"Ultra-Wideband for internet of things"

Laurent, Gwendal

ABSTRACT

Over the past couple of years, big names in the technology industry like Apple and Samsung started to release ultra-wideband-based products. This radio technology can perform high data rate transmission with very low power consumption and has great resistance to multipath fading. Additionally, ranging applications built on top of ultra-wideband can perform distance measurements with an accuracy of a few centimeters. In this study, a driver is implemented for the GRiSP 2 , a board for embedded systems and Internet of Things (IoT) running on the Erlang virtual machine out of the box. The driver is used to support a new sensor built by the company Peer Stritzinger GMbH based on the DWM1000 manufactured by the company Qorvo. This chip is IEEE 802.15.4-2011 compliant and uses ultra-wideband radio technology to send and receive data. On top of this driver, a simple medium access control (MAC) layer was built to send and receive MAC frames by following the IEEE 802.15.4-2011 standard. Finally, two-way ranging methods have been implemented to perform ranging operations between two GRiSP 2 cards. The results of this work show that the implementation is capable to send and receive MAC frames with a data rate of 31 kb/s and also perform accurate ranging operations.

CITE THIS VERSION

UCLouvain

École polytechnique de Louvain

Ultra-Wideband for internet of things

Author: **Gwendal LAURENT**
Supervisor: **Pr. Peter VAN ROY**
Readers: **Peer STRITZINGER, Pr. Ramin SADRE**
Academic year 2022-2023
Master [120] in Computer Science

Abstract

Over the past couple of years, big names in the technology industry like Apple and Samsung started to release ultra-wideband-based products. This radio technology can perform high data rate transmission with very low power consumption and has great resistance to multipath fading. Additionally, ranging applications built on top of ultra-wideband can perform distance measurements with an accuracy of a few centimeters.

In this study, a driver is implemented for the *GRiSP 2* , a board for embedded systems and Internet of Things (IoT) running on the Erlang virtual machine out of the box. The driver is used to support a new sensor built by the company Peer Stritzinger GMbH based on the DWM1000 manufactured by the company Qorvo. This chip is IEEE 802.15.4-2011 compliant and uses ultra-wideband radio technology to send and receive data. On top of this driver, a simple medium access control (MAC) layer was built to send and receive MAC frames by following the IEEE 802.15.4-2011 standard. Finally, two-way ranging methods have been implemented to perform ranging operations between two *GRiSP 2* cards.

The results of this work show that the implementation is capable to send and receive MAC frames with a data rate of 31 kb/s and also perform accurate ranging operations.

Acknowledgements

I would like to express my gratitude to my thesis supervisor Professor Peter Van Roy for his guidance and assistance at every step of my thesis. Moreover, I would like to thank him for giving me the opportunity to work on this subject.

I also would like to thank Peer Stritzinger for all the invaluable advice and insights he provided me over the course of this work.

Lastly, I would like to thank my family for their unwavering support.

Contents

List of Figures

Acronyms

AOA angle of arrival.

BPM Burst Position Modulation.

BPSK Binary Phase-Shift Keying.

BSP Board Support Packages.

CAP contention access period.

CFP contention-free period.

CMOS Complementary metal–oxide–semiconductor.

CSMA-CA Carrier-sense multiple access with collision avoidance.

FCS frame checking sequence.

FDT Flattened Device Tree.

FEC Forward error correction.

GTS guaranteed time slots.

IoT Internet of Things.

LR-WPAN low-rate wireless personnal area networks.

MAC medium access control.

MFR MAC footer.

MHR MAC header.

MISO Master In-Slave Out.

MOSI Master Out-Slave In.

NIF Native Implemented Function.

NiF Native-implemented Functions.

PAN Personal Area Network.

PER packet error rate.

PHR PHY header.

PHY physical layer.

RTLS real-time location system.

SECDED Single-error-correction double-error-detect.

SFD Start of frame delimiter.

SHR Synchronization header.

SPI Serial Peripheral Interface.

SPICLK clock signal.

SPICSn slave select signal.

SPIPHA clock/data phase.

SPIPOL clock polarity.

TDOA time difference of arrival.

TOF time of flight.

ToF Time-of-Flight.

UWB ultra-wideband.

Chapter 1

Introduction

In our interconnected world, IoT has become omnipresent in our daily lives and is expected to grow and expand more with the arrival of 5G. However, among all technologies used in that ecosystem, one is being overlooked, ultra-wideband (UWB). This technology has been proven to be resistant to multi-path fading [4] and is also able to perform high data rate transmission using very low power consumption [5]. Additionally, UWB is capable to realize ranging measurements 100 times more accurately than other technologies like Bluetooth or WiFi [6]. In 2021, Apple released their AirTag product which uses UWB and Bluetooth technology to track everyday objects like a set of keys or a wallet. More recently, in 2023, Samsung also released their first UWB chipset, the Exynos Connect U100 which they claim is able to perform ranging operation "down to single-digit centimeters" [7]. These two examples show the industry's recent interest in the technology.

The company Peer Stritzinger GmBH is planning to release a new UWB pmod based on the DWM1000 chip manufactured by the company Qorvo for their last version of the *GRiSP* board, the *GRiSP 2* . A prototype has already been built, but the board needs to be extended with a new driver to support it. Qorvo already published a driver written in C [8], but isn't compatible with the *GRiSP 2* because its runtime library is written in the Erlang programming language instead of C. With this driver, the company would be able to show potential clients how the new pmod will work and later on, build a new batch of pmod. Furthermore, a new version of the boards, the *GRiSP 0* , is planned to be released. This board would support only one radio interface. The first version of the *GRiSP 0* should start by supporting UWB and the driver would allow them to release a first set of prototypes.

This work will show how the driver was built layer by layer before showing a couple of applications implemented on top of it. More precisely, chapter 2 will give an overview of the different technologies and materials used in this study. Chapter 3 will explain the implementation of the driver which includes read and

write operations to the different registers of the DW1000 as well as transmission and reception operations. Then chapter 4, describes how the MAC layer has been implemented on top of the driver to achieve transmission and reception of MAC frames. Afterward, chapter 5 uses the MAC layer to perform ranging operations using two different two-way ranging methods. Finally, chapter 6 gives a summary of all the results as well as a list of possible improvements that could be done on the driver and the upper layers.

The goals of the thesis are to be able to exchange reliably a series of MAC frames between two devices even in the presence of network jitter and to perform multiple series of distance measurements using the two-way ranging methods in different situations and assess the precision of their implementations.

1.1 Related work

1.1.1 *GRiSP*

The *GRiSP* project and more precisely, the *GRiSP* base board has been the basis of previous works. In [9], the authors developed a fault-tolerant and distributed framework for asynchronous sensor fusion called Hera using the *GRiSP-Base* board and Erlang. They showed how to perform sensor fusion for position and orientation tracking and how efficient it can be. In [10], the author built a driver for the MRF24J40 microchip to enable IEEE 802.1.4 based communications between *GRiSP-Base* boards and also with the Zoleria RE-MOTE using Contiki.

1.1.2 Ultra-Wideband (UWB)

UWB isn't a new technology and past works like [11] and [12] were already claiming its potential and its possible applications in wireless personal area networks and sensor networks twenty years ago. In [13], it was shown that UWB radar systems are able to perform human detection through walls due to the technology's high range resolution and good penetration of obstacles. This kind of application can be used in emergency situations to find survivors inside buildings after earthquakes. Other applications use UWB combined with two-way ranging to localize objects. For example, this paper [14] uses the DW1000 to perform two-way ranging to build a localization system for drones performing inventory management. Moreover, UWB positioning measurements can be fused with the measurements of other sensors to get a more accurate and robust view of the environment. In [15], UWB position measurements were fused with an inertial measurement unit via an extended Kalman filter to build and assess an indoor positioning system.

1.2 Use cases

This work opens the door for a large number of applications for the pmod UWB. Indeed, the implementation of the driver is only the first stepping stone of using UWB on the *GRiSP* . The applications and use cases presented in this section could have a massive impact on our daily lives.

1.2.1 Real-time locating system (RTLS)

Multiple real-time location system (RTLS) applications have already been built using UWB. However, none were built using *GRiSP* . With the introduction of the pmod UWB in the *GRiSP* ecosystem, robust and low power RTLS applications will be possible. For example, on the industry level, such applications could perform smart inventory management or even localize equipment in a large warehouse.

Additionally, in healthcare RTLS has been used to locate patients inside a hospital in the case of emergencies or their activities inside their rooms. But also in the cases of Alzheimer's disease and dementia patients can be localized and be prevented from leaving the building by automatically locking doors for example [16] and a *GRiSP* based RTLS could extend the set of tools already existing.

Finally, in our daily lives, smart homes could use RTLS applications with *GRiSP* to locate objects in a house. Such applications could also be used on devices to make them more aware of their environment. For example, we could imagine autonomous robots like robotic vacuum cleaners using UWB to track down their position in a house and improve their efficiency.

1.2.2 Communications

Besides ranging applications, the new pmod is also able to transmit data using the IEEE 802.15.4-2011 standard. With this driver, it will be possible to implement layers like 6LoWPAN on the *GRiSP* and perform low power communications with other *GRiSP* boards but also other devices running 6LoWPAN with UWB.

1.2.3 Hera framework

The Hera framework already provides a sensor fusion for the Erlang programming language. In [9], the authors use the pmod MAXSONAR to achieve position tracking. Two-way ranging methods could be used to perform the same type of experiments with Hera and compare their results with the ones acquired with the pmod MAXSONAR.

Chapter 2

Material and resources

2.1 *GRiSP*

GRiSP [17] is a project developed by the company Peer Stritzinger GmbH. It combines both customizable hardware and software to provide embedded systems solutions.

The hardware, the board, has two versions. The first one is called the *GRiSP-Base* and the second one, the *GRiSP 2* , is its evolution. In the context of this master thesis, only the latest version will concern us. The *GRiSP* boards have multiple sockets to connect different modules called Pmods that can be connected through multiple interfaces like GPIO and SPI.

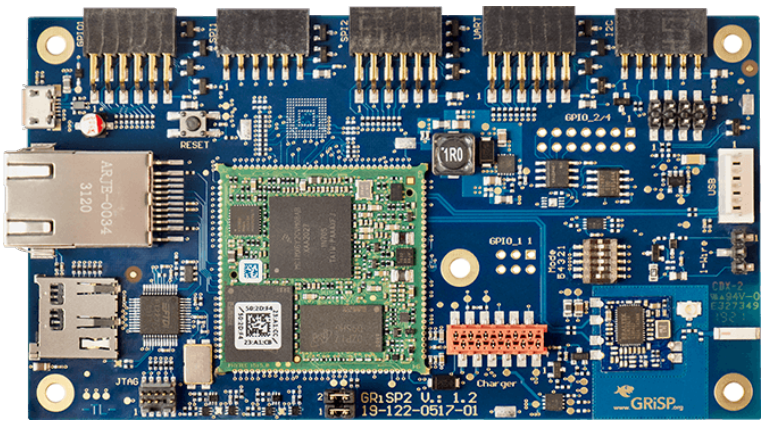

Figure 1: *GRiSP 2* (credits: grisp.org)

The software runs on a custom open-source operating system that combines both Erlang and RTEMS (Real-time executive for multiprocessor systems) a real-time operating system that supports 18 processor architectures and open standard

application programming interfaces [18]. This combination enables the board to run Erlang code out of the box and lets users create IOT applications directly in that programming language. The different basic protocols supported by the *GRiSP* uses port-drivers or a Native-implemented Functions (NiF), thus the code of the drivers for the different PMod accessories can also be written in Erlang and requires to seldom go back to C code level. [17]

2.2 Serial Peripheral Interface (SPI)

The Serial Peripheral Interface (SPI) is used in this thesis to communicate between the *GRiSP 2* and the Pmod UWB. It was originally designed by the company Motorola, but it became so popular that we could argue that it became a *de facto* public protocol [19]. A SPI system is composed of one master, the microcontroller providing the clock signal, and one or multiple slaves, the integrated circuits that receive the clock signal from the master [20]. It's a communication protocol that works on 4 signal lines: a clock signal (SPICLK), a slave select signal (SPICSn), a data line from the master to the slave named Master Out-Slave In (MOSI), and a data line from the slave to the master named Master In-Slave Out (MISO). The specifications of the SPI bus can vary from microcontrollers and to get the description that corresponds to a specific application one should refer to the user manual or the datasheet of the specific chip in use [19]. In this framework, the naming conventions will be the ones used in the DW1000 datasheet [3].

When the master wants to send or read requested data from a slave, it has to pull the SPICSn line, activate the clock signal on the SPICLK line, send data over the MOSI line and read the data coming from the MISO line.

2.3 Pmod & DWM1000

The Pmods are modules that can be connected to the *GRiSP* boards with the different interfaces available. These modules can be sensors (e.g. accelerometer, temperature, ...) or actuators (e.g. RC-servos, ...). The *GRiSP* software provides Erlang drivers for most of the Pmods.

This thesis will focus on the Pmod UWB built by the company Stritzinger itself. It communicates with the *GRiSP* board through the 12 pins SPI interface of the board (SPI type 2A). The Pmod uses the DWM1000 module built by the company Decawave (now owned by the company Qorvo). The module on boards the DW1000 single chip Complementary metal–oxide–semiconductor (CMOS) UWB transceiver as well as other RF components [21]. The DW1000 like the DWM1000 is also manufactured by the company Decawave. In this thesis, even though we are

working directly with the DWM1000, the actual operations are mostly performed on the DW1000.

Figure 2: DWM1000 (source: mouser.be)

Both the DWM1000 and the DW1000 are compliant with the IEEE 802.15.4-2011 UWB standard [1], which is a standard that defines the physical layer (PHY) and the MAC sublayer, for low-rate wireless personnal area networks (LR-WPAN) that are low-cost communication networks used to send data over a relatively short distance. Therefore, the physical layer of the DW1000 uses impulse radio and a modulation scheme that combines Burst Position Modulation (BPM) and Binary Phase-Shift Keying (BPSK). Additionally, the device support 6 channels, detailed in table 2.1.

Channel number	**Center frequency (MHz)**	**Bandwidth (MHz)**
1	3494.4	499.2
2	3993.6	499.2
3	4492.8	499.2
4	3993.6	1331.2
5	6489.6	499.2
7	6489.6	1081.6

Table 2.1: DW1000 channel table (Source: DW1000 user manual [2])

According to the DW1000 data sheet [3], the chip can be used to determine the location of another chip to a precision of 10 centimeters. Moreover, the chip

also supports concurrent data transfer and precision location. Finally, It has an extended communication range up to 290 meters at 110 kbps and with 10% packet error rate (PER)

The *GRiSP* already supports numerous pmods using SPI for communication and each one of them has its own driver code. Reading and understanding how the driver of the pmod nav and pmod dio work was one of the first steps of this work to understand the interactions between the board and the pmod over SPI.

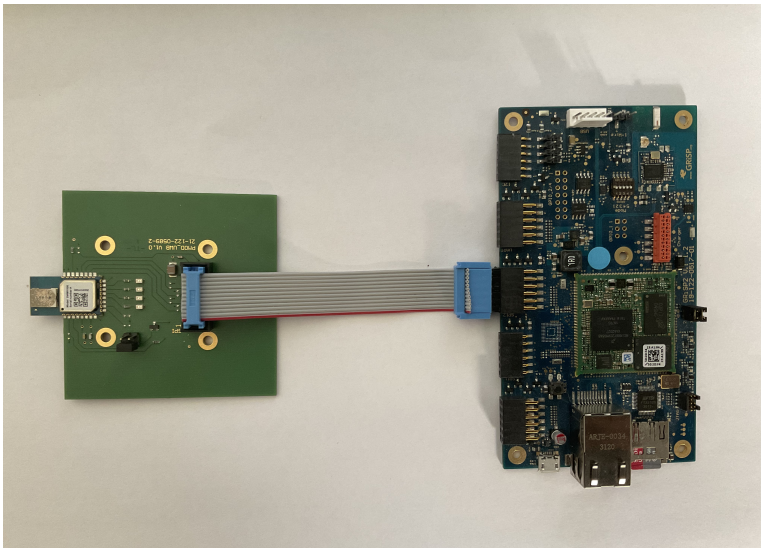

Figure 3: The pmod uwb (left) connected to the *GRiSP 2* (right) board

2.3.1 Pmod drivers architecture of the *GRiSP*

The architecture of the drivers already implemented for the *GRiSP* pmods all take advantage of the `gen_server` behavior provided by Erlang which implements a client-server model [22]. In this model, the server manages a resource that multiple clients want to share. In our case, the driver will manage a resource, the pmod, that multiple processes can share. A client can make two kinds of requests to the server. They can make synchronous requests called *Call* or they can make asynchronous requests called *Cast*.

The `gen_server` behavior also keeps in memory the server state. In the code of the pmod nav, for example, that state is used to store information like the bus used to communicate, its registers, and a cache. all related to each component of the pmod [23].

Finally, a `gen_server` can be part of a supervision tree. In the case of the *GRiSP* , the driver processes are children of the supervisor `grisp_devices_sup`.

A supervisor is another Erlang behavior that supervises worker processes. More precisely, it keeps track of its different child processes and lets users define a restart strategy if one of them crashes [24]. Here, `grisp_devices_sup` uses the *one_for_one* strategy, which means that if a driver crashes, then only that one will be restarted. In other words, if the driver throws an uncaught error and stops. It is then restarted by the supervisor and there is no need to restart the *GRiSP* board manually.

2.3.2 DW1000 register set

Using the SPI interface, the master device is able to access the register set of the DW1000. It is organized in multiple register files with their own size and identified with their register file IDs. The register files can be read-only, write-only, both read-write, or have a different read-write configuration for the registers that compose them (called special read/write). Some register files are also part of the double receive buffer which allows the reception of a frame while the master device is reading the previously received frame.

There are different types of register files. They can contain multiple fields and bit flags, sub-registers, or some, like the transmission buffer, contain only one field. Sub-registers are identified and can be accessed with a sub-address. They can be used in an objective to optimize the read/write operations and access only the sub-register instead of the full register file. In some cases, these sub-addresses must be used to avoid writing reserved areas within the register files.

2.4 Ultra-Wideband (UWB)

According to the FCC, UWB is any signal with more than 500MHz bandwidth with a band within 3.1 and 10.6GHz that respects a specific spectrum mask. [25]. There are two types of UWB communication systems: pulse-based or multicarrier-based. Pulse-based systems generate a short burst of pulse at a specific time, while multicarrier-based uses multiple carriers at the same time to transmit the data [5].

This radio technology has multiple advantages. First, UWB is able to transmit high data rates by using very low power. Indeed, according to Shannon's formula (equation 2.1), you can increase the channel capacity C) (i.e how many bits per second can be transmitted without error over the channel) by increasing exponentially the transmitted power or by increasing linearly the bandwidth (BW) [5].

$$C = BW \log_2(1 + S/N) \tag{2.1}$$

This is very convenient for IOT devices with a small battery because they can send data with high throughput without using too much power.

Second, UWB enables location tracking with an accuracy of up to a few centimeters. In fact, one of the major features of UWB is the usage of the Time-of-Flight (ToF) to calculate the distance between devices. This method is made possible by the modulation method used to transmit the data. Since it uses narrow pulses that have clean edges, it allows determining precisely the arrival time and the distance even in the presence of multi-paths. This makes UWB 100 more precise than other technologies like Wi-Fi or Bluetooth. Moreover, due to its low latency, UWB technologies can be used for real-time location and is 50 times faster than GPS, which makes the tracking of fast-moving objects like drones possible. This technology has many applications in today's connected world. This could go from locating the key of your car in your house to locating people in a building in the case of an emergency. [6]

2.5 IEEE 802.15.4-2011

The IEEE 802.15.4-2011 standard [1] defines the PHY and the MAC sub-layer for communications inside LR-WPAN which are simple and low-cost networks used to send information over a short range. The main target of this standard is devices operating in a range of 10 meters with low-data-rate wireless connectivity with low power consumption requirements. It has the capacity of using 64-bits extended address or allocated 16-bits short address, low power consumption, etc. The upper layers like the network layer or the application layer aren't described in this standard, but standards like 6LoWPAN have been developed to operate on top of IEEE 802.15.4-2011 .

This section will depict the standard as it is described in the standard definition [1] and as it is used by the DW1000 and explained in the annexes of its user manual [2].

A network running the IEEE 802.15.4-2011 standard can use two types of topologies: the star topology where devices are only allowed to communicate with one central controller or the peer-to-peer topology where all devices are allowed to communicate with any other as long as they are in range.

2.5.1 Ultra-Wideband PHY

The standard defines multiple PHY layers, the one that interests us in the context of this work is the UWB PHY because it is the one used by ultra-wideband devices and more particularly by the DW1000.

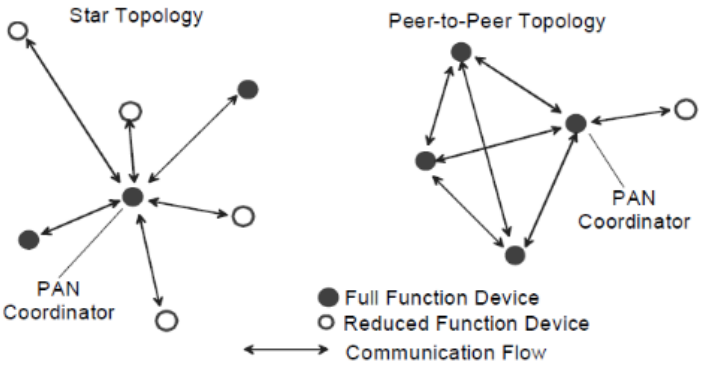

Figure 4: IEEE 802.15.4-2011 types of topologies (credits: IEEE[1])

The radio signals are based upon impulse radio signaling. The modulation scheme is BPM-BPSK, a combination of BPM and BPSK, and each symbol is composed of an active burst of UWB pulses. An UWB PHY frame is composed of 3 elements: a Synchronization header (SHR) preamble, a PHR, and a data field. The SHR itself is composed of 2 elements: a sync sequence (called preamble in the DW1000 user manual) and a Start of frame delimiter (SFD).

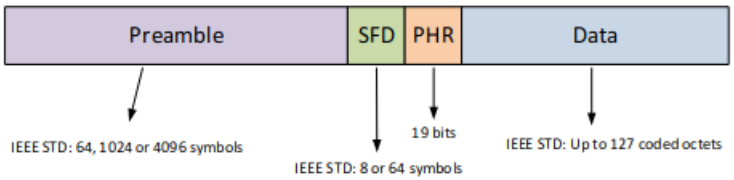

Figure 5: UWB PHY frame structure (credits: DW1000 user manual [2])

The SHR is made of a sequence of single pulses (either positive, negative, or none) determined by a preamble code composed of ternary symbols (1, -1, 0). The standard defines two lengths of preamble code and there are between 8 and 9 different codes per length and each code can only be used on specific channels. The codes are chosen such that the resulting symbol sequence has perfect periodic auto-correlation properties. Auto-correlation is a measure used in radar technologies to measure how similar a signal is to itself [26]. But, a further explanation of this property is outside the scope of this work. Nevertheless, according to the DW1000's user manual [2], this special structure of the preamble code allows the receiver to use multi-paths as an advantage to increase the operating range and also determine the arrival time of the first path.

The SFD marks the end of the preamble. The standard describes a "short SFD" for low and medium data rates and a "long SFD" used for faster data rates. Furthermore, the reception of the SFD marks the switch into BPM-BPSK

modulation. When it comes to the DW1000, this event is used for the time-stamping of the reception of the frames. Indeed, the timestamp can be determined with high accuracy due to its deterministic characteristics combined with the determination of the first arriving ray [2].

After the SHR comes the PHR, it contains, among other fields, the length of the frame payload and it uses 6 parity bits (Single-error-correction double-error-detect (SECDED)) to detect any channel errors during its transmission

Bit 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
R1	R0	L6	L5	L4	L3	L2	L1	L0	RNG	EXT	P1	P0	C5	C4	C3	C2	C1	C0
Data Rate		Frame Length							Ranging Packet	Header Extension	Preamble Duration		SECDED Check Bits					

Figure 6: PHR bit description (credits [1])

Finally, the last part of the UWB PHY is the actual data payload. It has a maximum size of 127 bytes. Like the PHR it is encoded using BPM-BPSK modulation and uses Reed Solomon code as Forward error correction (FEC). This section of the frame can be transmitted at data rates of 110 kbps, 850 kbps, 6.8Mbps, or 27Mbps (however, the DW1000 doesn't support a 27Mbps data rate).

2.5.2 MAC sub-layer

The MAC sub-layer is situated above the UWB-PHY layer. It is responsible, among other tasks, to manage the access to the radio channel. It also provides multiple features like beacon management, acknowledgment, etc. One possible option for the Personal Area Network (PAN) coordinator to control channel access is to use a superframe structure to bind the channel times. A superframe is delimited by beacon frames sent by the coordinator. The beacon frame is used to identify a PAN, synchronize the devices inside a PAN, and defines the structure of the superframe. A superframe can be divided into two parts. First, the contention access period (CAP), where devices compete with each other to communicate and use slotted Carrier-sense multiple access with collision avoidance (CSMA-CA). Second, the contention-free period (CFP), always situated at the end of the superframe, enables the coordinator to allocate guaranteed time slots (GTS) for applications with special needs like low-latency applications. Figure 7 shows an example of the structure of a superframe.

The MAC frames are put inside the data payload field of the UWB-PHY layer. A MAC frame is composed of three elements: a MAC header (MHR), a MAC

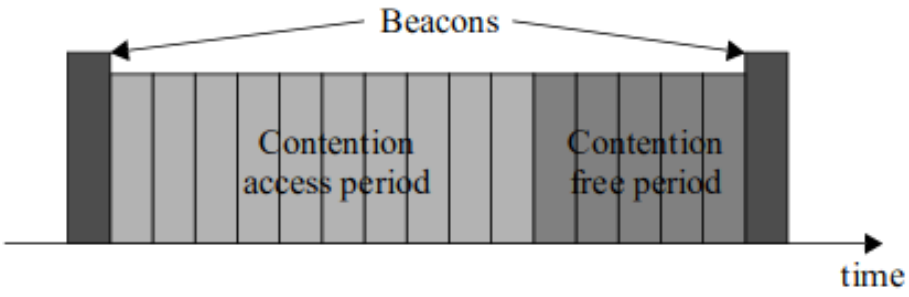

Figure 7: MAC Superframe (credits [1])

payload, and a MAC footer (MFR).

MAC Header (MHR)							MAC Payload	MAC Footer (MFR)
Frame Control	Sequence Number	Destination PAN Identifier	Destination Address	Source PAN Identifier	Source Address	Aux Security Header	Frame Payload	FCS
2 octets	1 octet	0 or 2 octets	0, 2 or 8 octets	0 or 2 octets	0, 2 or 8 octets	0, 5, 6 10 or 14 octets	Variable number of octets	2 octets

Figure 8: MAC frame and its fields (credits [2])

Bits 0 to 2	Bit 3	Bit 4	Bit 5	Bit 6	Bits 7 to 9	Bits 10 & 11	Bits 12 & 13	Bits 14 & 15
Frame Type	Security Enabled	Frame Pending	ACK Request	PAN ID Compress	Reserved	Dest. Address Mode	Frame Version	Source Address Mode

Figure 9: The frame control of the MAC header and it (credits [2])

The MHR begins with a two bytes frame control field. It is there to identify the type of the frame and the structure of the MAC header. As figure 9 shows, the first three bits of the frame control indicate the type of frame. A frame can be of type *beacon* (2#000), *data* (2#001), *acknowledgement* (2#010), or *MAC command* (2#011) while the other value (2#1xx) are reserved. Bit #3 indicates if there are auxiliary security headers in the frame. Bit #4 indicates if there is more data to receive. Bit #5 specifies if the transmitter of the frame is expecting an acknowledgment from the receiver. Bit #6 is the PAN compression field. When it is set to one and both the destination and source addresses are present, only the destination PAN ID is present in the MAC header and the source PAN ID is

assumed to be equal. Bits #7-9 are reserved. Bits #10-11 and Bits #14-15 are address compression fields. If their values are set to 2#00, then their corresponding PAN ID and address are not present in the MAC header. If their values are set to 2#10, then their corresponding address is a short address (16 bits). Finally, 2#01 is reserved and shouldn't be used, if their values are set to 2#11 then their corresponding address is an extended address (64 bits).

Finally, the MAC frame is ended by the MFR which is two bytes long and is in fact a frame checking sequence (FCS) CRC used to determine if the frame is corrupted or not.

Chapter 3

Implementation of the driver

This chapter explains how the driver was built starting from the setup of the SPI clock and performing data exchange on the SPI line. Before performing read and write operations, and finally, being able to send and receive data using UWB. The file containing the API of the driver is called `pmod_uwb.erl` and is present in the appendix A.

3.1 Interaction with the pmod

Before interacting with the board, we have to determine the communication mode (i.e. SPI mode) defined by the pair of parameters: clock polarity (SPIPOL) and clock/data phase (SPIPHA). The SPIPOL determines on which level the clock idles while the SPIPHA will determine which operation is performed on each edge of the clock [19]. Table 3.1 describes the different SPI modes possible.

