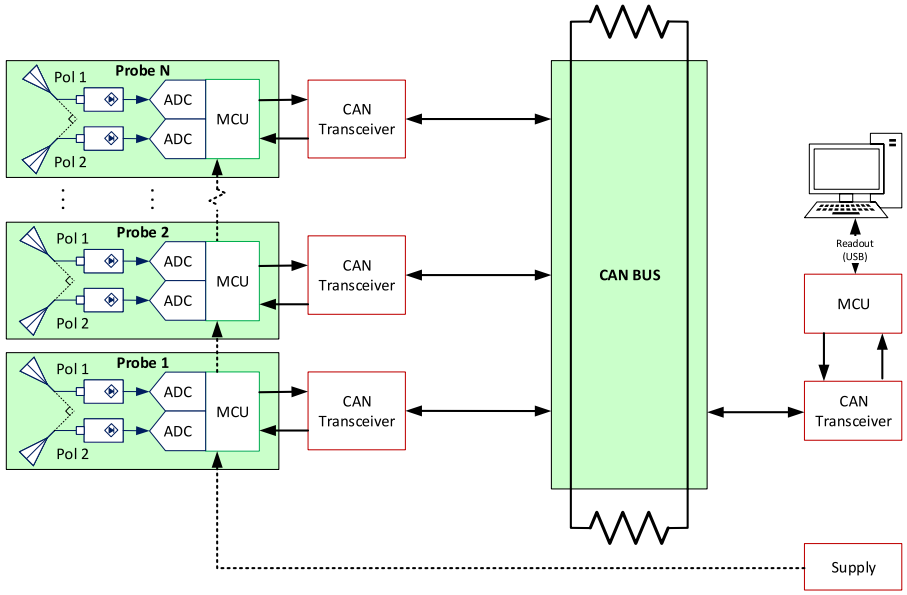
ADome

Implementing CAN in a multi-sensor measurement setup

A. J. Becoy & R. Zhang

Bsc Thesis



TU Delft

ADome

Implementing CAN in a multi-sensor measurement setup

by

A. J. Becoy & R. Zhang

to obtain the degree of Bachelor of Science in Electrical Engineering,
at the Delft University of Technology,
to be defended publicly on Monday June 28, 2021 at 09:00 AM.

Student numbers: 4904494 & 4835182
Project duration: June 19, 2021 – June 18, 2021
Thesis committee:

Dr. M. Spirito,	Associate Professor, ELCA, TU Delft, supervisor
Msc. F. A. Musters,	Parttime Reseacher, ELCA, TU Delft, supervisor
Prof. Dr. L. C. N. de Vreede,	Full Professor, ELCA, TU Delft
Dr. M. Alonso-del Pino,	Assistant Professor, Terahertz Sensing,TU Delft

TU Delft

Abstract

This thesis focuses on improving the readout of the ADome by implementing MCUs at each antenna probe, enabling local sampling and memory storage. The serial communication protocols CAN, SPI and I2C are considered and compared with one another. Ultimately, CAN is decided due to its robustness and simplicity which make the system cheap and ensures that the measurement will not get corrupted during transmission. Moreover, implementations of the new readout protocol are able to obtain measurement data store information at the local MCU. Test setups verification showed that antenna location can be stored and retrieved. Furthermore, the readout protocol is able to acquire multiple samples from the ADC locally.

Preface

This thesis is written in context of the Bachelor Graduation Project. The project was proposed by Dr. M. Spirito and Msc F. Musters. The latter developed the ADome concept during his master studies and is currently still engaging on researching and improving the ADome. The concept is a multi-probe measurement array, able to characterize antenna radiation patterns in real-time without moving parts.

After being introduced to the sub-projects of the ADome, we were really curious about improving the readout process by inserting micro-controllers at each probe to improve the readout process and expand the functionatlies of the ADome. Furthermore, we were really fascinated by the communication protocol Controller Area Network (CAN).

At this point, we would like to show our gratitude to the Delft University of Technology for teaching us the knowledge and giving us the opportunity to complete our bachelor degree. We have enjoyed the whole education period in the past three years.

Furthermore, we would like to thank Dr. M. Spirito, Msc F. Musters and R. Coesoij for their supervision and support throughout the project. You gave us many useful feedback and taught us how to be an professional engineer. Furthermore, we are also grateful for our entire team for displaying their competence, their assistance and camaraderie.

Finally, we would like to thank our friends and family for their unconditional support. Without them, we would not have been at where we are right now.

A. J. Becoy & R. Zhang
Delft, June 2021

Contents

1 Introduction

Devices for the second frequency range of 5G-NR are currently being developed. With this move to higher frequencies substantially more bandwidth will become available, allowing for higher data throughput. However, higher frequencies also complicate the development of the communication systems. Developments in wireless communication ever move towards higher frequency bands. Most recently, the first frequency range of the 5G standard for telecommunication is starting to become more adopted into mainstream consumer technology, with frequencies still below 6 GHz. The second frequency range foreseen for 5G, however, will range into the millimeter-wave spectrum.

At these higher frequencies, to compensate for the increased free-space loss and increase directivity of the system, antennas are often combined into arrays to use beamforming. These arrays are frequently designed as a monolithic entity to enable further miniaturisation. This makes individual testing of subsystems very complicated to impossible, increasing the relevance of over-the-air (OTA) testing.

In its current form, OTA characterization is a cumbersome process. The most used method involves near- or far-field scanning using a single probe that moves around the antenna-under-test (AUT). Another widely used technique is often known as a Compact Test Range (CATR), where a secondary feed antenna is used with a precisely manufactured reflector to create a region of plane waves where the AUT is placed. The AUT is then rotated in this field of plane waves to measure the response for different angles.

These methods both share weaknesses: they are relatively slow, due to only being able to measure a single point at a time. They require mechanical elements, increasing complexity and points of failure. Additionally, current testing methods typically use a network analyzer in their testing, requiring large lengths of RF cables to the AUT and probing elements.

To solve this problem, the ADome concept has been proposed [1]. The ADome consists of detectors arranged in a half-sphere, each with its own antenna and RMS power detector, allowing the calculation of received power to take place at the node itself. It is a far-field measurement method, but the far-field distance at 5G mm-wave frequencies and above is sufficiently small that it comfortable fits into a regular laboratory. A picture of the structure of the current version can be seen in figure 1.2.
A number of projects were carried out by this group to improve the current ADome prototype. These projects are mostly disconnected from each other as they all consider a different part of the system.

The first project is a new readout protocol. As the sensing probes acquire the measurements locally, the need of a readout protocol becomes necessary for the computer to process these data. The current implementation in the ADome is limited in its functionalities. Antenna probes locations must be computed manually, making it not user-friendly and difficult to use in case of large number of sensing antennas. Furthermore, the probes are not able to store any information, such as its own location. Therefore, a new readout protocol using CAN is proposed where a micro-controller unit (MCU) is integrated at each probe to allow local sampling and readout along with non-volatile memory storage. Due to the presence of local MCUs, the potential functionalities of the ADome can be extended.

The second project concerns an optical calibration system. At the moment, the location of each sensor node needs to be measured and entered manually in the readout system. This project proposes a method to automatically determine the positions of the antennas in the dome. This is done using a camera mounted in the center of the ADome. The antennas will be recognized from the images

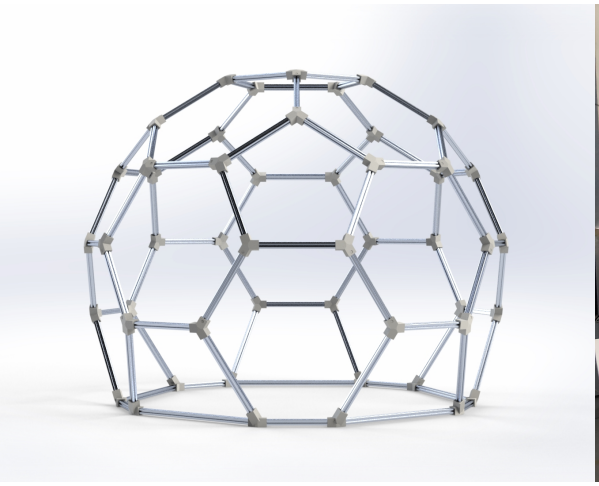


Figure 1.1: The concept of the ADome structure.

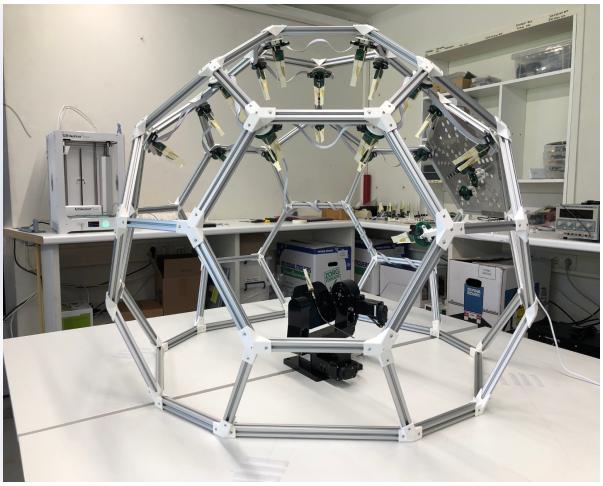


Figure 1.2: Implementation of the ADome concept with sensing probes visible and a gimbal to steer the antenna pointing direction.

taken by the camera using computer vision algorithms and the location in the images is mapped to a location in the dome.

The third project is an uncertainty assessment of the sensing nodes. The ADome sensing nodes use a RMS power detector to measure the received power. This project aims to characterize the measurement uncertainty of the detector for different excitations. A simulation testbench of the measurement setups is also created to allow assessing the impact of changes in the system without needing to change the physical setup. Characterizations is done for single- and multi-tone signals at different frequency and power levels.

State-of-the-art analysis

Digital communication protocols are widely used in many applications. Automotive industries have been using a communication protocol since the 90s, i.e. the Controller Area Network (CAN) developed by Robert Bosch GmbH [2]. CAN found its popularity namely due to its high-reliability multi-node communication with maximal data rate of 1 Mb/s without point-to-point wiring, therefore reducing the costs and weights significantly for the manufacturers [3]. Due to rapid development of the automotive industry, new protocols derived from CAN to accommodate the growth of Electrical Control Units (ECUs) [4]. In modern vehicles nowadays, bus protocols such as FlexRay, Local Interconnect Network (LIN), Media Oriented System Transport (MOST) are used for their designated purposes [5]. Nevertheless, CAN is still being applied this industry such as in hybrid electric vehicle [6].

CAN has not only been adapted in the automotive industries, but it is also applicable in radio astronomy. The renowned observatory Combined Array for Research in Millimetre-wave Astronomy (CARMA) has adopted the CAN modules in its monitor and control systems, each with specific functions such as antenna decoder readout and oscillator tuning [7]. Furthermore, CAN has its application in data distribution service (DDS). CAN enables a service to process an information from a large number of sensor measurements to be compacted in few data packets, further reducing the time to snapshot of the state of an environment and supporting real-time requirements [8].
Furthermore, Bosch also provides an alternative and faster version of CAN called CAN with Flexible Data-Rate (CAN FD) [9]. Compared to original CAN, CAN FD has increased bit rates up to 5 Mb/s and a more robust error-detection algorithm, and can hold more data per transmission [10]. As of 2018, designers and automotive manufacturers in the US, Europe and Asia has been planning to transition their applications from the CAN to CAN FD [11].

Document Structure

The topic of this thesis is the design and implementation of the new readout protocol for the ADome. Chapter 2 specifies the programme of requirements of the new system. Chapter 3 introduces and compares several communication protocols with each other to point out why CAN is the most obvious

choice for the readout protocol. Chapter 4 discusses the CAN theory in detail. Furthermore, the STMicroelectronics implementation of CAN, the bxCAN, is explained. Chapter 5 briefly introduces the old readout implementation of the ADome and why it is necessary to have a new readout system. The implementation and verification of the improved implementation can be found in Chapter 6. Finally, Chapter 8 discusses whether the requirements of the new system has been met and also suggests future improvements.

2

Programme of Requirements

The system should be complying with the requirements below. These are divided into functional and non-functional requirements.

1. **Functional Requirements**
    - The full readout process must be real-time, i.e. less than 25 ms.
    - The system must be able to retrieve 32-bit measurement data from each antenna probe.
    - The system must only execute operations requested from the PC.
    - The MCU cables and their functionality should not disturb the radiation pattern measurement itself.
    - Each MCU must be able to store properties such as its localisation information.
2. **Non-Functional Requirements**
    - Each device in the system should be interchangeable with barely any effort required.
    - The bus in the system should accommodate at least 100 devices simultaneously.
    - The user should be able to communicate the system from the host via the serial port.

3

Analysis of Communication Protocols

This chapter discusses several serial communication protocols and explains the reason behind the choice for CAN. Popular communication protocols such as SPI and I^2C are briefly introduced. Some protocols mentioned in the state-of-the-art analysis are not considered due to their cost-complexity factor and are also becoming rare in recent times.

3.1. Controller Area Network

Controller Area Network (CAN) is an asynchronous, multi-master, robust serial communication bus using multi-cast communication [12]. Each device connected to the bus is its own master and therefore can transmit or request messages to and from another device or more. The transmitted messages can be formatted in different structures, each fulfilling a different purpose. Furthermore, CAN uses differential signalling between two bus lines in twisted-pair to transmit these messages. The differential signalling and twisted-pair create noise immunity. CAN operates at data rated from 20 kbit/s to 1 Mbit/s depending on the programming of the so-called bit timing beforehand. In addition to these, it also implements built-in error checkings such as collision detection and avoidance by arbitration on message priority, and cyclic redundancy check. These are the part that makes CAN robust.

3.2. Serial Peripheral Interface

Serial Peripheral Interface (SPI) is a high-speed (usually up to 60 MHz) synchronous communication protocol with four bus lines: Serial Clock (SCKL), Master In Slave Out (MISO), Master Out Slave In (MOSI) and Chip Select (CS). The CS bus line is used to select a slave [13]. In case of multiple slaves, additional CS lines are required. For instance, with ten devices one would require 13 lines. This would the make SPI not immensely scalable as one works with increasing number of devices.
The message size of SPI is arbitrary, dependent on the design of the application.

3.3. Inter-integrated Circuit

Inter-integrated Circuit (I2C) is a synchronous, multi-master, serial communication bus [14]. It is mainly used for many similarities between seemingly unrelated designs such as intelligent controls, general-purpose circuits and application-oriented circuits.

I2C has the following features. It consists of bidirectional two-wire bus which allows all I2C-bus compatible devices to communicate with each other. These two bus lines consist of a serial data line (SDA) and a serial clock line (SCL). SDA is responsible for transferring data between the master and slave, whereas SCL carries the clock signal. Each device connected to the bus has a unique address and simple master/slave relationships with other devices, both of which are software-driven. And since it is a multi-master bus, it contains an error scheme with detects collisions from transmission of multiple devices, and prevents data corruption via arbitration. Furthermore, it has two directions of data transfers, bidirectional and unidirectional. The former offers several modes each has a maximum data rate ranging from 400 kbit/s to 3.4 Mbit/s, and the latter offers data rates up to 5 Mbit/s. Finally,

I2C even has on-chip filtering which prevents spikes to occur on the bus line in order to prevent data integrity.

3.4. Comparison

The described serial communication protocols are compared on several points, this can be seen in Table 3.1.

	CAN	**SPI**	**I2C**
Synchronicity	Asynchronous	Synchronous	Synchronous
Wiring complexity	Simple to implement new devices	Increasingly complex when more devices implemented	Simple to implement new devices
Maximal speed	1.0 Mbit/s	60 Mbit/s	3.4 Mbit/s
No. of masters and slaves	Multi-masters	One master to many slaves	Multi-masters
Message format	Multiple structures	None	1
Different bus lines required	2	3 + Number of slaves	2
Noise immunity	High	Low	Medium
Error detection and handling	•CSMA/CD+AMP •Frame check •Bit monitoring •CRC	None	•CSMA/CD •Arbitration

Table 3.1: This describes the main differences between the described serial communication protocols: CAN, SPI and I2C.

Firstly, SPI and I2C are synchronous because both use a clock line, while CAN is asynchronous since all of the devices have their particular clock specifically programmed for CAN communication.

