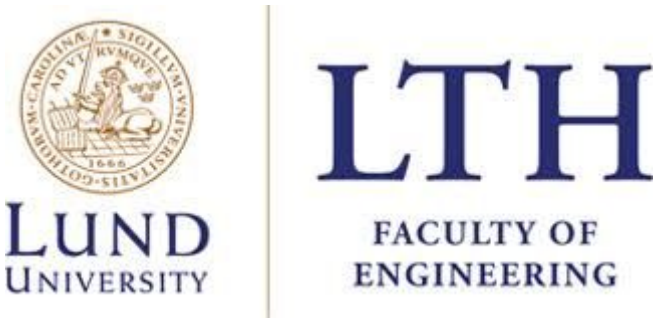
Millimeter-Wave Radar data processing

Jonas Tran

2020

Master's thesis in Electrical Measurements

Supervisor: Nebojsa Malesevic



Department of Biomedical Engineering

Abstract

Based on the feasibility study [4], the goal of this master thesis is to compile custom firmware for the XM112 Pulsed Coherent Radar Module, which features a state of the art radar sensor integrated with a microcontroller unit (MCU). This allows the use of custom firmware to check the plausibility of real-time data processing on the radar signals and will lay some of the groundwork for further development of detection algorithms.

In the feasibility study, machine learning was used in the implementation of real-time detection of wrong foot placements while walking on stairs with impressive results like less than 1 percent false positives.

The feasibility study on fall prevention used pre-recorded data which was transmitted over long wires to a system of two computers linked with wifi that handled the algorithms and data processing. Being able to process data directly on the MCU will improve upon some of the redundant setup that was used on the feasibility study.

The XM112 radar, which is used in this thesis, uses mm-waves which operate in the 30-300 GHz interval. The mm-wave technology works without a need for physical contact with the measured object and can provide information about the velocity, range and angle. The signals can pass through certain materials like clothing, plastic, and they are not affected by weather. Due to the short wavelengths used the antenna is also very small, resulting in a system which is very portable and can be fitted almost anywhere.[1]

This thesis will implement firmware based on the provided radar system software (RSS) from the manufacturer. The firmware will contain a selection protocol using serial communication and real-time data processing directly on the XM112 Microcontroller Unit.

Acknowledgments

This master thesis was worked on in summer 2020 at the department of Biomedical Engineering at Lund University.

I’d like to thank my supervisor Nebojsa for his patience in answering all my questions as well as for all the help he gave, without him this work would not be possible.

I’d also like to thank Anton Martinsen at Acconeer who provided support and technical help with the Acconeer source code.

Table of contents

Abstract 1

Acknowledgments 1

Table of contents 2

1. Introduction 3

1.1 Background 3

1.2 Motivation 4

1.3 Abbreviations 5

1.4 Objectives 5

1.5 Equipment 5

2. Theory 5

2.1 Radar basics 6

2.1.1 Coherency in radar 6

2.1.2 Pulsed radar 7

2.2 Module information 7

2.2.1 A111 radar sensor 7

2.2.2 XM112 high performance module 7

2.2.3 XB112 Breakout board 8

2.2.4 Acconeer A111 sensor miscellanous 9

2.3 Acconeer radar services 9

2.4 Acconeer radar detectors 10

2.5 HAL and HPL 11

2.6 UART protocol 12

2.7 A111 Radar parameters 13

3. Method 13

3.1 Preparing the XM112 module 13

3.2 Setting up a new project 14

3.3 Modifying the project 14

3.4 Increasing flash and heap size 14

3.5 Flashing XM112 with custom firmware 15

3.6 Using Termite to communicate via Serial interface 16

3.7 Validation test 16

4. Results 19

4.1 Added custom functions and code 19

4.1.1 Main.c file 20

4.1.2 CustomEnvelope.c file 20

4.2 Test run of termite with parameters 20

4.3 Validation test 22
4.3.1 Validation test protocol one 22
4.3.2 Validation test protocol two 23
4.3.3 Moving targets - validation test protocol one 24
4.3.4 Moving targets - validation test protocol two 27
4.4 Pie charts 30

5. Discussion and conclusions 33
5.1 Goals and results 33
5.2 Comparison to feasibility study 33
5.3 Selection Protocol 33
5.4 Validation test 34

6. References 36

7. Appendices 37
A) Technical specifications of XM112 37
B) System overview: 37
C) User guides (acconeer) 38
D) User Guide (Selection Protocol) 38
E) Atmel studios 7 code 38

1. Introduction

1.1 Background

During the second world war there was massive research and development in radar technology for military use, like aircraft, vehicle and missile detection. At the end of the war, the techniques also found its uses outside the military. Modern radars have uses in for example material/object identification, weapon systems, security, automotive industry, deep space, and in medicine, where they can be used to affect cell growth, enzyme activity, peripheral receptors etc. [11] [19]

Old radar systems used to be big, stationary, expensive with limited accuracy. Over time improvements in size, costs, efficiency etc. has made it a very popular technology with a bright future. [11]

Development of millimeter-wave (mm-wave) communications systems has seen a big increase in recent times partly due to the capacity requirement of the next generation network, 5G. The possibility to use the mm-wave radars in embedded systems can lead to cheap, high performance products in expanding markets, like the IoT industry.

While there are many advantages the mm-wave radar also has drawbacks like a limited range, interference from other electromagnetic hotspots and oversensitivity. Therefore some challenges still remains to be tackled. [10]

Some possible future applications of this master thesis are object/fall/obstacle detection, which can be used for fall prevention for elderly people. By focusing on preventive measures the risks for the elderly populations will be lowered and the strain on emergency departments will go down, which could possibly save lives.[2]

This work will be focusing on the a mm-wave radar sensor, but there are also other possible ways of obstacle detection which can be implemented i.e ultrasonic radar, infrared sensor or using a camera with a line-laser system [3].

1.2 Motivation

This section will provide a very brief summarization of the feasibility study that is the precursor to this thesis. It is strongly recommended to read the feasibility study to get an understanding of the motivations for this thesis.

For elderly people, a fall can have drastic consequences. Around 10% of all the deaths for elderly people are caused by falls on stairs. The number of deaths due to stair falls are rising each year and the fatality rate increases exponentially with age. Since most preventive measures are environmental or passive (handrails, non-slippery steps, ergonomic shoes, canes, etc) research was done to check the feasibility of preventing falls using active miniaturized radar technology. [4]

The idea of the research is to use signal processing algorithms on the mm-wave radar data to detect imminent falls and provide sensory feedback to the user. Based on the sensory feedback the user can correct a dangerous step about to be made. The mm-wave radar is supposed to be parallel to the shoe, actively measuring the distance to the floor.

In the study the mm-wave radar was connected to a Raspberry Pi which is a tiny computer. The Raspberry Pi mainly acted as a transmitter (through wifi protocol) to another computer that handles the data processing part which uses machine learning algorithms.

Three major scenarios were considered during data recording; normal walking, placing foot over one stair step and placing foot over two stair steps.

Results obtained in the research were very positive and encouraging. Foot overshooting was easily detected by the machine learning algorithm.

1.3 Abbreviations

MCU - microcontroller unit
RSS - radar system software
SDK - software development kit
GPIO - general purpose input output
UART - universal asynchronous receiver/transmitter
USART - universal synchronous/asynchronous receiver/transmitter
SPI - serial peripheral interface
USB - universal serial bus
HWAAS - hardware accelerated average samples
HAL - hardware abstraction layer
HRI - hardware register interface
HPL - hardware proxy layer
PCR - Pulsed Coherent Radar
PRF - Pulse Repetition Frequency

1.4 Objectives

The objectives and milestones of the project will roughly be:

1) Getting familiar with the XM112 product and the capabilities of the RSS
2) Setting up a new project in Atmel Studio 7 and integrating source files and code
3) Development, implementation and testing of custom code/firmware
4) Processing data and checking if the results are valid and satisfying
5) Creating a selection protocol for desired behaviour
6) Verification tests

1.5 Equipment

Hardware: XM112 radar sensor module, XB112 breakout board, A111 mm-wave radar, micro-USB cable, Windows 10 computer,

Software: Termite, Atmel Studio 7, BOSSA, Labview, Matlab

2. Theory

Some basic radar information will be presented in this section and some of the terminology presented here will be used in later chapters. For more detailed information about the radar module it is recommended to look at appendice A and the extra resources listed at the end of the report.

2.1 Radar basics

Radars are mainly divided into continuous wave and pulsed systems. The pulsed radar systems are further divided into noncoherent and coherent types.

The basic components of radar systems are transmitter, antenna and the receiver and usually a display or/and signal processor. A signal is generated from the transmitter and radiated from the antenna. When the signal hits an object, some of it will reflect back to the antenna, before reaching the receiver part of the system. The reflection will usually go through different stages of amplification, downconversion, detection and signal processing that is decided by the system. [16] [17]

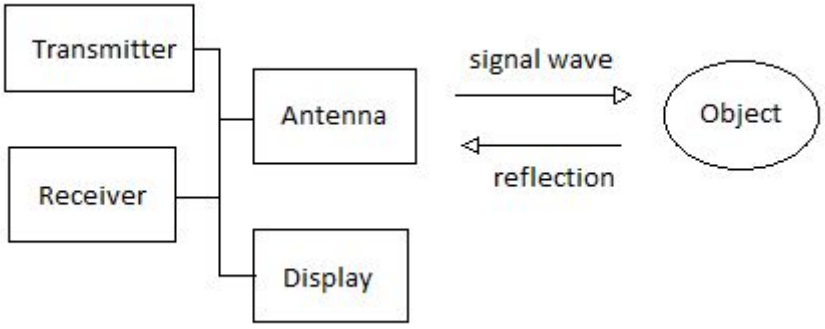


Figure 1. Simplified sketch of how a radar works.

2.1.1 Coherency in radar

The ability to distinguish targets in close proximity to each other depends on the transmitted pulse width and antenna beamwidth. A non-coherent radar system lacks the ability to measure the phase of the reflection, which is a reason why many modern radar systems demand that the system is coherent.

For a coherent radar system, a couple of stable oscillators are used to generate the transmission signal. By using stable oscillators as a reference the phase difference of the received signal can be measured. One drawback of coherent radar systems is an increased usage in power [5].

Coherency in radar systems is a description of the relationships of the phases between the transmitted and received pulses. A PCR has a constant relationship between transmitted and received pulses. If small phase shifts are detected between these the doppler frequency can be used to determine the velocity of a moving object. [12] [14] [15] [16] [18]

2.1.2 Pulsed radar

The pulsed wave radar sends short bursts of electromagnetic waves for measuring contrary to a continuous wave radar which does not pause its transmission. The pulse length is usually referred to as the pulse width and it is related to bandwidth and range resolution.

Pulse repetition frequency (PRF) should ideally be chosen to avoid doppler range ambiguities and maximize the transmitted power. A low PRF can provide long, accurate and unambiguous range measurement but the drawback is that it has doppler ambiguities. Medium PRF has medium ambiguities in both range and doppler, but compared to a low PRF it has better average transmitted power. A high PRF (also called a pulse doppler radar) has large ambiguities in range but high transmitted power and has a good clutter rejection ability, therefore it suffers less from unwanted echoes.

In a pulsed radar system, the velocity of a measured object can be found from the doppler frequency shift. [14] [15] [16] [18]

2.2 Module information

2.2.1 A111 radar sensor

The A111 sensor has a low power consumption, high accuracy and robustness. It is a short-range pulsed coherent radar. The characteristics of the sensor along with its small size makes it a very useful radar solution, with applications in motion-, obstacle detection, vital life sign monitoring, parking lot sensing etc. It is produced by Acconeer and operates in the 60 GHz ISM radio band. It is able to detect multiple objects at close proximity with a single- or continuous sweeps. The frequency of the continuous sweeps can go up to 1500 Hz. The measurement range varies between 60-2000 mm. [5]

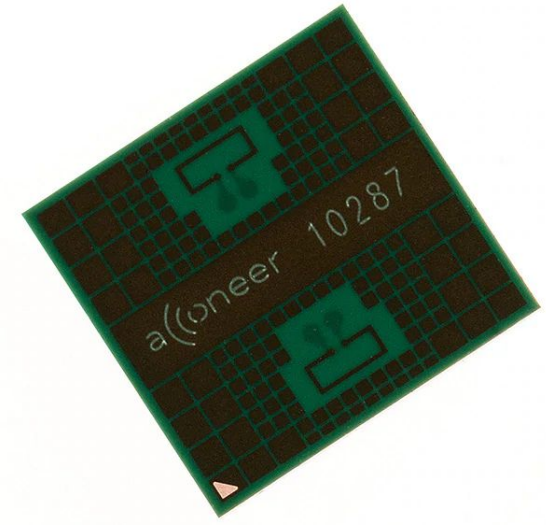


Figure 2. Picture of the A111 radar sensor from the acconeer website. [5]

2.2.2 XM112 high performance module

The XM112 module integrates the A111 radar sensor with a 32-bit Arm Cortex_M7 ATSAME70Q20A microprocessor on a printed circuit board which allows control of the module by two methods;

1) External host control via different protocols like SPI, UART or I^2C
2) Executing software directly on the microprocessor

The ATSAME70Q20A has 384 Kbytes of SRAM and 1 MB Flash memory. The maximum core processor frequency is 300 MHz. It has a 30 pin board-to-board connector to aid integration into another product. The XM112 is always delivered with the Acconeer RSS software and SDK where users can write their own applications.