SPIPOL	**SPIPHA**	**SPI mode**	**Description (from the master point of view)**
0	0	0	Data is sampled on the rising (first) edge of the clock and launched on the falling (second) edge
0	1	1	Data is sampled on the falling (second) edge of the clock and launched on the rising (first) edge
1	0	2	Data is sampled on the falling (first) edge of the clock and launched on the rising (second) edge
1	1	3	Data is sampled on the rising (second) edge of the clock and launched on the falling (first) edge

Table 3.1: Different SPI modes (source: DW1000 data sheet[3])

Figure 10 shows the two possible interactions with the DW1000 when the SPIPHA is set to 1. The blue line represents when the data is launched and the red line represents when it's sampled on the MISO and MOSI lines.

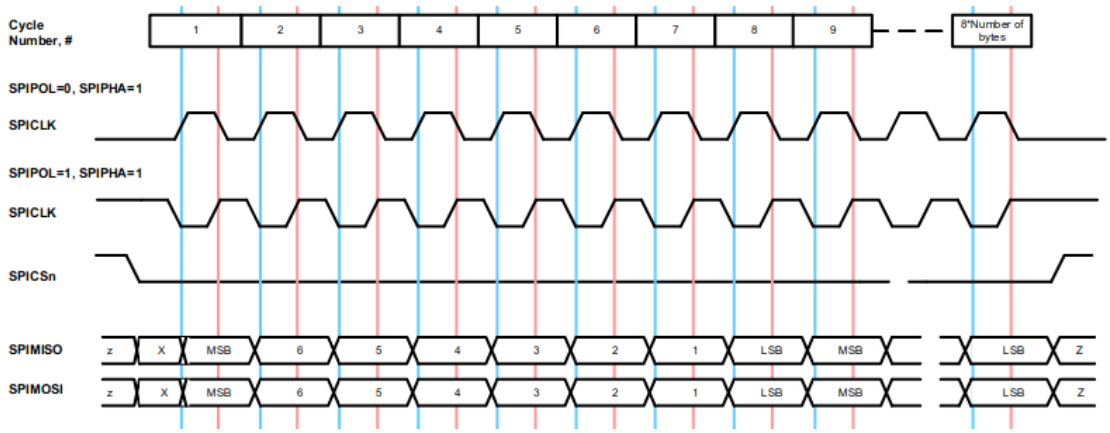

Figure 10: DW1000 - SPIPHA = 1 (source: DW1000 data sheet [3])

If these settings are not set correctly, the data sent over the different lines won't be correctly interpreted at the other endpoint. For example, if the polarity isn't set correctly, we could observe a bit shift between the expected data and the actual value read by the *GRiSP* .

On the DW1000, these parameters can be set through the pin GPIO5 for SPIPOL and GPIO6 for SPIPHA. Thus, we have to check the schematics of the pmod UWB to see their inputs. In this case, both GPIO5 and GPIO6 are plugged into a 3V. Therefore, SPIPHA and SPIPOL both have a value of 1. Both of these pins are only sampled on the rising edge of the RSTn, which means that this pin should be pulled at the startup of the driver to configure correctly the clock settings. However, the DW1000 datasheet [3] states that this pin should be configured as high impedance, but the current *GRiSP* Erlang runtime library doesn't support this kind of setting yet. Consequently, since we are not able to pull the RSTn pin, SPIPOL and SPIPHA are never sampled and their values are equal to "0". Therefore the SPI mode is the number 0 where data is sampled on the rising edge of the clock and launched on the falling edge.

In the driver code, the SPI mode is represented by a record given at each transaction call.

```
1 -define(SPI_MODE, #{clock => {low, leading}}
```

Listing 3.1: SPI_MODE macro used to define the clock settings

When those values are set, we are ready to perform our first transactions on the SPI.

3.1.1 Transaction format

A transaction can be divided into 2 parts.

The first one, the transaction header, contains information about the type of transaction (either read or write), the register file targeted by the operation, and an eventual offset/sub-addressing.

The second one, the transaction body, contains either the data read from the DW1000 or the data that has to be written on the DW1000. In the case of a write operation, both parts are sent by the master. Otherwise, in the case of a read, the transaction header is sent by the master, and the transaction body is sent by the slave. Figure 11 gives a visualization of the different transactions.

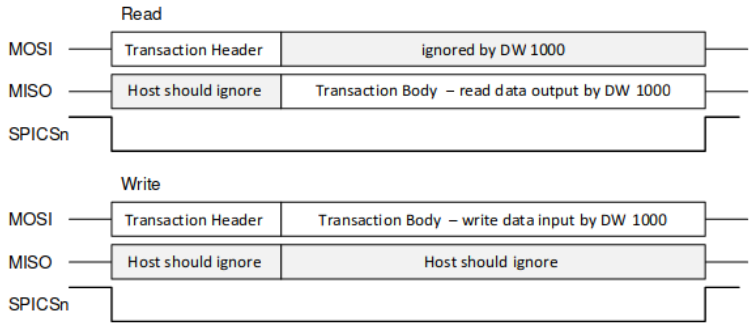

Figure 11: Different SPI transactions (source: DW1000 user manual [2])

There are three types of headers that can be used to communicate with the DW1000. The first header (figure 12) is only one byte long. Bits 0-5 contain the register file ID, a hexadecimal value that identifies each register file of the DW1000. Bits #6 indicates if the header contains a sub-address. In the case of this header, its value is set to 0. Bit #7 indicates the type of operation.

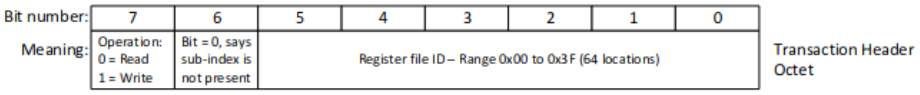

Figure 12: One byte header (source: DW1000 user manual [2])

The second header (figure 13) is two bytes long and gives a short sub-indexing that indicates a sub-address in the register file. The first byte of this header is similar to the previous one except that bit #6 has now a value of "1" to indicate that a sub-index is present. Bits 0-6 of the second byte specify the short sub-indexing (ranges from 0x00 to 0x7F) and bit #7 which indicates that we are using an extended address is set to "0" for the 2 bytes header.

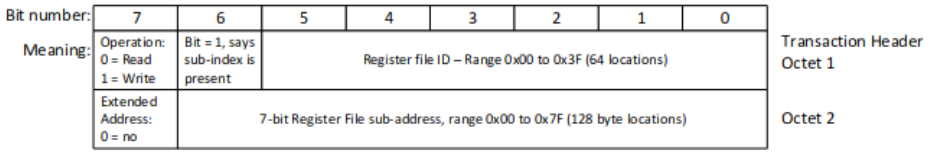

Figure 13: Two bytes header (source: DW1000 user manual [2])

The last header (figure 14) is three bytes long and gives a long sub-indexing which gives the possibility to use sub-addresses up to a value of 0x7FFF. To use this header, bit #7 of the second byte should be set to "1".

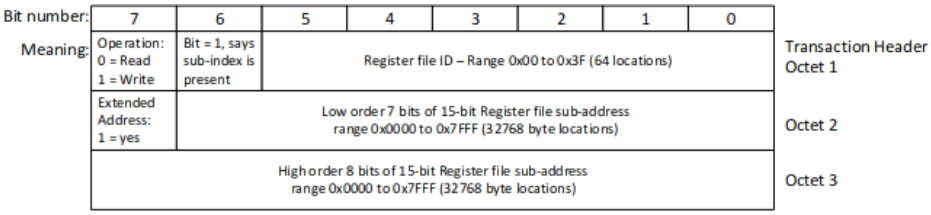

Figure 6: Three octet header of the long indexed SPI transaction

Figure 14: Three bytes header (source: DW1000 user manual [2])

3.1.2 Example

Let's take the example of a read operation performed on the register *DEV_ID* . This register is the device identifier and is hard-coded into the silicon of the chip. That makes it the perfect register to test how the driver interacts with the chip through the SPI interface because the expected result of the interaction is predictable. Furthermore, the user manual gives exactly the description of the transaction (Figure 15)

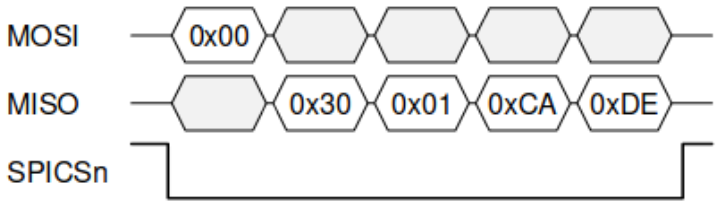

Figure 15: Description of a read operation performed on the DEV_ID register (source: DW1000 user manual [2]

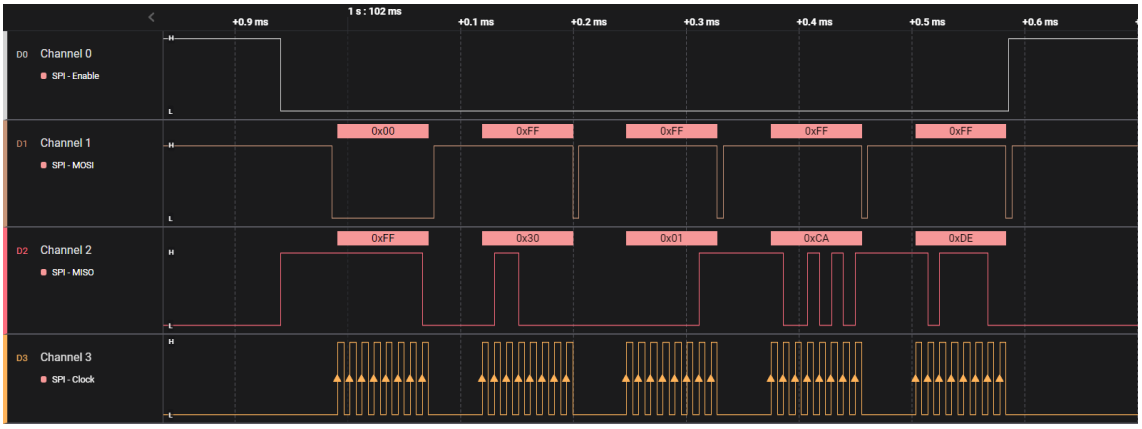

Figure 16: Read transaction on DEV_ID showed on the logic analyser

Figure 16 shows the output of a logic analyzer during a read operation on *DEV_ID* . The logic analyzer is able to sample the different SPI lines and display the evolution of their values over time.

Channel 0, the first row is SPICSn. We can see that when the transaction starts, the line is pulled down and when the transaction is done, the line is pulled back up.

Channel 1 is the MOSI line. The first byte has a value of 0x00. This is the transaction header. Its value tells us that the operation is a read operation (bit #7 has value 0), without sub-index (bit #6 has value 0) on the register file *DEV_ID* which has the ID value of 0 (bits #5-0 have a value of 0). The 4 bytes following the header all have a value of 0xFF and should be ignored because a read operation is performed.

Channel 2 is the MISO line. The first byte has a value of 0xFF and should be ignored. Then the last three bytes sent by the pmod have a value of 0x3001CADE which is the value we are expecting to read from the register file.

Channel 3 is the SPICLK. We can clearly see for each byte of the transaction, eight pulses corresponding to the eight bits being sent or received over the MOSI or the MISO lines.

3.2 Mapping the registers

The second step of building the driver was to map the register to allow a user to read and write values from them. This part needs to be written with a lot of rigor because it this the stepping stone of all the operations that can be made on the pmod.

In the driver previously implemented by the *GRiSP* team, the pmod_nav and the pmod_dio are also using the SPI interface which was a great starting point to

understand how the driver had to be built. Even though the approaches of the two drivers are different, they both use Erlang maps as output to store the values read from the pmod or as input to store the values that have to be written. Since both driver approaches were equivalent and the pmod_dio was easier to understand, it was decided to use the same approach for the mapping of the DW1000. For example, the result of a read operation on the register file *DEV_ID* is: `#{ridtag => "DECA", model => 1, ver => 3, rev => 0 }` and if we want to write the value "1" in the sub-register *TXSTRT* of the register file *SYS_CTRL* , then the following map needs to be given in the arguments of the write API call: `#{ txtstrt => 2#1 }` (Note that the value is written in the binary form but it can also be written in the decimal form).

Additionally, as we are about to see in the following sections, the DW100 contains different types of registers. In the process of writing the driver and its API, extra care has been taken to write a consistent read/write API for every register file to make the implementation more user-friendly.

3.2.1 Errors in the user manual

During the mapping of the register, a few errors were noticed in the user manual of the DW1000 [2], here is the list of all of them and how they were solved:

- In the register map overview, the length of the register file *DRX_CONF* is 44 bytes. However, the sum of the size of the sub-registers present in the overview of the register file is 45 bytes. In the API provided by Qorvo [8], the size of the register file is also 44 bytes but one sub-register with a length of one byte, *RXPACC_NOSAT* isn't present. Since the API provided by Qorvo works and got tested, the information contained inside should overrule the ones in the user manual. Thus, its version got used in the driver of the pmod.

- In the user manual as well as in the API, the size of the register file *RF_CONF* is set to 58. However, if we take the offset of the last sub-register *LDOTUNE* and add its size, 5 bytes, we reach an offset of 0x35 or 53 in decimal. To match the information provided by the chip manufacturer and to compensate for the 5 remaining bytes, a placeholder was introduced at the end of the register file in its mapping.

- The subregister *RF_CONF* has a size of 3 bytes in the user manual, but in the API, it has a size of 4 bytes. In the driver of the pmod, its size has been set to 4 bytes too.

- In the user manual and the API, the size of the register file *TX_CAL* is 52 bytes. Yet, when we sum the size of all its sub-registers, we obtain a size of

12 bytes. Since the difference was too big, no placeholder got introduced and the size of the register is set to 12 bytes in the pmod driver

- The size of *OTP_IF* in the register overview as well as in the DW1000 API is set to 18. However, if we sum the size of its subregisters given in the user manual and also in the API, we reach 19 bytes. Thus, in the driver, the size has been set to 19 bytes too.
- The register file with the ID 0x2E has two names in the user manual. In the register map overview, it's called *LDE_CTRL* but in its description, it's called *LDE_IF* . Additionally, in the API of the driver provided by Qorvo, only *LDE_IF* is used. However, to avoid any confusion for a user, both names are accepted by the driver of the pmod.
- The size of *DIG_DIAG* given in the user manual and the API is 41 bytes. Nevertheless, the sum of the size of all its subregisters is equal to 38. Thus, this value was used in the driver
- The size of *PMSC* given in the user manual and the API is 48 bytes. However, if we sum the size of all its subregisters we reach a value of 41 bytes. Additionally, if we try to compute its size by checking the last offset of its subregister we reach 0x2B which is 43 in decimal. In the driver of the pmod, the used size is 43 bytes.

3.2.2 Read the registers

The API function to read a register is `read/1`. Its only parameter is an atom corresponding to the mnemonic of the register file we are trying to access. It will return a map of the different elements stored inside the register file and their values. Figure 17 shows an example of an API call for the register file *DEV_ID* and the returned value

```
1> pmod_uwb:read(dev_id).
pmod_uwb:read(dev_id).
#{model => 1,rev => 0,ridtag => "DECA",ver => 3}
```

Figure 17: Read API call on register *DEV_ID*

This function will send a request to the `gen_server` of the driver which will internally call the function `read_reg/2`. Its first parameter `Bus` is a reference to the opened SPI bus stored in the internal state of the `gen_server`. The second parameter is again the mnemonic of the register file we are trying to access. This internal function will call the function `header/2` which builds the header of the

transaction and returns it in binary format. It then calls `grisp_spi:transfer/2` that returns the value contained in the register file in an Erlang bitstring format. Finally, the content of the bitstring is decoded using the function `reg/3` which performs a pattern matching on the different fields contained in the bitstring and then stores them in a map.

It is important to note that the value returned by the pmod UWB are sent in little-endian and the default configuration of Erlang interprets bytes in big-endian. Therefore, for most of the registers, the byte order needs to be reversed before decoding them.

During the implementation, the choice was made to read register files fully and not letting the possibility for users to read individual sub-registers. The goal here was to have a reliable operation without performing any optimization using the register offsets as little as possible.

There are two special register files that have to be processed differently than others. First, *TX_BUFFER* is a write-only register, and reading its value during transmission could corrupt its values. Therefore, an error is thrown if `read_reg/2` is called with `tx_buffer` is passed in the second parameter. Second, the *LDE_IF* or *LDE_CTRL* register has a size of 10Mb, and only 8 sub-registers are documented in the user manual with gaps that can reach 1026 bytes. Therefore, the choice was made to read each sub-register individually using the offsets of the different sub-registers and merge the results.

3.2.3 Write the registers

The API function to write a value in a register file is `write/2`. The first parameter is like in the write operation the mnemonic of the register file we are trying to access. The second parameter is a map similar to the one returned in the read operation but only containing the value the user wants to change.

```
2> pmod_uwb:write(panadr, #{pan_id => 16#DECA}).
```

Figure 18: Write API call on register *PANADR*

```
3> pmod_uwb:write(pmsc, #{pmsc_ctrl0 => #{sysclks => 2#01}}).
```

Figure 19: Write API call on register *PMSC*

Figure 18 and 19 show 2 examples of a writing operation on the pmod. The first one shows the operation on a "simple" register containing only two sub-fields: *PAN_ID* and *SHORT_ADDR* . Since *PAN_ID* is the only sub-field present in the map, the write operation will only modify that sub-field.

The second one performs the operation on a register file with a more complex structure. Indeed it is composed of multiple sub-registers containing multiple sub-fields. This is reflected in the map given in the parameter of the function by having a bigger depth than the one in Figure 18. Here, the write operation will be operated in the sub-register *PMSC_CTRL0* of the register file *PMSC* by changing the value of the sub-field *SYSCLKS* by the binary value "01". Additionally, it is also possible to write inside multiple sub-fields of one register and also in multiple sub-registers in the same API call shown in figure 20

```
1> pmod_uwb:write(pmsc, #{pmsc_ctrl0 => #{sysclk => 2#01, txclks => 2#01}, pmsc_ctrl1 => #{lderune => 2#0}}).
```

Figure 20: Write API call on register *PMSC* on multiple sub-registers and sub-fields

When a user wants to write a value in a register file, we must be careful to not overwrite values that are already present in the register file. Due to the nature of the SPI transaction, the minimum amount that someone can read or write from the device is one byte. This implies that if someone wants to change the value of a bit flag we need to write, at the minimum, 7 other bits that the user doesn't want to change. Thus we have to read their value first and write the same values alongside the changed bit flag. In the driver, in most of the cases, the choice was made that when the user wants to change a value inside a register file, the whole register file will be read and written again with the changed value. In some cases, this is not possible because of the configuration of the register file. Indeed, some register files either have read-only sub-register (for example *AGC_CTRL*) or reserved bytes that can't be overwritten (for example *EXT_SYNC*). In this case, each of their sub-registers is written individually. Additionally, since all of the sub-registers of these special register files are longer than one byte, instead of writing again all their values at each write, only the sub-registers targeted by a write will be written.

However, due to the choice that was made in the reading operation of only allowing reading register files in their entirety, we need to read the whole register file containing the sub-register and extract its values. Furthermore, since there is no guarantee that the values of a sub-register don't change between writes, the register file needs to be read each time we want to write in one of its sub-register. In other words, if a user wants to write simultaneously (i.e. on the same API call) inside n sub-registers, then the corresponding register file will be read n times.

Additionally, some register files like *RX_BUFFER* are read-only register files. If the user tries to write in those register files, an error will be thrown to protect their values.

Finally, the transaction body should be sent in little-endian. Thus, when the data is sent on the SPI interface, we must be sure that the endianness of the bytes has been changed.

3.3 Initialization of the pmod

After the startup of the *GRiSP 2* board, the driver needs to be loaded and the DW1000 needs to be initialized. This section describes the different steps and how the initialization has been implemented.

3.3.1 Checking the connected device

The first thing to do during the loading is to check if the user selected the correct slot of the *GRiSP 2* and connected the right pmod to it. After adding the driver process to the supervisor and opening the SPI bus, the driver checks the content of the register file *DEV_ID* with the ID 0x00. This read-only register file contains the register identification tag and model. With these 2 registers, we are able to check if the device is connected to the right device, the DW1000. Indeed the value of RIDTAG is constant over all Decawave parts and should always be 0xDECA and the value of MODEL should be 0x01 for the DW1000. If one of the values is different from the one expected, the driver throws an error and the initialization stops there.

3.3.2 Loading the leading edge algorithm

After checking if the connected device is the right one. The leading edge detection algorithm needs to be loaded from the ROM of the DW1000. This algorithm is responsible to find the first path of a transmitted message and to compute the timestamp of the reception. If the algorithm isn't loaded, it is still possible to perform message transmission if we deactivate the leading edge detection before the first reception. In that case, the RX timestamp won't be correct. Thus some algorithms like two-way ranging can't be performed because it relies on the timestamps to perform their measurements. Loading the algorithm from the ROM is done by following a precise procedure described in the user manual of the chip [2] and also in the examples for the C driver made by the company. Extra care must be taken when performing this operation because if the algorithm is activated but not loaded correctly at the reception of a packet, the chip can have unexpected behavior and get stuck in an undocumented state where data reception isn't possible anymore.

To load the algorithm, first, the value 0x301 needs to be written in the *PMSC_CTRL0* sub-register of the *PMSC* register file. This writes the value 0x01 *SYSCLKS* and sets the value an undocumented bit to 1. Then, the value 0x8000 needs to be written in the sub-register *OTP_CTRL* which sets the value of the bit flag *LDELOAD* to 1. This will copy the microcode from the ROM to the RAM. Finally, the value 0x200 needs to be written in the sub-register *PMSC_CTRL0*

which puts sets back *SYSCLKS* to automatic mode and sets back the value of the undocumented bit to 0.

3.3.3 Writing optimal values

When the DW1000 turns on, some of the default values aren't set up on optimal values for performances. It is the job of the driver designer to overwrite these values before using the chip to send frames. The user manual describes how to perform that initialization if we use the default configuration of the chip. Since it was decided to stick with these default values, the driver follows those instructions, but if one decides to change the default transmission channel for example, other values should be written for optimal operations. Table 3.2 shows the different sub-registers to overwrite and the values that have to be written inside if we keep the default configuration of the DW1000.

Register file	**Sub-register**	**Default value**	**New value**
AGC_CTRL	AGC_TUNE1	0x889B	0x8870
AGC_CTRL	AGC_TUNE2	/	0x2502A907
DRX_CONF	DRX_TUNE2	0x311E0035	0x311A002D
LDE_CFG	LDE_CFG1 (NTM sub-field)	0xC	0xD
LDE_CFG	LDE_CFG2	0x0000	0x1607
TX_POWER	N/A	0x1E080222	0x0E082848
RF_CONF	RF_TXCTRL	DE1E3DE0	0x001E7DE0
TX_CAL	TC_PGDELAY	0xC5	0xB5
FS_CTRL	FS_PLLTUNE	0x46	0xBE

Table 3.2: All the default values to overwrite

3.3.4 Writing custom configuration

After writing the optimal values, we still have to write some custom configurations that are either related to the pmod in itself or enable elements that are not turned on by default. The following list gives an explanation of each setting and its different effects.

1. Setting the sub-field *PLLLDT* to "1" to ensure that the PLL locks flags work correctly. The goal of this operation is to have better debugging and diagnostic capacities when a frame isn't correctly sent or received.

2. Setting the sub-fields *MSGP2* and *MSGP3* of the sub-register *GPIO_MODE* to "01". It indicates that the GPIO pins 2 and 3 are respectively operating as RXLED output and TXLED output.

3. Setting the sub-fields *MSGP0* and *MSGP1* of the sub-register *GPIO_MODE* to "01". It indicates that the GPIO pins 0 and 1 are respectively operating as RXOKLED output and SFDLED output.

4. Setting the sub-fields *GPDCE* and *KHZCLKEN* to a value of "1". *GPDCE* serves to enable the clock in charge of the feature that makes the LEDs blink. *KHZCLKEN* enables the kilohertz clock used by the same feature.

5. Setting the sub-field *BLNKEN* to a value of "1". Alongside the setup made in the three previous points, this bit enables the LED blinking functionality of the pmod. Even though they increase power consumption, being able to observe the LEDs blinking is a nice feature to have during development and especially when debugging.

6. Setting the value of the sub-field *EVC_EN* to "1" to enable event counters such as *EVC_FFR* which indicates the number of frames rejected by the frame filtering function. These counters are also great tools to have during debugging because they allow us to see what happened after a series of transmissions.

7. Setting the value of the sub-field *TXPSR* to "2#10" which changes the preamble symbols to 1024. This change is made because without modifying the preamble symbols settings, the auto-acknowledgment feature used in section 4 can't work.

3.3.5 Setting up SFD

As we've seen in the previous sections, the physical layer of IEEE 802.15.4 is divided into multiple fields. Among them, the SFD sequence marks the end of the preamble of the frame. The DW1000 manages that part of the physical layer by itself when we want to send a frame. However, if the auto-acknowledgment is the first frame transmitted after startup, the SFD sequence won't be initialized because its initialization is only done at the first user transmission request. Therefore, if we want the auto-acknowledgment to work right away after startup we need to trigger the loading of the sequence. The user manual explains that the most efficient way to perform that is to "simultaneously initiate and abort a transmission" which resolves into setting the value of the flags *TXSTRT* and *TROFF* to "1" simultaneously. Figure 21 shows how this operation is performed by the driver.

```
setup_sfd(Bus) ->
    write_reg(Bus, sys_ctrl, #{txstrt => 2#1, trxoff => 2#1}).
```

Figure 21: Setup of the SFD inside the code

3.4 Transmission

The transmission and reception of UWB-PHY frames are the highest-level operations provided by the driver. At this level, it only treats the data payload as one single block of data that should be transmitted or received. It's the role of the higher layers to manage the eventual content and the structure of the payload. This section will explain in detail how these two operations are actually performed by the driver.

3.4.1 Sending a frame

The procedure to send a UWB-PHY frame is the following. First, we have to write the data inside the subregister *TX_BUFFER* .

Second, we have to set the value of a couple of sub-fields inside the register file *SYS_CTRL* . The sub-register *TXBOFFS* allows the user to specify an offset inside the transmission buffer indicating the first byte of the PHY payload. This allows further optimization but isn't used here and its value is set to 0. We also have to set the value of the sub-field *TFLEN* which indicates the size of the data portion of the frame plus a two bytes CRC. The DW1000 takes care of the computation of the CRC and **replace** the last two bytes of the payload by the newly computed CRC. Thus if we don't add these two bytes to the value written inside *TFLEN* , the actual data we are trying to send will be shortened by two bytes.

Third, we can trigger the start of the transmission by the DW1000 by writing the value "1" inside the sub-field *TXSTRT* of the register file *SYS_CTRL* .

Finally, we have to make sure that the transmission occurred correctly. This can be done by checking the event status bit *TXFRS* ("transmit frame sent") of the register file *SYS_STATUS* . This verification is performed by a recursive function. This function does a read request on *SYS_STATUS* and checks the value of *TXFRS* . If it is set to "0", then a recursive call is performed. Otherwise, if the value is set to "1", then the function returns `ok` and stops. The whole operation is operated synchronously and the calling API function won't return before the transmission has been performed.

At this stage, waiting for the completion of the transmission might seem useless because the frame is transmitted right away. However, in the case where frames are sent with a delay, this functionality is useful to avoid performing other operations

(e.g. turning on the receiver) before the actual transmission of the frame.

On the driver, there are two functions available in the API: `transmission/1` and `transmission/2`. They both take a bitstring in their first parameter but the second function allows the user to specify options for the transmission while the first one will use the default settings.

There are four options possible to set:

- `wait4resp`: It indicates that the receiver should be turned on after the transmission of a frame. The DW1000 will clear the bit after enabling the receiver, thus this setting must be set at every transmission of a frame that requires it. By default, this option is disabled.

- `w4r_tim`: It specifies the delay in microseconds between the transmission of a frame and the automatic enabling of the reception if `wait4resp` is enabled. This is useful in the case where some kind of delay between a request frame and its response is known (for example in the double-sided two-way ranging). It is set by default at $500\mu s$ and only used when `wait4resp` is enabled.

- `txdlys`: It enables the "transmitter delay sending" setting used to control precisely the transmission of a frame at a time specified in `tx_delay`. it is by default disabled and should be enabled for every transmission that requires the setting.

- `tx_delay`: It specifies the exact clock time when the transmission of the next frame should occur if `txdlys` is enabled. These two options are useful when the program needs to know the exact transmission time of a transmission.

These options are specified at the time of transmission for two reasons. First, this gives one more layer of abstraction which relieves the user from writing directly inside the register files.

Second, and most important, `WAIT4RESP` and `TXDLYS` are both bits that should be written at the same time (i.e. in the same SPI transaction) as `TXSTRT`.

In the beginning, the API was also able to take a String in the data argument. It was really useful in the first stages of the implementation of the transmission for testing purposes. However, as the development of the driver made progress, that option became more and more useless and was finally dropped from the API in the later versions

3.4.2 Receiving a frame

To perform a reception with the pmod UWB, the user can call two functions: `transmit/0` and `transmit/1`. The first one is a simplification of the first one by

setting to `false` by default the parameter of the second one. This parameter: `RXEnabled` specifies if the reception has been enabled prior to the function call. This is because in some cases the reception can automatically be turned on after the transmission of a frame. This parameter is there to avoid trying to enable the reception a second time which might trigger some event status bit specifying an error during the reception of the frame before receiving the actual frame.