Secondly, SPI is typically used in point-to-point communication and it requires four different bus lines. It relatively simple with small number of nodes, however it will become increasingly complex as more devices are applied as an additional bus line is requirement for each additional slave. Furthermore, SPI has no message format thus it is up to the engineer to design a message format in order to achieve the requirements. CAN and I2C, on the other hand, are multi-master two-wire buses. These minimum number of bus lines reduce the expense for wiring and thus the complexity itself. Devices can be easily added and removed to and from these buses due to their wiring configurations. Despite the increasing complexity of SPI, it has a significantly higher maximal data transfer speed. I2C is only a bit faster than CAN, its maximal speed differs only by a factor of three.

Thirdly, CAN has a higher noise immunity due to its twisted-pair and differential signals. This makes CAN more robust compared to other two communication protocols.

Lastly, CAN and I2C has implemented some error detection and handling, where SPI has none. Therefore, SPI requires additional programming in order to mitigate errors. Comparing CAN to I2C, CAN has a greater number of error detection and handling schemes.

In conclusion, since the ADome depends on obtaining accurate measurements and the ability to add or replace new nodes, CAN would be the more appropriate choice. CAN is fairly simple and cheap, and devices can be easily implemented. It is also an remarkably robust in which the data is highly unlikely to be corrupted by noise and interruptions of other devices. And if errors do occur, it would be guaranteed to be recognised by Cyclic Redundancy Check (CRC) of the receiver and this receiver will send an error frame to the bus lines such that all devices are notified of this erroneous message.

4

Theory

This chapter describes the comprehensive theory of CAN according to the ISO 11898-1 standard and the embedded Flash memory that are used to improve the readout protocol and expand the functionality of the ADome.

4.1. Controller Area Network

CAN is a serial bus system serving as a communication protocol that supports distributed control systems [2], [15], [16], as seen in Figure 4.1. It is a multi-master bus that uses multi-cast communication, this means that any device connected to the bus can transmit a message depending on the availability of the bus. This message will be broadcasted to all devices on the bus, and these receive the message and determine whether it is of its interest. This can be seen in Figure 4.2.

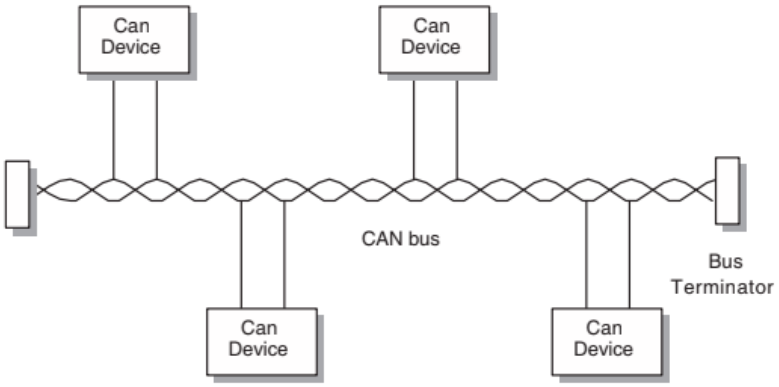


Figure 4.1: The CAN bus with devices connected to it. The bus is a twisted-pair and has bus terminator connecting to each of its end for noise immunity. This will be further elaborated [15, Figure 12-1, p.467].

4.1.1. Bus and bus states

The bus consists of a twisted-pair of two wires, CAN High (CAN-H) and CAN Low (CAN-L), which uses differential signalling. The devices are connected to this bus in a parallel configuration. Moreover, the wires are terminated with a resistor of 120Ω on each of their end together. These properties benefit is noise-immunity, such as opting for differential signalling compared to common-mode signalling. In addition, the wires are biased at the voltage level of $2.5\mathrm{V}$. The bus can be either in one of the following two states: *dominant* or *recessive*. In the dominant state, each of the two wires is put into its own respective voltage potential. CAN-H is driven to $3.5\mathrm{V}$ and CAN-L to $1.5\mathrm{V}$. Using a simple equation to determine the differential voltage:

$$\mathrm{V_{diff}} = \mathrm{V_{CANH}} - \mathrm{V_{CANL}}. \tag{4.1}$$

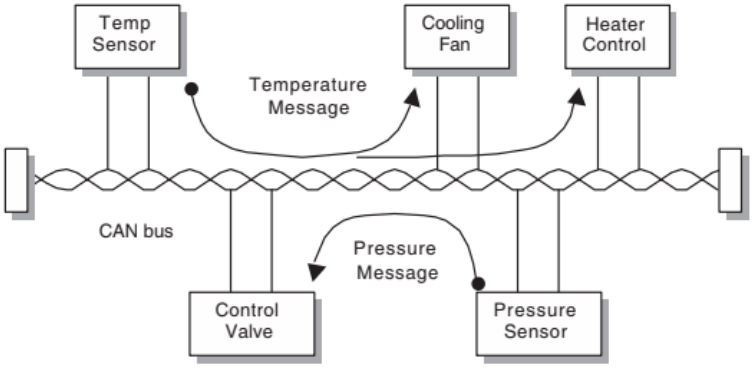


Figure 4.2: An example of specific devices on the CAN bus. Each message from one end is broadcasted on the bus and goes toward one end or more depending on their importance for these devices [15, Figure 12-3, p.470].

The differential voltage of the dominant state is $2.0\mathrm{V}$. To explain why it is called *dominant*, knowing the devices are connected in a wired-AND configuration, multiple devices can change the bus state to the dominant state. The advantage of this will be further elaborated in Section 4.1.4. The recessive state, on the other hand, both wires are instead driven to identical voltage potential of $2.5\mathrm{V}$. This gives a differential voltage of $0\mathrm{V}$. An illustration displaying the differential signalling against time t can be seen in Figure 4.3.

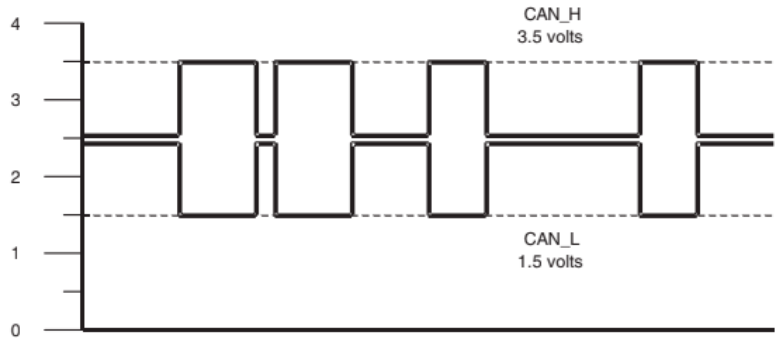


Figure 4.3: The bus state of the voltage levels of both CAN wires $\mathrm{V_{CANH}}$ and $\mathrm{V_{CANL}}$ as time advances [15, Figure 12-5, p.471].

In practice, the voltages are never exact and are always subdued by various noises. In order to determine the bus state through Eq. 4.1 while aiming to mitigate errors caused by these noises, voltages asymmetries are thus imposed. If $\mathrm{V_{diff}} \leq 0.5\mathrm{V}$, the bus is considered to be in the recessive state. If $\mathrm{V_{diff}} \geq 0.9\mathrm{V}$, the bus is instead considered to be in the dominant state. Any other voltage potential, $0.5\mathrm{V} \leq \mathrm{V_{diff}} \leq 0.9\mathrm{V}$, the bus is then considered to be in undefined. This region exists in order to differentiate the two states more properly and thus holds as a buffer between the two states for the purpose of reducing errors due to noise.

Furthermore, the dominant state can be regarded as logic '0'. During this state, the devices connected to this bus must detect that a device among them is using the bus. The recessive state should be therefore regarded as logic '1', and during this state the bus is idle and any device may use the bus to transmit. Once the bus is idle, any device may transmit, recalling the fact that the devices are connected in a wired-AND configuration.

4.1.2. Transceivers and message coding

Since the bus can be driven in two states, the transmitted messages on the bus are encoded in a sequence of logical bits. CAN uses a two-level signalling mechanism called the non-return-to-zero

(NRZ) bit encoding. Using this signalling mechanism, the logics can be more easily distinguished identified from one another as compared to rteturn-to-zero (RZ) bit encoding, as seen in Figure 4.4. A bit is transmitted for every clock cycle. This clock cycle is determined by the bit timing which is further elaborated in Section 4.1.5.

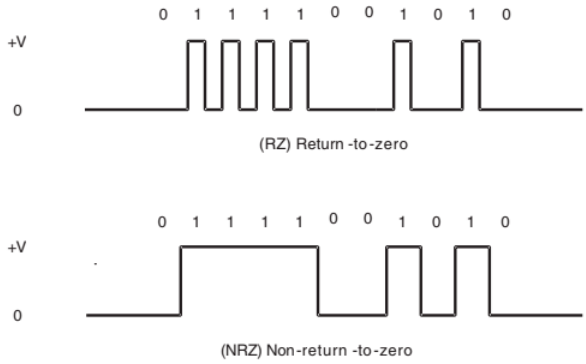


Figure 4.4: Comparison of the RZ encoding and the NRZ encoding [15, Figure 12-6, p.472].

Furthermore, every device transmits and reads the messages on the bus via its transceiver, as seen in Figure 4.5. In principle, the particular CAN controller within the device transmits a logical bit of a message to the transceiver sequentially. The transceiver then converts the logic to its corresponding bus state. For the reception of messages from the bus, the approach is contrariwise to the transmission.

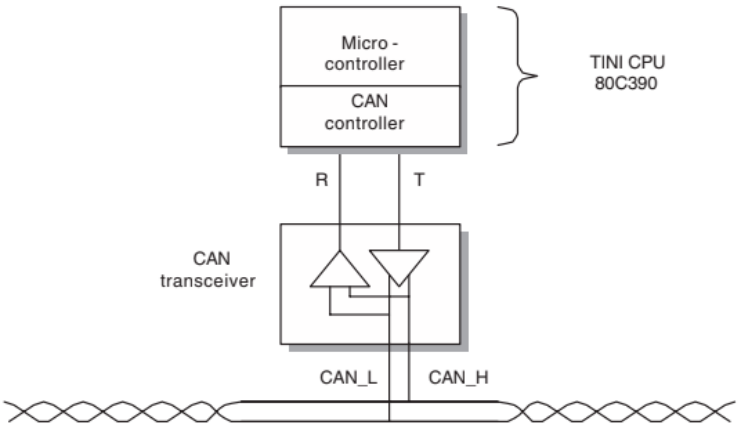


Figure 4.5: A schematic of the MCU connected to its CAN transceiver via the CAN Rx and CAN Tx ports. The CAN transceiver acts as a buffer between the MCU and the CAN bus [15, Figure 12-4, p.470].

4.1.3. Message Frames

Each message on the bus is transmitted as a data packet called a *frame*. This data packet is constructed out of a message identification, necessary protocol information, a header, the data it may want to transmit, and the footer. Furthermore, CAN also allows for multiple different types of frames, each fulfills a specific purpose. These frames are further explained as followed:

- Data frame: This data packet contains any kind of data from one device to another or more.
- Remote frame: This data packet is used to request other devices for specific type of data.
- Error frame: This data packet is sent from any device when this device detects an error on the bus.
- Overload frame: This data packet is used to notify other devices to delay in order to acquire more time to process the receiving data.

- Interframe space: This particular frame is used to separate data frames (or remote frames) altogether.

There are two kinds of data/remote frame: *standard* and *extended*. Firstly, the standard data frame is, as the name suggests, the conventional form of formatting the data packet. The composition of the standard data frame can be seen in Figure 4.6.

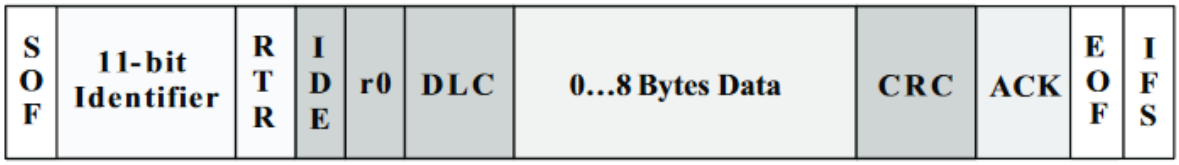


Figure 4.6: A standard data frame and its constituent fields [2, p.3].

This frame has a total length ranging from 37 bits to 101 bits, depending how many data is sent in this message. Furthermore, it also consists of the following fields in sequential order, including their corresponding bit-length:

1. **Start of Frame (SOF)** - 1 bit: this bit indicates that a device or more are pulling the bus to the dominant state.
2. **Arbitration Field** - 12 bits.
   (a) Identifier - 11 bits: this field indicates the identification of the message. It can be addressed arbitrarily by the implementation. The most significant bit (MSB) is transmitted first.
   (b) Remote Transmission Request (RTR) bit - 1 bit: this bit indicates whether the frame is a remote frame. This bit is dominant in data frame and recessive in remote frame.
3. **Control field** - 6 bits.
   (a) Reserved r0 - 2 bits: these two bits are reserved for future expansion and thus are left dominant.
   (b) Data length code (DLC) - 4 bits: these four bits together form a binary number which signifies the number of bytes present in the data field. Values larger than eight may not be used [2, p.12]. The MSB is transmitted first.
4. **Data field**- 0 to 8 bytes: this field contains the data that the device wants to transmit. MSB is transmitted first.
5. **Cyclic Redundancy Check (CRC) field** - 16 bit.
   (a) CRC sequence - 15 bit: this field is indicates the CRC code including the SOF bit, arbitration field, control field and the whole data field. Further description of the CRC is elaborated hereafter.
   (b) CRC delimiter - 1 bit: this recessive bit indicates the end of the CRC field.
6. **Acknowledgement (ACK) field** - 2 bit.
   (a) ACK slot - 1 bit: this dominant bit notifies the other devices to acknowledge the receipt of the frame.
   (b) ACK delimiter - 1 bit: this recessive bit indicates the end of the acknowledgement field.
7. **End of Frame (EOF)** - 7 bit: this whole field is pulled up at recessive state.

The main difference between a remote frame from a data frame is that the remote frame has its RTR bit is driven to recessive, and there is no data field and therefore its DLC in the control field may be ignored by the CAN controller. However, the DLC can be also instead for requesting the length of data frame and any other functionality in higher levels that the designer may wish to use.

The newer version of CAN introduces extended format [2]. The purpose of this format is to extend

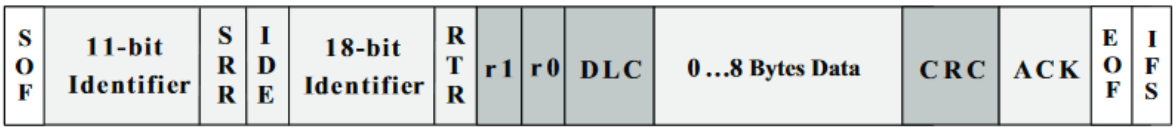


Figure 4.7: An extended data frame and its constituent fields. [2, p.4].

the limit of the standard identifier from 11-bit to 29-bit. In the extended format, the reserved bits are used and the arbitration field is extended with additional 20 bits after the 11-bit message ID. The composition of the extended format compared to the standard format can be seen in Figure 4.7.

There are few new bits introduced, in sequential order:

1. Substitute Remote Request (SRR) bit: This bit prevents messages in extended format to receive higher priority over the standard format. Therefore, this bit is set to recessive.
2. Extended Identification (IDE) bit: This bit is set to recessive to indicate that the message frame is in extended format.
3. Extended Identifier - 18 bits: This field extends the standard arbitration field.

Finally, the devices on the bus can also transmit other variants of frames in order to achieve certain objectives, as previously mentioned.
One such frame is the error frame. This is used whenever a device detects an error during transmission or reception and wants to let other devices know about the errors. Each error frame may consist one of the two error flags: active and passive. An active error flag consists of six consecutive dominant bits which should disrupt the bus traffic. This error frame then has the priority to take over the bus and letting other devices know about an urgent error such as transmit error or arbitration error, more on this in Section 4.1.4. A passive error flag consists of six recessive dominant bits. This error flag may notify other devices about an uncritical error, however it will not disrupt the bus traffic.
Overload frame is transmitted when a receiver requires more time to process a message on the bus. Finally, the interframe space is used to separate successive data and/or remote frames from one another, so that those frames will not get mixed up. These frames are internally-driven, meaning that they will be automatically written and transmitted in the CAN controller of each device.

4.1.4. Error detection and management

CAN has a considerably big arsenal of error detection schemes at its disposal, in order for CAN to work properly without any errors and inconveniences as much as possible.

Recall that each message frame contains a CRC value to verify the validity of the message. If the receiver determines no error as result of examining a frame and its CRC, then the receiver will acknowledge the receipt of the message. And if there are no devices to acknowledge the message frame, the transmitter will then know that there was an error during transmission.