Figure 3. A Picture of the XM112 module from the acconeer website. [5]

2.2.3 XB112 Breakout board

A breakout board, the XB112, is needed in order to flash and control the module externally. The breakout board features a handy USB-UART converter for easy access from an external host computer via USB which will also power the board while connected. There are also micro USB channels that allow connection and communication with an external computer. The XB112 is powered via a 5 V USB cable. [5]

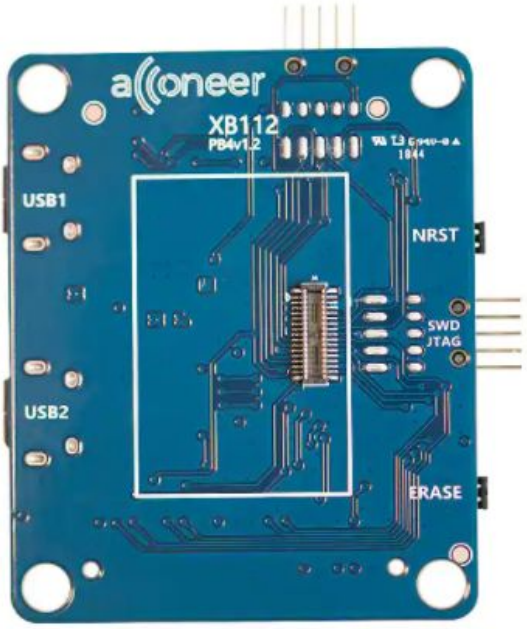


Figure 4. A Close up picture of the XB112 Breakout board. [5]

2.2.4 Acconeer A111 sensor miscellanous

The Acconeer A111 sensor is a pulsed coherent radar (PCR) which, in simple terms, calculates distance to objects by sending mm wavelength pulses (40-300 GHz) with known characteristics and measuring the elapsed time between the initial pulse and the reflection. Since we know that the mm-wave pulses travel with the speed of light, we can find the distance by multiplying the speed of light with time of flight and dividing the result by two. [5]

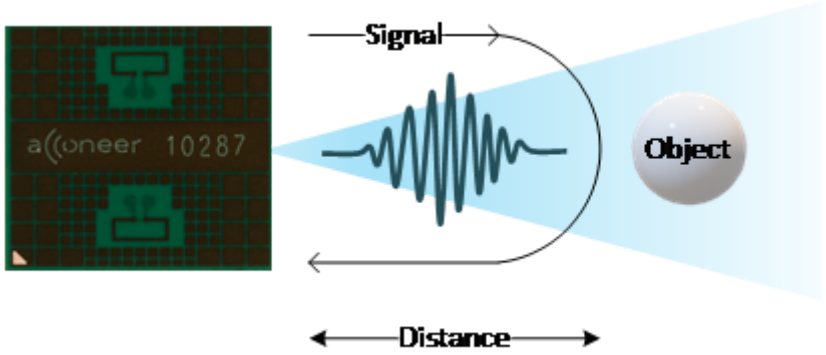


Figure 5. The A111 sensor measuring the time elapsed from an initial pulse and the reflection of an object [5]

Many of the sensor parameters like pulse length, sweep and update rate, signal gain, etc can be customized and adjusted for maximum performance in different environments.

Users have access to a library written by acconeer and can write their own application for the XM112 module. The library is written in the C-language and contains examples, references and documentation in order to help the user get acquainted with the SDK.

A good radar system takes into account the characteristics of the measured object and the surrounding environment which has an impact on the performance. Two of the most important metrics are the signal-to-interference ratio (SIR) and the resolution.
SIR is modelled as the ratio between the desired and undesired signal.
The resolution is a measurement of how well a radar can distinguish between objects at range and it is also called range resolution. [6] [7]

2.3 Acconeer radar services

There are four available Services that provide pre-processed data; **Envelope, IQ (in-phase quadrature), Power Bins** and **Sparse**. It is recommended to take a look at the characteristics of each service and carefully choose the one that is most suitable to the task at hand.

Envelope and **Power Bins** both give outputs consisting of the amplitude as a function of distance in a vector form. Power Bins have a lower quality but requires less processing power and memory. Both services measure the received energy at different distances from the radar sensor by sending pulses and evaluating the energy in the echoes by using time delays.

Power bins are mainly used by low power applications e.g. to measure large objects. It works well as a replacement for envelope mode when low complexity is acceptable. Compared to the envelope mode it has bad noise suppression, therefore it is important to make sure that the use-cases have a good SIR-ratio.

The **envelope** mode will be used and modified in this thesis. It returns a continuous stream of vectors, each consisting of over 500 data points, which will be used for the data processing part. Each data point in the vector represents the energy level at a certain distance from the radar sensor. One vector represents the gathered data of a measurement pulse. The reflection, depending on the material it reflects on, will have a small frequency change due to diffraction but in general the accuracy is very high. [5]

There is an option to use a time filter for the **envelope** mode in order to make the signal more stable. The time filter is a standard exponential smooth filter which removes high frequency noise.

IQ gives cartesian data in complex form. The cartesian data can be transformed into polar form which is useful since more accurate measurements can be made by having access to phase and amplitude. The IQ service gives information about in-phase and quadrature components and should be used when there is a need for phase information.

Sparse data is generated by sampling specific points that varies by distance (around 6 cm). It contains less data than Envelope and IQ service but also requires less processing. Due to the undersampled nature of the sparse service it is more useful at detecting moving rather than stationary targets. This makes it suitable to use together with small MCU. [5]

2.4 Acconeer radar detectors

The detectors use the unprocessed data output from the Service level as input and produces an output that can be used by the application. There are currently 4 detectors written by acconeer; distance peak, presence, distance basic and obstacle. Users are free to write their own detectors to suit their needs.

The **distance peak detector** algorithm receives the data from the envelope service and identifies the peaks by first averaging the sweeps and then comparing it with a threshold, which can be changed depending on the use case. Peaks that are too close to each other are merged. The peaks can be sorted by distance or by amplitude.

The **presence detector** takes the sparse service as input and is mainly used to detect changes over time. The algorithm consists of two main parts that detects motion between and inside frames. The two parts (inter-frame and intra-frame deviation) are used to detect slower movements between frames and faster movement inside frames. For the inter-frame detection, the mean sweep is fed through low pass filters and then based on the deviation. The intra-frame detection is based on the deviation from the mean for every frame and depth.

The **distance basic detector** returns the maximum of the envelope service data.

The **obstacle detector** is used to find the angles and distances of obstacles within the sensors detection area. A half power width beam of 60 degrees in the H-plane and 40 degrees in the E-plane is used in this mode. The detection algorithm uses fast Fourier transform (FFT) on the sweeps in order to estimate the power spectral density (PSD). [5]

2.5 HAL and HPL

The drivers are split into two layers, the HAL (hardware abstraction layer) and the HPL (hardware proxy layer). For a communication map between devices and software look at figure 6 below.

The part of the operative system that handles the communication between application and hardware is called HAL. The HPL is the part that implements the code that requires hardware access.

The RSS uses MCU functions in order to handle memory and communication with the sensor. It is used to interact with the A111 sensor. The HAL is a layer between the sensor and the RSS and implements pin configurations and drivers, these are specific to the MCU.

The HAL is implemented by the user and passed as a struct argument to an activation function in RSS. The HAL contains all the necessary information which the RSS uses in its communication with the hardware.

GPIO, delays, interrupts, IO, SPI, USART etc are implemented by Acconeer and ready for use in the program.

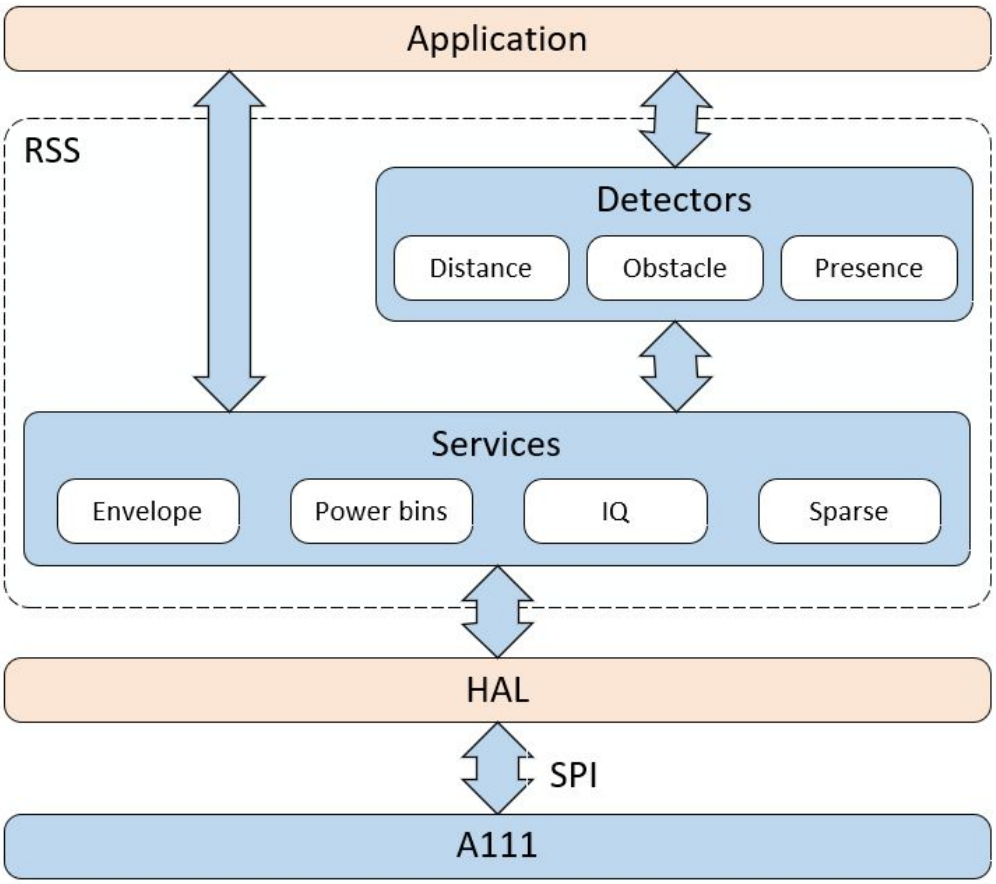


Figure 6. Device and software communication map. [5]

2.6 UART protocol

UART is a physical circuit that is used in data transfer. UART's are used to transmit data in the form of bytes by sending individual bits sequentially between two UART devices. The receiver then reconstructs each byte from the received bits.

Each UART device has a TX and RX pin which is used to transmit or receive data. Communication can be simplex, full duplex or half duplex. Simplex mode is one-way communication, half-duplex is two one-way communication that can be switched and full duplex is simultaneous two-way communication. In this thesis full duplex mode was used.

Data is transferred asynchronously at a specific baudrate where each data packet has start, stop and parity bits to help the process. [6]

Serial communication using UART:

In order to establish UART communication between the XM112 and the host computer, Termite is used. Termite is a terminal program that can send and receive bytes as hexadecimal values which is useful in this case, since most of the work was validated and checked for errors using dummy code printing messages.

2.7 A111 Radar parameters

Some important parameters to keep track of are:

Start (start length) - the starting distance for the measurement
Length (sweep length) - the distance that is measured from the start length
Update rate - the number of sweeps per second (in streaming mode)
HWAAS - radar pulse averaging amount in the sensor
Power save mode - Balancing power consumption and sweep accuracy and generation rate
Receiver gain - multiplication factor for the signal
Repetition mode - either “streaming” or “on demand” mode can be chosen where “streaming” mode feeds data at a configurable fixed rate and “on demand” mode where data is given when the application requests it

Note that the different detectors and services can have extra parameters.

*for more information check the system overview appendice

3. Method

3.1 Preparing the XM112 module

Since the XM112 is delivered non-flashed, one XB112 board is needed in order to flash the XM112. The XM112 should be attached to the XB112 board according to figure 2.

There are multiple ways to flash the module (see figure 7 below) but in this work .bin files together with BOSSA were used to flash the module. BOSSA is a software that is used for flashing Atmel’s SAM type of ARM microcontrollers. The module needs to be connected to the host computer by micro USB. USB channel 1 was used in this case. The “acc_module_server_xm112.bin” file provided by acconeer can optionally be used for a graphic representation of the sensor data and is useful to get familiar with the device. Uploading custom firmware will be done with the same method.

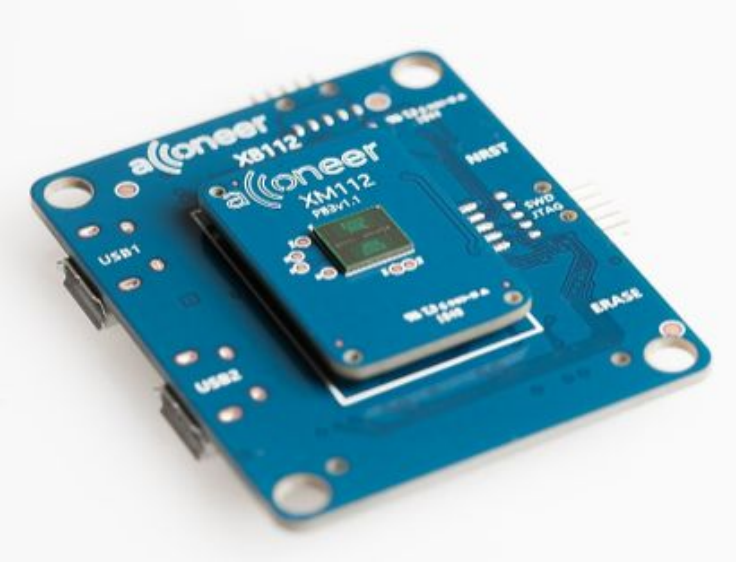


Figure 7. XM112 Radar sensor module attached to the XB112 breakout board. [5]

3.2 Setting up a new project

The integrated development platform (IDP) used for this project is Atmel Studio 7. This platform uses the C/C++ programming language for writing applications. There is also an option to use assembly code, but for this project the C language was used.

The “Atmel Studio 7 Integration Guide.pdf” is used in a new Atmel start project in order to integrate the example and reference code provided by Acconeer. See appendix c.

3.3 Modifying the project

Since the integration guide contains some errors they need to be fixed first before the example programs can be run.