On the DW1000, the reception can be divided into different steps described in the DW1000 user manual [2]:

1. Preamble detection: During that period, the device will try to detect the preamble sequence by cross-correlating chunks of the preamble symbols. It is possible to enable and set a timeout to allow the receiver to stop the detection of the preamble. If the timeout is triggered, the event status bit *RXRFTO* will be set to "1" and the reception will be aborted. Otherwise, if it isn't enabled or isn't triggered, *RXPRD* is set to "1" and the procedure continues to the next step

2. Preamble accumulation and SFD detection: after the detection of the preamble, the device will accumulate the preamble symbols and look for a particular sequence of symbols, the SFD. If the SFD isn't detected before a certain time after the detection of the preamble, the reception is aborted and *RXSFDTO* is set to "1". Otherwise, *RXSFDD* is set to "1" and the process moves to the next step.

3. PHR demodulation: At this step, the DW1000 will demodulate and decode the PHR inside the received frame. The PHR provides information about the length of the data payload and the data rate that has to be used in the demodulation of the payload. The PHR also contains a SECDED error check sequence able to correct one bit errors and detect two bits errors but can't correct them. If a two bits error occurs, then *RXPHE* will be set to "1" and by default will abort the reception. Otherwise, *RXPHD* is set to "1" and we proceed to the next step.

4. Data demodulation: In this step the data payload of the PHY frame is demodulated, and passed through the Reed Solomon decoder. If it detects a non-correctable error, it will set *RXFSL* to "1" and by default abort the reception. Afterward, the CRC of the frame is computed and checked with the actual transmitted CRC. If the values match, then *RXDFR* and *RXFCG* are set to "1". These bits indicate that a frame has been received correctly. Otherwise, if the CRCs don't match, *RXFCE* will be set to "1".

As we can see above, during the reception of a frame, flags are set depending on the situation, and in most cases, the reception will be disabled on the chip when

an error occurs. Therefore, during the reception of a frame, the driver has to check the status of the different error flags and the status of *RXFCG* which indicates the correct reception of a PHY frame. Also, these flags provide crucial information on the exact step where a reception failed. Additionally, if *EVC_EN* is set to "1", the DW1000 provides a register file with multiple sub-registers containing event counters that are incremented when specific errors occur. For example, the sub-register *EVC_STO* will count the number of times that a timeout occurred during the detection of the SFD.

The reception algorithm of the driver is quite simple. First, it will clear all the flags inside the register file *SYS_CTRL* that are related to the reception of a frame (error flags, timeout flags, and flags indicating a good operation). The driver performs this action because in some cases, some flags aren't reset by the chip when the reception is enabled. This can lead to a situation where the algorithm wrongly thinks that a frame has been received when it's not the case.

Second, if `RXEnabled` is set to false, then it performs a write in the sub-field *RXENAB* of the register file *SYS_CTRL* . This will turn on the antenna for reception.

Third, similarly to the transmission, the process will then call a recursive function that will check the value of the different error bit flags as well as the values of *RXFCG* and *RXDFR* . As long as none of these bits are set to one, the function will continue to perform recursive calls. If one of the error bits has a value of "1", the function will return the atom corresponding to the name of the sub-field. Otherwise, if *RXFCG* is set to one, the function will return `ok`. If a frame has been correctly received, the driver clears *RXFCG* by writing "1" inside of it. This is done because in some cases, when multiple frames were received, the bit wasn't reset when the reception was re-enabled. This made the driver believe that a correct reception occurred when it wasn't the case.

Finally, the driver pulls the received data from *RX_BUFFER* without the two CRC bytes and returns it in a tuple with the length of the payload minus the two last bytes.

At this stage, we could also reset the flags instead of doing it at the beginning of the reception. However, if we do that, that removes the debugging potential of these flags if something goes wrong during the operation. Additionally, another method is proposed in the code examples of the DW1000 API [8]. Indeed, the bits are cleared at the very end of the receptions in the main function. This method works but adds more load to the user who has to remember to perform the reset. Since the goal of the reception function is to totally abstract the whole operation from the user, the choice was made to hide the re-initialization of the flags inside the function call.

The user also has to be careful if `reception/1` is called with `RXEnabled` set

to 'true' without the reception being enabled first (automatically or manually). Indeed, the function won't enable the reception and will be stuck in a loop because no status flag will be set to indicate an error or a reception. Consequently, the driver will have to be restarted manually. This situation can't be avoided with defensive programming because there is no way for the driver to see if the reception has been enabled or not.

Chapter 4

MAC layer

The DW1000 provides support for the MAC layer but it doesn't implement it. It's the role of the host system to do it. In the context of this work, the goal was to provide support for a potential *GRiSP* application sending and receiving MAC frames using the pmod UWB. These features are essential in the potential future implementation of upper-level layers like 6LoWPAN.

In this section, the support provided by the DW1000 and its different features will be explained. Then the construction and decoding process of the header and the frame control of a MAC frame will be detailed. Afterward, it will describe how the messages are sent using the pmod. Additionally, a concrete example of a message exchange using the automatic acknowledgment feature will be described. Finally, an analysis of different measurements done on this example will be provided.

4.1 DW1000 support

The different MAC layer hardware features supported by the DW1000 are described in its user manual [2]. This section will provide a description of them following the specifications described in that document.

4.1.1 Frame filtering

The DW1000 provides a frame filtering feature that will parse MAC frames respecting the IEEE 802.15.4-2011 standard. To work, it must be enabled by writing the value "1" inside the configuration bit *FFEN* inside the register file *SYS_CFG* . When the feature is turned on, at the reception of a frame, it will check it with a set of rules and either accept the frame or reject it. If the frame is rejected, then the bit flag *AFFREJ* will be put at a value of "1".

The set of rules used to filter the frames described by the user manual is the following:

1. The type of the frame must match the ones allowed by the set of configuration bits inside *SYS_CFG* . For example, if only the configuration bit *FFAD* is set to "1", then the frame filtering will only accept MAC frames of type "Data".
2. The version of the MAC frame must be either 0x00 or 0x01.
3. If the destination PAN is present in the frame header, it has to be either the broadcast PAN ID (i.e. 0xFFFF) or it has to match the PAN ID saved inside the register file *PANADR* .
4. if the destination address is present, it must either be the short 16-bit broadcast address or match the short 16-bit address present in *PANADR* or match the 64-bit long address in *EUI* .
5. If the frame is a beacon frame, then it must come from the same PAN
6. The CRC must be correct

4.1.2 CRC generation and checking

On transmission, the DW1000 will compute the 2 CRC bytes of the MAC frame and include them at the end of the data payload. At the reception, it will compute the CRC of the received frame and compare it with the one received.

4.1.3 Automatic acknowledgement

With this feature, at the reception of a frame and if the settings are set correctly, the device will automatically send back an acknowledgment frame to the sender. In order to work, the frame filtering feature must be enabled on the receiver by setting *FFEN* to "1" in *SYS_CFG* . It must be set to accept the frame type that will be sent. Additionally, the receiver should also enable the auto-acknowledgment with *AUTOACK* set to "1". On the sender side, the MAC should be correctly formed and addressed per frame filtering rules and the acknowledgment request bit in the frame header should be set to "1".

The automatic acknowledgment feature of the DW1000 was one of the first features to be tested on the pmod once the transmission was working. Indeed, the feature gives instant feedback (acknowledgment or frame rejection) without passing through the driver which made testing easier. At first, the frame was hard-coded to understand how the feature was working and then the MAC layer got built around the feature.

4.2 MAC Header

To give support for the MAC layer, the program has to let the user encode and read MAC headers in an easy way. To do so, multiple options were possible.

First, use a function and have one parameter per possible field in the header. This solution could work but is quite tedious for the end user.

Second, regroup the fields in a tuple and put the tuple in the argument of the function. While this solution improves the first one because it avoids having a function with at least 13 parameters, its ergonomy isn't better. This is because the order of the parameters inside the tuple needs to be defined in advance. The user will have to memorize the order of all 13 parameters which is a great source of potential bugs.

Third, abstract the header and the frame control into 2 Erlang records. With this choice, the structure and the type of data used can be defined clearly and documented for later use. Furthermore, Erlang records can have default values. This means that the user doesn't have to specify every field if their default values match the intent of the user. For all these advantages, this solution was selected for the MAC layer of the pmod. Listing 4.1 gives the structure of the frame control record and listing 4.2 provides the one of the MAC header

```
1     -record(frame_control, {
2         frame_type = ?FTYPE_DATA :: ftype(),
3         sec_en = ?DISABLED :: flag(),
4         frame_pending= ?DISABLED :: flag(),
5         ack_req = ?DISABLED :: flag(),
6         pan_id_compr = ?DISABLED :: flag(),
7         dest_addr_mode = ?SHORT_ADDR :: addr_mode(),
8         frame_version = 2#00 :: integer(),
9         src_addr_mode = ?SHORT_ADDR :: addr_mode()}).
```

Listing 4.1: Frame control record

```
1     -record(mac_header, {
2         seqnum = 0 :: integer(),
3         dest_pan = <<16#FFFF:16>> :: addr(),
4         dest_addr = <<16#FFFF:16>> :: addr(),
5         src_pan = <<16#FFFF:16>> :: addr(),
6         src_addr = <<16#FFFF:16>> :: addr()}).
```

Listing 4.2: MAC header record

The encoding and decoding of the frame control are quite easy to perform since it has a fixed size. However, for the MAC header as a whole, these two operations need to be performed rigorously. Indeed depending on the settings set in the frame control, the size of some fields can change and some are even removed. Therefore, all the edge cases need to be checked. Because of that, some custom unit tests

have been created to make sure that the functions were able to create and parse correctly MAC frames. The implementation of these tests can be found in appendix D. Furthermore, to avoid any comprehension mistakes in the tests, some of the frames tested are the ones used in the code example given by Qorvo in the DW1000 API [8]. Still, these tests don't cover all of the edge cases, but they are a great tool to have during the development of the MAC layer to make sure that the code works before having to run it on the *GRiSP* board connected to the DW1000 which provides little debugging help.

For the parsing of the MAC header, we know that in every case, the first two bytes are the one of the frame control and the third byte is the sequence number. For the remaining bytes, we need to parse the PAN id and address fields based on the settings of the frame control following this sequence of steps:

1. If the destination address is present, the next two bytes are the destination PAN ID and we can continue to parse the destination address (2). Otherwise, we can jump (3).

2. If we reached that point, it means that the destination address mode is either the short address or the extended address. If it's the short address then the next 2 bytes are the address. Otherwise, if it's the extended address, then the next eight bytes represent the address. Then we can continue to parse the source PAN ID (3).

3. Here, the situation is a bit more complex, and multiple situations are possible:

   - If the source address mode is set to *2#00*, then neither the source PAN ID nor the source address is present in the header and the remaining bytes are the payload of the frame (if any) plus the CRC bytes.
   - If the source address mode isn't *2#00* and the PAN ID compression is disabled, then the next two bytes are the source PAN ID and we can go to (4).
   - If the source address mode isn't *2#00* and the PAN ID compression is enabled, then we need to check the destination address mode. If it isn't set to *2#00*, then the source PAN isn't present and we jump to (4).

4. Source address: Similarly to the destination address, if the frame control settings of the destination address say that a short address is used, then the next 2 bytes are the source address. If it says that an extended address is used, then the next 8 bytes are the source address. Then, what's left over is the payload and the 2 bytes of CRC.

This whole parsing operation is performed in the internal private function: `decode_addrs/3`.

After the parsing, some missing values can be deduced based on the settings set in the frame control. Indeed, if the PAN ID compression is enabled, then we can deduce the missing PAN ID based on the PAN ID present in the header. Additionally, if the compression is disabled, in some specific cases, it is still possible to deduce the missing values. Indeed, in the case of a missing destination PAN ID and address, if the frame isn't an acknowledgment nor a beacon frame then the frame destination is the PAN coordinator with the same PAN ID as the source. Furthermore, if the source address mode is set to *2#00* and the frame isn't an acknowledgment, then its source is the PAN coordinator with the same PAN ID as the destination.

We must also be careful of the endianness of the fields. Indeed, the DW1000 works with big-endian fields. Thus, the encoding and decoding operations must take that into account when working with a MAC header.

4.3 Transmission

4.3.1 Sending

To send a MAC frame the user can use `mac_send_data/3` and `mac_send_data/4`. The only difference between these two functions is that `mac_send_data/3` will perform transmission with the default transmission settings while `mac_send_data/4` lets the user define transmission settings useful for the MAC layer and the protocols on top of it like two way ranging. To use these functions, the user has to give the frame control and the MAC header as records and an Erlang bitstring for the payload. Since the CRC bytes are handled internally by the DW1000, they must not be included in the payload.

Internally, they are built on top of the transmission functions defined in section 3.4.1 so the only work for this layer is to build correctly the MAC frame based on the user choices for the frame control, the mac header and the payload. To do so, they call the internal function `mac_frame/3` which will build the frame into a bitstring and then appends it with the payload. The function then calls `transmit/2` which will send the MAC frame using the specified settings.

4.3.2 Receiving

To receive a MAC frame, the user can use either `mac_receive/0` or `mac_receive/1`. These functions are also based on the reception functions described in section 3.4.1. Thus, their arguments represent whether or not, the reception has been enabled prior to the call of the function. They both return a tuple of three elements: the

frame control, the MAC header, and the payload. Since they call `reception/1`, these two functions are synchronous and won't return before the reception of a frame, a timeout, or a reception error.

4.4 Example: Using the automatic acknowledgment feature of the DW1000

This section will illustrate the utilization of the MAC layer with a *GRiSP* application. The setup needed is composed of two *GRiSP 2* boards with a pmod UWB with the antenna at a distance of two meters.

One of the boards will be the *sender* and sends all the data frames and wait for the acknowledgment. The second board, the *receiver*, waits for data frames and automatically sends an acknowledgment frame to the sender. More precisely, the *sender* will send a total of n frames and after the transmission of each frame, it will automatically enable the reception and wait for the acknowledgment from the receiver. If a reception error or a timeout occurs, then the *sender* will try to send the frame again. If that retry process fails m times for the same frame, then an error is thrown and the experiment ends. On the *receiver* end, the auto-acknowledgment feature of the DW1000 is activated and it will wait for the reception of frames indefinitely and can only be stopped if the power is removed.

When the protocol was working under a stable network, some jitter got introduced within the *receiver* to test the ability of the sender to send back non-acknowledged frames. The jitter is simulated by getting a random number with `rand:uniform/1` and if that number is equal to one, it executes `timer:sleep(200)` which suspends the process of the *receiver* for 200ms. This has the consequence to enable the reception too late and miss the frame sent by the *sender* and thus makes it send back the lost frame.

There are two versions of this example present in the appendix of this document and on GitHub. The first one `ack_no_jitter` (appendix C.1) performs the protocol described here without artificial jitter. The second one, `ack:_jitter` (appendix C.2) introduces jitter for about 25% of the frames. To run both of these examples, you have to set up a development environment as described in the *GRiSP* wiki [1]. For both of them, the receiver has to be started by using `test_receiver_ack/0` on one device. Then the sender has to be launched on the second one by using `test_sender_ack/2` without forgetting to specify the number of frames to send during the exchange in the first argument and the size of the frames to send in the second one.

[1] https://github.com/grisp/grisp/wiki/Setting-Up-a-Development-Environment

4.5 Measurements

When the exchange of frames is done, some statistics are shown to the user. An example of such a report can be seen in figure 22. This gives us an approximation of the capacities of the driver. The statistics shown are the following:

- The number of frames sent gives the total number of sent frames (successful or not)
- "Success rate": the ratio between the number of successful frames (i.e. with an ACK from the receiver) and the total number of frames sent
- "Error rate": the ratio between the number of frames with an error and the total number of frames sent
- "Data rate": the ratio between the total amount of data sent and the total execution time. This gives the number of bits per second sent in average over the whole execution
- "Total time": the total execution time

```
------------------- Report -------------------
Sent 2004 frames - Success rate 0.998 (2000/2004) - Error rate 0.002 (4/2004)
Data rate 11080.9 b/s - In 167.494825 s
----------------------------------------------
```

Figure 22: Example of the statistical report for an exchange of 2000 frames containing 116 bytes of data

The measurements were done with three different *GRiSP* applications: `test_ack_no_jitter`, `test_ack_jitter` both described previously and `ack_fast_tx` which tries to send each frame as fast as possible. Compared to the first two applications, `ack_fast_tx` writes a frame inside the *TX_BUFFER* once at the beginning of the execution and sends it repeatedly during the whole execution.

Table 4.1 gives some measurements done on the devices with their antenna put two meters apart. For each run, the devices tried to send 10000 frames with a MAC data payload of 116 bytes to give a total MAC frame size of 127 bytes:

Name	**Total number of frame sent**	**Success rate**	**Data rate (bps)**	**Total time (s)**
test_ack_no_jitter	10002	1.000 (rounded)	32444.7	286.03
test_ack_jitter	17228	0.580	9515.9	975.21
ack_fast_tx	20032	0.499	12268.1	756.44

Table 4.1: Measurements for an exchange of 10000 frames between two devices 2 meters apart

As we can see, the exchange of frames without any jitter is highly reliable as only one frame got lost during the whole execution. The measured data rate of 32.445 kbps is smaller than the maximum raw data rate described by IEEE 802.15.4-2011 [1]. Yet, the driver isn't optimized and leaves space for improvements, especially at the lower level when values are written in the registers. Additionally, the protocol used here makes the sender wait for an acknowledgment after the transmission of each data frame which makes the transmission quite robust but also slow. However, in reality, the MAC layer acknowledgments are rarely used to the profit of the ones in higher layers. For example, we could imagine a protocol implemented on top of the MAC layer where one acknowledgment is sent every n frame.

Another interesting point of these results shows that despite having a lower success rate than `test_ack_jitter`, `ack_fast_tx` is faster and has a better data rate. This can be explained by the fact that the frames are sent faster since `ack_fast_tx` doesn't have to encode and write the content of the MAC frame inside *TX_BUFFER* for each transmission and retransmission.

Chapter 5

Two way ranging

UWB has great multipath resolution capability [27] and doesn't suffer from multipath fading [4]. This ability provides a fine delay resolution property and makes this technology well-suited for time-of-arrival-based techniques to achieve accurate localization [28]. In fact, UWB systems are the most accurate time-based technique used in geolocation compared to narrowband systems like Bluetooth. This is due to the fact that the accuracy of the estimation of the time of arrival is directly proportional to the size of the bandwidth [29]

In this study, we use one particular ranging technique: two-way ranging. This technique is used here because the DW1000 provides facilities that enable such algorithms [2]. However, this isn't the only technique to perform ranging with UWB. Indeed, angle of arrival (AOA) based algorithms compute the position of an object based on the estimation of the signal reception angles. Additionally, RSS-based algorithms estimate the position of the target based on the signal strength. Moreover, time difference of arrival (TDOA) techniques measure the time difference of arrival of a signal on different reference points. Lastly, some hybrids techniques, combining multiple position techniques are possible, like using both GPS and UWB or even combining AOA algorithms with TDOA with an extended Kalman filter [30]. However, a detailed discussion about these methods is outside the scope of this work.

This chapter will first give some examples of different applications using two-way ranging with UWB. Before describing the two methods used, namely single-sided two-way ranging and double-sided two-way ranging. Then, it will explain the implementations of the two methods using the MAC layer built in chapter 4. Finally, it will analyze the measurements performed on the pmod UWB using these methods.

5.1 Methods

The methods presented here estimate the distance between two transceivers by computing the time of flight (TOF) during an exchange of messages. The formulas presented here are described in [2] and [31].

5.1.1 Single-sided two-way ranging

In this method, only two messages are exchanged between device A, the *initator*, and device B, the *responder*. The protocol starts with Device A sending a poll message at time TX_{poll}. Device B then receives it at time RX_{poll} and replies with a message at time TX_{resp}. Finally, device A receives the reply at time RX_{resp}. Figure 23 gives a full overview of the messages exchanged during the protocol.

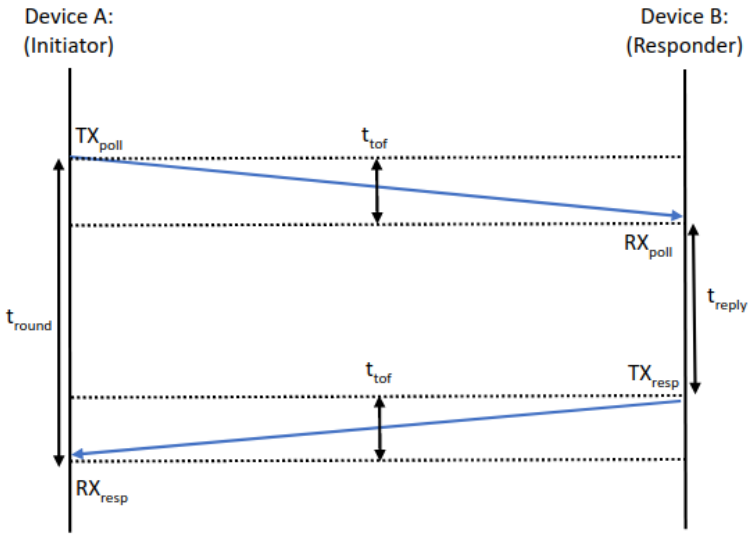

Figure 23: Message exchanges of single sided two way ranging

With these four timestamps it is possible to find t_{round} and t_{reply} using the formulas 5.1

$$\begin{aligned} t_{round} &= RX_{resp} - TX_{poll} \\ t_{reply} &= TX_{resp} - RX_{poll} \end{aligned} \tag{5.1}$$

Thus, we can find t_{tof} with 5.2:

$$t_{tof} = \frac{1}{2} * (t_{round} - t_{reply}) \tag{5.2}$$

Finally, the product of t_{tof} and the speed of light will give us the distance between the *initiator* and the *responder*.

This method has the advantage of being simple. Yet, since devices A and B compute t_{round} and t_{reply} with their own clocks, the method suffers greatly from the clock offset errors of the two local clocks from their nominal frequencies. As a result, the more t_{reply} is big, the more the error contained in $\hat{t}_{tof}$ (the estimated time of flight based on the real observations) increases. For example, according to [2] if t_{reply} is equal to 500 μs and the clock error is equal to 5ppm, then the induced error in the time of flight estimation is equal to 1.25ns. This means that the error in the estimated distance is around 37cm. Therefore, if t_{reply} is too big, the estimated distance is too inaccurate to be used in a real application.

5.1.2 Double-sided two-way ranging

In double-sided two-way ranging, different protocols are possible. This work only explores the alternative double-sided two-way ranging presented in [31] and also displayed in the DW1000 user manual [2] as "double-sided two-way ranging with three messages".

The exchange of the first two messages is similar to the single-sided method. Device A sends the first message at time TX_{poll} and is received by device B at time RX_{poll}. Then device B sends the second message at time TX_{resp} and device A receives it at time RX_{resp}. Device A will then send the third and final message at time TX_{final} and device B receives it at time RX_{final}.

With these six timestamps, it is possible to compute the following time periods:

$$\begin{aligned} t_{round1} &= RX_{resp} - TX_{poll} \\ t_{reply1} &= TX_{resp} - RX_{poll} \\ t_{round2} &= RX_{final} - TX_{resp} \\ t_{reply2} &= TX_{final} - RX_{resp} \end{aligned} \tag{5.3}$$

Additionally, we can define t_{round1} and t_{round1} in terms of t_{tof}, t_{reply1} and t_{reply2}:

$$\begin{aligned} t_{round1} &= 2t_{tof} + t_{reply1} \\ t_{round2} &= 2t_{tof} + t_{reply2} \end{aligned} \tag{5.4}$$

We can multiply both t_{round1} and t_{round2}:

$$t_{round1} * t_{round2} = (2t_{tof} + t_{reply1}) * 2t_{tof} + t_{reply2} \tag{5.5}$$

If we isolate t_{tof}, it gives us:

$$t_{tof} = \frac{t_{round1} * t_{round2} - t_{reply1} * t_{reply2}}{t_{round1} + t_{round2} + t_{reply1} + t_{reply2}} \tag{5.6}$$

Finally, we can compute the distance between device A and device B by multiplying t_{tof} with the speed of light.

Figure 24 displays an overall picture of the whole exchange of messages.

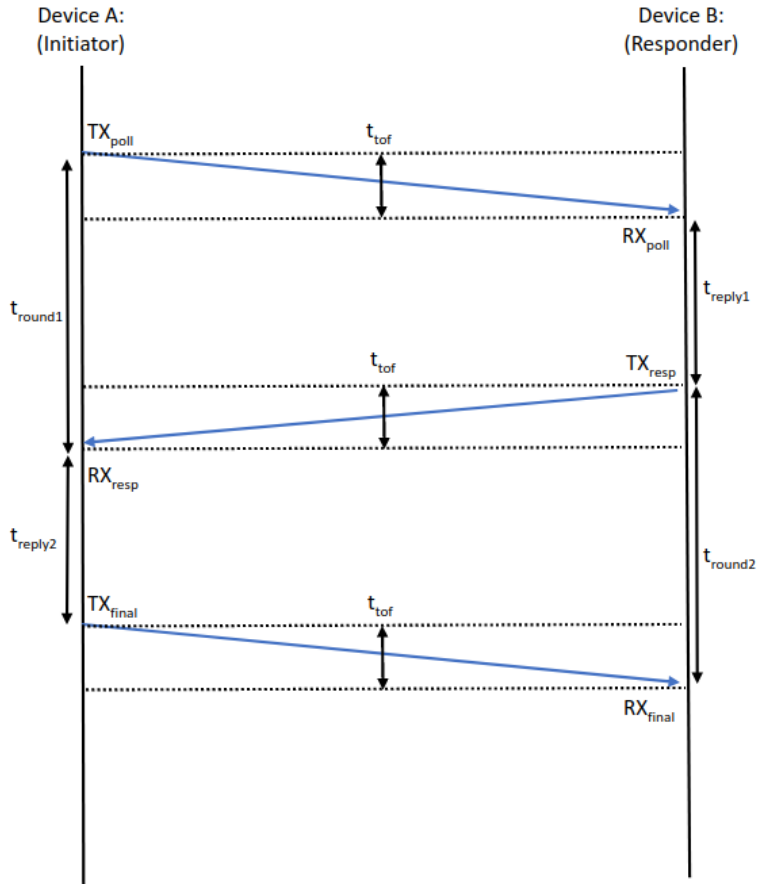

Figure 24: Message exchanges of double sided two way ranging

Compared to single-sided two-way ranging, this method is less affected by clock offset errors. Indeed, according to the DW1000 user manual [2], even with a clock offset error of 20 ppm, the induced error in the time of flight estimation is in the order of the picoseconds. Yet, error analysis only based on clock drift errors doesn't provide the full picture, and other sources of errors still persist which influences the accuracy of the real measurements if they aren't mitigated. [31]

5.2 Implementations

The implementations of the two methods described here are based on the code provided by Qorvo in their implementation of the DW1000 driver [8] and on the information contained in the user manual [2]. The codes of the protocols are a simple translation of the code provided to the Erlang programming language using the API of the driver implemented in the last two chapters.

5.2.1 Single-sided two-way ranging

The implementation of the single-sided two-way ranging can be found in appendix C.4 It is composed of two processes that should be run on two different *GRiSP* boards. Both processes will by default try to perform 250 frame exchanges but this value can be changed by modifying the value of the macro `NBR_MEASUREMENTS`.

The first process, `ss_initiator/0`, is the *initator* of the protocol. It will send the *poll* message, receive the *response* from the *responder*, and compute the distance between the two boards. It starts by setting up the transmission and reception antenna delay. These values are added to the raw timestamps to compensate for the delay between the internal digital timestamp of the RMARKER and the actual time the RMARKER is at the antenna [2]. Afterward, the process will call the protocol loop and start performing the measurements. One iteration of the protocol loop performs the following steps:

1. Send the poll message, a MAC data frame with the payload value "GRiSP"
2. Receive the response message from the responder and extracts the timestamps values
3. Read the value of TX_STAMP and RX_STAMP , which represent the timestamp of the transmission of the poll message and the timestamp of the reception of the response respectively
4. Compute the value of t_{round} and t_{reply} in the device time units which are around 15.65 picoseconds
5. Get the clock offset ratio between the two devices computed by the DW1000 at the reception
6. Compute the time of flight by using equation 5.2 applied with a correction of the clock offset and by multiplying the result by $15.65e-12$ to convert the value in DW1000 time unit in seconds

7. Compute the distance by multiplying the time of flight converted in seconds with the speed of light

The clock offset compensation is an addition by the Qorvo engineers in the DW1000 API example code [8] to increase the accuracy of the measurements by reducing the error induced by the difference of the clock frequencies of the two devices.

The second process, `ss_responder/0` is the *responder* of the protocol. It will receive the *poll* message and responds with a message including the reception and transmission timestamps. Like the *initiator*, the process starts by setting up the antenna delay and then starts the protocol loop. One iteration of the loop performs the following steps:

1. Receive the poll message from the *initiator*
2. Read the reception timestamp of the poll message
3. Compute the delayed transmission time by adding $20000\mu s$ converted to the DW1000 time unit to the reception timestamp.
4. Write the delayed transmission time in *DX_TIME*
5. Compute the estimated transmission time with the antenna delay based on the delayed transmission time computed in the previous step
6. Send a MAC data frame containing the reception timestamp of the poll message and the estimated transmission timestamp of the response message

The delay added before the transmission of the response at step 3 is set to let enough time for the driver to perform steps 2 to 6 after the reception of the first message. If this value is smaller, the *responder* won't have the time to perform these steps before the planned transmission time and make the protocol fail.