In addition, each transmitter performs a frame check and bit monitoring while transmitting a message. During frame check, if the transmitter detects the bus to be in the dominant state for any of the fixed bits within the frame which are the CRC delimiter, the acknowledgement delimiter, EOF and the interframe space, the transmitter then aborts the transmission and generates an error frame. In bit monitoring, if the transmitter detects a wrong bit on the bus such that it the value is different from what it was attempting to transmit, the transmitter aborts the transmission and generate the corresponding error frame. Note that the bit monitoring does not occur during the transmission of the arbitration field, because this field is determines which message gains the priority to be on the transmitted first.

The error detection scheme used for the arbitration field is the Carrier Sense Multiple Access with Collision Detection and Arbitration on Message Priority (CSMA/CD+AMP) protocol. This is used considering only one message may materialise on the bus and the fact that CAN uses multi-cast communication. It is therefore sensible for CAN to adopt a method to avoid collisions of multiple messages simultaneously appearing on the bus.

CSMA ensures that each device connected to the bus is idle and waits for a given time before it can attempt to send a message [16]. Next, CD+AMP is a technique on how to handle and resolve when a transmitter detects a collision. One could argue that this technique is partially of the principle of CSMA

Collision Avoidance (CSMA/CA) whereby the devices avoid further collisions by becoming idle after the detection of a collision. The main idea of this technique is to prioritise the message with the lowest identifier value. This method holds both for the standard and extended format.

Whenever multiple devices each tries to transmit a message, it would have to first synchronise itself with other devices. For this reason, all devices that want to transmit start at the header, field by field, simultaneously. When the transmitting devices reach at the transmission of the identifier field, AMP will handle the collision. Knowing the fact that the devices on the bus are connected in a wired-AND configuration, any device that tries to pull up the bus to the recessive state will be easily brought down by the another device or more that tries to pull down the bus to the dominant state. Therefore, any device that tries to transmit a message of a higher message ID will soon be overwritten from those that has lower values, bit-wise. This leads to a message with the lowest message ID obtaining the priority of being transmitted broadcast first. The devices that each are transmitting a message of the higher message IDs will abort transmission and become idle for the time being. After the transmission of the successful message, the other devices will retransmit and CSMA/CD+AMP plays its part again. This cycle goes on until all devices have transmitted their messages. An example for the arbitration scheme of the transmission of three separate messages each with different message ID can be seen in Figure 4.8. Whenever one or more transmitters (that are transmitting the recessive bit) detect a collision on a given time, they abort their transmissions and goes idle.

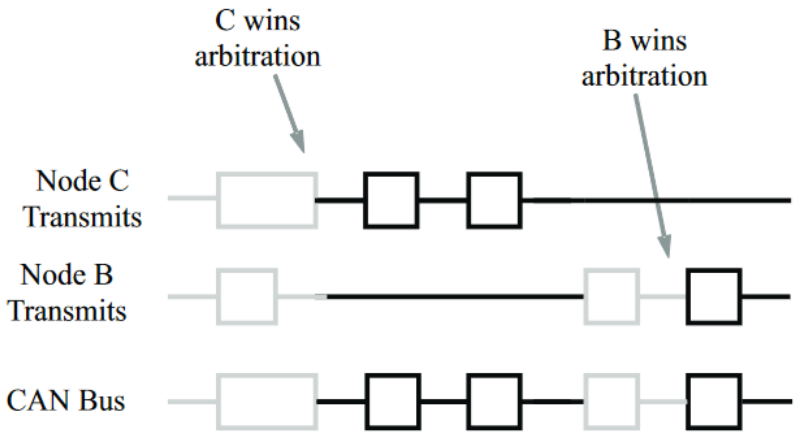


Figure 4.8: An example of arbitration during transmission of multiple devices on two different occassions. [2, p.5].

4.1.5. Bit timing

As mentioned in Section 4.1.4, the message transmission from every device is synchronous. Despite the fact that each device has its own internal clock, it has to have another clock system in order for it to synchronise with the data transmission on the bus. This is determined by the design choice of the the time required to transmit a bit on the bus. However, note that the new clock system and the nominal bit time are not the same. Rather, the new clock system defines a new time unit called *time quantum* $\mathrm{T_Q}$. This time unit is derived from a programmed clock frequency designated for CAN. In addition to this, some fraction of the clock in the transmitter is determined by the *bit rate prescaler*. Each transmitted bit may consist time quanta ranging from eight to a maximum of 25 time quanta with each segment contains different number of time quanta.

CAN divides the nominal bit time into four time segments in sequential order: a *synchronisation segment*, a *propagation segment*, and two *phase buffer segments*. This can be seen in Figure 4.9.

The synchronisation segment always consists of one time segment and this allows the devices to synchronise with one another. The propagation segment is used to compensate the physical delay that may occur in the network. The two phase buffer segments is used to compensate for errors appearing in the phases of the edges before and after the sample point, respectively. The second phase buffer segment has to have greater number of time quanta than that of the first phase buffer segment.

Each of these last three segments has a number of time quanta ranging from 1 to a maximum of 8.

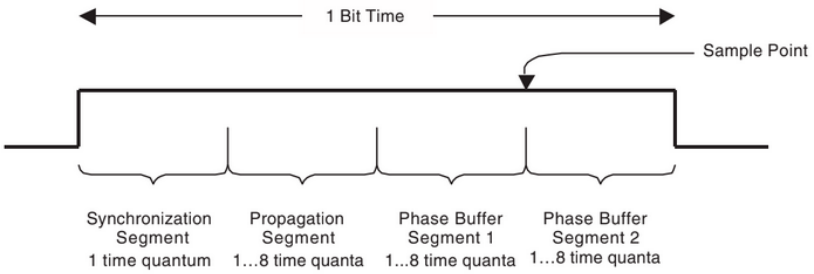


Figure 4.9: Bit time and its four constituent segments and the corresponding sample point [15, Figure 12-10, p.478].

The *sample point*, which lies between the two phase buffer segments, is the logical value for which the devices must interpret.

The nominal bit time, and in turn the bit rate, is determined by programming several of these aforementioned bit timing components. For STM32, which will be used in Chapter 6, the bit timing is determined by programming prescaler for time quanta, and the number of time quanta for the bit segment 1 (the propagation and the first phase buffer segments), bit segment 2 (the second phaser buffer segment) and the resynchronisation jump width.

The resynchronisation jump width is an additional parameter which is necessary to resynchronise the receiver clock based on the recessive-to-dominant falling edges. This is done by either by extending the first phase buffer segment or shortening the second phase buffer segment. The width of these adjustments $\mathrm{T_{SJW}}$ is described by the following inequalities:

$$0 < \mathrm{T_{SJW}} \leq 4\mathrm{T_Q}. \tag{4.2}$$

In order for the devices to be on the same level, the devices need to be synchronised with each other. The synchronisation of the devices appear at the start of the each by the falling edge of the SOF bits, and on each recessive-to-dominant falling edge in a message transmission.
In addition to synchronisation, bit stuffing is introduced which allows for the devices connected to the bus to resynchronise with the data in case of long consecutive runs of the same logical value. Recall that the error frame consists of six or more identical bits. Therefore, in order for other frames not to be mistaken as error frames, transmitter inserts a bit after five consecutive and identical bits a bit of the opposite value which alters the message. Vice versa for the receivers where it removes this bit. This also ensures that there are sufficient number of recessive-to-dominant falling edges for the devices to stay synchronised with one another.

4.2. bxCAN

Finally, to conclude the section on the principle of CAN bus, it is appropriate to outline the additional features that appear in the CAN peripheral interface of the MCU called *Basic Extended CAN* (bxCAN) [17]. This version supports standard and extended data frames, it also supports time triggered communication mode to schedule transmission based on time and achieve time synchronisation, allows for bit rates up to $1\ \mathrm{Mbit/s}$, and is designed to manage three simultaneous, to-be transmitted messages per transceiver efficiently. The bxCAN has many main features, however, only the ones that are highly effective into the design of the new and smart readout protocol will be mentioned in this thesis.

4.2.1. Transmission

When a device wants to transmit multiple messages at once, the transmission of one message usually takes more time than the program in the STM32 itself and sudden multiple transmission of messages would create transmission errors. In order to circumvent this problem, the bxCAN therefore implements three transmit mailboxes. Each of these mailboxes stores one message that the device wants to transmit. The order of which these stored messages are sent depend on which the Transmission Scheduler decides which mailbox goes first. The Transmission Scheduler is configurable either to prioritise the transmission of messages depending on the time sent, First In First Out (FIFO) priority, or in the order of the mailboxes itself.

When a stored message is sent, the corresponding transmit mailbox is emptied and becomes accessible for the next message. If the mailboxes of a device are fully filled and this device requests to send an additional message, it will detect a transmit error. This problem is up to the engineer to design a way to handle this. Lastly, a time stamp can be applied on the SOF. One receiver or more could then read on which exact time this message is sent.

4.2.2. Reception

In contrast to transmit mailboxes, bxCAN has two separate receive FIFO mailboxes where each can hold up to three messages at a time. When a device wants to read and copy a message stored into a receive FIFO, it will read the first message that has arrived. After reading this message, it is then removed and the message after it will be the next one in line. When a receive FIFO is fully filled and it receives another message, the new message will either replace the last received message in the receive FIFO or be discarded.

Furthermore, messages are first filtered on their identifiers through one or more filter banks before they reach the receive FIFOs. This is a great feature because it is performed on the background. When a message goes through a filter bank, it will be accepted to the corresponding receive FIFO that the filter bank is assigned to if it fulfills certain criteria. Otherwise the message will move on the next filter bank. After it reaches the final filter bank and has not yet fulfilled any criteria, the message will then be finally discarded. An example of this can be seen in Figure 4.10. The number corresponding to the filter bank of which matches with the message ID can be retrieved as the Filter Match Index (FMI).

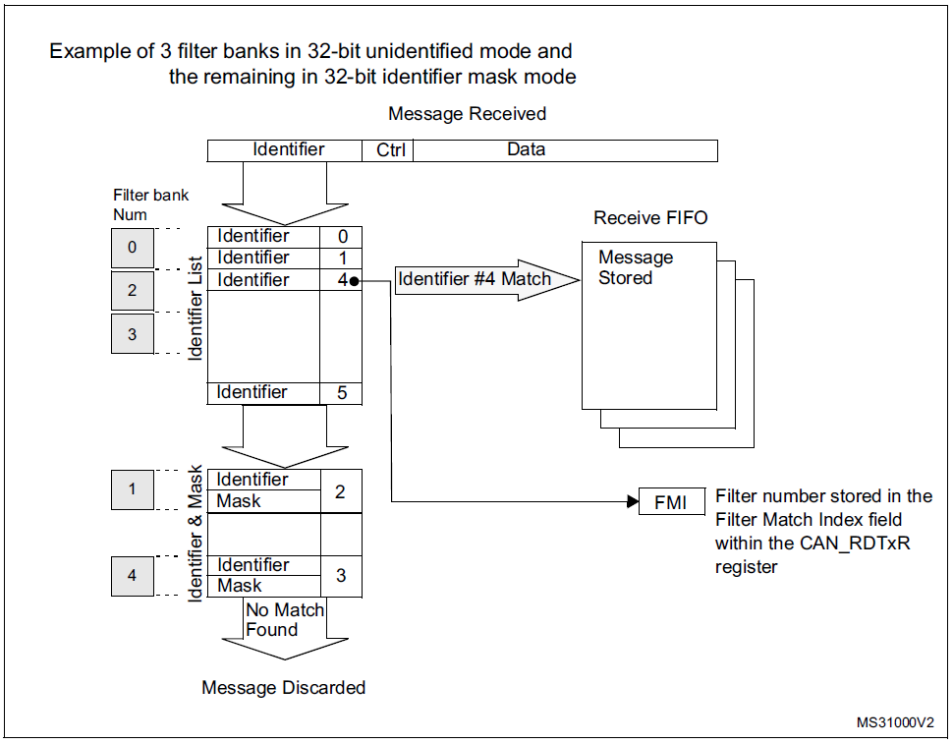


Figure 4.10: An illustration of a message ID of a received message passing through a number of filter banks. The message ID ultimately matches with the identifier #4 of the list of IDs in filter bank number two and the message gets placed into the receive FIFO. The number of the filter bank of which matches with the message ID can be retrieved as the Filter Match Index (FMI) [18].

The criteria that messages must fulfils depend on the message ID and which one of the two filter modes in a given filter bank is activated: *ID list mode* and *ID mask mode*. In ID list mode, there can be up to four IDs the messages must contain in order to be accepted. In ID mask mode, in order for a message to be accepted, its message ID is masked and compared to the filter ID, as described in the following condition:

$$\mathrm{ID}_{\mathrm{message}} \,\&\, \mathrm{ID}_{\mathrm{mask}} = \mathrm{ID}_{\mathrm{message}} \,\&\, \mathrm{ID}_{\mathrm{filter}}, \tag{4.3}$$

where $\mathrm{ID}_{\mathrm{message}}$, $_{\mathrm{mask}}$ and $\mathrm{ID}_{\mathrm{filter}}$ are the the message ID, mask and filter ID, in bits, respectively. This

filter mode can accept huge number of different IDs depending on the combination of the mask and the filter ID. Each bit has its use in masking: bit '1' means *compare*, whereas bit '0' means *don't care* [12]. For instance, one could use the mask $0xFFFF$ to such that the message ID has to exactly match with the filter ID. Whereas the filter ID and mask $0x0$ means that all message IDs are welcome instead.

4.3. Embedded Flash Memory

The Flash memory is a non-volatile memory able to store digital information and retain it even when the power of the device is switched off. It is a type of Electrical Electrically Erasable Programmable Read-Only Memory (EEPROM). The embedded Flash in the STM32L432KC MCU integrated circuit (IC) has a total size of 256 Kbytes, divided into 128 pages for arbitrary memory retention. A page is defined as a block of memory mapped to a set of addresses [19]. Each of these pages has 2048 available addresses, giving a total of 2 Kbytes available for use. [18].

Flash area	Flash memory addresses	Size (bytes)	Name
Main memory	0x0800 0000 - 0x0800 07FF	2 K	Page 0
	0x0800 0800 - 0x0800 0FFF	2 K	Page 1
	0x0800 1000 - 0x0800 17FF	2 K	Page 2
	0x0800 1800 - 0x0800 1FFF	2 K	Page 3
	-	-	-
	0x0801 F800 - 0x0801 FFFF	2 K	Page 63(1)
	-	-	-
	0x0803 F800 - 0x0803 FFFF	2 K	Page 127(2)
	-	-	-
	0x0807 F800 - 0x0807 FFFF	2 K	Page 255(3)
Information block	0x1FFF 0000 - 0x1FFF 6FFF	28 K	System memory
	0x1FFF 7000 - 0x1FFF 73FF	1 K	OTP area
	0x1FFF 7800 - 0x1FFF 780F	16	Option bytes

1. Main Flash memory space of 128K devices is limited to page 63.
2. Main Flash memory space of 256K devices is limited to page 127.
3. Main Flash memory space of 512K devices is limited to page 255.

Figure 4.11: Embedded Flash Memory Organization: The STM32L432KC can use page number 0 - 127, each with 2 Kbytes of size [18, p.78].

As shown in Figure 4.11, each page can store up to 2 Kbytes of data. The default state of the flash is 1. Turning a 1 into 0 is called byte programming. Turning a 0 into 1 is called erasing.

5

Readout Protocols

In this chapter, the implementation of the new readout protocol is described. The advantages and disadvantages of the new prototype, compared to the current integration in the ADome, are also discussed. First, a brief introduction is given to the old readout protocol. Then, the new readout protocol is introduced and finally, a comparison is made between the two.

For both implementations, the readout is done by retrieving data bit-wise from the 12-bit ADCs, which are connected to the power meters consecutively.

5.1. Current Readout Protocol

The old readout protocol, depicted in Figure 5.1, is currently integrated in the ADome using Serial Peripheral Interface (SPI) communication protocol with logic elements to send the measurement data to the computer. The SPI buses are connected to ADCs of the probes. Using enable signals, the data from the ADCs are transmitted to the MCU through the logic elements. As shown in Figure 5.1, the probes are serially connected which introduces dependencies between them. A failure in one of the nodes could lead to data losses of the subsequent nodes. Furthermore, one has to design the structure of the data packet and apply error-detection schemes and handling in order to decrease the likelihood of passing errors into the PC. This would heavily increase the complexity of the system.

