data storage and data transfer

In Sections 6.1.3 and 6.1.4 it is mentioned what was able to be implemented within the time frame but what could be expanded upon in a future work.
Data storage would be looked at, namely how to expand and resize paired arrays so that there would be room for more data values and what the limitation of this would be. The possibilities of using *alloc* to reserve space on the stack should be further investigated.

References

[1] *About nordicsemi*. https://www.nordicsemi.com/About-us. Accessed: 2022-05-15.

[2] *Basics - Zephyr*. https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/zephyr/develop/west/basics.html. Accessed: 2022-05-15.

[3] *Bluetooth Stack Architecture*. https://docs.zephyrproject.org/latest/connectivity/bluetooth/bluetooth-arch.html. Accessed: 2022-05-15.

[4] *Bluetooth Technology Overview*. https://www.bluetooth.com/learn-about-bluetooth/tech-overview/. Accessed: 2022-05-15.

[5] *Capillary Networks - a smart way to get things connected*. https://www.ericsson.com/en/reports-and-papers/ericsson-technology-review/articles/capillary-networks--a-smart-way-to-get-things-connected. Accessed: 2022-05-15.

[6] *DS3231*. https://datasheets.maximintegrated.com/en/ds/DS3231.pdf. Accessed: 2022-05-16.

[7] *GATT*. https://www.oreilly.com/library/view/getting-started-with/9781491900550/ch04.html. Accessed: 2022-05-15.

[8] Kristoffer Hilmersson and Filip Gummesson. "Time Synchronization in Short Range Wireless Networks". In: (2016).

[9] *Introduction - Zephyr Project Documentation*. https://docs.zephyrproject.org/latest/introduction/index.html. Accessed: 2022-05-15.

[10] *nRF52840*. https://www.nordicsemi.com/Products/nRF52840. Accessed: 2022-05-15.

[11] *nRF5340*. https://www.nordicsemi.com/Products/nRF5340. Accessed: 2022-05-15.

[12] Yeon Taek Oh. "Study of Orientation Error on Robot End Effector and Volumetric Error of Articulated Robot". In: *Applied Sciences* 9.23 (2019), p. 5149.

[13] *Provision*. https://docs.zephyrproject.org/3.0.0/reference/bluetooth/mesh/provisioning.html. Accessed: 2022-05-16.

[14] *Whitepaper on Bluetooth Mesh*. https://www.ericsson.com/en/reports-and-papers/white-papers/bluetooth-mesh-networking. Accessed: 2022-05-15.

[15] Martin Woolley. "Bluetooth Mesh Models Technical Overview". In: *no. March* (2019), pp. 1–41.

7 Appendix

7.1 Terminal output

```
0: time difference between the two interrupts 0
1: time difference between the two interrupts 30517
2: time difference between the two interrupts 0
3: time difference between the two interrupts 30517
4: time difference between the two interrupts 0
5: time difference between the two interrupts 0
6: time difference between the two interrupts 0
7: time difference between the two interrupts 0
8: time difference between the two interrupts 30517
9: time difference between the two interrupts 0
10: time difference between the two interrupts 0
11: time difference between the two interrupts 0
12: time difference between the two interrupts 0
13: time difference between the two interrupts 0
14: time difference between the two interrupts 30517
15: time difference between the two interrupts 30517
16: time difference between the two interrupts 0
17: time difference between the two interrupts 0
18: time difference between the two interrupts 0
19: time difference between the two interrupts 0
20: time difference between the two interrupts 0
21: time difference between the two interrupts 0
22: time difference between the two interrupts 0
23: time difference between the two interrupts 0
24: time difference between the two interrupts 0
25: time difference between the two interrupts 30517
26: time difference between the two interrupts 30517
27: time difference between the two interrupts 0
28: time difference between the two interrupts 0
29: time difference between the two interrupts 30517
30: time difference between the two interrupts 0
31: time difference between the two interrupts 30517
32: time difference between the two interrupts 0
33: time difference between the two interrupts 0
34: time difference between the two interrupts 0
35: time difference between the two interrupts 30517
36: time difference between the two interrupts 30517
37: time difference between the two interrupts 30517
38: time difference between the two interrupts 0
39: time difference between the two interrupts 0
40: time difference between the two interrupts 30517
41: time difference between the two interrupts 30517
42: time difference between the two interrupts 30517
43: time difference between the two interrupts 0
44: time difference between the two interrupts 30517
45: time difference between the two interrupts 30517
46: time difference between the two interrupts 30517
47: time difference between the two interrupts 0
48: time difference between the two interrupts 0
49: time difference between the two interrupts 30517
50: time difference between the two interrupts 30517
51: time difference between the two interrupts 0
52: time difference between the two interrupts 0
53: time difference between the two interrupts 30517
54: time difference between the two interrupts 0
55: time difference between the two interrupts 0
56: time difference between the two interrupts 30517
57: time difference between the two interrupts 30517
58: time difference between the two interrupts 30517
59: time difference between the two interrupts 30517
60: time difference between the two interrupts 0
61: time difference between the two interrupts 0
62: time difference between the two interrupts 30517
63: time difference between the two interrupts 30517
```