5.2.2 Double-sided two-way ranging

Similarly to the single-sided two-way ranging, the double-sided two-way ranging protocol implementation is composed of two processes that should be run on two different *GRiSP* boards. Their implementation can be found in appendix C.5.

When the protocol starts, the boards will perform 250 measurements before providing the measurements of the exchange. However, this time, the computation of the distance is done in the *responder* instead of the *initiator*. Both processes start by setting up their antenna delay and then start their protocol loop.

The protocol loop of the *initiator* performs the following operations:

1. Send the poll message to the *responder*
2. Receive the response message from the *responder*.
3. Read the transmission timestamp value of the poll message and the reception timestamp of the response message
4. Compute the delayed transmission time by adding to the reception timestamp, $30000\mu s$ converted to DW1000 time unit and write the result in *DX_TIME*
5. Compute the estimated transmission time of the final message by adding the antenna delay to the delayed transmission time computed in the previous step
6. Send a MAC data frame containing the transmission timestamp of the poll message, the reception timestamp of the response message, and the estimated transmission timestamp of the final message

The protocol loop of the *responder* performs the following operations:

1. Receive the poll message from the *initiator* and read the reception timestamp
2. Compute the delayed transmission time of the poll message by adding $30000\mu s$ converted to DW1000 time units to the reception timestamp value and writes it in *DX_TIME*
3. Transmit the response message with a payload value set to "Resp_TX".
4. Receive the final message from the *initiator* and extract the different timestamps inside the data payload
5. Read the transmission timestamp of the response message and the reception timestamp of the final message
6. Compute r_{round1}, t_{round2}, t_{reply1}, t_{reply2} using the different timestamps as described in equation 5.3
7. Compute t_{tof} in seconds by using equation 5.6 and converting its result to seconds.
8. Compute the distance in meters between the two devices by multiplying t_{tof} with the speed of light

5.2.3 Counter wrap around

In some cases, for both methods, the measured distance can have a massive negative value. This is due to a wrap-around in one of the device counters during the exchange of messages. Consequently, one of the values computed either with equation 5.1 for the single-sided method or with equation 5.3 in the case of the double-sided method has a negative value. This could happen quite often because the counter wrap-around period of the clock is 17.2074 seconds. Therefore, in the implementation of both methods, the measurement performed is thrown away if the natural order of the timestamps isn't respected. For example, in single-sided two-way ranging, the result isn't saved if the reception timestamp of the response is smaller than the transmission timestamp of the polling message or if the transmission timestamp of the response is smaller than the reception timestamp of the polling message.

5.3 Measurements

The first series of measurements have been performed by placing the devices at a known distance without moving them during the whole operation. For each two-way ranging method, one measurement has been performed at 25cm, at 2m with a clear line of sight, and at 2.5m with a wall between the *initiator* and the *responder* (i.e. without a clear line of sight). Table 5.1 gives the results of these static measurements:

Type of Measure	**Method**	**Average distance (m)**	**Standard deviation (m)**	**Minimum (m)**	**Maximum (m)**
25cm	single sided	-0.6107	0.6827	-2.5945	0.6146
	double sided	-0.1942	0.1496	-0.4381	0.0983
2m clear line of sight	single sided	2.8383	0.4531	1.8274	3.9669
	double sided	2.0969	0.1516	1.8191	2.4192
2.5m no clear line of sight	single sided	1.8247	0.38775	1.401	2.5382
	double sided	2.2730	0.1464	1.9918	2.5996

Table 5.1: Different measurements results made with single-sided two-way ranging and double-sided two-way ranging

The first elements we can see from these results are the negative values at close range. They are probably linked to the fact that the devices are not fully calibrated.

Secondly, we can observe that the standard deviations of the double-sided two-way ranging methods are all within $\sim$ 15cm. This shows that the measurements aren't too spread out. Additionally, when we look at the single-sided two-way ranging, the standard deviation of the three measures shows that the measurements are more spread out and thus less precise. The average distance measured is also way less precise even with the clock offset correction introduced by Qorvo. Yet, the average distances measured are all within the 1-meter range which is more accurate than expected with the long t_{reply} of the implementation.

Thirdly, we can see that even without a clear line of sight, the measured distance is still precise which corroborates with the results of [13].

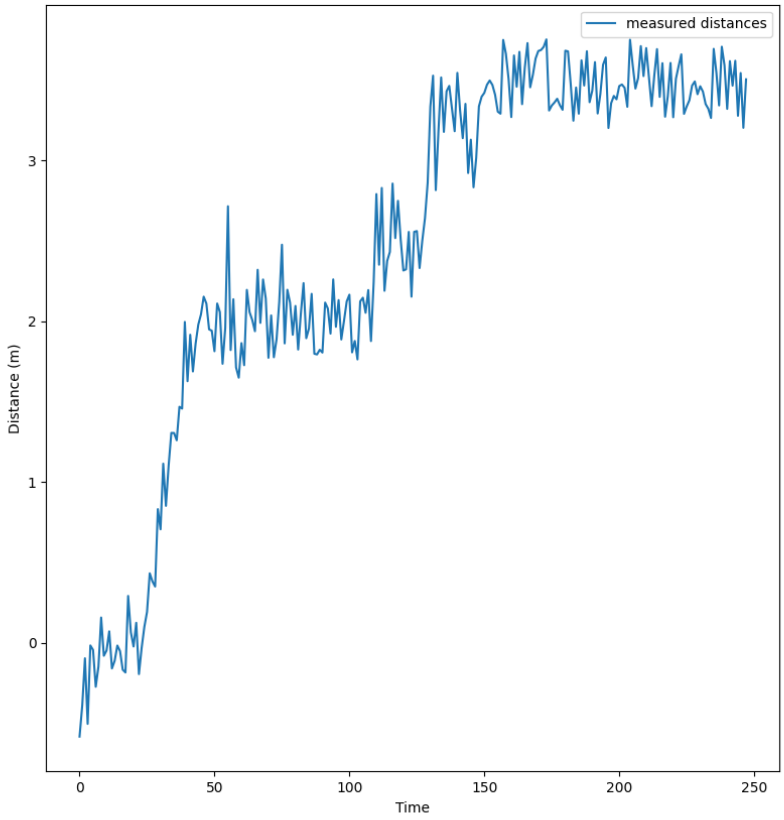

Figure 25: Graph showing the measured distance

Finally, a last set of measures has been performed with the double-sided two-way ranging method. This time, compared to the previous measurements done, one

device moved during the execution of the protocol.

More precisely, at the start, the devices were placed at a distance of 30cm. Then one of the devices moved at a distance of 2m where it stayed for a short period of time. Finally, the board was moved again at a distance of 3m from the other device and stayed there until the end of the measurements.

The plot in figure 25 shows the evolution of the measured distance over time. On the graph, the different stages of the experiments are visible with two distinct plateaux.

Chapter 6

Conclusion

6.1 Future work

Even though the objectives of this thesis were met. The current implementation of the driver and MAC layer can still be improved. This section details the possible upgrades that could be applied.

6.1.1 Improvement of the driver

First of all, the driver could be optimized to increase the data rate of an exchange of frames between two devices. The read/write operations were written with the idea of having a reliable and uniform implementation, thus no optimization was applied. However, these two operations are used by all the upper layers, meaning that if they are slow they will also affect the performances of the operations above them. One possible enhancement could be that when a user writes a value in a register file, its whole content is read first, then the value is changed and finally the whole content is written again. This procedure works correctly but in the case where a user wants to change the value of a single-bit flag, this method is quite inefficient. One could try to optimize that process by using offset and requesting the least amount of bytes possible while reading the register file.

Second, a more precise calibration of the devices should be done to increase the accuracy of the measurements but also avoid negative values at close range.

Third, the transmission of a frame will be a central element of all the potential applications that could be built on top of the driver. For that reason, we don't want this operation to be a bottleneck for future implementations. A big improvement of that functionality would be to build a Native Implemented Function (NIF). This would give us the advantage of the speed of the C programming language, but also we would give more control over its execution. For example, we would be able to

give higher priorities on the hardware level to avoid preemption from the CPU.

Fourth, this work was more focused on having basic functionalities like transmission and reception working than applying possible optimization. However, power optimization is an important point of any low-power IoT applications. In the future, the driver should support the sleep and deep sleep functionalities of the DW1000 to reduce power consumption when the device is idle.

In addition, the DW1000 also supports a non-standard PHY packet size of 1023 bytes. Even though packets of this size don't comply with the IEEE 802.15.4-2011 standard anymore if the data rate isn't sufficient for a specific application, one can extend the driver to support this extended frame size.

Finally, the double buffering mode of the DW1000 wasn't exploited in this study and the current version of the driver doesn't support it. With this feature enabled, the DW1000 is able to receive a frame while the host system performs operations on previously received data and thus should increase the transmission data rate. For that reason, future work should extend the driver to enable data exchanges using the double buffering mode.

6.1.2 MAC layer

With the current implementation of the MAC layer, an application is only able to encode a MAC frame before the transmission, decode a received MAC frame, and use the AUTOACK functionality of the DW1000. However, this covers only a small area of the functionalities and the responsibilities of the layer. Some functionalities like CSMA-CA are crucial for any potential applications using a network of devices. Hence, it is primordial to extend this layer before building the upper layers.

Additionally, no security is implemented on the current MAC layer. Nevertheless, IOT security is something important as more and more IOT devices are present in our daily lives. Thus, this aspect of the MAC layer shouldn't be overlooked and security support should be implemented in a future extension of the MAC layer.

6.1.3 Upper layers

Since the DW1000 is IEEE 802.15.4-2011 compliant, any upper level supporting this standard could be implemented on top of this work. In particular, an implementation of 6LoWPAN on the *GRiSP* is something that would open the door to a lot of IoT applications needing low power communication over a network of nodes inside a PAN. The cards could, for example, carry other pmods to perform different measurements and use 6LoWPAN over UWB communications to send them to a border router.

In addition, on top of 6LoWPAN, the Thread protocol is used by big companies in the IOT industry. Implementing the protocol on *GRiSP* could make the card

interoperable with smart home solutions already available in the market.

6.1.4 Adaptation of the *GRiSP* toolchain

The DW1000 allows the host to trigger a reset through the RSTn pin. Additionally, as described in section 3.1, on the rising edge of the signal, the GPIO 5 and 6 (controlling the SPIPOL and SPIPHA) are sampled. However, according to the DW1000 datasheet [3], the GPIO pin controlling the RSTn line should be configured as high-impedance. This type of setting for the pins of the *GRiSP 2* isn't supported yet by the *GRiSP* runtime library and the *GRiSP* toolchain. Indeed, currently, *GRiSP 2* uses a static Flattened Device Tree (FDT) to configure the hardware through the RTEMS Board Support Packages (BSP) and the bootloader. To let a user change the settings of a single pin at the initialization of a driver, future work should adapt the toolchain to support FDT overlays which will allow overwriting entries of the FDT to correctly setup the GPIO pin and then trigger a reset on the DW1000.

6.2 Results

To conclude, this thesis displays a successful implementation of a driver for the new pmod UWB built by the company Peer Stritzinger GMbH. On top of this, a simple MAC layer has been developed to send and receive MAC frames. The measurements performed on that layer showed that the current implementation is already able to achieve transmission at a data rate of 324443.7 bps (32.44 kb/s). Using the implemented MAC layer, it was then possible to perform two-way ranging with the single-sided two-way ranging and double-sided two-way ranging methods. Once again, the different measurements done with both methods were satisfying. The single-sided method revealed to be more precise than expected due to the introduction of the carrier integrator value by the engineer of Qorvo in their own implementation of the method and used as a basis in this work. Furthermore, the results of the double-sided methods showed a great ranging accuracy with a relatively small standard deviation within a sample of static measurements. Additionally, the results provided by both methods on ranging operations without a clear line of sight are consistent with the literature and show the ability of the UWB technology to penetrate through obstacles and achieve data transmission and accurate ranging operations.

All the code of the driver, the upper layers, and the examples used in this thesis can be found in the appendix of this document as well as on the GitHub repository of this work [1].

[1] `https://github.com/GwendalLaurent/pmod_uwb`

Chapter 7

Bibliography

[1] IEEE Computer Society, "IEEE Standard for Local and metropolitan area networks–Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)," *IEEE Std 802.15.4-2011 (Revision of IEEE Std 802.15.4-2006)*, pp. 1–314, 2011.

[2] *DW1000 user manual*, Decawave (Qorvo), 2017, version 2.18.

[3] Qorvo, "DW1000 datasheet," 2017, rev. 2.22.

[4] Win, Moe Z and Scholtz, Robert A, "On the robustness of ultra-wide bandwidth signals in dense multipath environments," *IEEE Communications letters*, vol. 2, no. 2, pp. 51–53, 1998.

[5] R. S. Kshetrimayum, "An introduction to UWB communication systems," *IEEE Potentials*, vol. 28, no. 2, pp. 9–13, 2009.

[6] *Getting Back to Basics with Ultra-Wideband (UWB)*, Qorvo, May 2021, White paper.

[7] Samsung, "Samsung Announces Ultra-Wideband Chipset With Centimeter-Level Accuracy for Mobile and Automotive Devices," 21/03/2023. [Online]. Available: https://news.samsung.com/global/samsung-announces-ultra-wideband-chipset-with-centimeter-level-accuracy

[8] Qorvo, "DW1000 API with STM32F10x Application Examples." [Online]. Available: https://www.qorvo.com/products/d/da008000

[9] S. Kalbusch, V. Verpoten, and P. Van Roy, "The Hera framework for fault-tolerant sensor fusion on an Internet of Things network with application to inertial navigation and tracking," Ph.D. dissertation, Master's thesis. UCLouvain. http://hdl. handle. net/2078.1/thesis: 30740, 2021.

[10] S. Bojabza and P. Van Roy, " Protocol stack for 802.15. 4 based personal network (6LoWPAN)[GRiSP project with Stritzinger]."

[11] Yang, Liuqing and Giannakis, Georgios B, "Ultra-wideband communications: an idea whose time has come," *IEEE signal processing magazine*, vol. 21, no. 6, pp. 26–54, 2004.

[12] Porcino, Domenico and Hirt, Walter, "Ultra-wideband radio technology: potential and challenges ahead," *IEEE communications magazine*, vol. 41, no. 7, pp. 66–74, 2003.

[13] Li, Jing and Zeng, Zhaofa and Sun, Jiguang and Liu, Fengshan, "Through-wall detection of human being's movement by UWB radar," *IEEE Geoscience and Remote Sensing Letters*, vol. 9, no. 6, pp. 1079–1083, 2012.

[14] Macoir, Nicola and Bauwens, Jan and Jooris, Bart and Van Herbruggen, Ben and Rossey, Jen and Hoebeke, Jeroen and De Poorter, Eli, "Uwb localization with battery-powered wireless backbone for drone-based inventory management," *Sensors*, vol. 19, no. 3, p. 467, 2019.

[15] Yao, Leehter and Wu, Yeong-Wei Andy and Yao, Lei and Liao, Zhe Zheng, "An integrated IMU and UWB sensor based indoor positioning system," in *2017 International Conference on Indoor Positioning and Indoor Navigation (IPIN)*. IEEE, 2017, pp. 1–8.

[16] Kamel Boulos, Maged N and Berry, Geoff, "Real-time locating systems (RTLS) in healthcare: a condensed primer," *International journal of health geographics*, vol. 11, no. 1, pp. 1–8, 2012.

[17] GRiSP, "GRiSP technical specifications," https://www.grisp.org/specs/, 2021.

[18] RTEMS, "RTEMS real time operating system (RTOS)," https://www.rtems.org/, 2021.

[19] F. Leens, "An introduction to I 2 C and SPI protocols," *IEEE Instrumentation & Measurement Magazine*, vol. 12, no. 1, pp. 8–13, 2009.

[20] *AN991/D: Using the Serial Peripheral Interface to Communicate Between Multiple Microcomputers*, NXP Semiconductors, 2002, rev. 1.

[21] Qorvo, "DWM1000 datasheet," 2016, rev. 1.8.

[22] Ericsson, "gen_server Behaviour," 2023. [Online]. Available: https://www.erlang.org/doc/design_principles/gen_server_concepts.html#synchronous-requests---call

[23] GRiSP, "grisp," https://github.com/grisp/grisp, 2023.

[24] Fred Hebert, "Who Supervises The Supervisors?" n.d. [Online]. Available: https://learnyousomeerlang.com/supervisors

[25] G. R. Aiello and G. D. Rogerson, "Ultra-wideband wireless systems," *IEEE microwave magazine*, vol. 4, no. 2, pp. 36–47, 2003.

[26] *Correlation.* John Wiley Sons, Ltd, 2000, ch. 9, pp. 349–392. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/047120059X.ch9

[27] Win, Moe Z and Scholtz, Robert A, "Impulse radio: How it works," *IEEE Communications letters*, vol. 2, no. 2, pp. 36–38, 1998.

[28] Dardari, Davide and Conti, Andrea and Ferner, Ulric and Giorgetti, Andrea and Win, Moe Z, "Ranging with ultrawide bandwidth signals in multipath environments," *Proceedings of the IEEE*, vol. 97, no. 2, pp. 404–426, 2009.

[29] Ghavami, Mohammad and Michael, Lachlan and Kohno, Ryuji, *Ultra wideband signals and systems in communication engineering.* John Wiley & Sons, 2007.

[30] A. Alarifi, A. Al-Salman, M. Alsaleh, A. Alnafessah, S. Al-Hadhrami, M. A. Al-Ammar, and H. S. Al-Khalifa, "Ultra wideband indoor positioning technologies: Analysis and recent advances," *Sensors*, vol. 16, no. 5, p. 707, 2016.

[31] C. Lian Sang, M. Adams, T. Hörmann, M. Hesse, M. Porrmann, and U. Rückert, "Numerical and experimental evaluation of error estimation for two-way ranging methods," *Sensors*, vol. 19, no. 3, p. 616, 2019.

Appendix A

Driver code

```
-define(ENABLED, 2#1).
-define(DISABLED, 2#0).
-type flag() :: ?DISABLED| ?ENABLED.

-type miliseconds() :: integer().
% w4r_tim is the delay between the tx is done and the moment the
    rx will be enabled (it is not a timeout)
-record(tx_opts, {wait4resp = ?DISABLED:: flag(), w4r_tim = 0 ::
    miliseconds(), txdlys = ?DISABLED:: flag(), tx_delay = 300 ::
    integer()}).

% map the r/w bit of the transaction header

-type writeOnly() :: tx_buffer.
-type readOnly() :: dev_id | sys_time | rx_finfo | rx_buffer |
    rx_fqual | rx_ttcki | rx_ttcko | rx_time | tx_time | sys_state
    | acc_mem.
```