Furthermore, this protocol is only able to acquire one data sample at one node at a time. The computer can request multiple successive readout instructions and perform data averaging at the computer itself to acquire a more representative measurement. The number of messages present on the bus lines $\mathrm{N}_{\mathrm{Message}}$ is therefore:

$$\mathrm{N}_{\mathrm{Message}} = \mathrm{N}_{\mathrm{Nodes}} \times \mathrm{N}_{\mathrm{Readout}} \tag{5.1}$$

where $\mathrm{N}_{\mathrm{Nodes}}$ is the total number of nodes and $\mathrm{N}_{\mathrm{sample}}$ the number of samples. This means that an increase of nodes or number of samples would lead to more traffic on the bus and therefore, resulting in a higher probability of bit and bus error.

5.2. New Readout Protocol

The new readout protocol, depicted in Figure 5.2, uses the CAN standard to communicate between the antenna nodes as mentioned in Chapter 1. The concept of one host and multiple slaves is still present in this readout since the main functionality of the system is to transfer measurement data from every node to the PC. Due to the presence of MCUs at each node, multiple data sampling from the ADC and data averaging are performed locally. This allows the number of readout messages on the bus lines to be:

$$\mathrm{N}_{\mathrm{Message}} = \mathrm{N}_{\mathrm{Nodes}} \tag{5.2}$$

Therefore, the bus lines have less traffic compared to the current implementation, resulting in a more robust communication. Although the CAN messages have a longer frame, which could lead to a higher chance of bit errors to occur. The error detection schemes implemented in the CAN protocols will manage this issue as described in Section 4.1.4. Nevertheless, the increment of bits in the messages

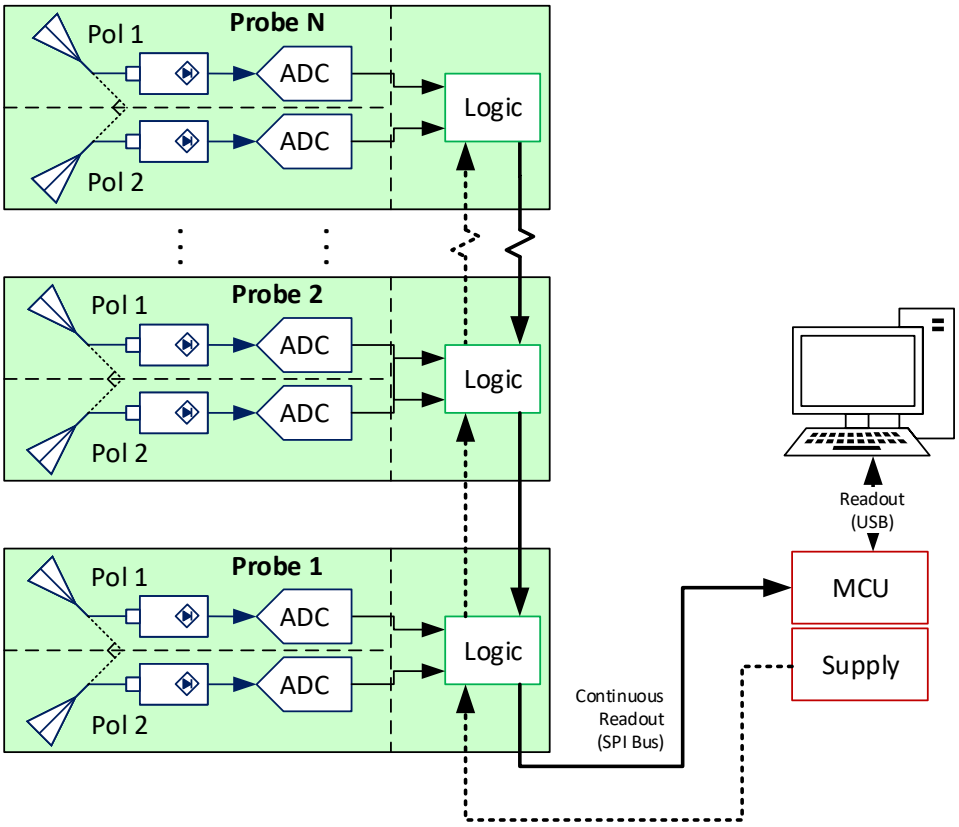


Figure 5.1: The schematic of the current setup which is used to read out all the antenna points.

could lead to a rise of the full readout time. However, inasmuch as the duration of this process can yet be considered as real-time for the user, the system requirements are still fulfilled.

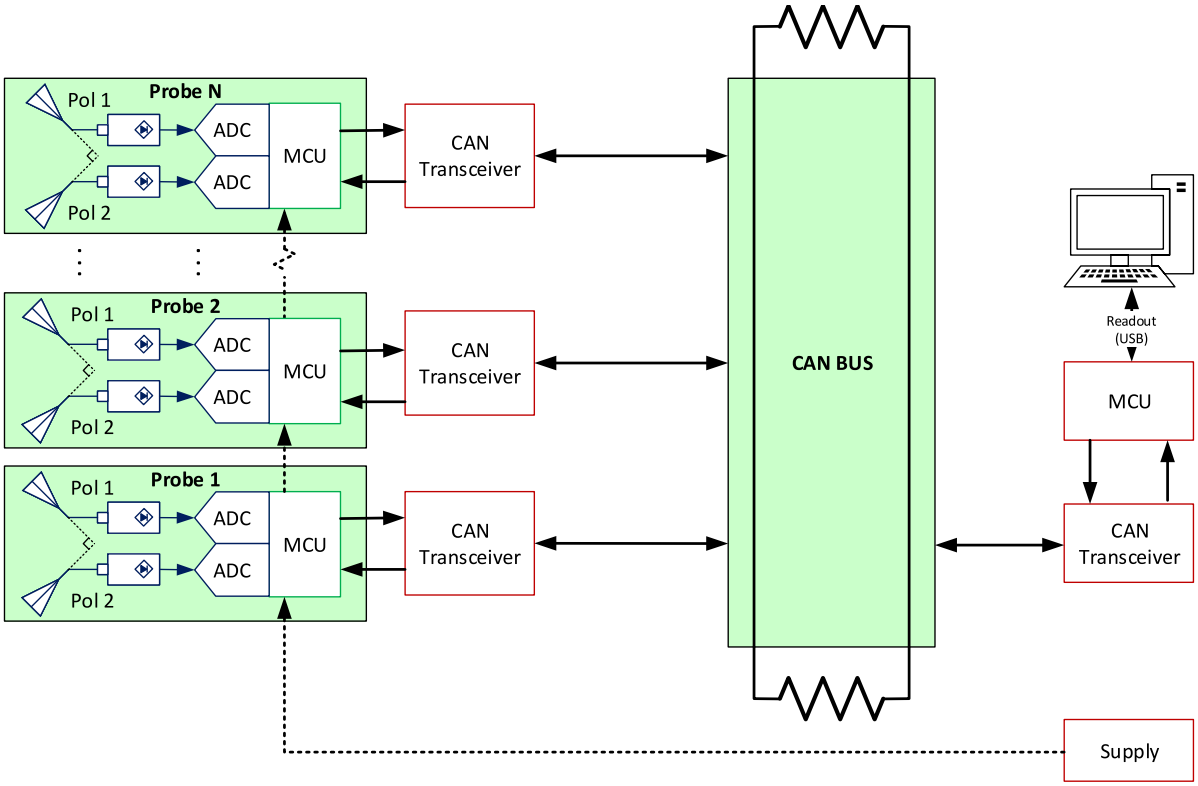


Figure 5.2: The schematic of the new setup which is used to operate on all the antenna points. The 120 Ω termination resistor of the CAN bus is also shown.

6

Implementation

In this chapter, the working of the new readout protocol is further elaborated. As described in Figure 5.2, Each node has a MCU in order to process CAN and its functionality. This MCU is the ultra-low-power microcontroller STM32L432KC that has a built-in CAN interfaces bxCAN, an embedded Flash memory of 256 KByte and a unique 96-bit ID that differentiates one MCU from another [17]. Moreover, each node connects to the bus via the CAN transceiver CAN FD 4 Click which is compatible for CAN FD as well as the original CAN [20].
There are two kinds of nodes: the *host* and the *slaves*. The slaves are responsible for creating measurements, whereby the host is responsible for managing which actions the slaves should do and act as a port between the bus and the PC. Aside from sending data between these kinds of nodes, they should also be able to request the other to do certain a task or more. These requests have in the form of data frames with DLC of one. In the data field, the request may contain one of the following codes and their corresponding description:

- ***REQUEST_MEASUREMENT_START*** - Host requests the slaves to start creating measurements.
- ***REQUEST_LOCALISATION_START*** - Host requests the slaves to start localisation.
- ***REQUEST_LOCALISATION_READY*** - Host induces a slave to be localised.
- ***REQUEST_LOCALISATION_ACK*** - Host acknowledges a slave that it has been localised and be ready to receive its location data.
- ***REQUEST_INITIALISATION_WARN*** - A slave warns the host that it has not yet been localised.

Nodes can differentiate these codes, because they are defined and known to the devices beforehand. The schematic on the process of the new readout protocol both for the host and the slaves can be seen in Figure 6.1. After the initialisation of the MCUs, the user may wish to either to obtain measurements or to localise the nodes. The latter may be necessary in case nodes have been interchanged. The main design of the measurement readouts, the localisation and the initiation of the MCUs are further explained in the following sections.

6.1. Localisation scheme

Considering the fact that the devices on the bus should be effortlessly interchangeable, this means that the any device may not have yet been assigned to an angular location. Therefore, the localisation of each node is required. The localisation scheme ensures that the slaves are identified by figuring out their angular positions, in terms of the azimuth angle ϕ and the polar angle θ. The schematic of this scheme can be seen in Figure 6.2.

This consists of two stages: the collection of identifiers and localisation of each node. During the collection of identifiers, the host requests by sending a remote frame containing the code ***REQUEST_LOCALISATION_START*** in the DLC. The slaves receives the the remote frame and recognise the code to start localising by returning a remote frame containing their unique ID. In the meantime,

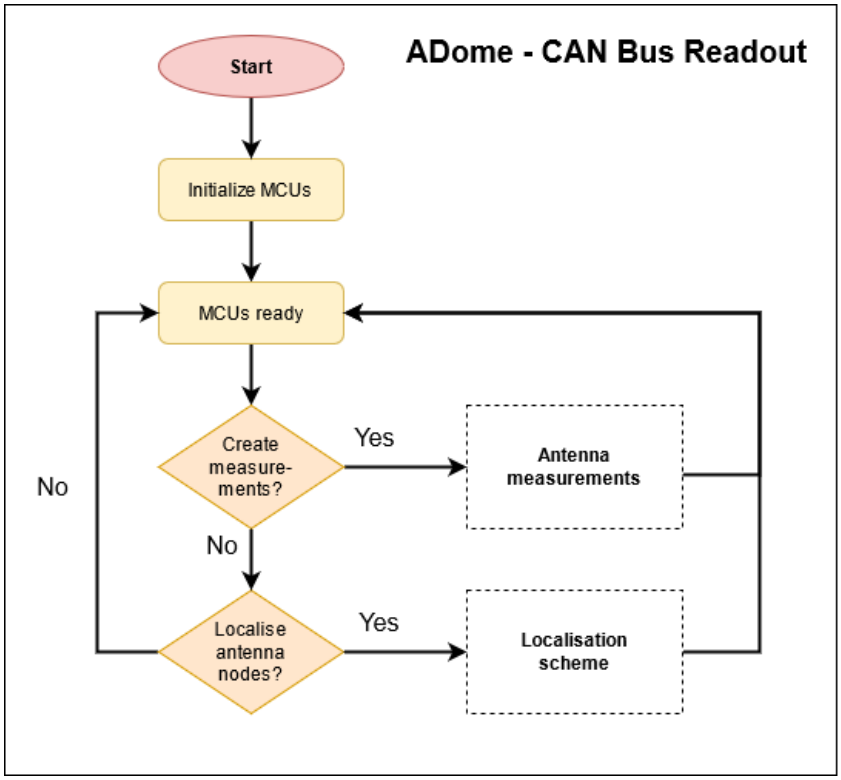


Figure 6.1: The flowchart diagram of the new readout protocol where the readout and localisation schemes are considered as black boxes for the meantime.

the host starts a timer and collects the incoming remote frames successively. When a host receives a remote frame, it collects the ID of the message from which node it originates from and store it into a list. Then the host resets the timer and collects another remote frame. When there is no frame to receive, the timer runs out, and the host finds a total number of identified nodes N and moves on two the next stage.

During this stage, the host goes through the list of identifiers and requests one slave at a time to be localised by sending a data frame containing the code ***REQUEST_LOCALISATION_READY***. This actuates the slave to activate its LED upon receiving the request, therefore one LED is on in the whole system at a time. The PC should localise the angular position of the LED via the optical part, which is outside the scope of this project. After this, the PC sends the location in two 32-bit angles to the host. The host saves the ID of the localised slave and its location into a list. In addition, the host assigns this slave a node number n which goes sequentially in the list $0 \leq n \leq N$. Afterwards, it then requests the localised slave with a data frame containing the code ***REQUEST_LOCALISATION_ACK*** so that the slave may deactivate its LED and be ready to receive the location. After this, the host transmits data frame containing the location which the slave then receives and saves into its own flash memory. After this, the host moves on to the next slave until all identified slaves are localised. Finally, the host sends a copy of the list of slaves and their locations to the PC. This ensures that the locations will not need to be sent at every obtainment of measurements.

6.2. Readout scheme

The readout scheme on obtaining the measurements of a given moment is fairly straightforward. The schematic of the readout scheme can be seen in Figure 6.3.

The host requests measurements by sending a data frame to all. This data frame contains the code ***REQUEST_MEASUREMENT_STARTS***. Each slave obtains the remote frame and recognise the code as a catalyst to create measurements. Then it creates an arbitrary number of successive measurements from each of the coupled antennas which then averages them in a 12-bit value. This data gets extended to a 16-bit to another 16-bit data of a different polarisation, giving a total data field of 32 bits.

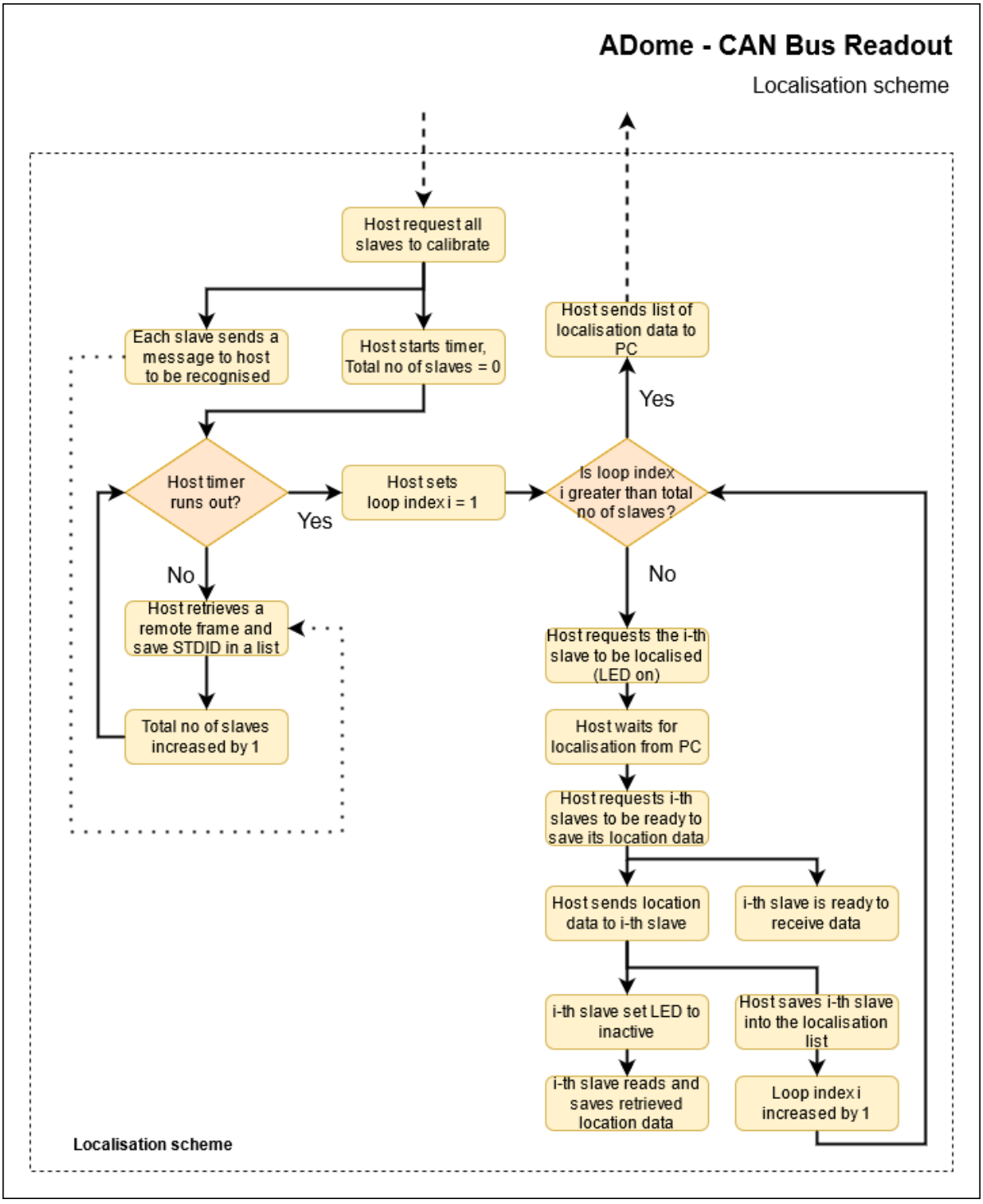


Figure 6.2: The flowchart diagram of performing out localisation where the measurement scheme is considered as a black box for the meantime.