A list of changes needed to be made (original code lines needs to be replaced with the suggested ones below):

1) In the driver_init.c file a pin function is wrongly defined and has to be changed to:
   gpio_set_pin_function(A111_SPI_SCK, MUX_PC24C_SPI1_SPCK);

2) Wrongly defined pins in atmel_start_pins.h needs to be redefined as:
   #define A111_SPI_SS_N GPIO(GPIO_PORTC, 25)
   #define A111_SPI_SCK GPIO(GPIO_PORTC, 24)

3) Defining the RX pin in the integration setup part 2.5, by ticking the box for RX

3.4 Increasing flash and heap size

The flash memory is a memory type that can be erased or rewritten electronically. It is non-volatile and will not lose the stored information even when the power is turned off. [20]

The heap is a part of the MCU RAM memory that is not automatically managed. Memory on the heap is allocated in the code and therefore needs to be handled by the user. [20]

In order to increase the flash and heap size, which is needed to run the example functions, the file same70q20_flash.ld needs to be edited. Any text editor software can be used for this but in this case Notepad++ was used.

The new sizes were changed from 1024 to 6400 bytes for the stack and from 512 to 20480 bytes for the heap.

The lines in the text file edited for that is:

STACK_SIZE = DEFINED(STACK_SIZE) ? STACK_SIZE : DEFINED(__stack_size__) ? __stack_size__ : 0x1900;
HEAP_SIZE = DEFINED(HEAP_SIZE) ? HEAP_SIZE : DEFINED(__heap_size__) ? __heap_size__ : 0x5000;

The example function written by Acconeer “acc_example_service_envelope.c” will be used as a base for our modified version of the function for this thesis. The implementation can be found in the result section of the report.

3.5 Flashing XM112 with custom firmware

When the written code is ready to be tested on the module it has to be built in Atmel Studio 7. It can be done by pressing f7 in Atmel Studio or pressing build -> build solution in the toolbar. The output .bin file will be found inside the debug folder which is located inside the project folder and uploaded using BOSSA (see figure 8 below).

For more details on how to upload custom firmware, see the “Getting Started Guide XM112.pdf” which can be downloaded from the Acconeer website.

Once the custom firmware is uploaded, termite is used to check the data from the sensor, via Serial interface. The printf function was used extensively during this thesis to check for errors and verify if the sensor data is correctly processed. The reason for this was that the debug function was not working correctly, which wasn’t solved by Acconeer.

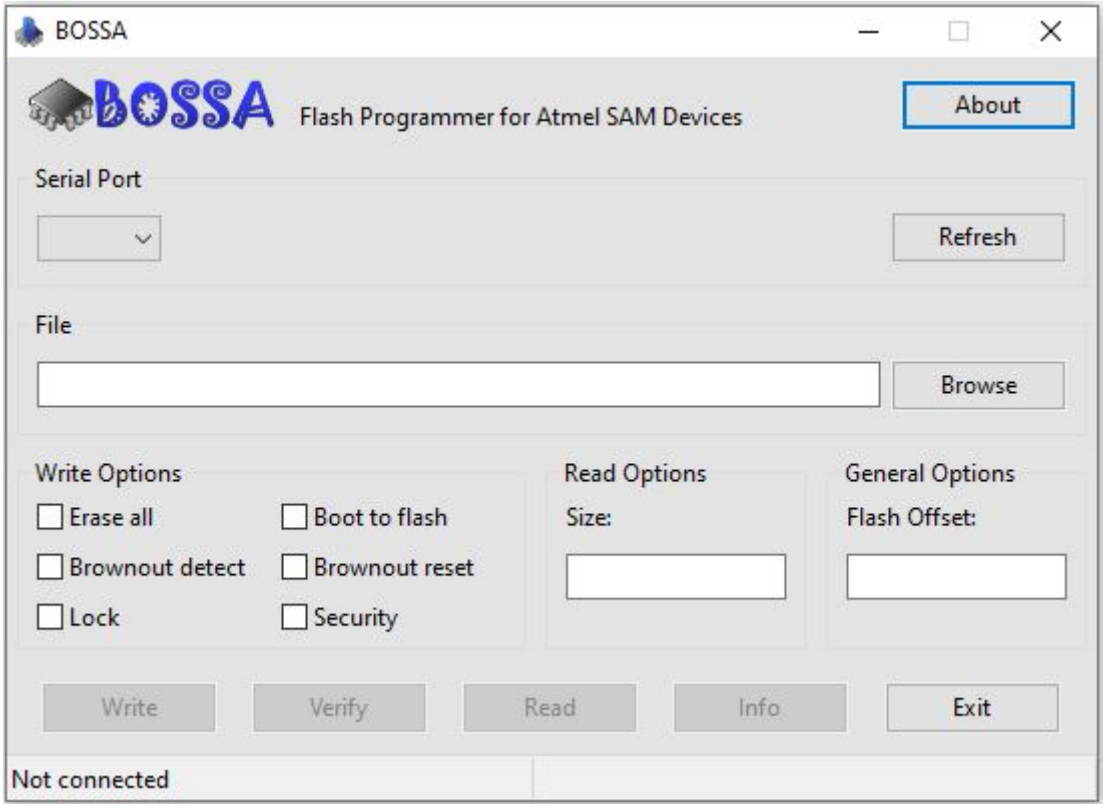


Figure 8. A picture of the BOSSA program.

3.6 Using Termite to communicate via Serial interface

Hex View needs to be enabled inside ‘Settings’ in order to send bytes in hexadecimal form, which the selection protocol is based on. Settings can be found in the toolbar and can be configured like figure 4. Each individual byte is sent with a 0x prefix together with its hexadecimal value. The baud rate used is 115200 and the connection via COM port.

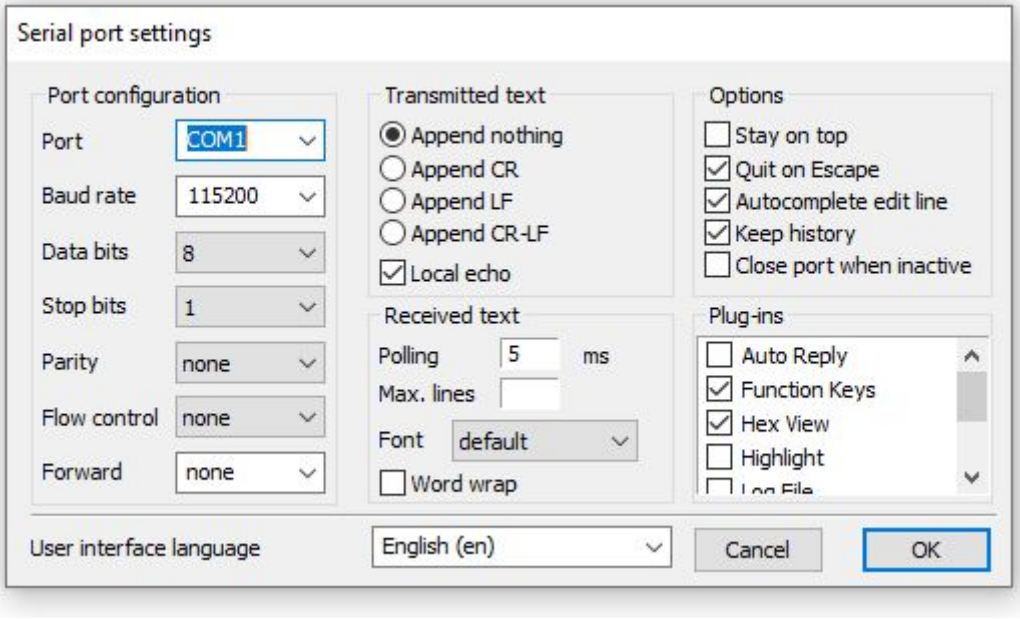


Figure 9. A picture of the settings in Termite that were used in the UART test runs.

3.7 Validation test

In order to validate that the software works as intended it was necessary to perform some validation tests and check the results for errors. Two testing protocol were defined as:

Protocol 1) Measuring the distance to a table by holding the sensor facing to the table and incrementing the distance every 10 seconds. The measurement starts at 5 cm and goes up to 75 cm and then back down to the starting position.

Distance checkpoints used in this protocol were in cm ; 5, 15, 35, 55, 65, 75.

Protocol 2) Measuring the distance from 5 to 75 and then back to the starting position in a continuous fashion by hand over 20 seconds at a steady pace.

Lastly some final tests were made using moving targets while the sensor was placed stationary on the table facing upwards.

The targets were moved according to the two test protocols and the materials used were; a 12x30 cm metal sheet, 20x30 cm plastic sheet, a concave and convex object both with 5 cm radius.

The settings for all the tests were (see section 2.7 and appendice D. for parameter information);
start length : 5 cm
sweep length : 60 cm
update rate : 20 Hz
cut point 1 : 33
cut point 2 : 66
scale band 1 : 1
scale band 2 : 1
scale band 3 : 1

Figure 10. A picture of the concave and convex objects that were used in the verification tests.

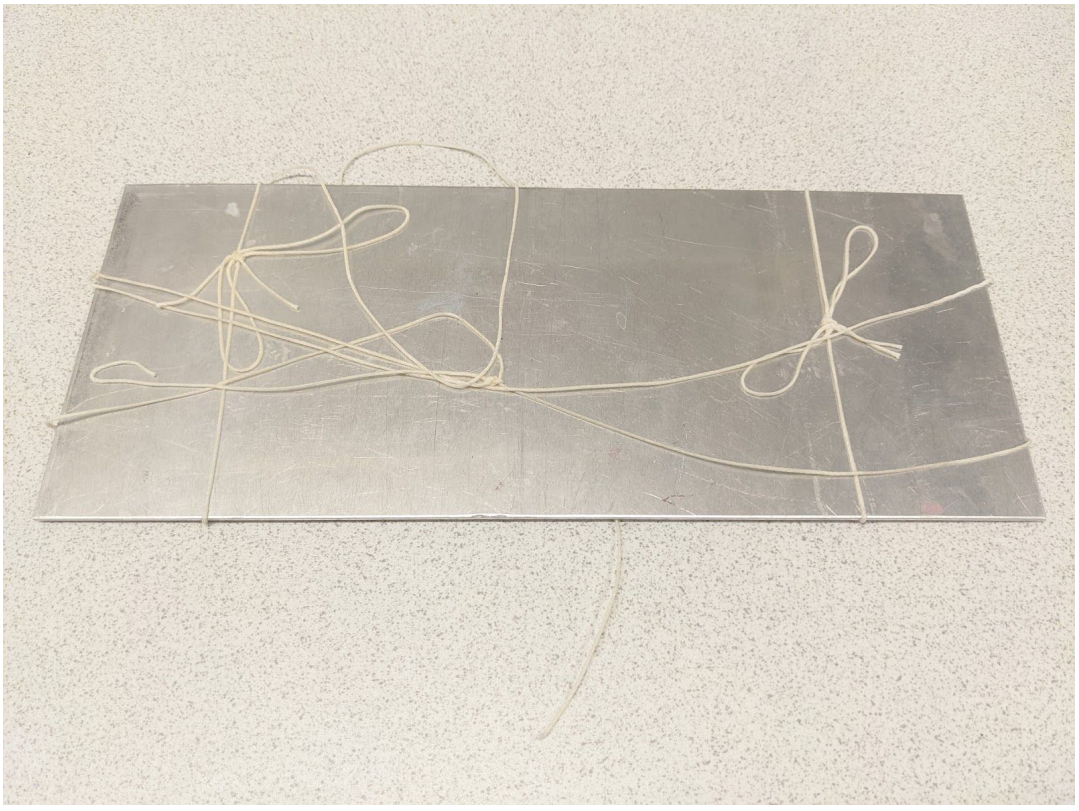


Figure 11 . A picture of the metal object that was used in the verification tests.

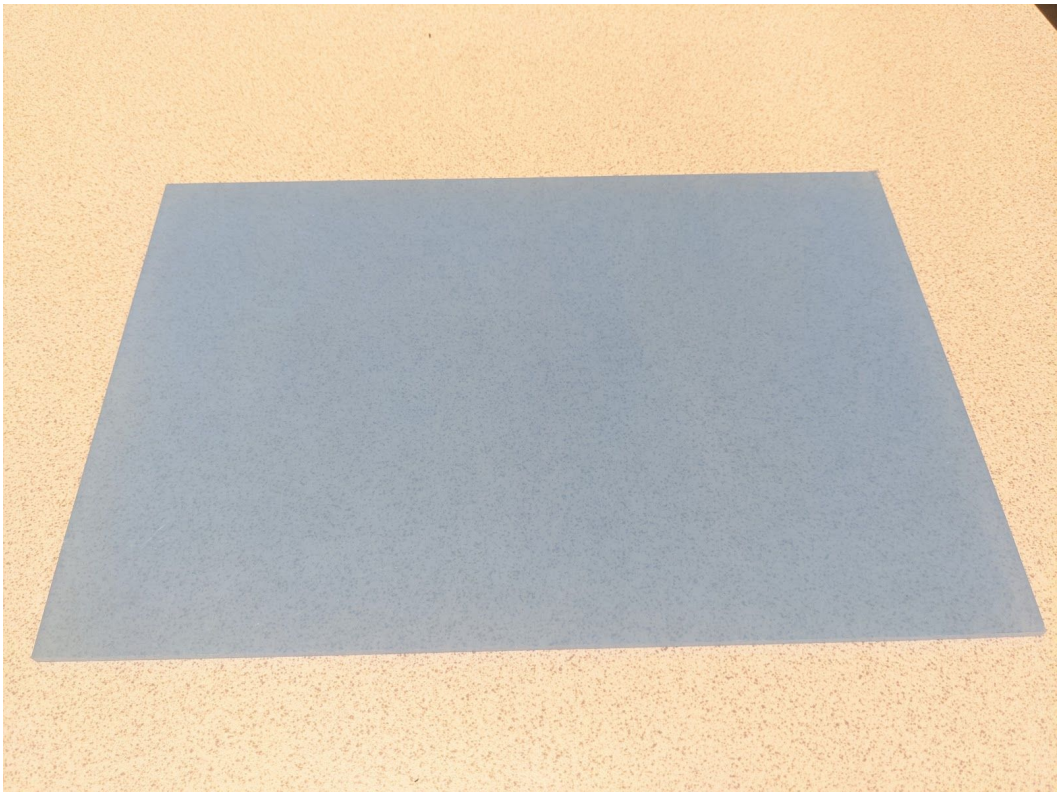


Figure 12. A picture of the plastic object that was used in the verification tests.