Listing A.1: pmod_uwb.hrl

```
-module(pmod_uwb).
-behaviour(gen_server).

%% API
-export([start_link/2]).
-export([init/1, handle_call/3, handle_cast/2]).
-export([read/1, write/2, write_tx_data/1, get_received_data/0,
    transmit/1, transmit/2, wait_for_transmission/0, reception/0,
    reception/1]).
-export([set_frame_timeout/1]).
-export([softreset/0, clear_rx_flags/0]).

-compile({nowarn_unused_function, [debug_read/2, debug_write/2,
    debug_write/3, debug_bitstring/1, debug_bitstring_hex/1]}).
```

```
12
13 % Include for the record "device"
14 -include("grisp.hrl").
15
16 -include("pmod_uwb.hrl").
17
18 % Define the polarity and the phase of the clock
19 -define(SPI_MODE, #{clock => {low, leading}}).
20
21 -define(WRITE_ONLY_REG_FILE(RegFileID), RegFileID == tx_buffer).
22 -define(READ_ONLY_REG_FILE(RegFileID), RegFileID==dev_id;
       RegFileID==sys_time; RegFileID==rx_finfo; RegFileID==rx_buffer;
        RegFileID==rx_fqual; RegFileID==rx_ttcko;
23                                          RegFileID==rx_time;
       RegFileID==tx_time; RegFileID==sys_state; RegFileID==acc_mem).
24
25 % The congifurations of the subregisters of these register files
       are different (some sub-registers are RO, some are RW and some
       have reserved bytes that can't be written)
26 % Thus, some registers files require to write their sub-register
       independently => Write the sub-registers one by one instead of
       writting the whole register file directly
27 -define(IS_SRW(RegFileID), RegFileID==agc_ctrl; RegFileID==
       ext_sync; RegFileID==ec_ctrl; RegFileID==gpio_ctrl; RegFileID==
       drx_conf; RegFileID==rf_conf; RegFileID==tx_cal;
28                             RegFileID==fs_ctrl; RegFileID==aon;
       RegFileID==otp_if; RegFileID==lde_if; RegFileID==dig_diag;
       RegFileID==pmsc).
29
30 -define(READ_ONLY_SUB_REG(SubRegister), SubRegister==irqs;
       SubRegister==agc_stat1; SubRegister==ec_rxtc; SubRegister==
       ec_glop; SubRegister==drx_car_int;
31                                          SubRegister==rf_status;
       SubRegister==tc_sarl; SubRegister==sarw; SubRegister==
       tc_pg_status; SubRegister==lde_thresh;
32                                          SubRegister==lde_ppindx;
       SubRegister==lde_ppampl; SubRegister==evc_phe; SubRegister==
       evc_rse; SubRegister==evc_fcg;
33                                          SubRegister==evc_fce;
       SubRegister==evc_ffr; SubRegister==evc_ovr; SubRegister==
       evc_sto; SubRegister==evc_pto;
34                                          SubRegister==evc_fwto;
       SubRegister==evc_txfs; SubRegister==evc_hpw; SubRegister==
       evc_tpw).
35
36 -type regFileID() :: atom().
37
38
39 %--- API
```

```
    ------------------------------------------------------------------------


%% @private
start_link(Connector, _Opts) ->
    gen_server:start_link({local, ?MODULE}, ?MODULE, Connector, []
    ).


%%
    ------------------------------------------------------------------------

%% @doc read a register file
%%
%% === Example ===
%% To read the register file DEV_ID
%% ```
%% 1> pmod_uwb:read(dev_id).
%% #{model => 1,rev => 0,ridtag => "DECA",ver => 3}
%% '''
%% @end
%%
    ------------------------------------------------------------------------

-spec read(RegFileID :: regFileID()) -> map() | {error, any()}.
read(RegFileID) when ?WRITE_ONLY_REG_FILE(RegFileID) ->
    error({read_on_write_only_register, RegFileID});
read(RegFileID) -> call({read, RegFileID}).

%%
    ------------------------------------------------------------------------

%% @doc Write values in a register
%%
%% === Examples ===
%% To write in a simple register file (i.e. a register without any
    sub-register):
%% ```
%% 1> pmod_uwb:write(eui, #{eui => 16#AAAAAABBBBBBBBBB}).
%% ok
%% '''
%% To write in one sub-register of a register file:
%% ```
%% 2> pmod_uwb:write(panadr, #{pan_id => 16#AAAA}).
%% ok
%% '''
%% The previous code will only change the values inside the sub-
    register PAN_ID
%%
```

```
78 %% To write in multiple sub-register of a register file in the
       same burst:
79 %% ```
80 %% 3> pmod_uwb:write(panadr, #{pan_id => 16#AAAA, short_addr =>
       16#BBBB}).
81 %% ok
82 %% '''
83 %% Some sub-registers have their own fields. For example to set
       the value of the DIS_AM field in the sub-register AGC_CTRL1 of
       the register file AGC_CTRL:
84 %% ```
85 %% 4> pmod_uwb:write(agc_ctrl, #{agc_ctrl1 => #{dis_am => 2#0}}).
86 %% '''
87 %% @end
88 %%
       ------------------------------------------------------------------------

89 -spec write(RegFileID :: regFileID(), Value :: map()) -> ok | {
       error, any()}.
90 write(RegFileID, Value) when ?READ_ONLY_REG_FILE(RegFileID) ->
91     error({write_on_read_only_register, RegFileID, Value});
92 write(RegFileID, Value) when is_map(Value) ->
93     call({write, RegFileID, Value}).
94
95 %%
       ------------------------------------------------------------------------

96 %% @doc Writes the data in the TX_BUFFER register
97 %%
98 %% Value is expected to be a <b>Binary</b>
99 %% That choice was made to make the transmission of frames easier
       later on
100 %%
101 %% === Examples ===
102 %% Send "Hello" in the buffer
103 %% ```
104 %% 1> pmod_uwb:write_tx_data(<<"Hello">>).
105 %% '''
106 %% @end
107 %%
       ------------------------------------------------------------------------

108 -spec write_tx_data(Value :: binary()) -> ok | {error, any()}.
109 write_tx_data(Value) -> call({write_tx, Value}).
110
111 %%
       ------------------------------------------------------------------------

112 %% @doc Retrieves the data received on the UWB antenna
```

```
113 %% @returns {DataLength, Data}
114 %% @end
115 %%
    ------------------------------------------------------------------------

116 -spec get_received_data() -> {integer(), bitstring()} | {error,
    any()}.
117 get_received_data() -> call({get_rx_data}).
118
119 %%
    ------------------------------------------------------------------------

120 %% @doc Transmit data with the default options (i.e. don't wait
    for resp, no delayn ...)
121 %%
122 %% === Examples ===
123 %% To transmit a frame:
124 %% ```
125 %% 1> pmod_uwb:transmit(<Version:4, NextHop:8>>).
126 %% ok.
127 %% '''
128 %% @end
129 %%
    ------------------------------------------------------------------------

130 -spec transmit(Data :: bitstring()) -> ok | {error, any()}.
131 transmit(Data) when is_bitstring(Data) ->
132     call({transmit, Data, #tx_opts{}}),
133     wait_for_transmission().
134
135 %%
    ------------------------------------------------------------------------

136 %% @doc Performs a transmission with the specified options
137 %%
138 %% === Options ===
139 %% * wait4resp: It specifies that the reception must be enabled
    after the transmission in the expectation of a response
140 %% * w4r-tim: Specifies the turn around time in microseconds. That
     is the time the pmod will wait before enabling rx after a tx.
    Note that it won't be set if wit4resp is disabled
141 %% * txdlys: Specifies if the transmitter delayed sending should
    be set
142 %% * tx_delay: Specifies the delay of the transmission (see
    register DX_TIME)
143 %%
144 %% === Examples ===
145 %% To transmit a frame with default options:
146 %% ```
```

```
147 %% 1> pmod_uwb:transmit(<Version:4, NextHop:8>>, #tx_opts{}).
148 %% ok.
149 %% '''
150 %% @end
151 %%
    ------------------------------------------------------------------------------

152 transmit(Data, Options) ->
153     case Options#tx_opts.wait4resp of
154         ?ENABLED -> clear_rx_flags();
155         _ -> ok
156     end,
157     call({transmit, Data, Options}),
158     case read(sys_status) of
159         #{hdpwarn := 2#1} -> error({hdpwarn});
160         _ -> ok
161     end,
162     wait_for_transmission().
163
164 %% Wait for the transmission to be performed
165 %% usefull in the case of a delayed transmission
166 wait_for_transmission() ->
167     case read(sys_status) of
168         #{txfrs := 1} -> ok;
169         _ -> wait_for_transmission()
170     end.
171
172 %%
    ------------------------------------------------------------------------------

173 %% @doc Receive data using the pmod
174 %% @equiv reception(false)
175 %%
176 %% @end
177 %%
    ------------------------------------------------------------------------------

178 -spec reception() -> {integer(), bitstring()} | {error, any()}.
179 reception() ->
180     reception(false).
181
182 %%
    ------------------------------------------------------------------------------

183 %% @doc Receive data using the pmod
184 %%
185 %% The function will hang until a frame is received on the board
186 %%
187 %% The CRC of the received frame <b>isn't</b> included in the
```

```
    returned value
%%
%% @param RXEnabled: specifies if the reception is already enabled
     on the board (or set with delay)
%%
%% === Example ===
%% ```
%% 1> pmod_uwb:reception().
%% % Some frame is transmitted
%% {11, <<"Hello world">>}.
%% '''
%% @end
%%
    ------------------------------------------------------------------------

-spec reception(RXEnabled :: boolean()) -> {integer(), bitstring()
    } | {error, any()}.
reception(RXEnabled) ->
    if not RXEnabled -> enable_rx();
       true -> ok
    end,
    case wait_for_reception() of
        ok -> % write(sys_status, #{rxfcg => 1}),
              get_received_data();
        Err -> Err
    end.


%% @private
enable_rx() ->
    % io:format("Enabling reception~n"),
    clear_rx_flags(),
    call({write, sys_ctrl, #{rxenab => 2#1}}).

wait_for_reception() ->
    % io:format("Wait for resp~n"),
    case read(sys_status) of
        #{rxrfto := 1} -> rxrfto;
        #{rxphe := 1} -> rxphe;
        #{rxfce := 1} -> rxfce;
        #{rxrfsl := 1} -> rxrfsl;
        #{rxpto := 1} -> rxpto;
        #{rxsfdto := 1} -> rxsfdto;
        #{ldeerr := 1} -> ldeerr;
        #{affrej := 1} -> affrej;
        #{rxdfr := 0} -> wait_for_reception();
        #{rxfce := 1} -> rxfce;
        #{rxfcg := 1} -> ok;
        #{rxfcg := 0} -> wait_for_reception();
```

```
232         % #{rxdfr := 1, rxfcg := 1} -> ok; % The example driver
    doesn't do that but the user manual says that how you should
    check the reception of a frame
233         _ -> error({error_wait_for_reception})
234     end.
235
236 %%
    ------------------------------------------------------------------------

237 %% @doc Set the frame wait timeout and enables it
238 %% @end
239 %%
    ------------------------------------------------------------------------

240 -spec set_frame_timeout(Timeout :: miliseconds()) -> ok.
241 set_frame_timeout(Timeout) ->
242     write(rx_fwto, #{rxfwto => Timeout}),
243     write(sys_cfg, #{rxwtoe => 2#1}). % enable receive wait
    timeout
244
245 %%
    ------------------------------------------------------------------------

246 %% @doc Performs a reset of the IC following the procedure
    described in section 7.2.50.1
247 %%
248 %% @end
249 %%
    ------------------------------------------------------------------------

250 softreset() ->
251     write(pmsc, #{pmsc_ctrl0 => #{sysclks => 2#01}}),
252     write(pmsc, #{pmsc_ctrl0 => #{softrest => 16#0}}),
253     write(pmsc, #{pmsc_ctrl0 => #{softreset => 16#FFFF}}).
254
255
256 clear_rx_flags() ->
257     write(sys_status, #{rxsfdto => 2#1, rxpto => 2#1, rxrfto =>
    2#1, rxrfsl => 2#1, rxfce => 2#1, rxphe => 2#1, rxprd => 2#1,
    rxdsfdd => 2#1, rxphd => 2#1, rxdfr => 2#1, rxfcg => 2#1}).
258
259 %--- Callbacks
    -----------------------------------------------------------------


260
261 %% @private
262 init(Slot) ->
263     % Verify the slot used
264     case {grisp_hw:platform(), Slot} of
```

```
265         {grisp2, spi2} -> ok;
266         {P, S} -> error({incompatible_slot, P, S})
267     end,
268     grisp_devices:register(Slot, ?MODULE),
269     Bus = grisp_spi:open(Slot),
270     case verify_id(Bus) of
271         ok -> softreset(Bus);
272         Val -> error({dev_id_no_match, Val})
273     end,
274     ldeload(Bus),
275     write_default_values(Bus),
276     config(Bus),
277     setup_sfd(Bus),
278     {ok, #{bus => Bus}}.
279
280 %% @private
281 handle_call({read, RegFileID}, _From, #{bus := Bus} = State) -> {
    reply, read_reg(Bus, RegFileID), State};
282 handle_call({write, RegFileID, Value}, _From, #{bus := Bus} =
    State) -> {reply, write_reg(Bus, RegFileID, Value), State};
283 handle_call({write_tx, Value}, _From, #{bus := Bus} = State) -> {
    reply, write_tx_data(Bus, Value), State};
284 handle_call({transmit, Data, Options}, _From, #{bus := Bus} =
    State) -> {reply, tx(Bus, Data, Options), State};
285 handle_call({delayed_transmit, Data, Delay}, _From, #{bus := Bus}
    = State) -> {reply, delayed_tx(Bus, Data, Delay), State};
286 handle_call({get_rx_data}, _From, #{bus := Bus} = State) -> {reply
    , get_rx_data(Bus), State};
287 handle_call(Request, _From, _State) -> error({unknown_call,
    Request}).
288
289 %% @private
290 handle_cast(Request, _State) -> error({unknown_cast, Request}).
291
292 %--- Internal
    -----------------------------------------------------------------


293
294 call(Call) ->
295     Dev = grisp_devices:default(?MODULE),
296     gen_server:call(Dev#device.pid, Call).
297
298
299 %%
    ------------------------------------------------------------------------------


300 %% @doc Varify the dev_id register of the pmod
301 %% @returns ok if the value is correct, otherwise the value read
302 %%
```

```
303 verify_id(Bus) ->
304     #{ridtag := RIDTAG, model := MODEL} = read_reg(Bus, dev_id),
305     case {RIDTAG, MODEL} of
306         {"DECA", 1} -> ok;
307         _ -> {RIDTAG, MODEL}
308     end.
309
310 %%
    -------------------------------------------------------------------------

311 %% @private
312 %% Performs a softreset on the pmod
313 %%
    -------------------------------------------------------------------------

314 softreset(Bus) ->
315     write_reg(Bus, pmsc, #{pmsc_ctrl0 => #{sysclks => 2#01}}),
316     write_reg(Bus, pmsc, #{pmsc_ctrl0 => #{softrest => 16#0}}),
317     write_reg(Bus, pmsc, #{pmsc_ctrl0 => #{softreset => 16#FFFF}})
    .
318
319 %%
    -------------------------------------------------------------------------

320 %% @private
321 %% Writes the default values described in section 2.5.5 of the
    user manual
322 %%
    -------------------------------------------------------------------------

323 write_default_values(Bus) ->
324     write_reg(Bus, lde_if, #{lde_cfg1 => #{ntm => 16#D}, lde_cfg2
    => 16#1607}),
325     write_reg(Bus, agc_ctrl, #{agc_tune1 => 16#8870, agc_tune2 =>
    16#2502A907}),
326     write_reg(Bus, drx_conf, #{drx_tune2 => 16#311A002D}),
327     write_reg(Bus, tx_power, #{tx_power => 16#0E082848}),
328     write_reg(Bus, rf_conf, #{rf_txctrl => 16#001E3FE3}),
329     write_reg(Bus, tx_cal, #{tc_pgdelay => 16#B5}),
330     write_reg(Bus, fs_ctrl, #{fs_plltune => 16#BE}).
331
332 %%
    -------------------------------------------------------------------------

333 %% @private
334 %%
    -------------------------------------------------------------------------
```

```
335 config(Bus) ->
336     write_reg(Bus, ext_sync, #{ec_ctrl => #{plldt => 2#1}}),
337     %write_reg(Bus, pmsc, #{pmsc_ctrl1 => #{lderune => 2#0}}),
338     % Now enable RX and TX leds
339     write_reg(Bus, gpio_ctrl, #{gpio_mode => #{msgp2 => 2#01,
    msgp3 => 2#01}}),
340     % Enable RXOK and SFD leds
341     write_reg(Bus, gpio_ctrl, #{gpio_mode => #{msgp0 => 2#01,
    msgp1 => 2#01}}),
342     write_reg(Bus, pmsc, #{pmsc_ctrl0 => #{gpdce => 2#1, khzclken
    => 2#1}}),
343     write_reg(Bus, pmsc, #{pmsc_ledc => #{blnken => 2#1}}),
344     write_reg(Bus, dig_diag, #{evc_ctrl => #{evc_en => 2#1}}), %
    enable counting event for debug purposes
345     % write_reg(Bus, sys_cfg, #{rxwtoe => 2#1}),
346     write_reg(Bus, tx_fctrl, #{txpsr => 2#10}). % Setting preamble
     symbols to 1024
347
348
349 %%
    ------------------------------------------------------------------------

350 %% @private
351 %% Load the microcode from ROM to RAM
352 %% It follows the steps described in section 2.5.5.10 of the
    DW1000 user manual
353 %%
    ------------------------------------------------------------------------

354 ldeload(Bus) ->
355     write_reg(Bus, pmsc, #{pmsc_ctrl0 => #{sysclks => 2#01}}),
356     write_reg(Bus, pmsc, #{pmsc_ctrl0 => #{otp => 2#1, res8 => 2#1
    }}), % Writes 0x0301 in pmsc_ctrl0
357     write_reg(Bus, otp_if, #{otp_ctrl => #{ldeload => 2#1}}), %
    Writes 0x8000 in OTP_CTRL
358     timer:sleep(150), % User manual requires a wait of 150 s
359     write_reg(Bus, pmsc, #{pmsc_ctrl0 => #{sysclks => 2#0}}), %
    Writes 0x0200 in pmsc_ctrl0
360     write_reg(Bus, pmsc, #{pmsc_ctrl0 => #{res8 => 2#0}}).
361
362 %%
    ------------------------------------------------------------------------

363 %% @private
364 %% If no frame is transmitted before AUTOACK, then the SFD isn't
    properly set
365 %% (cf. section 5.3.1.2 SFD initialisation)
366 %%
```

```
    %%--------------------------------------------------------------------

setup_sfd(Bus) ->
    write_reg(Bus, sys_ctrl, #{txstrt => 2#1, trxoff => 2#1}).

%%
    %%--------------------------------------------------------------------

%% @private
%% Transmit the data using UWB
%% @param Options is used to set options about the transmission
    like a transmission delay, etc.
%%
    %%--------------------------------------------------------------------

-spec tx(_, Data :: bitstring(), Options :: #tx_opts{}) -> ok.
tx(Bus, Data, #tx_opts{wait4resp = Wait4resp, w4r_tim = W4rTim,
    txdlys = TxDlys, tx_delay = TxDelay}) ->
    % Writing the data that will be sent (w/o CRC)
    DataLength = byte_size(Data) + 2, % DW1000 automatically adds
    the 2 bytes CRC
    write_tx_data(Bus, Data),
    % Setting the options of the transmission
    case Wait4resp of
        ?ENABLED -> write_reg(Bus, ack_resp_t, #{w4r_tim => W4rTim
    });
        _ -> ok
    end,
    case TxDlys of
        ?ENABLED -> write_reg(Bus, dx_time, #{dx_time => TxDelay})
    ;
        _ -> ok
    end,
    write_reg(Bus, tx_fctrl, #{txboffs => 2#0, tr => 2#0, tflen =>
    DataLength}),
    write_reg(Bus, sys_ctrl, #{txstrt => 2#1, wait4resp =>
    Wait4resp, txdlys => TxDlys}). % start transmission and some
    options

%%
    %%--------------------------------------------------------------------

%% @private
%% Transmit the data with a specified delay using UWB
%%
    %%--------------------------------------------------------------------

delayed_tx(Bus, Data, Delay) ->
    write_reg(Bus, dx_time, #{dx_time => Delay}),
```

```
398     DataLength = byte_size(Data) + 2, % DW1000 automatically adds
    the 2 bytes CRC
399     write_tx_data(Bus, Data),
400     write_reg(Bus, tx_fctrl, #{txboffs => 2#0, tflen => DataLength
    }),
401     write_reg(Bus, sys_ctrl, #{txstrt => 2#1, txdlys => 2#1}). %
    start transmission
402
403 %%
    --------------------------------------------------------------------

404 %% @private
405 %% Get the received data (without the CRC bytes) stored in the
    rx_buffer
406 %%
    --------------------------------------------------------------------

407 get_rx_data(Bus) ->
408     #{rxflen := FrameLength} = read_reg(Bus, rx_finfo),
409     Frame = read_rx_data(Bus, FrameLength-2), % Remove the CRC
    bytes
410     {FrameLength, Frame}.
411
412 %%
    --------------------------------------------------------------------

413 %% @private
414 %% @doc Reverse the byte order of the bitstring given in the
    argument
415 %% @param Bin a bitstring
416 %%
    --------------------------------------------------------------------

417 reverse(Bin) -> reverse(Bin, <<>>).
418 reverse(<<Bin:8>>, Acc) ->
419     <<Bin, Acc/binary>>;
420 reverse(<<Bin:8, Rest/bitstring>>, Acc) ->
421     reverse(Rest, <<Bin, Acc/binary>>).
422
423 %%
    --------------------------------------------------------------------

424 %% @private
425 %% @doc Creates the header of the SPI transaction between the
    GRiSP and the pmod
426 %%
427 %%  It creates a header of 1 bytes. The header is used in a
    transaction that will affect
428 %%  the whole register file (read/write)
```

```
429 %%
430 %% @param Op an atom (either <i>read</i> or <i>write</i>
431 %% @param RegFileID an atom representing the register file
432 %% @returns a formated header of <b>1 byte</b> long as described
    in the user manual
433 %%
    ------------------------------------------------------------------------

434 header(Op, RegFileID) ->
435     <<(rw(Op)):1, 2#0:1, (regFile(RegFileID)):6>>.
436
437 %%
    ------------------------------------------------------------------------

438 %% @private
439 %% @doc Creates the header of the SPI transaction between the
    GRiSP and the pmod
440 %%
441 %%  It creates a header of 2 bytes. The header is used in a
    transaction that will affect
442 %%  the whole sub-register (read/write)
443 %%  Careful: The sub-register needs to be mapped in the hrl file
444 %%
445 %% @param Op an atom (either <i>read</i> or <i>write</i>
446 %% @param RegFileID an atom representing the register file
447 %% @param SubRegister an atom representing the sub-register
448 %% @returns a formated header of <b>2 byte</b> long as described
    in the user manual
449 %%
    ------------------------------------------------------------------------

450 header(Op, RegFileID, SubRegister) ->
451     case subReg(SubRegister) < 127 of
452         true -> header(Op, RegFileID, SubRegister, 2);
453         _ -> header(Op, RegFileID, SubRegister, 3)
454     end.
455
456 header(Op, RegFileID, SubRegister, 2) ->
457     << (rw(Op)):1, 2#1:1, (regFile(RegFileID)):6,
458         2#0:1, (subReg(SubRegister)):7 >>;
459 header(Op, RegFileID, SubRegister, 3) ->
460     <<_:1, HighOrder:8, LowOrder:7>> = <<(subReg(SubRegister))
    :16>>,
461     << (rw(Op)):1, 2#1:1, (regFile(RegFileID)):6,
462        2#1:1, LowOrder:7,
463        HighOrder:8>>.
464
465 %%
    ------------------------------------------------------------------------
```

```
%% @private
%% @doc Read the values stored in a register file
%% ----------------------------------------------------------------------------

read_reg(Bus, lde_ctrl) -> read_reg(Bus, lde_if);
read_reg(Bus, lde_if) ->
    lists:foldl(fun(Elem, Acc) ->
                    Res = read_sub_reg(Bus, lde_if, Elem),
                    maps:merge(Acc, Res)
                end,
                #{},
                [lde_thresh, lde_cfg1, lde_ppindx, lde_ppampl,
    lde_rxantd, lde_cfg2, lde_repc]);
read_reg(Bus, RegFileID) ->
    Header = header(read, RegFileID),
    [Resp] = grisp_spi:transfer(Bus, [{?SPI_MODE, Header, 1,
    regSize(RegFileID)}]),
    % debug_read(RegFileID, Resp),
    reg(decode, RegFileID, Resp).


read_sub_reg(Bus, RegFileID, SubRegister) ->
    Header = header(read, RegFileID, SubRegister),
    HeaderSize = byte_size(Header),
    % io:format("[HEADER] type ~w - ~w - ~w~n", [HeaderSize,
    Header, subRegSize(SubRegister)]),
    [Resp] = grisp_spi:transfer(Bus, [{?SPI_MODE, Header,
    HeaderSize, subRegSize(SubRegister)}]),
    reg(decode, SubRegister, Resp).


%% ----------------------------------------------------------------------------

%% @doc get the received data
%% @param Length is the total length of the data we are trying to
    read
%% ----------------------------------------------------------------------------

read_rx_data(Bus, Length) ->
    Header = header(read, rx_buffer),
    [Resp] = grisp_spi:transfer(Bus, [{?SPI_MODE, Header, 1,
    Length}]),
    Resp.

% TODO: check that user isn't trying to write reserved bits by
```

```
    passing res, res1, ... in the map fields
%%
    ------------------------------------------------------------------------------

%% @doc used to write the values in the map given in the Value
    argument
%%
    ------------------------------------------------------------------------------

-spec write_reg(Bus::map(), RegFileID::regFileID(), Value::map())
    -> ok | {error, any()}.
% Write each sub-register one by one.
% If the user tries to write in a read-only sub-register, an error
     is thrown
write_reg(Bus, RegFileID, Value) when ?IS_SRW(RegFileID) ->
    maps:map(
        fun(SubRegister, Val) ->
            CurrVal = maps:get(SubRegister, read_reg(Bus,
    RegFileID)), % ? can the read be done before ? Maybe but not
    assured that no values changes after a write in the register
            Body = case CurrVal of
                       V when is_map(V) -> reg(encode,
    SubRegister, maps:merge_with(fun(_Key, _Old, New) -> New end,
    CurrVal, Val));
                       _ -> reg(encode, SubRegister, #{
    SubRegister => Val})
                   end,
            Header = header(write, RegFileID, SubRegister),
            % debug_write(RegFileID, SubRegister, Body),
            _ = grisp_spi:transfer(Bus, [{?SPI_MODE, <<Header/
    binary, Body/binary>>, 2+subRegSize(SubRegister), 0}])
        end,
        Value),
    ok;
write_reg(Bus, RegFileID, Value) ->
    Header = header(write, RegFileID),
    CurrVal = read_reg(Bus, RegFileID),
    ValuesToWrite = maps:merge_with(fun(_Key, _Value1, Value2) ->
    Value2 end, CurrVal, Value),
    Body = reg(encode, RegFileID, ValuesToWrite),
    % debug_write(RegFileID, Body),
    _ = grisp_spi:transfer(Bus, [{?SPI_MODE, <<Header/binary, Body
    /binary>>, 1+regSize(RegFileID), 0}]),
    ok.

%%
    ------------------------------------------------------------------------------

%% @doc write_tx_data/2 sends data (Value) in the register
```

```
    tx_buffer
%% @param Value is the data to be written. It must be a binary and
    have a size of maximum 1024 bits
%%
    ------------------------------------------------------------------------

write_tx_data(Bus, Value) when is_binary(Value), (bit_size(Value)
    < 1025) ->
    Header = header(write, tx_buffer),
    Length = byte_size(Value),
    % debug_write(tx_buffer, Body),
    _ = grisp_spi:transfer(Bus, [{?SPI_MODE, <<Header/binary,
    Value/binary>>, 1+Length, 0}]),
    ok.

%% ----- Register mapping
    ------------------------------------------------------------------

%%
    ------------------------------------------------------------------------

%% @doc Used to either decode the data returned by the pmod or to
    encode to data that will be sent to the pmod
%%
%% The transmission on the MISO line is done byte by byte starting
    from the lowest rank byte to the highest rank
%% Example: dev_id value is 0xDECA0130 but 0x3001CADE is
    transmitted over the MISO line
%%
    ------------------------------------------------------------------------