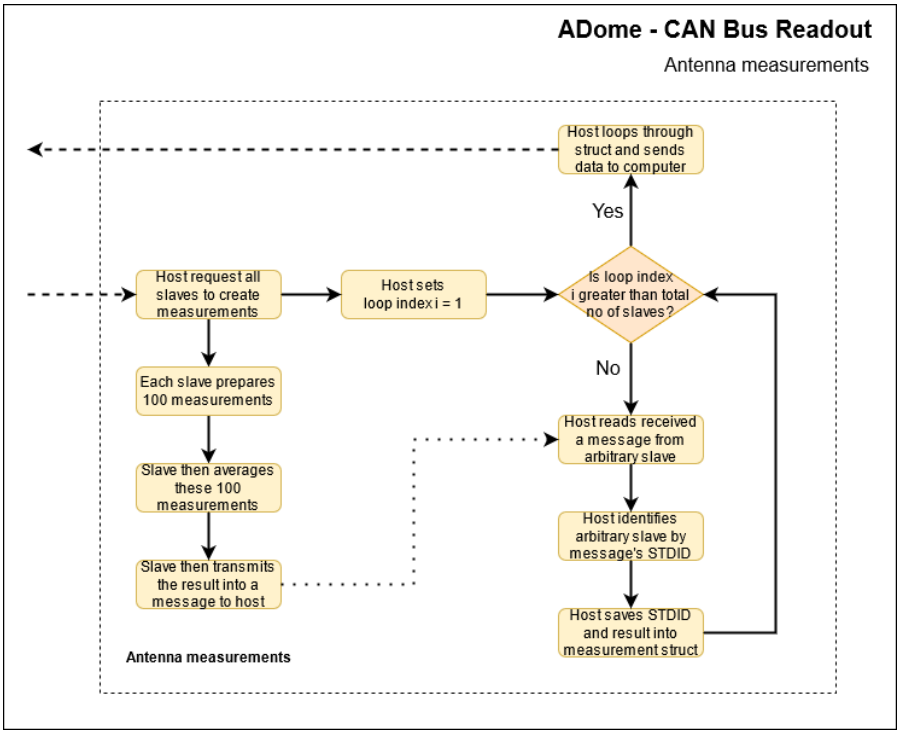


Figure 6.3: The flowchart diagram of performing out measurements where the localisation scheme is considered as a black box for the meantime.

This extension ensures that the PC can easily differentiate the two data of different polarisations as two bytes. Finally, the slave transmits its 32-bit data in a data frame including its particular node number towards the host. In practice, there may be 100 or more transmissions simultaneously. CAN handles this issue via CSMA/CD+AMP as described in Section 4.1.4, prioritising the transmissions based on the message identifiers incrementally.

In the meantime, after the transmission of the request from the host and assuming that the host knows the total number of slaves beforehand, it will collect all incoming data frames successively. When a data frame is collected, the host retrieves the ID of the message from which node it originates from, and the associated two 12-bit data. The ID and the data are then saved into a list of measurements. This cycle goes on until the host has collected a measurement from each node. This list of measurements is then sent to the PC.

6.3. Initialisation

The initialisation of a MCU is subdivided into the three stages: obtaining an ID, configuring the filters, and verifying identified slaves within the system.

6.3.1. Obtaining an ID

When the system is switched on, the MCUs are first initialised in order for them to work as intended. During the initialisation, the ID of the given MCU is determined first. Each MCU has a unique and pre-programmed 96-bit digit that is based on the identity of the wafer and the location on that wafer it originated from. So, the 32-bit word that describes the location is used to calculate the ID. The reason for this is that the available that the available MCUs are all made of the same wafer, they only differ from each other from their positions. This word consists of the X and Y coordinates, each containing a value of 16 bits, in arbitrary position respectively. The 11-bit ID, which is the size of the standard identifier, is calculated through the bit-wise AND & and bit-wise OR | operators, as described in Equation 6.1.

$$((X \,\&\, 0xFF00) \mid (X \,\&\, 0x00FF)) \,\&\, 0x3FE, \tag{6.1}$$

where X is the 32-bit word describing the X and Y coordinates on the wafer of the MCU. Using $0x3FE$ as mask limits the range of the calculation. This gives a range of identifiers from $0x000$ to $0x3FE$, giving a

total of 1022 different nodes each with a different ID. However, there will be an issue when two devices require the same ID after calculation and this would create an arbitration error during transmission. This issue requires further solution in the future.
The MSB of the ID field $0x400$ is reserved for a node to receive messages that is specifically made for it. To obtain its receiver ID, it calculates by taking its message ID with $0x400$ in a bit-wise OR (OR) operator, as described in Equation 6.2.

$$0x400 \leq \mathrm{X_{ID}}\ \mathrm{OR}\ 0x400 \leq 0x7FE, \tag{6.2}$$

where $\mathrm{X_{ID}}$ is the 11-bit ID of the node. The ID $0x7FF$ is left reserved for the broadcast ID which appears in every broadcast message that come from the host to all of the slaves. The host is determined by selecting a device via its ID.

6.3.2. Configuration the filters

After determining the IDs, the filters which is used to block insignificant messages are implemented.

Firstly, the host requires a particular receive FIFO which receives all message identifiers in the range of $0x000$ to $0x3FE$. This channel is achieved by creating a filter in ID mask mode where the filter ID is assigned with value of $0x400$ and a filter mask of $0x3FE$. Using these IDs, the filter allows the messages containing the IDs within the prescribed range into the particular channel.

Secondly, each slave has two different channels: *particular* and *broadcast*. Both channels use the filtering ID list mode. The particular channel is used to retrieve messages that are only intended for this given node. To achieve this, this channel uses the receiver ID of this node. The broadcast channel, on the other hand, is used to retrieve messages that are intended for all slaves. This has the broadcast ID of the host as the filter ID.

6.3.3. Verifying identified nodes

Finally, all of the nodes will verify their angular location in their flash memory. This will the system more smart. The host starts a timer and reads every incoming message in the meantime. Meanwhile, each slave reads the flash memory allocated to the location. There are two scenarios:

1. If the slave finds an arbitrary value, it does nothing. If the host does not receive any message and the timer runs out, the host acknowledges that all slaves have been localised. However, note that this method does not prevent a swap or more between localised slaves in the system.

2. If it does found to be empty, the slave warns the host by sending a data frame containing the code ***REQUEST_INITIALISATION_WARN***. When the host receives this message or more after the timer runs out, it then sends a notification to the PC to warn that one slave or more are unidentified and thus the system needs to start localising.

7

Verification

In this chapter, two of the implementations discussed in Chapter 6 will be verified. The setup that is used to verify the implementations consists of one host and two slaves, this can be seen in Figure 7.1. The slave nodes are connected to the antenna which are exposed to a RF source.

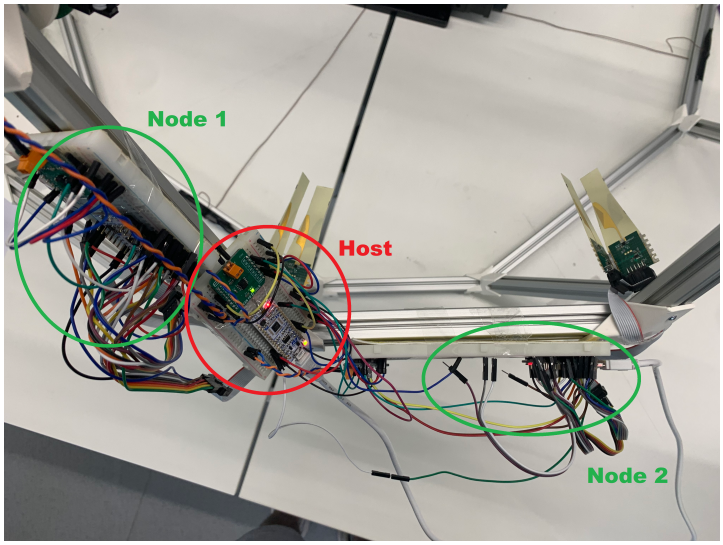


Figure 7.1: The setup of the verification with three nodes attached to the ADome.

7.1. Localisation scheme

The verification of the localisation scheme and the initialisation is as followed. Firstly, the total number of slaves that the host has identified must be equivalent to the actual number of slaves terminated to the bus.The transmission of the locations from the PC host to their associated nodes via the host must be validated as well. In order to verify the second stage of the localisation scheme, a localisation algorithm must be implemented in the ADome. This has not yet been fulfilled. Therefore, a test case has been in which dummy locations has been created. During the localisation of each node, the dummy angular position, the azimuth angle ϕ and the polar angle θ, are created in the PC. This position is generated in steps of $\pi/12$ rad for every increasing node number n, where θ lags behind ϕ. as described in the following two equations:

$$\phi = \frac{\pi}{12}(n+1), \tag{7.1}$$

$$\theta = \frac{\pi}{12}n. \tag{7.2}$$

Each of the two slaves should attain the prescribed value for each property as described in Table 7.1. Afterwards, the host return the this data to the PC via the serial port.

	Node 1	**Node 2**
Node number	0	1
Device ID	57	60
ϕ	$\pi/12$ (15°)	$\pi/6$ (30°)
θ	0	$\pi/12$ (15°)

Table 7.1: The two nodes and their associated properties.

In Figure 7.2, the result of the localisation scheme from the PC can be seen. It is shown that the total number of found slaves is equal to the actual number of slaves in setup. Therefore, validating the first stage of the localisation scheme. Furthermore, the host has also successfully assigned the other properties to their associated slaves.

```
The total number of slaves: 2
  Node Number: 0  1
 Azimuth Angle: 15  30
   Polar Angle: 0  15
   Device ID: 57  60
```

Figure 7.2: Result showing the properties of each node. The properties of the second node are shifted to the right. All angles are in degrees.

7.2. Readout scheme

The verification of the readout scheme is performed with two slaves terminated to one host. In addition, this setup is also verified where the slave is in and out of the line of sight of a powered, transmitting antenna, respectively. This setup can be seen in Figure 7.1. As described in Section 6.2, the PC requests the host via serial port to initiate the slave to create measurements. The slave activates its ADCs and obtain the two 12-bit measurement data from each of them. Then the slave sends this data towards the host and the host in turns sends it to the PC via serial port.

As Figure. 7.3 and 7.4 shows, the resulting data is in mV voltages while the data being received are in bits. This conversion follows the following equation:

$$\mathrm{V}_{\mathrm{data}}^{\mathrm{v,h}} = \frac{\mathrm{V}_{\mathrm{ref}}}{2^{\mathrm{ADC}_{\mathrm{bits}}} - 1} \tag{7.3}$$

where $\mathrm{V}_{\mathrm{data}}^{\mathrm{v,h}}$ is the retrieved data in V. $\mathrm{V}_{\mathrm{ref}}$ is the reference voltage 2.048 V [1]. The $\mathrm{ADC}_{\mathrm{bits}}$ are the 12-bits data from one polarisation.

The found voltages for the horizontal polarisation of the antenna $\mathrm{V}_{\mathrm{ANTH-LOS}}$ and that of the vertical $\mathrm{V}_{\mathrm{ANTV-LOS}}$ for which the slave is in the line of sight can be in seen in Figure 7.3. $\mathrm{V}_{\mathrm{ANTH-OOS}}$ and $\mathrm{V}_{\mathrm{ANTV-OOS}}$ for which the slave is out of the line of sight can be in seen in Figure 7.4. The slave appears to send the appropriate value as $\mathrm{V}_{\mathrm{ANTH-OOS}} < \mathrm{V}_{\mathrm{ANTH-LOS}}$ and $\mathrm{V}_{\mathrm{ANTV-OOS}} < \mathrm{V}_{\mathrm{ANTV-LOS}}$.

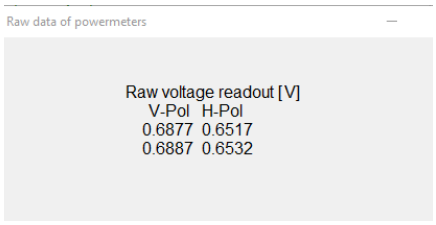


Figure 7.3: Voltage readout of the slave that is in the line of the powered antenna.

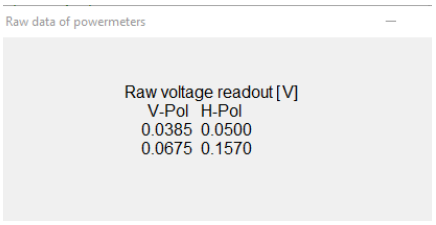


Figure 7.4: Voltage readout of the slave that is out of the line of the powered antenna.

As these measurements data are obtained from the slaves, the CAN implementation of this prototype can directly be verified as data transmission are possible.

8

Conclusion

This project serves as a study to improve the functionality of a static multi-probe measurement setup also known as the ADome. To be specific, a new system is proposed to improve the readout process of the dome and also extends the functionalities of it, such as information storage without losing this while no power is supplied. In this thesis, popular communications such as CAN, SPI and I2C are briefly studied and the choice for CAN is explained.

The speed of CAN allows real-time communication. However, the exact duration of the readout process cannot be determined due to insufficient time for testing. From the MATLAB readout process, the results were immediately obtained after sending the command and therefore, the process is real-time for the user. Whether this still holds for large number of nodes, the prototype should be able to be integrated in the ADome.

Furthermore, the ADome is able to acquire multiple measurement data from the ADCs locally and average this to obtain a more representable data, therefore reducing the bus traffic compared to the current ADome readout setup. Despite the reduction of speed over the bus lines, it still complies to the ”real-time” requirement. Besides, the location of the antennas are stored in their own memory due to the embedded Flash memory in their IC processor. This creates a more user-friendly environment as the locations are not manually inserted anymore to compute the radiation pattern. Finally, the new implementation introduces robustness in the system and reduces the amount of wire required in the ADome . Hence, the cabling costs are reduced but the costs of the antenna nodes are increased due to the local MCU.

As a conclusion, the new proposed implementation with CAN allows a simple and a low-cost system to be implemented in the ADome. A prototype was built and its functionalities are tested to fulfil the requirements. Therefore, it is recommended to put this new subsystem in use for the ADome.

Future work

Although the prototype of the new system complies to the requirements, the system can still be improved:

- The current state cannot be used in case of large number of sensing probes as this would create extremely difficult wiring problems between the MCU and the antennas. Therefore, it is recommended to develop a PCB of this prototype. This would reduce the costs of the MCU as only the processor IC is required instead of the entire evaluation board. Furthermore, the CAN transceiver can also be directly integrated on the PCB.
- Further inspection of the determination of the standard identifier. Currently, it only uses the the unique location of an arbitrary wafer. An arbitration error will occur when two MCUs were originated in identical wafer location but each from a different wafer.
- The new readout protocol could benefit from transitioning to CAN FD which would increase the data rate up to 5 Mb/s, can hold up to 64 bytes per message and makes the system more robust.

Bibliography

[1] F. Musters, “Real-time 3D characterization of antenna systems,” MSc Thesis, Delft University Of Technology, 2019.

