SDR The Last Frontier

Communication Technology

Master Thesis
Group 921
June 3, 2021

AALBORG UNIVERSITY
STUDENT REPORT

Aalborg University
Department of Electronic Systems
Communication Technology
9^{th}-10^{th} semester

Department of Electronic Systems
Communication Technology
Fredrik Bajers Vej 7B
9220 Aalborg
http://es.aau.dk/

Title:
SDR The Last Frontier

Theme:
Analysis and Design of Communication Technologies

Project Period:
Autumn Semester 2020 - Spring semester 2021

Project Group:
921

Participant(s):
Andreas Casparsen
Panagiotis Antoniou
Jonas Ingerslev Christensen

Supervisor(s):
Jimmy Jessen Nielsen
Israel Leyva Mayorga
German Madueno (Keysight Labs)
Maxime Remy (Keysight Labs)

Copies:
Digital version

Page Numbers:
142

Date of Completion:
June 3, 2021

Abstract:

Software Defined Radio (SDR) is a piece of equipment that uses software as a replacement for traditional hardware components. Recently, a new type of SDR, which is a PCI-connected SDR, has been created. These have a very low latency meaning that it should be possible to implement protocols like Bluetooth and WiFi on them, which was not previously possible due to high latency.
This project focuses on investigating whether or not it is possible to use Bluetooth Low Energy (BLE) with a PCI-connected SDR to communicate while upholding the time requirements of the BLE protocol. A Proof of Concept (PoC) is created to investigate this and test this hypothesis. The PoC contains one of these new SDRs, and a lot of focus has been put on configuring it for optimal performance in terms of speed. The PoC also uses a third-party library as a foundation to implement the lower layers of BLE's protocol stack. The library is also modified to function with the specific use cases of this PoC. Critical link layer functionality was also implemented to handle different communication scenarios.
The results are that the PoC can receive and respond within the required time of the BLE protocol. The PoC offers a functioning lower-layer implementation of BLE along with a foundation on which further BLE functionalities can be implemented. The project also gives recommendations on how to further expand on this PoC.

Preface

This project is written by group 921 from Aalborg University during the 9^{th} and 10^{th} semester of Communication Technology. The theme for the semester is analysis and design of communication technologies.

Acronyms used in the report are shown under the section 'Acronyms' on the following page. The Vancouver system is used for citation, where a number is inserted within text to refer to a specific reference, listed in the section 'References' on page 126. This page contains all details about the used references.

The group would like to thank its supervisors from AAU for their great support during this project.
The group would also like to thank Keysight for their exciting project proposal, as well as their cooperation and support during this project.

Acronyms

BLE Bluetooth Low Energy 7, 8, 10, 11, 14, 24, 25, 26, 30, 31, 32, 33, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 49, 51, 52, 53, 54, 55, 56, 59, 62, 63, 64, 79, 81, 95, 96, 97, 103, 107, 111, 112, 118, 119, 120, 122, 123, 124, 125, 129

CRC Cyclic Redundancy Code 26, 27, 32, 39, 49, 50, 51, 55, 56, 57, 61, 69, 96, 97, 98, 102, 107, 119

FIR Finite Impulse Response 52, 71, 73

FM Frequency Modulation 52

FSK Frequency Shift Keying 52

GFSK Gaussian Frequency Shift Keying 30, 51, 52

GMSK Gaussian Minimum Frequency Shift Keying 51

HCI Host Controller Interface 47, 48, 54

IFS Inter Frame Space 5, 24, 37, 62, 66, 67, 68, 69, 72, 84, 96, 106, 107, 108, 109, 110, 111, 112, 114, 118, 119, 122

IoT Internet of Things 7

PDU Protocol Data Unit 26, 27, 32, 49, 50, 51

PoC Proof of Concept 39, 40, 41, 42, 45, 47, 53, 58, 59, 60, 63, 66, 70, 72, 75, 77, 78, 79, 82, 93, 95, 97, 101, 104, 105, 106, 107, 108, 109, 111, 112, 115, 116, 118, 119, 120, 122, 123, 124, 125

SDR Software-Defined Radio 7, 8, 9, 10, 11, 12, 13, 14, 16, 17, 19, 23, 24, 25, 37, 38, 39, 41, 42, 43, 44, 45, 46, 48, 51, 52, 53, 54, 55, 56, 62, 63, 64, 66, 67, 70, 71, 72, 76, 79, 81, 83, 91, 92, 95, 96, 107, 108, 111, 113, 114, 115, 116, 118, 119, 120, 122, 124, 125, 128

Contents

1 Introduction

In recent years, communication technology has been growing in both complexity and scale at an exponential rate. Networks are being deployed in all facets of the world to solve a large variety of different problems. When these networks are implemented, different scenarios require different solutions meaning that a multitude of protocols is utilized. An example of this is Internet of Things (IoT) infrastructures that are deployed in areas both industrial and private to collect information from multiple devices and automate different tasks. IoT networks can be wildly different in requirements of power, range, and communication load, meaning that some networks use protocols that focus on long-range while sacrificing speed (LoRa). In contrast, other protocols are used to ensure high-speed messages (Wi-Fi). This means that there are a lot of different networks being used for multiple purposes.

Each of these networks has been implemented using a specific protocol, limiting the IoT devices to protocol-specific hardware. The problem is that a company or home may use multiple networks that use different protocols, meaning that different hardware equipment needs to be deployed for each protocol. This leads to an interest in finding a standardized piece of equipment that can communicate with multiple protocols with minimal cost.

A component that could be used to solve the hardware issue is an Software-Defined Radio (SDR) device. An SDR is a piece of hardware that uses software to replace the hardware elements of traditional radio. The SDR only has a simple RF circuit to receive and send from and is usually installed in a PC or an embedded system. The traditional goal of an SDR is to achieve flexibility. Ideally, an SDR can be used to work with many different protocols. This is what makes it a valuable tool for developing prototypes and equipment, where it is necessary to adapt to a different protocol while utilizing the same hardware [1]. The SDR is also attractive in terms of price as it is cost-effective to use a device where the software can be updated to ensure that it functions with rapidly changing wireless standards or for developing new equipment [2].

One issue that has been plaguing the SDR is latency, as information traditionally has been passed between the SDR and PC via Ethernet or USB. This does not pose a problem for most of the used protocols, but there are some exceptions. Bluetooth Low Energy (BLE) and WIFI, for example, have stringent timing restrictions, where classic SDRs fail to comply with their requirements. A new SDR made available for this project with faster performance than its predecessors may resolve this. The greater speed may result in the last frontier that SDR have not yet reached to be explored.

1.1 Initial Problem Statement

With the current technological progression, advanced SDRs are available on the market that utilizes the PCI-e bus for connectivity, dramatically improving the overall latency of

the system. These advancements make room for new research on the SDR front. More specifically, in this project, the requirements and the challenges of implementing the low layers (Physical, Link Layer) of the BLE protocol on top of a state-of-the-art SDR will be studied, presented, and discussed.

Based on the topics covered in the introduction, this report will investigate the following initial problem statement:

Is it possible to implement wireless protocols such as BLE and Wi-Fi in new generation of SDRs?

- *How does a modern SDR function?*
- *What are some of the limitations of an SDR's operation in regard to wireless protocols?*

2 Technical analysis

To better understand the content outlined by the initial problem, the SDR must be investigated, starting with how an SDR works on primarily a conceptual level. Secondary the current state of the art is also to be examined to understand better how far the technology currently is and where more work is required.

2.1 Software Defined Radio

SDR defines a collection of hardware and software technologies where some or all of the radio's operating functions (also referred to as physical layer processing) are implemented through modifiable software or firmware operating on programmable processing technologies. These devices include field-programmable gate arrays (FPGA), digital signal processors (DSP), general-purpose processors (GPP), programmable System on Chip (SoC), or other application specific programmable processors. These technologies allow adding new wireless features and capabilities to existing radio systems without requiring new hardware. The SDR is defined as "Radio in which some or all of the physical layer functions are defined."

Traditional hardware-based radio devices limit cross-functionality and are only modifiable through physical intervention. The traditional radio results in higher production costs and minimal flexibility in supporting multiple waveform standards. By contrast, software-defined radio technology provides an efficient and comparatively inexpensive solution to this problem, allowing multimode, multi-band, and multi-functional wireless devices to be enhanced using software upgrades.

Some flexibility and resource management applications that SDR devices provide are broadcast transmissions in multi-media mobile environments [1], cooperative wireless networks diversity [2]

Currently, SDRs are commonly used via a USB connection (USRP), up to 2.5 GB/s (USB 3.2), which come with speed limitations when precise timing is required. This speed limitation is one of the main reasons why protocols like BLE and WiFi could not be implemented, fulfilling their strict timing restrictions.

2.1.1 Aspects of the SDR

As shown in figure 2.1, when using the SDR, I/Q samples must be provided, which are transformed into frequency signals at baseband, and that afterward can be up-converted unto the carrier frequency. Upon reception, this is down-converted and from baseband converted to I/Q samples that can be processed. These represent the signal to be sent in the time domain and are protocol-independent. However, specific protocols will need their own modulator for transforming bit-streams into modulated I/Q samples and a demodulation technique to transform I/Q samples into bit-streams.

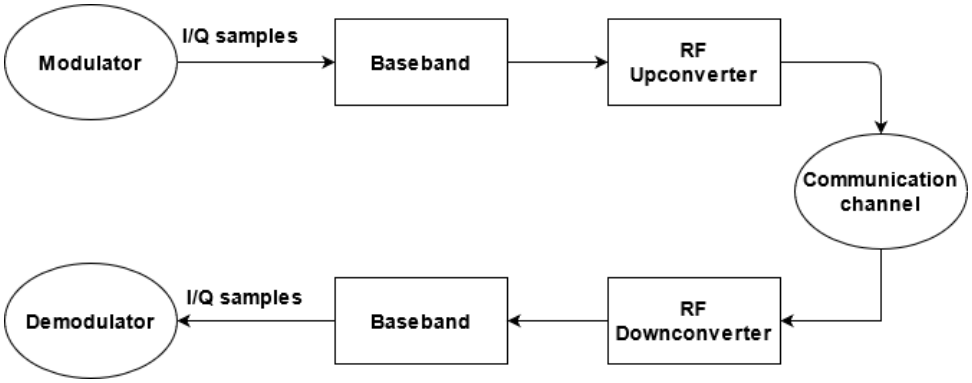

Figure 2.1: No protocol specific hardware required. Modulator and Demodulator will need to be protocol specific software.

The I/Q samples to baseband and the up-conversion and reverse operations are functionality that there exists very little control over except for choosing the carrier frequency. The general use of I/Q samples means executing the two operations, transmission, and reception requires no protocol-specific hardware. Thus both can be realized by protocol-dependent software modules *Modulator* and *Demodulator*. As a result of these modules, it also means the only thing of interest and use for this project is the code aspect of the SDR to utilize and configure it in a compelling fashion.

2.1.2 State of the art

When investigating the research field of SDR 's and supported protocols, it becomes apparent that little research has been performed regarding a full implementation of high-speed requirement protocols such as Bluetooth or WiFi. The speed in question is down to 150 μs (BLE specifically) [3] [Vol 6 Part B] and 10μs [4] for the interframe space, which is the time between a received packet and the time before the original transmitter must receive an ACK or NACK. These protocols are of interest regarding IoT and ensuring interconnection between different devices and protocols.

There is, however, an example of a group trying to combine an SDR with a Bluetooth-connected device in order to connect to a phone by Yoon et al. [5] and its continuation by Yoon et al. [6]. The purpose is to stream radio channels through the SDR, via Bluetooth, to the phone. While the purpose was not to utilize the SDR for Bluetooth itself, as a separate device was used for this, it highlights the value of Bluetooth. If Bluetooth were to be available for use on the SDR, another set of hardware could be cut out of the equation for even more convenient use.

Beyond this instance, examples of other groups researching the ability to use BLE or WiFi for actual communication purposes on an SDR platform do not seem to exist. With that being said, there still are groups that tried to utilize an SDR in combination with Bluetooth and WiFi protocols more as a utility purpose, e.g., [7] where the SDR uses WiFi and through simply listening and recording communication to fingerprint devices. Thereby it is a one-sided use of WiFi with no response. Alternatively, there are [8]

et al. and [9] et al. who focus more on the possible use cases of an SDR platform if it was possible to apply protocols like BLE. [8] focuses on a so-called AAL (Ambient assisted living)/smart home and sensors in such a network. In this case, SDR is generally attractive to utilize different devices over different protocols. In this case, Bluetooth, BLE and WiFi are all protocols of interest for this purpose. [9] focuses on the benefits of introducing an SDR platform in the health sector for better interoperability between different vendors without requiring an extra layer of software and processors. For this purpose, they are generally interested in the 2.4 GHz GMS band, meaning WiFi and Bluetooth are all of interest for this purpose.

Ultimately these sources are not the core of what is of interest to this project, but they do show that the SDR uses these protocols to a partial extent. Generally speaking, a full-on implementation and use of these protocols on an SDR platform does not seem to exist as of now. At the same time, it also shows that there is an interest and many possible appliances if it were possible.

Seeing as this new SDR that is available for this project is meant to be faster than older SDR units. It is hopefully also fast enough to explore a new previously unexplored area of high-speed requirement communication protocols that could be applied on an SDR. Thus, this project will continue with BLE as a case study for this SDR unit to investigate its capabilities. WiFi is not looked into as it requires even more from the SDR thus BLE should be possible before WiFi and therefore also investigated first.

2.1.3 SDR code investigation

Utilizing and interacting with the SDR requires to call functions where the definition of those are unknown to the user as the developer did not release the source code for those. A predefined configuration of how to operate the SDR is present from the beginning. This file is known as the "SDR example code." It allows a developer to see how functions are called and how one can interact with the SDR. Beyond this, a header file also exists, which contains more function declarations and parameters. These help getting more insight into interacting with internal values of the SDR. This section will describe the key parameters and functions in this file, how they work, and how effectively at the base configuration. It will, however, not go into deep details with everything, only those things that have been deemed valuable for this project.

2.2 Key parameters

The SDR utilizes parameters to define how it should work. Tuning these parameters is relevant for ensuring the desirable behavior of the system. The most important parameters are the following:

Tx & Rx Frequency: Allows the user to choose what frequency the SDR should be receiving or transmitting on.
Sampling rate: Allows the user to choose how fast sampling should be done. It should match the sample per symbol rate of BLE to modulate and demodulate packets correctly,

assuming no other technique is used to make up for the difference.
Buffer size: This is in part a derived value as there exist internal buffers that have a minimum size of 100μs of the sampling rate. For example, if the sampling rate is 10MHz, then the minimum buffer size is 1000 samples. It can also be set differently if a user intends to do so through the dma_buf_len variable.
Tx & Rx gain: The Rx gain defines how to configure how sensitive the device should be while listening. The Tx gain determines how loud the transmission is.
Tx_buf & Rx_buf: These are used for storing I/Q samples to be transmitted and storing read I/Q samples, respectively. These are both of the type complex floats. When loading I/Q samples into the Tx_buf, the samples must adhere to the specified requirements. These are that the samples must be in the interval of -1,1. The samples must also be 32bit little-endian for this to work as it is supposed to. The Rx_buf will, when reading samples from the SDR, also follow this standard.
dma_buf_count & dma_buf_len Multiplied these produce the length of the internal buffer of the SDR. Default setting sets the internal to be 10ms, with dma_buf_count being 100 and dma_buf_len and being μs in time, respectively. Through these, it is configurable to be longer or smaller.

2.2.1 Primary functions

Code snippet 1 shows how these two functions are defined and that they contain a lot of the same type of parameters.

Code snippet 1: The function used to transmit and receive samples.

```
msdr_read(MultiSDRState *s, int64_t *ptimestamp,
const void **samples, int count, int port_index, int timeout_ms);

msdr_write(MultiSDRState *sdr_state, int64_t *ptimestamp,
const void **samples, int count, int port_index, int *pcur_timestamp);
```

The first parameter used for both, **sdr_state**, is a struct that must be created and referenced when initially taking control over the SDR and for functions involving it. The **ptimestamp** is when the operation should start executing. It is required to be in the future. Otherwise, unpredictable behavior occurs, e.g., no transmission or only half the transmission occurs. It is also required to be in sample time.
Samples is an array containing the samples being sent or received (tx_buf vs. rx_buf). for rx_buf it has a minimum size equal to the dma_buf_len.
The **count** is the number of samples in the buffer intended to be read or transmitted.
The **port index** allows the use of additional ports if more SDRs are connected to the computer device at the same time.
The **timeout_ms** is the allowed execution time of msdr_read before a timeout occurs.
The **pcur_timestamp** is when the msdr_write finished executing in sample time.

The sample time previously described is a concept used for the SDR. It is the method used to measure how much time has progressed since the beginning of the program. The speed at which the number climbs is dependent on the sampling rate because a higher sampling rate produces more samples per second.

The two operations, msdr_read, and msdr_write are a bit slow on execution time in their initial state. The read operation takes on average 100 μs to execute. The write operation must be called sufficiently in advance, which is roughly around 380 μs in order for the transmission to occur. This time is, however, for a best-case execution, which will be further described in section 2.3.2. If the write is not ordered sufficiently in the future, the write operation does not occur. Alternative, it only happens partially, e.g., the latter half of the content of the Tx buffer is transmitted only. The time to execute msdr_write is generally fast and dependent on the number of samples.

There also exist other functions for "using" the SDR. First off, there exists msdr_open which puts a lock on the device driver such that other instances cannot access the SDR at the same time. A function linked to msdr_open is the msdr_close which removes the lock and should only be called when exiting the program. Another set of useful functions are msdr_start and msdr_stop. msdr_start is the function that allows a user to start the SDR and configure it upon having a lock on the driver. The configuration entails parameters such as sampling rate, Tx and Rx frequencies, and buffer configuration. msdr_stop will end the current configuration, which will allow a user to call msdr_start again with a new configuration, e.g., a new sampling rate.

2.2.2 Frequency shifting

As stated before, to start the SDR with new parameters, it requires msdr_start to be called again. This requirement means changing things such as frequency requires the current session of msdr_start to be closed with msdr_stop. Afterward, new parameters can be configured, and a new msdr_start, with new parameters, called. By measuring the time it takes to execute msdr_start, it is found that it takes about 100 milliseconds to execute. This lengthy time is related to the SDR calibrating for the intended frequency for transmission and reception. The slow change of frequency can be improved because the calibration can be done on channels intended to be used beforehand. However, this is not a supported feature as of now, and the frequency shifting and generally changing the configuration is a slow endeavor.

2.3 SDR Parameter Configuration

Previously in section 2.1.3 the initial state of the SDR and its configuration was described and shortly investigated. To further investigate options for improvements in the time aspect through a different configuration, an in-depth study of the SDR is performed. The first thing of interest in this regard is the DMA buffer configuration. Originally the DMA buffer length is 100mμs in samples and a DMA buffer count of 100, meaning an internal buffer of 10 microseconds. The buffer configuration is looked into, as msdr_read is a blocking function that always returns the number of samples that the DMA buffer is in length.
Besides the buffer configuration, the sampling rate is also examined, as early observations showed reception-transmission time change somewhat at different sampling rates, so the exact effect is further analyzed.

The reception-transmission time is defined as the time between receiving an outside input and transmitting a "response." It is measured on an oscilloscope as the time between the beginning of the outside device's signal and the beginning of the SDR's signal.

2.3.1 DMA buffers reconfiguration

DMA buffers are investigated by experimenting with the length of each buffer. The DMA buffer count is not described because the only thing found while investigating these is that as large as possible is the best. The smaller the DMA buffer count, the more often discontinuities occur. The performance otherwise is the same between small and large DMA buffer count. Therefore, the largest configuration of 150 buffers is used.

The first column on 1 is the DMA length in question. The second is the average execution time of 10000000 msdr_reads. The test is performed at 16MHz as a default setting with a DMA count of 150.

DMA length(μs)	Average execution timen(μs)
100	99.94
80	79.81
60	59.86
40	39.87
20	19.91
10	9.9
9	8.87
8	7.87
7	6.86
6	5.88
5	4.88
4	3.9
3	2.97
<3	UNSTABLE

Table 1: Measurements of average msdr_read execution time at different DMA buffer lengths

Table 1 confirms the previously mentioned point of reducing the DMA buffer length should bring the time between samples are collected down. Thus, bringing the sampling closer to continuous time. It is, however, limited to a buffer length of 3 μs as smaller sizes cause crashes. This buffer size produces an attractive result as samples are received very close to symbol speed of BLE, 1 μs. The fact that samples are received often should lead to a minimal amount of "wasted" time while waiting to sample a bit instead of the original 100 μs.

2.3.2 Change for msdr_write

A subsequent result of reducing the size of the DMA buffers is the change of behavior observed on the receive-transmit time. The expectancy was that the main result of this change of configuration would be to make the receive-transmit time more consistent, but it also brought it down significantly.

In order to better understand how much the receive-transmit time is affected, a similar test is run. This test uses the same 16 MHz sampling rate and the same DMA buffer lengths tested previously. For this test, ten "samples" of the reception-transmission time are observed on the oscilloscope. The average will be shown in the second column, while the range of reception-transmission time observed is in the third column.

dma len(μs)	Reception-Transmission time	Observed range
100	451	$380 \leq x \leq 520$
80	400	$350 \leq x \leq 460$
60	365	$330 \leq x \leq 410$
40	313	$280 \leq x \leq 350$
20	271	$250 \leq x \leq 300$
10	248	$220 \leq x \leq 300$
9	255	$230 \leq x \leq 300$
8	239	$200 \leq x \leq 280$
7	229	$190 \leq x \leq 270$
6	240	$200 \leq x \leq 290$
5	228	$200 \leq x \leq 250$
4	218	$180 \leq x \leq 250$
3	172	$170 \leq x \leq 190$
<3	UNSTABLE	UNSTABLE

Table 2: Performance of receive-transmit at different DMA buffer lengths.

On table 2 it is easily noticeable that as the length of the DMA buffers decreases, the observed time difference on the oscilloscope is also on a decreasing trend. This trend applies both to the average as well as the range of observed time differences. The best performance for the average, and the one with minimum range, is the DMA buffer size of 3.

The average reception-transmission time looks good for this configuration, and the range has been significantly reduced compared to prior buffer lengths.
At the smaller DMA buffers, the difference is not very large. It acts almost contradictory, e.g., DMA buffer length of 8 being as fast as DMA buffer length of 6, and variations being sporadic. However, this contradictory behavior is not looked further into, as the primary purpose is to investigate the trend and performance at different configurations, which is achieved.

This test means that moving forward using a DMA buffer length of $3\mu s$ is done as it provides the best performance for both msdr_read execution time. Meanwhile, it also provides a much-needed performance enhancement for msdr_write at the same time. Furthermore, it also appears to be the most stable configuration with slight variation in the range of receive-transmit times.

2.3.3 Effect of small internal buffer

The result of having a small internal buffer is that the number of samples that can be transmitted with msdr_write grows equally tiny. With the current configuration, the size is $450\mu s$ worth of samples that can be called at once. The small number of samples means any msdr_write called has a limitation of $450\mu s$ worth of samples.
If one were to have a transmission of a size greater than this, it would severely limit the capabilities of the SDR to transmit it. The requirement to order the msdr_write into the future also affects the present. This behavior is due caused by the internal buffer being a ring buffer.

A test will be performed to illustrate the behavior of msdr_write viewing future and present as the same. First, the perspective of an msdr_write call being made with 100 μs of samples called for a specific point in time.
Afterward, a secondary perspective of an msdr_write being called right after the first one. When the second msdr_write is called, it will empty 50 μs of the internal buffer 450us after the first msdr_write was meant to transmit. The result of this is, however, that the samples from the first msdr_write are overwritten.

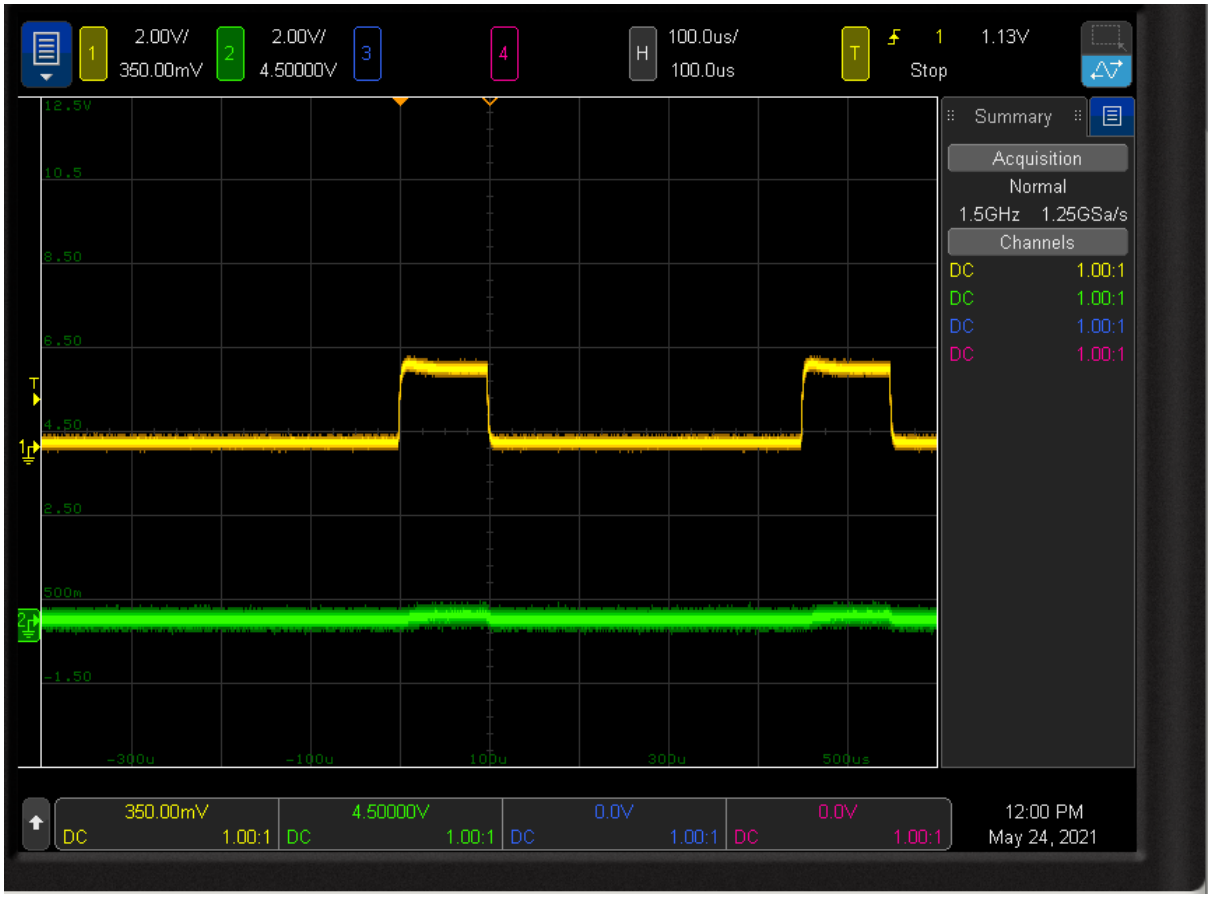

Figure 2.2: 100 μs of samples can be seen as the "high" period is 100 μs long. After an internal buffer period the signal reappears.

```
Writing 100µs of samples for time 20
Configured write TX for          20(index 20)     at 0(index 0)
```

Figure 2.3: The perspective from the code side, where msdr_write is called for a later point in time that what it is at.

In figure 2.2 it can be seen that the yellow signal, the signal from SDR is high for 100 μs and that 450 μs later, a new yellow signal appears. This observation is caused by the internal buffer running around in a ring. On figure 2.3 a different perspective from the code is viewed.

From the code perspective, the msdr_write is called at time 0 for time 20. The interpretation of this is that 20 μs from the moment the call is made, the transmission should start. The meaning of the index value is to illustrate the ring buffer that msdr_write uses. It shows the associated index of the ring buffer that samples start being loaded into. This behavior is shown in μs as opposed to samples for a more intuitive view.

The performance of the second test, where the second write operation that empties the internal buffer is called right after the first one, is also performed, which changes the result of the oscilloscope.

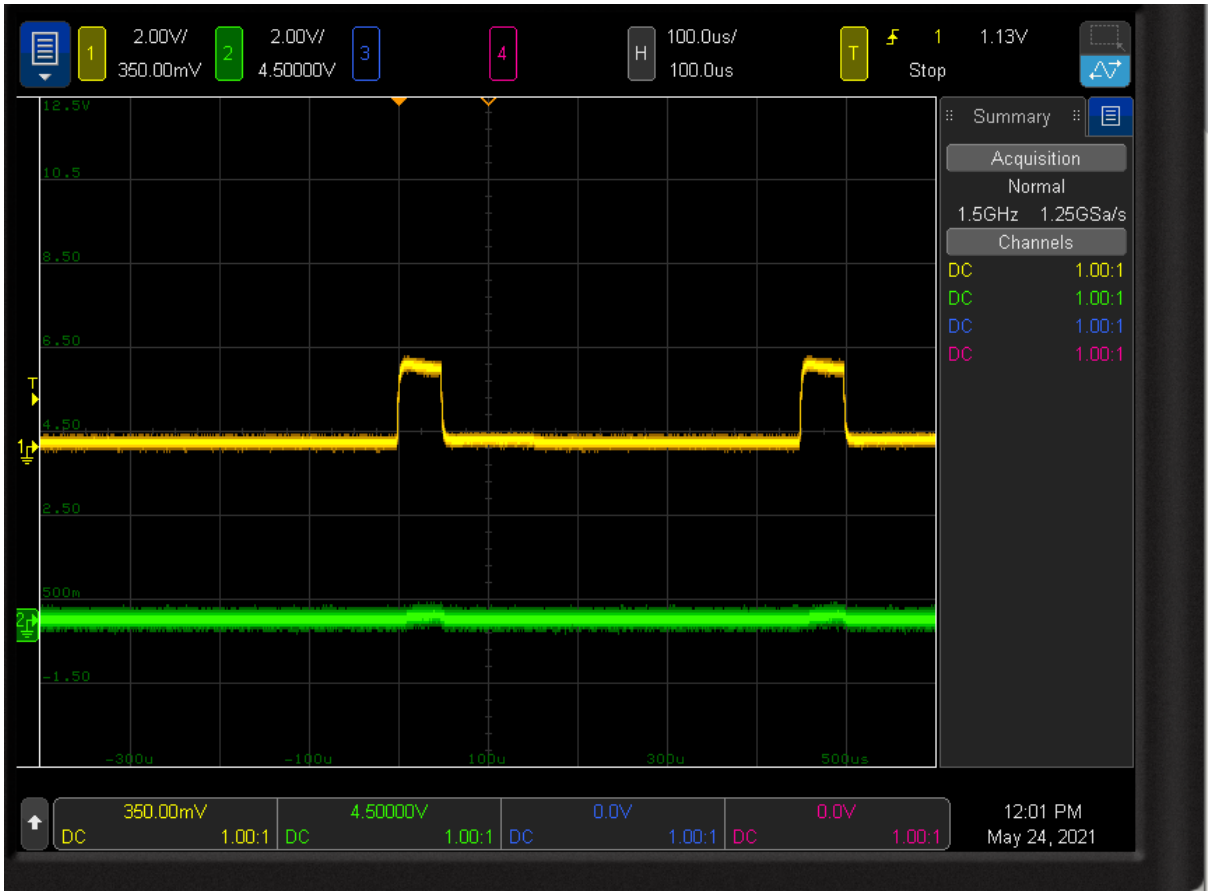

Figure 2.4: The secondary write was called. This results in only 50 μs high period as opposed to the 100 μs from before.

```
Writing 100µs of samples for time 20
Configured write TX for         20(index 20)    at 0(index 0)

emptying 50µs of samples for time 520
Configured write TX for         520(index 70)   at 7(index 7)
```

Figure 2.5: The code side perspective again. This time 2 write operations are called with with the intended start 500 μs. apart i.e. for time 20 and for time 520.

As can be seen in figure 2.4 suddenly, only a 50 μs period is visible on the oscilloscope. Investigating what happened on the code side of this in figure ref2.5 it can be seen that the emptying write call happened 7 μs later, but the call was made for time 520, which is a ring buffer translates to index 70, as 520 mod 450 = 70.
The result of this is that even though the call was supposed to affect the second run of the internal buffer, it changed the current content of the internal buffer by overwriting part of it. Furthermore, if one were to call an msdr_write emptying the 100 μs at any point where the resulting internal buffer index was 20, none of the original samples would be present in the internal buffer.

This observation of the future and the present being connected ultimately means that

the write operation is limited to take only content worth 450 μs for transmission in this setting. It further means that a larger transmission of 450 μs cannot easily be split up into additional write operations, as the second write operation may overwrite the first one.

2.3.4 Sampling rate

While configuring the SDR for test runs and experimenting with different setups, it was noticed that the receive-transmit time appeared to vary between different sampling rates. Due to observing this odd behavior, an investigation into the sampling rate is performed. The reasoning for the test is to look for a pattern to understand this functionality. The execution of the test to investigate this is by using the previously found 3 μs buffer and varying sample rate. Furthermore, the reception-transmission time is recorded the same way it was for the DMA buffer length test. The test will involve ten "samples" of the reception-transmission time for each sampling rate checked.
The first column on table 3 shows the sampling rate, and the second shows the average reception-transmission time. The range of observed reception-transmission times is not provided because it is the same with an almost invisible difference of less than 5 μs.

Sampling rate MHz	Reception-Transmission time
4	UNSTABLE
8	260
16	170
32	130
40	105
>44	UNSTABLE

Table 3: Receive-transmit time being dependent on sampling rate.

Table 3 shows the investigated sampling rates. Starting at half the default sampling rate of 8 MHz, going further down may cause crashes at this DMA buffer configuration. It continues up until 40MHz. Investigation of higher values is not performed because the device started crashing frequently. Nonetheless, from the table, it can be seen that the higher the sampling rate, the lower the reception-transmission time. As a side note, this time seemingly continues to decrease as a singular test showed that at 50MHz, the receive-transmit time was as low as 90 μs.
Being faster would be preferable, but stability is also essential, which means the chosen sampling rate is 40MHz. To show how the system now operates from the perspective of the oscilloscope in figure 2.6 is provided. Here it can be seen that 105 μs passed between the activation from the generator and the transmission from the SDR.

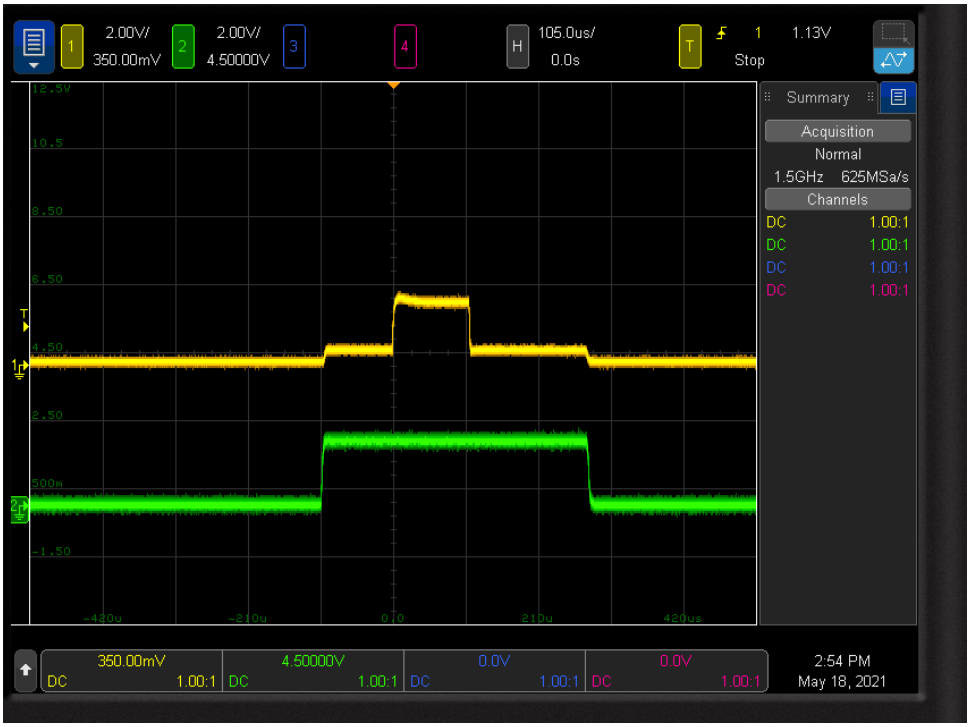

Figure 2.6: receive-transmit delay viewed on an oscilloscope. Difference between first green high, and actual first yellow high is the delay. The small bump on the yellow is caused by leakage.

2.3.5 msdr_read execution investigation

A later investigation on the exact effect that changing the DMA buffers has on executing msdr_read is also performed. This is because it is observed that the msdr_read time differed more than expected at different configurations. Furthermore, part of the performance appears reliant on both the sampling rate and DMA buffer length. This dependency is described thoroughly in this section as to how these are correlated.

The test to highlight this is executed by running the msdr_read function 10 million times and then exiting the program. Instead of the execution time being averaged, the distribution of execution times of each msdr_read is plotted.

This is done as a comparison between 16 MHz (Original setting) and 40 MHz. To allow for better readability across the entire graph, it is plotted as the logarithm to the amount of occurrences. A view of different section in a non-log graph can be found in section 11.3. Figure 11.1 and 11.2 11.3 show the distribution of no sleep.

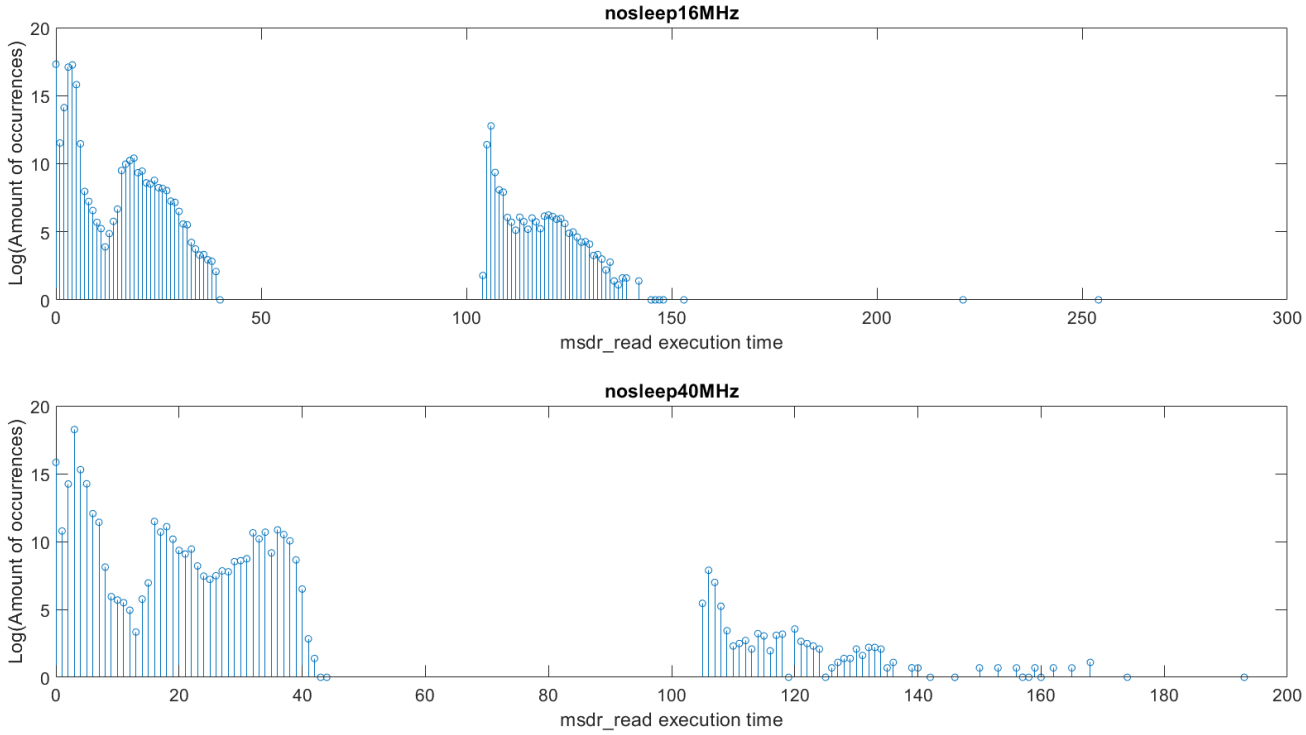

Figure 2.7: Histogram of how 3 μs buffer is distributed. Logarithm of the total occurrences for each index, for a better view.