reg(encode, SubRegister, Value) when ?READ_ONLY_SUB_REG(
    SubRegister) -> error({writing_read_only_sub_register,
    SubRegister, Value});
reg(decode, dev_id, Resp) ->
    <<
      RIDTAG:16, Model:8, Ver:4, Rev:4
    >> = reverse(Resp),
    #{
        ridtag => integer_to_list(RIDTAG, 16), model => Model, ver
    => Ver, rev => Rev
    };
reg(decode, eui, Resp) ->
    #{
        eui => reverse(Resp)
    };
reg(encode, eui, Val) ->
    #{
        eui:= EUI
```

```
564     } = Val,
565     reverse(<<
566         EUI:64
567     >>);
568 reg(decode, panadr, Resp) ->
569     <<
570         PanId:16, ShortAddr:16
571     >> = reverse(Resp),
572     #{
573         pan_id => PanId, short_addr => ShortAddr
574     };
575 reg(encode, panadr, Val) ->
576     #{
577         pan_id := PanId, short_addr := ShortAddr
578     } = Val,
579     reverse(<<
580         PanId:16, ShortAddr:16
581     >>);
582 reg(decode, sys_cfg, Resp) ->
583     <<
584         FFA4:1, FFAR:1, FFAM:1, FFAA:1, FFAD:1, FFAB:1, FFBC:1,
    FFEN:1, % bits 7-0
585         FCS_INIT2F:1, DIS_RSDE:1, DIS_PHE:1, DIS_DRXB:1, DIS_FCE
    :1, SPI_EDGE:1, HIRQ_POL:1, FFA5:1, % bits 15-8
586         _:1, RXM110K:1, _:3, DIS_STXP:1, PHR_MODE:2, % bits 23-16
587         AACKPEND:1, AUTOACK:1, RXAUTR:1, RXWTOE:1, _:4 % bits
    31-24
588     >> = Resp,
589     #{
590         aackpend => AACKPEND, autoack => AUTOACK, rxautr => RXAUTR
    , rxwtoe => RXWTOE,
591         rxm110k => RXM110K, dis_stxp => DIS_STXP, phr_mode =>
    PHR_MODE,
592         fcs_init2F => FCS_INIT2F, dis_rsde => DIS_RSDE, dis_phe =>
     DIS_PHE, dis_drxb => DIS_DRXB, dis_fce => DIS_FCE, spi_edge =>
     SPI_EDGE, hirq_pol => HIRQ_POL, ffa5 => FFA5,
593         ffa4 => FFA4, ffar => FFAR, ffam => FFAM, ffaa => FFAA,
    ffad => FFAD, ffab => FFAB, ffbc => FFBC, ffen => FFEN
594     };
595 reg(encode, sys_cfg, Val) ->
596     #{
597         aackpend := AACKPEND, autoack := AUTOACK, rxautr := RXAUTR
    , rxwtoe := RXWTOE,
598         rxm110k := RXM110K, dis_stxp := DIS_STXP, phr_mode :=
    PHR_MODE,
599         fcs_init2F := FCS_INIT2F, dis_rsde := DIS_RSDE, dis_phe :=
     DIS_PHE, dis_drxb := DIS_DRXB, dis_fce := DIS_FCE, spi_edge :=
     SPI_EDGE, hirq_pol := HIRQ_POL, ffa5 := FFA5,
600         ffa4 := FFA4, ffar := FFAR, ffam := FFAM, ffaa := FFAA,
```

```
    ffad := FFAD, ffab := FFAB, ffbc := FFBC, ffen := FFEN
    } = Val,
    <<
        FFA4:1, FFAR:1, FFAM:1, FFAA:1, FFAD:1, FFAB:1, FFBC:1,
    FFEN:1, % bits 7-0
        FCS_INIT2F:1, DIS_RSDE:1, DIS_PHE:1, DIS_DRXB:1, DIS_FCE
    :1, SPI_EDGE:1, HIRQ_POL:1, FFA5:1, % bits 15-8
        2#0:1, RXM110K:1, 2#0:3, DIS_STXP:1, PHR_MODE:2, % bits
    23-16
        AACKPEND:1, AUTOACK:1, RXAUTR:1, RXWTOE:1, 2#0:4 % bits
    31-24
    >>;
reg(decode, sys_time, Resp) ->
    <<
        SysTime:40
    >> = reverse(Resp),
    #{
        sys_time => SysTime
    };
reg(decode, tx_fctrl, Resp) ->
    <<
        IFSDELAY:8, TXBOFFS:10, PE:2, TXPSR:2, TXPRF:2, TR:1, TXBR
    :2, R:3, TFLE:3, TFLEN:7
    >> = reverse(Resp),
    #{
        ifsdelay => IFSDELAY, txboffs => TXBOFFS, pe => PE, txpsr
    => TXPSR, txprf => TXPRF, tr => TR, txbr => TXBR, r => R, tfle
    => TFLE, tflen => TFLEN
    };
reg(encode, tx_fctrl, Val) ->
    #{
        ifsdelay := IFSDELAY, txboffs := TXBOFFS, pe := PE, txpsr
    := TXPSR, txprf := TXPRF, tr := TR, txbr := TXBR, r := R, tfle
    := TFLE, tflen := TFLEN
    } = Val,
    reverse(<<
        IFSDELAY:8, TXBOFFS:10, PE:2, TXPSR:2, TXPRF:2, TR:1, TXBR
    :2, R:3, TFLE:3, TFLEN:7
    >>);
% TX_BUFFER is write only => no decode
reg(decode, dx_time, Resp) ->
    #{
        dx_time => reverse(Resp)
    };
reg(encode, dx_time, Val) ->
    #{
        dx_time := DX_TIME
    } = Val,
    reverse(<<
```

```
639         DX_TIME:40
640     >>);
641 reg(decode, rx_fwto, Resp) ->
642     <<
643         RXFWTO:16
644     >> = reverse(Resp),
645     #{
646         rxfwto => RXFWTO
647     };
648 reg(encode, rx_fwto, Val) ->
649     #{
650         rxfwto := RXFWTO
651     } = Val,
652     reverse(<<
653         RXFWTO:16
654     >>);
655 reg(decode, sys_ctrl, Resp) ->
656     <<
657         WAIT4RESP:1, TRXOFF:1, _:2, CANSFCS:1, TXDLYS:1, TXSTRT:1,
    SFCST:1, % bits 7-0
658         _:6, RXDLYE:1, RXENAB:1, % bits 15-8
659         _:8, % bits 23-16
660         _:7, HRBPT:1 % bits 31-24
661     >> = Resp,
662     #{
663         sfcst => SFCST, txstrt => TXSTRT, txdlys => TXDLYS,
   cansfcs => CANSFCS, trxoff => TRXOFF, wait4resp => WAIT4RESP,
664         rxenab => RXENAB, rxdlye => RXDLYE,
665         hrbpt => HRBPT
666     };
667 reg(encode, sys_ctrl, Val) ->
668     #{
669         sfcst := SFCST, txstrt := TXSTRT, txdlys := TXDLYS,
   cansfcs := CANSFCS, trxoff := TRXOFF, wait4resp := WAIT4RESP,
670         rxenab := RXENAB, rxdlye := RXDLYE,
671         hrbpt := HRBPT
672     } = Val,
673     <<
674         WAIT4RESP:1, TRXOFF:1, 2#0:2, CANSFCS:1, TXDLYS:1, TXSTRT
   :1, SFCST:1, % bits 7-0
675         2#0:6, RXDLYE:1, RXENAB:1, % bits 15-8
676         2#0:8, % bits 23-16
677         2#0:7, HRBPT:1 % bits 31-24
678     >>;
679 reg(decode, sys_mask, Resp) ->
680     <<
681         MTXFRS:1, MTXPHS:1, MTXPRS:1, MTXFRB:1, MAAT:1, MESYNCR:1,
    MCPLOCK:1, Reserved0:1, % bits 7-0
682         MRXFCE:1, MRXFCG:1, MRXDFR:1, MRXPHE:1, MRXPHD:1, MLDEDON
```

```
    :1, MRXSFDD:1, MRXPRD:1, % bits 15-8
683         MSLP2INIT:1, MGPIOIRQ:1, MRXPTO:1, MRXOVRR:1, Reserved1:1,
     MLDEERR:1, MRXRFTO:1, MRXRFSL:1, % bits 23-16
684         Reserved2:2, MAFFREJ:1, MTXBERR:1, MHPDDWAR:1, MPLLHILO:1,
     MCPLLLL:1, MRFPLLLL:1 % bits 31-24
685     >> = Resp,
686     #{
687         mtxfrs => MTXFRS, mtxphs => MTXPHS, mtxprs => MTXPRS,
    mtxfrb => MTXFRB, maat => MAAT, mesyncr => MESYNCR, mcplock =>
    MCPLOCK, res0 => Reserved0, % bits 7-0
688         mrxfce => MRXFCE, mrxfcg => MRXFCG, mrxdfr => MRXDFR,
    mrxphe => MRXPHE, mrxphd => MRXPHD, mldeon => MLDEDON, mrxsfdd
    => MRXSFDD, mrxprd => MRXPRD, % bits 15-8
689         mslp2init => MSLP2INIT, mgpioirq => MGPIOIRQ, mrxpto =>
    MRXPTO, mrxovrr => MRXOVRR, res1 => Reserved1, mldeerr =>
    MLDEERR, mrxrfto => MRXRFTO, mrxrfsl => MRXRFSL, % bits 23-16
690         res2 => Reserved2, maffrej => MAFFREJ, mtxberr => MTXBERR,
     mhpddwar => MHPDDWAR, mpllhilo => MPLLHILO, mcpllll => MCPLLLL
    , mrfpllll => MRFPLLLL % bits 31-24
691     };
692 reg(encode, sys_mask, Val) ->
693     #{
694         mtxfrs := MTXFRS, mtxphs := MTXPHS, mtxprs := MTXPRS,
    mtxfrb := MTXFRB, maat := MAAT, mesyncr := MESYNCR, mcplock :=
    MCPLOCK, res0 := Reserved0, % bits 7-0
695         mrxfce := MRXFCE, mrxfcg := MRXFCG, mrxdfr := MRXDFR,
    mrxphe := MRXPHE, mrxphd := MRXPHD, mldeon := MLDEDON, mrxsfdd
    := MRXSFDD, mrxprd := MRXPRD, % bits 15-8
696         mslp2init := MSLP2INIT, mgpioirq := MGPIOIRQ, mrxpto :=
    MRXPTO, mrxovrr := MRXOVRR, res1 := Reserved1, mldeerr :=
    MLDEERR, mrxrfto := MRXRFTO, mrxrfsl := MRXRFSL, % bits 23-16
697         res2 := Reserved2, maffrej := MAFFREJ, mtxberr := MTXBERR,
     mhpddwar := MHPDDWAR, mpllhilo := MPLLHILO, mcpllll := MCPLLLL
    , mrfpllll := MRFPLLLL % bits 31-24
698     } = Val,
699     <<
700         MTXFRS:1, MTXPHS:1, MTXPRS:1, MTXFRB:1, MAAT:1, MESYNCR:1,
     MCPLOCK:1, Reserved0:1, % bits 7-0
701         MRXFCE:1, MRXFCG:1, MRXDFR:1, MRXPHE:1, MRXPHD:1, MLDEDON
    :1, MRXSFDD:1, MRXPRD:1, % bits 15-8
702         MSLP2INIT:1, MGPIOIRQ:1, MRXPTO:1, MRXOVRR:1, Reserved1:1,
     MLDEERR:1, MRXRFTO:1, MRXRFSL:1, % bits 23-16
703         Reserved2:2, MAFFREJ:1, MTXBERR:1, MHPDDWAR:1, MPLLHILO:1,
     MCPLLLL:1, MRFPLLLL:1 % bits 31-24
704     >>;
705 reg(decode, sys_status, Resp) ->
706     <<
707         TXFRS:1, TXPHS:1, TXPRS:1, TXFRB:1, AAT:1, ESYNCR:1,
    CPLOCK:1, IRQS:1, % bits 7-0
```

```
708         RXFCE:1, RXFCG:1, RXDFR:1, RXPHE:1, RXPHD:1, LDEDONE:1,
    RXSFDD:1, RXPRD:1, % bits 15-8
709         SPL2INIT:1, GPIOIRQ:1, RXPTO:1, RXOVRR:1, Reserved0:1,
    LDEERR:1, RXRFTO:1, RXRFSL:1, % bits 23-16
710         ICRBP:1, HSRBP:1, AFFREJ:1, TXBERR:1, HPDWARN:1, RXSFDTO
    :1, CLCKPLL_LL:1, RFPLL_LL:1, % bits 31-24
711         Reserved1:5, TXPUTE:1, RXPREJ:1, RXRSCS:1 % bits 39-32
712     >> = Resp,
713     #{
714         txfrs => TXFRS, txphs => TXPHS, txprs => TXPRS, txfrb =>
    TXFRB, aat => AAT, esyncr => ESYNCR, cplock => CPLOCK, irqs =>
    IRQS, % bits 7-0
715         rxfce => RXFCE, rxfcg => RXFCG, rxdfr => RXDFR, rxphe =>
    RXPHE, rxphd => RXPHD, ldedone => LDEDONE, rxsfdd => RXSFDD,
    rxprd => RXPRD, % bits 15-8
716         splt2init => SPL2INIT, gpioirq => GPIOIRQ, rxpto => RXPTO,
     rxovrr => RXOVRR, res0 => Reserved0, ldeerr => LDEERR, rxrfto
    => RXRFTO, rxrfsl => RXRFSL, % bits 23-16
717         icrbp => ICRBP, hsrbp => HSRBP, affrej => AFFREJ, txberr
    => TXBERR, hdpwarn => HPDWARN, rxsfdto => RXSFDTO, clkpll_ll =>
     CLCKPLL_LL, rfpll_ll => RFPLL_LL, % bits 31-24
718         res1 => Reserved1, txpute => TXPUTE, rxprej => RXPREJ,
    rxrscs => RXRSCS
719     };
720 reg(encode, sys_status, Val) ->
721     #{
722         txfrs := TXFRS, txphs := TXPHS, txprs := TXPRS, txfrb :=
    TXFRB, aat := AAT, esyncr := ESYNCR, cplock := CPLOCK, irqs :=
    IRQS, % bits 7-0
723         rxfce := RXFCE, rxfcg := RXFCG, rxdfr := RXDFR, rxphe :=
    RXPHE, rxphd := RXPHD, ldedone := LDEDONE, rxsfdd := RXSFDD,
    rxprd := RXPRD, % bits 15-8
724         splt2init := SPL2INIT, gpioirq := GPIOIRQ, rxpto := RXPTO,
     rxovrr := RXOVRR, res0 := Reserved0, ldeerr := LDEERR, rxrfto
    := RXRFTO, rxrfsl := RXRFSL, % bits 23-16
725         icrbp := ICRBP, hsrbp := HSRBP, affrej := AFFREJ, txberr
    := TXBERR, hdpwarn := HPDWARN, rxsfdto := RXSFDTO, clkpll_ll :=
     CLCKPLL_LL, rfpll_ll := RFPLL_LL, % bits 31-24
726         res1 := Reserved1, txpute := TXPUTE, rxprej := RXPREJ,
    rxrscs := RXRSCS
727     } = Val,
728     <<
729         TXFRS:1, TXPHS:1, TXPRS:1, TXFRB:1, AAT:1, ESYNCR:1,
    CPLOCK:1, IRQS:1, % bits 7-0
730         RXFCE:1, RXFCG:1, RXDFR:1, RXPHE:1, RXPHD:1, LDEDONE:1,
    RXSFDD:1, RXPRD:1, % bits 15-8
731         SPL2INIT:1, GPIOIRQ:1, RXPTO:1, RXOVRR:1, Reserved0:1,
    LDEERR:1, RXRFTO:1, RXRFSL:1, % bits 23-16
732         ICRBP:1, HSRBP:1, AFFREJ:1, TXBERR:1, HPDWARN:1, RXSFDTO
```

```
    :1, CLCKPLL_LL:1, RFPLL_LL:1, % bits 31-24
733         Reserved1:5, TXPUTE:1, RXPREJ:1, RXRSCS:1 % bits 39-32
734     >>;
735 reg(decode, rx_finfo, Resp) ->
736     <<
737         RXPACC:12, RXPSR:2, RXPRFR:2, RNG:1, RXBR:2, RXNSPL:2, _
    :1, RXFLE:3, RXFLEN:7
738     >> = reverse(Resp),
739     #{
740         rxpacc => RXPACC, rxpsr => RXPSR, rxprfr => RXPRFR, rng =>
    RNG, rxbr => RXBR, rxnspl => RXNSPL, rxfle => RXFLE, rxflen =>
    RXFLEN
741     };
742 reg(decode, rx_buffer, Resp) ->
743     #{ rx_buffer => reverse(Resp)};
744 reg(decode, rx_fqual, Resp) ->
745     <<
746         CIR_PWR:16, PP_APL3:16, FP_AMPL2:16, STD_NOISE:16
747     >> = Resp,
748     #{
749         cir_pwr => CIR_PWR, pp_apl3 => PP_APL3, fp_ampl2 =>
    FP_AMPL2, std_noise => STD_NOISE
750     };
751 reg(decode, rx_ttcki, Resp) ->
752     #{
753         rx_ttcki => reverse(Resp)
754     };
755 reg(decode, rx_ttcko, Resp) ->
756     <<
757         _:1, RCPHASE:7, RSMPDEL:8, _:5, RXTOFS:19
758     >> = reverse(Resp),
759     #{
760         rcphase => RCPHASE, rsmpdel => RSMPDEL, rxtofs => RXTOFS
761     };
762 reg(decode, rx_time, Resp) ->
763     <<
764         RX_RAWST:40, FP_AMPL1:16, FP_INDEX:16, RX_STAMP:40
765     >> = reverse(Resp),
766     #{
767         rx_rawst => RX_RAWST, fp_ampl1 => FP_AMPL1, fp_index =>
    FP_INDEX, rx_stamp => RX_STAMP
768     };
769 reg(decode, tx_time, Resp) ->
770     <<
771         TX_RAWST:40, TX_STAMP:40
772     >> = reverse(Resp),
773     #{
774         tx_rawst => TX_RAWST, tx_stamp => TX_STAMP
775     };
```

```
776 reg(decode, tx_antd, Resp) ->
777     #{
778         tx_antd => reverse(Resp)
779     };
780 reg(encode, tx_antd, Val) ->
781     #{
782         tx_antd := TX_ANTD
783     } = Val,
784     reverse(<<
785         TX_ANTD:16
786     >>);
787 reg(decode, sys_state, Resp) ->
788     <<
789         _:8, PMSC_STATE:8, _:3, RX_STATE:5, _:4, TX_STATE:4
790     >> = reverse(Resp),
791     #{
792         pmsc_state => PMSC_STATE, rx_state => RX_STATE, tx_state
    => TX_STATE
793     };
794 reg(decode, ack_resp_t, Resp) ->
795     <<
796         ACK_TIME:8, _:4, W4R_TIME:20
797     >> = reverse(Resp),
798     #{
799         ack_tim => ACK_TIME, w4r_tim => W4R_TIME
800     };
801 reg(encode, ack_resp_t, Val) ->
802     #{
803         ack_tim := ACK_TIME, w4r_tim := W4R_TIME
804     } = Val,
805     reverse(<<
806         ACK_TIME:8, 2#0:4, W4R_TIME:20
807     >>);
808 reg(decode, rx_sniff, Resp) ->
809     <<
810         Reserved0:16, SNIFF_OFFT:8, Reserved1:4, SNIFF_ONT:4
811     >> = reverse(Resp),
812     #{
813         res0 => Reserved0,
814         sniff_offt => SNIFF_OFFT,
815         sniff_ont => SNIFF_ONT,
816         res1 => Reserved1
817     };
818 reg(encode, rx_sniff, Val) ->
819     #{
820         res0 := Reserved0,
821         sniff_offt := SNIFF_OFFT,
822         sniff_ont := SNIFF_ONT,
823         res1 := Reserved1
```

```
824     } = Val,
825     reverse(<<
826         Reserved0:16, SNIFF_OFFT:8, Reserved1:4, SNIFF_ONT:4
827     >>);
828 % Smart transmit power control (cf. user manual p 104)
829 reg(decode, tx_power, Resp) ->
830     <<
831         BOOSTP125:8, BOOSTP250:8, BOOSTP500:8, BOOSTNORM:8
832     >> = reverse(Resp),
833     #{
834         boostp125 => BOOSTP125, boostp250 => BOOSTP250, boostp500
    => BOOSTP500, boostnorm => BOOSTNORM
835     };
836 reg(encode, tx_power, Val) ->
837     % Leave the possibility to the user to write the value as one
838     case Val of
839         #{ tx_power := ValToEncode } -> reverse(<<ValToEncode
    :32>>);
840         #{ boostp125 := BOOSTP125, boostp250 := BOOSTP250,
    boostp500 := BOOSTP500, boostnorm := BOOSTNORM } ->reverse(<<
    BOOSTP125:8, BOOSTP250:8, BOOSTP500:8, BOOSTNORM:8>>)
841     end;
842 reg(decode, chan_ctrl, Resp) ->
843     <<
844         RX_PCODE:5, TX_PCODE:5, RNSSFD:1, TNSSFD:1, RXPRF:2, DWSFD
    :1, Reserved0:9, RX_CHAN:4, TX_CHAN:4
845     >> = reverse(Resp),
846     #{
847         rx_pcode => RX_PCODE, tx_pcode => TX_PCODE, rnssfd =>
    RNSSFD, tnssfd => TNSSFD, rxprf => RXPRF, dwsfd => DWSFD, res0
    => Reserved0, rx_chan => RX_CHAN, tx_chan => TX_CHAN
848     };
849 reg(encode, chan_ctrl, Val) ->
850     #{
851         rx_pcode := RX_PCODE, tx_pcode := TX_PCODE, rnssfd :=
    RNSSFD, tnssfd := TNSSFD, rxprf := RXPRF, dwsfd := DWSFD, res0
    := Reserved0, rx_chan := RX_CHAN, tx_chan := TX_CHAN
852     } = Val,
853     reverse(<<
854         RX_PCODE:5, TX_PCODE:5, RNSSFD:1, TNSSFD:1, RXPRF:2, DWSFD
    :1, Reserved0:9, RX_CHAN:4, TX_CHAN:4
855     >>);
856 reg(encode, usr_sfd, Value) ->
857     #{
858       usr_sfd := USR_SFD
859      } = Value,
860     reverse(<<
861         USR_SFD:(8*41)
862     >>);
```

```
reg(decode, usr_sfd, Resp) ->
    <<
        USR_SFD:(8*41)
    >> = reverse(Resp),
    #{
      usr_sfd => USR_SFD
     };
% AGC_CTRL is a complex register with reserved bits that can't be written
reg(encode, agc_ctrl1, Val) ->
    #{
        res := Reserved, dis_am := DIS_AM
    } = Val,
    reverse(<<
        Reserved:15, DIS_AM:1
    >>);
reg(encode, agc_tune1, Val) ->
    #{
      agc_tune1 := AGC_TUNE1
     } = Val,
    reverse(<<
        AGC_TUNE1:16
    >>);
reg(encode, agc_tune2, Val) ->
    #{
        agc_tune2 := AGC_TUNE2
     } = Val,
    reverse(<<
        AGC_TUNE2:32
    >>);
reg(encode, agc_tune3, Val) ->
    #{
        agc_tune3 := AGC_TUNE3
     } = Val,
    reverse(<<
        AGC_TUNE3:16
    >>);
reg(decode, agc_ctrl, Resp) ->
    <<
        _:4, EDV2:9, EDG1:5, _:6, % AGC_STAT1 (RP => don't save reserved bits)
        _:80, % Reserved 4
        AGC_TUNE3:16, % AGC_TUNE3
        _:16, % Reserved 3
        AGC_TUNE2:32, % AGC_TUNE2
        _:48, % Reserved 2
        AGC_TUNE1:16, % AGC_TUNE1
        Reserved0:15, DIS_AM:1, % AGC_CTRL1 (RW => save reserved bits)
```

```
909             _:16 % Reserved 1
910         >> = reverse(Resp),
911         #{
912             agc_ctrl1 => #{res => Reserved0, dis_am => DIS_AM},
913             agc_tune1 => AGC_TUNE1,
914             agc_tune2 => AGC_TUNE2,
915             agc_tune3 => AGC_TUNE3,
916             agc_stat1 => #{edv2 => EDV2, edg1 => EDG1}
917         };
918 reg(encode, ec_ctrl, Val) ->
919         #{
920             res := Reserved, ostrm := OSTRM, wait := WAIT, plplldt :=
        PLLLDT, osrsm := OSRSM, ostsm := OSTSM
921          } = Val,
922         reverse(<<
923             Reserved:20, OSTRM:1, WAIT:8, PLLLDT:1, OSRSM:1, OSTSM:1 %
        EC_CTRL
924         >>);
925 reg(decode, ext_sync, Resp) ->
926         <<
927             _:26, OFFSET_EXT:6, % EC_GLOP
928             RX_TS_EST:32, % EC_RXTC
929             Reserved:20, OSTRM:1, WAIT:8, PLLLDT:1, OSRSM:1, OSTSM:1 %
        EC_CTRL
930         >> = reverse(Resp),
931         #{
932             ec_ctrl => #{res => Reserved, ostrm => OSTRM, wait => WAIT
        , plplldt => PLLLDT, osrsm => OSRSM, ostsm => OSTSM},
933             rx_ts_est => RX_TS_EST,
934             ec_golp => #{offset_ext => OFFSET_EXT}
935         };
936 % "The host system doesn't need to access the ACC_MEM in normal
        operation, however it may be of interest [...] for diagnostic
        purpose" (from DW1000 user manual)
937 reg(decode, acc_mem, Resp) ->
938         #{
939             acc_mem => reverse(Resp)
940         };
941 reg(encode, gpio_mode, Val) ->
942         #{
943             msgp8 := MSGP8, msgp7 := MSGP7, msgp6 := MSGP6, msgp5 :=
        MSGP5, msgp4 := MSGP4, msgp3 := MSGP3, msgp2 := MSGP2, msgp1 :=
        MSGP1, msgp0 := MSGP0
944          } = Val,
945         reverse(<<
946             2#0:8, MSGP8:2, MSGP7:2, MSGP6:2, MSGP5:2, MSGP4:2, MSGP3
        :2, MSGP2:2, MSGP1:2, MSGP0:2, 2#0:6 % GPIO_MODE
947         >>);
948 reg(encode, gpio_dir, Val) ->
```

```
949     #{
950         gdm8 := GDM8, gdm7 := GDM7, gdm6 := GDM6, gdm5 := GDM5,
    gdm4 := GDM4, gdm3 := GDM3, gdm2 := GDM2, gdm1 := GDM1, gdm0 :=
     GDM0,
951         gdp8 := GDP8, gdp7 := GDP7, gdp6 := GDP6, gdp5 := GDP5,
    gdp4 := GDP4, gdp3 := GDP3, gdp2 := GDP2, gdp1 := GDP1, gdp0 :=
     GDP0
952      } = Val,
953     reverse(<<
954         2#0:11, GDM8:1, 2#0:3, GDP8:1, GDM7:1, GDM6:1, GDM5:1,
    GDM4:1, GDP7:1, GDP6:1, GDP5:1, GDP4:1, GDM3:1, GDM2:1, GDM1:1,
     GDM0:1, GDP3:1, GDP2:1, GDP1:1, GDP0:1 % GPIO2_DIR
955     >>);
956 reg(encode, gpio_dout, Val) ->
957     #{
958         gom8 := GOM8, gom7 := GOM7, gom6 := GOM6, gom5 := GOM5,
    gom4 := GOM4, gom3 := GOM3, gom2 := GOM2, gom1 := GOM1, gom0 :=
     GOM0,
959         gop8 := GOP8, gop7 := GOP7, gop6 := GOP6, gop5 := GOP5,
    gop4 := GOP4, gop3 := GOP3, gop2 := GOP2, gop1 := GOP1, gop0 :=
     GOP0
960      } = Val,
961     reverse(<<
962         2#0:11, GOM8:1, 2#0:3, GOP8:1, GOM7:1, GOM6:1, GOM5:1,
    GOM4:1, GOP7:1, GOP6:1, GOP5:1, GOP4:1, GOM3:1, GOM2:1, GOM1:1,
     GOM0:1, GOP3:1, GOP2:1, GOP1:1, GOP0:1 % GPIO_DOUT
963     >>);
964 reg(encode, gpio_irqe, Val) ->
965     #{
966         girqe8 := GIRQE8, girqe7 := GIRQE7, girqe6 := GIRQE6,
    girqe5 := GIRQE5, girqe4 := GIRQE4, girqe3 := GIRQE3, girqe2 :=
     GIRQE2, girqe1 := GIRQE1, girqe0 := GIRQE0
967      } = Val,
968     reverse(<<
969         2#0:23, GIRQE8:1, GIRQE7:1, GIRQE6:1, GIRQE5:1, GIRQE4:1,
    GIRQE3:1, GIRQE2:1, GIRQE1:1, GIRQE0:1 % GPIO_IRQE
970     >>);
971 reg(encode, gpio_isen, Val) ->
972     #{
973         gisen8 := GISEN8, gisen7 := GISEN7, gisen6 := GISEN6,
    gisen5 := GISEN5, gisen4 := GISEN4, gisen3 := GISEN3, gisen2 :=
     GISEN2, gisen1 := GISEN1, gisen0 := GISEN0
974      } = Val,
975     reverse(<<
976         2#0:23, GISEN8:1, GISEN7:1, GISEN6:1, GISEN5:1, GISEN4:1,
    GISEN3:1, GISEN2:1, GISEN1:1, GISEN0:1 % GPIO_ISEN
977     >>);
978 reg(encode, gpio_imod, Val) ->
979     #{
```

```
980         gimod8 := GIMOD8, gimod7 := GIMOD7, gimod6 := GIMOD6,
    gimod5 := GIMOD5, gimod4 := GIMOD4, gimod3 := GIMOD3, gimod2 :=
     GIMOD2, gimod1 := GIMOD1, gimod0 := GIMOD0
981      } = Val,
982     reverse(<<
983         2#0:23, GIMOD8:1, GIMOD7:1, GIMOD6:1, GIMOD5:1, GIMOD4:1,
    GIMOD3:1, GIMOD2:1, GIMOD1:1, GIMOD0:1 % GPIO_IMOD
984     >>);
985 reg(encode, gpio_ibes, Val) ->
986     #{
987         gibes8 := GIBES8, gibes7 := GIBES7, gibes6 := GIBES6,
    gibes5 := GIBES5, gibes4 := GIBES4, gibes3 := GIBES3, gibes2 :=
     GIBES2, gibes1 := GIBES1, gibes0 := GIBES0
988      } = Val,
989     reverse(<<
990         2#0:23, GIBES8:1, GIBES7:1, GIBES6:1, GIBES5:1, GIBES4:1,
    GIBES3:1, GIBES2:1, GIBES1:1, GIBES0:1 % GPIO_IBES
991     >>);
992 reg(encode, gpio_iclr, Val) ->
993     #{
994         giclr8 := GICLR8, giclr7 := GICLR7, giclr6 := GICLR6,
    giclr5 := GICLR5, giclr4 := GICLR4, giclr3 := GICLR3, giclr2 :=
     GICLR2, giclr1 := GICLR1, giclr0 := GICLR0
995      } = Val,
996     reverse(<<
997         2#0:23, GICLR8:1, GICLR7:1, GICLR6:1, GICLR5:1, GICLR4:1,
    GICLR3:1, GICLR2:1, GICLR1:1, GICLR0:1 % GPIO_ICLR
998     >>);
999 reg(encode, gpio_idbe, Val) ->
1000     #{
1001         gidbe8 := GIDBE8, gidbe7 := GIDBE7, gidbe6 := GIDBE6,
    gidbe5 := GIDBE5, gidbe4 := GIDBE4, gidbe3 := GIDBE3, gidbe2 :=
     GIDBE2, gidbe1 := GIDBE1, gidbe0 := GIDBE0
1002      } = Val,
1003     reverse(<<
1004         2#0:23, GIDBE8:1, GIDBE7:1, GIDBE6:1, GIDBE5:1, GIDBE4:1,
    GIDBE3:1, GIDBE2:1, GIDBE1:1, GIDBE0:1 % GPIO_IDBE
1005     >>);
1006 reg(encode, gpio_raw, Val) ->
1007     #{
1008         grawp8 := GRAWP8, grawp7 := GRAWP7, grawp6 := GRAWP6,
    grawp5 := GRAWP5, grawp4 := GRAWP4, grawp3 := GRAWP3, grawp2 :=
     GRAWP2, grawp1 := GRAWP1, grawp0 := GRAWP0
1009      } = Val,
1010     reverse(<<
1011         2#0:23, GRAWP8:1, GRAWP7:1, GRAWP6:1, GRAWP5:1, GRAWP4:1,
    GRAWP3:1, GRAWP2:1, GRAWP1:1, GRAWP0:1 % GPIO_RAW
1012     >>);
1013 reg(decode, gpio_ctrl, Resp) ->
```

```
1014     <<
1015         _:23, GRAWP8:1, GRAWP7:1, GRAWP6:1, GRAWP5:1, GRAWP4:1,
     GRAWP3:1, GRAWP2:1, GRAWP1:1, GRAWP0:1, % GPIO_RAW
1016         _:23, GIDBE8:1, GIDBE7:1, GIDBE6:1, GIDBE5:1, GIDBE4:1,
     GIDBE3:1, GIDBE2:1, GIDBE1:1, GIDBE0:1, % GPIO_IDBE
1017         _:23, GICLR8:1, GICLR7:1, GICLR6:1, GICLR5:1, GICLR4:1,
     GICLR3:1, GICLR2:1, GICLR1:1, GICLR0:1, % GPIO_ICLR
1018         _:23, GIBES8:1, GIBES7:1, GIBES6:1, GIBES5:1, GIBES4:1,
     GIBES3:1, GIBES2:1, GIBES1:1, GIBES0:1, % GPIO_IBES
1019         _:23, GIMOD8:1, GIMOD7:1, GIMOD6:1, GIMOD5:1, GIMOD4:1,
     GIMOD3:1, GIMOD2:1, GIMOD1:1, GIMOD0:1, % GPIO_IMOD
1020         _:23, GISEN8:1, GISEN7:1, GISEN6:1, GISEN5:1, GISEN4:1,
     GISEN3:1, GISEN2:1, GISEN1:1, GISEN0:1, % GPIO_ISEN
1021         _:23, GIRQE8:1, GIRQE7:1, GIRQE6:1, GIRQE5:1, GIRQE4:1,
     GIRQE3:1, GIRQE2:1, GIRQE1:1, GIRQE0:1, % GPIO_IRQE
1022         _:11, GOM8:1, _:3, GOP8:1, GOM7:1, GOM6:1, GOM5:1, GOM4:1,
      GOP7:1, GOP6:1, GOP5:1, GOP4:1, GOM3:1, GOM2:1, GOM1:1, GOM0
     :1, GOP3:1, GOP2:1, GOP1:1, GOP0:1, % GPIO_DOUT
1023         _:11, GDM8:1, _:3, GDP8:1, GDM7:1, GDM6:1, GDM5:1, GDM4:1,
      GDP7:1, GDP6:1, GDP5:1, GDP4:1, GDM3:1, GDM2:1, GDM1:1, GDM0
     :1, GDP3:1, GDP2:1, GDP1:1, GDP0:1, % GPIO_DIR
1024         _:32, % Reserved
1025         _:8, MSGP8:2, MSGP7:2, MSGP6:2, MSGP5:2, MSGP4:2, MSGP3:2,
      MSGP2:2, MSGP1:2, MSGP0:2, _:6 % GPIO_MODE
1026     >> = reverse(Resp),
1027     #{
1028         gpio_mode => #{msgp8 => MSGP8, msgp7 => MSGP7, msgp6 =>
     MSGP6, msgp5 => MSGP5, msgp4 => MSGP4, msgp3 => MSGP3, msgp2 =>
      MSGP2, msgp1 => MSGP1, msgp0 => MSGP0},
1029         gpio_dir => #{gdm8 => GDM8, gdm7 => GDM7, gdm6 => GDM6,
     gdm5 => GDM5, gdm4 => GDM4, gdm3 => GDM3, gdm2 => GDM2, gdm1 =>
      GDM1, gdm0 => GDM0,
1030                       gdp8 => GDP8, gdp7 => GDP7, gdp6 => GDP6,
     gdp5 => GDP5, gdp4 => GDP4, gdp3 => GDP3, gdp2 => GDP2, gdp1 =>
      GDP1, gdp0 => GDP0},
1031         gpio_dout => #{gom8 => GOM8, gom7 => GOM7, gom6 => GOM6,
     gom5 => GOM5, gom4 => GOM4, gom3 => GOM3, gom2 => GOM2, gom1 =>
      GOM1, gom0 => GOM0,
1032                        gop8 => GOP8, gop7 => GOP7, gop6 => GOP6,
     gop5 => GOP5, gop4 => GOP4, gop3 => GOP3, gop2 => GOP2, gop1 =>
      GOP1, gop0 => GOP0},
1033         gpio_irqe => #{girqe8 => GIRQE8, girqe7 => GIRQE7, girqe6
     => GIRQE6, girqe5 => GIRQE5, girqe4 => GIRQE4, girqe3 => GIRQE3
     , girqe2 => GIRQE2, girqe1 => GIRQE1, girqe0 => GIRQE0},
1034         gpio_isen => #{gisen8 => GISEN8, gisen7 => GISEN7, gisen6
     => GISEN6, gisen5 => GISEN5, gisen4 => GISEN4, gisen3 => GISEN3
     , gisen2 => GISEN2, gisen1 => GISEN1, gisen0 => GISEN0},
1035         gpio_imod => #{gimod8 => GIMOD8, gimod7 => GIMOD7, gimod6
     => GIMOD6, gimod5 => GIMOD5, gimod4 => GIMOD4, gimod3 => GIMOD3
```

```
    , gimod2 => GIMOD2, gimod1 => GIMOD1, gimod0 => GIMOD0},
1036        gpio_ibes => #{gibes8 => GIBES8, gibes7 => GIBES7, gibes6
    => GIBES6, gibes5 => GIBES5, gibes4 => GIBES4, gibes3 => GIBES3
    , gibes2 => GIBES2, gibes1 => GIBES1, gibes0 => GIBES0},
1037        gpio_iclr => #{giclr8 => GICLR8, giclr7 => GICLR7, giclr6
    => GICLR6, giclr5 => GICLR5, giclr4 => GICLR4, giclr3 => GICLR3
    , giclr2 => GICLR2, giclr1 => GICLR1, giclr0 => GICLR0},
1038        gpio_idbe => #{gidbe8 => GIDBE8, gidbe7 => GIDBE7, gidbe6
    => GIDBE6, gidbe5 => GIDBE5, gidbe4 => GIDBE4, gidbe3 => GIDBE3
    , gidbe2 => GIDBE2, gidbe1 => GIDBE1, gidbe0 => GIDBE0},
1039        gpio_raw => #{grawp8 => GRAWP8, grawp7 => GRAWP7, grawp6
    => GRAWP6, grawp5 => GRAWP5, grawp4 => GRAWP4, grawp3 => GRAWP3
    , grawp2 => GRAWP2, grawp1 => GRAWP1, grawp0 => GRAWP0}
1040    };
1041 reg(encode, drx_tune0b, Val) ->
1042     #{
1043         drx_tune0b := DRX_TUNE0b
1044      } = Val,
1045     reverse(<<
1046         DRX_TUNE0b:16
1047     >>);
1048 reg(encode, drx_tune1a, Val) ->
1049     #{
1050         drx_tune1a := DRX_TUNE1a
1051      } = Val,
1052     reverse(<<
1053         DRX_TUNE1a:16
1054     >>);
1055 reg(encode, drx_tune1b, Val) ->
1056     #{
1057         drx_tune1b := DRX_TUNE1b
1058      } = Val,
1059     reverse(<<
1060         DRX_TUNE1b:16
1061     >>);
1062 reg(encode, drx_tune2, Val) ->
1063     #{
1064         drx_tune2 := DRX_TUNE2
1065      } = Val,
1066     reverse(<<
1067         DRX_TUNE2:32
1068     >>);
1069 reg(encode, drx_sfdtoc, Val) ->
1070     #{
1071         drx_sfdtoc := DRX_SFDTOC
1072      } = Val,
1073     reverse(<<
1074         DRX_SFDTOC:16
1075     >>);
```

```
1076 reg(encode, drx_pretoc, Val) ->
1077     #{
1078         drx_pretoc := DRX_PRETOC
1079      } = Val,
1080     reverse(<<