[2] *CAN Specification Version 2.0*, Robert Bosch GmbH, 1991.

[3] T. Nolte, H. Hansson, and L. Lo Bello, “Automotive communications-past, current and future,” Jan. 2005. DOI: 10.1109/ETFA.2005.1612631.

[4] A. Albert and R. Gmbh, “Comparison of event-triggered and time-triggered concepts with regard to distributed control systems,” *Embedded World*, vol. 171902, pp. 235–252, Jan. 2004.

[5] P. Carsten, T. Andel, M. Yampolskiy, and J. Mcdonald, “In-vehicle networks,” pp. 1–8, Apr. 2015. DOI: 10.1145/2746266.2746267.

[6] K. Ismail, A. Muharam, and M. Pratama, “Design of CAN bus for research applications purpose hybrid electric vehicle using ARM microcontroller,” *Energy Procedia*,

[7] D. Woody, B. Wiitala, S. Scott, J. Lamb, R. Lawrence, C. Giovanine, S. Fredsti, A. Beard, C. Pryke, M. Loh, C. Greer, J. Cartwright, C. Gutierrez-Kraybill, A. Bolatto, and S. Muchovej, “Controller-area-network bus control and monitor system for a radio astronomy interferometer,” *The Review of scientific instruments*, vol. 78, p. 094 501, Oct. 2007. DOI: 10.1063/1.2780135.

[8] R. Rekik and S. Hasnaoui, “Application of a CAN BUS transport for DDS Middleware,” *IEEE Xplore*,

[9] *CAN with Flexible Data-Rate*, Robert Bosch GmbH, 2012.

[10] K. Lennartsson, “Comparing CAN FD with Classical CAN,”

[11] O. Esparza, W. Leichtfried, and F. González, “Transitioning applications from CAN 2.0 to CAN FD,” *CAN in Automation*,

[12] M. di Natale, H. Zeng, P. Giusto, and A. Ghosal, *Understanding and Using the Controller Area Network Communication Protocol: Theory and Practice*, 1st ed. Springer-Verlag New York, 2012.

[13] L. E. Frenzel, *Chapter Thirty-Five - Serial Peripheral Interface (SPI)*, L. E. Frenzel, Ed. Oxford: Newnes, 2016, pp. 143–145.

[14] *I^2C-bus specification and user manual*, NXP Semiconductors, 1982.

[15] B. Demuth and D. Eisenrach, *Designing Embedded Internet Devices*. Newnes, 2003.

[16] D. Paret and R. Riesc, *Multiplexed Networks for Embedded Systems: CAN, LIN, FlexRay, Safe-by-Wire..* Wiley, 2007.

[17] *STM32L432KB STM32L432KC*, STMicroelectronics, 2018.

[18] *STM32L41xxx/42xxx/43xxx/44xxx/45xxx/46xxx Advanced Arm®-based 32-bit MCUs*, RM0394 Rev 4, STMicroelectronics, 2018.

[19] D. Patterson and J. L. Hennessy, *Computer Organization and Design - The Hardware/Software Interface*. Morgan Kaufman, 2014.

[20] *CAN FD 4 Click*, Mikroelektronika D.O.O., 2015.

A

Software for Prototype Readout

A.1. Main File

```
/*
 * application_main.c
 *
 *  Created on: May 24, 2021
 *      Author: Alexander James Becoy & Remy Zhang
 *      Project: BAP ADome Subproject 1
 *
 */
#include "main.h"
#include <vector>
using namespace std;

/*****************************************************************************/
/*                 Local struct vector.                                      */
/*****************************************************************************/

struct NodeInformation
{
  int     NodeNumber;   /* Ranges from 1 to N, where N = total number of slaves.          */
  uint32_t  AzimuthAngle; /* The location of the slave in term of its azimuth angle.
      */
  uint32_t  PolarAngle;   /* The location of the slave in term of its polar angle.
    */
  uint32_t  deviceID;   /* The found standard identifier of the unique MCU of the slave.
      */
  uint8_t   data[TX_DLC_MEASUREMENT]; /* The found measurement data from the H and V antennas
     of a given moment.  */
};

NodeInformation myNodeInformation;  /* Information of this given slave. */
vector<NodeInformation> n;      /* List of identified slaves. */



/*****************************************************************************/
/*                  Local variables.                                         */
/*****************************************************************************/
CAN_FilterTypeDef sFilterConfig;

uint8_t  TxData_Measurements[TX_DLC_MEASUREMENT];
uint32_t TxMailbox;
uint32_t CAN_STDID;
uint32_t CAN_ID_RX;

int Node_Number;
int Warning_Localization;
int Slaves_Completed;
int Slaves_Total;
```

```
45 int Slave_Found;
46
47 int   data[200];
48 int   Serial_Command_Localization, Serial_Command_Measurement, DataType;
49 volatile uint8_t UARTRX[2]; //Small buffer for saving UART RX data command
50 UART_HandleTypeDef *p_huart2; //local reference to UART2
51
52 //int numOfNodes = 1; //can be varied by input
53 int sampleDelayUs = 20; // can be varied by input
54 int DataBuffer[2];
55
56 __IO ITStatus UartReady = RESET; // Flag indicating RX data available on UART2
57
58
59
60 /*****************************************************************************/
61 /*                  Local function prototypes.                   */
62 /*****************************************************************************/
63
64 /* Measurement function prototypes */
65 void slaveProcess(CAN_HandleTypeDef *hcan);
66 void sendRequest(CAN_HandleTypeDef *hcan, uint32_t SetSTDID, uint32_t RequestType);
67
68 /* Idle prototypes */
69 void hostCompleteMeasurements(CAN_HandleTypeDef *hcan);
70 void slaveSendMeasurements(CAN_HandleTypeDef *hcan);
71
72 /* Localization function prototypes */
73 void hostStartLocalization(CAN_HandleTypeDef *hcan);
74 void slaveContendLocalization(CAN_HandleTypeDef *hcan);
75 void slaveCollectLocalization(CAN_HandleTypeDef *hcan);
76
77 /* Local startup function prototypes */
78 void CAN_Filter_Init(CAN_HandleTypeDef *hcan, uint32_t CAN_ID_TX, uint32_t CAN_ID_RX,
       uint32_t setBank, uint32_t setFIFO);
79 void initalizeChannel(CAN_HandleTypeDef *hcan, uint32_t CAN_ID_TX, uint32_t CAN_ID_RX);
80 void hostInitializeLocation(CAN_HandleTypeDef *hcan);
81 void slaveInitializeLocation(CAN_HandleTypeDef *hcan);
82
83 /* Local memory function prototypes */
84 void flashDeleteNodes(void);
85 void flashErase(uint32_t PageNumber);
86 void flashOverwrite(uint32_t PageNumber, uint32_t DataIndex, uint64_t Data);
87 uint64_t flashRead(uint32_t PageNumber, uint32_t DataIndex);
88
89 /* Local CAN function prototypes */
90 void sendMessage(CAN_HandleTypeDef *hcan, uint32_t SetId, uint32_t SetRTR, uint32_t SetDLC,
       uint8_t TxData[]);
91 void waitTransmission(CAN_HandleTypeDef *hcan);
92 RxMessageTypeDef readMessage(CAN_HandleTypeDef *hcan, uint8_t RxData[], uint32_t checkFIFO);
93 int checkParticularFilled(CAN_HandleTypeDef *hcan);
94 int checkBroadcastFilled(CAN_HandleTypeDef *hcan);
95
96 /* Local serial and antenna function prototypes */
97 int readADCBits(int Polarization);
98 //void sendShiftStartbits(void);
99 //void selectNextAntenna(void);
100 void sendDataSerial(int DataType);
101 void updateSerialCommunication(CAN_HandleTypeDef *hcan);
102
103 /*****************************************************************************/
104 /*            Local main application functions.                */
105 /*****************************************************************************/
106
107 void application_init(CAN_HandleTypeDef *hcan, UART_HandleTypeDef *huart2)
108 {
109   p_huart2 = huart2; // Save pointer to UART in variable
110   // Make sure all signals are low at beginning
111   HAL_GPIO_WritePin(LD3_GPIO_Port,LD3_Pin,GPIO_PIN_RESET);
112   HAL_GPIO_WritePin(SCLK_GPIO_Port, SCLK_Pin, GPIO_PIN_RESET);
113   HAL_GPIO_WritePin(EnableConversion_GPIO_Port, EnableConversion_Pin, GPIO_PIN_RESET);
```

```
114   HAL_GPIO_WritePin(PowerMeterStartBit_GPIO_Port, PowerMeterStartBit_Pin, GPIO_PIN_RESET);
115   HAL_GPIO_WritePin(ADCStartBit_H_GPIO_Port, ADCStartBit_H_Pin, GPIO_PIN_SET);
116   HAL_GPIO_WritePin(ADCStartBit_V_GPIO_Port, ADCStartBit_V_Pin, GPIO_PIN_SET);
117
118   /* Initialize the standard identifier for the node. */
119   uint32_t Id_Buffer;
120   Id_Buffer = ((HAL_GetUIDw0() & 0x0FF00000) >> 0xC) + (HAL_GetUIDw0() & 0x000000FF);
121   CAN_STDID = Id_Buffer & 0x3FF;
122
123   /* Identify which MCU is the host. */
124   if (CAN_STDID == CAN_ID_HOST_IDENTIFIER)
125     CAN_STDID = CAN_ID_TX_HOST;
126   else
127   {
128     CAN_ID_RX = CAN_STDID | 0x400;
129     Slaves_Total = 1;
130   }
131
132   /* Initialize the filters for the host and the slaves. */
133   initalizeChannel(hcan, CAN_STDID, CAN_ID_RX);
134
135   /* Determine whether a slave or more are unidentified. */
136   if (CAN_STDID == CAN_ID_TX_HOST)
137     hostInitializeLocation(hcan);
138   else
139     slaveInitializeLocation(hcan);
140
141   char text[] = "Insert the following commands:\nB#,b# = Number of boards \nD#,d# = Sample
        delay in us \nM#,m# = Retrieve data";
142   while(HAL_UART_Transmit_IT(p_huart2, (uint8_t*) text, strlen(text)) != HAL_OK);
143   HAL_Delay(100);
144 }
145
146 int application_main(CAN_HandleTypeDef *hcan)
147 {
148   updateSerialCommunication(hcan);  /* Check on any received commands. */
149   if (CAN_STDID == CAN_ID_TX_HOST)
150   {
151
152     if (Serial_Command_Measurement == SERIAL_COMMAND_ENABLED)
153     {
154       sendRequest(hcan, CAN_ID_TX_HOST, REQUEST_MEASUREMENT_START);
155       hostCompleteMeasurements(hcan);
156       delayUS_ASM(10);
157       sendDataSerial(SERIAL_COMMAND_MEASUREMENT);
158       Serial_Command_Measurement = 0;
159     }
160     else if (Serial_Command_Localization == SERIAL_COMMAND_ENABLED)
161     {
162       sendRequest(hcan, CAN_ID_TX_HOST, REQUEST_LOCALIZATION_START);
163       hostStartLocalization(hcan);
164       sendDataSerial(SERIAL_COMMAND_LOCALIZATION);
165       Serial_Command_Localization = 0;
166     }
167   }
168   else
169   {
170     slaveProcess(hcan);
171   }
172
173   return 1;
174 }
```

A.2. General Scheme Functions

```
void slaveProcess(CAN_HandleTypeDef *hcan)
{
  /* The whole process of the antenna nodes. This function processes by probing for request messages sent by
   * the host MCU node.
   */
  uint8_t RequestCode[TX_DLC_REQUEST];  /* Declare the array to store the request. */
  static RxMessageTypeDef RequestMessage;

  if (checkParticularFilled(hcan))  /* Check whether the node received a request in its particular channel. */
  {
    RequestMessage = readMessage(hcan, RequestCode, CAN_FILTER_PARTICULAR); /* Collect the request code. */
    if (RequestMessage.DLC == TX_DLC_REQUEST)
    {
      switch (RequestCode[ARR_REQUEST_LOCATION])  /* Determine the type of request. */
      {
        case REQUEST_LOCALIZATION_READY:       /* Host requests a particular to be localized. */
          HAL_GPIO_WritePin(LD3_GPIO_Port,LD3_Pin,GPIO_PIN_SET);
          break;
        case REQUEST_LOCALIZATION_ACK:         /* Host completed localizing a particular node. */
          HAL_GPIO_WritePin(LD3_GPIO_Port,LD3_Pin,GPIO_PIN_RESET);
          slaveCollectLocalization(hcan);
          break;
        default:                       /* False alarm and do nothing. */
          break;
      }
    }
  }

  if (checkBroadcastFilled(hcan)) /* Check whether the node received a request in its broadcast channel. */
  {
    RequestMessage = readMessage(hcan, RequestCode, CAN_FILTER_BROADCAST);  /* Collect the request code. */
    if (RequestMessage.DLC == TX_DLC_REQUEST)
    {
      switch (RequestCode[ARR_REQUEST_LOCATION])  /* Determine the type of request. */
      {
        case REQUEST_MEASUREMENT_START:      /* Host starts measurement scheme. */
          slaveSendMeasurements(hcan);
          break;
        case REQUEST_LOCALIZATION_START:       /* Host starts localization scheme. */
          Slave_Found = 0;
          flashDeleteNodes();
          slaveContendLocalization(hcan);
        default:                       /* False alarm and do nothing. */
          break;
      }
    }
  }

}

void sendRequest(CAN_HandleTypeDef *hcan, uint32_t SetSTDID, uint32_t RequestType)
{
  /* This function allows MCUs create and send request to all other nodes. */
  uint8_t RequestMessage[TX_DLC_REQUEST] = {(uint8_t) RequestType}; /* Initialize the array for transmission of request. */
  sendMessage(hcan, SetSTDID, CAN_RTR_DATA, TX_DLC_REQUEST, RequestMessage);  /* Send request to all other nodes. */
}
```

A.3. Measurement Functions

```
/*-------------------------------------------------------------------------------------------*/
/*                    MEASUREMENT FUNCTIONS.                                                 */
/*-------------------------------------------------------------------------------------------*/

void hostCompleteMeasurements(CAN_HandleTypeDef *hcan)
{
  /* This function collects a measurement of each antenna of a given time
   * and verifies whether all antenna nodes have completed the measurements.
   */
  HAL_GPIO_WritePin(LD3_GPIO_Port,LD3_Pin,GPIO_PIN_SET);     /* DEBUG */
  RxMessageTypeDef Measurements_Message;
  uint8_t Measurements_Data[TX_DLC_MEASUREMENT];  /* Declare array to save the data of the
    received measurements. */

  Slaves_Completed = 0; /* Initiate slaves counter. */

  /* Collect a measurement of each antenna node. */
  while (Slaves_Completed < Slaves_Total)
  {
    if (checkParticularFilled(hcan))  /* If message is arrived in the received channel. */
    {
      Measurements_Message = readMessage(hcan, Measurements_Data, CAN_RX_PARTICULAR);

      /* Collect the received message from a node. */
      n[Measurements_Message.StdId].data[0] = Measurements_Data[0];
      n[Measurements_Message.StdId].data[1] = Measurements_Data[1];
      n[Measurements_Message.StdId].data[2] = Measurements_Data[2];
      n[Measurements_Message.StdId].data[3] = Measurements_Data[3];

      Slaves_Completed++; /* A measurement of a node is collected, move to the next antenna
    node. */
    }
  }

  HAL_GPIO_WritePin(LD3_GPIO_Port,LD3_Pin,GPIO_PIN_RESET);  /* DEBUG */
}

void slaveSendMeasurements(CAN_HandleTypeDef *hcan)
{
  /* This function drives the antenna node to collect the measurements from the antenna
   * through the ADC. After a short delay to compensate the time the host needs to initialize
   * hostCompleteMeasurements(), this is then sent to the host MCU.
   */
  HAL_GPIO_WritePin(LD3_GPIO_Port,LD3_Pin,GPIO_PIN_SET);  /* DEBUG*/
  uint8_t TxData[TX_DLC_MEASUREMENT]; /* Declare the array to store the to be sent
    measurement. */
  const int Max_Iteration = 254;      /* Number of iterations N for which the measurement
    should average from. */
  int i, ADC_Data_H, ADC_Data_V;
  int Averaged_Data_H, Averaged_Data_V; /* The found averaged data for each antenna of
    different polarization. */

  DataBuffer[POLARIZATION_V] = DataBuffer[POLARIZATION_H] = 0;  /* Initialize the buffers for
     data. */
  Averaged_Data_H = Averaged_Data_V = 0;              /* Initialize the variables for
    averaged data. */

  /* Iterate N measurements and sum them into data buffer. */
    for (i = 0; i < Max_Iteration; i++)
    {
      ADC_Data_V = readADCBits(POLARIZATION_V); // Read out 12 or 14 bits value from adc of
    both antennas
        ADC_Data_H = readADCBits(POLARIZATION_H); // Read out 12 or 14 bits value from adc of
     both antennas
        DataBuffer[POLARIZATION_V] = DataBuffer[POLARIZATION_V] + ADC_Data_V;
        DataBuffer[POLARIZATION_H] = DataBuffer[POLARIZATION_H] + ADC_Data_H;
        delayUS_ASM(1);
```

```
    }

  /* Average the found measurements and send it to the host. */
  Averaged_Data_V = DataBuffer[POLARIZATION_V]/Max_Iteration;
  Averaged_Data_H = DataBuffer[POLARIZATION_H]/Max_Iteration;
  TxData[0] = (Averaged_Data_V & 0xFF00) >> 8;
  TxData[1] = (Averaged_Data_V & 0x00FF);
  TxData[2] = (Averaged_Data_H & 0xFF00) >> 8;
  TxData[3] = (Averaged_Data_H & 0x00FF);

  sendMessage(hcan, Node_Number, CAN_RTR_DATA, TX_DLC_MEASUREMENT, TxData); /* Send the
    measurements to the host MCU. */
  waitTransmission(hcan); /* Wait until the message is successfully sent. */

  HAL_GPIO_WritePin(LD3_GPIO_Port,LD3_Pin,GPIO_PIN_RESET);   /* DEBUG*/
}
```