The result on figure 2.7 is unexpected as the blocking time appears to be somewhat dependent on the sampling rate, where read operations at over 100 μs may occur. It also appears much more frequently at a lower sampling rate as opposed to 40MHz

A behavior found while investigating the usage of msdr_read is that the antenna always appears to be sampling. When calling msdr_read, it is the oldest samples in the internal buffer that are returned. This means that if not enough samples are present in the internal buffer, then msdr_read will block for the period to get enough samples. Following that, if another program execution occurs between consecutive msdr_read operations, then more samples are collected by the antenna. The collection of more samples should, in principle, mean that the msdr_read blocking time is reduced by increasing the time between msdr_read is called.

This line of thought led to introducing a short sleep period to examine if this would impact the msdr_read execution time. The reasoning is that some of the time that it otherwise would spend blocking would be spent sleeping, which would change how the execution time is distributed. The result of this test is plotted with the logarithm to the amount of occurrences for each execution time. A full view can be found in section 11.3. Figure 11.4 and 11.5 showing the distribution of usleep, and figure 11.6 and 11.7 showing the distribution of nanosleep.

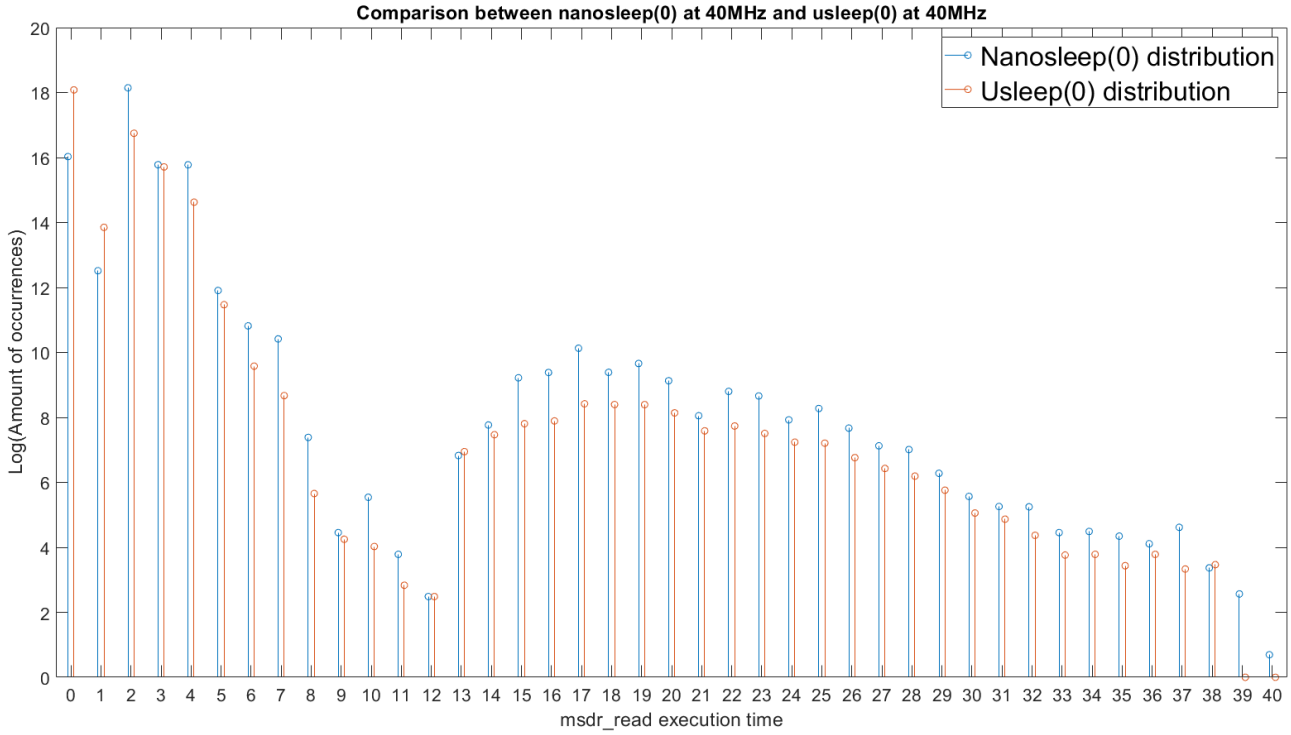

Figure 2.8: Histogram of how usleep and nanosleep are distributed compared to each other. Logarithm of the total occurrences for each index, for a better view.

As seen in figure 2.8 2 different sleeping periods were used, but they produced very different distributions as a result. Both are without any of the huge msdr_read execution of more than 100 μs. With that being said, the frequency of 20-30 μs execution is far higher for the nanosleep. This result would suggest that usleep would be the superior result.

Issues did, however, start to occur with the accuracy when using the sleep solution, as it became more inaccurate than intended. This inaccuracy will be described shortly in the following section. The general point is that usleep caused variance in the receive-transmit time, and configuring it for an exact time became impossible.

This behavior is not observed for the nanosecond solution where it always hit the 150 μs mark as ordered, on runs where it worked. However, this is the limitation of the nanosleep solution that requires additional attempts at starting the code to get a good run where msdr_read execution time does not go above 100 μs. In contrast, the usleep solution always did this but was inconsistent at always writing to a specific point in time.

2.3.6 Msdr_Write investigation

As a result of the behavior seen for msdr_read where sleep periods had unexpected effects, the effects for msdr_write are also investigated.

The behavior is first tested by varying the number of samples transmitted with each msdr_write. One thousand executions of msdr_write will be performed, and their used CPU-time averaged. The minimum and maximum values will also be shown of these. The buffer size varies from 0% of the internal buffer to the entire internal buffer written each time with 20% increments. The first column shows the sizes in percent where the 100% buffer is 450 μs worth of samples, and the others are scaled accordingly. The second column shows the mean execution time, and the third column shows the observed range of execution time. The measurement is all in nanoseconds.

Size in percent	Mean(Nanosecond)	Range (Nanosecond)
100 %	471	$35 \leq x \leq 16576$
80 %	385	$35 \leq x \leq 22057$
60 %	293	$35 \leq x \leq 9723$
40 %	205	$35 \leq x \leq 6755$
20 %	117	$35 \leq x \leq 3546$
0 %	38	$35 \leq x \leq 295$

Table 4: MSDR_write test for CPU execution time.

Generally observed on table 4 from this test is that the larger the number of samples to be put into the internal buffer, the longer it takes to process. The lowest execution spotted is the same for all, while the maximum one increases heavily upwards as the number of samples to be written increases. The msdr_write with no samples is especially fast. While it carries no meaning when intending to send samples, it is meaningful if planning to track how time progresses for the SDR.

To examine the effect of using a sleep period, this is also shortly tested where the test to be performed will be different. The difference is that instead of checking the execution time on CPU level, it will instead be checked as the time progressed in samples between each write converted to time for easier reading. Using the sample time will leave a lower resolution as at 40 MHz sampling rate, it means 1 sample is equal to 25 nanoseconds.
Thus, this test will be checked as the progress made in sample time between one write, a sleep period, and another write. The purpose of this is done to show an observed problem occurring when msdr_write is combined with a sleep period. This problem is only investigated for the buffer configuration of 20% of the full internal buffer.

Sleep time	Mean(Nanosecond)	Range(Nanosecond)
$10\mu s$	11375	$11000 \leq x \leq 35000$
$5\mu s$	6575	$6000 \leq x \leq 35000$
$2\mu s$	3100	$2000 \leq x \leq 11000$
$1\mu s$	2375	$1000 \leq x \leq 34000$
$0\mu s$	2250	$2000 \leq x \leq 38000$
No sleep	100	$0 \leq x \leq 7000$

Table 5: msdr_write test for effect of sleeping periods.

From the results shown in table 5 it is found that the returned timestamp when sleeping becomes very odd and hard to predict. This unpredictability means using usleep for msdr_write has no apparent positive effects as opposed to msdr_read. Note that this is only when performing asleep in between each operation of either msdr_read or msdr_write.
Generally, it is also seen that the execution time can be very explosive with how long time it takes to perform the msdr_write. However, this can generally be minimized by reducing the size of samples written to the internal buffer and not calling usleep.

2.4 SDR investigation conclusion

After investigating and the performance of different parameters associated with the SDR, and configuring them, it can be seen that the performance of the SDR has been improved.

The DMA buffers may be reduced significantly down to 3 μs. This leads to a much better performance when trying to optimize receive-transmit time. If the DMA buffer length is decreased further the performance of receive-transmit is increased but causes inconsistencies in the system that leads to the SDR crashing.
DMA buffer count has little effect except for making the internal buffer larger. It is however of interest to minimize the number of discontinuities that may occur, especially with large read operations.
The result of a small internal buffer poses some limitations in regard to how many samples can be transmitted at once, and poses some challenges in regards to ensuring that samples will not be overwritten if a alter msdr_wrie is called.
Furthermore for msdr_write the execution time is reliant on the the amount of samples to be written, as one might expect. The positive from this is that the smaller the buffer the more reliant the execution time is also of the msdr_write. Additionally it is found that introducing usleep in combination with msdr_write only leads to negative results.

Increasing the Sampling rate has a positive effect on the receive-transmit time. If increased further, the effects are assumed to grow stronger, but at the cost of an inconsistent system
The msdr_read execution time is highly variable and is dependent on both sampling rate and DMA buffer length. The variance seen may be reduced by including a sleep functionality, where nanosleep performs the best.

This investigation therefore concludes that the SDR has capabilities to receive and transmit within the IFS internal of the BLE protocol. That is if the parameters of the SDR are tuned and configured to a point where it gets close to crashing, but just barely remains operational, as further increases may lead to a crash. A few challenges have also become apparent in the form of msdr_read not being consistently the same, as well as the msdr_write limitation.

2.5 Bluetooth Low Energy

This section is made to present the main functionality of BLE and its lower layers.

2.5.1 General Description

This section will give a general description of BLE and some of the features of the protocol to give an example of a communication protocol not yet supported by SDRs. It will utilize [3] as its source and will thus describe features as they are defined in Bluetooth version 4.2. This description may not include some of the newer functionalities of BLE since the 4.2 version was released in late 2014, and the newest version is 5.2, which came out at the very end of 2019.

BLE was introduced in Bluetooth 4.0 Core Specification. Its goal was to design a radio standard with low power consumption optimized for Internet-of-Things Applications where almost any modern mobile platform can connect to. It is optimized for small and discrete data transfers triggered by local events, for example, periodic sensor updates. It is built using the Bluetooth stack, with the difference that some of the classic Bluetooth layers have been redesigned to match the low power requirements.

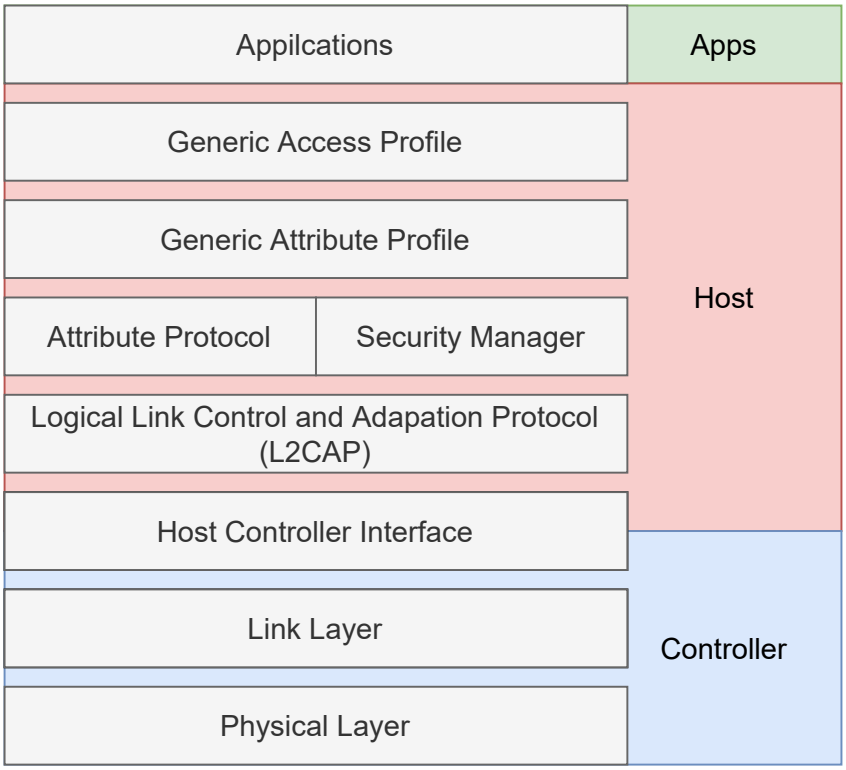

Figure 2.9: BLE Architecture

- Controller: Combines the Physical and the Link Layer. It is responsible for radio transmissions, modulation/demodulation, and forwarding the bitstream to higher layers.

- Host-Controller Interface (HCI): A linking layer that connects the Host and the Controller. Provides a protocol with standard methods for accessing the Bluetooth baseband capabilities. (HCI protocol defined in BT specification)
- Host: Combines all the layers responsible for accessing data, device discovery, and security features.
- Application: The application contains the user interface and is responsible for handling the use-case it implements.

More information about the roles will be given as the layers of the protocol stack are presented.

The protocol operates in the ISM band (2400 - 2483.5 MHz). It utilizes 40 channels in this spectrum, each with a 2MHz bandwidth. Three of these 40 channels are explicitly used for advertising events, and these are (2402, 2426, 2480). The other channels are used for data transmissions. As WiFi also operate within this spectrum interference may occur between the two protocols.

In order to give a detailed description of the functionalities and operations of BLE, its packet structure needs to be presented first.

2.5.2 Packet Structure

BLE packets contain four mandatory fields: Preamble, access-address, Protocol Data Unit (PDU) and a Cyclic Redundancy Code (CRC). Based on the type of the packet (advertisement packet or data packet), the PDU struct changes. In the case of advertisement packets, the PDU contains only the payload, which is from 0 to 37 bytes, and in the case of a data packet, it contains the payload which is from 0 to 255 bytes plus a MIC of 4 bytes, if encryption is used. Also, the Header of the PDU field changes, depending on being an Advertisement packet or a Data packet. These fields and how they are placed can be seen in figure 2.10.

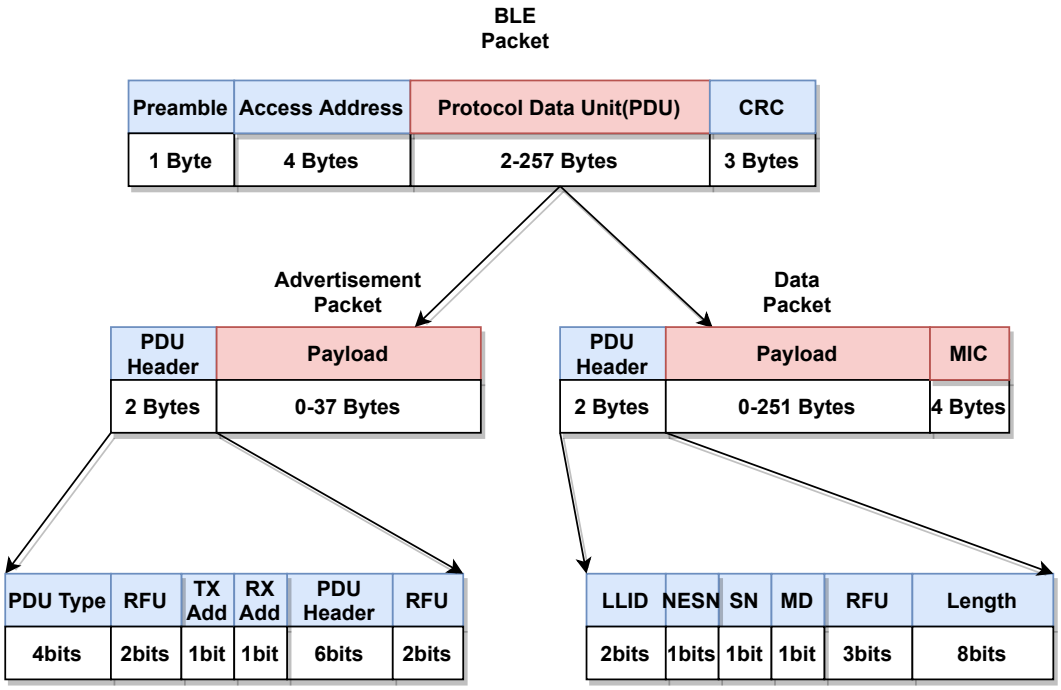

Figure 2.10: BLE Packet Structure, [3] [Vol 6, Part B]

- Preamble: Used for transmission synchronization between transmitter and receiver. It is one predefined byte. Advertisement packets use "10101010" as a preamble value. Data packets use "10101010" if LSB of access address is 0, "01010101" otherwise
- Access Address: For Data packets, it consists of 32 bits generated by the BLE device. For advertisement packets is has a predefined value of "0x8E89BED6" in hex.
- PDU: Contains the PDU header, which is a 2-Byte field and the payload of the packet. In the case of Data packets, a 4-Byte MIC is also included, depending on encryption. The PDU Header consists of 2 bytes and contains information about the type of the PDU (Advertising, scanning, etc.), sequence numbers, payload length, and more.
- CRC: Consists of 24 bits, and it is used for error detection.

Advertisement PDU Header

- PDU Type (4bits): Defines different types of PDUs that be transmitted

0000	ADV_IND	A connectable undirected advertisement PDU. Anyone can attempt to connect to this. Data contained inside may be read by scanners.
0001	ADV_DIRECT_IND	A connectable directed advertisement PDU. Only a specific device address may respond.
0010	ADV_NONCONN_IND	Non-connectable undirected advertisement packet. Data contained inside may be read by scanners.
0011	SCAN_REQ	A scanner request for data. Contains no data itself only the scanner's device address.
0100	SCAN_RSP	The response to a scanner's request. Data is contained inside.
0101	CONNECT_REQ	This is used as a response to ADV_IND and ADV_DIRECT_IND. A connection is made when an initiator sends this.
0110	ADV_SCAN_IND	Used in scan-able undirected advertisement packets.
0111-1111	RFU	Reversed for future use.

- RFU (2bits): Reserved for Future Use
- TXAdd (1bit): Contains information specific to the type of PDU
- RXAdd (1bit): Contains information specific to the type of PDU
- Length (6bits): Denotes the length of the payload on octets
- RFU (2bits): Reserved for Future Use

Data PDU Header

- LLID (2bits): Indicates whether the packet is an LL data PDU or LL control PDU

00	RFU
01	LL Data PDU, continuation fragment
10	LL DATA PDU, start of L2CAP message
11	LL CONTROL PDU

- NESN (1bit): Next Expected Sequence Number. Used for the ACK/NACK generation
- SN (1bit): Sequence Number. Used to keep track of transmitted packets. The receiver uses the SN to define NESN

- MD (1bit): More Data. Indicates that the device has more data to send
- RFU (3bits): Reserved for Future Use
- Length (8bits): Denotes the length of the payload on octets

LL Data PDU An LL Data PDU is a data channel PDU that is used to send L2CAP data. The LLID field shall be set to either 01 or 10. An LL Data PDU with the LLID field in the Header set to 01b, and the Length field set to 00000000b, is known as an Empty PDU. The Master's Link Layer may send an Empty PDU to the Slave to allow the Slave to respond with any Data Channel PDU, including an Empty PDU.

LL Control PDU An LL Control PDU is a Data Channel PDU that is used to control the Link Layer connection. The payload consists of OpCode and CtrlData fields. The Opcode identifies different types of LL Control PDU. The CtrlData field is defined by each Opcode.

2.5.3 Layers

In this project, the main focus will be on the Physical and Link Layer. An in-depth description of said layers will be presented in this section, and a trivial description of the higher layers. On figure 2.9 the layer architecture is presented

Physical Layer

The PHY layer handles the analog communications and is responsible for translating a waveform into a bitstream and the reverse. It is the lowest layer of the protocol stack and provides its services to the link layer. As mentioned, BLE operates in the unlicensed 2.4GHz ISM band and utilizes frequency hopping to mitigate interference. Furthermore, it employs two multiple access schemes: Frequency division multiple access (FDMA) and time division multiple access (TDMA). Forty (40) physical channels, separated by 2 MHz, are used in the FDMA scheme. Three (3) are used as advertising channels, and 37 are used as data channels. A TDMA based polling scheme is used in which one device transmits a packet at a predetermined time, and a corresponding device responds with a packet after a predetermined interval. Advertisement channels are used for device discovery, connection establishment, and broadcast transmissions. In contrast, data channels are used for bidirectional communication between devices and frequency hopping schemes. BLE's physical channel is sub-divided into time units known as events. Data is transmitted between devices in packets that are positioned in these events.

There are two types of events: Advertising and Connection events. Based on these events, BLE devices may take up different roles in a communication scheme:

- Advertisers: Devices that transmit advertising packets to offer a service

- Scanners: Devices that receive advertising packets on the advertising channels without the intention to connect to them
- Initiators: Devices that need to form a connection to another device and listen for connectable advertising packets

BLE PHY Layer is using the Gaussian Frequency Shift Keying (GFSK) modulation to translate data to radio transmissions. The modulation scheme generates one symbol from one bit, which contains a distinct piece of information. Symbols are assigned to frequencies, and a Fourier Transform is applied to those frequencies to move them to the time plane, resulting in a sinusoidal waveform. Said waveform is being forwarded to the physical antenna and is being transmitted over the air. The reverse process takes place on the receiver side to demodulate a BLE packet.

Link Layer

The operation of the Link Layer can be described in terms of a state machine with the following five states:

- Standby State
- Advertising State
- Scanning State
- Initiating State
- Connection State

The Link Layer state machine allows only one state to be active at a time. The Link Layer defines an Advertising State, a Scanning State, an Advertising State, a Standby State, and a Connection State. The Link Layer in the Standby State does not transmit or receive any packets. The Link Layer may have multiple instances of the Link Layer state machine. Each instance of the Link Layer state machine should support at least an Advertising State or Scanning State. The default state is the Standby State, which can be entered from any other state. The Link Layer in the Advertising State will transmit advertisement packets. It can also listen to responses triggered by these advertisement packets. A device in the Advertising State is known as an *Advertiser*. The Advertising State can be entered from the Standby State.
The Link Layer in the Scanning State will be listening for advertising channel packets from devices that are advertising.
A device in the Scanning State is known as a *scanner*. It will be listening for advertising channel, and possibly respond to packets from a specific device(s). The possible responses will will be a request for more information. The Scanning State can be entered from the Standby State. The Link Layer in the Initiating State .
A device in the Initiating State is known as an *Initiator*. The Initiating State can be

entered from the Standby State. The Connection State can be entered either from the Initiating State upon sending a connection request packet or from the Advertising state upon receiving a connection request. A device in the Connection State is known as being in a connection. How these states can transition illustrated in figure 2.11.

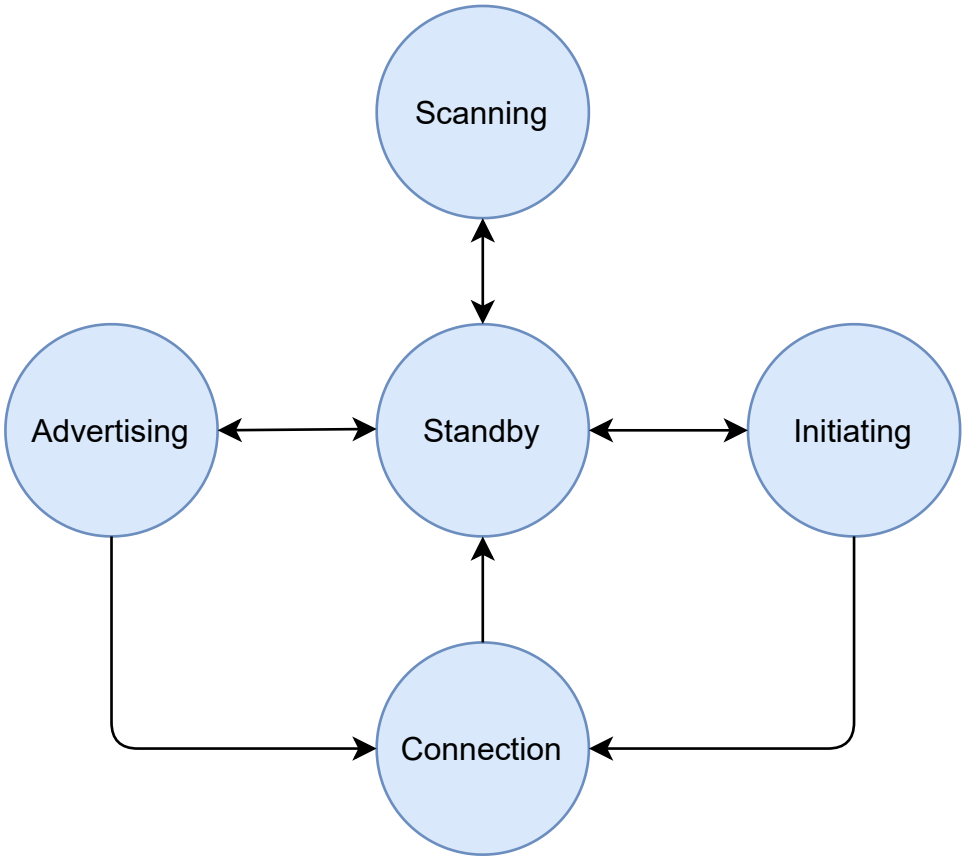

Figure 2.11: Link Layer State Machine

When two devices are in a connection, they act in different roles. A Link-Layer in the Master Role is called a *Master*. A Link-Layer in the Slave Role is called a *Slave*. The Master controls the timing of a connection event. A connection event is a point of synchronization between the Master and the Slave. There can be only one connection between two BLE device addresses. An initiator can not send a connection request to an advertiser it is already connected to. If an advertiser receives a connection request from an initiator it is already connected to, it must ignore that request. While in the Connection State, the Link Layer transmits only Data Channel PDUs in connection events. The timing of connection events is determined by two parameters: the connection event interval and the slave latency. The connection event interval is specified in the Initiator's CONNECT_REQ PDU packet and is a multiple of 1.25ms in the range of 7.5ms and 4.0s. The start of connection events are spaced regularly with an interval of connInterval and shall not overlap. The slave latency parameter defines the number of consecutive connection events that the Slave is not required to listen to the Master. The connection event is considered open while both devices continue to send packets. The MD bit of the Header of the Data Channel PDU indicates that the device has more data to

send. If neither of the devices has their MD bit in their packets set, the last packet from the Slave closes the connection event. If the Slave does not receive a packet sent from the Master, the Master will close the connection event. Likewise, if the Master does not receive a packet sent from the Slave, the Slave will close the connection event. Finally, two consecutive packets received with an invalid CRC check will close the connection event.

Apart from the State Machine functionalities, Link Layer is responsible for CRC check, packet verification, and data whitening. These steps can be seen in figure 2.12. The CRC is a 24-bit polynomial in the form of $x^{24} + x^{10} + x^9 + x^6 + x^4 + x^3 + x + 1$. All bits in the PDU are being processed for the CRC calculation starting from the least significant bit. For Data Channel PDU, the shift register containing the polynomial shall be initialized once the connection is established, and the value will be exchanged between the devices. For Advertising Channel PDU the shift register contains a fixed value of 0x555555 [3] Vol 6. part B.

Data whitening is used to avoid lengthy sequences of zeros or ones in the data bitstream. It is applied on the PDU and CRC fields of all Link Layer PDUs. It is first performed after the CRC in the transmitter. De-whitening is performed before the CRC in the receiver. All BLE packets go through the data whitening process, which essentially scrambles the data. The scrambling is based on a pseudo-random sequence generated from the channel of the devices in communication. After the CRC has been added to the packet, the data is XORed with the pseudo-random sequence. Afterward, the output is transmitted. Upon reception, during the de-whitening, the device uses the same sequence to derive the original packet from the scrambled one before proceeding to verify the CRC.

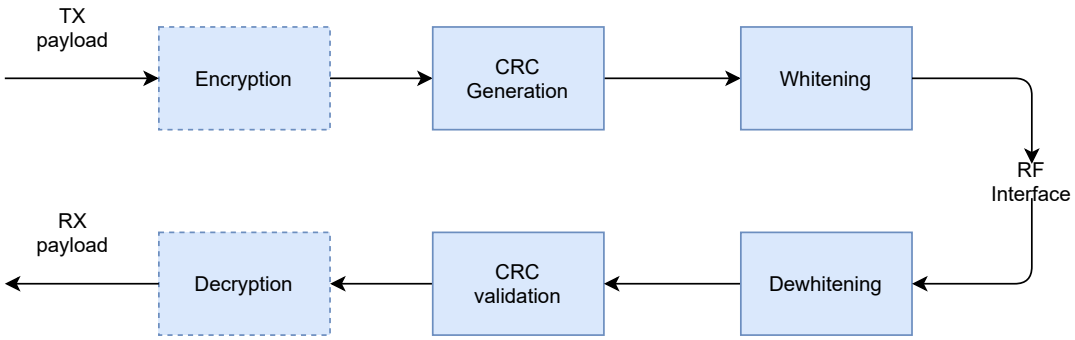

Figure 2.12: Link Layer Operations

L2CAP

The Logical Link Control and Adaption Layer Protocol sits on top of the HCI layer on the host side. It provides data services to upper-layer protocols with protocol multiplexing capacity, segmentation, and reassembly operations. These operations allow L2CAP and higher-level protocols (ATT) to use larger payload sizes and reduce the overhead. When segmentation is used, larger packets are split into multiple link-layer packets and are reassembled by the link layer. L2CAP layer utilizes channels, which are the logical connections between two endpoints, defined by their Channel Identifiers (CID).

GAP & GATT

Generic Access Profile (GAP) defines the general topology of the BLE network stack. Its operations include peer discovery, data broadcast, and secure connection establishment. Generic Attribute Profile(GATT) describes how data is transferred once devices have established a connection. GATT uses the Attribute Protocol (ATT) as its transport protocol to exchange data between devices. This data is organized hierarchically in sections called services, which group related pieces of user data called characteristics. The characteristics can be defined as data container units. They include at least two attributes: the characteristic declaration, which provides the metadata about the actual user data, and the characteristic value, which contains the actual user data in its value field. It is also worth noting that GATT follows a server/client connection relationship where a client finds a server through service discovery processes and then sends them RPC requests to access desired attributes and receives responses from them.

2.6 Connection Establishment

The connection establishment happens in the Link Layer of the BLE Protocol stack as it is the one responsible for advertising, scanning, creating, and maintaining connections. Assuming there are two devices: Host A (Initiator) and Host B (Advertiser). A sequence diagram of how these interact with each other can be seen in figure 2.13.

1. Discovery Phase: The Advertiser broadcasts ADV_IND packets, which contain information such as the MAC Address, RSSI, etc., to let Initiators and Scanners become aware of its presence.
2. Connecting Phase: The Initiator selects an advertiser from the received ADV_IND packets and responds with CONNECT_REQ packet, which contains frequency hopping sequence, connection interval, timeouts, etc. At this point, the role of Host A changes from Initiator in a connection.
3. Connected Phase: Once the devices are connected, they go through an operations sequence and information exchanges necessary before any user-initiated data is exchanged between the two. These operations contain:
    - Version Exchange: Bluetooth Version used
    - Feature Exchange: List of enabled Features
    - MTU Exchange: The MTU gets determined by the minimum value of MTU that the client and the server support.
    - Attribute Discovery: Client-Side operation. It caches the server's information to skip the discovery process the next time it connects to the server.

    At this point, the Initiator becomes Master, and the advertiser becomes Slave in the Server/Client relationship. Once this sequence of operations has been completed, the devices can begin exchanging data at regular intervals known as connection events.

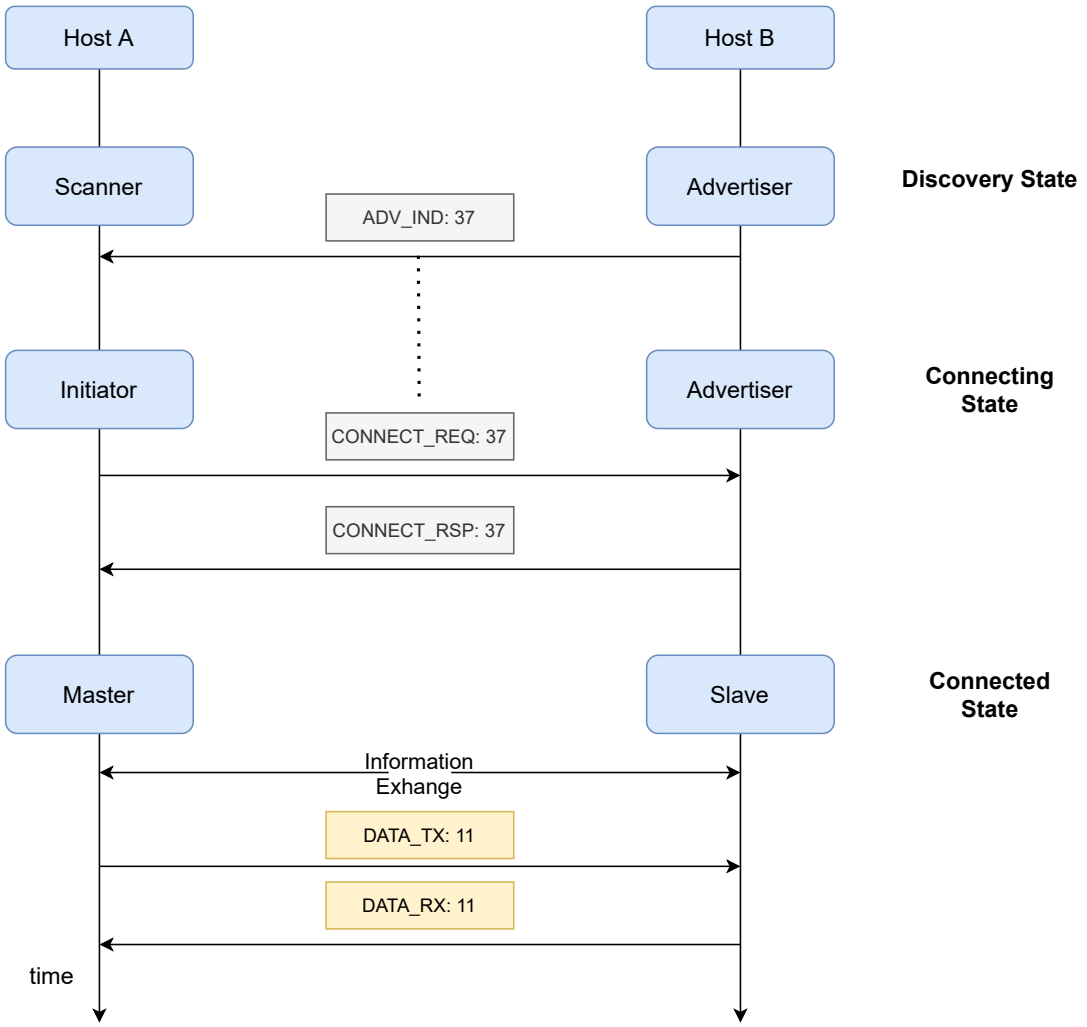

Figure 2.13: BLE Connection Establishment Procedure, [3] [Vol 6, Part B]

Frequency hopping in Bluetooth Low Energy

When transitioning into the connected state, frequency hopping is a necessity. While advertising the limit is lower as advertisement events occur less frequency. There can also be no hopping in this state if only one channel is used. The same applies to the Initiator if they choose to only listen on one advertisement frequency.
On the other hand, once two devices are in a connected state, they need to frequency hop often. This hopping is caused by each connection event requiring the next data channel to be used in the data channel map. The exact number is decided upon by the *Master* when sending the CONNECT_REQ as the parameter ConnInterval defines how often a connection event is performed. Depending on the value, the exact number of frequency hops will fall into the interval 133-0.25 hops per second.

2.6.1 ACKs and timings

For Bluetooth 4.0, the BLE Radio is capable of transmitting one symbol per microsecond, and one bit of data can be encoded in each symbol. A new feature was added in the 4.2 specification revision, known as LE Data Packet Length Extension. This feature allows the device to extend the payload length from 27 to 251, leading to 265 Bytes maximum size packets. One bit per symbol and one symbol per microsecond give a raw radio bitrate of 1 Megabit per second (Mbps). This raw bitrate is not the throughput that will be observed for several reasons:

1. There is a mandatory 150μs delay that must be between each packet sent. This delay is known as the Inter Frame Space (IFS).
2. The BLE Link Layer protocol is reliable, meaning every packet of data sent from one side must be acknowledged (ACK'd) by the other. The size of an ACK packet is 80bits and thus takes 80μs to transmit.
3. The BLE protocol has overhead for every data payload which is sent, so some time is spent sending headers, etc. for data payloads

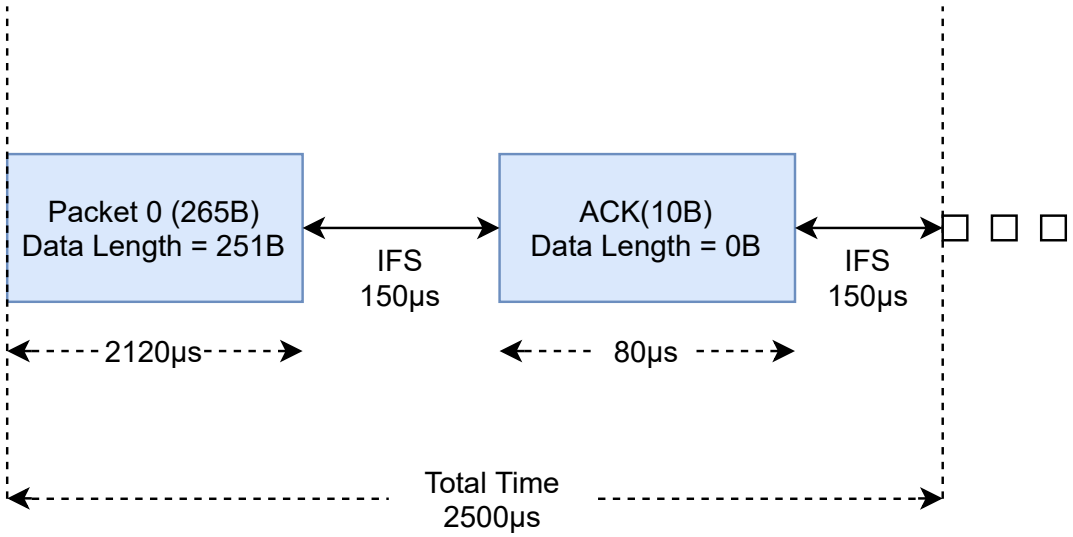

Figure 2.14: Total transmission time

The time it takes to transmit one packet can be computed as: data_packet_transmission + IFS + ack_transmission + IFS. The throughput is equal to 265/2500 = 106kBytes/sec = 0.848Mbps The goodput(data throughput) is equal to 251/2500 = 100.4kBytes/sec = 0.803Mbps

2.6.2 HCI

The Bluetooth core system consists of a Host and one or more Controllers. A Host is a logical entity defined as all of the layers below the non-core profiles and above the Host Controller Interface (HCI). A Controller is a logical entity defined as all of the layers below HCI. An implementation of the Host and The Controller may contain the respective

parts of the HCI. Host Controller Interface (HCI) enables communication between the Host and the Controller via a serial interface. Since the Controller deals with strict real-time requirements and contact with the physical layer, while the Host is responsible for complex implementations of the functionalities of BLE, it is recommended to keep these entities separated.

The Host Controller Interface (HCI) provides a command interface to the Controller and link manager and access to hardware status and control registers. This interface provides a uniform method of accessing the Bluetooth baseband capabilities. The HCI layer is implemented through a transport protocol such as SPI or UART. The API defines a set of commands and events to translate raw data into data packets to send them from the Controller to the Host and vice versa. [3] [Vol 1, Part A]

3 Problem statement

New technologies allow for SDRs to be connected to a host via PCI connection which offers excellent speed increases in the data communication, up to 128 GB/s (PCI V6.0 x16). This speedup could allow for previously unsupported protocols to be integrated and consistent concerning their timing requirements. Therefore, an investigation will be performed to find out if the timings of the current implementation can be reduced to follow the requirements defined in the BLE 4.2 core specification.

Based on these points, the project will focus on the following problem statement:

How can a Bluetooth Low Energy controller be implemented in a state-of-the-art Software Defined Radio platform while meeting the timing requirements of the protocol?

The above problem statement includes the following subproblems:

- How can the time required by the SDR for basic transmission/reception operations be optimized to reduce time consumption and reach the 150 $\pm$ 2μs requirement of the IFS?
- What is required to correctly implement the BLE link layer, a valid BLE frame generator, and minimize the execution time?
- How well can this implementation cooperate with another commercial BLE device?

3.1 Usecase

To define what is required for a BLE link-layer implementation, this section is used to describe the required functionalities according to the BLE specification.