The verification tests data was recorded with a laptop in Labview. The results were exported and imported into Matlab which was used to draw the different graphs that can be seen in the results section.

Figure 13. A picture of the verification test environment. The radar is held and moved by hand, pointing downwards to measure the distance to the table.

4. Results

Future algorithms can easily be included since access to the envelope data is available at all times inside the code. An easy way to write new algorithms is simply to make a new function like the “* values() “ function which is called inside CustomEnvelope. Depending on what the user wants to accomplish the “* values() “ function can be edited, removed or simply left empty. For more information about the functions see the below section.

4.1 Added custom functions and code

The written code for each file can be seen at appendice E.

4.1.1 Main.c file

The main.c file includes the selection protocol which receives the user input before passing along the parameters to the envelope function that returns the radar data based on the envelope detector. The selectionProtocolChecker function is run every time the user input is read to ensure that the user entered an acceptable value. Once all values are entered and acceptable, the program is ready to start when the start-command is input. It is not supposed to and doesn't give a warning if an incorrect value has been entered.

The io_read function is written by Acconeer and pauses the program while waiting for the user input. User inputs are entered in hexadecimal form.
For more information on the selection protocol see appendix d.

4.1.2 CustomEnvelope.c file

This function is based on the envelope service that is written by acconeer.

The print_sum_data functions receives a pointer to the radar data, which is a vector, and the length of the vector and summarizes the data before returning the result. This function was used as a simple check to see if the data can be received and processed correctly.

The values function gives access to a vector containing the different parts of the integrals of the radar sensor data. The cutpoints are used to determine how big the different parts of the integrals should be and the scale bin is optionally used to amplify or deamplify any of the 3 parts. Each part has its own corresponding scale value that is chosen before the program is started. The function also allows the user to choose start length, sweep length and update rate.

The customEnvelope function is the one that is being called inside the main program and it creates and uses the configuration for the radar system before destroying it to free up space. It also calls the values function which handles the data processing. The three print functions were used in the testing stage and are not used in the final build.

4.2 Test run of termite with parameters

Start distance	1 (mm)
Sweep length	511 (mm)
Update rate	20000 (mHz)
Cut point 1	32 (%)

Cut point 2	64 (%)
Scale band 1	1
Scale band 2	1
Scale band 3	1

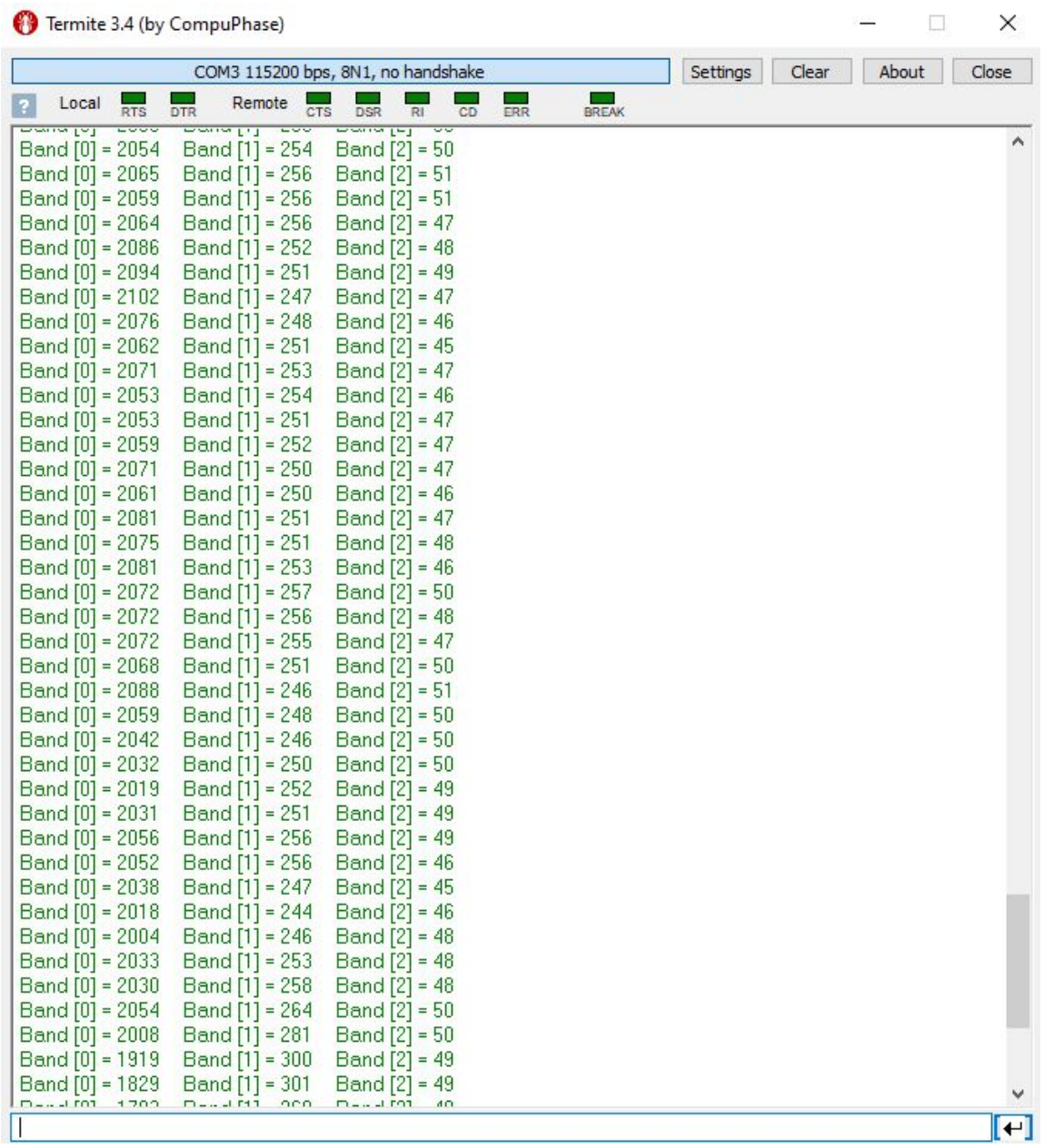


Figure 14. Termite testrun using serial communication using UART. The XM112 was connected to an external computer via USB.

4.3 Validation test

4.3.1 Validation test protocol one

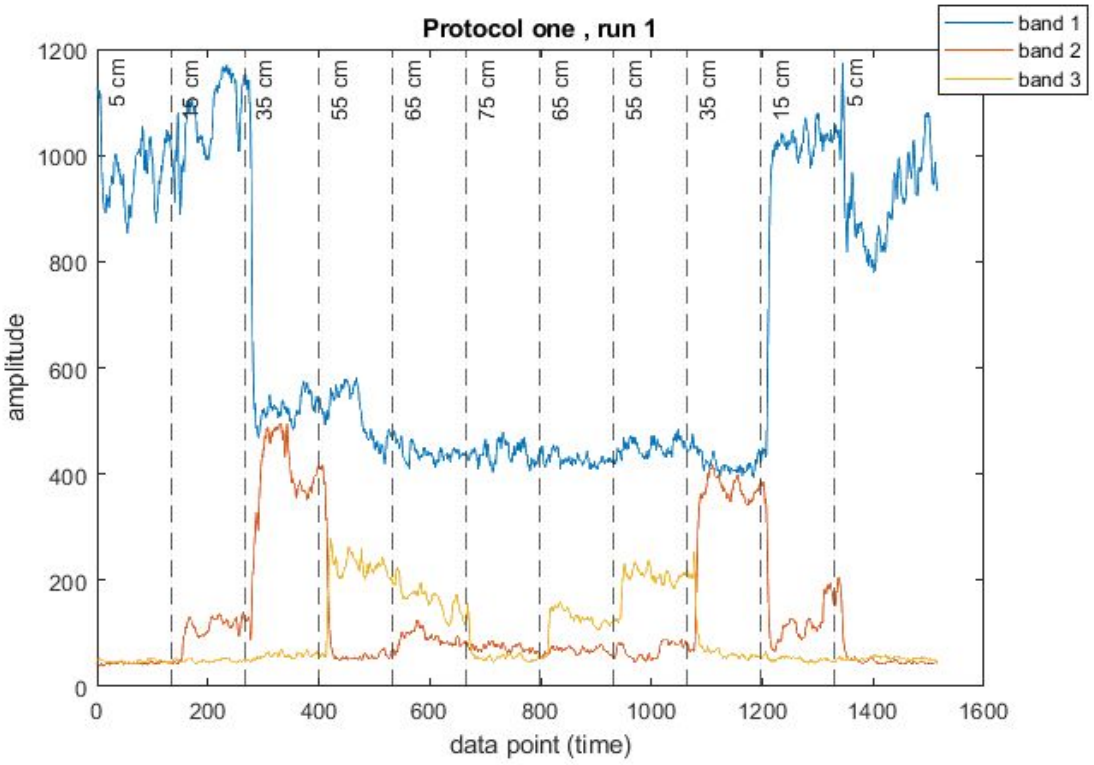


Figure 15. Plot of the data in Matlab from the first test run of verification test protocol one.

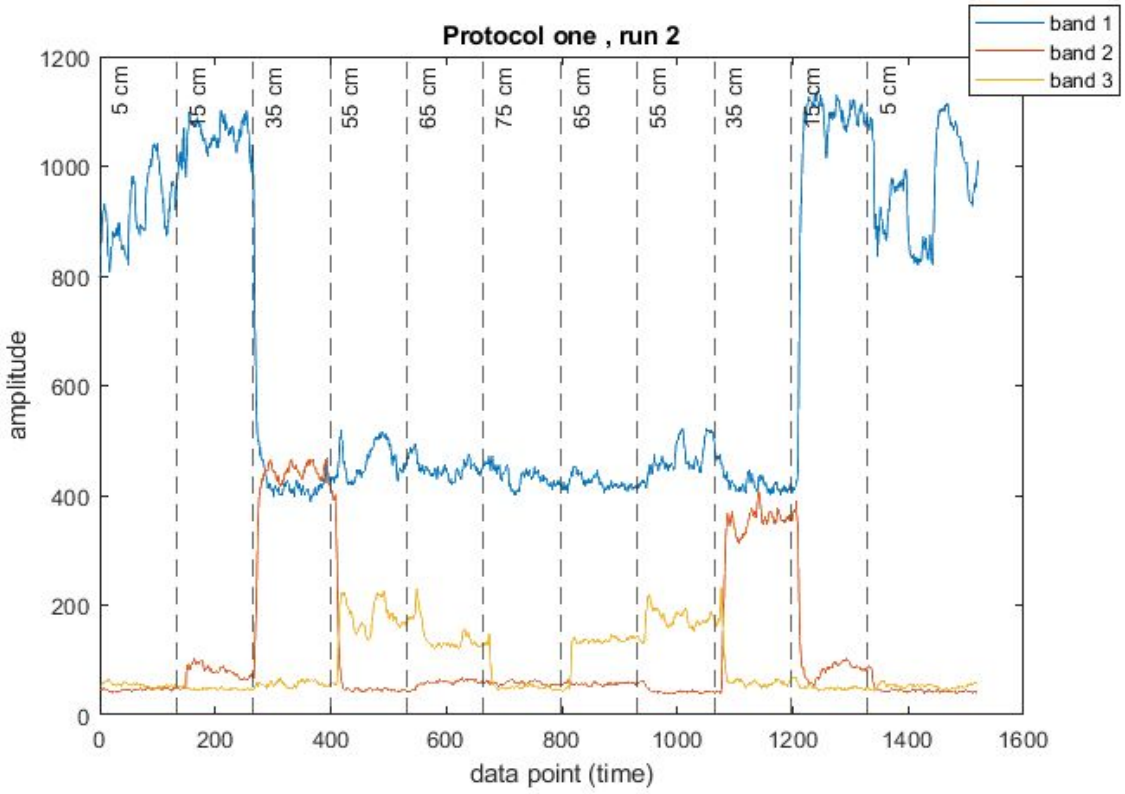


Figure 16. Plot of the data in Matlab from the second test run of verification test protocol one.