A.4. Localization functions

```
/*-------------------------------------------------------------------------------------------
    */
/*                LOCALIZATION FUNCTIONS.                    */
/*-------------------------------------------------------------------------------------------
    */

void hostStartLocalization(CAN_HandleTypeDef *hcan)
{
  RxMessageTypeDef Request_Message;       /* Rx header for the remote frame of a found node
    via its remote frame.          */
  uint32_t Timestamp_Start;           /* Declare variables to store starting timestamp for
    timer.          */

  int Number_Nodes_Found;             /* Initialize a variable to store the number of nodes
    that have yet been found. */
  uint8_t Empty_Data[TX_DLC_REMOTE];        /* Declare a null array to "store" the remote
    frame. */

  Timestamp_Start = HAL_GetTick();        /* Set the timer. */
  Number_Nodes_Found = 0;             /* Set the node counter to zero. */
  n.clear();                    /* Clear the previous list of nodes. */

  /* Find all the nodes. */
  while (HAL_GetTick() - Timestamp_Start < 3000)
  {
    if (checkParticularFilled(hcan))  /* Determine if a node has been found through its
    remote frame. */
    {
      Request_Message = readMessage(hcan, Empty_Data, CAN_FILTER_PARTICULAR); /* Retrieve the
     header of the found node. */

      /* Stack this node information into the list of nodes n. */
      myNodeInformation.NodeNumber    = Number_Nodes_Found;
      myNodeInformation.deviceID      = Request_Message.StdId;
      n.push_back(myNodeInformation);

      Number_Nodes_Found++;
      Timestamp_Start = HAL_GetTick();  /* Reset the timer. */
    }
  }

  HAL_GPIO_WritePin(GPIOB, GPIO_PIN_3, GPIO_PIN_SET); /* Debug */

  uint8_t Node_Index; /* Declare a variable to iterate over the stored array of standard
    identifiers. */
  uint8_t Message_NodeNumber[1];
  uint8_t Message_Location[TX_DLC_LOCATION] = {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};  /*
    Initialize a teset location*/
  uint32_t CAN_ID_RX_UID;

```

```
  for (Node_Index = 0; Node_Index < Number_Nodes_Found; Node_Index++)
  {
    CAN_ID_RX_UID = n[Node_Index].deviceID | 0x400;
    Message_NodeNumber[0] = (uint8_t) Node_Index;

    sendRequest(hcan, CAN_ID_RX_UID, REQUEST_LOCALIZATION_READY);

    /*****************************************************************************/
    /*              COMPUTER LOCALIZATION PART.                  */
    /*****************************************************************************/
    Message_Location[3] = (Node_Index + 1) * 15;  /* DEBUG */
    Message_Location[7] = (Node_Index) *15;

    /* Stack this node information into the list of nodes n. */
    n[Node_Index].AzimuthAngle = Message_Location[3];
    n[Node_Index].PolarAngle = Message_Location[7];

    sendRequest(hcan, CAN_ID_RX_UID, REQUEST_LOCALIZATION_ACK);
    waitTransmission(hcan);

    HAL_Delay(100);

    sendMessage(hcan, CAN_ID_RX_UID, CAN_RTR_DATA, 1, Message_NodeNumber);  /* Send the node
    number. */
    waitTransmission(hcan);
    sendMessage(hcan, CAN_ID_RX_UID, CAN_RTR_DATA, TX_DLC_LOCATION, Message_Location);  /*
    Theta and phi angles. */
    waitTransmission(hcan);

    HAL_Delay(100); /* Make the particular have enough time to save the info in the FLASH */
  }

  HAL_GPIO_WritePin(GPIOB, GPIO_PIN_3, GPIO_PIN_RESET);
  Slaves_Total = n.size();
}

void slaveContendLocalization(CAN_HandleTypeDef *hcan)
{
  /* Send request to the host to be localized. */
  if (Slave_Found == 0)
  {
    uint8_t Remote_Data[TX_DLC_REMOTE];
    HAL_Delay(100);
    sendMessage(hcan, CAN_STDID, CAN_RTR_REMOTE, TX_DLC_REMOTE, Remote_Data);

    Slave_Found = 1;
  }
}

void slaveCollectLocalization(CAN_HandleTypeDef *hcan)
{
  /* Obtain the information of node number and location of this node. */
  RxMessageTypeDef Localizations_Message;
  uint8_t RxMessage_NodeNumber[1];
  uint8_t RxMessage_Angles[TX_DLC_LOCATION];

  uint64_t Node_UID;
  uint64_t Node_AzimuthAngle, Node_PolarAngle;
  int LoopIndex;

  int Message_Received = 0;
  while (!Message_Received)
  {
    if (checkParticularFilled(hcan))
    {
      /* Retrieve its particular node number. */
      Localizations_Message = readMessage(hcan, RxMessage_NodeNumber, CAN_RX_PARTICULAR);
      Node_UID = (uint64_t) RxMessage_NodeNumber[0];

      /* Retrieve its particular location in terms of angles. */
      Localizations_Message = readMessage(hcan, RxMessage_Angles, CAN_RX_PARTICULAR);
```

```
110
111       /* Assemble the received messages into their corresponding datatype. */
112       Node_AzimuthAngle = Node_PolarAngle = 0;
113       for (LoopIndex = 0; LoopIndex < TX_DLC_LOCATION; LoopIndex++)
114       {
115         if (LoopIndex < 4)  /* Retrieve the azimuth angle phi. */
116         {
117           Node_AzimuthAngle = Node_AzimuthAngle | RxMessage_Angles[LoopIndex];
118           if (LoopIndex < 3)
119             Node_AzimuthAngle = Node_AzimuthAngle << 8;
120         }
121         else        /* Retrieve the polar angle theta. */
122         {
123           Node_PolarAngle = Node_PolarAngle | RxMessage_Angles[LoopIndex];
124           if (LoopIndex < 7)
125             Node_PolarAngle = Node_PolarAngle << 8;
126         }
127       }
128
129       Message_Received = 1;
130     }
131   }
132
133   /* Save the information into the Flash memory. */
134   flashOverwrite(PAGE_UID, NODE_NUMBER_PARTICULAR, Node_UID);
135   flashOverwrite(PAGE_AZIMUTH, NODE_NUMBER_PARTICULAR, Node_AzimuthAngle);
136   flashOverwrite(PAGE_POLAR, NODE_NUMBER_PARTICULAR, Node_PolarAngle);
137
138 }
```

A.5. Local Startup functions

```
1  /*-------------------------------------------------------------------------------------
       */
2  /*              INITIALIZATION FUNCTIONS.                    */
3  /*-------------------------------------------------------------------------------------
       */
4
5  void CAN_Filter_Init(CAN_HandleTypeDef *hcan, uint32_t CAN_ID_TX, uint32_t CAN_ID_RX,
       uint32_t setBank, uint32_t setFIFO)
6  {
7    CAN_FilterTypeDef sFilterConfig;  /* Declare the configuration of the RX FIFO. */
8
9    sFilterConfig.FilterScale = CAN_FILTERSCALE_16BIT;
10
11   if (CAN_ID_TX == CAN_ID_HOST)
12   {
13     sFilterConfig.FilterBank = CAN_FILTERBANK_PARTICULAR;
14     sFilterConfig.FilterMode = CAN_FILTERMODE_IDMASK;
15     sFilterConfig.FilterIdHigh = 0;
16     sFilterConfig.FilterIdLow = 0;
17     sFilterConfig.FilterMaskIdHigh = CAN_FILTERMASK_HOST << CAN_FILTERSTDID_SHIFT;
18     sFilterConfig.FilterMaskIdLow = CAN_FILTERMASK_HOST << CAN_FILTERSTDID_SHIFT;
19   }
20   else
21   {
22     uint32_t FilterID;
23     sFilterConfig.FilterMode = CAN_FILTERMODE_IDLIST;
24
25     if (setFIFO == CAN_RX_PARTICULAR)
26     {
27       FilterID = CAN_ID_RX << CAN_FILTERSTDID_SHIFT;
28       sFilterConfig.FilterBank = CAN_FILTERBANK_PARTICULAR;
29     }
30     else  /* if (setFIFO == CAN_RX_BROADCAST) */
31     {
32       FilterID = CAN_ID_RX_SLAVE << CAN_FILTERSTDID_SHIFT;
33       sFilterConfig.FilterBank = CAN_FILTERBANK_BROADCAST;
34     }
```

```
35
36      sFilterConfig.FilterIdHigh     = FilterID;
37      sFilterConfig.FilterIdLow      = FilterID;
38      sFilterConfig.FilterMaskIdHigh = FilterID;
39      sFilterConfig.FilterMaskIdLow  = FilterID;
40    }
41
42    sFilterConfig.FilterFIFOAssignment = setFIFO;
43    sFilterConfig.FilterActivation = ENABLE;
44    sFilterConfig.SlaveStartFilterBank = 0;
45
46    if (HAL_CAN_ConfigFilter(hcan, &sFilterConfig) != HAL_OK) /* Check whether the
        configuration is successfully initialized. */
47    {
48      Error_Handler();  /* An error occurred during the initialization of the filter
        configuration. */
49    }
50
51  }
52
53  void initalizeChannel(CAN_HandleTypeDef *hcan, uint32_t CAN_ID_TX, uint32_t CAN_ID_RX)
54  {
55    /* Initiate the CAN filter depending on the FIFO (FIFO0 or FIFO1).
56     * FIFO1 is the broadcast channel, which will receive every message on the bus.
57     * FIFO0 is the particular channel, which will receive all messages that is specifically
        sent to this node.
58     */
59    if (CAN_ID_TX == CAN_ID_HOST)
60    {
61      CAN_Filter_Init(hcan, CAN_ID_TX, CAN_ID_RX, CAN_FILTERBANK_PARTICULAR,
        CAN_FILTER_PARTICULAR);  /* Particular channel. */
62    }
63    else  /* if (CAN_ID_TX == CAN_ID_SLAVE) */
64    {
65      CAN_Filter_Init(hcan, CAN_ID_TX, CAN_ID_RX, CAN_FILTERBANK_PARTICULAR,
        CAN_FILTER_PARTICULAR);  /* Particular channel. */
66      CAN_Filter_Init(hcan, CAN_ID_TX, CAN_ID_RX, CAN_FILTERBANK_BROADCAST,
        CAN_FILTER_BROADCAST);  /* Broadcast channel. */
67    }
68  }
69
70  void hostInitializeLocation(CAN_HandleTypeDef *hcan)
71  {
72    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_3, GPIO_PIN_SET); /* DEBUG */
73
74    RxMessageTypeDef Initializations_Message;
75    uint8_t RequestCode[TX_DLC_REQUEST];  /* Declare the array to store the request. */
76    uint32_t Timestamp_Start; /* Declare variables to store starting timestamp for timer. */
77
78    Timestamp_Start = HAL_GetTick();
79    Warning_Localization = 0;
80
81    Slaves_Total = 0;
82    /* Find all the nodes. */
83    while (HAL_GetTick() - Timestamp_Start < 1000)
84    {
85      if (checkParticularFilled(hcan))
86      {
87        Initializations_Message = readMessage(hcan, RequestCode, CAN_RX_PARTICULAR);
88
89        if (RequestCode[ARR_REQUEST_LOCATION] == REQUEST_INITIALIZATION_WARN)
90          Warning_Localization = Warning_Localization + 1;  /* SEND WARNING */
91        else
92          Slaves_Total++;
93
94        Timestamp_Start = HAL_GetTick();
95      }
96    }
97
98    /* Send to serial port about the situation: unidentified nodes found or no issue. */
99    if (Warning_Localization > 0)
```

```
100   {
101     char text[] = "There seems to be unidentified node(s) in the system.\n Please calibrate
        new locations to these nodes.";
102     while(HAL_UART_Transmit_IT(p_huart2, (uint8_t*) text, strlen(text)) != HAL_OK);
103     HAL_Delay(100);
104   }
105
106   else
107   {
108     char text[] = "This system is ready to use for measurements.\n";
109     while(HAL_UART_Transmit_IT(p_huart2, (uint8_t*) text, strlen(text)) != HAL_OK);
110     HAL_Delay(100);
111   }
112
113   HAL_GPIO_WritePin(GPIOB, GPIO_PIN_3, GPIO_PIN_RESET); /* DEBUG */
114 }
115
116 void slaveInitializeLocation(CAN_HandleTypeDef *hcan)
117 {
118   HAL_GPIO_WritePin(GPIOB, GPIO_PIN_3, GPIO_PIN_SET); /* DEBUG */
119
120   uint64_t UID, Azimuth_Angle, Polar_Angle;
121   UID = flashRead(PAGE_UID, NODE_NUMBER_PARTICULAR);
122   Azimuth_Angle = flashRead(PAGE_AZIMUTH, NODE_NUMBER_PARTICULAR);
123   Polar_Angle = flashRead(PAGE_POLAR, NODE_NUMBER_PARTICULAR);
124
125   if ((UID == FLASH_64_EMPTY) || (Azimuth_Angle == FLASH_64_EMPTY) || (Polar_Angle ==
        FLASH_64_EMPTY))
126   {
127     sendRequest(hcan, CAN_STDID, REQUEST_INITIALIZATION_WARN);
128   }
129   else
130   {
131     sendRequest(hcan, CAN_STDID, REQUEST_INITIALIZATION_ACK);
132     Node_Number = (uint32_t) flashRead(PAGE_UID, 0);
133   }
134
135   HAL_GPIO_WritePin(GPIOB, GPIO_PIN_3, GPIO_PIN_RESET); /* DEBUG */
136 }
```