Combination A	Combination B
Initiating plus any combination C of other states	Connection (Master role) plus the same combination C
Connection (Master role) plus Initiating plus any combination C of other states	Connection (Master role) to more than one device in the slave role plus the same combination C
Connectable or a directed advertising state plus any combination C of other states	Connection (Slave role) plus the same combination C
Connection (Slave role) plus Connectable or a directed advertising state plus any combination C of other states	Connection (Slave role) to more than one device in the master role plus the same combination C

Figure 3.1: Supported states required for the link layer implementation. Figure is from [3] Vol 6, part B, table 1.1.

As per the BLE, 4.2 specification, a Link Layer implementation is not required to support all states. However, if it supports a *Combination A* state, the corresponding *Combination B* must also be supported, from figure 3.1. Combination C, which refers to other states, is allowed to be empty.

Because the goal is to investigate if the SDR can be used for BLE communication, a complete implementation is not required. Therefore, it is chosen to support the first Combination A and B states for this implementation. This choice means the implementation will first of all support the initiating state. It will also support the connection state, in which the role Master is only supported. The following message diagram is used as an example of the communication that must occur, from being an initiator to being a Master, to describe the requirements for this Combination.

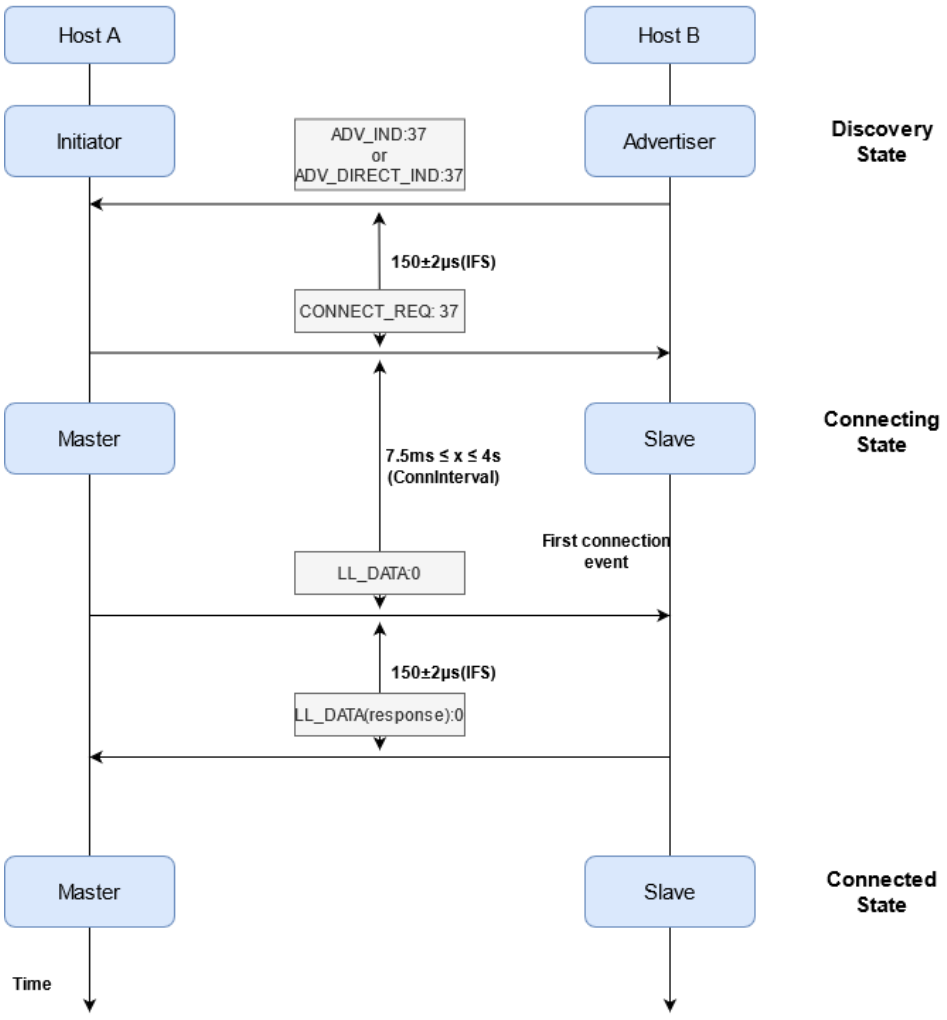

Figure 3.2: Sequence diagram of the messages required to create a connection.

On figure 3.2 is the sequence diagram that describes the required frames that must be supported. This figure also includes the timing requirements to reach a full connection state between a master and a slave.

If the Proof of Concept (PoC) is to also reach a further point with actual user-defined data transmission it may require the operations described in section 2.6 and figure 2.10. This requirement is due to the specification not strictly mentioning that these are required, but the Slave or Master may request it from the other device. Therefore, depending on the commercial device used, these may be required.

3.2 Requirements

The purpose of this section is to clearly state the system requirements and what is necessary to implement. These are intended to live up to the problem statement previously mentioned as well as the theory presented previously in the BLE part.

1. **Critical link layer & physical layer implementation:**
   (a) The ability to generate the following BLE frames for transmission:
      i. CONNECT_REQ - Connection request, as response to ADV_IND.
      ii. LL_DATA_PDU - Data packets, also used for Acknowledgement and negative Acknowledgment.
   (b) The ability to decapsulate the following frame types when received and decapsulation of data content.
      i. ADV_IND - Undirected connectable advertisement packet.
      ii. ADV_DIRECT_IND - Directed connectable advertisement packet.
      iii. LL_DATA_PDU - Data packets.
   (c) Functionality to employ modulation on the bits of the frames for transmission by the SDR.
   (d) The ability to demodulate data upon reception by the SDR.
   (e) Functionality to ensure switching BLE channels. Required to change from advertisement channel to a data channel, and data channels between connection events.
      i. 0.25 - 133.33 hops per second - Derived from the ConnInterval values.
   (f) Functionality to perform CRC check on received frames.
   (g) Respond $\leq 150\mu s$ after reception of a frame, with an appropriate answer.
      i. The accuracy of this must be $\pm 2\mu s$.
2. **Initiator requirements**
   (a) Only decapsulate connectable advertisement frames from white-listed device Addresses.
   (b) Ignore frames from unlisted device Addresses.
3. **Master role requirements:**

(a) Send connection requests upon reception of ADV_IND packets to establish a connection.

(b) Send connection requests upon reception of ADV_DIRECT_IND packets to establish a connection if the Master is the intended receiver.

(c) Perform the first connection events to finalize the connection.

4. **Data exchange requirements:**

   (a) The ability to generate the following BLE frames for transmission:

      i. LL_VERSION_IND - Bluetooth controller version.
      ii. LL_FEATURE_REQ - Contains Master Link Layer features.
      iii. LL_FEATURE_REP - Contains either Master or Slave Link Layer features.

   (b) The ability to decapsulate the following BLE frames for transmission:

      i. LL_VERSION_IND - Bluetooth controller version.
      ii. LL_FEATURE_REQ - Contains Master Link Layer features.
      iii. LL_FEATURE_REP - Contains either Master or Slave Link Layer features.

   (c) Upon reception of a control packet, send their associated control packet response.

   (d) The ability to discover features that the GATT server has available.

   (e) The ability to conduct MTU (Maximum Transmission Unit) exchange and set it to the lowest value.

The requirements ultimately lead to 4 milestones that are desired to be realized. The following milestones require all prior ones fulfilled before they can be realized.

The first milestone is the Link Layer and Physical Layer implementation. This milestone ensures the required functionalities to act according to the BLE protocol. That also includes the ability to transmit and receive data correctly.
The second milestone is the initiator state, an intermediate step before the connection state and Master role. The primary purpose is to only react to the correct packet types and device addresses.
The third milestone is the implementation of the Master Role who is required to establish a connection using the tools developed from the previous milestones.
The fourth and last milestone is to prove the functionalities of the PoC by communicating with a commercial BLE device.

4 Design

As can be seen from the problem statement, the report aims to create a PoC, that utilizes the lower layers of BLE along with an SDR to be able to send and receive messages. This PoC will work towards achieving the time constrains induced by BLE and verify that response packets can be transmitted successfully, as presented in the first milestone. Furthermore, for the second milestone, some aspects of the Link Layer state machine will be implemented to follow the logic of a connection establishment procedure. Based on this Link Layer Manager, the functionalities of the Master Role of a BLE connection will be integrated in order for the system to be able to handle connections and proceed to data exchanges. This will cover the third milestone. To create this PoC, the primary software modules that are needed are the following:

1. Higher Level Interfacing - Gives data-payload to Link Layer Manager for transmission and receives data-payload from Link Layer Manager.
2. Link Layer Manager - Manages connection and what to send. Sends input to Frame generator and receives input from Frame generator.
3. Frame Generator - Receives input from the link layer and generates a frame for transmission.
4. Frame decapsulator - Receives bit-stream from Demodulator, looking for the beginning of a frame, and decapsulates the frames from this bit-stream. Hands output over to Link Layer Manager
5. Modulator - Receives frames for transmission and transforms said bit-stream into I/Q samples for transmission by the SDR
6. Demodulator - Receives I/Q samples from SDR and outputs a bit-stream.

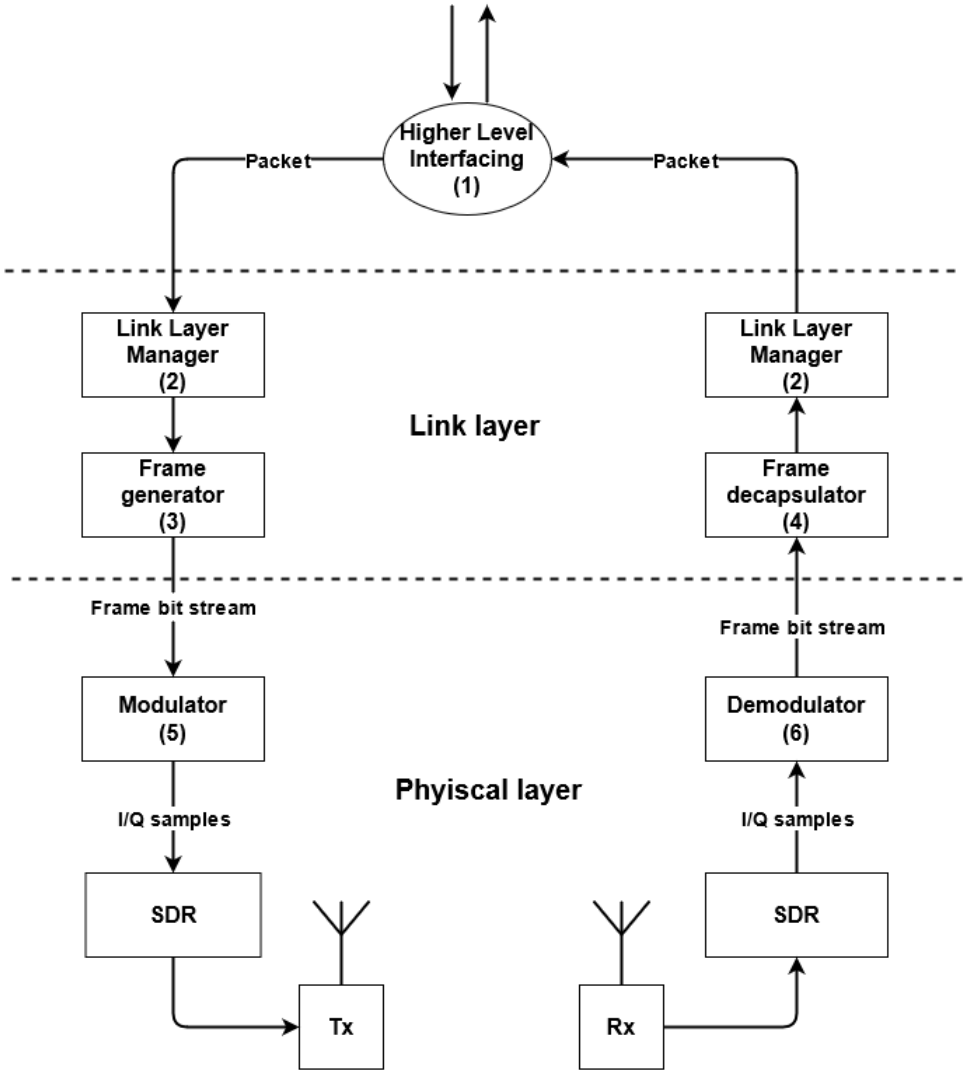

Figure 4.1: Block diagram of the PoC.

Illustrated on figure 4.1, the PoC will operate in the lower layers of the BLE stack, due to the purpose of this project is investigating the feasibility of implementing BLE while using the new SDR for the physical interactions. The PoC will further be divided into the previously mentioned six modules to simplify the design process and implementation. Beyond these, a configuration for how the SDR should operate must also be defined and used.

In order to verify both time execution and validity of BLE packets, a set of equipment for this exact purpose is also necessary for the development of the PoC.

4.1 Equipment

The test setup used for validations consists of four devices: A signal generator, a Signal analyzer, an oscilloscope, and a computer with an SDR installed. In this section, these

devices will be described, along with the purpose each of them serves. All equipment and software is made available by Keysight Laboratories for the purpose of this project.

4.1.1 Signal Generator

The signal generator is a device used to send out different signals so that devices can be tested. The device used in this testbed is referred to as *LittleTwo*. It serves the purpose of sending Bluetooth packets so that it can be tested whether or not the SDR is correctly receiving and demodulating packets. In figure 4.2 below, the interface of the generator can be seen.

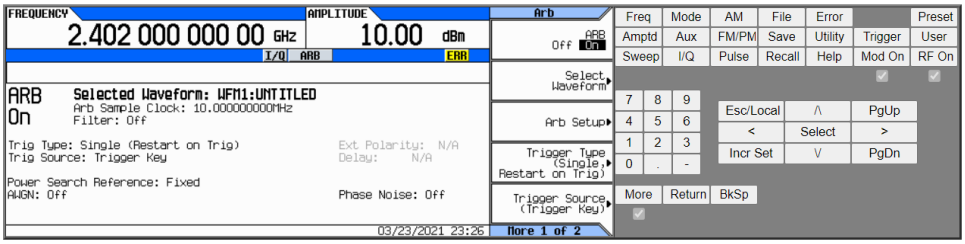

Figure 4.2: The front panel of the Signal generator.

As can be seen, there are controls for manipulating different characteristics of the signal being generated. Relevant examples in this project are altering the frequency of the signal and the power to improve the reception.

4.1.2 Signal Studio for Bluetooth

Signal Studio for Bluetooth is utilized to get the Signal generator to send valid BLE packets. Signal Studio for Bluetooth gives the ability to create a custom packet in adherence with the BLE specification and upload it to the Signal generator. The program includes picking the preferred channel to send on, the preferred payload, and the option of data whitening.

Figure 4.3: Example of packet type in Signal Studio for Bluetooth.

An example of how a packet type may be defined can be seen on figure 4.3. In this case the points of relevance are mentioned below:

- Packet type (ADV_IND)
- Channel index (37)
- Data whitening (On)
- Txadd (0)
- Advertiser address (0000000008), also referred to as, device address
- Payload (PN9) which is a specific random bit sequence.

Other parameters are also modifiable, while others e.g. Length of octets is determined by the exact amount of payload. This method for packet definition can be done for all BLE link layer packet types.

4.1.3 Signal Analyzer

The next vital device to describe is the Signal Analyzer. As the name suggests, the Signal Analyzer is used to receive and investigate signals of many different types. In this test setup, the Analyzer is used to validate the messages that are sent from the SDR. It is used for that purpose to investigate whether or not the system is working as intended.

Figure 4.4: The front panel of the Signal analyzer. Configured for 2.402GHz, and a packet is demodulated seen from the 296bits payload length.

As can be seen in figure 4.4 above, the Signal Analyzer is receiving a signal, and it is possible to see the frequency spectrum of that reception. The view is concentrated on the specific frequency used by the signal generator to send the packet. The most important thing to note from the Analyzer is the field that shows the packet type and the field that gives the packet length. These two pieces of information can be used to confirm that a packet sent from the SDR is generated correctly.

4.1.4 Oscilloscope

In this test setup, the oscilloscope is set to measure the time difference between messages. The Oscilloscope can show waveforms corresponding to signals it receives as voltage levels. The scope can show multiple signals (up to 4) so that it is possible to compare the timing and power differences. In this specific case, the Oscilloscope will be used to see the time between incoming messages and responses to these so that it can be seen whether or not the eventual PoC adheres to the timing requirements of BLE. An example of what it looks like when the Oscilloscope receives different signals can be seen below in figure 4.5.

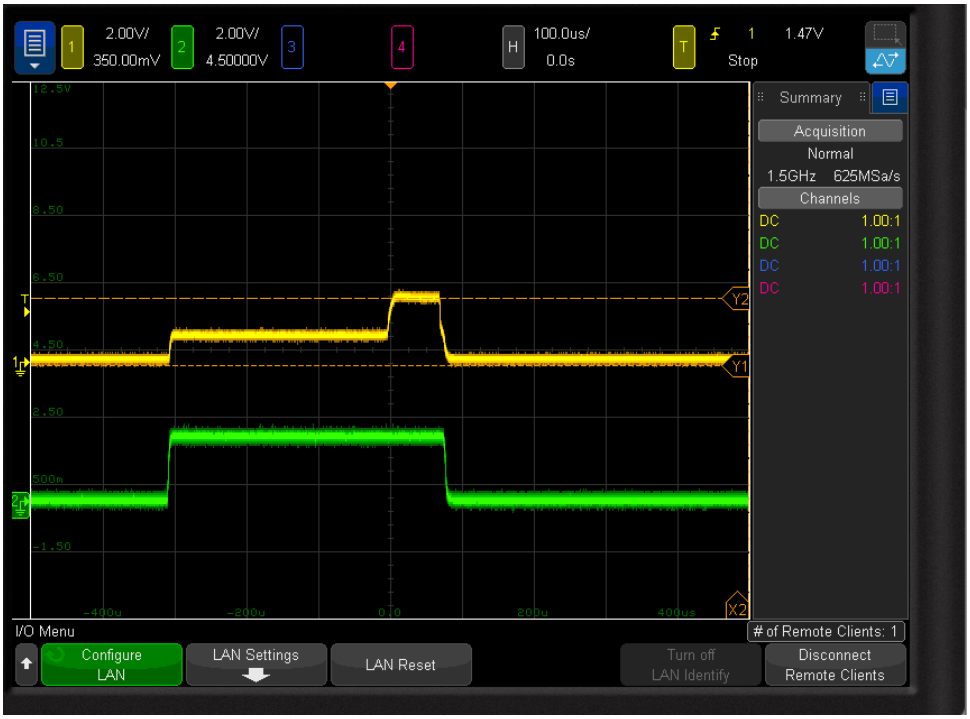

Figure 4.5: The front panel of the oscilloscope

On the front panel of the Oscilloscope, two signals are being received. In this image, the generator is sending, which is the signal represented by the green line. The yellow line is the signal coming from the SDR. The slight increase in signal strength from the SDR when the generator is transmitting is caused by leakage. An important thing to note in this figure is that the spike in strength seen at the end of the yellow line is a message being received by the Oscilloscope. Having these signals next to each other will make it possible to compare timings.

Lastly, there is the computer, *LittleTwo* that uses the SDR. The computer itself is running Fedora, and the goal of this computer is to implement different technologies that can use the SDR to send and receive messages. In this case, the goal will be to implement BLE and use the previously described devices to test its functionality.

As can be seen from the above description of the different elements, the test setup has multiple essential parts. A Signal Generator will be used to transmit signals and a Signal Analyzer to receive and verify the integrity of the packets. The SDR in use shall receive messages from the Signal Generator and will respond. The time difference between generator transmitting and SDR transmitting can be viewed on the Oscilloscope. Both the Signal Generator and the SDR are connected to the Analyzer and an oscilloscope to visualize and verify their corresponding output 4.6.

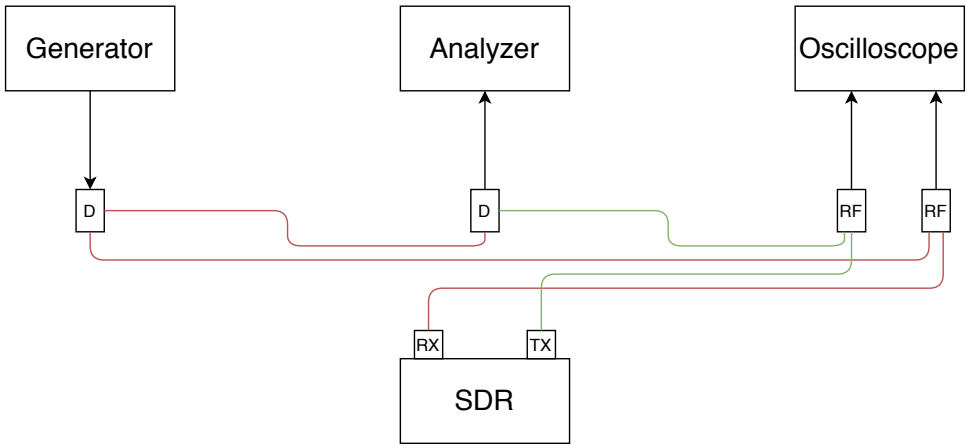

Figure 4.6: D stands for a signal divider. RF is the receiving interface of the oscilloscope. RX is the receiving antenna of the SDR. Tx is the transmitting antenna of the SDR. The green line shows the transmission path from the SDR. The red line shows transmission path from the Signal Generator

With the equipment chosen and set up in the appropriate configuration, the next thing to focus on will relate to software. In the next section, the design of how the SDR will interact with the BLE stack.

4.2 Higher level interfacing

With the equipment setup defined, the design will move onto the purpose of Higher-level interfacing and work towards describing each of the modules from a top-down view similar to what was shown in figure 4.1.

Higher layer interfacing will involve working with a Host Controller Interface (HCI) in the BLE stack. Using this would mean a generalized way for the Host and the Controller to interact with each other, which minimizes the workload when trying to implement the Host layer onto the Controller layer. Seeing as this project is focused on the Controller, that means having a working HCI could allow for easier future development of adding the Host on top of this PoC.

With that being said, the main purpose of this project would be to have a placeholder implementation for the HCI. The primary purpose of this module is to supply the Controller layer with data to be transmitted and have a place where data would be handed over to by the Link Layer Manager upon reception and decapsulation of the payload. Thus, if anything, this will not be an actual implementation of HCI, and is therefore not named as such, but will instead serve more as an imitation.

4.3 Link Layer Manager

The immediate communication after Higher-level interfacing will be with the *Link Layer Manager*. The manager handles the packet type to be transmitted (data PDU, adver-

tisement, and control packets) and establishes connections. Its main purpose is to pass on parameters for a frame into the *frame generator*, and manage the Link Layer state machine. The tasks include handling the packet type used and the frequency it should be transmitted on, according to the channel map of any connection.

On the reception side it is used to parse the decapsulated frames upwards in the stack to HCI, as well as listing and unlisting Device Addresses that are to be communicated with dependent on established connections.
On top of that, the Link Layer Manager is meant to handle different parameters associated with each link that the device may have.

The primary purpose of this module is to have a way of controlling things such as the link-layer state machine. Furthermore, it will also handle variables such as channel map for each connection, device Address that is being communicated with, and other variables associated with each possible link.

4.3.1 Master Role

The SDR starts in the Initiating State, where it looks for ADV_IND or ADV_DIRECT_IND packets to initiate a connection. Once an Advertisement packet with a matching Access Address is detected, the SDR begins the procedure of preparing the corresponding ACK and transitions from the Initiator to the Master role. The first step is to parse the received advertising packet and extract the required information contained, such as the Advertiser's Address, which is needed in order to reply. The reply is contained in the ACK packet in the form of a CONNECT_REQ packet. The packet must be transmitted in the IFS of $150\mu s$. The transmission of the CONNENCT_REQ packet establishes the connection between the two devices. The values contained in that packet define multiple communication parameters such as the ConnInterval, the WindowSize, the ChannelMap, and more, which are needed for future data exchange. From this moment and forward, the SDR exits the Initiating State and enters the Connection State with the Master role in this relationship. Similarly, the device which sent the ADV_IND packet will move from the Advertising State to the Connected State and become the Slave in the established connection.

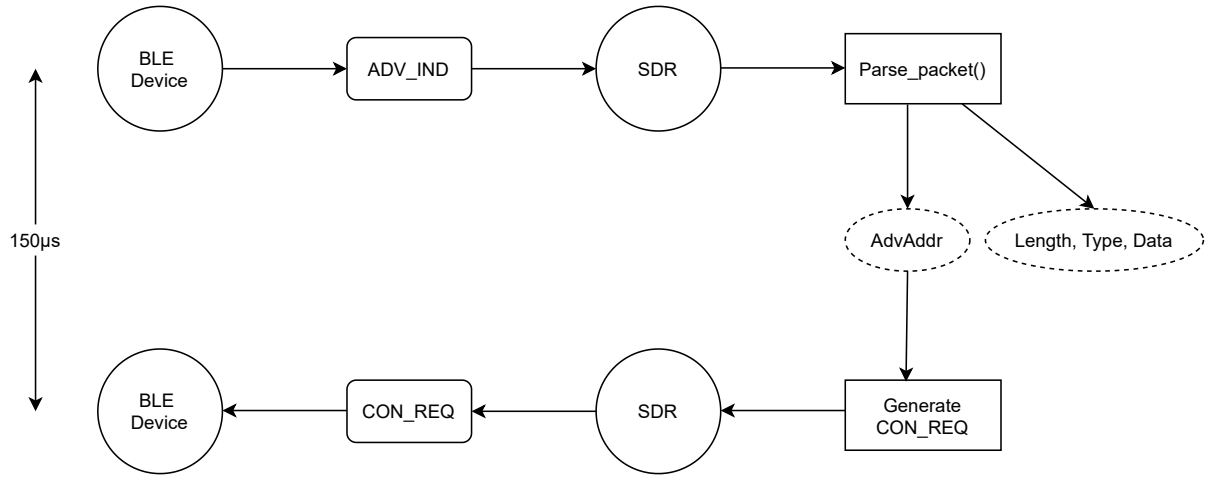

Figure 4.7: Procedure of entering the Master role from the SDR perspective once an ADV_IND is received. The same procedure applies for ADV_DIRECT_IND packets

4.4 Frame generator

The *frame generator* is located right below the Link Layer Manager and will be used to generate frames that can be sent to other Bluetooth devices. It will receive all relevant frame content from the *Link Layer manager*, who in turn receives it from a higher layer, where the packet payload may be larger. Therefore, the *frame generator* must be able to take the parts of the payload of a higher layer packet, given by the *Link Layer manager*, and transmit it as frames.
Thus, the generator will make BLE frames based on the structure given in the specification and thereby encapsulate data given by the upper layers.[3] From the previously BLE theory section, it can be seen on figure 2.10 that the first element in the frame is a 1-byte preamble. This preamble gives the receiver the necessary information for frequency synchronization, symbol timing estimation, and AGC training. As described in section 2.5, the following field in the header is the Access Address field. This field is used to define the connection between two devices on the link layer. Then comes the PDU field, which will consist of the data given by the upper layers, and a MIC that makes sure that the data has not been altered in some way. The last field of the frame is the CRC field, which is used to perform a check of the entire frame to make sure that it has not been compromised.

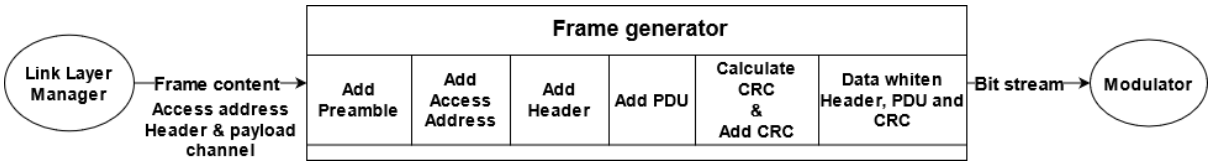

Figure 4.8: BLE frame generator

The processes that need to be performed by the *frame generator* can be seen in figure 4.8. It will be responsible for preparing frames by receiving necessary information: the

Access Address, PDU header, and payload. With this information, the correct preamble can be chosen and added before these three aforementioned parts of the frame. In the end, a CRC is calculated and added to the frame.

Afterward, the data whitening process is executed on the Header, PDU and CRC. The whitening will be a binary XOR executed on each of the bytes and will be dependent on the channel that the frame is to be transmitted on.

4.5 Frame decapsulator

The *frame decapsulator* will work oppositely from the *frame generator* as it is meant for decapsulating the content of a received frame. At the same time, this function is required to constantly run as it has to check each received bit if the beginning of a frame is present. This process will also entail synchronization.

The decapsulator will need to synchronize with a valid preamble. Upon finding one, it will start decapsulating the content of the frame. It must then find the access address and ensure that this is a valid one that it is meant to communicate with. If an access address is not meant to be looking for, then that frame must be ignored. The same applies to device addresses.
After ensuring that this access address is one of interest, the header field and all its relevant content must be de-whitened and the relevant content such as PDU length extracted. Another XOR operation would realize this by undoing the original whitening operation
The total size of the frame can be found using the length from the header field, which will be required to know for how much content can be decapsulated.
Finally, upon having received the entire frame and rebuilt it, the whitening process is undone, and the CRC must be computed for the payload and compared to the received CRC. If the CRC is correct, the PDU content can be parsed upwards in the stack to the process that the data belongs to. Depending on the result of this CRC check, an ACK or a NACK will be transmitted.

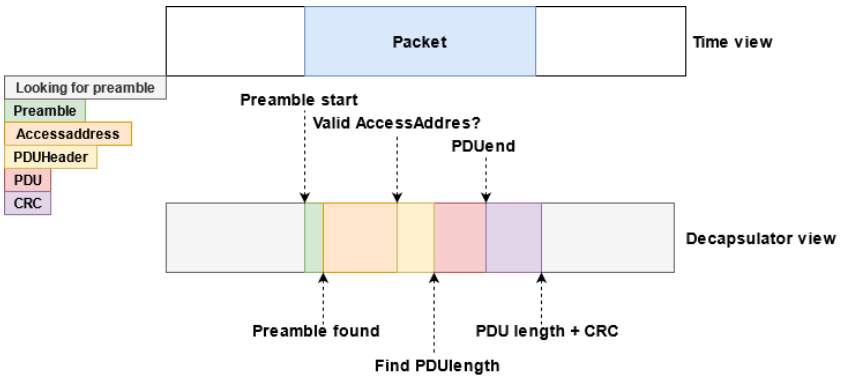

Figure 4.9: Visualization of frame decapsulator.

Figure 4.9 shows the general idea of how it is supposed to work when decapsulating the

frame. The CRC check and parsing of the PDU content to a higher layer will happen after the operations on the figure have been executed.

4.6 Modulation

After a frame is generated, it must be prepared correctly for on-air transmission, and this is realized through modulating the bits that the frame consists of.

Modulation is the process by which a signal is strengthened to reach its destination despite interference, distance to the receiver, or noise.[10] The important thing is to keep the characteristics of the original signal, the so-called base-band signal, when it is transmitted via the carrier signal, so that the receiver can demodulate it correctly. Modulation is important as it allows for smaller antennas, a better range of signals, and sending multiple signals without interfering with each other.[10]

Modulation for BLE requires GFSK to be used, or the "special case" Gaussian Minimum Frequency Shift Keying (GMSK). These two modulation types are a variant of frequency modulation, which is based on increasing or decreasing the frequency when sending 1-bits or 0-bits.

Both are valid in the eyes of BLE, as it is just required that the so-called Bandwidth time product is 0.5, and the modulation index is between 0.45 and 0.55. GMSK uses a modulation index at specifically 0.5 whereas GFSK is not bound to a specific value. The modulation index is related to the frequency that the 0-bits and 1-bits are mapped to when the base-band signal is mounted on the carrier signal. The larger the modulation index the more bandwidth is occupied [11] [12].

In order to transmit data, the modulation has to be executed in the correct fashion, where the process of this can be seen in figure 4.10

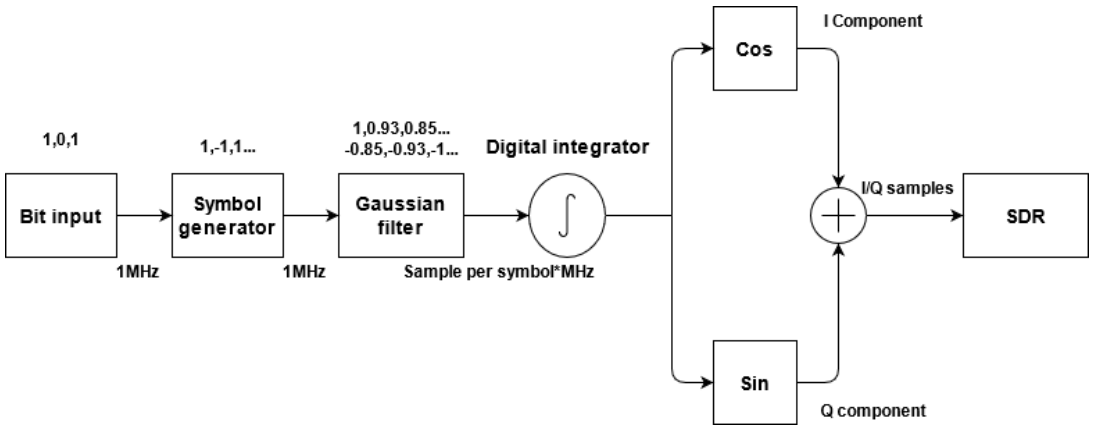

Figure 4.10: Block diagram of the general operations and their input to realize modulation.

Figure 4.10 illustrates the process that bits will go through to be transmitted through the SDR. First, they are encoded into symbols. Then they are passed through a Gaussian filter through convolution and afterward to a digital integrator which is utilized to

generate I/Q samples, which can further be fed to the Tx buffer of the SDR. I/Q samples result from the modulation scheme and are transformed into the baseband signal that is mounted on the carrier frequency. This process is something that is done by the physical SDR.

This process starts with a stream of bits encoded into a Non-return to zero (NRZ) symbol space, which would be {1,-1}. The symbol generator thereby transforms 1-bits to 1 and 0-bits to -1. Furthermore, between these symbols, samples of value 0 are added in. Thus [1,0] as bits will look like [1,0,0,0,-1,0,0,0] if samples per symbol ratio is 4. More zeros would be seen if the ratio was of a greater size. This process is done to use convolution with a Gaussian filter.

The Gaussian filter utilized is a finite impulse response that is distributed according to the Gaussian distribution. The reasoning behind using it is to minimize spectral bandwidth in comparison to, e.g., Frequency Shift Keying (FSK) modulation. [13]. The difference in result is that FSK for example, has sharp changes between the symbols, whereas GFSK has smaller changes between these, with going in the direction of the 1 or -1 rather than going straight to it. This different approach leads to some intersymbol interference for GFSK, and if a sequence of the same bits is transmitted, the effect of the Gaussian filter dies out.[11]

Equation 1 shows how a Gaussian Finite Impulse Response (FIR) is defined mathematically, and thus how a filter can be generated using the four inputs that it takes.

$$h[k] = \frac{\sqrt{2\pi}}{\sqrt{ln(2)}} \cdot \frac{BTs}{Ts} \cdot e^{-\left(\frac{\sqrt{2\cdot\pi}}{\sqrt{ln(2)}} \cdot \frac{BTs \cdot k}{OSR}\right)^2} \tag{1}$$

Where:

k is the kth index in the impulse response

Bts bandwidth time product which for BLE is set to 0.5.

Ts is 1μs.

OSR An oversampling factor which is an integer e.g. 4.

There are naturally infinite indexes that could be generated. To realize the FIR filter values that fall below a comparatively small value are discarded without it having any relevant consequences, meaning everything after reaching this small value can be ignored.

4.7 Demodulation

Demodulation of GFSK is realizable by different methods. One of these ways, as described by [11], can be achieved by applying a "Delay and multiply" transformation. This method is typically applicable to Frequency Modulation (FM). It is based on shifting the passband

signal to the baseband and extracting the I and Q components of the signal. Afterward, the following equation is applied.

$$d(k) = q_k * i_{k-1} - i_k * q_{k-1} \tag{2}$$

Where:

q_k is the kth Quadrature component

i_k is the kth In phase component

$k-1$ elements is the previous set of In phase and Quadrature components. This could also be different than just the previous sample.

In equation 2 two sets of samples are used to calculate the bit the samples are generated from. Depending on what the equation yields, as a result, the bit is either 0 or 1. The exact way of determining is as follows if d(t) $<$ 0, it means the samples belong to a 0 bit, and d(t) $>$ 0 means they belong to a 1 bit. The purpose behind this being to calculate the derivative of the phase to find out if it is positive or negative. Thus, the original bit can be found again by computing the d(t) function with I/Q samples.

It should further be mentioned that in this case, only the difference between two consecutive samples is used as an example. It might not be enough if a higher sampling rate is used, seeing as noise may have a more significant effect. The result is that the size of the difference between samples is somewhat dependent on the sampling rate.

This description is the general notion of how the demodulation may be realized without going into too much detail.

4.8 BTLE Library

To focus on points of interest for the project and allow more time to focus on time-optimization instead of implementing everything related to BLE a library has been chosen. The BTLE library is used as it fulfills many uses outlined in section 4.4 to 4.7 and as this will allow for more time, which will simplify the focus of the project. It does, however, not cover the *Link Layer manager* and *Higher Layer Interfacing* which still require an implementation. Other aspects are somewhat entangled and may require modifications, but a foundation that the Bluetooth part of the PoC requirements are realized through this library.

BTLE is a free and open-source BLE software suite developed by Xianjun Jiao [14]. It includes a complete BLE sniffer, used for receiving on a broadcasting or a fixed channel and track a channel hopping communication link. Also, it includes a BLE frame generator used to create frames. The layers are implemented in C language. The libraries provided by this implementation will be used mainly on this project to handle modulation/demodulation and frame generation. For transmission/reception events, the SDR provided library will be used to transform the received signal into samples which will

then be forwarded to BTLE. Higher layer operations are a low priority, and if required, they will be built on top of the *Link Layer manager* through HCI.

To validate the authenticity of the library, two functions that followed the SDR example code are used. Mainly one function was used to validate the authenticity of the produced samples by the library, which it did. More can be found in chapter 11, as it was a tool used earlier on to ensure the correct use of the SDR. Beyond this, the equipment from section 4.1 is also used for this purpose.

This section will describe the two main functionalities of this library being the Receiver and Generator, regarding how they work. It will also serve as a finishing touch to BLE part of the design.

4.8.1 BTLE-Tx

Originally this code was written to work with bladerf and hackrf these two are other types of SDR units. I/Q Samples are generated and transmitted on those boards using the following call syntax:

Code snippet 2: How an ADV_IND type packet originally would be generated.

```
1 ./btle_tx 37-ADV_IND-TxAdd-0-RxAdd-0-AdvA-90D7EBB19299-AdvData-02010507020318021804 18-space-1 r1
```

Where:

37 is the channel the packet should be made for and transmitted on, in this case, that is 2.402MHz.

ADV_IND is the packet type used, and different packet types can take different inputs when being called.

$TxAdd - 0$ indicates if the transmitters access address is public (TxAdd=0), or if it is random (TxAdd=1).

$RxAdd - 0$ indicates in advertisement PDU's if the advertiser's access address is public (RxAdd=0) or if it is random (RxAdd=1).

$AdvA - 90D7EBB19299$ is the access address of the transmitter.

$AdvData - 020105070203180218 0418$ is the payload being transmitted in hex.

$space - 1$ is a millisecond period of nothing being sent. This parameter is not relevant for this implementation and will not be used any further.

$r1$ is the number of times the packet should be repeated.

When being called, the Tx part of the library interprets the command-line arguments for what each of the fields means regarding creating a frame as seen on code-snippet 2. The command-line arguments do not require the order to be specific except that channel and packet type are the first two values specified. Other values such as *TxAdd* or *RxAdd* do not require any specific order as the code looks through the command line input for the occurrence of these and takes the following value after the dash.
After extracting parameters seen from above for the packet that is related to what is

to be sent, by who and how other derived values are calculated, i.e., preamble, payload length, and CRC. At the end of it all, a binary sequence that is the packet is saved in an array. This binary sequence is then data whitened according to the channel this is to be transmitted on. Upon finishing this data whitening operation, the bits are given to a sample generator that generates the I/Q samples based on the packet's data-whitened state. The I/Q sample generator takes the binary sequence and a Gaussian filter dependent on the used sampling rate. The output samples are then by default in the range of {-128,128}.

The project does not work with bladerf or hackrf. Therefore some changes had to be done to begin with, which led to removing all mentions of these. These changes include actively saving samples to a file, as this is the only way to retrieve them as well as changing the I/Q samples to fit the used SDR. Through these changes, the file could be loaded into sdr_play, a script provided with the SDR that functioned to transmit I/Q data (More in section 11). The samples were then transmitted to examine the validity of the packets that were generated. This validation was done to ensure that the BTLE library created correct packets and I/Q samples.