4.3.2 Validation test protocol two

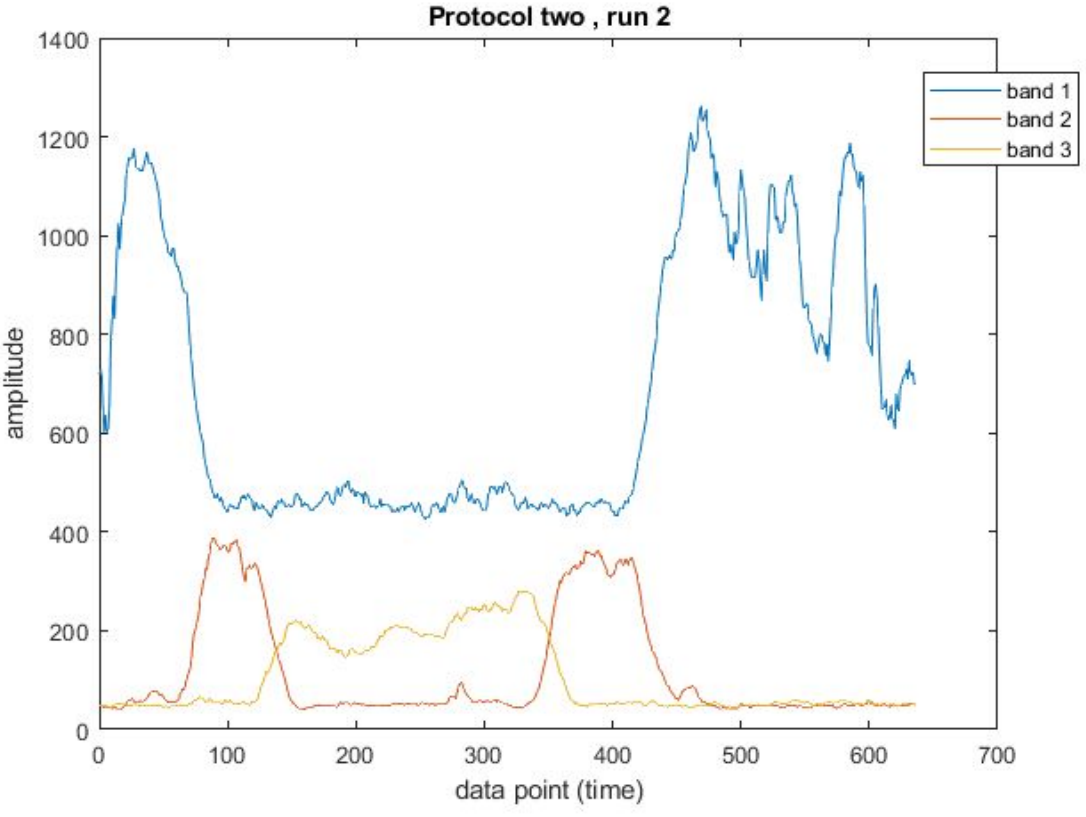


Figure 17. Plot of the data in Matlab from the second test run of verification test protocol two.

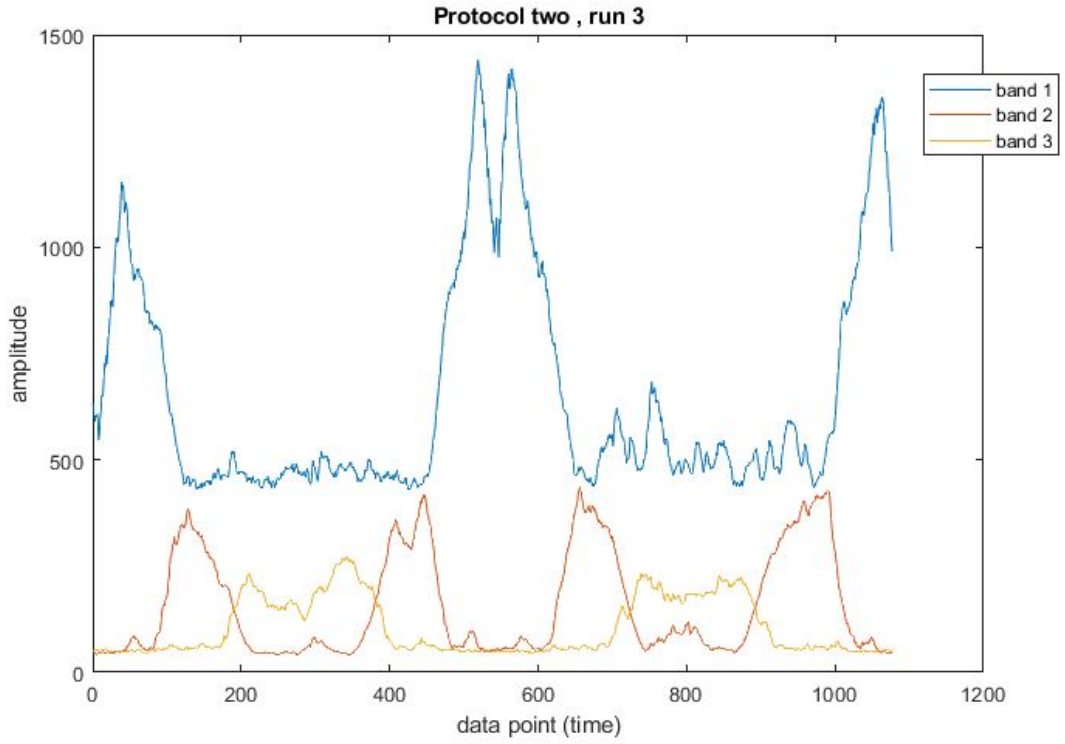


Figure 18. Plot of the data in Matlab from the third test run of verification test protocol two.

4.3.3 Moving targets - validation test protocol one

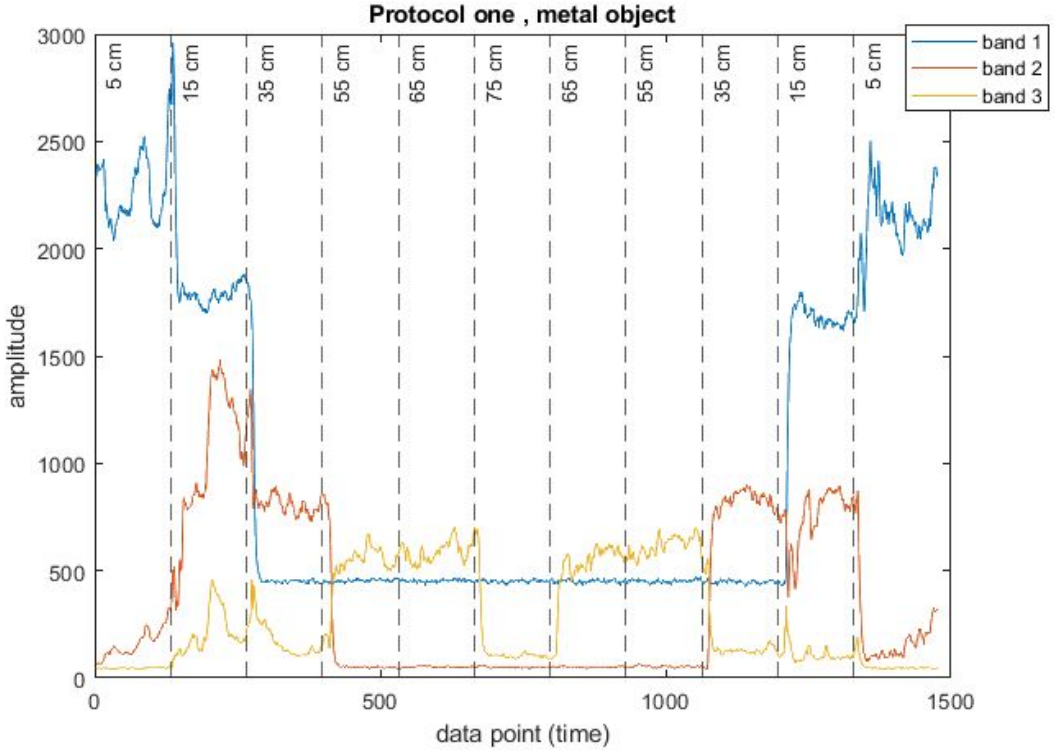


Figure 19. Plot of the data in Matlab from the test run of verification test protocol one using a metal object.

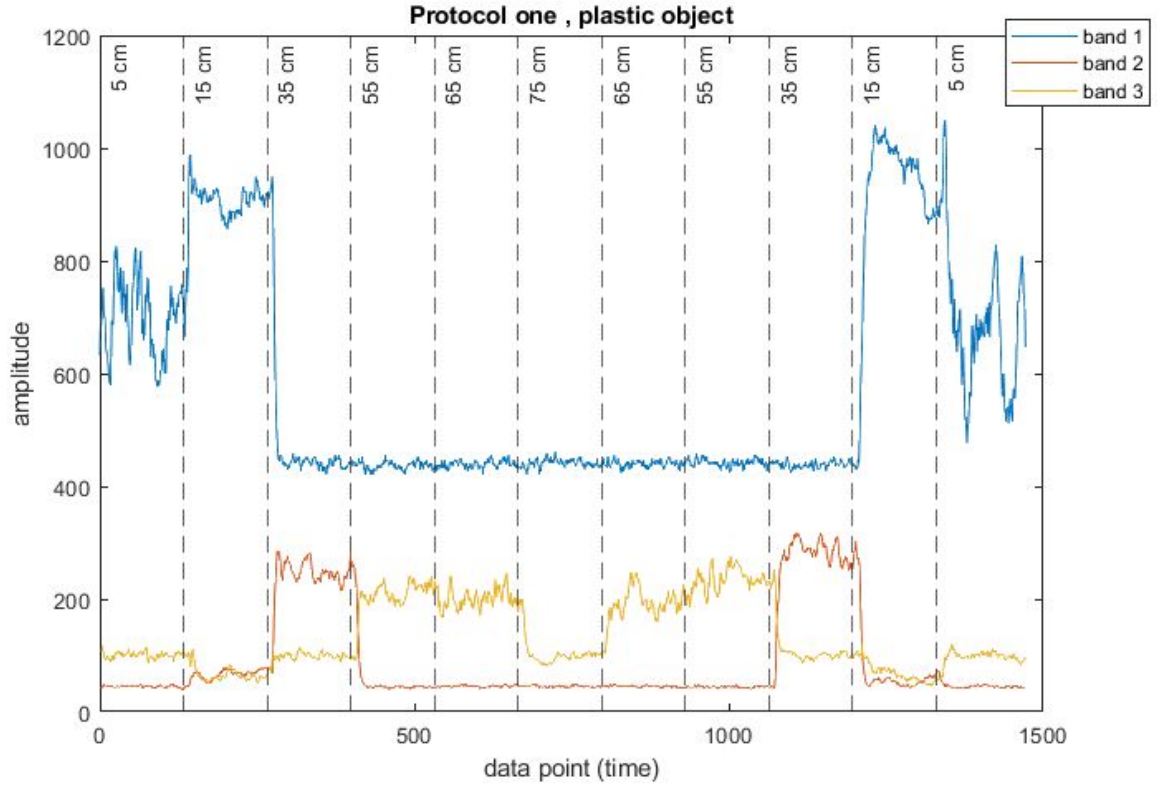


Figure 20. Plot of the data in Matlab from the test run of verification test protocol one using a plastic object.

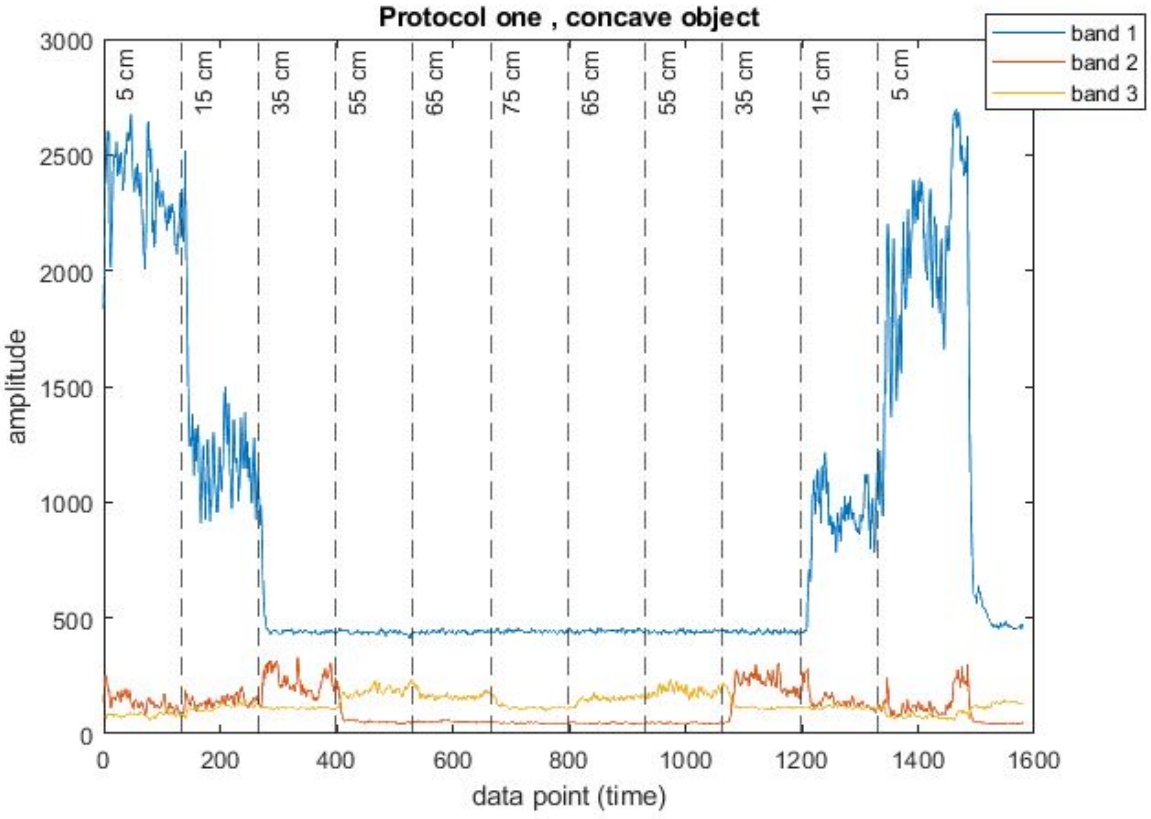


Figure 21. Plot of the data in Matlab from the test run of verification test protocol one using a concave object.