1081         DRX_PRETOC:16
1082     >>);
1083 reg(encode, drx_tune4h, Val) ->
1084     #{
1085         drx_tune4h := DRX_TUNE4H
1086      } = Val,
1087     reverse(<<
1088         DRX_TUNE4H:16
1089     >>);
1090 reg(decode, drx_conf, Resp) ->
1091     <<
1092         %RXPACC_NOSAT:8, % present in the user manual but not in
    the driver code in C
1093         _:8, % Placeholder for the remaining 8 bits
1094         DRX_CAR_INT:24,
1095         DRX_TUNE4H:16,
1096         DRX_PRETOC:16,
1097         _:16,
1098         DRX_SFDTOC:16,
1099         _:160,
1100         DRX_TUNE2:32,
1101         DRX_TUNE1b:16,
1102         DRX_TUNE1a:16,
1103         DRX_TUNE0b:16,
1104         _:16
1105     >> = reverse(Resp),
1106     #{
1107         drx_tune0b => DRX_TUNE0b,
1108         drx_tune1a => DRX_TUNE1a,
1109         drx_tune1b => DRX_TUNE1b,
1110         drx_tune2 => DRX_TUNE2,
1111         drx_tune4h => DRX_TUNE4H,
1112         drx_car_int => DRX_CAR_INT,
1113         drx_sfdtoc => DRX_SFDTOC,
1114         drx_pretoc => DRX_PRETOC %,
1115         % rxpacc_nosat => RXPACC_NOSAT
1116     };
1117 reg(encode, rf_conf, Val) ->
1118     #{
1119         txrxsw := TXRXSW, ldofen := LDOFEN, pllfen := PLLFEN,
    txfen := TXFEN
1120      } = Val,
1121     reverse(<<
1122         2#0:9, TXRXSW:2, LDOFEN:5, PLLFEN:3, TXFEN:5, 2#0:8 %
```

```
    RF_CONF
1123     >>);
1124 reg(encode, rf_rxctrlh, Val) ->
1125     #{
1126         rf_rxctrlh := RF_RXCTRLH
1127      } = Val,
1128     reverse(<<
1129         RF_RXCTRLH:8 % RF_RXCTRLH
1130     >>);
1131 % user manual gives fields but encoding should be done as one
    following table 38
1132 reg(encode, rf_txctrl, Val) ->
1133     #{
1134       rf_txctrl := RF_TXCTRL
1135      } = Val,
1136     reverse(<<
1137         RF_TXCTRL:32
1138     >>);
1139 reg(encode, ldotune, Val) ->
1140     #{
1141       ldotune := LDOTUNE
1142      } = Val,
1143     reverse(<<
1144         LDOTUNE:40
1145     >>);
1146 reg(decode, rf_conf, Resp) ->
1147     <<
1148         _:40, % Placeholder for the remaining 40 bits
1149         LDOTUNE:40, % LDOTUNE
1150         _:28, RFPLLLOCK:1, CPLLHIGH:1, CPLLLOW:1, CPLLLOCK:1, %
    RF_STATUS
1151         _:128, _:96, % Reserved 2 - On user manual 16 bytes but
    offset gives 28 bytes (16 bytes (128 bits) + 12 bytes (96 bits)
    )
1152         RF_TXCTRL:32, % cf. encode function: Reserved:20, TXMQ:3,
    TXMTUNE:4, _:5 - RF_TXCTRL
1153         RF_RXCTRLH:8, % RF_RXCTRLH
1154         _:56, % Reserved 1
1155         _:9, TXRXSW:2, LDOFEN:5, PLLFEN:3, TXFEN:5, _:8 % RF_CONF
1156     >> = reverse(Resp),
1157     #{
1158         ldotune => LDOTUNE,
1159         rf_status => #{rfplllock => RFPLLLOCK, cplllow => CPLLLOW,
     cpllhigh => CPLLHIGH, cplllock => CPLLLOCK},
1160         rf_txctrl => RF_TXCTRL,
1161         rf_rxctrlh => RF_RXCTRLH,
1162         rf_conf => #{txrxsw => TXRXSW, ldofen => LDOFEN, pllfen =>
     PLLFEN, txfen => TXFEN}
1163     };
```

```
reg(encode, tc_sarc, Val) ->
    #{
        sar_ctrl := SAR_CTRL
     } = Val,
    reverse(<<
      2#0:15, SAR_CTRL:1
    >>);
reg(encode, tc_pg_ctrl, Val) ->
    #{
        pg_tmeas := PG_TMEAS, res := Reserved, pg_start := PG_START
     } = Val,
    reverse(<<
        2#0:2, PG_TMEAS:4, Reserved:1, PG_START:1
    >>);
reg(encode, tc_pgdelay, Val) ->
    #{
        tc_pgdelay := TC_PGDELAY
     } = Val,
    reverse(<<
        TC_PGDELAY:8
    >>);
reg(encode, tc_pgtest, Val) ->
    #{
        tc_pgtest := TC_PGTEST
     } = Val,
    reverse(<<
        TC_PGTEST:8
    >>);
reg(decode, tx_cal, Resp) ->
    <<
        TC_PGTEST:8, % TC_PGTEST
        TC_PGDELAY:8, % TC_PGDELAY
        _:4, DELAY_CNT:12, % TC_PG_STATUS
        _:2, PG_TMEAS:4, Reserved0:1, PG_START:1, % TC_PG_CTRL
        SAR_WTEMP:8, SAR_WVBAT:8, % TC_SARW
        _:8, SAR_LTEMP:8, SAR_LVBAT:8, % TC_SARL
        _:8, % Place holder to fill the gap between the offsets
        _:15, SAR_CTRL:1 % TC_SARC
    >> = reverse(Resp),
    #{
        tc_pgtest => TC_PGTEST,
        tc_pgdelay => TC_PGDELAY,
        tc_pg_status => #{delay_cnt => DELAY_CNT},
        tc_pg_ctrl => #{pg_tmeas => PG_TMEAS, res => Reserved0, pg_start => PG_START},
        tc_sarw => #{sar_wtemp => SAR_WTEMP, sar_wvbat => SAR_WVBAT},
        tc_sarl => #{sar_ltemp => SAR_LTEMP, sar_lvbat =>
```

```
    SAR_LVBAT},
1210        tc_sarc => #{sar_ctrl => SAR_CTRL}
1211    };
1212 reg(encode, fs_pllcfg, Val) ->
1213     #{
1214         fs_pllcfg := FS_PLLCFG
1215      } = Val,
1216     reverse(<<
1217         FS_PLLCFG:32
1218     >>);
1219 reg(encode, fs_plltune, Val) ->
1220     #{
1221         fs_plltune := FS_PLLTUNE
1222      } = Val,
1223     reverse(<<
1224         FS_PLLTUNE:8
1225     >>);
1226 reg(encode, fs_xtalt, Val) ->
1227     #{
1228       res := Reserved, xtalt := XTALT
1229      } = Val,
1230     reverse(<<
1231         Reserved:3, XTALT:5
1232     >>);
1233 reg(decode, fs_ctrl, Resp) ->
1234     <<
1235         _:48, % Reserved 3
1236         Reserved:3, XTALT:5, % FS_XTALT
1237         _:16, % Reserved 2
1238         FS_PLLTUNE:8, % FS_PLLTUNE
1239         FS_PLLCFG:32, % FS_PLLCFG
1240         _:56 % Reserved 1
1241     >> = reverse(Resp),
1242     #{
1243         fs_xtalt => #{res => Reserved, xtalt => XTALT},
1244         fs_plltune => FS_PLLTUNE,
1245         fs_pllcfg => FS_PLLCFG
1246     };
1247 reg(encode, aon_wcfg, Val) ->
1248     #{
1249         onw_lld := ONW_LLD, onw_llde := ONW_LLDE, pres_slee :=
    PRES_SLEE, own_l64 := OWN_L64, own_ldc := OWN_LDC, own_leui :=
    OWN_LEUI, own_rx := OWN_RX, own_rad := OWN_RAD
1250      } = Val,
1251     reverse(<<
1252         2#0:3, ONW_LLD:1, ONW_LLDE:1, 2#0:2, PRES_SLEE:1, OWN_L64
    :1, OWN_LDC:1, 2#0:2, OWN_LEUI:1, 2#0:1, OWN_RX:1, OWN_RAD:1 %
    AON_WCFG
1253     >>);
```

```
1254 reg(encode, aon_ctrl, Val) ->
1255     #{
1256         dca_enab := DCA_ENAB, dca_read := DCA_READ, upl_cfg :=
     UPL_CFG, save := SAVE, restore := RESTORE
1257       } = Val,
1258     reverse(<<
1259         DCA_ENAB:1, 2#0:3, DCA_READ:1, UPL_CFG:1, SAVE:1, RESTORE
     :1 % AON_CTRL
1260     >>);
1261 reg(encode, aon_rdat, Val) ->
1262     #{
1263         aon_rdat := AON_RDAT
1264       } = Val,
1265     reverse(<<
1266         AON_RDAT:8 % AON_RDAT
1267     >>);
1268 reg(encode, aon_addr, Val) ->
1269     #{
1270         aon_addr := AON_ADDR
1271       } = Val,
1272     reverse(<<
1273         AON_ADDR:8 % AON_ADDR
1274     >>);
1275 reg(encode, aon_cfg0, Val) ->
1276     #{
1277         sleep_tim := SLEEP_TIM, lpclkdiva := LPCLKDIVA, lpdiv_en
     := LPDIV_EN, wake_cnt := WAKE_CNT, wake_spi := WAKE_SPI,
     wake_pin := WAKE_PIN, sleep_en := SLEEP_EN
1278       } = Val,
1279     reverse(<<
1280         SLEEP_TIM:16, LPCLKDIVA:11, LPDIV_EN:1, WAKE_CNT:1,
     WAKE_SPI:1, WAKE_PIN:1, SLEEP_EN:1 % AON_CFG0
1281     >>);
1282 reg(encode, aon_cfg1, Val) ->
1283     #{
1284         res := Reserved, lposc_c := LPOSC_C, smxx := SMXX,
     sleep_ce := SLEEP_CE
1285       } = Val,
1286     reverse(<<
1287         Reserved:13, LPOSC_C:1, SMXX:1, SLEEP_CE:1 % AON_CFG1
1288     >>);
1289 reg(decode, aon, Resp) ->
1290     <<
1291         Reserved:13, LPOSC_C:1, SMXX:1, SLEEP_CE:1, % AON_CFG1
1292         SLEEP_TIM:16, LPCLKDIVA:11, LPDIV_EN:1, WAKE_CNT:1,
     WAKE_SPI:1, WAKE_PIN:1, SLEEP_EN:1, % AON_CFG0
1293         _:8, % Reserved 1
1294         AON_ADDR:8, % AON_ADDR
1295         AON_RDAT:8, % AON_RDAT
```

```
1296        DCA_ENAB:1, _:3, DCA_READ:1, UPL_CFG:1, SAVE:1, RESTORE:1,
    % AON_CTRL
1297        _:3, ONW_LLD:1, ONW_LLDE:1, _:2, PRES_SLEE:1, OWN_L64:1,
    OWN_LDC:1, _:2, OWN_LEUI:1, _:1, OWN_RX:1, OWN_RAD:1 % AON_WCFG
    >> = reverse(Resp),
1299    #{
1300        aon_cfg1 => #{res => Reserved, lposc_c => LPOSC_C, smxx =>
    SMXX, sleep_ce => SLEEP_CE},
1301        aon_cfg0 => #{sleep_tim => SLEEP_TIM, lpclkdiva =>
    LPCLKDIVA, lpdiv_en => LPDIV_EN, wake_cnt => WAKE_CNT, wake_spi
    => WAKE_SPI, wake_pin => WAKE_PIN, sleep_en => SLEEP_EN},
1302        aon_addr => AON_ADDR,
1303        aon_rdat => AON_RDAT,
1304        aon_ctrl => #{dca_enab => DCA_ENAB, dca_read => DCA_READ,
    upl_cfg => UPL_CFG, save => SAVE, restore => RESTORE},
1305        aon_wcfg => #{onw_lld => ONW_LLD, onw_llde => ONW_LLDE,
    pres_slee => PRES_SLEE, own_l64 => OWN_L64, own_ldc => OWN_LDC,
    own_leui => OWN_LEUI, own_rx => OWN_RX, own_rad => OWN_RAD}
1306    };
1307 reg(encode, otp_wdat, Val) ->
1308    #{
1309        otp_wdat := OTP_WDAT
1310    } = Val,
1311    reverse(<<
1312        OTP_WDAT:32 % OTP_WDAT
1313    >>);
1314 reg(encode, otp_addr, Val) ->
1315    #{
1316        otpaddr := OTP_ADDR, res := Reserved
1317    } = Val,
1318    reverse(<<
1319        Reserved:5, OTP_ADDR:11 % OTP_ADDR
1320    >>);
1321 reg(encode, otp_ctrl, Val) ->
1322    #{
1323        ldeload := LDELOAD, res1 := Reserved1, otpmr := OTPMR,
    otpprog := OTPPROG, res2 := Reserved2, otpmrwr := OTPMRWR, res3
    := Reserved3, otpread := OTPREAD, otp_rden := OTPRDEN
1324    } = Val,
1325    reverse(<<
1326        LDELOAD:1, Reserved1:4, OTPMR:4, OTPPROG:1, Reserved2:2,
    OTPMRWR:1, Reserved3:1, OTPREAD:1, OTPRDEN:1 % OTP_CTRL
1327    >>);
1328 reg(encode, otp_stat, Val) ->
1329    #{
1330        res := Reserved, otp_vpok := OTP_VPOK, otpprgd := OTPPRGD
1331    } = Val,
1332    reverse(<<
1333        Reserved:14, OTP_VPOK:1, OTPPRGD:1 % OTP_STAT
```

```
1334      >>);
1335  reg(encode, otp_rdat, Val) ->
1336      #{
1337          otp_rdat := OTP_RDAT
1338       } = Val,
1339      reverse(<<
1340          OTP_RDAT:32 % OTP_RDAT
1341      >>);
1342  reg(encode, opt_srdat, Val) ->
1343      #{
1344          otp_srdat := OTP_SRDAT
1345       } = Val,
1346      reverse(<<
1347          OTP_SRDAT:32 % OTP_SRDAT
1348      >>);
1349  reg(encode, otp_sf, Val) ->
1350      #{
1351          res1 := Reserved1, ops_sel := OPS_SEL, res2 := Reserved2,
     ldo_kick := LDO_KICK, ops_kick := OPS_KICK
1352       } = Val,
1353      reverse(<<
1354          Reserved1:2, OPS_SEL:1, Reserved2:3, LDO_KICK:1, OPS_KICK
     :1 % OTP_SF
1355      >>);
1356  reg(decode, otp_if, Resp) ->
1357      <<
1358          Reserved5:2, OPS_SEL:1, Reserved6:3, LDO_KICK:1, OPS_KICK
     :1, % OTP_SF
1359          OTP_SRDAT:32, % OTP_SRDAT
1360          OTP_RDAT:32, % OTP_RDAT
1361          Reserved4:14, OTP_VPOK:1, OTPPRGD:1, % OTP_STAT
1362          LDELOAD:1, Reserved1:4, OTPMR:4, OTPPROG:1, Reserved2:2,
     OTPMRWR:1, Reserved3:1, OTPREAD:1, OTPRDEN:1, % OTP_CTRL
1363          Reserved0:5, OTP_ADDR:11, % OTP_ADDR
1364          OTP_WDAT:32 % OTP_WDAT
1365      >> = reverse(Resp),
1366      #{
1367          otp_sf => #{res1 => Reserved5, ops_sel => OPS_SEL, res2 =>
      Reserved6, ldo_kick => LDO_KICK, ops_kick => OPS_KICK},
1368          otp_srdat => OTP_SRDAT,
1369          otp_rdat => OTP_RDAT,
1370          otp_stat => #{res => Reserved4, otp_vpok => OTP_VPOK,
     otpprgd => OTPPRGD},
1371          otp_ctrl => #{ldeload => LDELOAD, res1 => Reserved1, otpmr
      => OTPMR, otpprog => OTPPROG, res2 => Reserved2, otpmrwr =>
     OTPMRWR, res3 => Reserved3, otpread => OTPREAD, otp_rden =>
     OTPRDEN},
1372          otp_addr => #{otpaddr => OTP_ADDR, res => Reserved0},
1373          otp_wdat => OTP_WDAT
```

```
    };
reg(decode, lde_thresh, Resp) ->
    <<
      LDE_THRESH:16
    >> = reverse(Resp),
    #{
      lde_thresh => LDE_THRESH
    };
reg(encode, lde_cfg1, Val) ->
    #{
      pmult := PMULT, ntm := NTM
     } = Val,
    reverse(<<
        PMULT:3, NTM:5
    >>);
reg(decode, lde_cfg1, Resp) ->
    <<
      PMULT:3, NTM:5
    >> = reverse(Resp),
    #{
      lde_cfg1 => #{pmult => PMULT, ntm => NTM}
    };
reg(decode, lde_ppindx, Resp) ->
    <<
      LDE_PPINDX:16
    >> = reverse(Resp),
    #{
      lde_ppindx => LDE_PPINDX
    };
reg(decode, lde_ppampl, Resp) ->
    <<
      LDE_PPAMPL:16
    >> = reverse(Resp),
    #{
      lde_ppampl => LDE_PPAMPL
    };
reg(encode, lde_rxantd, Val) ->
    #{
      lde_rxantd := LDE_RXANTD
     } = Val,
    reverse(<<
        LDE_RXANTD:16
    >>);
reg(decode, lde_rxantd, Resp) ->
    <<
      LDE_RXANTD:16
    >> = reverse(Resp),
    #{
      lde_rxantd => LDE_RXANTD
```

```
    };
reg(encode, lde_cfg2, Val) ->
    #{
        lde_cfg2 := LDE_CFG2
     } = Val,
    reverse(<<
        LDE_CFG2:16
    >>);
reg(decode, lde_cfg2, Resp) ->
    <<
      LDE_CFG2:16
    >> = reverse(Resp),
    #{
      lde_cfg2 => LDE_CFG2
    };
reg(encode, lde_repc, Val) ->
    #{
        lde_repc := LDE_REPC
     } = Val,
    reverse(<<
        LDE_REPC:16
    >>);
reg(decode, lde_repc, Resp) ->
    <<
      LDE_REPC:16
    >> = reverse(Resp),
    #{
      lde_repc => LDE_REPC
    };
reg(encode, evc_ctrl, Val) ->
    #{
        evc_clr := EVC_CLR, evc_en := EVC_EN
     } = Val,
    reverse(<<
        2#0:30, EVC_CLR:1, EVC_EN:1 % EVC_CTRL
    >>);
reg(encode, diag_tmc, Val) ->
    #{
        tx_pstm := TX_PSTM
     } = Val,
    reverse(<<
        2#0:11, TX_PSTM:1, 2#0:4 % DIAG_TMC
    >>);
reg(decode, dig_diag, Resp) ->
    <<
        _:11, TX_PSTM:1, _:4, % DIAG_TMC
        _:64, % Reserved 1
        _:4, EVC_TPW:12, % EVC_TPW
        _:4, EVC_HPW:12, % EVC_HPW
```

```
1472            _:4, EVC_TXFS:12, % EVC_TXFS
1473            _:4, EVC_FWTO:12, % EVC_FWTO
1474            _:4, EVC_PTO:12, % EVC_PTO
1475            _:4, EVC_STO:12, % EVC_STO
1476            _:4, ECV_OVR:12, % EVC_OVR
1477            _:4, EVC_FFR:12, % EVC_FFR
1478            _:4, EVC_FCE:12, % EVC_FCE
1479            _:4, EVC_FCG:12, % EVC_FCG
1480            _:4, EVC_RSE:12, % EVC_RSE
1481            _:4, EVC_PHE:12, % EVC_PHE
1482            _:30, EVC_CLR:1, EVC_EN:1 % EVC_CTRL
1483        >> = reverse(Resp),
1484        #{
1485            diag_tmc => #{tx_pstm => TX_PSTM},
1486            evc_tpw => EVC_TPW,
1487            evc_hpw => EVC_HPW,
1488            evc_txfs => EVC_TXFS,
1489            evc_fwto => EVC_FWTO,
1490            evc_pto => EVC_PTO,
1491            evc_sto => EVC_STO,
1492            evc_ovr => ECV_OVR,
1493            evc_ffr => EVC_FFR,
1494            evc_fce => EVC_FCE,
1495            evc_fcg => EVC_FCG,
1496            evc_rse => EVC_RSE,
1497            evc_phe => EVC_PHE,
1498            evc_ctrl => #{evc_clr => EVC_CLR, evc_en => EVC_EN}
1499        };
1500 reg(encode, pmsc_ctrl0, Val) ->
1501        #{
1502            softreset := SOFTRESET, pll2_seq_en := PLL2_SEQ_EN,
       khzclken := KHZCLKEN, gpdrn := GPDRN, gpdce := GPDCE,
1503            gprn := GPRN, gpce := GPCE, amce := AMCE, adcce := ADCCE,
       otp := OTP, res8 := Res8, res7 := Res7, face := FACE, txclks :=
        TXCLKS, rxclks := RXCLKS, sysclks := SYSCLKS % Here we need
       res8 for the initial config of the DW1000. We need to write it
1504         } = Val,
1505        reverse(<<
1506            SOFTRESET:4, 2#000:3, PLL2_SEQ_EN:1, KHZCLKEN:1, 2#011:3,
       GPDRN:1, GPDCE:1, GPRN:1, GPCE:1, AMCE:1, 2#0000:4, ADCCE:1,
       OTP:1, Res8:1, Res7:1, FACE:1, TXCLKS:2, RXCLKS:2, SYSCLKS:2 %
       PMSC_CTRL0
1507        >>);
1508 reg(encode, pmsc_ctrl1, Val) ->
1509        #{
1510            khzclkdiv := KHZCLKDIV, lderune := LDERUNE, pllsyn :=
       PLLSYN, snozr := SNOZR, snoze := SNOZE, arxslp := ARXSLP,
       atxslp := ATXSLP, pktseq := PKTSEQ, arx2init := ARX2INIT
1511         } = Val,
```

```
1512     reverse(<<
1513         KHZCLKDIV:6, 2#01000000:8, LDERUNE:1, 2#0:1, PLLSYN:1,
     SNOZR:1, SNOZE:1, ARXSLP:1, ATXSLP:1, PKTSEQ:8, 2#0:1, ARX2INIT
     :1, 2#0:1 % PMSC_CTRL1
1514     >>);
1515 reg(encode, pmsc_snozt, Val) ->
1516     #{
1517         snoz_tim := SNOZ_TIM
1518      } = Val,
1519     reverse(<<
1520         SNOZ_TIM:8 % PMSC_SNOZT
1521     >>);
1522 reg(encode, pmsc_txfseq, Val) ->
1523     #{
1524         txfineseq := TXFINESEQ
1525      } = Val,
1526     reverse(<<
1527         TXFINESEQ:16 % PMSC_TXFINESEQ
1528     >>);
1529 reg(encode, pmsc_ledc, Val) ->
1530     #{
1531         res31 := RES31, blnknow := BLNKNOW, res15 := RES15, blnken
     := BLNKEN, blink_tim := BLINK_TIM
1532      } = Val,
1533     reverse(<<
1534         RES31:12, BLNKNOW:4, RES15:7, BLNKEN:1, BLINK_TIM:8 %
     PMSC_LEDC
1535     >>);
1536 % mapping pmsc ctrl0 from: https://forum.qorvo.com/t/pmsc-ctrl0-
     bits8-15/746/3
1537 reg(decode, pmsc, Resp) ->
1538     % User manual says: reserved bits should be preserved at their
      reset value => can hardcode their values ? Safe to do that ?
1539     <<
1540         Res31:12, BLNKNOW:4, Res15:7, BLNKEN:1, BLINK_TIM:8, %
     PMSC_LEDC
1541         TXFINESEQ:16, % PMSC_TXFINESEQ
1542         _:(25*8), % Reserved 2
1543         SNOZ_TIM:8, % PMSC_SNOZT
1544         _:32, % Reserved 1
1545         KHZCLKDIV:6, _:8, LDERUNE:1, _:1, PLLSYN:1, SNOZR:1, SNOZE
     :1, ARXSLP:1, ATXSLP:1, PKTSEQ:8, _:1, ARX2INIT:1, _:1, %
     PMSC_CTRL1
1546         SOFTRESET:4, _:3, PLL2_SEQ_EN:1, KHZCLKEN:1, _:3, GPDRN:1,
      GPDCE:1, GPRN:1, GPCE:1, AMCE:1, _:4, ADCCE:1, OTP:1, Res8:1,
     Res7:1, FACE:1, TXCLKS:2, RXCLKS:2, SYSCLKS:2 % PMSC_CTRL0
1547     >> = reverse(Resp),
1548     #{
1549         pmsc_ledc => #{res31 => Res31, blnknow => BLNKNOW, res15
```

```
    => Res15, blnken => BLNKEN, blink_tim => BLINK_TIM},
1550          pmsc_txfseq => #{txfineseq => TXFINESEQ},
1551          pmsc_snozt => #{snoz_tim => SNOZ_TIM},
1552          pmsc_ctrl1 => #{khzclkdiv => KHZCLKDIV, lderune => LDERUNE
    , pllsyn => PLLSYN, snozr => SNOZR, snoze => SNOZE, arxslp =>
    ARXSLP, atxslp => ATXSLP, pktseq => PKTSEQ, arx2init =>
    ARX2INIT},
1553          pmsc_ctrl0 => #{softreset => SOFTRESET, pll2_seq_en =>
    PLL2_SEQ_EN, khzclken => KHZCLKEN, gpdrn => GPDRN, gpdce =>
    GPDCE, gprn => GPRN, gpce => GPCE, amce => AMCE, adcce => ADCCE
    , otp => OTP, res8 => Res8, res7 => Res7, face => FACE, txclks
    => TXCLKS, rxclks => RXCLKS, sysclks => SYSCLKS}
1554      };
1555 reg(decode, RegFile, Resp) -> error({unknown_regfile_to_decode,
    RegFile, Resp});
1556 reg(encode, RegFile, Resp) -> error({unknown_regfile_to_encode,
    RegFile, Resp}).
1557
1558 rw(read) -> 0;
1559 rw(write) -> 1.
1560
1561 % Mapping of the different register IDs to their hexadecimal value
1562 regFile(dev_id) -> 16#00;
1563 regFile(eui) -> 16#01;
1564 % 0x02 is reserved
1565 regFile(panadr) -> 16#03;
1566 regFile(sys_cfg) -> 16#04;
1567 % 0x05 is reserved
1568 regFile(sys_time) -> 16#06;
1569 % 0x07 is reserved
1570 regFile(tx_fctrl) -> 16#08;
1571 regFile(tx_buffer) -> 16#09;
1572 regFile(dx_time) -> 16#0A;
1573 % 0x0B is reserved
1574 regFile(rx_fwto) -> 16#0C;
1575 regFile(sys_ctrl) -> 16#0D;
1576 regFile(sys_mask) -> 16#0E;
1577 regFile(sys_status) -> 16#0F;
1578 regFile(rx_finfo) -> 16#10;
1579 regFile(rx_buffer) -> 16#11;
1580 regFile(rx_fqual) -> 16#12;
1581 regFile(rx_ttcki) -> 16#13;
1582 regFile(rx_ttcko) -> 16#14;
1583 regFile(rx_time) -> 16#15;
1584 % 0x16 is reserved
1585 regFile(tx_time) -> 16#17;
1586 regFile(tx_antd) -> 16#18;
1587 regFile(sys_state) -> 16#19;
1588 regFile(ack_resp_t) -> 16#1A;
```

```
% 0x1B is reserved
% 0x1C is reserved
regFile(rx_sniff) -> 16#1D;
regFile(tx_power) -> 16#1E;
regFile(chan_ctrl) -> 16#1F;
% 0x20 is reserved
regFile(usr_sfd) -> 16#21;
% 0x22 is reserved
regFile(agc_ctrl) -> 16#23;
regFile(ext_sync) -> 16#24;
regFile(acc_mem) -> 16#25;
regFile(gpio_ctrl) -> 16#26;
regFile(drx_conf) -> 16#27;
regFile(rf_conf) -> 16#28;
% 0x29 is reserved
regFile(tx_cal) -> 16#2A;
regFile(fs_ctrl) -> 16#2B;
regFile(aon) -> 16#2C;
regFile(otp_if) -> 16#2D;
regFile(lde_ctrl) -> regFile(lde_if); % No size ?
regFile(lde_if) -> 16#2E;
regFile(dig_diag) -> 16#2F;
% 0x30 - 0x35 are reserved
regFile(pmsc) -> 16#36;
% 0x37 - 0x3F are reserved
regFile(RegId) -> error({wrong_register_ID, RegId}).

% Only the writtable subregisters in SRW register files are
    present here
% AGC_CTRL
subReg(agc_ctrl1) -> 16#02;
subReg(agc_tune1) -> 16#04;
subReg(agc_tune2) -> 16#0C;
subReg(agc_tune3) -> 16#12;
subReg(agc_stat1) -> 16#1E;
subReg(ec_ctrl) -> 16#00;
subReg(gpio_mode) -> 16#00;
subReg(gpio_dir) -> 16#08;
subReg(gpio_dout) -> 16#0C;
subReg(gpio_irqe) -> 16#10;
subReg(gpio_isen) -> 16#14;
subReg(gpio_imode) -> 16#18;
subReg(gpio_ibes) -> 16#1C;
subReg(gpio_iclr) -> 16#20;
subReg(gpio_idbe) -> 16#24;
subReg(gpio_raw) -> 16#28;
subReg(drx_tune0b) -> 16#02;
subReg(drx_tune1a) -> 16#04;
subReg(drx_tune1b) -> 16#06;
```

```
1637 subReg(drx_tune2) -> 16#08;
1638 subReg(drx_sfdtoc) -> 16#20;
1639 subReg(drx_pretoc) -> 16#24;
1640 subReg(drx_tune4h) -> 16#26;
1641 subReg(rf_conf) -> 16#00;
1642 subReg(rf_rxctrlh) -> 16#0B;
1643 subReg(rf_txctrl) -> 16#0C;
1644 subReg(ldotune) -> 16#30;
1645 subReg(tc_sarc) -> 16#00;
1646 subReg(tc_pg_ctrl) -> 16#08;
1647 subReg(tc_pgdelay) -> 16#0B;
1648 subReg(tc_pgtest) -> 16#0C;
1649 subReg(fs_pllcfg) -> 16#07;
1650 subReg(fs_plltune) -> 16#0B;
1651 subReg(fs_xtalt) -> 16#0E;
1652 subReg(aon_wcfg) -> 16#00;
1653 subReg(aon_ctrl) -> 16#02;
1654 subReg(aon_rdat) -> 16#03;
1655 subReg(aon_addr) -> 16#04;
1656 subReg(aon_cfg0) -> 16#06;
1657 subReg(aon_cfg1) -> 16#0A;
1658 subReg(otp_wdat) -> 16#00;
1659 subReg(otp_addr) -> 16#04;
1660 subReg(otp_ctrl) -> 16#06;
1661 subReg(otp_stat) -> 16#08;
1662 subReg(otp_rdat) -> 16#0A;
1663 subReg(otp_srdat) -> 16#0E;
1664 subReg(otp_sf) -> 16#12;
1665 subReg(lde_thresh) -> 16#00;
1666 subReg(lde_cfg1) -> 16#806;
1667 subReg(lde_ppindx) -> 16#1000;
1668 subReg(lde_ppampl) -> 16#1002;
1669 subReg(lde_rxantd) -> 16#1804;
1670 subReg(lde_cfg2) -> 16#1806;
1671 subReg(lde_repc) -> 16#2804;
1672 subReg(evc_ctrl) -> 16#00;
1673 subReg(diag_tmc) -> 16#24;
1674 subReg(pmsc_ctrl0) -> 16#00;
1675 subReg(pmsc_ctrl1) -> 16#04;
1676 subReg(pmsc_snozt) -> 16#0C;
1677 subReg(pmsc_txfseq) -> 16#26;
1678 subReg(pmsc_ledc) -> 16#28.
1679
1680
1681 % Mapping of the size in bytes of the different register IDs
1682 regSize(dev_id) -> 4;
1683 regSize(eui) -> 8;
1684 regSize(panadr) -> 4;
1685 regSize(sys_cfg) -> 4;
```

```
1686 regSize(sys_time) -> 5;
1687 regSize(tx_fctrl) -> 5;
1688 regSize(tx_buffer) -> 1024;
1689 regSize(dx_time) -> 5;
1690 regSize(rx_fwto) -> 2; % user manual gives 2 bytes and bits 16-31
       are reserved
1691 regSize(sys_ctrl) -> 4;
1692 regSize(sys_mask) -> 4;
1693 regSize(sys_status) -> 5;
1694 regSize(rx_finfo) -> 4;
1695 regSize(rx_buffer) -> 1024;
1696 regSize(rx_fqual) -> 8;
1697 regSize(rx_ttcki) -> 4;
1698 regSize(rx_ttcko) -> 5;
1699 regSize(rx_time) -> 14;
1700 regSize(tx_time) -> 10;
1701 regSize(tx_antd) -> 2;
1702 regSize(sys_state) -> 4;
1703 regSize(ack_resp_t) -> 4;
1704 regSize(rx_sniff) -> 4;
1705 regSize(tx_power) -> 4;
1706 regSize(chan_ctrl) -> 4;
1707 regSize(usr_sfd) -> 41;
1708 regSize(agc_ctrl) -> 33;
1709 regSize(ext_sync) -> 12;
1710 regSize(acc_mem) -> 4064;
1711 regSize(gpio_ctrl) -> 44;
1712 regSize(drx_conf) -> 44; % user manual gives 44 bytes but sum of
       register length gives 45 bytes
1713 regSize(rf_conf) -> 58; % user manual gives 58 but sum of all its
       register gives 53 => Placeholder for the remaining 8 bytes
1714 regSize(tx_cal) -> 13; % user manual gives 52 bytes but sum of all
        sub regs gives 13 bytes
1715 regSize(fs_ctrl) -> 21;
1716 regSize(aon) -> 12;
1717 regSize(otp_if) -> 19; % user manual gives 18 bytes in regs table
       but sum of all sub regs is 19 bytes
1718 regSize(lde_ctrl) -> undefined; % No size ?
1719 regSize(lde_if) -> undefined; % No size ?
1720 regSize(dig_diag) -> 38; % user manual gives 41 bytes but sum of
       all sub regs gives 38 bytes
1721 regSize(pmsc) -> 44. % user manual gives 48 bytes but sum of all
       sub regs gives 41 bytes
1722
1723 %% Gives the size in bytes
1724 subRegSize(agc_ctrl1) -> 2;
1725 subRegSize(agc_tune1) -> 2;
1726 subRegSize(agc_tune2) -> 4;
1727 subRegSize(agc_tune3) -> 2;
```

```
1728 subRegSize(agc_stat1) -> 3;
1729 subRegSize(ec_ctrl) -> 4;
1730 subRegSize(gpio_mode) -> 4;
1731 subRegSize(gpio_dir) -> 4;
1732 subRegSize(gpio_dout) -> 4;
1733 subRegSize(gpio_irqe) -> 4;
1734 subRegSize(gpio_isen) -> 4;
1735 subRegSize(gpio_imode) -> 4;
1736 subRegSize(gpio_ibes) -> 4;
1737 subRegSize(gpio_iclr) -> 4;
1738 subRegSize(gpio_idbe) -> 4;
1739 subRegSize(gpio_raw) -> 4;
1740 subRegSize(drx_tune0b) -> 2;
1741 subRegSize(drx_tune1a) -> 2;
1742 subRegSize(drx_tune1b) -> 2;
1743 subRegSize(drx_tune2) -> 4;
1744 subRegSize(drx_sfdtoc) -> 2;
1745 subRegSize(drx_pretoc) -> 2;
1746 subRegSize(drx_tune4h) -> 2;
1747 subRegSize(rf_conf) -> 4;
1748 subRegSize(rf_rxctrlh) -> 1;
1749 subRegSize(rf_txctrl) -> 4; % ! table in user manual gives 3 but
      details gives 4
1750 subRegSize(ldotune) -> 5;
1751 subRegSize(tc_sarc) -> 2;
1752 subRegSize(tc_pg_ctrl) -> 1;
1753 subRegSize(tc_pgdelay) -> 1;
1754 subRegSize(tc_pgtest) -> 1;
1755 subRegSize(fs_pllcfg) -> 4;
1756 subRegSize(fs_plltune) -> 1;
1757 subRegSize(fs_xtalt) -> 1;
1758 subRegSize(aon_wcfg) -> 2;
1759 subRegSize(aon_ctrl) -> 1;
1760 subRegSize(aon_rdat) -> 1;
1761 subRegSize(aon_addr) -> 1;
1762 subRegSize(aon_cfg0) -> 4;
1763 subRegSize(aon_cfg1) -> 2;
1764 subRegSize(otp_wdat) -> 4;
1765 subRegSize(otp_addr) -> 2;
1766 subRegSize(otp_ctrl) -> 2;
1767 subRegSize(otp_stat) -> 2;
1768 subRegSize(otp_rdat) -> 4;
1769 subRegSize(otp_srdat) -> 4;
1770 subRegSize(otp_sf) -> 1;
1771 subRegSize(lde_thresh) -> 2;
1772 subRegSize(lde_cfg1) -> 1;
1773 subRegSize(lde_ppindx) -> 2;
1774 subRegSize(lde_ppampl) -> 2;
1775 subRegSize(lde_rxantd) -> 2;
```

```
1776 subRegSize(lde_cfg2) -> 2;
1777 subRegSize(lde_repc) -> 2;
1778 subRegSize(evc_ctrl) -> 4;
1779 subRegSize(diag_tmc) -> 2;
1780 subRegSize(pmsc_ctrl0) -> 4;
1781 subRegSize(pmsc_ctrl1) -> 4;
1782 subRegSize(pmsc_snozt) -> 1;
1783 subRegSize(pmsc_txfseq) -> 2;
1784 subRegSize(pmsc_ledc) -> 4;
1785 subRegSize(_) -> error({error}).
1786
1787 %--- Debug ----------------------------------------------------------------
1788
1789 debug_read(Reg, Value) ->
1790     io:format("[PmodUWB] read [16#~2.16.0B - ~w] --> ~s -> ~s~n",
1791         [regFile(Reg), Reg, debug_bitstring(Value),
     debug_bitstring_hex(Value)]
1792     ).
1793
1794 debug_write(Reg, Value) ->
1795     io:format("[PmodUWB] write [16#~2.16.0B - ~w] --> ~s -> ~s~n",
1796         [regFile(Reg), Reg, debug_bitstring(Value),
     debug_bitstring_hex(Value)]
1797     ).
1798 debug_write(Reg, SubReg, Value) ->
1799     io:format("[PmodUWB] write [16#~2.16.0B - ~w - 16#~2.16.0B - ~
     w] --> ~s -> ~s~n",
1800         [regFile(Reg), Reg, subReg(SubReg), SubReg,
     debug_bitstring(Value), debug_bitstring_hex(Value)]
1801     ).
1802
1803 debug_bitstring(Bitstring) ->
1804     lists:flatten([io_lib:format("2#~8.2.0B ", [X]) || <<X>> <=
     Bitstring]).
1805
1806 debug_bitstring_hex(Bitstring) ->
1807     lists:flatten([io_lib:format("16#~2.16.0B ", [X]) || <<X>> <=
     Bitstring]).
```