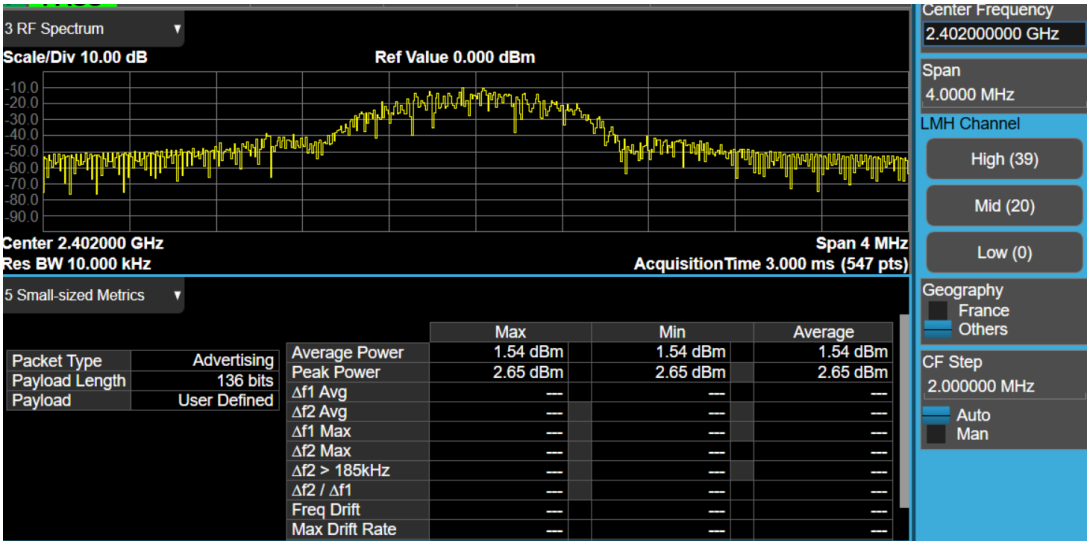

Figure 4.11: How a packet was demodulated by the signal analyzer looks like

In figure 4.11 the packet type and payload length can be read. According to the generated packet's specifications, these fit what they are supposed with an access address of 6 bytes and a payload of 11 bytes. Adding those numbers together and multiplying with 8 gives the seen payload length of 136 bits.

Code snippet 3: Example of LL_DATA PDU creation

```
1 ./btle_tx 9-LL_DATA-AA-60850A1B-LLID-1-NESN-0-SN-0-MD-0-DATA-XX-CRCInit-A77B22 r1
```

As the focus of the sdr_example code was to minimize the time such that it fits within the requirements of BLE it was also checked with what kind of time it took to generate packets. The reference point used for this purpose is the call seen in code-snippet 3. This packet is used as a reference due to it being the smallest possible frame for BLE.

Hence this lays a foundation for how fast the smallest BLE frames may be generated. Another thing worth noting is that different parameters were used for this one as opposed to code-snippet 2. The different set of parameters used is simply because different packet types take different input, allowing for freedom to create frames relatively freely.

```
[root@alliot-littletwo terminalbtle]# ./iqsamplegenerator 9-LL_DATA-AA-60850A1B-LLID-1-NESN-0-SN-0-MD-0-DATA-NULL-CRCInit-A77B22 r1
Running the program took 94µs
```

Figure 4.12: Terminal execution time example of LL_DATA type packet.

Executing the code with these parameters took $95 \pm 5 \mu s$ on average to run the btle_tx code. This number was the time required to generate a frame of 80 bits as specified by code-snippet 3 and modulating its I/Q samples. The performance is measured after shedding unnecessary elements. These mainly include prints and excessive file writing. The file with the I/Q samples to be transmitted was still kept, as this is currently the only way to extract them. Measurement of the program execution was done by measuring the beginning of the program until the end.

New filter

As a result of the SDR having its best performance at 40MHz, a new filter is necessary. The filter is required because the BTLE library does not contain any pre-made filters that will allow for a sample per symbol of 40, which is what is needed. Only lower sample per symbol rates at 4 is defined by default. The one used to produce the average execution described was this filter and sample per symbol rate.
The need for a new filter means the modulation theory from section 4.6 will be used to produce a new filter made for 40MHz to modulate the samples correctly for this new sampling rate.

4.8.2 BTLE-Rx

The BTLE-Rx code was, like in the case of Tx, made to work with bladerf and hackrf SDR units. It is worth mentioning that the BTLE library is an independent library designed to work with the aforementioned devices and has not been designed to work with the SDR used in this project. It does, however, offer the functionality that is needed to make a functioning Proof of Concept. The implementation has functionalities that allows one to demodulate frames received on different BLE channels and determine the relevant type of frame and it is also able to compute the CRC to make sure that a frame has not been compromised. The hardware specific functionality is not needed for this project and has thus been removed. The main function of interest in the BTLE Rx library is the `receiver()` function. This function is implemented so that it will take the samples in a buffer and demodulate where it first finds the access address and thereafter the information contained in the header with which it then can demodulate the whole frame. This means that the entire frame should be in the buffer given to the function. The function can be seen in code snippet 4 below.

Code snippet 4: The function used to demodulate and parse frames.

```
1 void receiver(IQ_TYPE *rxp_in, int buf_len, int channel_number, uint32_t access_addr,
2 uint32_t crc_init, int verbose_flag, int raw_flag)
```

Where:

$*rxp\_in$ is a pointer the the I/Q samples in a buffer that should be demodulated

buf_len is the length of the buffer that is given to the function

$channel\_number$ The channel (frequency) on which the samples have been received

$access\_addr$ is the access address of the receiving device

crc_init is an initiation value that will allow the function to calculate the CRC. This initiation value can vary based on the frame type.

$verbose\_flag$ is used for extra printouts for debugging

raw_flag The raw_flag is used to determine if the data should be dewhitened

When the function is called it produces an output of the frame which looks like figure 4.13 below:

```
XXXus PktBAD Ch37 AA:8e89bed6 ADV_PDU_t0:ADV_IND T0 R0 PloadL45 Error: ADV payload length should be 6~37!
ADV_PDU_t0:ADV_IND T0 R0 PloadL37 AdvA:000000000008 Data:ffc1f3e84c90728be7b3518963ab232302841872aa612f3b51a8e53749fbc9 CRC1
```

Figure 4.13: The result of the receiver being printed in the terminal

In the picture, two different outputs of the receiver function can be seen. The first line in the picture is the result of the receiver trying to demodulate samples that are either not a frame or samples that make up an incomplete frame. In this specific case, it has found the error that the payload does not match the required length. The second line in the picture is the result of the receiver correctly demodulating a frame where it then outputs the relevant information such as the access address of the sender and the payload of the frame.

As the receiver undertakes the process of demodulating frames, it goes through different phases related to discovering new incoming frames and unpacking the information. When the receiver is called, the function is passed a buffer and then the receiver initializes different variables that are used to hold different parts of the frame and values used for comparison. To find the beginning of a frame the *$search\_unique\_bits$* function is called, that looks through the buffer given when the receiver was called. This function goes through the buffer and looks for the unique sequence of bits defined in the setup of the receiver. The function looks for the access address sent in the frame. To actually compare the buffer content to the desired bit sequence the search function has to demodulate the content of the buffer. The buffer that is being searched through contains samples that make up symbols which can be demodulated into the bits from the original messages. It is important to construct each symbol based on their corresponding samples, meaning that it is necessary to be synchronized with the samples to correctly demodulate the frame. If this synchronization is not there, the code could perform demodulation over samples

belonging to different symbols resulting in a wrong result. To achieve this synchronization in practise, the search function starts on the first sample in the buffer and demodulates the content from there. As the function demodulates it will compare the sequence it is finding to the sequence that is being searched for. If a bit in the sequence does not match the one demodulated, it will stop and move forward to the next sample and repeat the process. Only going forward one sample after each iteration ensures that the search function will make the future demodulation synchronized to the right samples, as when the function finds the correct bit sequence, it will return the placement of the beginning of the frame down to the first sample. One thing to note is that this function is affected by the sampling rate, as changing the sampling rate changes the samples per symbol which leads to more samples having to be investigated.

The receiver jumps to the beginning of the header frame and demodulates it using the function *demod_byte*. The demod_byte function is simple, as it takes a buffer of samples, demodulates the number of bytes inputted and returns the demodulated bytes in a new buffer. The only requirement for the input is that there should be at least one byte to be demodulated for it to function properly. The receiver then runs the demodulated data through a *scramble_byte* function to make the data readable by de-whitening it (see section 2.5.3 for more details). The receiver parses the information in the header, making sure that there are no obvious discrepancies in the information, meaning that it checks if the stated number of bytes in the payload is neither too small or to large to match the received packet. Then it uses the payload length given by the header to figure out how much is left to demodulate. Afterwards the receiver executes the the demod_byte and the scramble_byte function as before to get the remainder of the packet. Finally, the receiver performs a crc check to see that the packet has not been corrupted, before it parses the information from the frame payload, prints the results, and finishes executing.

As time is an important factor in the PoC it will make sense to look at the time performance of the elements of the receiver mentioned in the paragraph above. The average execution time of the different processes in the receiver can be seen on table 6 below.

	search unique bits	demod byte (pr. byte)	scramble byte	CRC check	parse header	parse payload
avg. time	2898ns (32 bits)	46ns (pr. byte)	14ns (pr. byte)	238ns	5476ns	8212ns

Table 6: The average execution time of different processes in the receiver. (1 million executions)

The first thing to note is that the parsing payload process in the receiver changes based on the packet, however this change is inconsequential. The parse payload is, in any case, the slowest part of the receiver but that is not a problem since it is only called when the elements of the packet has been found and demodulated. One thing that is important to comment on is the search unique bit function. As written in table 6 the time showcased is how long the search function takes when going through the exact length of the header. In actuality, executing this function takes longer depending how many samples comes before the actual frame beginning. The important thing is that the search function is fast enough to avoid building up a backlog of samples so that it can adhere to the IFS

in terms of responding. In practice that means that the receiver needs to search at least one new bit pr microsecond to avoid backlog. This is upheld in the current version of the receiver and will need to remain that way after modifications on the receiver, which will be discussed in the following section.

4.8.3 Demodulation buffer

An immediate concern that appears with this implementation is the size of the buffer that is given to the receiver function. As mentioned above the receiver needs the buffer to contain a full frame before it can perform demodulation. This means that the buffer used has to be big enough to ensure that at least one frame has been received. This means demodulating samples as they are received is impossible with the original implementation. It also raises questions in regard to how big the buffer should be to contain the largest BLE frame as it may differ from 112 bits to 2120. Therefore, this requires modifications for better performance.

Demodulation design

As mentioned before, the current implementation of the receiver function of the BLE library that is being utilized in this project requires the whole frame. This will take up a lot of time and will prevent the PoC from adhering to the time requirements. Using a smaller buffer will decrease time consumption but in return, it will become more difficult, if not impossible, to demodulate a packet. An illustration of this problem can be seen below on figure 4.14.

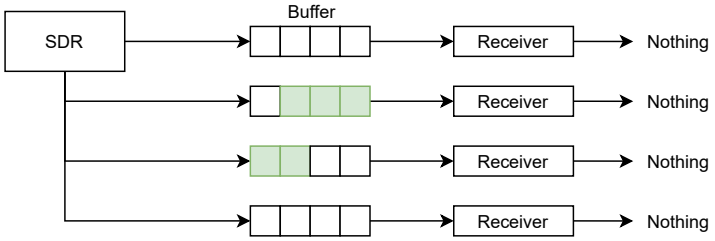

Figure 4.14: An illustration of the buffer size and its influence on reception. The coloured boxes are components of an incoming frame.

As can be seen, the frame has been received but due to the buffer being too small, the frame has been split into multiple receptions and will thus not be interpreted correctly. Instead of looking for an entire frame in each buffer, the receiver should decapsulate frame content as it is received. This would make it possible for the receiver to use small buffers to demodulate a frame. The library chosen for this project already has a function to look for a sequence of bits and compare them to specific parts of a frame. This function can therefor be applied to locate the beginning of a packet as it arrives at the receiver. The buffer used will only have to be slightly bigger than the bit sequence that is being

searched for and it can then be treated as a window going over the samples coming in and looking for the bit sequence. An illustration of this process can be seen below on figure 4.15.

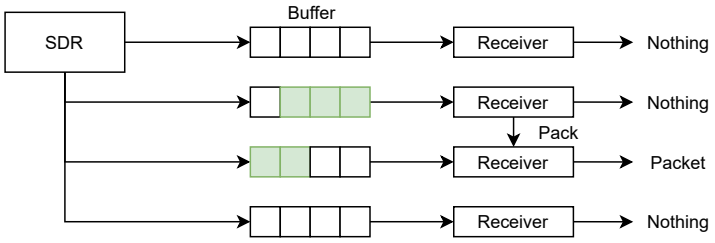

Figure 4.15: An illustration of the frame being captured over multiple buffers. The coloured boxes are components of an incoming frame.

It is worth noting that this process of demodulating a frame as it is received will allow the PoC to extract useful information as soon as that part of the frame has been received and demodulated. This will increase flexibility further in the PoC.

4.8.4 Response optimization

Originally, each operation of the current PoC waits for the previous one to finish before starting, as showcased in 4.16.

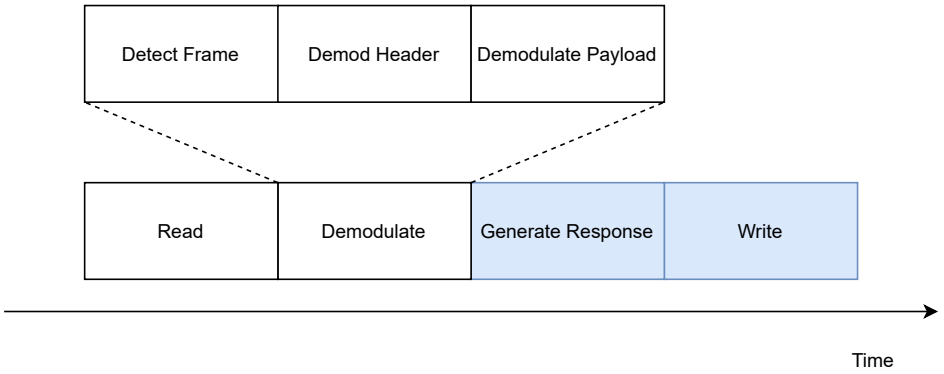

Figure 4.16: Original method. Generation called after entire demodulation.

An upgrade of this design will now be presented, which came as a solution to achieve better response times from the moment a packet is received to the moment its response is transmitted. The responses that will be focused on from this point are the following:

- CONNECT_REQ
- ACK/NACK
- LL_FEATURE_REQ

- LL_FEATURE_RSP
- LL_VERSION_IND

The purpose of the new design is to begin the preparation of the responses as soon as possible. The RX part of the custom library is to be altered for optimized response generation to achieve this. Compared with the BTLE TX part, it lacks flexibility in creating multiple packet types and sizes. Henceforth, the focus will be on connect_request and ACK/NACK, as these are required to reach the critical milestone. Connect request is just a single frame that will require two fields upon reception of an ADV_IND and ADV_DIRECT_IND. Based on the device address found here, it must be added to the connect_req packet. Afterward, the packet is ready for modulation and transmission. The connect request is only required to be made and sent once upon establishing the connection between Master and Slave. Regarding the Feature and Version packets, these too have a constant packet structure known beforehand. For that reason, they too can be prepared.

Opposed to the connect_req ACK and NACK are two separate responses while only one is sent off depending on the result of the CRC. The ACK and NACK are especially relevant because they must be made for each transmission occurring in a connection event.

The received packet's header contains all the necessary information for the ACK to be generated, like the Access Address, the packet type, tx_add, rx_Add, and the sequence numbers. So, the moment the header is demodulated, the ACK can be generated, as seen in figure 4.17. The demodulation of the payload is only needed for the CRC verification, which will turn the ACK into a NACK for transmission in case it fails. Generating a NACK packet does not need to begin from the start since it only differs by 1 bit from the ACK packet, so a bit flip in the already generated packet is sufficient.

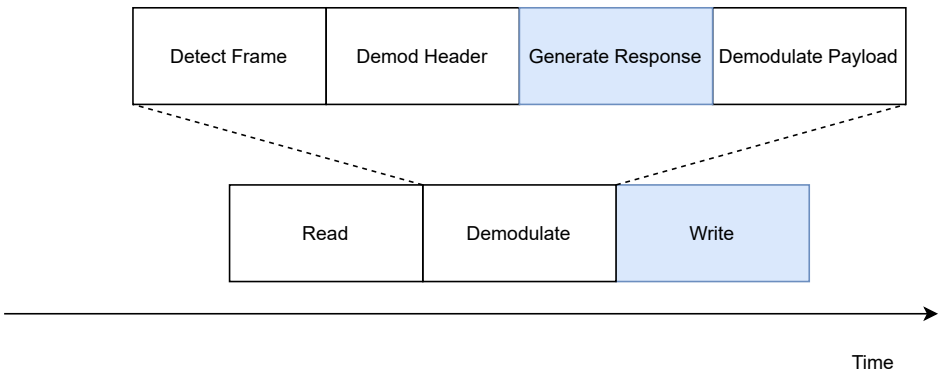

Figure 4.17: Optimized method. ACK generation occurring in the middle of the demodulation process.

They need to have optimized ACK, and NACK generation, which arose from the observation that the BTLE TX library was relatively slow at doing frame generation and modulation. With ACK and NACK being the significant bottlenecks from the perspective of execution time, these had to be reduced. If not, the original $95\mu s$ time to execute

frame generation with BTLE TX would most likely be far too heavy to make everything happen in time for the IFS interval of $150 \pm 2\mu s$.

With this implementation of BLE described and modifications planned for better performance, the design can move onto the last area of interest being the SDR.

4.9 Tx_handler

From the SDR investigation made in section 2.3.3, it was found that only 450 bits worth of data can be transmitted at once. Splitting the transmission up in separate msdr_writes is also difficult to time correctly. Furthermore, there is also the obstacle of samples remaining in the internal buffer indefinitely.

Based on these two issues, an abstraction is intended to be made to easier use the msdr_write operation and transmit frames of variable size and ensure cleanup of the internal buffer.

The abstraction creates a function that handles the logic involved in timing when to transmit and clean up. It is also intended not to have any size restriction such that even the largest BLE frames, as well as the smallest, can utilize this function.

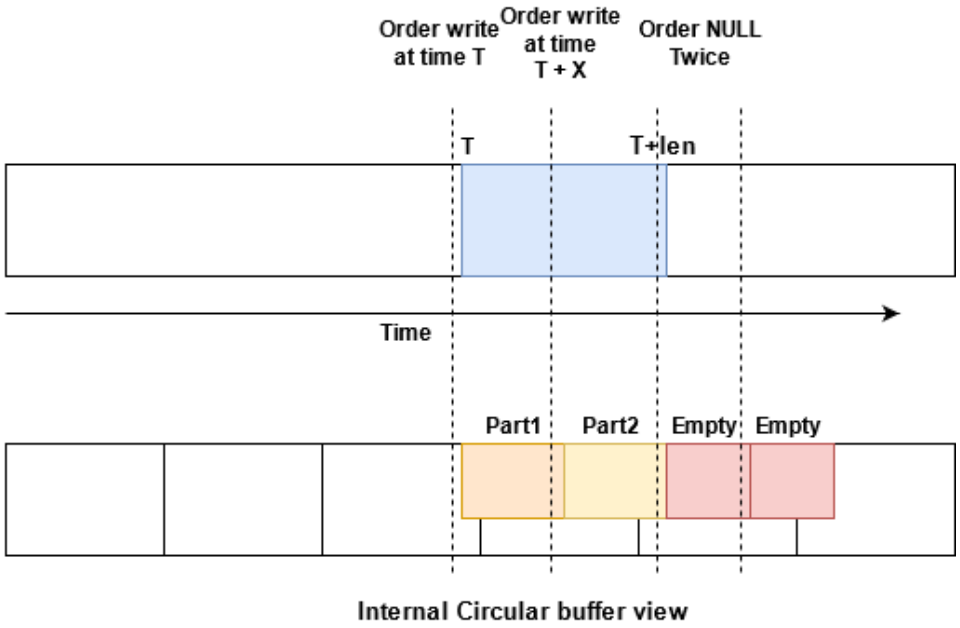

Figure 4.18: Handler concept. Ability to transmit frames of any size independent of internal buffer size

Figure 4.18 shows the concept of how the handler will solve both of the issues. The handler will need to do these operations to interact correctly with the internal buffer of the SDR, order write operations at the correct time, and fill/empty the internal buffer with the samples that should be put in.

The offset present between the dotted line and where the packet starts is the msdr_write having to be called earlier than it is meant to be executed.

4.10 Frequency shifting

Due to the slow nature of the SDR to change frequency channel, a different solution is chosen.

Through the use of Frequency shifting, received and transmitted signals may be shifted. This can be done towards or away from the center frequency, depending on the use case. The project intends to work with a 40 MHz sampling rate that leaves 20 Mhz bandwidth in each direction from the center frequency that may be used.
If the center frequency is at 2420MHz, then the range of frequency is 2400-2440 MHz.

4.11 System Architecture

In order to make the PoC run as intended, an architecture must be used such that different operations can occur concurrently. This architecture effectively means a threaded system is of interest where different operations are performed simultaneously. This functionality will minimize the workload as a serial solution would too inefficient. The key concept is to divide the operations into independent threads to maximize the parallel tasks. As mentioned earlier in this chapter, the basic operations of the PoC consist of the SDR functionalities. These functionalities involve receiving samples from the RX thread and transmitting an ACK or NACK once a BLE frame has been fully demodulated. These work in synergy with the higher layer's operations demodulating the received samples, generating the ACK/NACK, and modulating it before transmitting. Based on these tasks, one thread is created and assigned for each operation, with minimal interaction with the other threads. Two threads are assigned for interacting with the SDR. The **RX_Thread** is responsible for receiving samples from the antenna, while the **TX_Thread** is responsible for feeding samples to the antenna for transmission. For the higher layer operation, one thread is demodulating the samples received from the RX_Thread, which is called **Demod_Thread**. The last thread, the **Mod_Thread**, is responsible for generating and modulating the ACKS and NACKS before forwarding them to the TX_Thread. A more detailed explanation of the functionalities of each thread and their interactions will be now presented. The complete architecture of the system is presented in figure 4.19.

4.11.1 RX_Thread

The RX_Thread interacts with the SDR antenna by calling the msdr_read() function continuously. The samples returned are saved on a shared ring buffer with the Demod_Thread.

4.11.2 Demod_Thread

The Demod_Thread reads samples from the shared ring buffer. Once the size of the samples reaches a certain length, the thread begins looking for an BLE Access Address in those samples. If such a bitstream is detected, it means that there is a packet following the Access Address, and the demodulation procedure begins. Once the header of the said packet has been demodulated, the Mod_Thread is signaled to begin generating an ACK. At the same time, the Demod_Thread keeps running to demodulate the payload in order to verify the CRC.

4.11.3 Mod_Thread

The moment the Mod_Thread awakens, it begins generating and modulating an ACK and a NACK. Once the generation is complete, it proceeds to the modulation of these packets. Finally, once the Demod_Thread validates the CRC, it saves the modulated samples of the ACK or NACK to the tx_buff accordingly. Afterward, it signals the TX_Thread that a transmission should be made.

4.11.4 TX_Thread

The TX_Thread is responsible for transmitting the samples saved on the tx_buff using the SDR's function msdr_write(). Before each call, the tx_handler is being called to empty the tx_buff from previous samples and align the new samples with SDR's internal buffers.

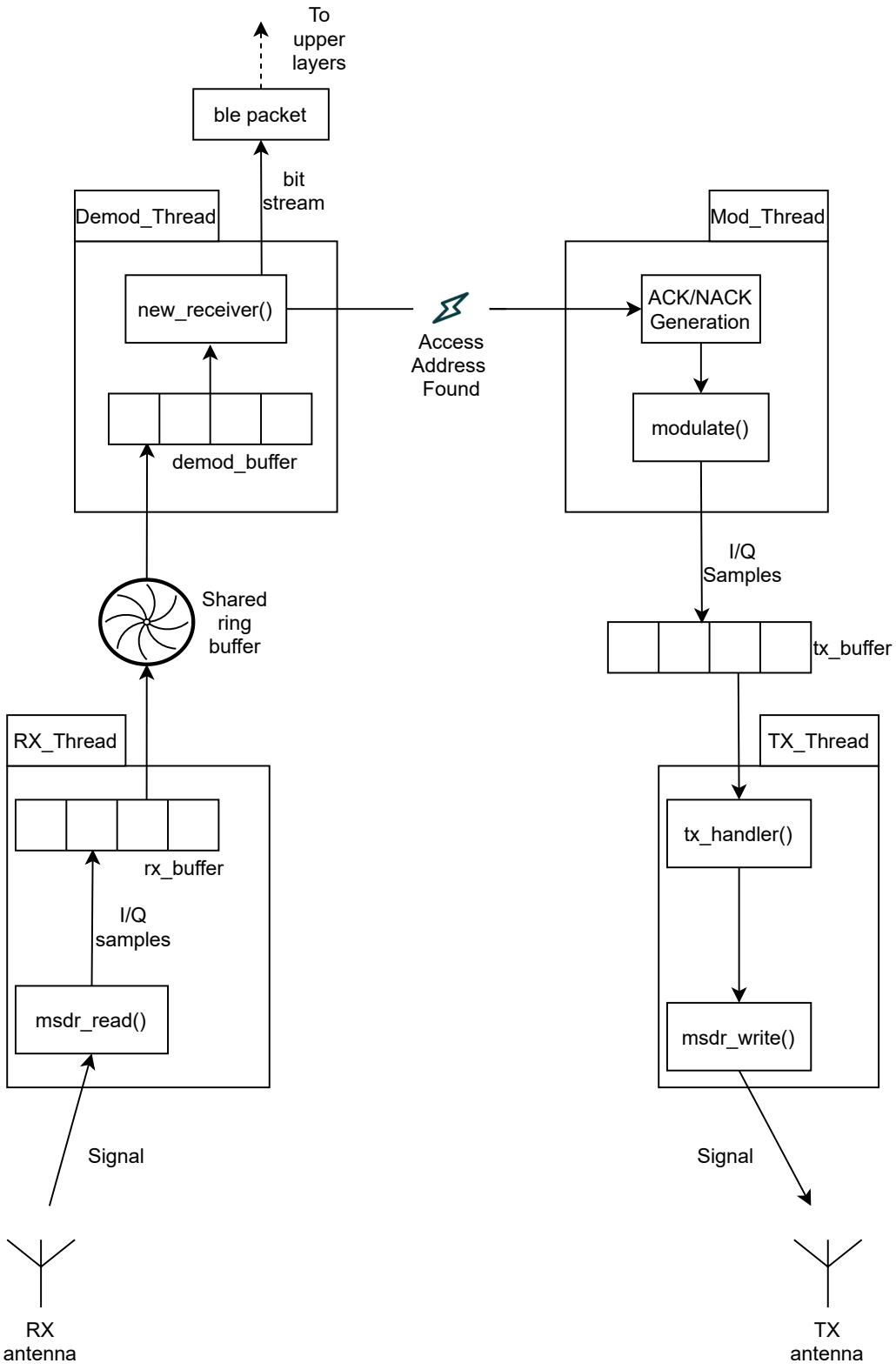

Figure 4.19: Threaded System Architecture

4.12 Proof of Concept

The PoC is a combination of the previously mentioned modules, the *frame decapsulator* and *demodulator* combined into the block called "demod". The *frame generator* and *modulator* are also combined and refered to as "modulate". Finally, the SDR is to handle transmission and reception of these I/Q samples. This block is split up as "Read" and "Write." The two other aspects that are *Link Layer Manager* and *Higher Level Interfacing* are not considered at this point. Both are assumed to be negligible in execution time.

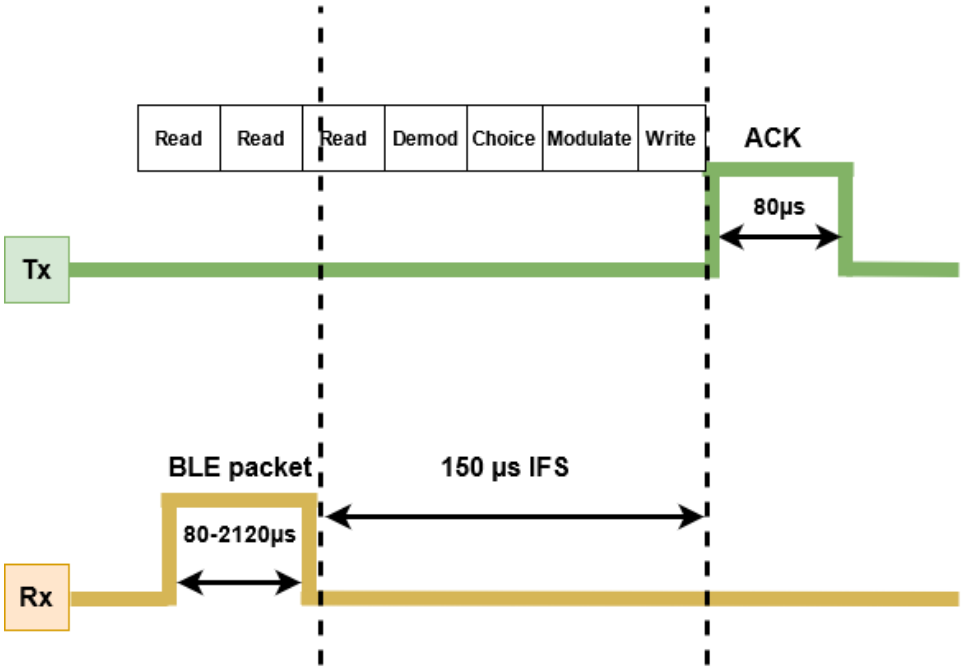

Figure 4.20: Default view of timebudget. Must fit demodulation, modulation and transmission within a 150 μs window.

In figure 4.20 it is illustrated that the operations have to be performed within 150 μs for the IFS to be upheld. Within this span of 150 μs, the SDR must read and send I/Q samples. An answer must also be chosen in between the demodulation and modulation based on whether the packet was correctly received. Thus, if an ACK should be generated transmitted or not.

This structure lays the foundation of the time budget available for this project. The ultimate goal is to do all these operations within 150μs.

4.12.1 Serial view

Starting with the previous figure 4.20 is now presented with the execution time of each function. Each of the operations is in its original state before any additional changes are made to them. For the SDR functions, this does not apply as the best performance configuration that was found is used instead. It is also assumed that choosing to create ACK or NACK is instantaneous and is thus not considered.

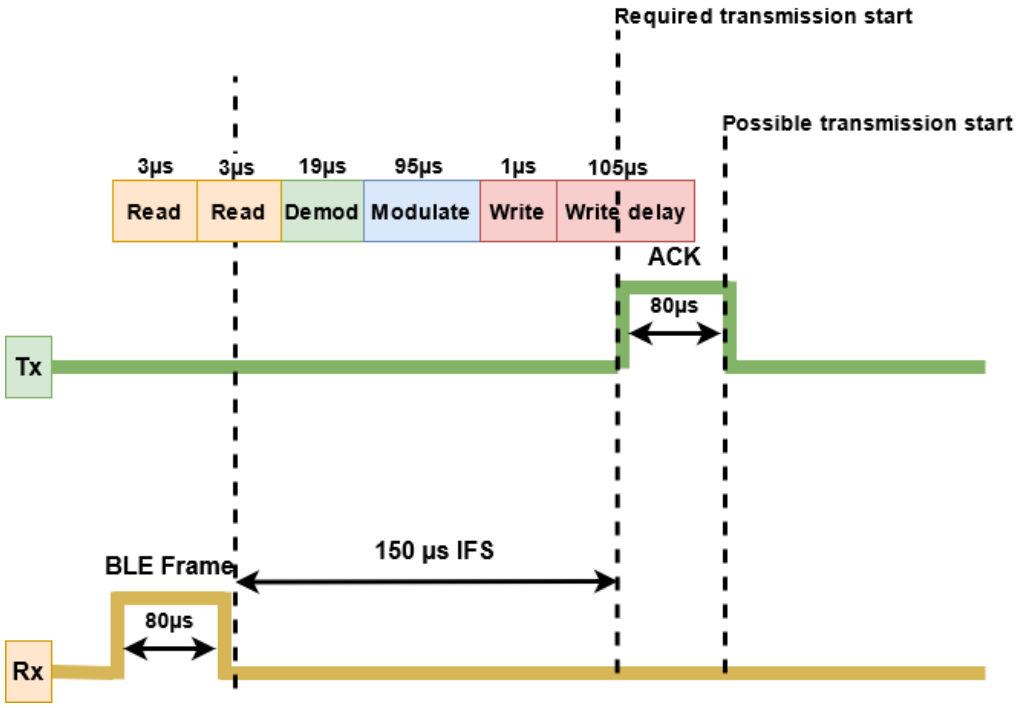

Figure 4.21: Serial perspective of function execution.

The numbers on figure 4.21 are the time required for executing the operations as of now. These numbers show that there are changes required because executing the combined operations is too slow for the IFS to be upheld. With that being said, some issues stand out more than others. The execution time goes beyond the IFS when combining the serial execution time of modulation and the write delay .
A secondary point of interest is that with the current implementation of the BTLE library, the total number of samples is required before the frame can be demodulated. Hence the 19 μs is a bit of an understatement in reality.

The IFS is close to being reached with the configuration of the SDR assuming the modulation and demodulation can be sped up and optimized. Assuming demodulation starts when the entire frame is present, the current implementation can at the earliest transmit 220-223 μs after the end of the received frame. The range is dependent on how the last read operation lines up with the last bit of frame. That is to say, if the beginning of the read operation contains the last samples required for demodulation, then it will take 223μs. On the other hand, if the last read operation has the last required samples at the end, the delay is 220 μs.

With the last point made, it should be emphasized that this is assuming that the read operation has a consistent execution time. Unfortunately as described in section 2.3.5 inconsistencies also occur. This behavior means the last read operation could take up to 40 μs. It may even take up to 260 μs before the write can be made in a worst-case scenario. Therefore barely reducing the computations down to IFS may not be enough, and bringing it lower is in the best of interest to handle this inconsistent behavior.

4.12.2 Parallel functionalities

The prior illustration had the view of a serial system, which allowed for a more clear perspective of the actions to be performed. In section 4.11 it is also suggested that a serial program is not of interest. Therefore, a perspective of the entire system from different threads is provided to more clearly see how modules are called according to each other.

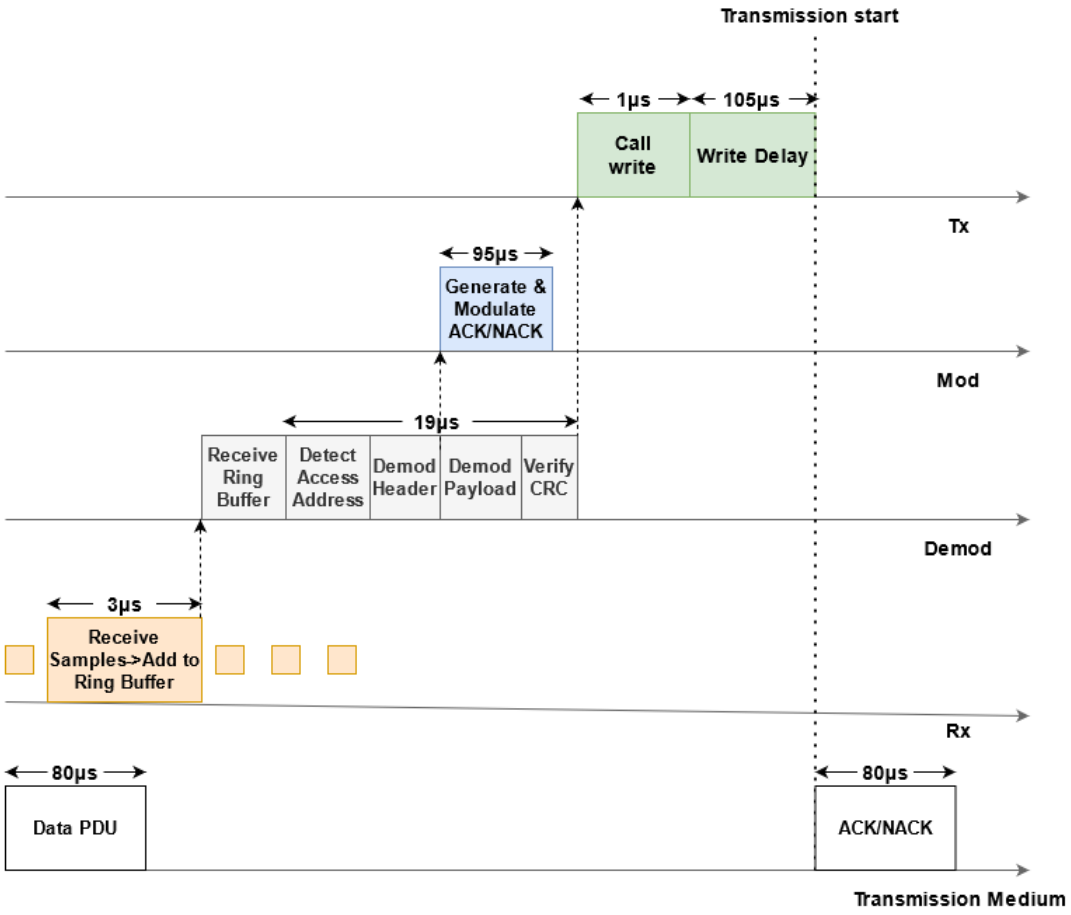

Figure 4.22: Threaded perspective of function execution.

On figure 4.22 is an illustration of where different operations will occur and how long. Due to the threaded nature of the system, where the modulation may occur parallel to demodulation, the transmission of the response at the IFS is reduced slightly by starting modulation earlier than previously.

The main obstacle to be overcome is the reduction of the generation and modulation, as this poses the largest time consumption.

4.12.3 Increased IFS design

Currently, time is essential for this project, and more time must be freed where possible. Assuming the continuous demodulation, previously mentioned in 4.8.3 is possible, a trick might be applied. It can be utilized that the preamble and access address are the same for a connection. This feature applies to all packets between a Master and Slave as well. It means that the deviation will first occur at the header, depending on the packet type sent.
Ultimately this means that everything up until that point could be transmitted beforehand. Upon knowledge of the exact response, the remaining bits + CRC can be transmitted as a continuation of the previous bits. This functionality will mainly be of interest for ACK and NACK packets. This effectively means that the IFS has another 40 μs added to it. During this time, the exact choice of response can be chosen. It does, however, require knowing that a response must be made at a specific point in time. It should be emphasized that this requires demodulation while still receiving the frame as described in section 4.8.3.

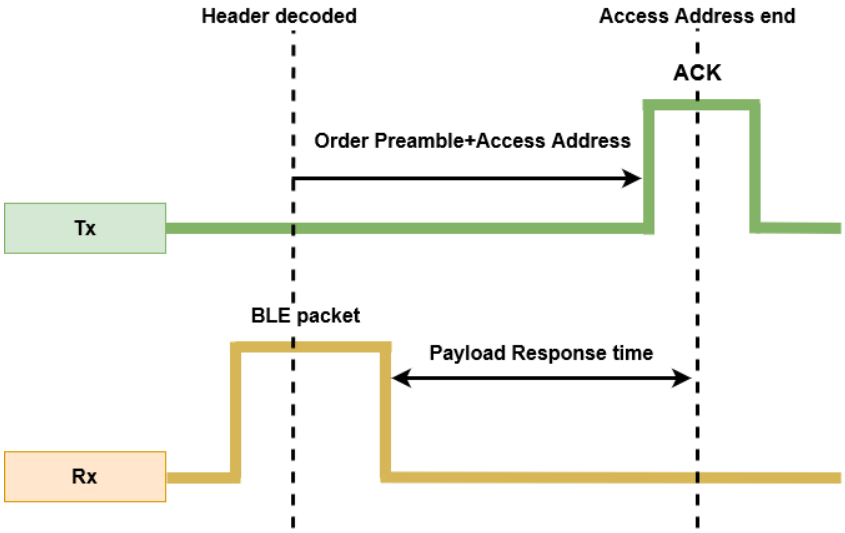

Figure 4.23: Conceptual design of sending preamble and Access Address when the header is decoded. The rest of the frame will be transmitted at a later time.

As seen in figure 4.23 the time to transmit the bits of importance, for ACK or NACK, can be seen. In principle, this trick has made the window to choose whether it is an ACK or a NACK larger than previously, effectively increasing the IFS from 150 to 190 μs.
It should be noted that to increase the speed further if needed, the preamble and access address may be modulated in advance to reduce part of the load even further.