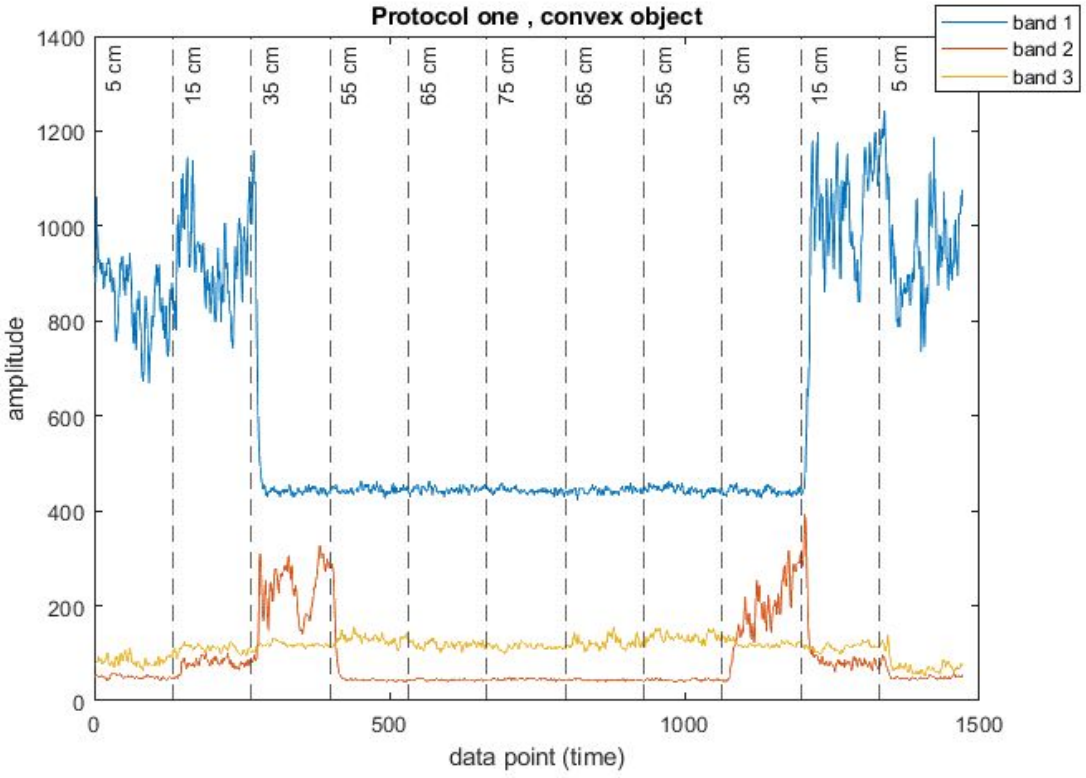


Figure 22. Plot of the data in Matlab from the test run of verification test protocol one using a convex object.

4.3.4 Moving targets - validation test protocol two

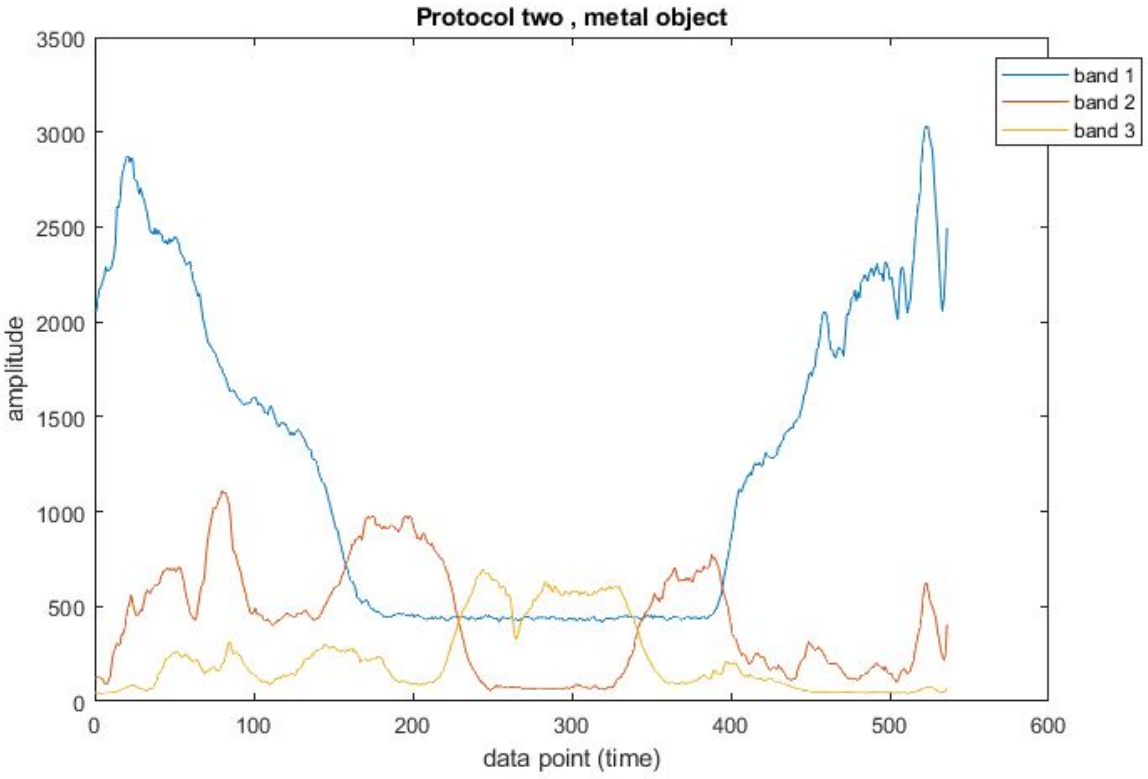


Figure 23. Plot of the data in Matlab from the test run of verification test protocol two using a metal object.

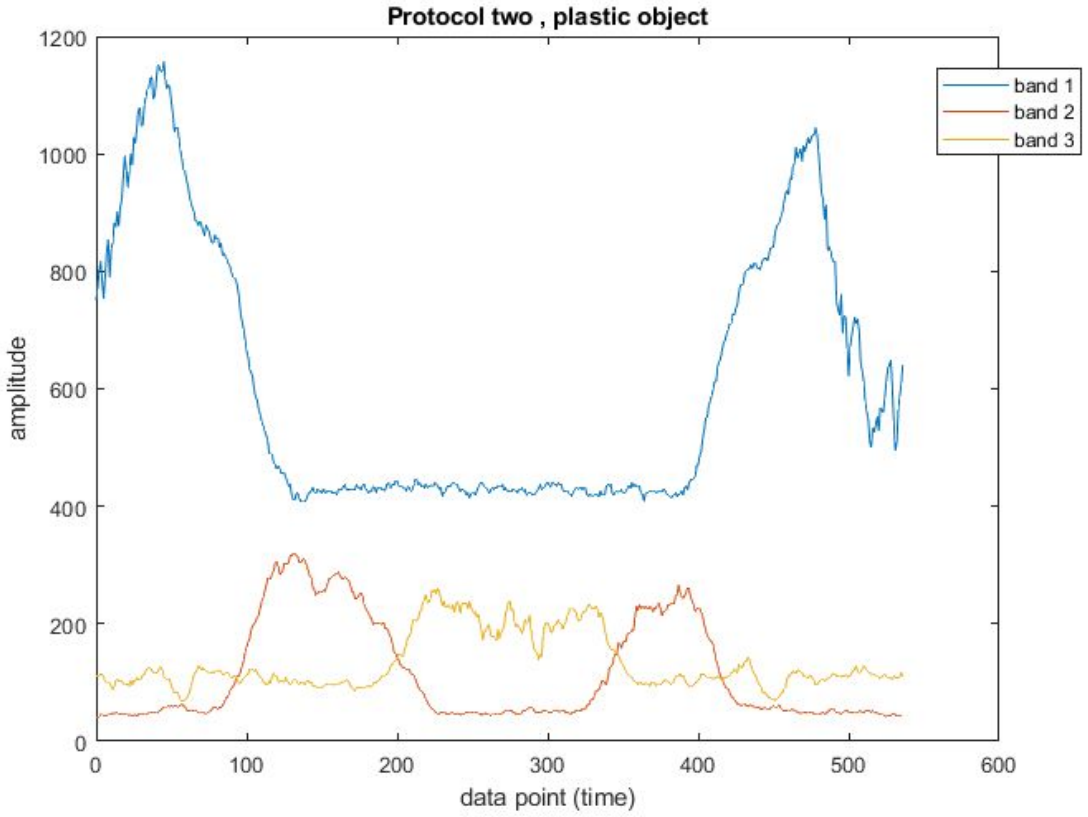


Figure 24. Plot of the data in Matlab from the test run of verification test protocol two using a plastic object.

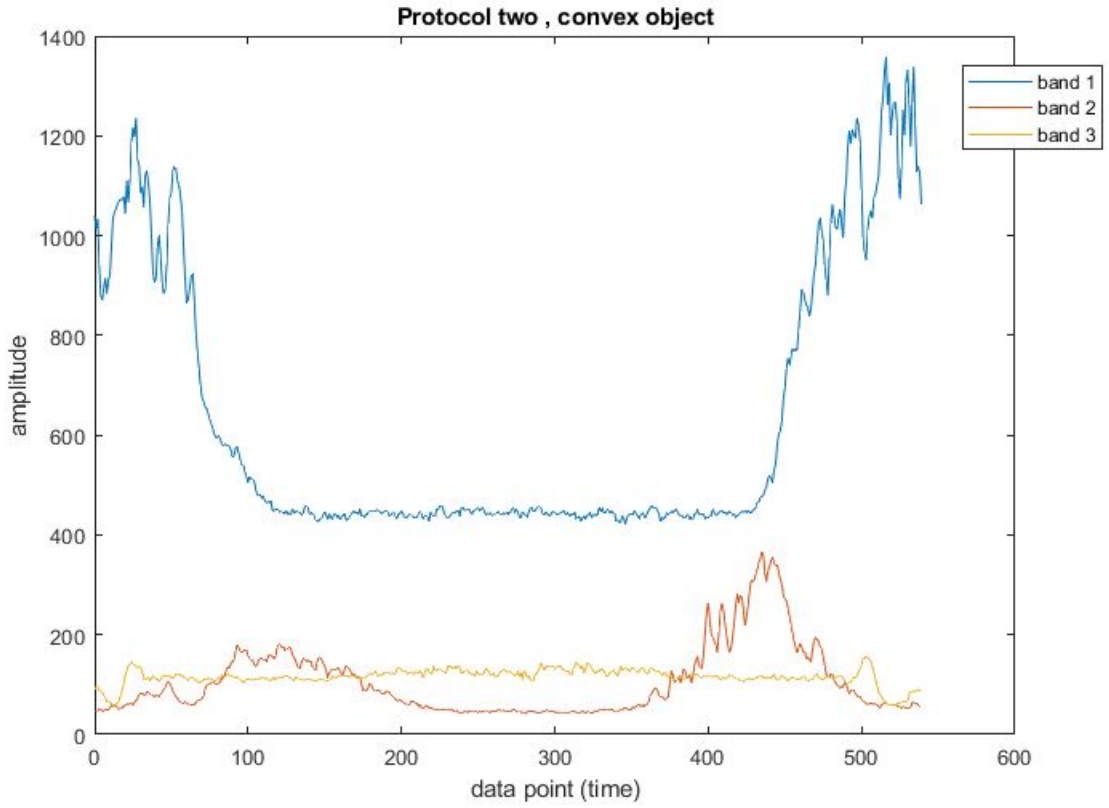


Figure 25. Plot of the data in Matlab from the test run of verification test protocol two using a convex object.

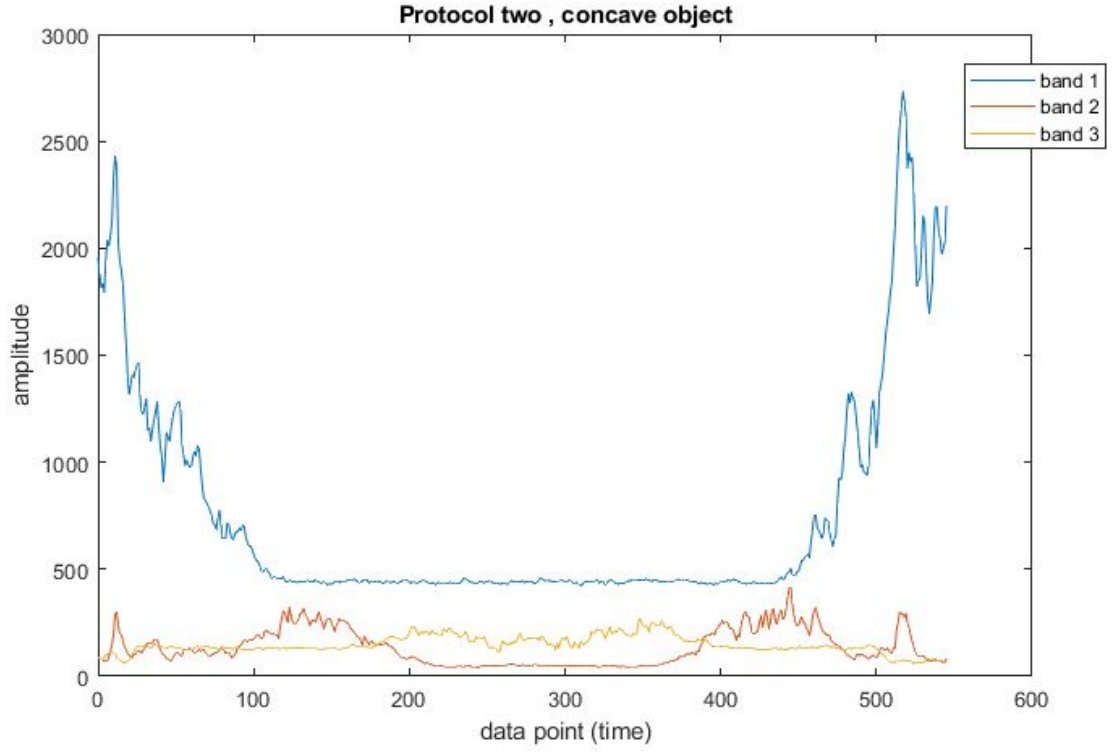


Figure 26. Plot of the data in Matlab from the test run of verification test protocol two using a concave object.

4.4 Pie charts

In this section some pie charts of the normalized mean values of the three bands will be presented. In total 11 charts representing the different segments will be shown for a test run. The figures will show a graphical representation of how the normalized mean percentages change over time. The diagrams will aid in interpreting the results of the test runs.