Figure 13: 100 prints of time for interrupt handling

```
64: time difference between the two interrupts 0
65: time difference between the two interrupts 0
66: time difference between the two interrupts 30517
67: time difference between the two interrupts 0
68: time difference between the two interrupts 0
69: time difference between the two interrupts 30517
70: time difference between the two interrupts 0
71: time difference between the two interrupts 0
72: time difference between the two interrupts 0
73: time difference between the two interrupts 0
74: time difference between the two interrupts 0
75: time difference between the two interrupts 0
76: time difference between the two interrupts 0
77: time difference between the two interrupts 30517
78: time difference between the two interrupts 30517
79: time difference between the two interrupts 30517
80: time difference between the two interrupts 0
81: time difference between the two interrupts 0
82: time difference between the two interrupts 0
83: time difference between the two interrupts 30517
84: time difference between the two interrupts 0
85: time difference between the two interrupts 30517
86: time difference between the two interrupts 30517
87: time difference between the two interrupts 0
88: time difference between the two interrupts 30517
89: time difference between the two interrupts 30517
90: time difference between the two interrupts 0
91: time difference between the two interrupts 30517
92: time difference between the two interrupts 0
93: time difference between the two interrupts 30517
94: time difference between the two interrupts 0
95: time difference between the two interrupts 0
96: time difference between the two interrupts 0
97: time difference between the two interrupts 30517
98: time difference between the two interrupts 30517
99: time difference between the two interrupts 0
100: time difference between the two interrupts 30517
```

Figure 14: 100 prints of time for interrupt handling

7.2 Counter

Counter	µs	Counter	µs	Counter	µs	Counter	µs
1	9979	26	14832	51	11292	76	13062
2	12115	27	14679	52	13428	77	31494
3	11444	28	14526	53	36957	78	12146
4	10529	29	8636	54	14496	79	10925
5	9583	30	33539	55	9308	80	16296
6	13885	31	8484	56	13153	81	36102
7	9735	32	9277	57	12115	82	14709
8	8331	33	10040	58	13062	83	9460
9	10834	34	13519	59	8850	84	10254
10	15076	35	14923	60	15137	85	10376
11	10468	36	10742	61	10376	86	16388
12	10132	37	15717	62	13397	87	10376
13	14954	38	7843	63	13885	88	30701
14	13397	39	9796	64	13855	89	14252
15	8911	40	15259	65	34576	90	12604
16	17242	41	12817	66	35492	91	11200
17	8698	42	12390	67	11993	92	8514
18	15747	43	9125	68	7324	93	8728
19	14252	44	10986	69	58075	94	11414
20	10529	45	8179	70	59021	95	8484
21	11383	46	12207	71	9125	96	40863
22	10590	47	41290	72	15503	97	8820
23	12329	48	16693	73	9308	98	14069
24	13184	49	41016	74	13184	99	40039
25	13794	50	14648	75	14435	100	16357

Figure 15: 100 BLE mesh messages and their response time