An additional option made possible by demodulation while still receiving the frame is that modulation may occur while still receiving the frame. Combining the functionality from the threaded perspective means that the responses may be called when the header is demodulated. The specific response to send can be chosen after the CRC has been calculated, but the creation of responses, ACK or NACK, can be begun early.

5 Implementation

From the previous chapter there are different things that require implementation to create the PoC described in section 4.12. The procedure of how the implementation will be presented is that individual modules will be defined and described when standing on their own. These will then be evaluated on their performance in this initial state.
After having finalized the implementation of all modules they are to be combined which will result in the final state of the PoC.

5.1 Bluetooth Low Energy TX implementation

To achieve the intended functionality of the PoC, the BTLE library had to be operational while using the SDR. Originally BTLE TX and BTLE RX libraries are separate executables. It is desired to integrate both of their functionalities to the SDR code. The integration will ultimately result in the SDR receiving samples, the BTLE RX library demodulating them, the BTLE TX library modulating the answer, and the SDR transmitting this response.

Therefore a solution that incorporates all of these functionalities is desired, and the following section will delve into how the BTLE library is adapted and made run-able in the SDR example code.

5.1.1 Adapting BTLE-Tx initial implementation

The original configuration had to be changed to adapt the Tx part to the PoC. Initially, the Tx part of the code was configured to save the I/Q samples to a file. Afterward, the SDR example code could be run, where the I/Q samples were loaded into a buffer for transmission. In the original version generating the samples of a frame was slow and took on average 95 μs. If communication with dynamic frame generation and sample generation had to occur on demand, then the sample generation is far too slow. One way of reducing the sample generation time is by not doing file writing and keeping it in memory.

In order to call the Tx function in the SDR code, the main function of the original BTLE Tx is transformed into a function of its own that can be called from inside the SDR code. By performing this change, samples are not written to a file. They are instead written to a buffer in the memory.

Code snippet 5: Adapted BTLE Tx main function declaration

```
1 generator(int argc, char argv, IQTYPE* passaround_buf, int* sample_count)
```

Codesnippet 5 presents how the declaration is defined after performing the restructuring. This function will instead be called with three inputs to this generator function. The first would be three constantly due to the "r1" parameter seen in code-snippet 2. The second input is the command-line arguments for the original BTLE executable previously seen in, e.g., code-snippet 2. Meaning the first two inputs is, in principle, the string initially

passed as an argument from the terminal.
The third input is a pointer to a buffer for returning I/Q samples. The last input is a way to extract how many samples were modulated.

The greatest difference between this and the original code is the addition of this passaround_buf and sample_count. A better approach is chosen as it is slow to write to a file and read from it. Said approach was the passaround_buf which is used to save I/Q samples in memory as the generator is simply a function being called as part of the SDR code.

Code snippet 6: Time measurement of execution time

```
1 Gettimeofday(&st);
2 generator(int argc, char argv, IQTYPE* passaround_buf, int* sample_count);
3 Gettimeofday(&et);
4 int elapsed = (et.sec - st.sec) * 1000000 + (et.usec - st.usec);
```

As a result of adapting the BTLE Tx library inside the SDR code through the generator function, the execution time has been optimized. Using the ACK packet as a reference point, it only takes 9 μs to modulate the packet. This number is produced using the same packet type used to evaluate the original BTLE version, which took 95 μs to create and modulate 4.8.1.

Thus with this adaption, the BTLE Tx library is callable while running the SDR. To change the frame parameters is assumed to be almost instantaneous and is thus not considered much further. A thing worth mentioning is that larger frames will require a longer period to be produced, as part of the load is bit count dependent.

5.1.2 New Filter

The BTLE library originally provided filters for sampling rates of 4MHz, 6MHz, 8MHz, 10MHz. The current implementation requires 40MHz in order to function correctly. Therefore a Gaussian FIR filter fit for 40 samples per symbol must be made. This filter is required due to the performance increase seen at higher sampling rates found in section 2.3.4.

A new Gaussian FIR filter for 40 samples per symbol may be created using equation 1 found in Appendix. This filter and more can be found in Appendix table 19. Given the implementation in the BTLE library, a few changes are made to adapt the filter to the code.

After adding the new filter, the sample generation turns out to be very slow. This observation leads to the following small test to investigate how the execution time was related to the samples per symbol used. The test is performed for Sample per Symbol at 4, 8, 16, 32, and 40.

Sample per symbol	Filter length	Average Execution(μs)	Execution range(μs)
4	9	9	$7 \leq x \leq 12$
8	18	16	$14 \leq x \leq 21$
16	31	38	$36 \leq x \leq 42$
32	59	102	$96 \leq x \leq 123$
40	73	170	$159 \leq x \leq 180$

Table 7: Frame generation and modulation time dependency on Sample per Symbol and filter length.

Time-wise, the new filter was much slower to run through, as seen in table 7, where a higher sampling rate leads to a slower execution. The most significant bottleneck of the PoC is caused by the SDR's driver delay. The solution to minimizing the driver delay appeared to be increasing the sampling rate while reducing the DMA buffer length. This change means that the 40MHz sampling rate is almost non-negotiable, meaning 40 samples per symbol is required. At the moment, to generate and modulate an ACK packet, it takes 170 μs, which is too slow if frames are to be generated in time for transmission and uphold the IFS.
It is worth noting that 32 MHz could also be used regarding driver delay, but it leaves much less extra time before the deadline, and the modulation time for this is still considerably large.

The only part of the BTLE TX library dependent on the Sample Per Symbol rate is the modulation. This dependency means that this function will require an alternative method. This method must perform modulation that is time-wise optimized and scales better with higher sampling rates than the original.

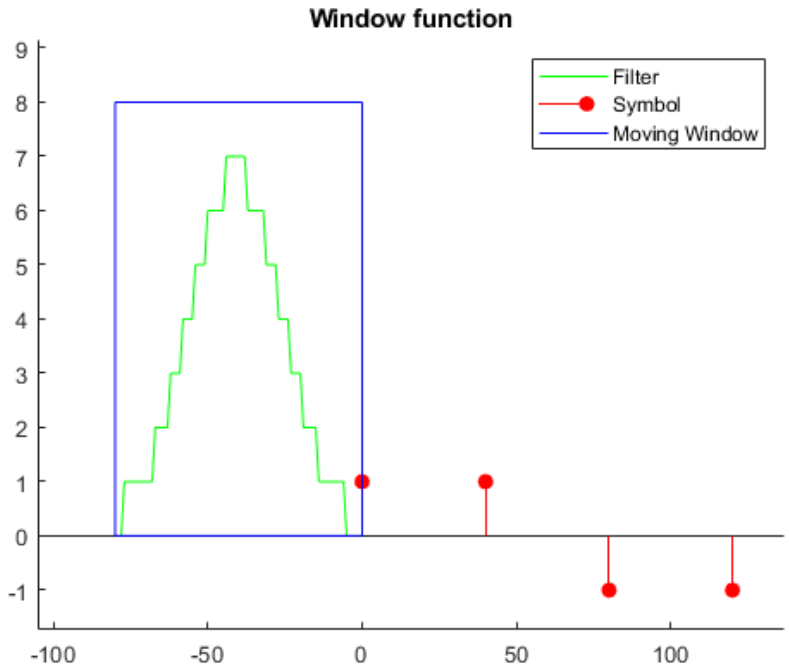

Figure 5.1: Concept behind the windowing process that generates samples from symbols and a gaussian FIR filter.

Originally the method of generating the I/Q samples involves using a window function that moves the filter across all symbols like shown in figure 5.1. The implementation iterates the number of samples multiplied by the length of the filter (73 at 40 Samples per Symbol). This leads to a high amount of time used iterating through these samples before finding the exact degree that a sample has. From that point, the I/Q samples associated with this degree may be found by calculating its cosine and sinus result, which is also shown further in code snippet 11.

Code snippet 7: Original windowing function implementation

```
for(i = 0; i < sample_count;i++){
    acc = 0;
    for(j=0;j < filter_length; j++){
        acc = acc + filter_coef[j] * symbol[i+j]; // Degree on the associated I/Q samples
    }
    tmp = (tmp+acc)&1023; //tmp grows large and is thus limited to 1023 values, or 360 degrees
    sample[i*2] = cos[tmp]; //Real part, Sampled cosine values in appropriate indexes
    sample[i*2+1] = sin[tmp];//imaginary part, Sampled sinus values in appropriate indexes
}
```

In code-snippet 11 is the original version of the windowing function, which moves across all the samples and finds the appropriate I/Q component for the samples given the filter and symbols.

This for-loop will iterate longer at higher sampling rates due to two facts, the samples per symbol and filter length, which was also seen on table 7. A higher sampling rate means the amount of symbols turn into more samples. A higher sampling rate also leads to

more filter coefficients, as the FIR filter coefficients decrease slower. Hence the sampling rate impacts two aspects that both increase the execution time.

One thing of interest in this piece of code is the **acc** variable, as that is an accumulation of all intermediary values that contribute to the value of each one sample. After accumulating everything that adds to one sample's angle, its cosine and sinus value are found and used as the I/Q samples.

A different method than the initial can be utilized by precomputing the accumulation of values that contribute to each sample. With knowledge of how they overlap and how they affect each other, this can be realized. Given that the length of the filter is 73 non-zero values and that each bit, or symbol, takes up to 40 samples, it means that an individual symbol is affected by its two immediate neighboring bits only.

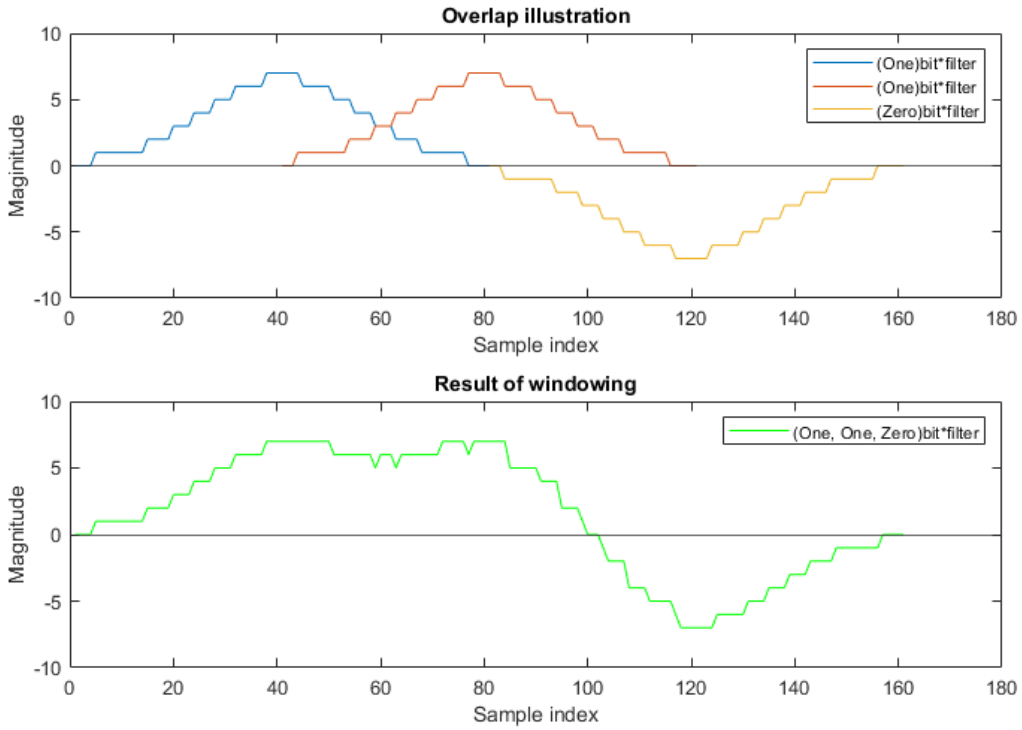

Figure 5.2: Illustration of overlap that occurs between on at most 3 bits. Only the two neighboring bits affect the samples. This can be seen from the fact that the samples associated with the blue graph are not affected by the yellow, only the red. The red, on the other hand, is affected by both the blue and yellow graphs. However, future or past bits should not influence sample indexes associated with the orange graph. For the yellow, it is only affected by the red. The bottom figure is the result of windowing over these bits all at once.

The example illustrated in figure 5.2 shows the binary sequence 1, 1, 0 being windowed with the filter previously mentioned and shown in figure 5.1. The first part shows how these different bits are windowed individually and how these indexes are overlapped and affected by previous and future bits. The point of interest is that a single symbol only

occupies 40 samples. This means that a single bit is associated with only 40 samples. The center of a symbol, index 40, 80, and 120, and 20 samples in each direction are required. Thus by knowing the neighboring bits of a bit being modulated, one can find a precomputed amount of 40 samples to associate with said bit pattern. This is what can be seen in the bottom figure of figure 5.2.

This can be expanded to encompass all the possible combinations meaning 2^3. However, that only applies to the middle bit. The beginning and end look differently as they have only one neighboring bit, which means the total amount of unique patterns is $2^3 + 2 * 2^2 = 16$. By precomputing these 16 unique patterns, no matter what binary sequence occurs, the pattern they result in is ready at hand to be used. This is a trade-off between time and memory as more memory is occupied with these patterns. Memory is fortunately not a concern on the device used, and as time is of the essence, this trade-off is very worthwhile.

This different approach means that on runtime, the amount of iterations in the for-loop instead of samples*filter_length becomes equal to the number of samples. This reduction is due to the implementation since the **acc** variable in code snippet 11 is the one precomputed and thus, the **tmp** variable is still being computed during each iteration. The difference in time caused by this change is that the modulation time on its own has been reduced from the 170 μs average on table 7 to an average of $14\mu s$ which is a significant reduction and should be sufficient given that modulation occurs during demodulation. Thus further optimization in modulation speed will not necessarily have any other effect.

5.2 Bluetooth Low Energy RX implementation

As mentioned in the design section 4.8.2, The Bluetooth reception needs to be changed so that it can demodulate over multiple calls of the function and thus lose its reliance on large buffers. This change means that the new function will have to keep track of the demodulation process as it runs as samples come in small buffers. As this function is altered in terms of functionality, it is also important to measure the speed of different processes so that the new function also is fast enough to adhere to the time requirements imposed on the PoC.

5.2.1 Bit by bit demodulation

The main thing to change in the demodulation process is to make it work over multiple reception buffers. This means that the function should continuously update how far it has come in the demodulation process and what elements are needed to finish the current frame. The overall logic of the new demodulation method is illustrated in figure 5.3.

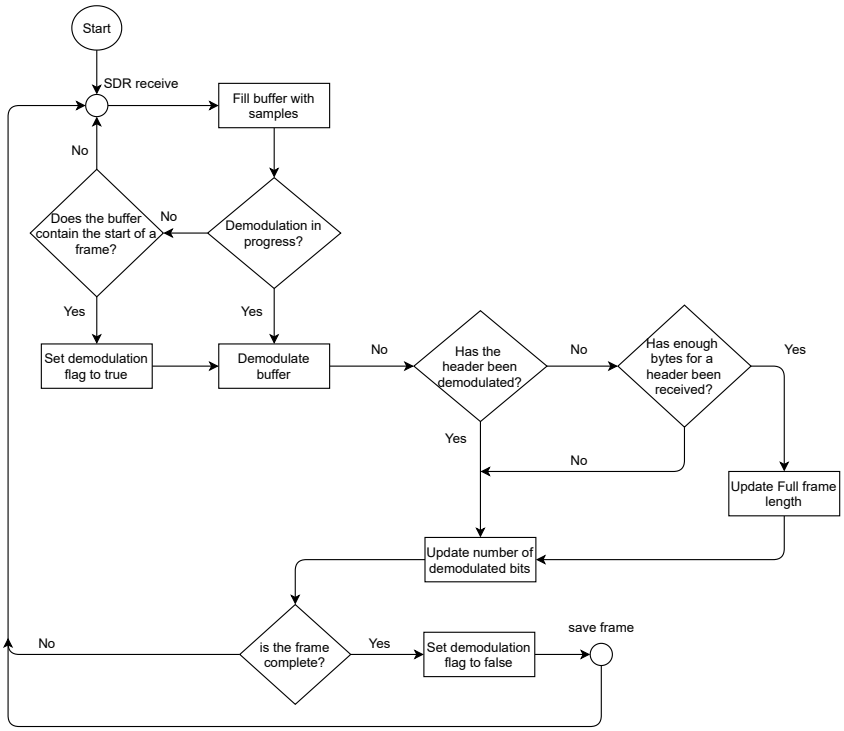

Figure 5.3: A flowchart showcasing the logic of the bit by bit demodulation

An important thing to note is that the flowchart does not have an endpoint. This is because this function has been designed to run constantly while the SDR is running. The function thus turns on with the device and runs in perpetuity. The function gets samples from the *msdr_read* function and saves them in a buffer. It then checks whether or not it is currently working on a frame. If it is not, the function will be looking for the beginning of a frame. It is specifically looking for the access address of a frame via the search_unique_bits function. When the access address appears in the buffer, the function will state that it is now in the process of demodulating, and from there, it will demodulate samples as they arrive. While the function is demodulating, it will keep checking how many bits have been demodulated. This is to react whenever essential information is present. When enough of the frame has been demodulated, meaning when the header has arrived, the function will read the header to find out how long the whole packet will be. When it knows that, it will keep demodulating until the whole length of the frame has been demodulated, and it will then parse the frame. Finally, it will reset all the variables and buffers used in the process and start looking for the beginning of the next incoming frame.

The bit by bit demodulation function has been implemented as a modified version of the original receiver function seen in section 4.8.2. It would therefore make sense to take a look at what the new version of the function, called *newReceiver()*, takes as input.

Code snippet 8: The function used to demodulate and parse frames.

```
1 int newReceiver(IQ_TYPE *rxp_in, int buf_len, int channel_number, uint32_t access_addr,
2 uint32_t crc_init, dataCnt *dta, RFPort *rfp)
```

The initial variables given to the function are the same as in the original, and their purpose has not changed. The flag variables have been removed as they are not deemed necessary and were used for prints and simplifying the demodulation procedure. The major change to the function is the addition of the struct *dataCnt *dta*. This struct carries a range of variables that are used to keep track of the demodulation process. The struct is given as a pointer so that the data can be saved somewhere else and used over multiple loops of the receiver as described with the flowchart (figure 5.3) above. The main variables in the struct are:

- *Byte_num* & *bit_num* which are integers used to keep track of how much has been demodulated of the current frame. Byte_num is the variable that is used to decide when we are done.
- *headerB* is a Boolean variable that is used to check if the header has been demodulated. If the variable is false, the header has not been demodulated yet. When the header is found, the Boolean is set to true. This is done so that the header process will not be repeated until the code looks for a new packet.
- *demoder* is also a Boolean, and its purpose is to keep track of whether or not a packet is currently being demodulated.

The struct also contains other variables, but they are less important when it comes to an understanding of how the code functions. Another struct that has been added to the function input is the *RFPort *rfp*. This variable has been added to the function as it is used to share information between different threads running in the PoC. The function itself also now is an integer function. The purpose of this change is debugging as it makes the process of getting different values out of the function to make sure that it is finding the packet correctly or if the demodulation is working as intended. The values returned are not needed for the actual functionality of the PoC.

With these alterations in the Rx part of the BTLE library, the functionality will now allow for the code to receive over multiple reception buffers. One concern that needs to be investigated is the efficiency of the implementation to see if it can live up to the requirements.

5.2.2 Rx performance

The main thing that needs to be explored with this new implementation of the receiver is whether or not it is fast enough. To that end, the main element of the reception module that is affected by altering the sampling rate will be investigated. Altering the sampling rate affects the speed of the *search_unique_bits* function in the receiver. This change in speed is because increasing the sampling rate increases the samples per symbol. This, in turn, means that the function has to go through more samples to look for the access address. The average time of going through the function at different sampling rates can

be seen below in table 8.

	4 MHz	8 MHz	16 MHz	32 MHz	40 MHz
search_unique_bits	915ns	1803ns	3402ns	6910ns	8188ns

Table 8: The average execution time on a 35 bit buffer for each function at different sampling rates. (1 million executions)

The search_unique_bits function is slow. It can be seen that the average execution time rises dramatically as the sampling rate increases. It is also important to note that three new bits are investigated each time the function is executed. This is because the function is looking for the correct sequence of bits. If the function skips too many, it will potentially miss a frame. The buffer, therefore, consists of the 32 previous bits and 3 new incoming bits to make sure that parts of the access address are not thrown out. That means that the time measurements shown in table 8 effectively is the time it took to check 3 bits. The search_unique_bits, in its current state, the search_unique is only fast enough at 4 MHz sampling rate, but as has been stated earlier in the implementation, a higher sampling rate is required.

Another function to look at is the *demod_bit* function. This function has been changed to only demodulate one bit at a time instead of an entire buffer. The average time to demodulate 1 bit with this new function was 23ns, which is in line with the time it took for the *demod_byte* in the original implementation (46ns).

The main thing to focus on from this point is making the search_unique_bits faster at a higher sampling rate. This can be done in either of two ways. One option is to look for something smaller such as the preamble. This will make the process of the search function take less time as the preamble is only 8 bits instead of 32. This solution, unfortunately, has its drawbacks. Changing to the preamble meant it was easier for noise to appear as the synchronization sequence. A threshold could be implemented to filter out noise, but another option was preferred.
The other option is to down-sample by using decimation. This will allow the function to work with incoming data, as though it is at a sampling rate of 4 MHz instead of the required 40. Practically this is done using a decimation factor of 10, meaning that each time a sample is used, 10 are skipped before choosing the next sample. In an environment with relatively little noise, decimation can be performed without needing a filter, which is usually the case. As the PoC is relatively unaffected by noise, this is the option chosen for this PoC. The implementation of the decimation is trivial as the function is simply set to skip 10 samples when searching the buffer, making the function fast enough to function in the new implementation as it now takes the aforementioned 915ns to go through the function on a sampling rate of 40 MHz.

With the receiver implemented, the next thing to do is work on the responses that need

to be given after receiving a message. To that point, the next section will go over the generation of acknowledgments as responses to received messages.

5.2.3 Response Generation

The urge of creating an optimized packet generator for responses came from the need of having a fully accessible BLE frame structure tailored to the needs of the PoC. This is because direct access to the header fields of response packets simplifies and speeds up generating the response packets compared to iterating their corresponding memory buffers and changing bits to achieve the desired values. The timings measured on the original BTLE's packet generator were around 95μs originally. The optimization and integration of BTLE into the SDR code reduced this to 14μ on average. The custom-made generator takes around 8us to generate a response, so speed-wise it is also an improvement. A more detailed explanation of its functionalities will be presented below.

To begin with, certain attributes of the response packets are static and already known, so memory pre-initializations can begin as soon as the program starts instead of waiting for the call of the packet generation. Other attributes can be learned when the received packet's header is demodulated. This means they can be pipelined in the responses frame structure during this process. For example, the size of the ACK is always 80bits, the payload field is empty, and the preamble and the access address is constant for a Master-Slave pair. Once demodulation is finished, the acknowledgment can be generated, and its values are filled in the pre-initialized memory buffer. Then, the CRC is calculated, and a scramble operation is performed on specific fields of the packet. Afterward, the response is modulated and forwarded to the SDR antenna for transmission. It cannot be sent yet, though, as the CRC check of the received packet needs to be performed. The check must be performed to decide if the packet transmitted will be an ACK or a NACK. This is a bit flip operation which costs less than 1μs in terms of timing expenses. The sequence of operations up to the point where the ACK is modulated takes 3-5μs. The transmission part introduces the SDR into the design. The modulated packet is being fed into the tx_buffer, and the tx_thread is signaled to begin transmitting the samples.

5.3 Tx Handler

From section 4.9 it was described that a function meant to simplify interaction with the SDR was required.

Due to the functionality of a ring buffer, progress must be made in the time between each msdr_write. The investigation made in section 2.3.6 showed that sleeping might cause some inconsistent behavior by the SDR. The execution of msdr_write was shown not always to be perfect. It was, however, found that calling it with no samples was fast and had the least variance in observed execution time. For that reason, msdr_write is chosen to be used to progress in time. The logic for how the Tx Handler will operate is defined in figure 5.4

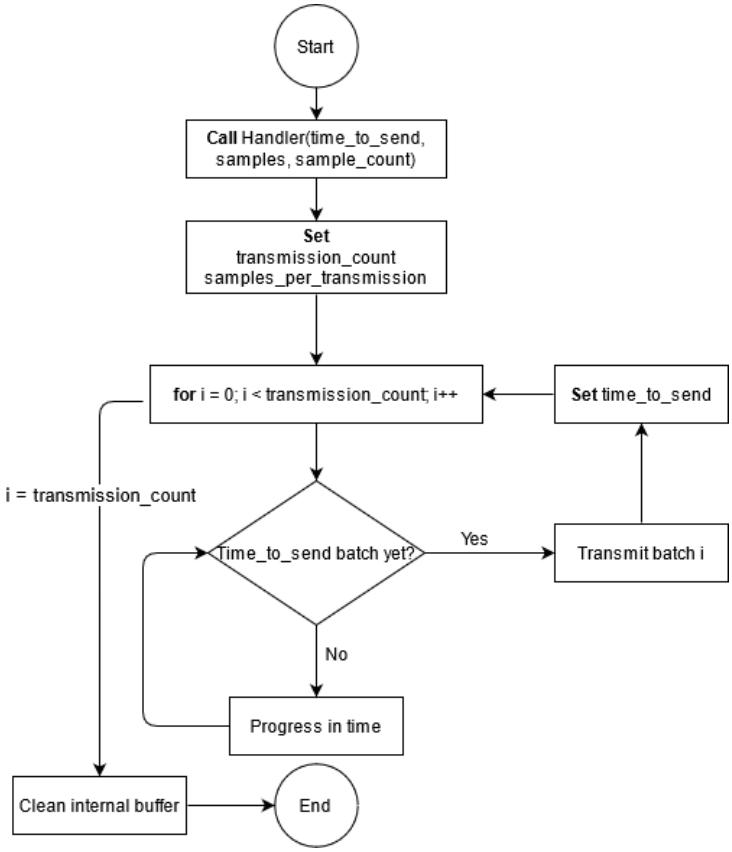

Figure 5.4: Flowchart logic of the handler concept.

Figure 5.4 shows a flowchart of the logic that the handler utilizes. The function takes the first timestamp (time_to_send) to utilize for the first batch of samples. It also takes a pointer to the samples to be transmitted and the number of samples.
Upon being called, it initializes some variables and calculates the number of times samples are to be transmitted. The important variables defined is the number of transmissions and the number of samples to be transmitted in each transmission.
After calculating over how many times the samples are to be transmitted, it will go into a while loop where it checks the current time and compares it to when it is supposed to call the write. The ordering of samples will occur a while before it actually should be executed in case of unexpected delays. That means part of the internal buffer will contain the samples, but they are first transmitted when the internal buffer reaches this point.
To progress in time, "dummy" writes are called onto the other half of the internal buffer than where the internal buffer is currently transmitting samples from.

The purpose is not to overwrite samples that are to be sent. The program will then try to transmit the current batch of samples, e.g., 45 microseconds worth of samples, then set the timestamp for the next batch of samples to be sent 45 microseconds later in time. This will continue being done until all transmissions of the samples are executed.
At this point, the cleaning of the internal buffer will take place. Cleaning is done over two runs to clean where the currently transmitting samples stop in the internal buffer. A

second run cleans the entire internal buffer after the last batch is finished transmitting. The first cleanup is executed to ensure that samples from the previous msdr_writes would not suddenly be transmitted again. After the final cleanup, the function is finalized.

On figure 5.5 is an example of the terminal output of how this looks for a large set of samples. The thing to pay attention to is the difference between ordered at and ordered for. The critical point for msdr_write to be called successfully is that the called for is larger than called at. It is worth mentioning that the timestamps shown are normalized according to when the samples are supposed to be transmitted at. This means the line "Configured Tx 1 for 0 at -43" means that the first call was done 43 μs before the transmission is supposed to start. Furthermore, the program attempts to only send 45 μs of samples at a time. This is seen by the constant difference in the "configured for" value.

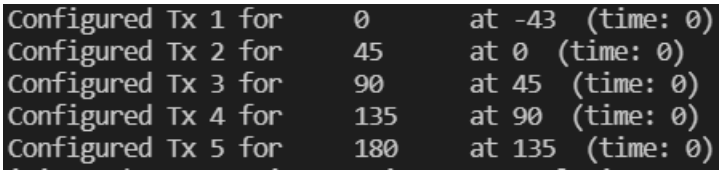

Figure 5.5: Terminal view of how many samples to be sent at split into additional msdr_write operations.

Using this function, the transmission of packets both smaller or larger than the internal buffer is possible. It also serves as a simplification of how transmissions can be called without having to think of the SDR specific operations. Thus a simplification for the usage of the SDR has been made.

5.4 Frequency shifting

Since using the SDR functionality to change frequency for transmission is not fast enough since it takes 100 microseconds, a different approach is chosen. Because a high sampling rate is used for other purposes, this adds other advantages for dealing with frequency shifting.

By performing I/Q sample frequency shifting, it is possible to transmit on different frequencies than the one the SDR is tuned in on. The exact scope is 20 MHz in each direction which means 19 out of 40 channels of BLE can be covered with one physical SDR unit. It is not 20 channels because the entirety of the outermost channel cannot be fully sampled.

The function to perform the I/Q shift was at hand at Keysight technologies, and so as not to reinvent the wheel, this function was borrowed to change the frequency quickly.

One obstacle was found as executing the function in its initial state. The received samples take too long to be frequency shifted. It takes 5 μs to I/Q shift 3 μs worth of samples

upon reception. This execution time means that the computation time would need to be decreased to use this functionality actively.

Code snippet 9: Frequency shifting function

```
t = i/sample_rate;
re = sample[0][i].re * cos( t * Fshift * 2 * PI)
   - sample[0][i].im * sin( t * Fshift * 2 * PI );

im = sample[0][i].im * cos( t * Fshift * 2 * PI)
   + sample[0][i].re * sin( t * Fshift * 2 * PI );
sample[0][i].re = re;
sample[0][i].im = im;
```

code snippet 9 is the required calculation to shift the samples to a specific frequency. **sample_rate** is how many samples are collected a second, in this case, configured to 40MHz. **Fshift** is how many frequencies the signal should be shifted. It can be shifted half the sample rate in either positive or negative direction of the center frequency. **t** is a derived intermediary value that changes the output of the cosine and sinus values between each iteration. The intermediary values **re** and **im** are required since if one were to change the real part of the sample, then the imaginary value would be using an incorrect value when it executed its shifting.

A simple solution is used to decrease execution time by using a lookup table for the cosine and sinus functions. This is because these functions consume a lot of execution time, and a lookup table is an easy solution to reduce this. As it would also turn out, the amount of unique values in each table are only 20 at most. The addition of further values would simply be repeating the pattern.

This lookup table is made for all possible frequency shifts that may be used. That results in 19 lookup tables for both sinus and cosine values. The change in execution time is significantly reduced to 300 nanoseconds instead of the 5 μs, which means samples can be collected and shifted without losing any samples due to discontinuities.

A thing to note in this regard is that the 300 nanoseconds are only for 3 μs of samples. Frequency shifting more bits worth of samples would therefore also take longer.

5.5 Initiator Role

This section provides the implementation and optimization for the functionalities of the Initiating State of the Link Layer, integrated in the PoC. Figure 5.6 shows the flow diagram of the implementation.
By default, the Link Layer will begin in the Initiating State, listening to an advertising channel and looking for specific Access Addresses to establish a connection. In the Initiating State, the Link Layer looks only for ADV packets. These packets (ADV_IND, ADV_DIRECT_IND) are signaling that the advertising device wishes to establish a connection. This first step is implemented by inspecting the received packet after demodulating it. The pdu_type field of the Header is used to define the type of the received packet. If the packet is neither of ADV_IND or ADV_DIRECT_IND pdu_type, the

packet is ignored.
Upon receival of one of these Advertisement packets, the device address is extracted. It is used for the generation of the CON_REQ packet. It is important to mention that the number and the position of the bits that define the AdvA inside the Header are always the same for every ADV_IND and ADV_DIRECT_IND packets. This makes the operation fast and straightforward. Once this value has been extracted, the Link Layer proceeds to construct a CON_REQ packet, which uses the AdvA as its destination identifier. This step links the Initiating State with the Connection State.

Once the CON_REQ is generated and modulated, the SDR transmits it to the destination device. If the transmission happens inside the IFS, the two devices are considered connected, where the SDR becomes the Master, and the advertiser becomes the Slave.

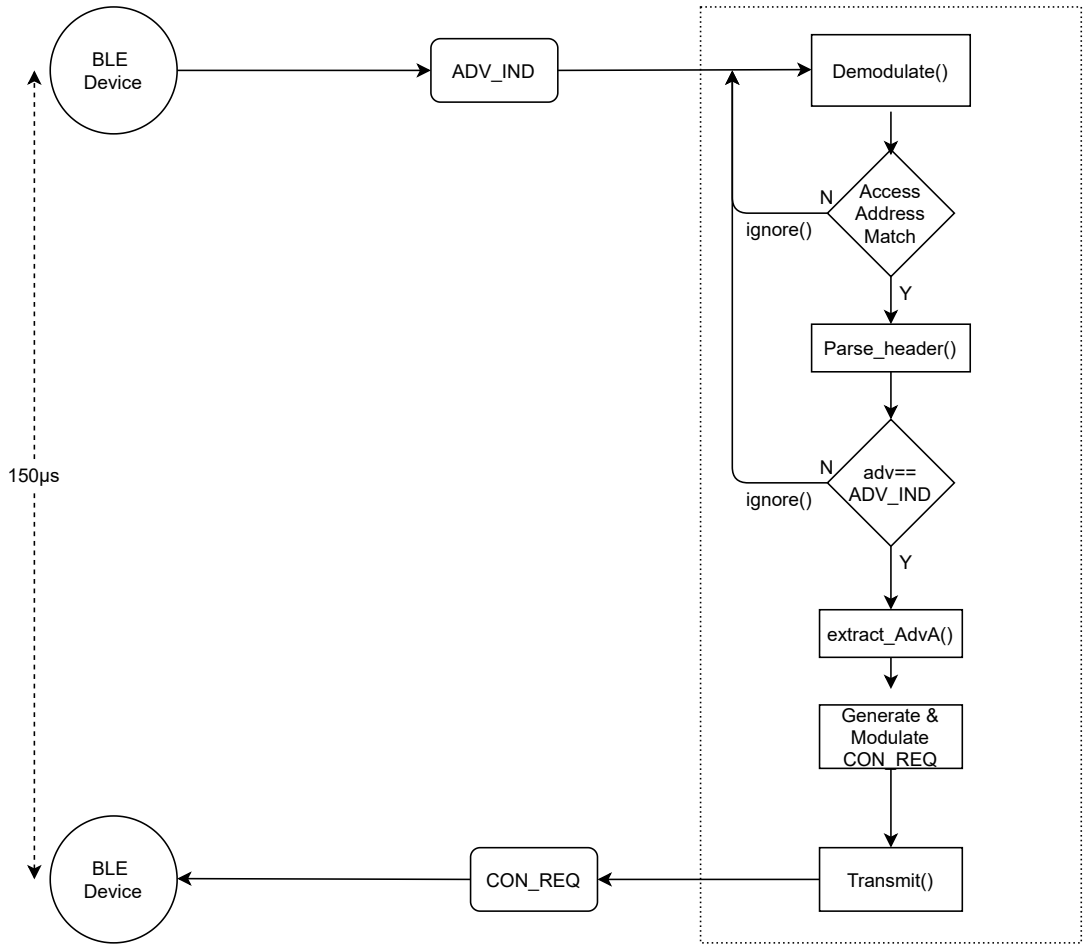

Figure 5.6: Flow Diagram for the implementation of the Initiating State. Same operations apply for ADV_DIRECT_IND packets

5.5.1 Initiator Optimization

A simple optimization technique has been applied to the above implementation. The CON_REQ packet does not need to be generated every time an ADV packet is detected. Instead, it can be generated once at the beginning of the program, with the desired parameters for the connections, and get loaded into the memory of the system only once. This is allowed because the only field of the CON_REQ that changes for a new connection is the advertiser's device address, which is known beforehand as it has to be white-listed. All the other fields can and shall remain the same. This method saves up a small but not negligible portion of the time inside the IFS. With this method, for every CON_REQ, 6 bytes are being accessed and written over in an already existing buffer inside the memory. Without this optimization, for every CON_REQ, 44 new bytes had to be allocated and initialized inside the IFS.

5.6 Threaded system architecture

In order to create a full implementation of the proof of concept that can receive, demodulate, modulate and transmit, the different modules previously explored and implemented have to be combined to a final product. The primary functionality is the ability to receive a packet and send off a response that arrives at the IFS interval.

5.6.1 RX thread

The first thread to be implemented is the Rx thread, as it lays the foundation for the entire receive and transmit concept in combination with demodulation.

The Rx thread is implemented in two phases. An initial phase where variables are initialized and a second phase running constantly and putting received samples into a buffer.

Code snippet 10: Rx thread content

```
#define DEFAULT_BUFFER 120// 3microseconds of samples at 40MHz
#define SHARED_BUF_LENGTH 2000*DEFAULT_BUFFER //Total length of the shared buffer

void *rx_thread(RFPort *rfp)//a pointer containing relevant cross thread parameters.
{
    init...
    while(keep_running)
    {
        clock_gettime(MONOTONIC, &start);
        ret = msdr_read(rfp->sdr_state, &timestamp,
            (void **) rfp->rx_buf, rfp->buf_len, rfp->port_index, 5000);
        clock_gettime(MONOTONIC, &end);

        exec_time = (end.tv_sec - start.tv_sec) * 1000000 + (end.tv_nsec - start.tv_nsec) / 1000;
        //Microsecond resolution

        nanosleep(0); //Found to reduce msdr_read execution time.
        if (exec_time > 50){
            exit(0);
        }

        shared_buf_index = (rfp->batchnumber*(DEFAULT_BUFFER)) % SHARED_BUF_LENGTH;
        memcpy(&rfp->sharedbuf[0][shared_buf_index],     &rfp->rx_buf[0][0], sizeof(Complex) * ret);
        rfp->batchnumber = rfp->batchnumber + 1; //Used to signal to demod thread
                                                 //that new samples are available


    }
}
```

On code-snippet 11 is a general version of what happens in the second phase of the Rx thread. In each run, msdr_read is called, and the execution time is checked. This is caused by the finding in section 2.3.5. The time is validated to not be above the threshold. if not, the received I/Q samples are put into a buffer that is shared with all threads.

This is because the Rx thread's sole purpose is to call msdr_read at all times. This ensures that discontinuities do not occur, which might be the case if samples are lost due to not reading continually. A secondary effect is that the shared buffer only functions as a ring buffer, meaning discontinuities may also occur in this buffer. This will naturally also require monitoring. The cause of them being that the demodulation thread is not demodulating fast enough.

5.6.2 Real-Time demodulation thread

Originally the demodulation code required a large number of samples wherein the frame's entirety must be contained. Afterward, with a single call of the demodulation code, the entire frame was demodulated.
This was altered to sampling a large number of samples, which then, the demodulation function could be called multiple times to demodulate. The change was done to prove

that the partial demodulation functioned correctly. Thus the first step is to remove the sampling for a period of time. This is done to demodulate as fast as possible. By splitting reception and demodulation into two threads it means real-time partial demodulation is achieved.

This leads to the initial step of the implementation seen in figure 5.7, where a Demodulation thread and Rx thread are defined along with their function calls and interaction between each other.

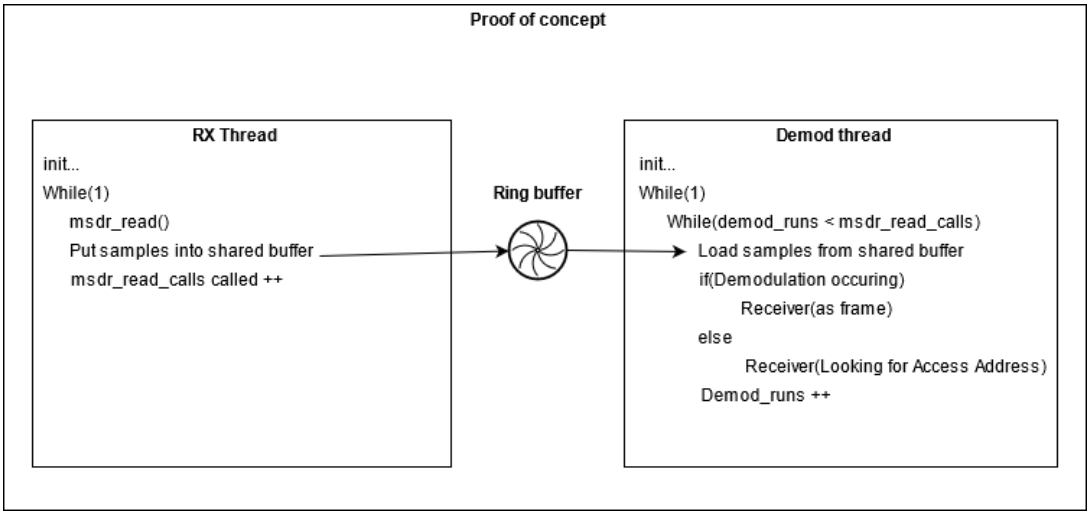

Figure 5.7: Functionalities in the two threads and their interaction through the ring buffer.