A.6. Local Memory functions

```
1  /*-------------------------------------------------------------------------------------------
       */
2  /*                  FLASH MEMORY FUNCTIONS.                        */
3  /*-------------------------------------------------------------------------------------------
       */
4
5  void flashDeleteNodes(void)
6  {
7    /* This function deletes the pages that usually contain the information of each node. This
       is
8     * used before localization in order to save the relevant and remove those that are
       irrelevant. */
9    flashErase(PAGE_UID);
10   flashErase(PAGE_AZIMUTH);
11   flashErase(PAGE_POLAR);
12 }
13
14 void flashErase(uint32_t PageNumber)
15 {
16   /*  Erases a whole FLASH page.  */
17   HAL_FLASH_Unlock();                 /* Unlock FLASH. */
18
19   FLASH_EraseInitTypeDef FlashEraseInit;        /* Initialize erase typedef.      */
20   FlashEraseInit.TypeErase = FLASH_TYPEERASE_PAGES; /* Set type erase to deleting pages.  */
21   FlashEraseInit.Page = PageNumber;          /* Starting page number to be deleted.  */
22   FlashEraseInit.NbPages = MAX_PAGE_ERASE;      /* Number of pages to be deleted = 1.   */
23   uint32_t PageError = 0;               /* Set page error.            */
```

```
24    HAL_FLASHEx_Erase(&FlashEraseInit,&PageError);     /* Calling the erase function.      */
25
26    HAL_FLASH_Lock();                 /* Lock FLASH. */
27  }
28
29  void flashOverwrite(uint32_t PageNumber, uint32_t DataIndex, uint64_t Data)
30  {
31    /*  Overwrites a FLASH page with a new data. */
32    uint64_t DataTemp[MAX_DOUBLEWORDS]; /* Initialize temporary array of retrieved doublewords
        of given page number. */
33    uint32_t LoopIndex;
34    for (LoopIndex = 0; LoopIndex < MAX_DOUBLEWORDS; LoopIndex++) /* Store the original data in
        the given page. */
35    {
36      DataTemp[LoopIndex] = flashRead(PageNumber, LoopIndex); /* Store retrieved data into the
        temporary 32-bit array. */
37    }
38
39    flashErase(PageNumber);               /* Erase the given page.  */
40
41    HAL_FLASH_Unlock();                 /* Unlock FLASH. */
42    uint32_t PageAddress;               /* Initialize page address. */
43    PageAddress = STARTING_ADDRESS + (PageNumber * PAGE_WIDTH); /*  Get the corresponding
        address of the page number.
44                                  *  Address 0x0800 0000 is the starting address of
45                                  *  the FLASH memory, and 0x0800 is the page size. */
46
47    for (LoopIndex = 0; LoopIndex < MAX_DOUBLEWORDS; LoopIndex++) /* Return the original data
        before the page erase. */
48    {
49      if (LoopIndex != DataIndex) /* Compare if the loop index is on the desired address to be
        overwritten. */
50        HAL_FLASH_Program(FLASH_TYPEPROGRAM_DOUBLEWORD, PageAddress, DataTemp[LoopIndex]);  /*
          Write the original data of its
51                                                *  corresponding page address. */
52      else
53        HAL_FLASH_Program(FLASH_TYPEPROGRAM_DOUBLEWORD, PageAddress, Data); /* Overwrite the
        new data on
54                                               * the desired address. */
55      PageAddress = PageAddress + DOUBLEWORD_WIDTH; /* Move pointer to the address of the next
        doubleword. */
56    }
57    HAL_FLASH_Lock();                 /* Lock FLASH. */
58  }
59
60  uint64_t flashRead(uint32_t PageNumber, uint32_t DataIndex)
61  {
62    /*  Reads the doubleword data on a given FLASH page number and the data index.  */
63    const uint8_t MAX_LENGTH = 2;           /* Maximum constant to create a doubleword. */
64    const uint8_t DOUBLEWORD_OFFSET = 32;       /* Offset for second half of doubleword. */
65    uint32_t PageAddress;               /* Initialize page address and data.  */
66    uint32_t * DataAddress;               /* Initialize data address.       */
67    uint32_t DataTemp[MAX_LENGTH];            /* Initialize temporary array of retrieved data.
        */
68    uint64_t Data;                    /* Initialize the storage variable for data. */
69
70    PageAddress = 0x08000000 + (PageNumber * 0x0800); /*  Get the corresponding address of the
        page number.
71                                Address 0x0800 0000 is the starting address of the
72                                FLASH memory, and 0x0800 is the page size. */
73    int LoopIndex;                    /* Starting for-loop index.       */
74    for (LoopIndex = 0; LoopIndex < MAX_LENGTH; LoopIndex++)
75    {
76      DataAddress = (uint32_t *)(PageAddress + (0x0008 * DataIndex) + (0x0004 * LoopIndex));
        /*  Calculate the address of
77      the data based on the address of the page number and the index of the desired data. */
78      DataTemp[LoopIndex] = *DataAddress;       /* Store the retrieved 32-bit data in
        corresponding index. */
79    }
80    if (DataIndex % 2 != 0)
81      Data = ((uint64_t)DataTemp[1] << DOUBLEWORD_OFFSET) | DataTemp[0]; /* Append the
```

```
    retrieved 32-bit into a 64-bit data. */
  else
    Data = ((uint64_t)DataTemp[0] << DOUBLEWORD_OFFSET) | DataTemp[1]; /* Append the
    retrieved 32-bit into a 64-bit data. */
  return Data;  /* Return the retrieved 64-bit data. */
}
```

A.7. Local CAN functions

```
/*-------------------------------------------------------------------------------------------
    */
/*                  LOCAL FUNCTIONS.                        */
/*-------------------------------------------------------------------------------------------
    */

void sendMessage(CAN_HandleTypeDef *hcan, uint32_t SetId, uint32_t SetRTR, uint32_t SetDLC,
    uint8_t TxData[])
{
  /* Send the message via the CAN bus with the corresponding ID,
   * type of frame, the length of the data, and the data itself.
   */

  CAN_TxHeaderTypeDef TxHeader; /* Declare TX header to send the message. */

  TxHeader.IDE = CAN_ID_STD;        /* Set the ID as standard identifier.              */
  TxHeader.StdId = SetId;          /* Set the incident ID into the header.           */
  TxHeader.ExtId = 0;            /* Set no extended ID into the header.            */
  TxHeader.RTR = SetRTR;           /* Set the message either as data frame or remote frame.
    */
  TxHeader.DLC = SetDLC;           /* Set the length of the data.                 */
  TxHeader.TransmitGlobalTime = DISABLE;  /* Disable the transmission of the time of the
    message sent. */

  if (HAL_CAN_AddTxMessage(hcan, &TxHeader, TxData, &TxMailbox) != HAL_OK)  /* Check whether
    the message is successfully sent. */
  {
    Error_Handler();  /* An error occurred during the transmission. */
  }
}

void waitTransmission(CAN_HandleTypeDef *hcan)
{
  /* Makes the MCU wait until the message is successfully sent and removed from the mailbox.
    */
  while (HAL_CAN_IsTxMessagePending(hcan, TxMailbox));
}

RxMessageTypeDef readMessage(CAN_HandleTypeDef *hcan, uint8_t RxData[], uint32_t checkFIFO)
{
  /* Read the message depending on the given channel and returns the corresponding standard
    id and the data length code. */
  RxMessageTypeDef RxMessage;
  CAN_RxHeaderTypeDef RxHeader;

  if (checkFIFO == CAN_RX_PARTICULAR)
  {
    if (checkParticularFilled(hcan))  /* Check whether a message is actually received in the
    given channel. */
    {
      if (HAL_CAN_GetRxMessage(hcan, CAN_RX_PARTICULAR, &RxHeader, RxData) != HAL_OK)
      /* Check whether the reading of the received message is successful. */
      {
        Error_Handler();  /* An error occurred during the reading of the received message. */
      }
    }
  }
  else  /* if (checkFIFO == CAN_RX_BROADCAST) */
  {
    if (checkBroadcastFilled(hcan))
```

```
    {
      if (HAL_CAN_GetRxMessage(hcan, CAN_RX_BROADCAST, &RxHeader, RxData) != HAL_OK)
      /* Check whether the reading of the received message is successful. */
      {
        Error_Handler();  /* An error occurred during the reading of the received message. */
      }
    }
  }
  RxMessage.StdId = RxHeader.StdId;
  RxMessage.DLC = RxHeader.DLC;
  return RxMessage;
}

int checkParticularFilled(CAN_HandleTypeDef *hcan){

  /* Check whether FIFO0 / particular channel is filled with received messages. */
  if (HAL_CAN_GetRxFifoFillLevel(hcan, CAN_RX_PARTICULAR) > 0)
    return FIFO_FILLED; /* There is at least one message in the particular channel. */
  else
    return FIFO_EMPTY;  /* There is no message in the particular channel. */
}

int checkBroadcastFilled(CAN_HandleTypeDef *hcan)
{
  /* Check whether FIFO1 / broadcast channel is filled with received messages. */
  if (HAL_CAN_GetRxFifoFillLevel(hcan, CAN_RX_BROADCAST) > 0)
    return FIFO_FILLED; /* There is at least one message in the broadcast channel. */
  else
    return FIFO_EMPTY;  /* There is no message in the broadcast channel. */
}
```

A.8. Local Serial and Antenna functions

```
/*-------------------------------------------------------------------------------------------*/
/*              LOCAL SERIAL AND ANTENNA FUNCTIONS.                  */
/*-------------------------------------------------------------------------------------------*/

int readADCBits(int Polarization)
{
    // Perform conversion by raising chip select / conversion pin
    // Give pulse and wait short time to allow conversion (min 1.4 ns)

  HAL_GPIO_WritePin(PowerMeterStartBit_GPIO_Port, PowerMeterStartBit_Pin, GPIO_PIN_SET);

  int i;
    int adc_read_value = 0;

    if (Polarization == POLARIZATION_V)
    {
      HAL_GPIO_WritePin(ADCStartBit_V_GPIO_Port,ADCStartBit_V_Pin, GPIO_PIN_RESET);
      for(i = 0; i < ADC_bits; i++)
      {
      //serial clock is global for all boards
      HAL_GPIO_WritePin(SCLK_GPIO_Port, SCLK_Pin, GPIO_PIN_SET);
      int misoVal = (int) HAL_GPIO_ReadPin(ADCIN_V_GPIO_Port, ADCIN_V_Pin);
      adc_read_value += (misoVal > 0) << (ADC_bits  - i - 1);
      HAL_GPIO_WritePin(SCLK_GPIO_Port, SCLK_Pin, GPIO_PIN_RESET);
    }
      HAL_GPIO_WritePin(ADCStartBit_V_GPIO_Port,ADCStartBit_V_Pin, GPIO_PIN_SET);
    }
    else  /* if (Polarization == POLARIZATION_H) */
    {
      HAL_GPIO_WritePin(ADCStartBit_H_GPIO_Port,ADCStartBit_H_Pin, GPIO_PIN_RESET);
    for(i = 0; i < ADC_bits; i++)
    {
      //serial clock is global for all boards
      HAL_GPIO_WritePin(SCLK_GPIO_Port, SCLK_Pin, GPIO_PIN_SET);
```

```
      int misoVal = (int) HAL_GPIO_ReadPin(ADCIN_H_GPIO_Port, ADCIN_H_Pin);
      adc_read_value += (misoVal > 0) << (ADC_bits  - i - 1);
      HAL_GPIO_WritePin(SCLK_GPIO_Port, SCLK_Pin, GPIO_PIN_RESET);
    }
    HAL_GPIO_WritePin(ADCStartBit_H_GPIO_Port,ADCStartBit_H_Pin, GPIO_PIN_SET);
    }
    HAL_GPIO_WritePin(PowerMeterStartBit_GPIO_Port, PowerMeterStartBit_Pin, GPIO_PIN_RESET);
    return adc_read_value;
}


void updateSerialCommunication(CAN_HandleTypeDef *hcan)
{
  //If UART is still busy, don't bother to send/receive anything
  if (HAL_UART_GetState(p_huart2) != HAL_UART_STATE_READY)
      return;

    if(UartReady == SET) { //If something is received, handle the command
      // Reset the serial commands.
      UartReady = RESET;
      Serial_Command_Measurement = Serial_Command_Localization = 0;

      // Echo back the command for debugging
      //HAL_UART_Transmit_IT(p_huart2, (uint8_t*) RXdata, 2);

        char command = UARTRX[0];
        int value = (int)UARTRX[1];

        if(command == 'b' || command == 'B') Slaves_Total = value;         // Set Boards
        if(command == 'd' || command == 'D') sampleDelayUs = value;        // Set sample delay
     time

        if(command == 'm' || command == 'M') Serial_Command_Measurement = 1;  // Start
    measurements
        else Serial_Command_Measurement = 0;

        if(command == 'c' || command == 'C') Serial_Command_Localization = 1; // Start
    localization
        else Serial_Command_Localization = 0;
    }
    else
    {
      //Send a RX command, the HAL library will check if it was doing so already
      HAL_UART_Receive_IT(p_huart2, (uint8_t*) UARTRX, 2);
    }

}



void application_Datareceived(void){
  // To be called from the HAL_UART_RxCpltCallback in main.ck
   /* Set transmission flag: transfer complete*/
  UartReady = SET;
}

void sendDataSerial(int DataType)
{
  //Measurement Data Type
  if (DataType == SERIAL_COMMAND_MEASUREMENT)
  {
    int i;
    int numBytes = Slaves_Total * SERIAL_LENGTH_MEASUREMENTS;
    uint8_t bytes[numBytes];

    for(i = 0; i < numBytes; i++) {
      if (CAN_STDID == CAN_ID_HOST)
      {
        /* LSB first order. */
        bytes[(i*SERIAL_LENGTH_MEASUREMENTS) + 0] = n[i].data[3];
        bytes[(i*SERIAL_LENGTH_MEASUREMENTS) + 1] = n[i].data[2];
```

```
103         bytes[(i*SERIAL_LENGTH_MEASUREMENTS) + 2] = n[i].data[1];
104         bytes[(i*SERIAL_LENGTH_MEASUREMENTS) + 3] = n[i].data[0];
105       }
106       else /* if (CAN_STDID == SLAVES) */
107       {
108         /* LSB first order. */
109         bytes[3] = TxData_Measurements[0];
110         bytes[2] = TxData_Measurements[1];
111         bytes[1] = TxData_Measurements[2];
112         bytes[0] = TxData_Measurements[3];
113       }
114     }
115
116       int loopcounter = 0;
117     while (HAL_UART_GetState(p_huart2) != HAL_UART_STATE_READY)
118     {
119       loopcounter++;
120     }
121
122     HAL_UART_Transmit(p_huart2, (uint8_t*) bytes, numBytes, HAL_MAX_DELAY);
123   }
124   //Calibration Data Type
125   else if (DataType == SERIAL_COMMAND_LOCALIZATION)
126   {
127
128     int numBytes = Slaves_Total * SERIAL_LENGTH_LOCALIZATION;
129     int i;
130     uint32_t NodeNumber, AzimuthAngle, PolarAngle, DeviceID;
131     uint8_t bytes[numBytes];
132
133     bytes[0] = Slaves_Total;
134
135     for(i = 0; i < Slaves_Total; i++ ) {
136
137       NodeNumber    = (uint32_t) n[i].NodeNumber;
138       AzimuthAngle  = n[i].AzimuthAngle;
139       PolarAngle    = n[i].PolarAngle;
140       DeviceID    = n[i].deviceID;
141
142       // Node Number (LSB first order)
143       bytes[(i*SERIAL_LENGTH_LOCALIZATION) + 1] = (NodeNumber & 0x000000FF);
144       bytes[(i*SERIAL_LENGTH_LOCALIZATION) + 2] = (NodeNumber & 0x0000FF00) >> 8;
145
146       // Azimuth angle (LSB first order)
147       bytes[(i*SERIAL_LENGTH_LOCALIZATION) + 3] = (AzimuthAngle & 0x000000FF);
148       bytes[(i*SERIAL_LENGTH_LOCALIZATION) + 4] = (AzimuthAngle & 0x0000FF00) >> 8;
149       bytes[(i*SERIAL_LENGTH_LOCALIZATION) + 5] = (AzimuthAngle & 0x00FF0000) >> 16;
150       bytes[(i*SERIAL_LENGTH_LOCALIZATION) + 6] = (AzimuthAngle & 0xFF000000) >> 24;
151
152       // Polar angle (LSB first order)
153       bytes[(i*SERIAL_LENGTH_LOCALIZATION) + 7] = (PolarAngle & 0x000000FF);
154       bytes[(i*SERIAL_LENGTH_LOCALIZATION) + 8] = (PolarAngle & 0x0000FF00) >> 8;
155       bytes[(i*SERIAL_LENGTH_LOCALIZATION) + 9] = (PolarAngle & 0x00FF0000) >> 16;
156       bytes[(i*SERIAL_LENGTH_LOCALIZATION) + 10] = (PolarAngle & 0xFF000000) >> 24;
157
158       // Device ID (LSB first order)
159       bytes[(i*SERIAL_LENGTH_LOCALIZATION) + 11] = (DeviceID & 0x000000FF);
160       bytes[(i*SERIAL_LENGTH_LOCALIZATION) + 12] = (DeviceID & 0x0000FF00) >> 8;
161     }
162
163       int loopcounter = 0;
164       // Wait for UART to be available, to be sure the data will be transmitted
165     while (HAL_UART_GetState(p_huart2) != HAL_UART_STATE_READY)
166     {
167       loopcounter++;
168     }
169     HAL_UART_Transmit(p_huart2, (uint8_t*) bytes, numBytes, HAL_MAX_DELAY);
170   }
171   // For now, use non-interrupt transmit, for some reason using the _IT version
172   // and letting other code run afterwards, the TX data gets corrupted.
173 }
```