Each of the bands are normalized separately.

The normalization is done according to the following equation:

$$\frac{(band\_x\_signal - min(band\_x\_signal))}{(max(band\_x\_signal) - min(band\_x\_signal))},$$

where band_x_signal is the current data element in the data vector, min() is the smallest data element in the vector and max() is the largest data element in the vector.

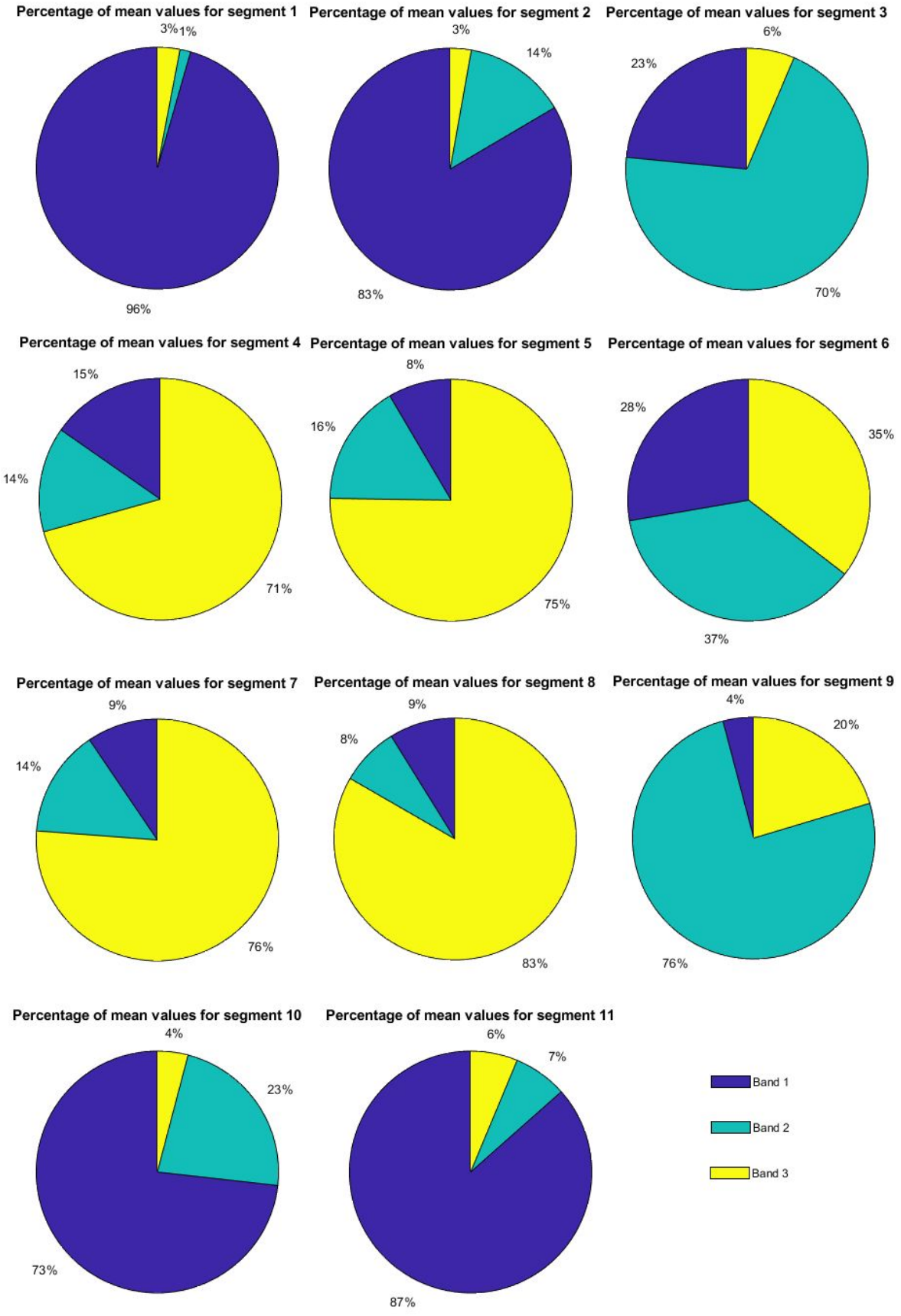


Figure 27. Pie chart for protocol 1 - run 1. Related to figure 15.

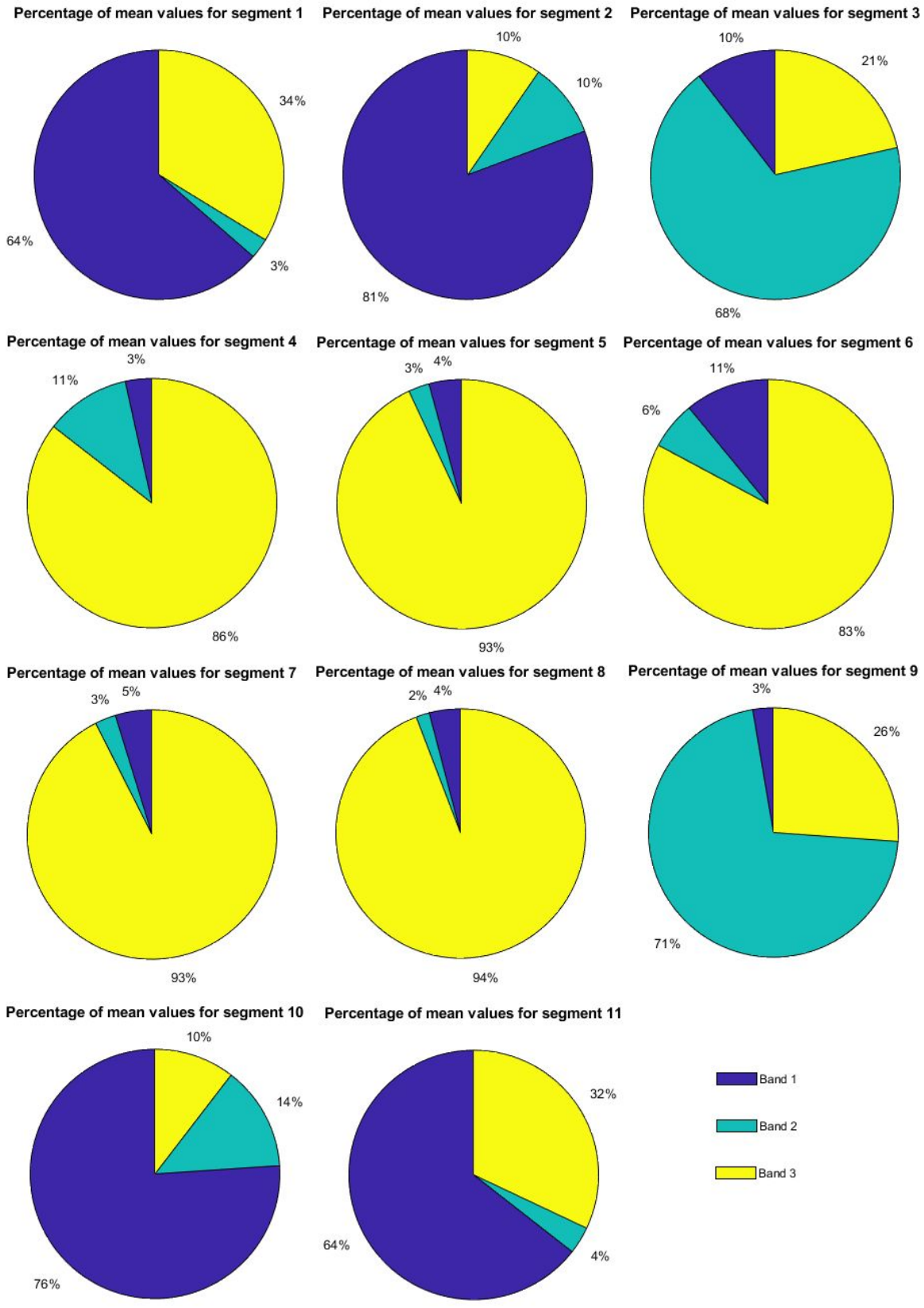


Figure 28. Pie chart for protocol 1 - plastic object. Related to figure 20.

5. Discussion and conclusions

5.1 Goals and results

The goal of accessing, processing and transferring data in real time directly on the MCU to a host computer was achieved. As shown in the results section. The verification tests showed that the data processing works to a satisfying degree.

Upon starting the customEnvelope mode the radar data streaming will be looping for a finite time. Preferably it should loop until a stop signal is given. This could be solved with interrupt handling in the code, which will probably require a rewritten selection protocol.

In the future, this work can be used as a reference to quickly help users set up new projects and contribute with their own modifications.

5.2 Comparison to feasibility study

This work contains some improvements over the feasibility study in regards to the setup and data transferring which was the aim of this master thesis.

The feasibility study was not done in real-time, instead the raw data was sent from A111 to RaspberryPI using SPI and long wires, and from the RaspberryPi to a PC over wifi which was inefficient and a bit wasteful with hardware. Now this disadvantage has been bypassed and only fewer values need to be sent since they are already pre-processed. This also means that only one MCU will be needed for future use.

Since the data is processed by the MCU before being sent there is a lowered communication load which will decrease power usage and improve general performance.

5.3 Selection Protocol

The selection protocol is based on the original firmware from acconeer where inputs are sent through serial communication with hexadecimal numbers. A selection protocol is needed in order to be able to choose different parameters and which mode to be run by the RSS. Currently there is only one custom written mode but if more are written they will be able to be chosen with the selection protocol.

Expanding the selection protocol with additional modes should be possible in theory, one idea is to add more boolean flags, one for each algorithm or mode.
There will be two cases depending on if the set of parameters are the same for each new mode;

1)If the parameters for the algorithms are the same as the CustomEnvelope it will be easy to add new modes/algorithms. In order to do this new flags will be added for every single mode.

The selection protocol will then determine which mode to run with simple if statements. The done2 boolean variable can be changed to an integer variable and the condition to run a mode will be to check if done is true and done2 equals a specific value for a specific mode.

2)This case will be the hard one, if the set of parameters of a new mode are not identical to the customEnvelope mode. One possible solution might be:
Adding a new selectionProtocolChecker that will need to be written for every new set of parameters. Every selectionProtocolChecker will need to be run one after another after every input and be linked to yet another boolean variable. The io_read will continue to run as long as the conditions to run a mode is not met (which is having a valid set of parameters and receiving a command to start a mode).

5.4 Validation test

Both test protocols were designed to gather as much data as possible without taking too much time for efficiency. The tests are meant to simulate walking up or down stair steps. The distances aren't increasing with the same amount in order to also record data from more drastic distance changes.

The setup of the validation tests were not very accurate but for the verification processes extreme accuracy were not needed since the interesting parts of the tests were checking if the sensor data behaved as expected at various distances. This means expecting the amplitude of the first band to be high when the sensor is near a target and lower when the sensor is moving further away. The second band should start with a low amplitude and increase until a maxima is reached and then decrease again. The third band should start with a low amplitude and end with a high maxima. The third and second band having a lower amplitude is in accordance with theory. An object should give a weaker reflection if it is placed at a further distance from a sensor.

If we look carefully at the results (Figure 15 to 26) we can see that almost all of the bands behave mostly as expected. The second and third bands are showing a smaller amplitude due to the beam having an angle which will cause the beam to hit a bigger area at further distances. This scattering will cause the lower amplitude that the sensor detects. The signal loses energy according to the inverse square-law.

Figure 18 shows some abnormality in the middle of the test perhaps due to going too close with the sensor, the power will drop when it goes closer than 5 cm. Since all of the test runs except the one in Figure 18 behaved as expected we can conclude that most likely some mistake occurred in that particular test.

If we compare the results of the metal and plastic objects we see that the metal object has higher amplitude because the metallic surface gives off stronger reflection from the electromagnetic waves since it is more conductive than plastic. [13]

The pie charts are intended to give more clarity to the results by showing the normalized mean values of each band. It is normalized in order to compare the values in a common scale. It shows how big each band's mean value is in percentage of each of the segments.

Measurement errors could also depend on the angle of the targets. When the radar was stationary on the table there seemed to be a very slight angle, so it wasn't perfectly perpendicular. Some of the data points of the second protocol might also contain errors since moving the radar and targets by hand isn't accurate. There were some runs in protocol 2 where the radar and targets were moving too fast and had to pause for a while for the timer to catch up. Although some might argue that ideally the second protocol should be moved by a device with constant velocity instead, movement by hand also has it's uses since normal movement isn't mechanically perfect. With this in mind, it is worth pointing out that the hand movement in both protocols should be considered a simulation of movement down a stair.

One important thing to notice is that although the update rate was set as 20 Hz the graphs shows that the actual update rate might differ. In the verification test protocol one the sensor changes position every 10th second but the graph shows big changes at around every 150 samples.

Upon further investigation with a time recorder in Labview the actual mean update rate was found to be around 13.3 Hz by dividing the elapsed time with the number of samples. This is a significant difference of around 35% between the selected update rate and the actual one!

Further tests with different update rates specified in the software shows that some significant lagging (roughly 30%) is occurring at update rates over 20 Hz. One software setting that improved the lag was downsampling. Neither improving the code nor changing HWAAS gave better results, in fact the results were almost identical. Improvement to the code consisted of a couple of removed single line code and a for-loop of the total envelope vector size.

The assumption therefore is that some significant amount of code (multiple for-loops of the envelope data vector size) can be added without affecting the time constraints of the MCU. While the advertised specifications claim that significantly faster update rates can be achieved, most likely a lot of accuracy will be sacrificed in these cases.