By implementing the Demodulation thread as shown in the figure 5.7, it is possible to demodulate as samples are received. The Rx thread is constantly running to ensure no discontinuities appear by losing samples due to a lack of msdr_read calls. The demod thread then runs whenever new samples are present in the shared buffer. With that being said, additional changes also had to be made in the Demodulation thread. This was related to how it operates and uses samples to demodulate the frames.
This is caused by the access address synchronization, which produces an offset that needs to be handled. If the offset is allowed to exist, errors will occur in demodulation. This offset is negative and will range from 0 to samples_per_symbol.
This means nothing when the buffer used is large, as when proving partial demodulation works, this caused no problems. The issue is much more apparent when demodulating a set of bits as they are received with msdr_read. In order to better showcase the cause of this issue figure, 5.8 illustrates the issue.

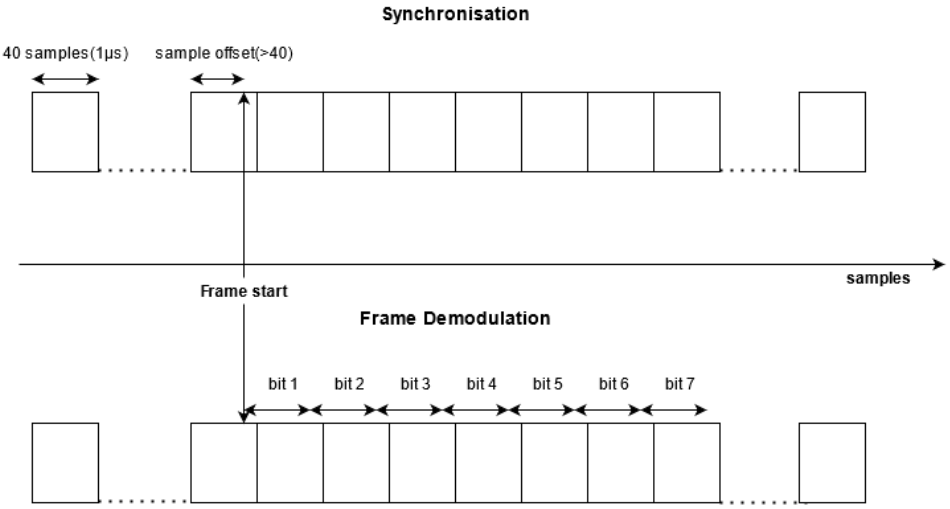

Figure 5.8: Offset induced when a frame is found. The beginning of a bit, or symbol period is offset compared to the samples received.

Illustrated in figure 5.8 is an offset produced as a result after the synchronization with the frame. This offset defines where the beginning of a symbol starts. This ultimately means that this offset has to be kept in mind when loading samples from the shared buffer. If the offset is not adjusted for, the demodulation would occur with samples between 2 symbol periods instead of just being inside 1 symbol period.

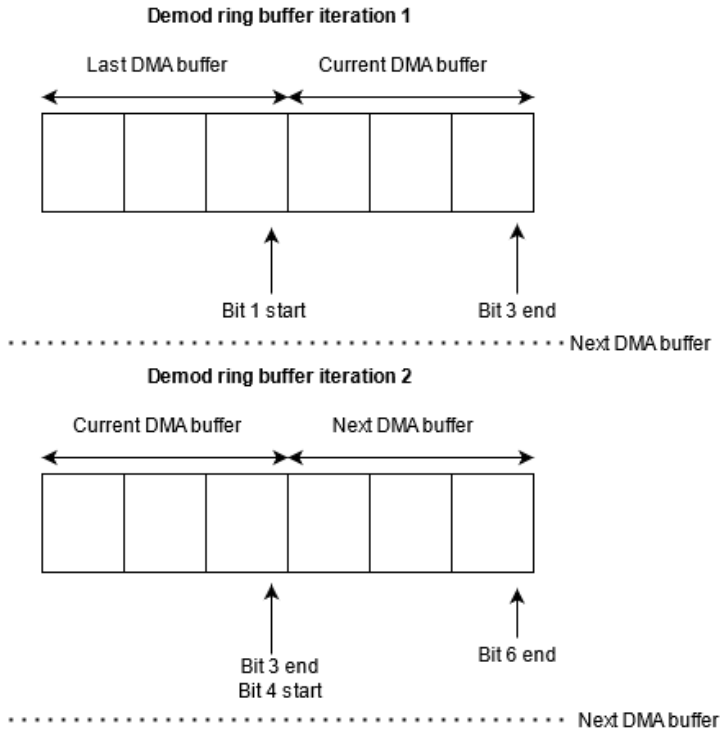

Figure 5.9: Solution to the offset in the beginning of each bit, or symbol period.

This issue is clearly present when less than the entire frame is in the buffer, but it can

be countered by not demodulating exactly the current batch of samples loaded from the shared buffer. Instead, load this into a small ring buffer containing two batches of samples, as shown on figure 5.9.
By keeping the samples from the previous iteration of the while loop shown in figure 5.7 in the buffer, the entirety of a DMA buffer can be demodulated. At the end of the iteration, the latter half of this ring buffer is moved into the former half, such that in the next iteration, the new samples can be moved into the latter half of the buffer. By doing this in each iteration of the while(demod_runs < msdr_read_calls), whenever a new frame is found to start, the samples are present in the small demodulation ring buffer. This means that the offset will not pose an issue for real-time demodulation by implementing it like this.

Beyond this change, the implementation in the BTLE Rx library from section 5.2 remains the exact same for the purpose of threading the system, hence this is not delved into. However, one issue that occurred when trying to thread parts of the system is the modulation thread, as it was too inefficient.

Modulation of ACK and NACK

The initial implementation of the entire system follows precisely the architecture displayed in the figure. 4.19. In this version, as shown in figure 5.10, the Mod_Thread is running as its own entity, generating and modulating responses every time a packet is detected. When the CRC has been verified, the packet to be transmitted is loaded into the tx buff, and the TX_Thread is awakened. After measuring the execution time, the parallel modulation of ACK and NACK in the same time takes 19μs on average.

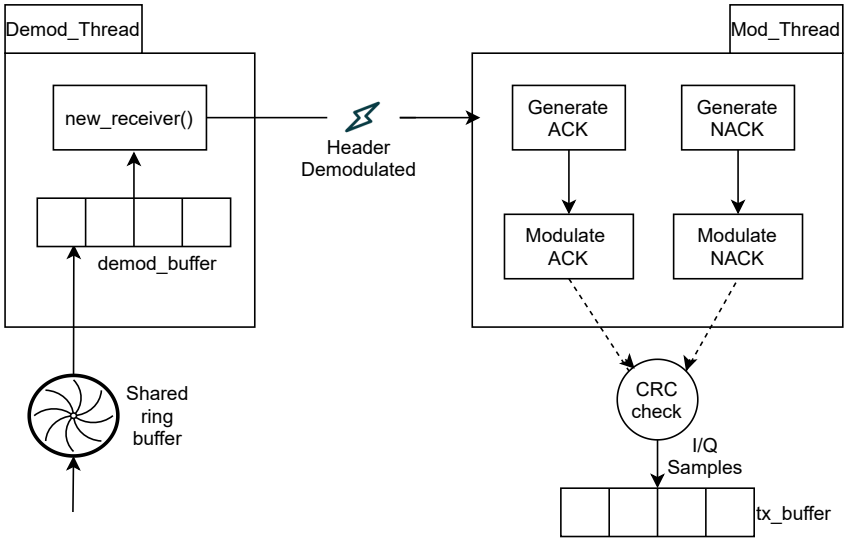

Figure 5.10: Implementation of parallel modulation design

The final version of the architecture, the Mod_Thread, has been integrated inside the Demod_Thread, as displayed in the figure 5.11. This means that the generation and modulation of ACK and NACK do not wake up a separate thread. More specifically, the

generation and modulation happen serially once the Demod_Thread detects a packet. In this process, the modulation takes 9μs on average, which leads to an overall improvement of the speed of the whole system.

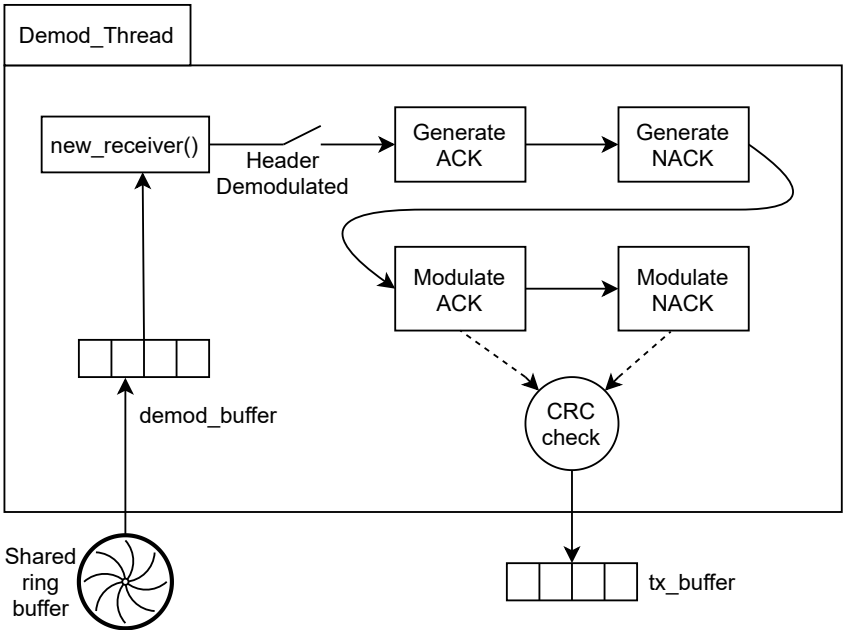

Figure 5.11: Implementation of serial modulation design

The most probable explanation of this behavior where the parallel implementation is slower is caused by the alternating context switching between two threads generating the ACK and the NACK. Waking up the thread and performing operations on the same critical section is believed to cause this overhead of 10μs when compared to the serial implementation. A thorough investigation of this behavior did not take part, as it is out of this project's scope.

5.6.3 CPU load

After implementing the Demodulation and Rx thread, an issue started to occur. The issue is that the device would crash, or the SSH connection would be lost for a while.
It is found that a high level of CPU usage occurs while running the implementation. This behavior is further investigated since the issue at hand does not happen at a low sampling rate or small DMA buffer length configuration.

Therefore, this behavior is intended to be investigated for the prospect of implementing the Tx thread. The investigation is required as it might be too much given what is currently experienced.

To investigate this, the Rx and Demodulation threads CPU consumption is respectively measured by running the program and reading the CPU % used for the given thread in htop.

The SDR itself must be killed specifically through the "kill" command after stopping the program. For that reason, the CPU time it consumes can be measured after stopping the other threads from running. This measurement technique is chosen as the exact CPU consumption is not shown as a thread.
This CPU usage is not listed anywhere and is taken straight from reading how much of the CPUs is used in htop. Generally, when nothing is running, this is around 0-2% so background related processes should have a minimal effect even if this measuring method is sub-optimal and inaccurate.
A point regarding the demodulation thread is that its CPU usage is artificially high for the most part. This is caused by the function running in a while(1) loop and sleeping 1 microsecond if nothing is ready for demodulation.

	Sample Rate	16MHz	32MHz	40MHz
DMA len				
100	Rx	5	8	8
	Demod	39	40	40
	SDR	3	3	5
50	Rx	5	8	8
	Demod	37	41	42
	SDR	6	7	7
20	RX	7	9	9
	Demod	40	42	43
	SDR	12	21	9
10	RX	8	11	14
	Demod	42	44	46
	SDR	21	31	44
8	RX	8	12	15
	Demod	44	46	49
	SDR	42	45	45
6	RX	9	13	15
	Demod	44	47	50
	SDR	33	48	64
5	RX	12	13	16
	Demod	46	49	50
	SDR	57	36	55
4	RX	11	16	16
	Demod	48	51	52
	SDR	34	100	52
3	RX	14	70	103
	Demod	52	55	56
	SDR	47	84	68

Table 9: CPU usage for the 3 different threads at different sampling rates and DMA buffer lengths

On table 9 is the listed execution times at different sampling rates and DMA buffer lengths. A general trend is that a higher sampling rate and smaller DMA buffers lead to a higher CPU load. With that being said, there are instances where the SDR itself uses more than expected CPU, e.g., at 32MHz and DMA buffer length of 4.

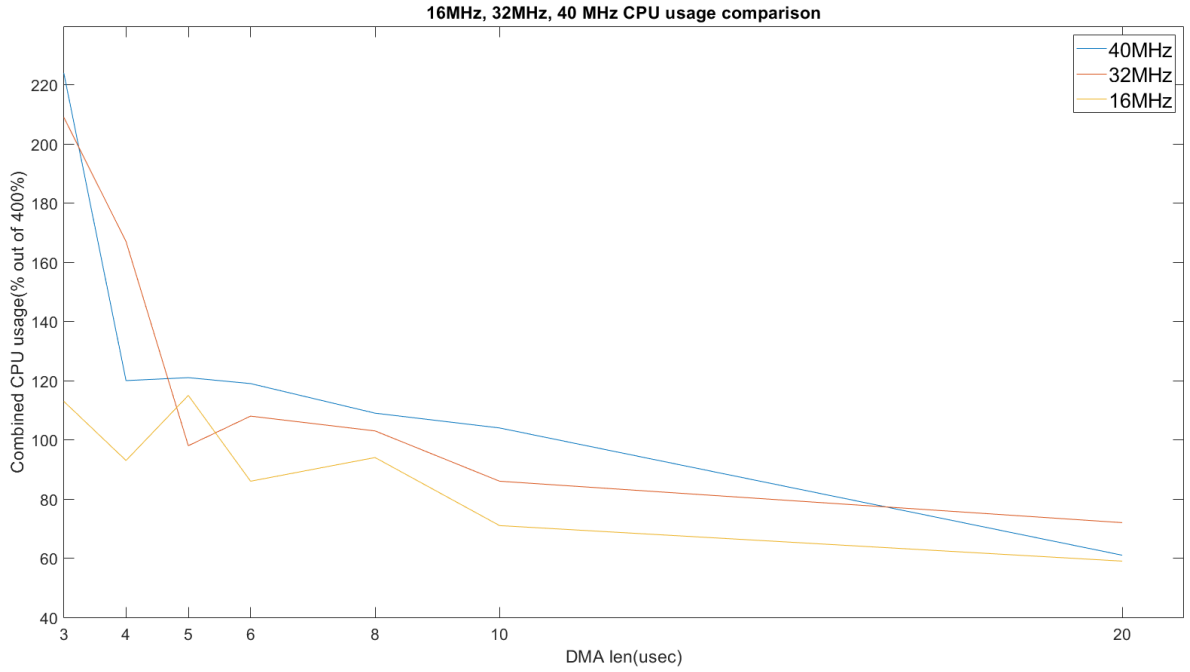

Figure 5.12: Graph of the combined CPU load. Maximum load is acheived when all 4 cores are running at 100%, which would be described as 400% CPU use.

In figure 5.13 the summed used CPU is illustrated. It is shown as the combined CPU usage at the different configurations. However, only within the 3-20 μs DMA buffer length interval. The trend overall continues with lower and lower usage. The y axis defined has a maximum height of 400%. This occurs when all 4 CPUs are being used fully at once, and at its worst, it reaches 227%.

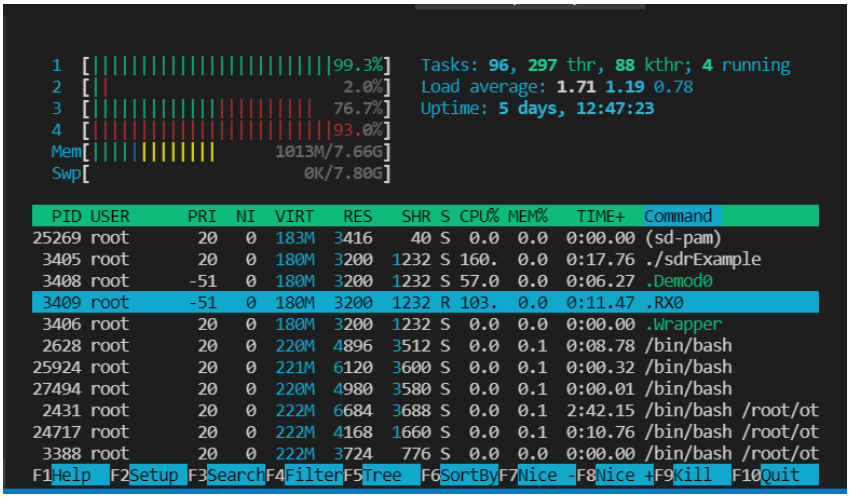

Figure 5.13: View from htop, when the program is running at 3 μs DMA buffer and 40 MHz

The chosen method for collecting these values is not perfect, as illustrated by figure 5.13 which is the terminal view of htop for a 3 μs DMA buffer, and 40 MHz sampling is shown. It is only summed up to 227% out of 400% on table 9. However, from the terminal view, the active usage should be 250% meaning there is something else occurring in the background, perhaps related to the SDR that is not actively listed.
With that being said, the method is not altered because the fundamental goal is to investigate how much of the CPU is used, and a clear trend is visible at the moment.

Ultimately the issue with the device crashing appears related to the CPU being pushed hard. A quick solution to this problem is provided using a better computer with 8 cores instead of the current 4. Therefore, instead of optimizing the threads' processing speed, a better computer is used henceforth.
This means it will be used mainly to test and include the Tx thread, but the general processing performance previously described appears only slightly affected by this. The SDR related aspects seem indifferent to this change.

5.6.4 TX thread

The Tx thread was realized through signaling between the threads and utilizing the Handler implemented in section 5.3. In its normal state, the thread keeps running a while loop. Inefficient but upon the if the condition being fulfilled transmission may occur fast.

Code snippet 11: Tx thread content

```
static void *tx_thread_func(void *opaque) {
init...
    while (keep_running) {
        if(rfp->ready_to_send == 1) { //Signaled by the demod thread
            rfp->ready_to_send = 0;
            //Handles transmission of large and small frames. Clean up of internal buffer afterward also
            tx_handler(rfp->time_to_send, rfp->samples, rfp->sample_size, rfp);


            printf("ACK SENT\n");
        }
    }
    return NULL;
}
```

The signaling is simple in this implementation. Upon finishing the CRC calculation, the correct frame loaded into a buffer flag is set. This is the "rfp->ready_to_send" flag. After calling the handler and finalizing it, the transmission is done, and the thread returns to its while(1) state waiting for the next transmission.

5.7 CPU allocation issue

After the addition of the Tx thread, the PoC has a newfound problem. The problem in question is that CPU allocation does not appear as intended. This phenomenon is referred to as a **Bad run**. This leaves challenges for the Rx thread and Tx thread and the PoC as a whole. For the Rx thread is means the msdr_read execution time of 100 μs(2.3.5) start to appear again. For the Tx thread, on the other hand, it means transmissions are not made in time.
Both of these two challenges lead to a sub-optimal performance by the PoC. The performance is therefore afflicted by the resource allocation made by the computer device.

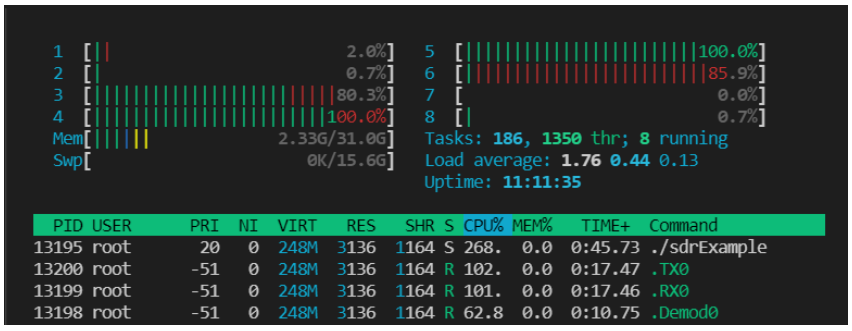

Figure 5.14: Good run, Tx and Rx are given a full core each.

Figure 5.14 is an example of a situation where the resource allocation occurs as it should. The figure is an example of CPU utilization shown by htop. It can be seen that the Rx and Tx threads are given full CPU time. It should also be noted that 4 cores are being fully used by the PoC at the moment to ensure this performance. The occurrence of these runs is around 30% of the time.

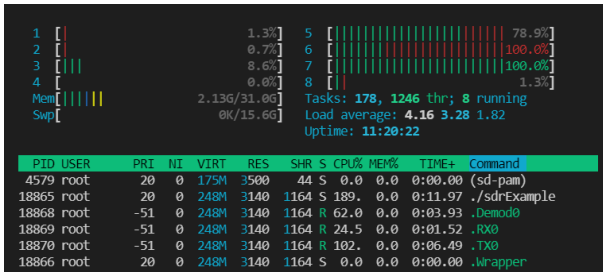

Figure 5.15: Bad run for Rx, only 25% given to it.

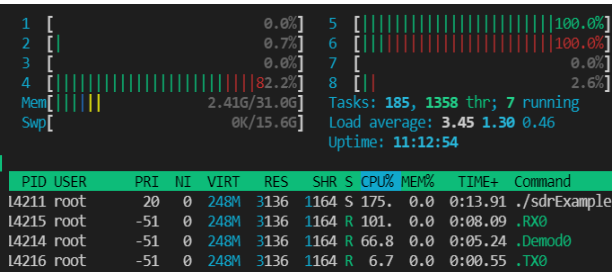

Figure 5.16: Bad run for Tx, only 7% given to it

Figure 5.16 and 5.15 is each an example of bad allocation. In this case, the Rx and Tx threads are given very little CPU time. This occurs even though more cores are left available, and could be used. The behavior is sporadic, where many times in a row, no issue is found, and then the allocation is bad for a period of time. No pattern can be found for it at the moment because of that.
As this is not considered part of the project, time is not spent trying to resolve this issue. For that reason, a simpler solution is chosen. This solution is based on rebooting the program when this phenomena occur. Fortunately, it is observable in the program that this occurs and does therefore not require outside observations. The program, therefore, stops running whenever such a situation occurs and is restarted manually.
Tests are, for that reason, intended to be performed on correct runs only. This is justified by the fact that an automatic reboot system could be created. Time has just not been set aside to implement this as of now.

6 Test

To investigate how well the developed PoC, performs according to the requirements of BLE, testing will be performed. The tests will be oriented towards verifying the implemented functionalities.

A short description of each test is given in table 10, as well as the requirements tested. A more in depth description of the test will given in the associated test section.
The full set of equipment used for the tests can be seen in figure 6.1. It includes a Signal Generator that is used to transmit BLE frames. The frames created is done using a piece of software called Signal Studio for Bluetooth(described in section 4.1.2). It allows creation and modulation of the defined BLE packet type. The second piece of equipment, the Spectrum Analyzer. It is used to demodulate frames, sent by the PoC. The Oscilloscope is used to measure differences in time between the Signal Generator and the PoC. Lastly the PoC, which has the SDR mounted and executes BLE operations.

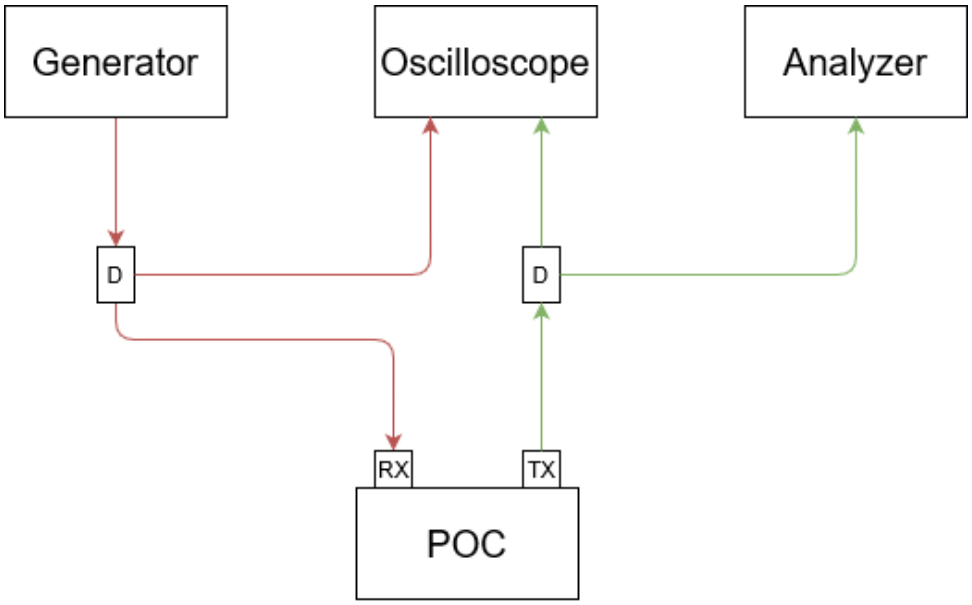

Figure 6.1: Complete test setup used.

The red connection refers to a signal produced by the generator. The green connection refers to a signal produced by the PoC.
For the tests executed the test-setup will be used. Additionally only some of the program execution are considered for the testing. This is caused by the issue presented in section 5.7, where CPU allocation causes sub-optimal performance. Therefore tests will only involve execution of the program with no problematic CPU allocation.

Test name	Functionalities	Test description	Requirements tested
Frame demodulation(6.1)	Ability to demodulate samples into bits, decapsulate BLE frames, & CRC validation.	This test is performed by receiving a frame from the generator. The frame is demodulated and the result is compared with the defined packet transmitted by the generator. The CRC is then validated.	(1.b.i), (1.b.ii), (1.b.iii),(1.d), (1.f) ,(4.b.i), (4.b.ii), (4.b.iii),
Frame modulation(6.2)	Ability to generate BLE frames, & ability modulation bits into I/Q samples.	This test is performed by comparing the packet structure of a demodulated frame with a created frame. The frame is question will have all fields defined in the same way. This means the binary structure should be the same. For modulation itself this is validated by the Spectrum Analyzer demodulated a transmitted frame.	(1.a.i), (1.a.ii), (1.c), (4.a.i), (4.a.ii), (4.a.iii)
IFS interval(6.3)	Ability to uphold IFS.	This test is performed by receiving a frame from the generator. It is then demodulated, and a response is created. This response is transmitted 150 μs after the end of the received frame.	(1.g)
Frequency shifting(6.4)	Ability to Frequency shift onto different channels.	This test is performed by configuring the SDR to a center frequency. The Generator will then send transmit on different channels within the bandwidth. I/Q shifting is employed to shift samples to the center frequency and demodulate. Afterward, a response is created and I/Q shifted onto the same frequency. It is then sent to the Spectrum Analyzer for validation.	(1.e)
Initiator state(6.5)	Ability to act as an initiator.	This is tested by listening for the device address sent by the generator,and only responding with a CONNECT_REQ if the device address is correct.	(3.a)

Table 10: Test definitions and requirements tested.

6.1 Frame demodulation

The test to validate the demodulation capabilities of the PoC is performed. For that purpose figure 6.2 is used to show the equipment used.

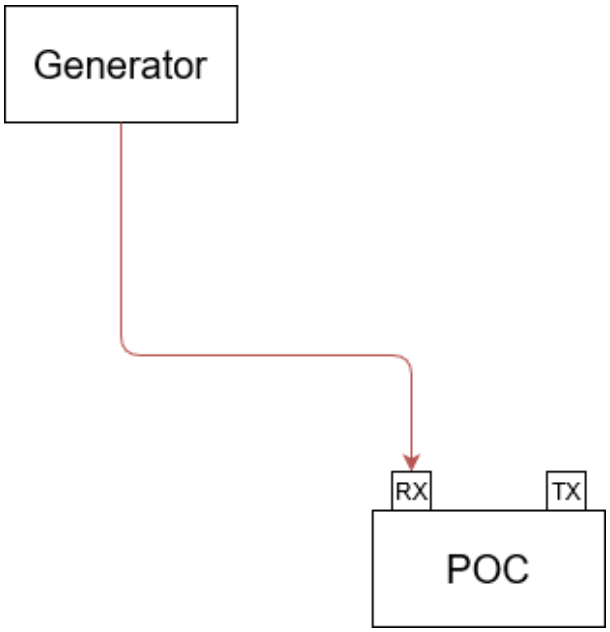

Figure 6.2: Equipment used for frame demodulation testing

Three separate functionalities are to be tested. These are: Demodulation, Packet decaptulation, and CRC validation.
Demodulation is not given a separate test, as decapsulating a frame correctly, proves this functionality.
The frame decapsulation test is outlined by 6 sub-tests. Each of these tests involves the transmission of a specific BLE packet type from the generator. The packet is demodulated by the program, and the packet type and its relevant information is extracted. This is compared against the frame defined in Signal Studio for Bluetooth.

1. ADV_IND
2. ADV_DIRECT_IND
3. LL_DATA
4. LL_FEATURE_REQ
5. LL_FEATURE_RSP
6. LL_VERSION_IND

If the extracted information matches the defined information in Signal Studio for Bluetooth, then the test is considered a success.

The final test of CRC validation is performed by taking a correctly received frame. This frame has its CRC calculated and validated against the received CRC. Afterwards, a single byte is flipped, and the CRC is recalculated for this deliberately incorrect frame.

Demodulation

The first packet type tested is ADV_IND. An example of the packet setup in Signal Studio for Bluetooth can be found in appendix figure 11.8. The fields used to define this frame are:

- Packet type: ADV_IND
- TxAdd: 0
- Rxadd: 0
- Payload length: 11
- Device address: 0000000008
- Payload: FF00AAFF00

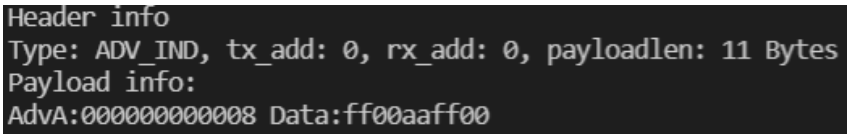

Figure 6.3: ADV_IND output

Figure 6.3 presents the information extracted from the demodulated frame. In this figure it can be seen that the data matches the defined frame. Therefore, this is considered a success.

The second packet type tested is the ADV_DIRECT_IND where, an example of the definition in Signal Studio for Bluetooth can be found in appendix figure 11.9. The fields fields used and their content can be seen below:

- Packet type: ADV_DIRECT_IND
- TxAdd: 0
- Rxadd: 0
- Payload length: 12
- Device address: 0000000008
- Initiator address: 0000000008

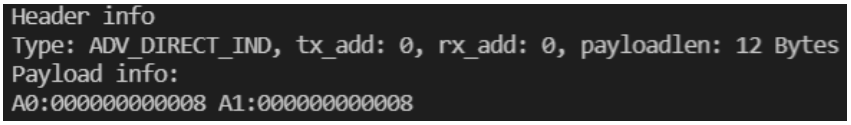

Figure 6.4: ADV_DIRECT_IND output

Figure 6.4 presents the information extracted from the demodulated frame. In this figure, it can be seen that the data matches the defined frame. Therefore, this is considered a success.

The third packet type tested is the LL_DATA, where an example of the definition in Signal Studio for Bluetooth can be found in appendix figure 11.10. The fields fields used and their content can be seen below:

- Packet type: LL_DATA
- LLID: 1
- NESN: 0
- SN: 1
- MD: 1
- Payload length: 5
- Payload: FF00AAFF00

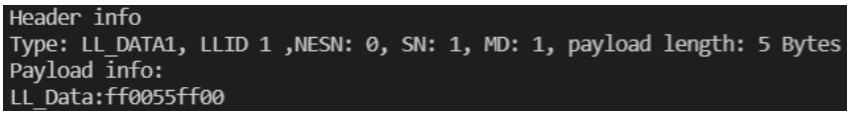

Figure 6.5: LL_DATA output

Figure 6.5 presents the information extracted from the demodulated frame LL_DATA packet. In this figure, it can be seen that the data matches the defined frame. Therefore, this is considered a success.

The fourth packet type tested is the LL_FEATURE_REQ, where an example of the definition in Signal Studio for Bluetooth can be found in appendix figure 11.11. The fields fields used and their content can be seen below:

- Packet type: LL_FEATURE_REQ
- LLID: 3
- NESN: 0

- SN: 1
- MD: 1
- Payload length: 9
- Opcode: 08
- Features : 0000000000000000

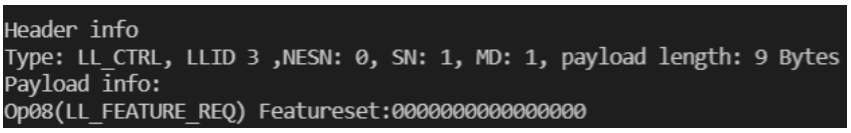

Figure 6.6: LL_FEATURE_REQ output

Figure 6.6 presents the information extracted from the demodulated frame LL_FEATURE_REQ packet. In this figure, it can be seen that the data matches the defined frame. Therefore, this is considered a success.

The fifth packet type tested is the LL_FEATURE_RSP, where an example of the definition in Signal Studio for Bluetooth can be found in appendix figure 11.12. The fields fields used and their content can be seen below:

- Packet type: LL_FEATURE_RSP
- LLID: 3
- NESN: 0
- SN: 1
- MD: 1
- Payload length: 9
- Opcode: 09
- Features : 0000000000000000 (Hex)

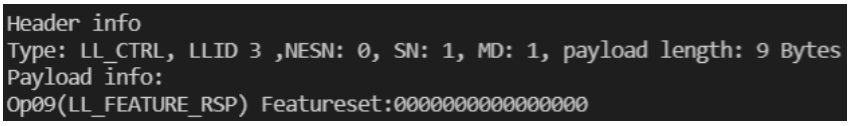

Figure 6.7: LL_FEATURE_RSP output

Figure 6.7 presents the information extracted from the demodulated frame LL_FEATURE_RSP packet. In this figure, it can be seen that the data matches the defined frame. Therefore, this is considered a success.

The sixth packet type tested is the LL_Version_IND, where an example of the definition in Signal Studio for Bluetooth can be found in appendix figure 11.13. The fields fields used and their content can be seen below:

- Packet type: LL_VERSION_IND
- LLID: 3
- NESN: 0
- SN: 1
- MD: 1
- opcode :0C
- Payload length: 6
- Version number: 8
- Company ID: 0000
- Subversion: 0000

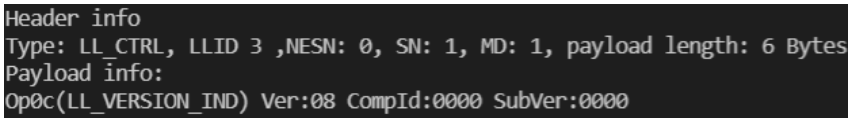

Figure 6.8: LL_VERSION_IND

Figure 6.8 presents the information extracted from the demodulated frame LL_VERSION_IND packet. In this figure, it can be seen that the data matches the defined frame. Therefore, this is considered a success.

From these 6 sub-tests it was shown that packets were decapsulated correctly by extracting relevant information. Subsequently from this it can be inferred that demodulation must have operated correctly as well. For that reason both packet decapsulation and demodulation are considered validated.
The validated packet types can be seen on table 11 which shows the proven demodulatable packet types. It also highlight their respective importance, when relating it to the milestones made for the PoC

Base required packet types		
ADV_IND	✓	Figure 6.3
ADV_DIRECT_IND	✓	Figure 6.4
LL_DATA_PDU	✓	Figure 6.5
Extended required frames		
LL_FEATURE_REQ	✓	Figure 6.6
LL_FEATURE_RSP	✓	Figure 6.7
LL_VERSION_IND	✓	Figure 6.8

Table 11: Demodulated frame types that can be demodulated. Base required frames refer to milestone 1. Extended required frames relate to milestone 4.

CRC

To validate that CRC checks are performed correctly this is tested on a received frame. This should first show a valid CRC check. That is it went through successfully and the computed CRC is equal to the received one. Afterward, a byte is changed in the received data. This CRC is computed for the new sequence and compared against the received CRC. After this the CRC check should produce a negative answer.

```
|10011000|10100000|11111111|00000000|10101010|11111111|00000000|10111101|01000110|11000011
crccheck 1

|10011000|10100000|11111111|00000000|01010101|11111111|00000000|10111101|01000110|11000011
crccheck 0
```

Figure 6.9: CRC check

From this test, it is shown that the original demodulated packet structure leads to a valid CRC check. By changing a byte the CRC check subsequently fails, as the received sequence is not equal to the calculated one.

From these tests performed the requirements associated with the test are considered validated. This is caused by each functionality showing that they work as intended.

6.2 Frame modulation

The equipment utilized to validated frame creation and modulation is shown in figure 6.10.

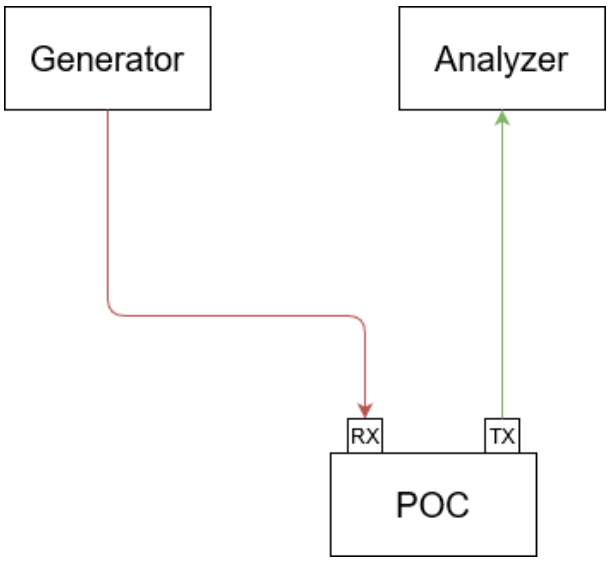

Figure 6.10: Equipment used for frame modulation testing

The frame creation functionalities tested are: The ability to create the correct five BLE packet types. This will be based on five sub-tests, named below:

1. CONNECT_REQ
2. LL_DATA
3. LL_FEATURE_REQ
4. LL_FEATURE_RSP
5. LL_VERSION_IND

Because packet decapsulation was proven to work in section 6.1, it will be used to validate created frames. This is done by a binary comparison between a demodulated frame and a created one. The frame content will for that reason need to be the same for both.
Each sub-test is considered valid if the decapsulated packet structure matches the created one perfectly.

For the purpose of testing modulation, the Spectrum Analyzer will be used. The Spectrum Analyzer can show the packet type and PDU length. By modulating a frame and transmitting it to the Spectrum Analyzer the correctness of the functionality may be proven. This is because the Spectrum Analyzer will only show the correct payload length, if the message was modulated correctly.
This test is performed last because it requires a correct BLE frame for modulation before it can be validated.

Frame generation

The first packet type tested is the CONNECT_REQ, where the exact definition in Signal Studio for Bluetooth can be found in 11.14 along with the arguments used to produce the same frame by the PoC. The bit wise comparison can be seen on figure 6.11

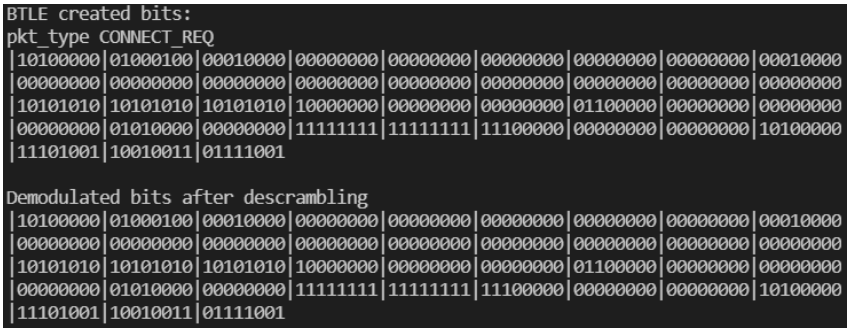

Figure 6.11: CONNECT_REQ output

From the bit-wise comparison seen in figure 6.11 it can be seen the same binary structure is present. This suggests that the PoC can create CONNECT_REQ packets.

Second packet type tested is LL_DATA, where the exact definition in Signal Studio for Bluetooth can be found in 11.10, along with the arguments used to produce the same frame by the PoC. The bit wise comparison can be seen on figure 6.12

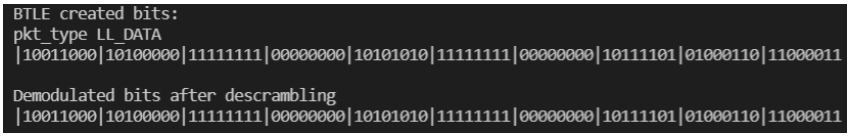

Figure 6.12: LL_DATA output

From the bit-wise comparison seen in figure 6.12 it can be seen the same binary structure is present. This suggests that the PoC can create LL_DATA packets.

Third packet type tested LL_FEATURE_REQ where the exact definition in Signal Studio for Bluetooth can be found in 11.11, along with the arguments used to produce the same frame by the PoC. The bit wise comparison can be seen on figure 6.13

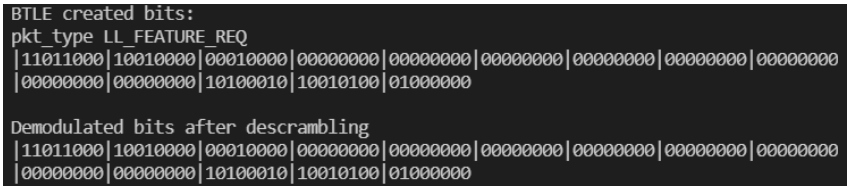

Figure 6.13: LL_FEATURE_REQ output

From the bit-wise comparison seen in figure 6.13 it can be seen the same binary structure is present. This suggests that the PoC can create LL_FEATURE_REQ packets.

Fourth packet type tested LL_FEATURE_RSP where the exact definition in Signal Studio for Bluetooth can be found in 11.12, along with the arguments used to produce the same frame by the PoC. The bit wise comparison can be seen on figure 6.14

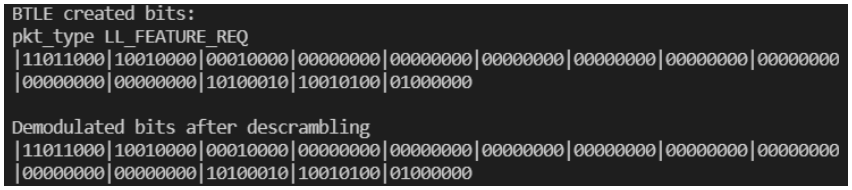

Figure 6.14: LL_FEATURE_RSP output

From the bit-wise comparison seen in figure 6.14 it can be seen the same binary structure is present. This suggests that the PoC can create LL_FEATURE_RSP packets.

Fifth packet type tested LL_VERSION_IND, where the exact definition in Signal Studio for Bluetooth can be found in 11.13, along with the arguments used to produce the same frame by the PoC. The bit wise comparison can be seen on figure 6.15

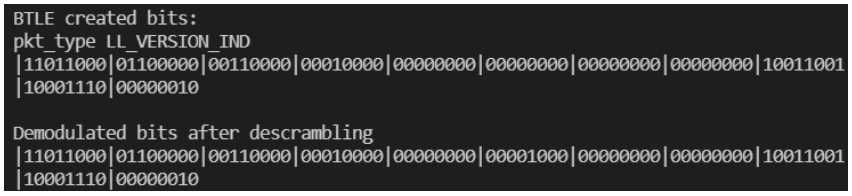

Figure 6.15: LL_VERSION_IND output

From the bit-wise comparison seen in figure 6.15 it can be seen the same binary structure is present. This suggests that the PoC can create LL_VERSION_IND packets.

As a result of these 5 tests table 12 is made. It shows the proven frames, and the relevance they are given. This is related to the milestones, where the base required packet types are required for the first milestone.

Base required packet types		
CONNECT_REQ	✓	Figure:6.11
LL_DATA_PDU	✓	Figure:6.12
Extended required packet types		
LL_FEATURE_REQ	✓	Figure:6.13
LL_FEATURE_RSP	✓	Figure:6.14
LL_VERSION_IND	✓	Figure:6.15

Table 12: Base required packet type that can be created correctly for the first milestone are realized. Extended packet types meant for milestone 4.

Modulation

With the creation of each packet type being verified, the next test will validate the ability to modulate.
For this a CONNECT_REQ is modulated and sent by the PoC, which payload wise always has a size of 272 bits. The modulated frame is transmitted to the Signal Analyzer, where it is demodulated.
While the optimal solution would be for a frame to be demodulated correctly by another device, this alternative test is still chosen. The reasoning being that it still illustrates the ability to correctly modulate bits. Meanwhile another device, except for the Signal Analyzer, is not available to verify the modulation functionality.

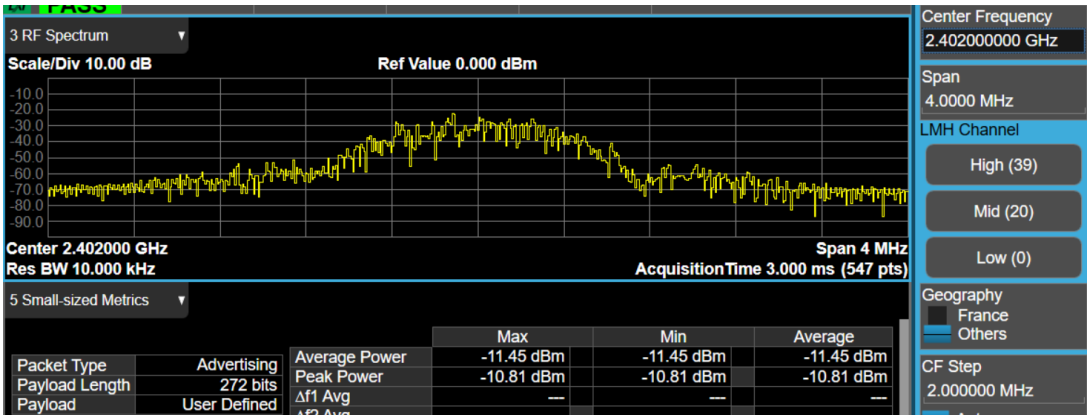

Figure 6.16: Modulation is performed correctly as the analyzer can demodulate the payload length correctly.

As seen in figure 6.16 the payload field is set to 272. This means the Analyzer could correctly demodulate up to the header aspect of the frame. while not the entire frame is demodulated and verified modulation is still considered functional. Therefore modulation is considered validated.

From these tests it has been validated that both frame creation and modulation works correctly. The method chosen for showing this is for both less than optimal. They are, however, still considered valid as no other equipment is available to show this.

6.3 IFS Interval

The setup for this test makes use of the signal generator, which will send out messages for the PoC to respond to. The oscilloscope will also be utilized to see the difference in time between the message sent by the generator and the response sent by the PoC.

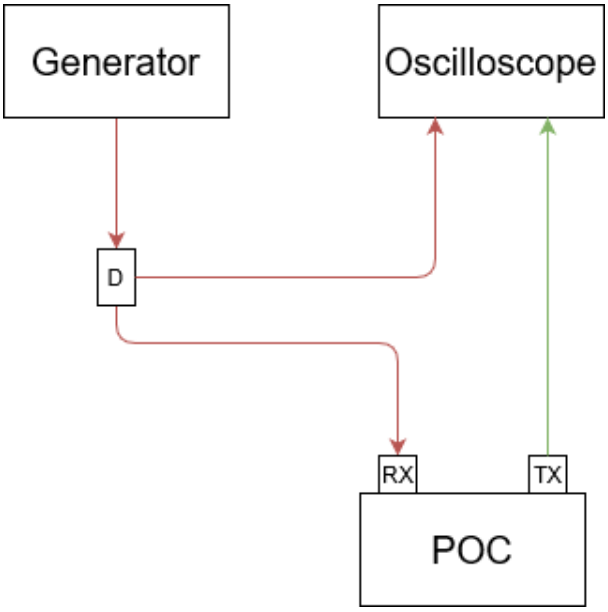

Figure 6.17: Equipment used for IFS testing

The IFS is the crucial requirement to uphold, as it is necessary for BLE communication. It is also the primary functionality not previously achieved with other SDR units. This is why it is tested how well the IFS and its deviation is upheld in this implementation. The test is run with the PoC, where upon demodulation of a full frame and CRC calculation, a response is created. The response is scheduled for 150 μs after the last bit of the CRC.
The program is triggered by the generator, where the generator will keep sending new frames. The program will send a response to each, as long as it is running.

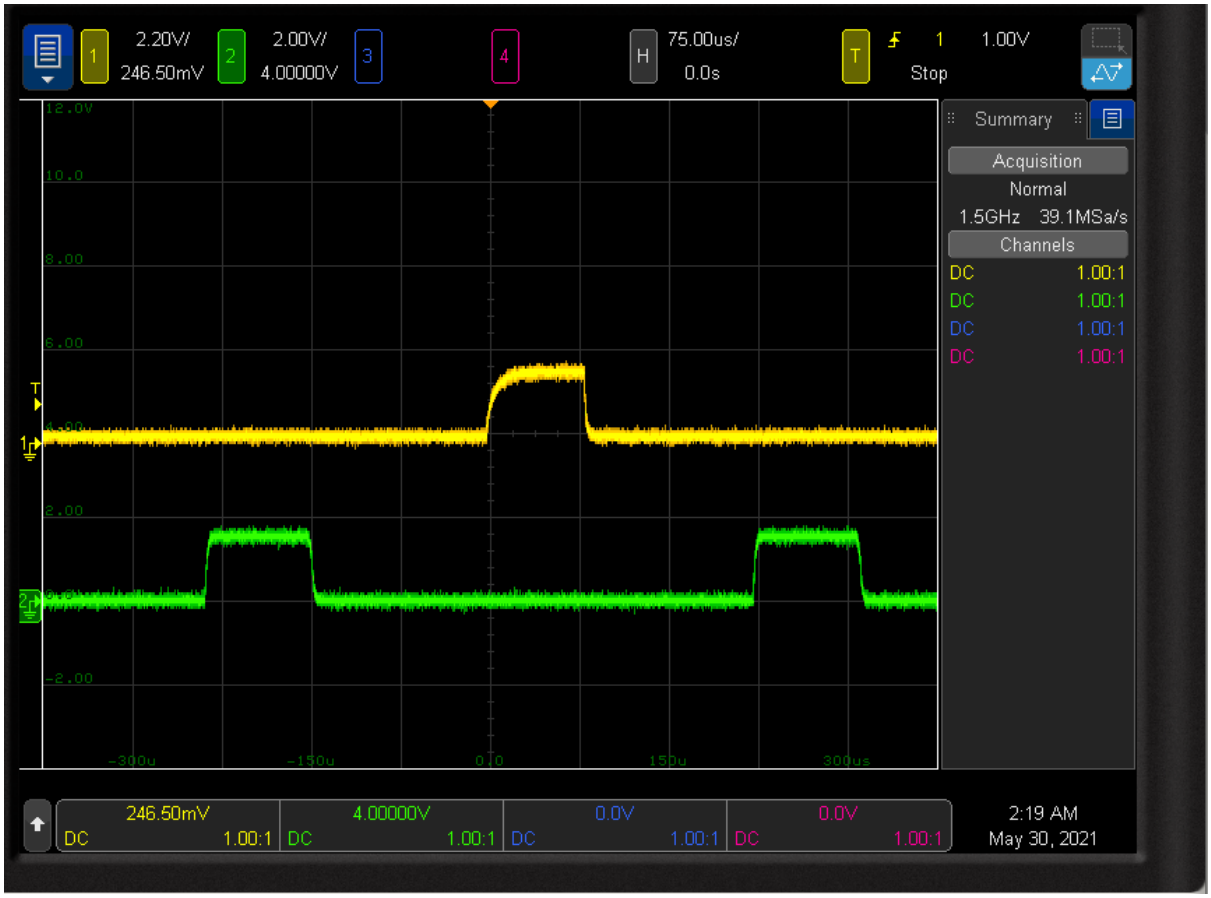

Figure 6.18: Example view of receive-transmit time on the oscilloscope.

Figure 6.18 is an example of a zoomed in view of the difference between the Signal Generator's signal and the signal of the SDR's response. It can be seen that the there are two blocks of 75 μs between them, which is the IFS interval. One thing to note, is that to get an accurate reading on the Oscilloscope alone is difficult hence this data is processed differently.
The voltage levels of the two signals shown on the oscilloscope is extracted in a .csv format and processed in Matlab. This is done to better process the data of each response time, and get an accurate result of each one. It also allows to process many responses at once. The resolution is done at 10 data points per microsecond.
The processing involves using the amplitude of voltage levels to determine the beginning and end of a frame. The difference in time is then defined as difference in the data points between the end of Generator's frame, and the beginning of the PoC frame. This is also normalized to time i.e. 10 data points is one microsecond.

The previously described issue regarding CPU allocation, section 5.7, is especially detrimental for upholding the IFS. For that reason it is important for this test, that only the so-called good runs are used for evaluation.
This is justified by the fact the project is related to the feasibility of an implementation. The CPU allocation performed by the Operating System is not related. This especially

applies as more cores are left available but unused, when the bad CPU allocation occur. The effect of only using good runs is that msdr_read does not exceed 40 μs execution time. The Tx thread also performs its msdr_write operations on time.

For demonstrating the capability of the PoC to transmit within the IFS, Five separate test runs are made. To show the individual performance of each test the results for the reception-transmission time is shown in table 13. The first column shows the test in question. The second column shows the mean observed time difference. The third column shows the range of observed values for that test.
The number of reception-transmissions recorded for each of the test is 85. That is the amount that could be fit into the scope of the oscilloscope for data extraction. The fourth column shows the accuracy of instances uphold the IFS.

Test:	Mean(μs)	Range (μs)	Accuracy (%)
Test 1	149.27	$148 \leq x \leq 152$	100%
Test 2	149.25	$148 \leq x \leq 152.4$	95%
Test 3	149.25	$147.7 \leq x \leq 151.8$	98%
Test 4	149.39	$147.9 \leq x \leq 152.1$	98%
Test 5	149.2	$147.9 \leq x \leq 159.8$	95%

Table 13: IFS test results.

From the five tests on table 13, the mean values all appear within the allowed IFS interval. One issue occurs in regards to the observed range of values. The issue is that not all fall within the IFS and its allowed deviation of $\pm 2\mu s$. This is not preferable, but part of the issue can be related to the data processing afterwards.
Depending on how accurate the method used to determine the beginning of a packet from the generator and one from the program the result changes. That is to say, the post-processing process can also be a cause of this.
An important number seen in test 5 is the maximum range of roughly 160 μs. This is quite far from the allowed IFS. It is, however, a single instance and it could be caused by a bit flip in the header. If the header was demodulated to be one byte longer it could possibly create the observed result. The demodulated frame was not recorded at any point, hence this cannot be verified.

To give a further understanding of how the test results were distributed in time between reception and transmission, figure 6.19 was made. It is a collection of how each of the test was distributed.
The green bins are those that fall into $148\mu s \leq x \leq 152\mu s$, meaning those instances where the IFS is upheld with the allowed deviation. The red bins is that of those that fell outside this interval.

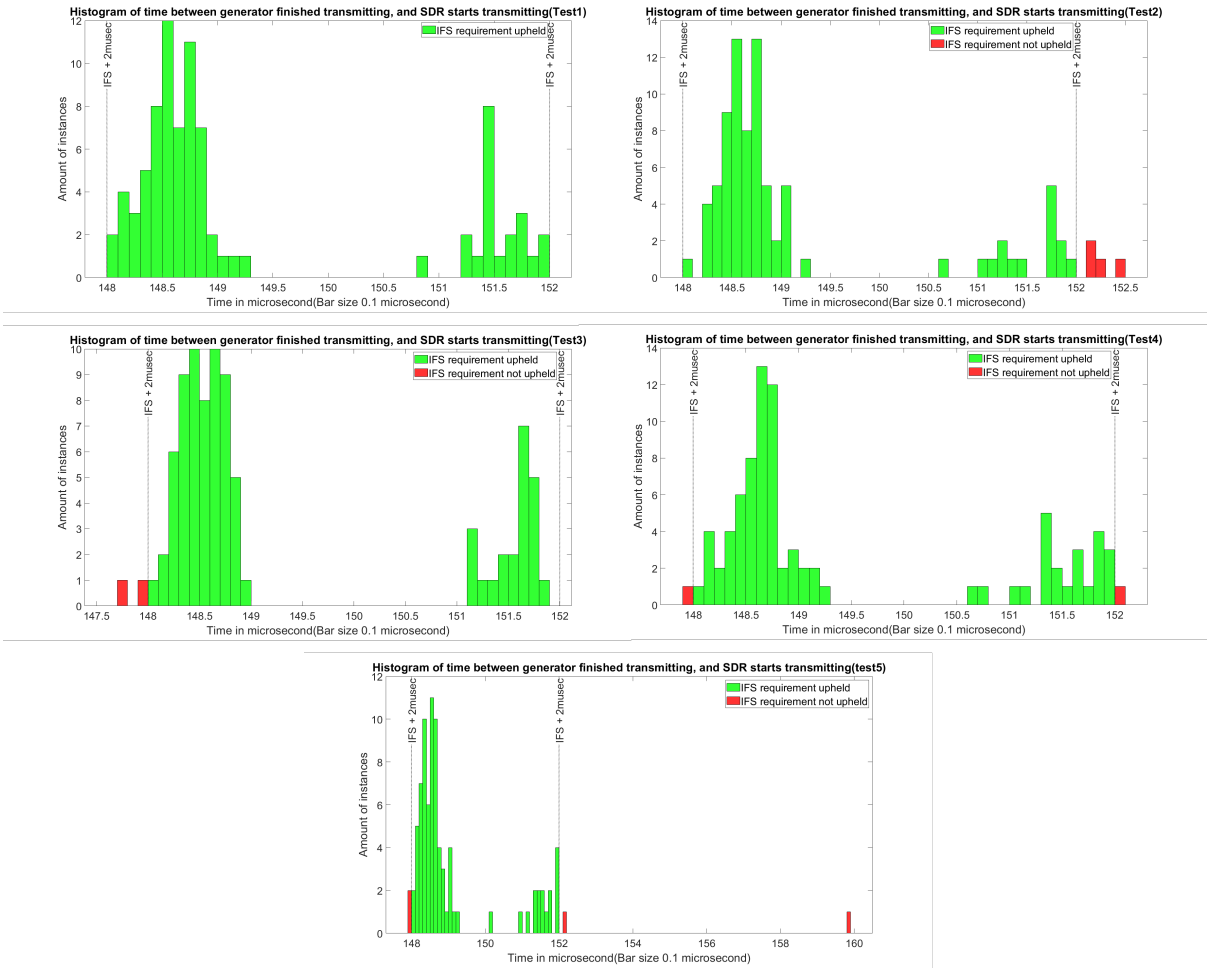

Figure 6.19: Histogram of the receive-transmit times of the five tests.

As can be seen on figure 6.19, the results vary in terms of transmissions that do not fit in the IFS. In test 1 there were none outside the IFS, while all the others had them. The test suggests that the accuracy of writing for a specific point is not optimal with the current implementation. Especially on the last test (figure 6.19 test 5), where the highest observed msdr_write time was near 160 μs. This was as previously mentioned only a single instance.

To give a final perspective of how these instances of response-transmission are distributed figure 6.20 is made which shows the overall distributed times when the tests are combined.

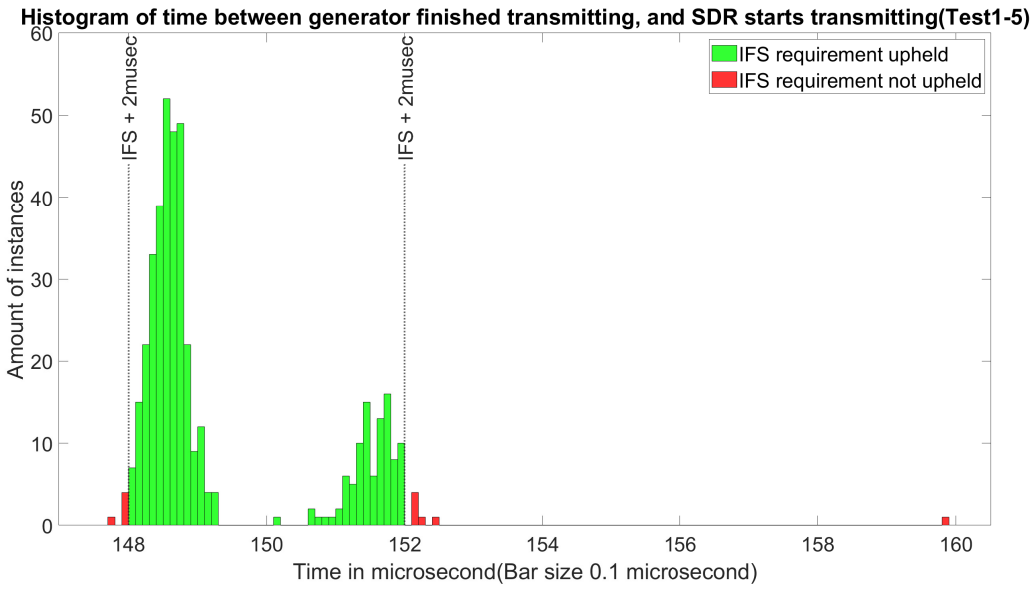

Figure 6.20: Full histogram of the recorded receive-transmit times

In figure 6.20 413 of the 425 instances operate within the IFS of $150 \pm 2\mu s$. This shows that 97% of the frames sent are received within the allowed time interval. These results mean the the PoC is able to meet the BLE specifications in terms of the IFS and thus is able to communicate with other BLE devices.

Ultimately this test confirms that transmission within the previously impossible IFS is possible with this SDR. This is while also including other required BLE functionalities. This involves demodulating the packet, and modulating the response on the run, without earlier preparation.

6.4 Frequency Shifting

To validate the ability to frequency shift this test is performed. It uses the same test-setup defined in figure 6.10

Execution of this test is performed by configuring the SDR to a single center frequency.

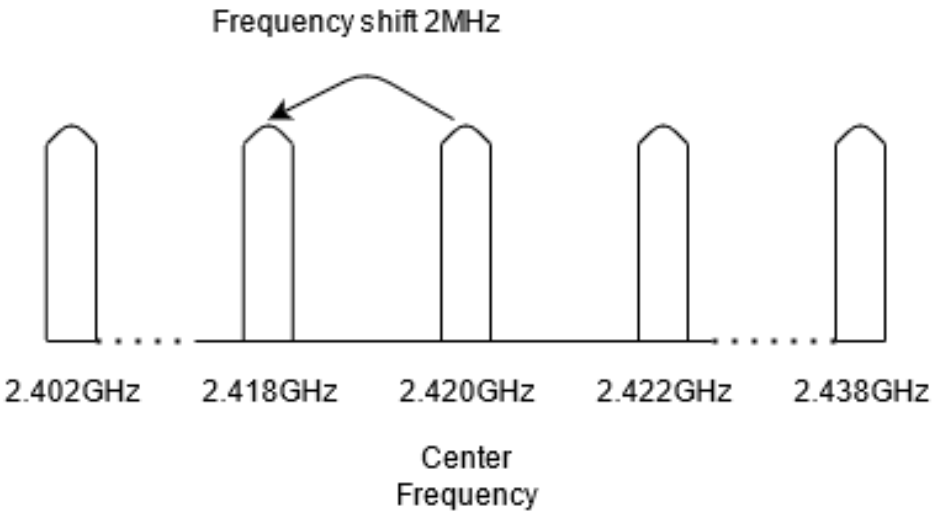

Figure 6.21: Frequency shifting illustration

The test involves 19 sub-tests for each of the channels to be used. Figure 6.21 visualises the concept of how this is done. The PoC is configured to the center frequency. The testing then involves shifting I/Q samples at base-band upon reception, and before transmission unto the other channels.

For each test the Generator is configured to a single frequency which it will transmit on. The PoC will shift the I/Q samples at base-band from that frequency to the center frequency. Here the result is demodulated. It should be the same decapsulated data for each test performed.
After demodulation a response is created for the same frequency as the generator. After modulating the response the samples are shifted at base-band onto this frequency. The frame is then transmitted and received by the Analyzer.
On the Analyzer it can be validated through the demodulated header, that the frame is still correct even after the I/Q shifting.

Performing these two tests confirms the functionality of the PoC to switch frequency at will. This applies to both the reception and transmission of frames. It leads to the required functionality to perform frequency hopping in BLE.
It should be noted that the processing time to switch the frequency is negligible hence it does not infringe on the IFS.

The packet to be sent by the generator is device address 0000000008, payload FF00AAFF00. This is supposed to be demodulated for the reception part to be considered valid.
The packet sent to the Analyzer is a simple LL_DATA pdu. Payload length is zero as it is an ACK/NACK structure. This must be observed on the analyzer for the test to be considered valid.

An example of the frequency shifting being performed are illustrated by the three picture:6.22 6.23, and 6.24. The example is related to shifting from 2.420Ghz to 2.438Ghz

i.e. the farthest that can be shifted given the bandwidth.

```
LL_PDU_t1:LL_DATA1 NESN0 SN1 MD1 PloadLen5
Payload:LL_Data:ff00aaff00 CRC1
```

Figure 6.22: Configured for the center frequency 2.420GHz. Still demodulates correctly.

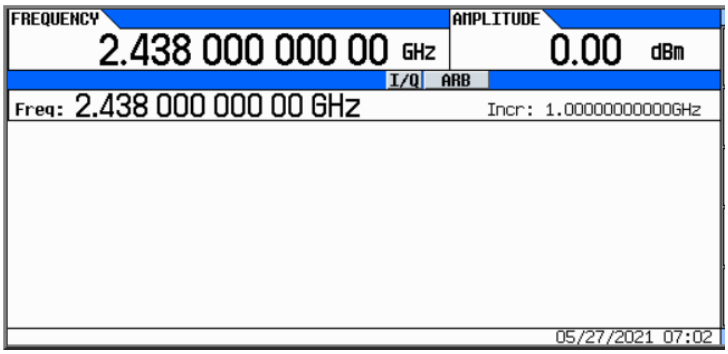

Figure 6.23: Generator configured for 2.438GHz.

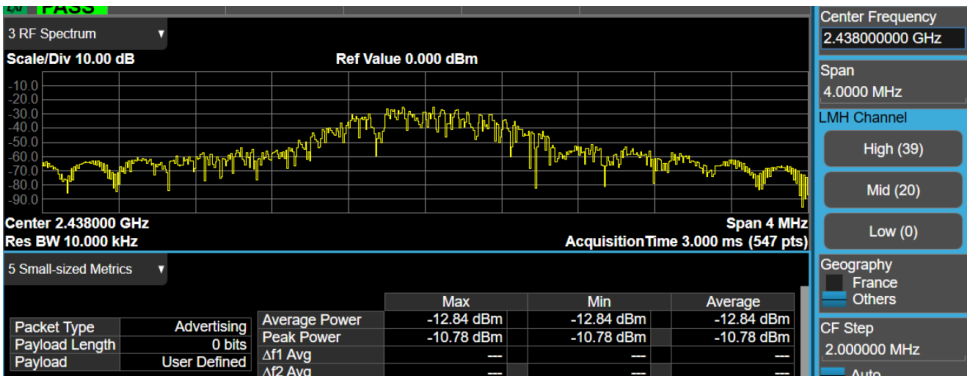

Figure 6.24: Analyzer configured to 2.438GHz.

From figure 6.22 it is seen that the Tx frequency of the SDR is 2420MHz . This is the previously mentioned center frequency.
On figure 6.23 the generator be seen where its transmission frequency is set to 2.438Ghz, ie.. 2438 MHz. It is transmitting a LL_DATA packet meant for that frequency. The demodulated output can be seen in figure 6.22 where the interesting things are the AdvA (device address) and Data field. These fit with what was defined for transmission by the generator.
Figure 6.24 shows that the transmission went through correctly. This can be seen from the fact that the center frequency is set to be 2.438GHz, and payload length is correctly demodulated.

The same result was found for all of the frequency shifting tests performed. That means the same payload was demodulated as 00ffaaff00 for each frequency shift done. The

analyzer likewise always showed payload length 0, when tuned to the correct correct channel.

Freq	Demodulate correct	Transmitted correct
2402	✓	✓
2404	✓	✓
2406	✓	✓
2408	✓	✓
2410	✓	✓
2412	✓	✓
2414	✓	✓
2416	✓	✓
2418	✓	✓
2420	✓	✓
2422	✓	✓
2424	✓	✓
2426	✓	✓
2428	✓	✓
2430	✓	✓
2432	✓	✓
2434	✓	✓
2436	✓	✓
2438	✓	✓

Table 14: Frequency shifting test. All frequencies can be used

Shown on table 14 is the result of the test. As seen for all 19 channels both the reception on the SDR and before transmission, are shifted correctly.

This means all 19 channels can be covered. One challenge is when performing frequency shifting on large amounts of samples at once. The currently transmitted response of 80 bits were transmitted correctly according to IFS as otherwise no result would appear on the Analyzer. There is is unfortunately a limiting factor as larger frames, with more bits associated with it, take longer to I/Q shift.

6.5 Initiator state

This test aims to validate the correct functionality of the Initiating State of the Link Layer and consists of 3 phases:

- Phase 1: Correctly receive Advertisement packets.
- Phase 2: Generate and modulate a CON_REQ packet in the form of an ACK to the received Advertisement packet.

- Phase 3: Transmit the CON_REQ within the IFS to establish the connection and transit to Link Layer's Connected State.

For this test, a commercial device has been used to establish communication with the SDR platform. The product used is a Pycom LoPy v4 [15] with the Pycom Expansion Board v3.1 [16]. This device was selected because it is a development board, which means that it's functionalities are programmable to a certain point, using a defined API. The reasons for choosing commercial devices for this tests are two: First, they support basic Link Layer operations, such as the Advertising and the Scanning State. Second, one Pycom can be used to transmit packets, while the other is scanning and printing the received packets. This setup is used on phase 2 of this test. The testing phases are visualised in the following diagram 6.25 with respect to the integration of the Initiator State in the PoC(5.5)

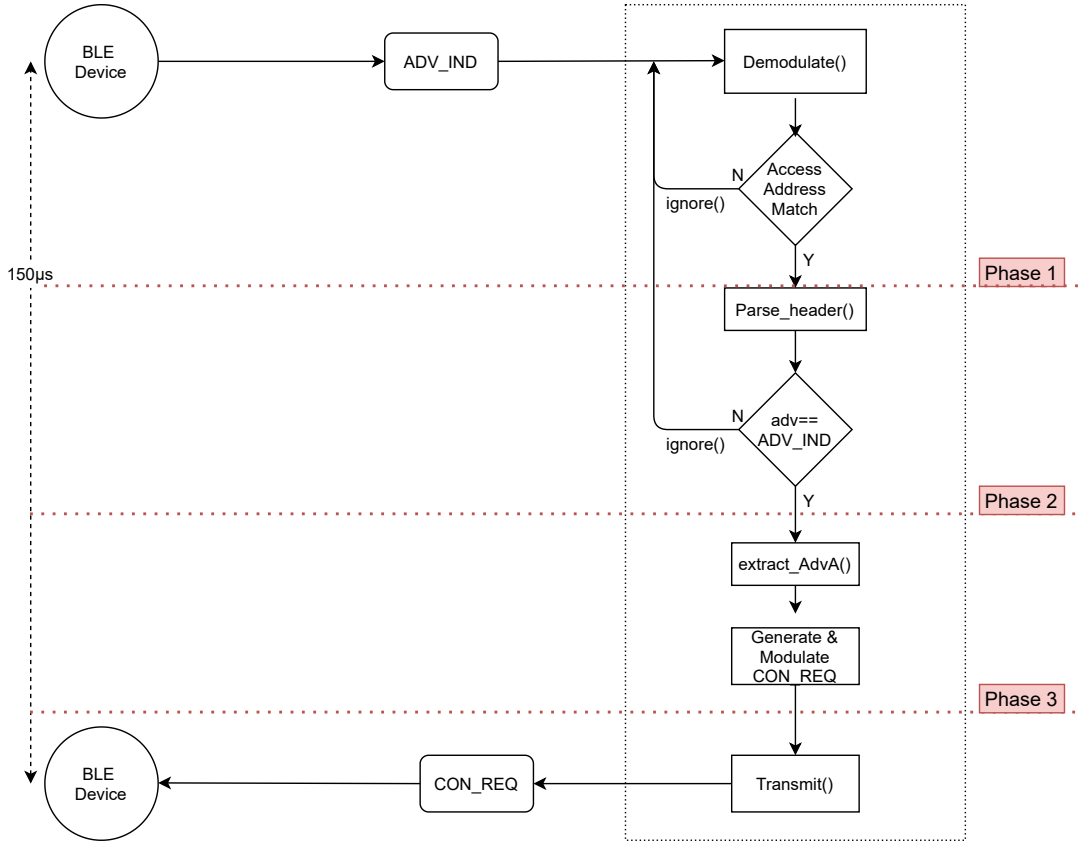

Figure 6.25: The Initiator role implementation split into test phases. The procedure also applies to ADV_DIRECT_IND messages.

To confirm that the Link Layer handles the Initiating State correctly, the sequence of the described operations has to be validated.
Firstly, the detection of advertisement packets. The pycom is set to transmit ADV_IND packets periodically in an advertisement channel. The SDR is set to listen on the same channel. Whenever a signal is captured, the frame is demodulated. If the Access Address is a public one, used for advertisement, the program proceeds with the demodulation

of the header. If not, it keeps looking for the Access Address in the received sample stream. During the demodulation of the header, the adv_type is always in a fixed position inside the frame. By extracting and comparing these bits with the ADV_IND and ADV_DIRECT_IND enumerated values, the Link Layer decides whether or not to begin the preparation of a CON_REQ and transit to Connected State.

For the second step in order to validate that the AdvA is being extracted correctly, two pycoms are used. One is advertising and the other is scanning for advertisement packets. This allows to have an overview of how the bit stream of the advertisement packets looks like. Once the scanning pycom receives a packet, it prints it to the terminal output (6.26). On the same time, the SDR is listening on the same channel as the advertising pycom. When detecting a packet and performing the operation described in the previous step, it extracts the AdvA field from the received packet. The most important part of this step, is to confirm that the PoC extracts the AdvA correctly. The scanning pycom is used to confirm the validity of the payloads, as the outputs are being compared. From the figure 6.27, it seems that extraction of the AdvA in the PoC happens successfully and the Link Layer proceeds to adding the AdvA to the CON_REQ packet.

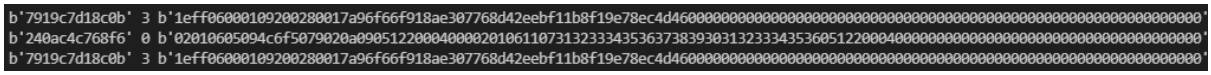

Figure 6.26: The AdvA and the payload of the advertising device, displayed by the scanning Pycom

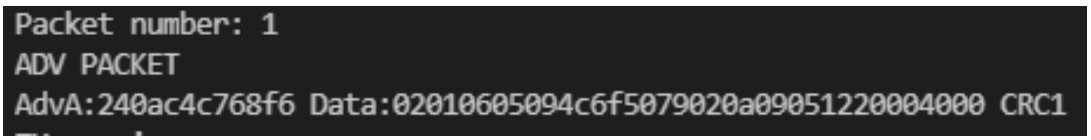

Figure 6.27: Information of the received packet, as extracted and parsed from the PoC

For the third and final step, the target is to transmit the CON_REQ within the IFS. From the analyzer and the oscilloscope it can be seen that the CON_REQ packet has been generated, modulated and transmitted successfully. The issue at this stage is, that the Pycom device does not offer an API suitable for interfacing with the Link Layer and for this reason there is no validation that the CON_REQ has been received successfully on the advertising device. This appears to be a missing piece for validating the CON_REQ transmission. However, the previous tests presented in this chapter(Frame Modulation, IFS Interval) confirm that the CON_REQ has been successfully generated and transmitted inside the desired IFS. From this moment and forward, the Link Layer should switch to the Connected State and become the Master, while the Pycom should switch to the Connected State and become the Slave.

Summary

To summarise the tests performed and the milestones reached throughout this section this small summary is used.
From the tests performed it has been found that the first milestone has all its requirements realized. This means the first milestone is considered reached.
The second and third milestone are on their way and testing related to these was placed in the Initiator test. Challenges relating to the device used to validate these states caused issues meaning for the most part these milestones are not validated.
For the fourth milestone the packet types are validated, but work is still required. A part of this work is related to milestone 2 and 3 not being fully implemented.

7 Discussion

After having tested the different elements of the PoC, it is important to discuss those results to get a detailed view of what has been achieved. The discussion will first go through relevant topics related to the implementation and the results of the different tests that have been performed. At the end of the discussion, the overall results of the project shall lead to the conclusion of this report.

7.1 The Bluetooth implementation

It has been proven that the PoC is capable of receiving and responding almost perfectly within the IFS and its deviation ($+/-2\mu s$). The speed at which the PoC operates shows that the device upholds the time requirements and is even faster than necessary, meaning that it is feasible to use the SDR with low latency protocols such as BLE. It also means there is still space to implement more BLE functionalities while staying within the IFS. It can also be seen in the test results that the PoC can generate frames and modulate them correctly. This functionality is ensured been by using a third-party BLE library as a foundation and then greatly expanding on its functionality to work in real-time communication. Similarly, in the area of demodulation, the receiver function from the library has been greatly altered to function with the PoC. This modification primarily relates to demodulating incoming frames in real-time. This functionality was not possible with the original library. The original one required the entire frame present before demodulation was possible. The new version is both faster and more flexible, as it has no such requirement.

A different topic for the BLE implementation is synchronization with the incoming samples. Synchronization occurs with the access address of the BLE packet, where the receiver will go through the incoming samples and find the beginning of a frame. When it has found the correct sequence of bits, it knows what samples to use for demodulation. This was the original functionality in the chosen third-party library. A point that can be made is that this should be changed to preamble synchronization. Preamble synchronization would be more flexible due to only 2 patterns exist, as opposed to the access address synchronization with up to 2^{32} unique access addresses. Seeing as it has no other detrimental effects on the functionality, access address synchronization was selected. This has been chosen to prioritize more relevant aspects of the project.
However, a problem occurred, where a high sampling rate was used to ensure IFS compliance. Because of this sampling rate, the original synchronization was too slow. Preamble detection was a viable alternative, but instead, the down-sampling solution was chosen. It was a simple solution, which functioned adequately in the lab environment. For usage with other devices, however, it is sub-optimal as down-sampling would also require frequency filtering.

Another aspect to consider regarding the implementation of Bluetooth is the function-

alities that have not been fully implemented yet. The first thing to comment on is the lack of standard frequency shifting capability. The SDR is not able to jump between frequencies at the required speed, meaning that frequency shifting has been achieved in a slightly different way. Making use of the forced high sampling rate that the PoC uses, I/Q frequency shifting is used instead. The frequency shifting gives the PoC a limited ability to frequency shift by tuning the I/Q sample to a different frequency than what the SDR is configured for. That was not feasible by using the base SDR as it cannot perform frequency shifting at the fastest pace of a commercial BLE device. When testing this, it was shown that the PoC was able to transmit and receive in the correct frequencies, meaning that the I/Q frequency shifting works as intended. A thing to note about this implementation is that it is limited to 19 out of 40 BLE channels with one SDR. This amount is problematic as the SDR will be unable to communicate on the same number of channels as real-world devices can. A solution for this issue is to use another SDR for the PoC to cover the remaining channels. It should also be mentioned that large BLE frames may not currently be sent in time. This is caused by the frequency shifting function being reliant on the number of samples to be shifted. Development is, however, on its way to not fall behind in time due to a high amount of samples.

Another component that still needs to be implemented is the Link Layer manager, with all its states being fully functional. The reason for not implementing link-layer management is that the project has been focused on making a low-level connection work. This includes the implementation of elements required for a master-slave connection. Some of these elements are the Advertising State, the Initiating State, and its transition to the Connection State. This implementation can showcase that the PoC can communicate with real-world devices. Part of the missing implementation exists in the third-party BLE library and simply needs to be adapted to fit into the PoC. On the other hand, it could be argued that not having link-layer management means that the PoC is not able to uphold a "real" connection with commercial devices. As said before, it can communicate with commercial devices, but it cannot establish and uphold connections as specified by the link-layer of BLE in full.

7.2 The SDR

For the SDR aspect of this implementation, the primary challenge has been to configure it. The configuration entailed ensuring optimal speed performance to send response within the IFS of BLE. Consistent behavior and stability have also been a point of interest.
In the current state, the SDR can respond at the earliest in 105 μs from reception. This result shows not only that it can uphold BLE's strict timing restrictions but also that there is a margin for error. It does, however, still have some shortcomings in the form of its operational behavior. The inconsistent execution time of msdr_read is an example, where it has been observed to take as much as 40 μs to finish reading samples. Using this worst-case scenario, it would leave the CRC execution and time to call msdr_write 5μs. This number is a very small margin of error made available for the BLE implementation

to fit within. This is, however, improved by sending parts of the BLE frame earlier. These include the preamble and the access address.