6. References

[1] Shahid Mumtaz, Jonathan Rodriguez, Linglong Dai. “mmWave Massive MIMO” , 2017

[2] Emily Boynton. “For elderly, even short falls can be deadly” https://www.urmc.rochester.edu/news/story/3020/for-elderly-even-short-falls-can-be-deadly.aspx , 2010

[3] Tzung-Han Lin, Chi-Yun Yang, Wen-Pin Shih. “Fall prevention shoes using camera-based line-laser obstacle detection system” https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5450170/ , 2017

[4] Nebojša Malešević1, Cheng Feng. Wang2, Katherine Rich2, Christian Antfolk1 1Lund University, Sweden, 2University of Toronto, Canada (3). “Fall prevention for elderly people using radar sensor: Feasibility study”

[5] https://www.acconeer.com/

[6] C.Gordon Bell, J. Craig Mudge, John E. McNamara. “Computer Engineering“, 2014

[7] Mark A. Richards. “Fundamentals of Radar Signal Processing” , 2014

[8] Sebastian Heunisch. “Millimeter-Wave Radar for Low-Power Applications”, 2019

[9] Barend van Liempd, Jan Craninckx. “Ultra-sensitive and low-power radar enables intuitive human-computer interaction”, 2020 https://www.imec-int.com/en/articles/ultra-sensitive-and-low-power-radar-enables-intuitive-human-computer-interaction

[10] Federica Sbaccanti, Niall McMahon. “Automotive Safety and Security” , 2020 https://cipher.ai/wp-content/uploads/2020/04/Auto-Safety-Security_April-2020.pdf

[11] A.M. (Tony) Ponsford. “Introduction to Radar” , 2014

[12] James A. Scheer and James L. Kurtz. “Coherent Radar Performance Estimation” , 1993

[13] Christian Wolff. “Radartutorial” http://read.pudn.com/downloads167/ebook/769566/Radar_basics.pdf

[14] David Jenn. “Radar fundamentals”

https://faculty.nps.edu/jenn/Seminars/RadarFundamentals.pdf
[15] Jon Wilson, “Sensor Technology Handbook”, 2005

[16] Merrill Skolnik, “Introduction to Radar Systems, Second Edition” , 1981

[17] Louis N. Ridenour, “Radar System Engineering”, 1947

[18] Bassem R. Mahafza, “Radar Systems Analysis and Design Using MATLAB” , 2000

[19] Pakhomov, A. G. , Murphy, P. R. “Low-intensity millimeter waves as a novel therapeutic modality”, 2000

[20] Randal E. Bryant, David R. O’Hallaron. “Computer Systems A Programmer’s Perspective”, 2011

7. Appendices

A) Technical specifications of XM112

Features:

- Small form factor: 24 mm x 16 mm
- A111 60 GHz pulsed coherent radar
- Arm Cortex-M7 ATSAME70Q20A microprocessor
- 1.8 V single supply operating voltage
- Operating temperature: -40°C to +85°C
- External interfaces: SPI, UART, and I2C
- SWD interface for SW Flash and debug
- Slim 30-pin board-to-board connector
- Module software, SDK, and Exploration tool available for download

https://www.digikey.se/en/product-highlight/a/acconeer-ab/xm112-pulsed-coherent-radar-module?utm_adgroup=General&utm_source=google&utm_medium=cpc&utm_campaign=Dynamic%20Search_EN_Product&utm_term=&productid=&gclid=EAIaIQobChMIqYS0tObv6QIV1MAYCh2GBwGYEAAYASAAEgJRb_D_BwE

B) System overview:

https://acconeer-python-exploration.readthedocs.io/en/latest/sensor_introduction.html#system-overview

C) User guides (acconeer)

The following guides can be found as an attachment or at the acconeer website:
“User guide HAL software integration.pdf”
“XM112 Module software user guide.pdf”
“Atmel Studio 7 Integration Guide.pdf”

D) User Guide (Selection Protocol)

All inputs need to be 5 bytes long. The first four bytes represent the value for the parameter which is chosen with the last byte. All parameters must be input before the program will execute.
Example: 0x41 0x00 0x00 0x00 0x01 will give a start length of 65 mm.

Flags:
0x01 = start length , limits [-700 , 7000]   , unit : mm
0x02 = sweep length, limits [0 , 7700]   , unit : mm
0x03 = update rate, limits [0 , 15000000]   , unit : mHz
0x04 = cut point 1 , limits[0-100]
0x05 = cut point 2 , limits[0-100]
0x06 = scale band 1 , limits[>0] , A scale bin value of 1 will multiply the first part with 0,001.
0x07 = scale band 2
0x08 = scale band 3
0x09 = start Envelope Program. It will only run if the other parameters are valid. The first four bytes can be any value but cannot be skipped.

Cut point: The envelope data is divided into 3 parts based on the cut point values. Example: cutpoint 1 with value 10 and cutpoint 2 with value 30 will divide the vector into three parts, where the first part (the parts will be referred to as bands) is 0-10 % , the second part is 10-30 % and the last part is 30-100%.

Scale band: The sum of each part of the envelope data multiplied with the corresponding scale band value. Example: the first part is multiplied with scale band 1.

E) Atmel studios 7 code

----------Main.c----------

```
#include <atmel_start.h>
#include "CustomEnvelope.h"
#include "example_service_envelope.h"
#include "example_detector_distance_peak.h"

#include "hal_usart_sync.h"
```

```
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

bool selectionProtocolChecker(int SD, int SL, int UR, int C1, int C2, int S1, int S2,
int S3);

int main(void)
{
	atmel_start_init();

	struct io_descriptor *io;
	usart_sync_get_io_descriptor(&A111_DEBUG_LOG_HANDLE, &io);
	usart_sync_enable(&A111_DEBUG_LOG_HANDLE);

	// initialize parameters and flags for selection protocol
	int startDist = -1000;
	int length = -1000;
	int updateRate = -1000;
	int cut1 = 0;
	int cut2 = 0;
	int S1= 0;
	int S2= 0;
	int S3= 0;
	bool done = false;
	bool done2 = false;
	uint8_t buf1[5];
	uint32_t dummy = 0;

	/*selection protocol loop, which will continue until all parameters are valid and
start flag to run measurement is received */
		while (1) {

	 if(!done || !done2){
		dummy = io_read(io, buf1, 5);

		// will have to run X times to make sure that all parameters are chosen
		switch(buf1[4]){
			case 1:
			startDist = buf1[0] + (buf1[1] << 8) + (buf1[2] << 16) + (buf1[3]
<< 24);
			done = selectionProtocolChecker(startDist, length, updateRate,
cut1, cut2, S1, S2, S3);
			break;
			case 2:
			length = buf1[0] + buf1[1] << 8 +buf1[2] << 16 + buf1[3] << 24;
			done = selectionProtocolChecker(startDist, length, updateRate,
cut1, cut2, S1, S2, S3);
			break;
			case 3:
			updateRate = buf1[0] + (buf1[1] << 8) + (buf1[2] << 16) + (buf1[3]
```

```
<< 24);
                        done = selectionProtocolChecker(startDist, length, updateRate,
cut1, cut2, S1, S2, S3);
                        break;
                        case 4:
                        cut1 = buf1[0] + (buf1[1] << 8) + (buf1[2] << 16) + (buf1[3] <<
24);
                        done = selectionProtocolChecker(startDist, length, updateRate,
cut1, cut2, S1, S2, S3);
                        break;
                        case 5:
                        cut2 = buf1[0] + (buf1[1] << 8) + (buf1[2] << 16) + (buf1[3] <<
24);
                        done = selectionProtocolChecker(startDist, length, updateRate,
cut1, cut2, S1, S2, S3);
                        break;
                        case 6:
                        S1 = buf1[0] + (buf1[1] << 8) + (buf1[2] << 16) + (buf1[3] << 24);
                        done = selectionProtocolChecker(startDist, length, updateRate,
cut1, cut2, S1, S2, S3);
                        break;
                        case 7:
                        S2 = buf1[0] + (buf1[1] << 8) + (buf1[2] << 16) + (buf1[3] << 24);
                        done = selectionProtocolChecker(startDist, length, updateRate,
cut1, cut2, S1, S2, S3);
                        break;
                        case 8:
                        S3 = buf1[0] + (buf1[1] << 8) + (buf1[2] << 16) + (buf1[3] << 24);
                        done = selectionProtocolChecker(startDist, length, updateRate,
cut1, cut2, S1, S2, S3);
                        break;
                        case 9:
                        if(done){
                        done2 = true;
                        }
                        break;

                        default:
                        break;

                }

        }

               if(done && done2){
                       customEnvelope(startDist, length, updateRate, cut1, cut2, S1, S2,
S3);
               }

       }

}
```

```
//function is used to check that all parameters are valid
bool selectionProtocolChecker(int SD, int SL, int UR, int C1, int C2, int S1, int S2,
int S3) {

	if(SD < -700 || SD > 7000){
		return false;
	}

	if(SL < 0 || SL > 7700){
		return false;
	}
	if(UR < 0 || UR >1500000){
		return false;
	}

	if(C1 < 1 || C1 >97 ){
		return false;
	}
	if(C2 < 3 && C2 > 99 ){
		return false;
	}

	if(S1 <= 0 || S2 <=0 || S3 <=0   ){
		return false;
	}

	return true;
}
```

----------CustomEnvelope.c----------

IMPORTANT NOTICE: The “/* YOUR CUSTOM CODE HERE */” marker is just one proposal to where custom code can be put. It is important to understand how the selection protocol works together with the CustomEnvelope file in order to use the functions efficiently.

```
bool customEnvelope(int start, int length, int updateRate, int cut1, int cut2, int
scaleBand1, int scaleBand2, int scaleBand3)
{
	float startf = start;  // received in mm
	float lengthf = length;
	float updateRatef = updateRate;
	float sb1f = (float) scaleBand1/1000;
	float sb2f = (float) scaleBand2/1000;
	float sb3f = (float) scaleBand3/1000;

	acc_hal_t hal = acc_hal_integration_get_implementation();
		if (!acc_rss_activate(&hal))
		{
			fprintf(stderr, "acc_rss_activate() failed\n");
			return false;
```

```
		}

	acc_service_configuration_t envelope_configuration =
acc_service_envelope_configuration_create();

		if (envelope_configuration == NULL)
		{
			fprintf(stderr, "acc_service_envelope_configuration_create()
failed\n");
			acc_rss_deactivate();
			return false;
		}

	acc_service_requested_start_set(envelope_configuration, (float) startf/1000);
	acc_service_requested_length_set(envelope_configuration, (float) lengthf/1000);
	acc_service_repetition_mode_streaming_set(envelope_configuration, (float)
updateRatef/1000);

	acc_service_handle_t handle = acc_service_create(envelope_configuration);

		if (handle == NULL)
		{
			fprintf(stderr, "acc_service_create() failed\n");

acc_service_envelope_configuration_destroy(&envelope_configuration);
			acc_rss_deactivate();
			return false;
		}

	acc_service_envelope_configuration_destroy(&envelope_configuration);

	acc_service_envelope_metadata_t envelope_metadata;
	acc_service_envelope_get_metadata(handle, &envelope_metadata);

	if (!acc_service_activate(handle))
	{
		fprintf(stderr, "acc_service_activate() failed\n");
		printf("acc_service_activate() failed\n");
		acc_service_destroy(&handle);
		acc_rss_deactivate();
		return false;
	}

	bool                              success    = true;
	const int                         iterations = 10000;
	uint16_t                          data[envelope_metadata.data_length];
	acc_service_envelope_result_info_t result_info;


	for (int i = 0; i < iterations; i++)
	{
		success = acc_service_envelope_get_next(handle, data,
envelope_metadata.data_length, &result_info);
```

```
            if (!success)
            {
                    fprintf(stderr, "acc_service_envelope_get_next() failed\n");
                    printf("acc_service_envelope_get_next() failed\n");
                    break;
            }

            int *p;

          /* YOUR CUSTOM CODE HERE start*/
            p = values(data, envelope_metadata.data_length , cut1, cut2, sb1f, sb2f,
sb3f);

            printf("Band [0] = %d     ", p[0]);
            printf("Band [1] = %d     ", p[1]);
            printf("Band [2] = %d \n ", p[2]);

          /* YOUR CUSTOM CODE HERE end*/
        }

        bool deactivated = acc_service_deactivate(handle);
        acc_service_destroy(&handle);
        acc_rss_deactivate();
        return deactivated && success;
}




void print_sum_data(uint16_t *data, uint16_t data_length)
{
        uint16_t result = 0;
        for (uint16_t i = 0; i < data_length; i++)
        {
                result = result + data[i];
        }

        printf("Sum of envelope data result = %6u \n" , (unsigned)result );
}


void print_sum_data2(uint16_t *data, uint16_t data_length)
{
        int envelope[1034];

        for(uint16_t i = 0; i < data_length; i++)
        {
                envelope[i] = data[i];
        }

        int result = 0;
```

```
        for(int i = 0; i < 1034; i++)
        {
                result = result + envelope[i];
        }

        printf("Sum of envelope data 2 result = %d \n" , result );

}

static int * values(uint16_t *data, uint16_t data_length, int cut1, int cut2 , float s1,
float s2, float s3)
{
        static int band[2] = {0};
        band[0] = 0;
        band[1] = 0;
        band[2] = 0;
        int endL = data_length;
        int envelope[endL];

        for(int t = 0 ; t<=endL ; t++){
                envelope[t] = 0;
        }

        float cutpoint1 = cut1*0.01*data_length;
        float cutpoint2 = cut2*0.01*data_length;
        int nn = (int) cutpoint1;
        int mm = (int) cutpoint2;

        for(uint16_t i = 0; i < data_length; i++)
        {
                envelope[i] = data[i];
        }

        for(int i = 0 ; i < nn ; i++){
                band[0] = band[0] + envelope[i];
        }
        band[0] = (int) band[0] * s1;

        for(int i = nn ; i<mm ; i++){
                band[1] = band[1] + envelope[i];
        }
        band[1] = (int) band[1] * s2;

        for(int i = mm ; i<endL ; i++){
                band[2] = band[2] + envelope[i];
        }
        band[2] = (int) band[2] * s3;

        return band;
}
```