Another challenge that would occur if the SDR were to be put into a real-life scenario with its current configuration is its high sampling rate. The sampling rate is configured to 40MHz gives advantages to the lab scenario that the SDR has been placed within. In this setting, frequency shifting is sufficiently fast and covers 40MHz of bandwidth. This is, however, an unintended feature that came from the necessary configuration of the SDR.
The issue becomes apparent if this device were to be used in a real-life setting. It would receive data from all of the channels within the 40MHz bandwidth, which may entail a large amount of both BLE and WiFi communication. Further signal processing and filtering would be required to diminish the effect of this. Which would lead to a higher execution time for the already small margin of error for this implementation.

The silver lining for this is that the SDR is a prototype device. So, if these inconsistencies were to be relieved, then an implementation without an unnecessarily high sampling rate and consistent msdr_read execution times may be possible. These are very limiting factors for communication with other BLE devices. If not for these two unexpected properties, the SDR would have even greater potential. It is in this regards also worth noting that the SDR has driver functionalities to be updated.
This relates to faster frequency shifting. It is also assumed that the unexpected time improvement from a high sampling rate could become independent of the sampling rate. For these reasons, while the result of the project is good, further improvement is on the horizon.

A whole other issue also present with this implementation is how the computer device is spending a tremendous amount of resources to make sure reception and demodulation occur as real-time as possible. Meanwhile, the transmission thread is also configured to run constantly. This is done to ensure the small margin for error is not compromised. This leads to a very high CPU usage resulting in roughly 400% out of 800%, where 100% is a full core being used. This is possibly implementation-related and could presumably be diminished significantly. However, some aspects cannot be improved upon by a developer. These are related to the SDR itself, e.g., its power-hungry behavior.
This is something that only the SDR manufacturers could alleviate since its internal operations are black-boxed.
With the CPU challenge being a sub-optimal one, it is not that detrimental for the project. This is because the focus has not been related to the optimization of CPU usage. It is instead related to proving the usage of the PCI connected SDR for BLE communication. This would need to be rectified for a complete implementation, as a relatively strong CPU would otherwise be required.

All in all, this project has focused on creating a PoC. There are different elements of BLE that have not been implemented, and there are different elements related to the SDR that still warrant investigation. Ultimately, it comes down to whether or not the PoC proved what it needed to prove, meaning that the results of the project should be

held up to the problem statement to see if the goal has been reached. This will be done in the conclusion section, while elements that can be expanded upon will be described in future work.

8 Conclusion

Ultimately, to conclude on the work produced in this report and to see if the goals that were set out have been reached, it is necessary to look back at the problem statement, which is the following:

How can a Bluetooth Low Energy controller be implemented in a state-of-the-art Software Defined Radio platform while meeting the timing requirements of the protocol?

It has been shown that the PoC can utilize the BLE protocol and communicate within the timing requirements of the protocol (the $150\mu s$ IFS). This has been the essential part of the project, and it works, which is crucial as it has not been done before. The critical elements of the link layer have also been implemented. A foundation for further development has also been laid.
Not all of the milestones were reached as elements of the BLE protocol beyond the lower-layer aspects were not implemented in full. It is worth noting that the initial steps to implementing these parts of BLE have been made. Having the initial steps implemented means that there is a solid foundation for further work on the PoC. Even with these shortcomings, the implementation could uphold the timing requirements of the IFS. To put the achievements of the project into perspective, it makes sense to look at the previously defined milestones from section 3.2:

1. Critical link-layer & physical layer implementation.
2. Initiator implementation.
3. Master role implementation.
4. Data exchange requirements.

The first milestone is implementing the lower-layer (physical and link layer) requirements of BLE on the PoC. This milestone also includes making that work within the time limit set by the IFS. The first milestone is the most critical one as getting BLE to work with the SDR within the time constraint of the protocol is what is being asked about in the problem statement. Milestone 1 has been completed, which means that the PoC shows that it is feasible to use the SDR with protocols that require low latency.

Milestone two begins structuring the Link Layer state-machine, starting from the Initiating State. It utilizes the functionalities implemented in the previous milestone and introduces logic on top of the PoC to integrate the desired states. This results in a Link-Layer manager, where the Initiating State functions are desired. This means that it is able to :

- Look only for Advertisement packets.

- Extract the necessary information from them.
- Generate and modulate valid response packets using the extracted information.
- Transmit the modulated packets within the IFS.

Milestone three contains the implementation and the base functionalities of the Master Role in a BLE connection. In the current state, the PoC can transition to the Connection State and become a Master after successfully transmitting a connection request packet from the Initiating State. The functionalities of the Master Role, after a connection has been established, have not been implemented. This is because the main focus of the project resides in the lower layers of BLE protocol stack, and it was considered preferable to utilize the time given into implementing functionalities in these layers. However, the foundations to integrate the Master Role in the PoC have been laid. What is missing is a handler to guarantee that the Master Role configurations are being upheld during data exchange.

The fourth and final milestone is not implemented in the sense where data can be exchanged by following the conditions required for the Connection State. This is because the PoC needs to support the complete functionality of the Master role. The backbone logic of handling a connection is absent from this implementation. This means that, even though the system can correctly transmit a response when instructed to do so, it does not support the connection configurations to preserve the link.

This project has made the first move to conquer the last frontier. The feasibility of using the SDR with the high-speed protocol BLE has been proven. The POC should, from this point, be further expanded upon to realize a full BLE implementation.

9 Future work

Due to work done during this project, a PoC with the ability to communicate within the specified time constraints and relevant functionality of lower-level BLE has been created. Additionally, a large part of the functionality has been implemented without the logic needed to uphold and make connections. It is now worth thinking about where to go from here. There are still unanswered questions and much room for more BLE functionality implementation. These will be described in this final section of the report.

The PoC has some areas that should be investigated further. A lot of the current behavior that is not understood is related to the SDR. The fact that the SDR functions with greater stability when under strain is something that should not be the case. The relationship between the performance of the SDR and altering the different parameters also needs to be investigated. This could be useful because the SDR could be used with lower sampling rates to make less noise in the environment around the PoC. As was mentioned in the discussion, the functions in the SDR were not accessible in full, as they were treated as a black box system. This means that investigation of the SDR would have to be performed by its manufacturers, or the source code would have to be made available for others to investigate. Something that should also be looked into is introducing proper frequency shifting instead of being reliant on using a high sampling rate combined with I/Q sample frequency shifting. A method to be investigated for solving the problem of frequency shifting is pre-calibration of transmission and reception frequency. Pre-calibration should make the PoC able to frequency shift at a pace that can function with BLE. Another place where performance concerning the SDR is important is CPU usage. It was necessary to change out the equipment of the PoC to use a computer with more cores as the load was too much for the previous one. Some of this load comes from the implementation of the demodulation and TX thread as they both execute a `while(1)` loop even if they are doing nothing. It needs to be investigated further what causes this load beyond the while-loops, and if these issues can be diminished so that the PoC can work on less powerful devices making it more versatile. Lastly, some functionalities have been prepared which need further testing. An example of this is establishing a full connection. In this example, the framework was built, but getting another device set up was not tested due to problems with getting another device operational.

Beyond looking at the unconventional parts of the PoC something else to look at would be to expand the functionality of the PoC. Currently, the BLE implementation only reaches the lower layers, meaning there still is a lot of BLE functionalities left to implement. One big thing is to introduce a link-layer manager that can aid the PoC to connect to other real-world devices in a more automated manner and generally offer the standard connection setup of the protocol. This means that the link-layer functionalities should be implemented to get the state machine working to use the different states such as scanner, advertiser, and connecting to uphold a connection with another device. This also means that the master-slave functionality should be implemented in full to give the PoC the ability to communicate correctly. Related to this is the fact that the full capabilities of

the BLE library should be explored and investigated to see if all the different frame types are created and received correctly. Testing this requires that the master-slave relationship gets implemented in full. Beyond that, the higher layers should also be investigated and implemented, getting the PoC from concept to a marketable device that could be used in conjunction with networks that utilize BLE devices. Something else that could be added to the SDR is an API to make it easier to interact with the SDR. The TX handler is somewhat doing this already, meaning there is a starting point for implementing such an API. Also, there would be a point in adding some sort of user interface to the PoC to allow for customizing the configuration to change factors such as the frequencies being used and the frame/access address being looked for.

References

[1] Glenn J. Bradford. "A FRAMEWORK FOR IMPLEMENTATION AND EVALUATION OF COOPERATIVE DIVERSITY IN SOFTWARE-DEFINED RADIO". In: (). URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.218.5199&rep=rep1&type=pdf.

[2] Gerry Corley Ronan Farrell Magdalena Sanchez. "Software-Defined Radio Demonstrators: An Example and Future Trends". In: (). URL: https://downloads.hindawi.com/journals/ijdmb/2009/547650.pdf.

[3] *Bluetooth Core Specification*. 4.2. Download the 4.2 version pdf. Bluetooth SIG working group. URL: https://www.bluetooth.com/specifications/archived-specifications/.

[4] wlanpedia. "Interframe Space". In: (). URL: https://en.wikipedia.org/wiki/Short_Interframe_Space.

[5] Saet-Byeol Yu Hyoseok Yoon Ji-Eun Lee and Se-Ho Park. "Bluetooth-enabled Software Defined Radio Platform". In: (). URL: https://www.researchgate.net/publication/345717501_Bluetooth-enabled_Software_Defined_Radio_Platform.

[6] Se-Ho Park Hyoseok Yoon Saet-Byeol Yu. "Software Defined Radio Controller Using Bluetooth". In: (). URL: https://www.researchgate.net/publication/327063552_Software_Defined_Radio_Controller_Using_Bluetooth.

[7] Guevara Noubir Tien Dang Vo-Huu Triet Dang Vo-Huu. "Fingerprinting Wi-Fi Devices Using Software Defined Radios". In: (). URL: https://dl.acm.org/doi/10.1145/2939918.2939936.

[8] Petar Knežević Igor Vitas Dina Šimunić. "Evaluation of Software Defined Radio systems for smart home environments". In: (). URL: https://ieeexplore.ieee.org/document/7160335.

[9] Konstantin Chomu Raúl Chávez-Santiago Aleksandra Mateska and Ilangko Balasingham Liljana Gavrilovska. "Applications of Software-Defined Radio (SDR) Technology in Hospital Environments". In: (). URL: https://www.researchgate.net/publication/257600976_Applications_of_software-defined_radio_SDR_technology_in_hospital_environments.

[10] N/A. *Analog communication - Modulation*. Last visited: Jan 21, 2021. URL: https://www.tutorialspoint.com/analog_communication/analog_communication_modulation.htm.

[11] S. Gerez. "Implementation of Digital Signal Processing : Some Background on GFSK Modulation". In: 2013. URL: https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwi1sJPfsNzuAhUENuwKHd3zAVAQFjAAegQIARAC&url=http%3A%2F%2Fsabihgerez.com%2Fut%2Fsendfile%2Fsendfile.php%2Fgfsk-intro.pdf%3Fsendfile%3Dgfsk-intro.pdf&usg=AOvVaw0L73-a2gMzbzmkeetguscm.

[12] Liberg Olof et al. “Cellular internet of things. Technologies, Standards, and Performance”. In: 2018. Chap. 9, pp. 327–360. ISBN: 978-0-12-812458-1. Academic Press.

[13] Tomislav MATIĆ Tomislav ŠVEDEK Marijan HERCEG. *A simple signal shaper for GMSK/GFSK and MSK modulator based on sigma-delta look-up table*. Last visited: Jan 21, 2021. URL: https://www.researchgate.net/publication/26625015_A_simple_signal_shaper_for_GMSKGFSK_and_MSK_modulator_based_on_sigma-delta_look-up_table.

[14] JiaoXianjun. “BTLE”. In: (). URL: https://github.com/JiaoXianjun/BTLE.

[15] pycom. *pycom$_l$opyv*4. URL: https://pycom.io/product/lopy4/.

[16] pycom. *pycom$_e$xp$_b$oardv*3.1. URL: https://docs.pycom.io/datasheets/expansionboards/expansion3/.

[17] Joseph D. Gaeddert. *liquid-dsp*. URL: https://github.com/jgaeddert/liquid-dsp/.

10 Appendix

This section contain different parts of the report that diverted too much away from the points of interest. It is therefore thought to be put in here if a reader wishes further information on an aspect of the report.
Different directions pursued before reaching the final solution are also presented as well as extra tables and figures that would be too much in the actual report, but may be of interest.

11 SDR-tools

Beyond the previously mentioned tools for the SDR in section2.1.3 2 programs were also presented which were used in combination with the library chosen instead of self implementation for validation purposes. In order to verify that the SDR example code was being utilized and altered correctly a couple of tools were present alongside the SDR example code that could verify this. These are 2 executable files, SDR_play and spectrum. The former could take a file with I/Q samples and transmit them according to a set of setting such as sampling rate and frequency and the amount of times the file should be played. The ladder worked by listening for at least 1 second and writing all I/Q samples to a file for the specified period. This one also took inputs such as sampling rate and frequency.

These two were used as a way of authenticating the validity of I/Q samples that were generated by the BTLE library to ensure that no error was made in the SDR example code meaning potential errors could be isolated to belonging to either the changes made in SDR example code, or to how the BTLE library generated the samples. For usage of spectrum it was used for validating the reception to ensure that too was configured as it should i.e. if receiving from the generator, the same should be seen on the analyzer if sent after recording with either spectrum or sdr example code.

Early on the combination of these were used such that the SDR received a packet from the signal generator, using spectrum. Afterwards the file that was outputted was played using sdr_play. The goal was to make sure that the signal analyzer saw the same content from both the generator, and from the SDR, which it also did. This was done as a "sanity" check when trouble presented itself demodulating the signal on the analyzer from the samples generated by the BTLE library, and seeing as this combination worked it meant the issue could only be something related to how I/Q samples were generated. This naturally allowed the focus to be shifted towards what was being done wrong, which at the time turned out to be not using data whitening, as that had been overlooked.

11.1 Liquid library

Initially while looking for a possible library that good perform part of the necessary processing required to make I/Q samples this library [17] was examined.

It employed Gaussian Minimum Frequency shifting modulation and demodulation techniques which could be used to realize the digital modulation and demodulation. It meant that crafting the frame to be sent on a binary level still had to be implemented as well as the data whitening.
The library was found to work correctly and could also create samples correctly according to the required standard when the BLE frames were produced correctly and given for modulation.

It was originally planned to be used and then build the BLE logic and functionalities on top. Because the BTLE library was found as well and after initial investigation it was found to be superior as it had a large amount of the BLE functions available already. Thus this direction was not pursued any further, so as to not reinvent the wheel. It is still worth noting that this was one of the initial directions the project took, when only modulation and demodulation was intended to be outsourced to a third party library.

11.2 Gaussian filters

Index	Rounded value
-4	2
-3	11
-2	32
-1	53
0	60
1	53
2	32
3	11
4	2

Table 15: 4MHz filter coefficients

Originally filter used from BTLE, for 4 samples per symbol, is presented on table 15.

Index	Rounded value
-8	1
-7	3
-6	5
-5	9
-4	14
-3	21
-2	27
-1	31
0	33
1	31
2	27

3	21
4	14
5	9
6	5
7	3
8	1
9	1

Table 16: 8MHz filter coefficients

The filter coefficients for 8 samples per symbol are presented on table 16.

Index	Rounded value
-15	1
-14	1
-13	2
-12	3
-11	4
-10	5
-9	6
-8	7
-7	9
-6	10
-5	12
-4	13
-3	15
-2	16
-1	16
0	16
1	16
2	16
3	15
4	13
5	12
6	10
7	9
8	7
9	6
10	5
11	4
12	3
13	2
14	1
15	1

Table 17: 16MHz filter coefficients

The filter coefficients for 16 samples per symbol are presented on table 17.

Index	Rounded value
-29	1
-28	1
-27	1
-26	1
-25	1
-24	1
-23	2
-22	2
-21	2
-20	2
-19	3
-18	3
-17	3
-16	4
-15	5
-14	5
-13	5
-12	5
-11	6
-10	6
-9	6
-8	7
-7	7
-6	7
-5	8
-4	8
-3	8
-2	8
-1	8
0	8
1	8
2	8
3	8
4	8
5	8
6	7
7	7
8	7

9	6
10	6
11	6
12	5
13	5
14	5
15	5
16	4
17	3
18	3
19	3
20	2
21	2
22	2
23	2
24	1
25	1
26	1
27	1
28	1
29	1

Table 18: 32MHz filter coefficients

The filter coefficients for 32 samples per symbol are presented on table 18.

Index	Rounded value
-36	1
-35	1
-34	1
-33	1
-32	1
-31	1
-30	1
-29	1
-28	1
-27	1
-26	2
-25	2
-24	2
-23	2
-22	2
-21	3
-20	3

-19	3
-18	3
-17	4
-16	4
-15	4
-14	4
-13	5
-12	5
-11	5
-10	5
-9	6
-8	6
-7	6
-6	6
-5	6
-4	6
-3	7
-2	7
-1	7
0	7
1	7
2	7
3	7
4	6
5	6
6	6
7	6
8	6
9	6
10	5
11	5
12	5
13	5
14	4
15	4
16	4
17	4
18	3
19	3
20	3
21	3
22	2
23	2
24	2

25	2
26	2
27	1
28	1
29	1
30	1
31	1
32	1
33	1
34	1
35	1
36	1

Table 19: 40MHz filter coefficients

The filter coefficients for 40 samples per symbol are presented on table 19.

A common denominator for all of these tables and their coefficients is that they all accumulate to the value 256. This has no inherent meaning as it is implementation specific. It is related to 90 degrees in the BTLE library with its predefined tables of cosine and sinus values. Hence if a different sized predefined table were to be used, for a better resolution on Cosine and Sinus values, the filter coefficients on the tables would need to be scaled alongside.

11.3 msdr_read distributions

To ensure the non log distribution are also visible if of interest these are shown here. They are split into section of interest as singular peaks may cause others appear non-distinguishable even if they are.

Nosleep distribution

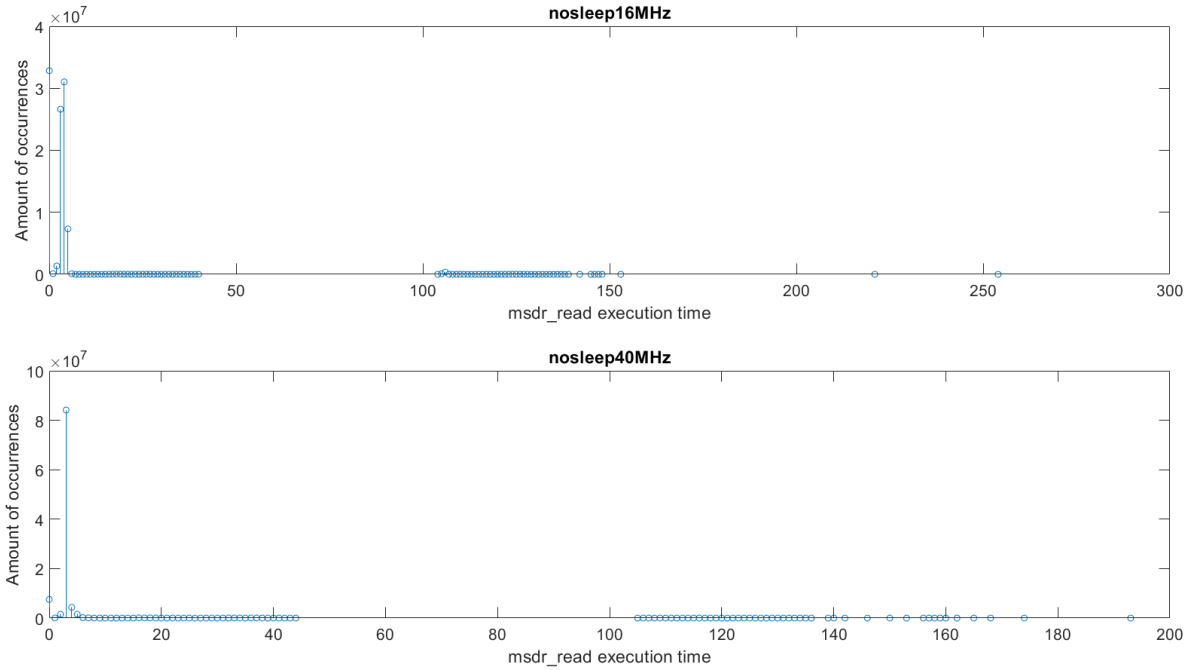

Figure 11.1: nosleep, normal view

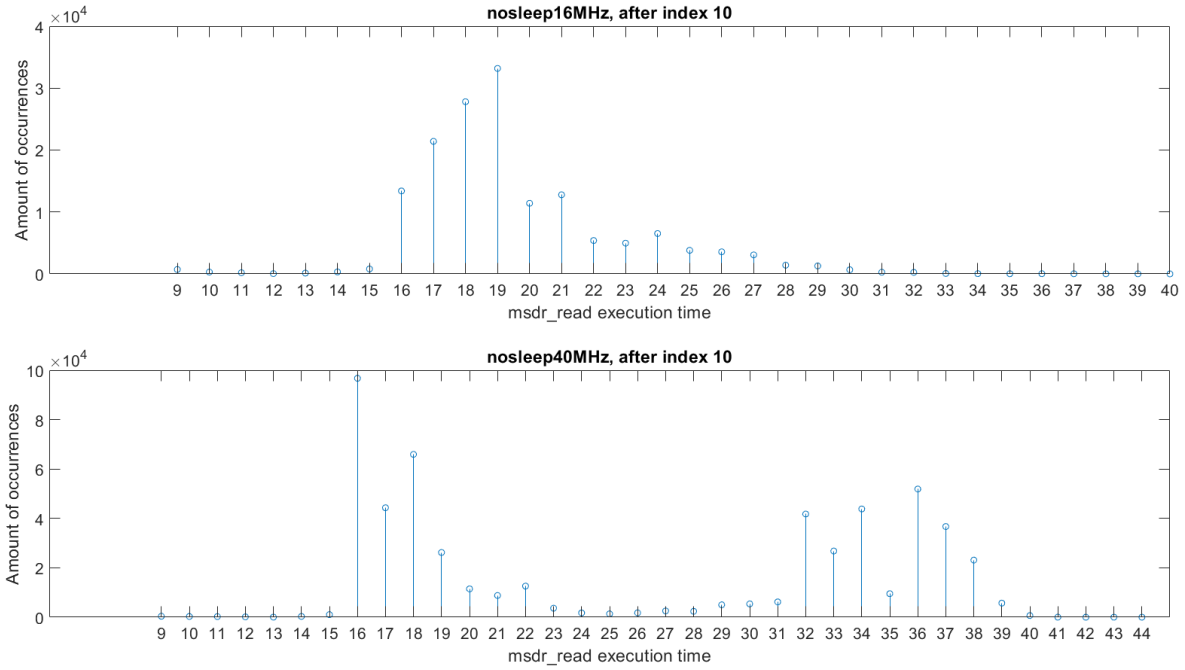

Figure 11.2: nosleep, normal view after 10

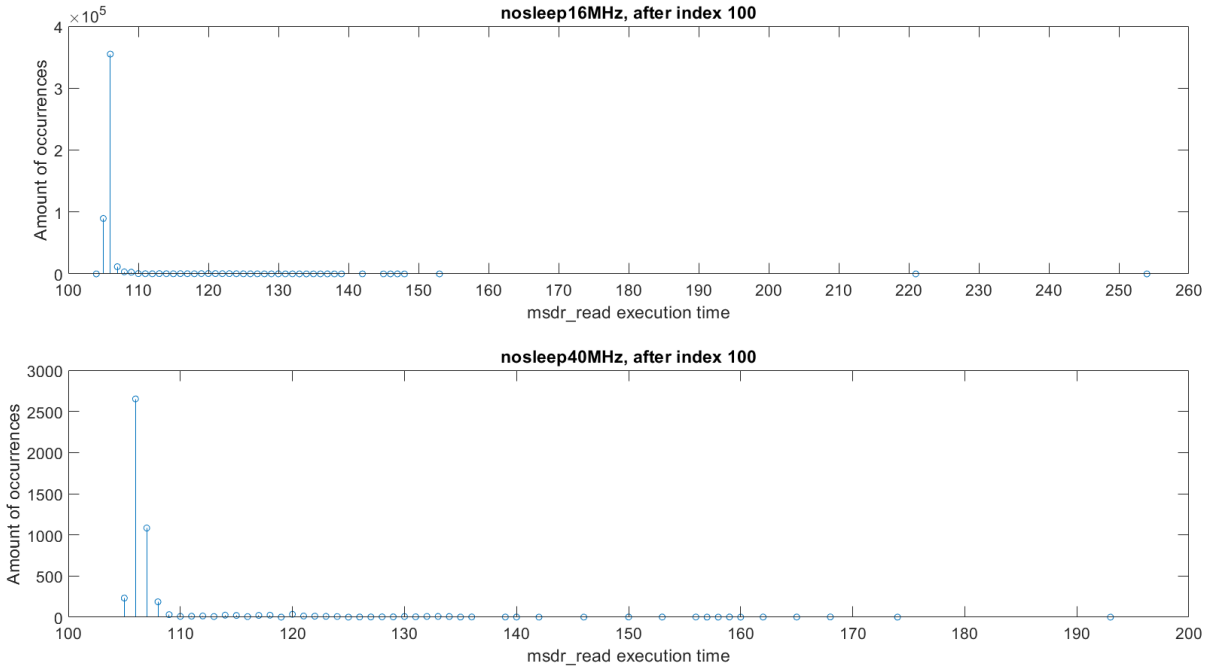

Figure 11.3: sleep, normal view, after 100

Main points of interest for this purpose is the difference spotted running at these 2 sampling rates. This is due to the fact they exhibit different behaviour in all 3 different figures shown. Generally speaking however the 16MHz seems after worse than 40Mhz due to the excessive, in comparison, appearance of msdr_read execution at $100\mu s$ or higher. This is mainly highlighted by the fact that at 16MHz at $105\mu s$ more than 300000 occurrences were spotted, while at the similar area from 104-110 for 40MHz this is short of even 500 occurrences.
Executions at the 100 mark and higher are especially dangerous as the chance to transmit samples in time becomes impossible.

μsleep distribution

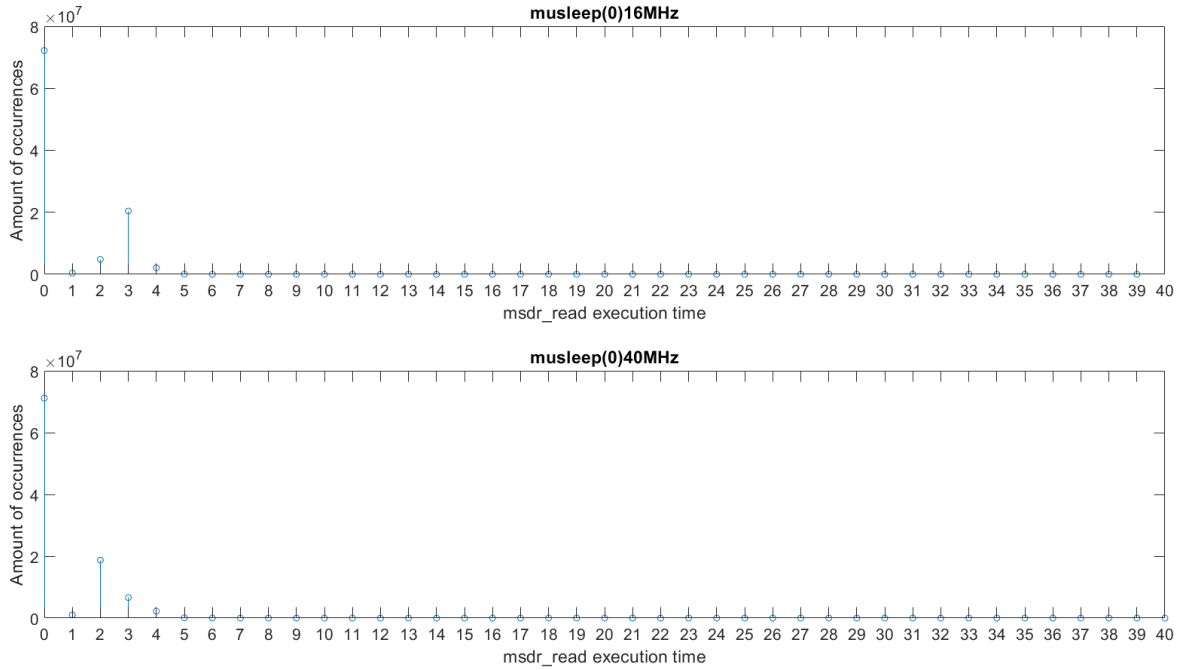

Figure 11.4: usleep, normal view

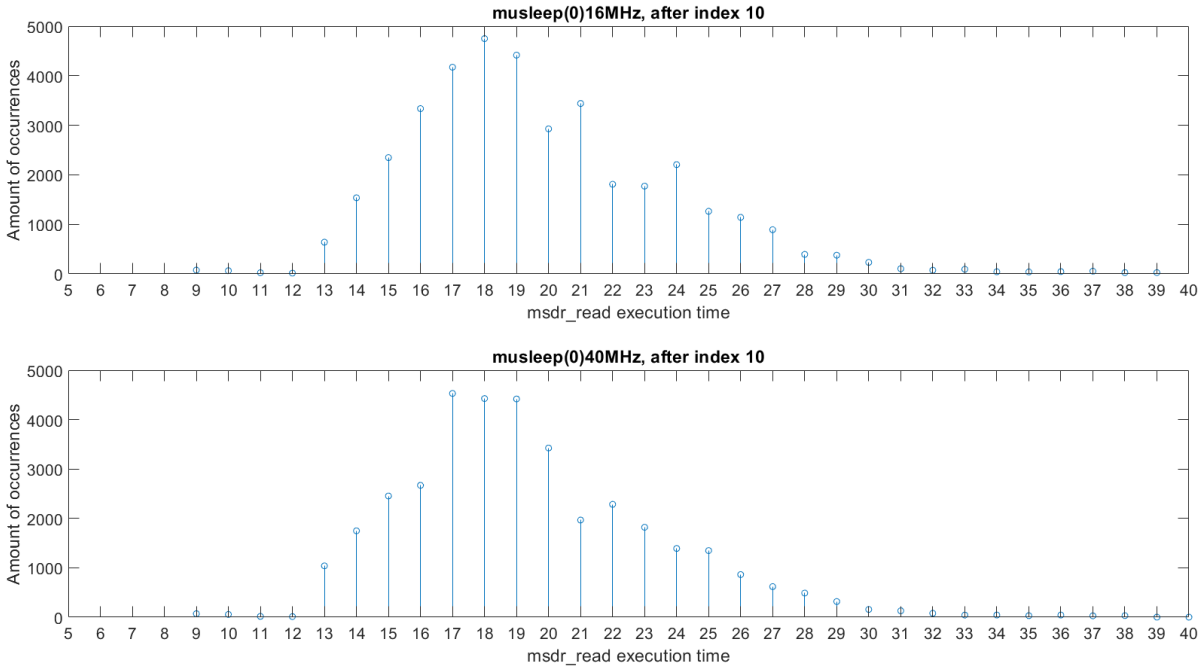

Figure 11.5: usleep, normal view after 10

Nanosleep distribution

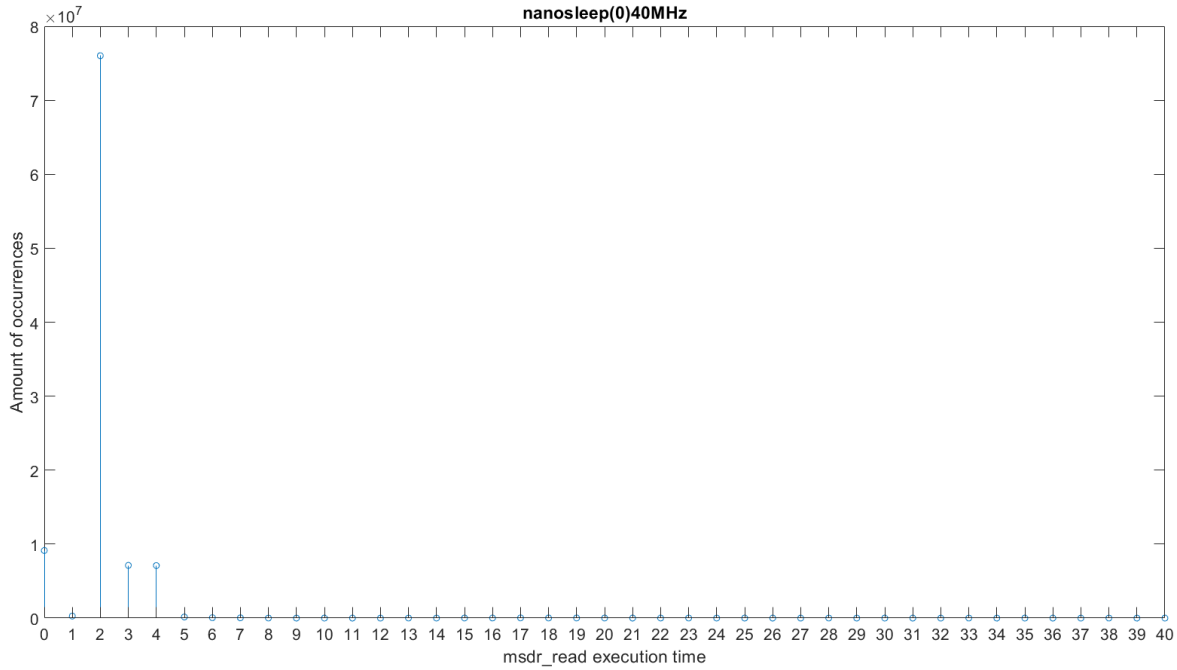

Figure 11.6: nanosleep, normal view

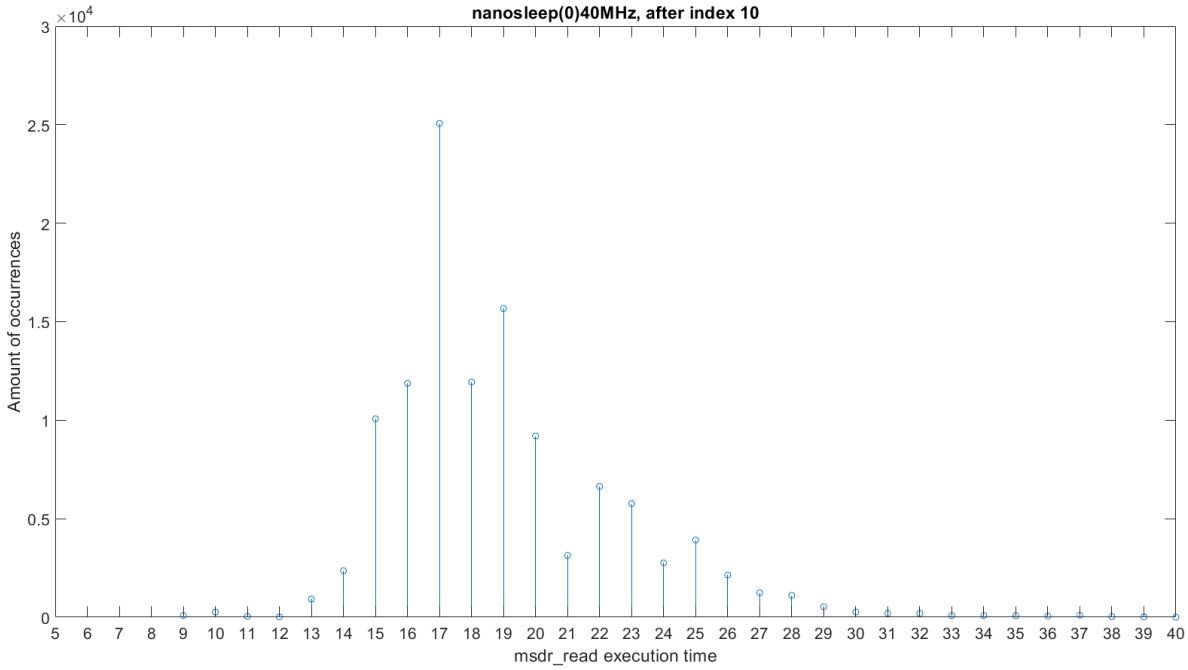

Figure 11.7: nanosleep, normal view after 10

11.4 Test data

1. General Setting	
Channel Type	Advertising
Packet Type	ADV_IND
Channel Index	37
Access Address	8E89BED6
CRC Preload	555555
Data Whitening	On
Idle Interval	249 us
Packet Length	168 us
2.PDU Header Setting	
Packet Type	00
TxAdd	0
3.PDU Length Setting	
Length (Octets)	11
4.PDU Payload Setting	
Advertiser Address (AdvA)	000000000008 [Hex]
Payload Data	40 bits binary data [11111111...]
Data Length (Octets)	5

Figure 11.8: ADV_IND definition.

1. General Setting	
Channel Type	Advertising
Packet Type	ADV_DIRECT_IND
Channel Index	37
Access Address	8E89BED6
CRC Preload	555555
Data Whitening	On
Idle Interval	449 us
Packet Length	176 us
2.PDU Header Setting	
Packet Type	01
TxAdd	0
RxAdd	0
3.PDU Length Setting	
Length (Octets)	12
4.PDU Payload Setting	
Advertiser Address (AdvA)	000000000008 [Hex]
Initiator Address (InitA)	000000000008 [Hex]

Figure 11.9: ADV_DIRECT_IND definition.

1. General Setting	
Channel Type	Data
Packet Type	**LL_DATA**
Channel Index	0
Access Address	8E89BED6
CRC Preload	555555
Data Whitening	**On**
Idle Interval	**505 us**
Packet Length	120 us
2.PDU Header Setting	
LLID	1
NESN	0
SN	**1**
MD	**1**
3.PDU Length Setting	
Length (Octets)	**5**
4.PDU Payload Setting	
Payload Distribution	Single Packet
Payload Data	**40 bits binary data [11111111...]**
Data Continuous	On
Data Repetition	1
Data Length	40
Number Of Full Packets	1
Number Of Partial Packets	0
Number Of Padding Packets	0

Figure 11.10: LL_DATA definition.

1. General Setting	
Channel Type	Data
Packet Type	**LL_FEATURE_REQ**
Channel Index	0
Access Address	8E89BED6
CRC Preload	555555
Data Whitening	**On**
Idle Interval	**473 us**
Packet Length	152 us
2.PDU Header Setting	
LLID	3
NESN	0
SN	**1**
MD	**1**
3.PDU Length Setting	
Length (Octets)	9
4.PDU Payload Setting	
Opcode	0x08
Feature Set	**0000000000000000**

Figure 11.11: LL_FEATURE_REQ definition.

1. General Setting	
Channel Type	Data
Packet Type	LL_FEATURE_RSP
Channel Index	0
Access Address	8E89BED6
CRC Preload	555555
Data Whitening	On
Idle Interval	473 us
Packet Length	152 us
2.PDU Header Setting	
LLID	3
NESN	0
SN	1
MD	1
3.PDU Length Setting	
Length (Octets)	9
4.PDU Payload Setting	
Opcode	0x09
Feature Set	0000000000000000

Figure 11.12: LL_FEATURE_RSP definition.

1. General Setting	
Channel Type	Data
Packet Type	LL_VERSION_IND
Channel Index	0
Access Address	8E89BED6
CRC Preload	555555
Data Whitening	On
Idle Interval	497 us
Packet Length	128 us
2.PDU Header Setting	
LLID	3
NESN	0
SN	1
MD	1
3.PDU Length Setting	
Length (Octets)	6
4.PDU Payload Setting	
Opcode	0x0C
Version Number (VersNr)	08
Company Identifier (CompId)	0000
Sub-version Number (SubVersNr)	0000

Figure 11.13: LL_VERSION_IND definition.

1. General Setting	
Channel Type	Advertising
Packet Type	**CONNECT_REQ**
Channel Index	**37**
Access Address	8E89BED6
CRC Preload	555555
Data Whitening	**On**
Idle Interval	**273 us**
Packet Length	**352 us**
2.PDU Header Setting	
Packet Type	05
TxAdd	0
RxAdd	0
3.PDU Length Setting	
Length (Octets)	34
4.PDU Payload Setting	
Initiator Address (InitA)	000000000008 [Hex]
Advertiser Address (AdvA)	000000000008 [Hex]
LL Connection Access Address (AA)	00000000
CRC Initialization (CRCInit)	555555
WinSize	1
WinOffset	0
Interval	6
Latency	0
Timeout	10
Channel Map (ChM)	**000007FFFF**
Hop Length (Hop)	5
Sleep Clock Accuracy (SCA)	500 ppm

Figure 11.14: CONNECT_REQ definition.