Listing A.2: pmod_uwb.erl

Appendix B

MAC layer code

```
-include("pmod_uwb.hrl").

-define(FTYPE_BEACON, 3#000).
-define(FTYPE_DATA, 2#001).
-define(FTYPE_ACK, 2#010).
-define(FTYPE_MACCOM, 2#011).

-define(NONE, 2#00).
-define(SHORT_ADDR, 2#10).
-define(EXTENDED, 2#11).


-type ftype() :: ?FTYPE_BEACON | ?FTYPE_DATA | ?FTYPE_ACK | ?FTYPE_MACCOM.
-type addr_mode() :: ?NONE | ?SHORT_ADDR | ?EXTENDED.
-type addr() :: bitstring().

% @doc frame control of a MAC header for IEEE 802.15.4
-record(frame_control, {frame_type = ?FTYPE_DATA :: ftype(),
                        sec_en = ?DISABLED :: flag(),
                        frame_pending= ?DISABLED :: flag(),
                        ack_req = ?DISABLED :: flag(),
                        pan_id_compr = ?DISABLED :: flag(),
                        dest_addr_mode = ?SHORT_ADDR :: addr_mode(),
                        frame_version = 2#00 :: integer(),
                        src_addr_mode = ?SHORT_ADDR :: addr_mode()
    }).

% @doc MAC header for IEEE 802.15.4
% Doesn't include the frame control nor a potential auxiliary security header
-record(mac_header, {seqnum = 0 :: integer(),
```

```
30                      dest_pan = <<16#FFFF:16>> :: addr(),
31                      dest_addr = <<16#FFFF:16>> :: addr(),
32                      src_pan = <<16#FFFF:16>> :: addr(),
33                      src_addr = <<16#FFFF:16>> :: addr()}).
```

Listing B.1: mac_layer.hrl

```
1  -module(mac_layer).
2
3  -include("mac_layer.hrl").
4
5  -export([mac_send_data/3, mac_send_data/4, mac_receive/0,
       mac_receive/1]).
6  -export([mac_decode/1]).
7  -export([mac_frame/2, mac_frame/3]).
8
9
10 %--- API
       ----------------------------------------------------------------------
11
12 %
       ----------------------------------------------------------------------
13 % @doc builds a mac frame without a payload
14 % @equiv mac_frame(FrameControl, MacHeader, <<>>)
15 % @end
16 %
       ----------------------------------------------------------------------
17 -spec mac_frame(FrameControl :: #frame_control{}, MacHeader :: #
       mac_header{}) -> bitstring().
18 mac_frame(FrameControl, MacHeader) ->
19     mac_frame(FrameControl, MacHeader, <<>>).
20
21 %
       ----------------------------------------------------------------------
22 % @doc builds a mac frame
23 % @returns a MAC frame ready to be transmitted in a bitstring (not
       including the CRC automatically added by the DW1000)
24 % @end
25 %
       ----------------------------------------------------------------------
26 -spec mac_frame(FrameControl :: #frame_control{}, MacHeader :: #
       mac_header{}, Payload :: bitstring()) -> bitstring().
27 mac_frame(FrameControl, MacHeader, Payload) ->
28     Header = build_mac_header(FrameControl, MacHeader),
```

```
29      <<Header/bitstring, Payload/bitstring>>.
30
31  %
        ------------------------------------------------------------------------
32  % @doc Sends a MAC frame using the pmod_uwb without any options
33  % The 2 bytes CRC are automatically added at the end of the
        payload and
34  % must not be included in the Payload given in the arguments
35  %
        ------------------------------------------------------------------------
36  -spec mac_send_data(FrameControl :: #frame_control{}, MacHeader ::
        #mac_header{}, Payload :: bitstring()) -> ok.
37  mac_send_data(FrameControl, MacHeader, Payload) ->
38      mac_send_data(FrameControl, MacHeader, Payload, #tx_opts{}).
39
40  %
        ------------------------------------------------------------------------
41  % @doc Sends a MAC frame using the pmod_uwb using the specified
        options
42  % The 2 bytes CRC are automatically added at the end of the
        payload and
43  % must not be included in the Payload given in the arguments
44  % @end
45  %
        ------------------------------------------------------------------------
46  -spec mac_send_data(FrameControl :: #frame_control{}, MacHeader ::
        #mac_header{}, Payload :: bitstring(), Option :: #tx_opts{})
        -> ok | {FrameControl :: #frame_control{}, MacHeader :: #
        mac_header{}, Payload :: bitstring()}.
47  mac_send_data(FrameControl, MacHeader, Payload, Options) ->
48      Message = mac_frame(FrameControl, MacHeader, Payload),
49      pmod_uwb:transmit(Message, Options).
50
51  %
        ------------------------------------------------------------------------
52  % @doc Receive a frame using the pmod_uwb and decode the frame
53  %
54  % @equiv mac_receive(false)
55  %
56  % @return the received mac frame decoded
57  %
        ------------------------------------------------------------------------
58  -spec mac_receive() -> {FrameControl :: #frame_control{},
```

```
    MacHeader :: #mac_header{}, Payload :: bitstring()}.
mac_receive() ->
    mac_receive(false).

%
    ------------------------------------------------------------------------

% @doc Receive a frame using the pmod_uwb and decode the frame
% @param RXEnab indicates if the reception was already enabled (or
    is enabled with delay)
% <b>Warning:</b> if this function is called with RXEnab = true
    and the reception isn't set, the driver will be stuck in a loop
    without any timeout
% @return the received mac frame decoded
%
    ------------------------------------------------------------------------

-spec mac_receive(RXEnab :: boolean()) -> {FrameControl :: #
    frame_control{}, MacHeader :: #mac_header{}, Payload ::
    bitstring()}.
mac_receive(RXEnab) ->
    case pmod_uwb:reception(RXEnab) of
        {_Length, Data} -> mac_decode(Data);
        Err -> Err
    end.

%--- Internal
    ------------------------------------------------------------------


%
    ------------------------------------------------------------------------

% @doc builds a mac header based on the FrameControl and the
    MacHeader structures given in the args.
% <b> The MAC header doesn't support security fields yet </b>
% @returns the MAC header in a bitstring
% @end
%
    ------------------------------------------------------------------------

-spec build_mac_header(FrameControl :: #frame_control{}, MacHeader
    :: #mac_header{}) -> bitstring().
build_mac_header(FrameControl, MacHeader) ->
    FC = build_frame_control(FrameControl),

    DestPan = reverse_byte_order(MacHeader#mac_header.dest_pan,
    <<>>),
    DestAddr= reverse_byte_order(MacHeader#mac_header.dest_addr,
```

```
    <<>>),
89      DestAddrFields = case FrameControl#frame_control.
    dest_addr_mode of
90                           ?NONE -> <<>>;
91                           _ -> <<DestPan/bitstring, DestAddr/
    bitstring>>
92                       end,
93
94      SrcPan = reverse_byte_order(MacHeader#mac_header.src_pan,
    <<>>),
95      SrcAddr= reverse_byte_order(MacHeader#mac_header.src_addr,
    <<>>),
96      SrcAddrFields = case {FrameControl#frame_control.src_addr_mode
    , FrameControl#frame_control.pan_id_compr, FrameControl#
    frame_control.dest_addr_mode} of
97                           {?NONE, _, _} -> <<>>;
98                           {_, ?DISABLED, _} -> <<SrcPan/bitstring,
    SrcAddr/bitstring>>; % if no compression is applied on PANID
    and SRC addr is present
99                           {_, ?ENABLED, ?NONE}  -> <<SrcPan/
    bitstring, SrcAddr/bitstring>>; % if there is a compression of
    the PANID but the dest addr isn't present
100                          {_, ?ENABLED, _} -> <<SrcAddr/bitstring>>
    % if there is a compression of the PANID and the dest addr is
    present
101                      end,
102     <<FC/bitstring,  (MacHeader#mac_header.seqnum):8,
    DestAddrFields/bitstring, SrcAddrFields/bitstring>>.
103
104
105 %
    ----------------------------------------------------------------------------

106 % @doc decodes the MAC frame given in the arguments
107 % @return A tuple containing the decoded frame control, the
    decoded mac header and the payload
108 % @end
109 %
    ----------------------------------------------------------------------------

110 -spec mac_decode(Data :: bitstring()) -> {FrameControl :: #
    frame_control{}, MacHeader :: #mac_header{}, Payload ::
    bitstring()}.
111 mac_decode(Data) ->
112     <<FC:16/bitstring, Seqnum:8, Rest/bitstring>> = Data,
113     FrameControl = decode_frame_control(FC),
114     decode_rest(FrameControl, Seqnum, Rest).
115
116 %
```

```
    %%-------------------------------------------------------------------------

% @private
% @doc Decodes the remaining sequence of bit present in the payload after the seqnum
% @end
%
    %%-------------------------------------------------------------------------

decode_rest(#frame_control{frame_type = ?FTYPE_ACK} = FrameControl, Seqnum, _Rest) ->
    {FrameControl, #mac_header{seqnum = Seqnum}, <<>>};
decode_rest(FrameControl, Seqnum, Rest) ->
    [DestPan_, DestAddr, SrcPan_, SrcAddr, Payload] = lists:flatten(decode_addrs(dest_pan_id, Rest, FrameControl)),
    DestPan = case {DestPan_, FrameControl#frame_control.pan_id_compr, FrameControl#frame_control.frame_type} of
                  {<<>>, ?ENABLED, _} -> SrcPan_; % Can always deduce if the compression is enabled
                  {<<>>, ?DISABLED, ?FTYPE_ACK} -> <<>>; % if compression isn't enabled and ACK => can't deduce
                  {<<>>, ?DISABLED, ?FTYPE_BEACON} -> <<>>; % if compression isn't enabled and BEACON => can't deduce
                  {<<>>, ?DISABLED, _} -> SrcPan_; % Other wise destination is PAN coord with same PANID as SRC
                  {_, _, _} -> DestPan_
              end,
    SrcPan = case {SrcPan_, FrameControl#frame_control.pan_id_compr, FrameControl#frame_control.frame_type} of
                 {<<>>, ?ENABLED, _} -> DestPan;
                 {<<>>, ?DISABLED, ?FTYPE_ACK} -> <<>>; % if compression is disabled and frame type is an ACK => can't deduce (e.g. ACK comming from outside the PAN
                 {<<>>, ?DISABLED, _} -> DestPan;
                 {_, _, _} -> SrcPan_
             end,
    MacHeader = #mac_header{seqnum = Seqnum, dest_pan = DestPan, dest_addr = DestAddr, src_pan = SrcPan, src_addr = SrcAddr},
    {FrameControl, MacHeader, Payload}.


%
    %%-------------------------------------------------------------------------

% @private
% @doc decode the address fields present in the remaining sequence of bits based on the settings inside Framecontrol
%
% The first parameter is an atom representing the the field that
```

```
    should be parsed next
% @end
%
    ------------------------------------------------------------------------

decode_addrs(dest_pan_id, Rest, FrameControl) ->
    case FrameControl#frame_control.dest_addr_mode of
        ?NONE -> [<<>>, <<>>, decode_addrs(src_pan_id, Rest,
    FrameControl)];
        _ -> <<PanID:16/bitstring, Tail/bitstring>> = Rest,
             [reverse_byte_order(PanID), decode_addrs(dest_addr,
    Tail, FrameControl)]
    end;
decode_addrs(dest_addr, Rest, FrameControl) ->
    case FrameControl#frame_control.dest_addr_mode of
        ?SHORT_ADDR -> <<Addr:16/bitstring, Tail/bitstring>> =
    Rest,
                       [reverse_byte_order(Addr), decode_addrs(
    src_pan_id, Tail, FrameControl)];
        ?EXTENDED -> <<Addr:64/bitstring, Tail/bitstring>> = Rest,
                     [reverse_byte_order(Addr), decode_addrs(
    src_pan_id, Tail, FrameControl)];
        _ -> io:format("Frame control dest_addr: ~w~n", [
    FrameControl#frame_control.dest_addr_mode])
    end;
decode_addrs(src_pan_id, Rest, FrameControl) ->
    case {FrameControl#frame_control.pan_id_compr, FrameControl#
    frame_control.src_addr_mode} of
        {?ENABLED, _} -> [<<>>, decode_addrs(src_addr, Rest,
    FrameControl)];
        {_, ?NONE} -> [<<>>, <<>>, Rest];
        _ -> <<PanID:16/bitstring, Tail/bitstring>> = Rest, % If
    compr disabled and src_addr_mode isn't none
             [reverse_byte_order(PanID), decode_addrs(src_addr,
    Tail, FrameControl)]
    end;
decode_addrs(src_addr, Rest, FrameControl) ->
    case FrameControl#frame_control.src_addr_mode of
        ?NONE -> [<<>>, Rest];
        ?SHORT_ADDR -> <<Addr:16/bitstring, Payload/bitstring>> =
    Rest,
                       [reverse_byte_order(Addr), Payload];
        ?EXTENDED -> <<Addr:64/bitstring, Payload/bitstring>> =
    Rest,
                     [reverse_byte_order(Addr), Payload]
    end.

%
    ------------------------------------------------------------------------
```

```
180 % @private
181 % @doc Creates a MAC frame control
182 % @param FrameType: MAC frame type
183 % @param AR: ACK request
184 % @end
185 %
    ------------------------------------------------------------------------

186 -spec build_frame_control(FrameControl :: #frame_control{}) ->
    bitstring().
187 build_frame_control(FrameControl) ->
188     #frame_control{pan_id_compr=PanIdCompr,ack_req=AckReq,
    frame_pending=FramePending,sec_en=SecEn,
189                    frame_type=FrameType,src_addr_mode=SrcAddrMode,
    frame_version=FrameVersion,dest_addr_mode=DestAddrMode} =
    FrameControl,
190     <<2#0:1, PanIdCompr:1, AckReq:1, FramePending:1, SecEn:1,
    FrameType:3, SrcAddrMode:2, FrameVersion:2, DestAddrMode:2,
    2#0:2>>.
191
192
193 %
    ------------------------------------------------------------------------

194 % @private
195 % @doc Decode the frame control given in a bitstring form in the
    parameters
196 % @end
197 %
    ------------------------------------------------------------------------

198 -spec decode_frame_control(FC :: bitstring) -> #frame_control{}.
199 decode_frame_control(FC) ->
200     <<_:1, PanIdCompr:1, AckReq:1, FramePending:1, SecEn:1,
    FrameType:3, SrcAddrMode:2, FrameVersion:2, DestAddrMode:2, _
    :2>> = FC,
201     #frame_control{frame_type = FrameType, sec_en = SecEn,
    frame_pending = FramePending, ack_req = AckReq, pan_id_compr =
    PanIdCompr, dest_addr_mode = DestAddrMode, frame_version =
    FrameVersion, src_addr_mode = SrcAddrMode}.
202
203 %--- Tool functions
    ------------------------------------------------------------------

204
205 reverse_byte_order(Bitstring) -> reverse_byte_order(Bitstring,
    <<>>).
206 reverse_byte_order(<<Head:8>>, Acc) ->
207     <<Head:8, Acc/bitstring>>;
```

```
208 reverse_byte_order(<<Head:8, Tail/bitstring>>, Acc) ->
209     reverse_byte_order(Tail, <<Head:8, Acc/bitstring>>).
```

Listing B.2: mac_layer.erl

Appendix C

Examples

C.1 ack_no_jitter

```
% @doc robot public API.
-module(robot).

-behavior(application).

-include("mac_layer.hrl").

% Callbacks
-export([test_receiver_ack/0]).
-export([test_sender_ack/2]).
-export([start/2]).
-export([stop/1]).



%--- API ----------------------------------------------------------------


test_receiver_ack() ->
    pmod_uwb:write(panadr, #{pan_id => 16#BEEF, short_addr => 16#0002}),
    pmod_uwb:write(sys_cfg, #{ffad => 2#1, autoack => 2#1}), % allow ACK and data frame reception and enable autoack
    pmod_uwb:write(sys_cfg, #{ffen => 2#1}), % enable frame filtering and allow ACK frame reception and enable autoack
    receive_data_jitter().

% @doc Test the sender
% @param NbrFrames => The number of frames to send
```

```
% @param FrameSize => The size of the frame to send in bytes
test_sender_ack(NbrFrames, FrameSize) ->
    pmod_uwb:write(panadr, #{pan_id => 16#BEEF, short_addr => 16#0001}),
    pmod_uwb:write(rx_fwto, #{rxfwto => 16#FFFF}),
    pmod_uwb:write(sys_cfg, #{rxwtoe => 2#1}),
    #{short_addr := SrcAddr} = pmod_uwb:read(panadr),
    pmod_uwb:write(sys_cfg, #{ffaa => 2#1, autoack => 2#1}), % allow ACK and data frame reception and enable autoack
    pmod_uwb:write(sys_cfg, #{ffen => 2#1}), % enable frame filtering and allow ACK frame reception and enable autoack
    Data = <<0:(FrameSize*8)>>,
    Start = os:timestamp(),
    {Success, Error, Total} = send_data_wait_ack(0, NbrFrames, SrcAddr, 10, 10, Data, {0, 0, 0}),
    End = os:timestamp(),
    Time = timer:now_diff(End, Start)/1000000,
    io:format("------------------- Report --------------------~n"),
    io:format("Sent ~w frames - Success rate ~.3f (~w/~w) - Error rate ~.3f (~w/~w)~n", [Total, Success/Total, Success, Total, Error/Total, Error, Total]),
    io:format("Data rate ~.1f b/s - In ~w s ~n", [(bit_size(Data)*NbrFrames)/Time, Time]),
    io:format("-----------------------------------------------~n").

%--- Private ----------------------------------------------------------------

receive_data_jitter() ->
    case mac_layer:mac_receive(false) of
        {#frame_control{frame_type = ?FTYPE_DATA, pan_id_compr = ?ENABLED} = _FrameControl, MacHeader, _Data} ->
            io:format("Received data from ~w with seqnum ~w~n", [MacHeader#mac_header.src_addr, MacHeader#mac_header.seqnum]),
            pmod_uwb:wait_for_transmission();
            % pmod_uwb:write(sys_status, #{txfrs => 2#1});
        {_, _, _} -> io:format("Received unexpected frame~n");
        Err -> io:format("Reception error: ~w~n", [Err])
    end,
    receive_data_jitter().
%
%
%----------------------------------------------------------------------------

% @private
% @param Cnt: number of MAC message already sent
% @param Max: total number of MAC message to send
% @param SrcAddr: the address of the device sending the MAC message
```

```
61 % @param TrialsLeft: the number of reception attempts left
62 % @param TotTrialsAllowed: the maximum number of times we will try
       to receive a frame after a bad reception
63 % @TODO use RXAUTR later on
64 %
   ------------------------------------------------------------------------

65 -spec send_data_wait_ack(Cnt :: integer(), Max :: integer(),
   SrcAddr :: integer(), TrialsLeft :: integer(), TotTrialsAllowed
    :: integer(), Data :: bitstring(), _Stats) -> ok | {error, any
   ()}.
66 send_data_wait_ack(_, _, _, 0, _, _, Stats) -> error({
   reception_error, "Max trials reached", Stats});
67 send_data_wait_ack(Max, Max, _, _, _, _, Stats) -> Stats;
68 send_data_wait_ack(Cnt, Max, SrcAddr, TrialsLeft, TotTrialsAllowed
   , Data, {Success, Error, TotalFrameSent}) ->
69     Seqnum = Cnt rem 16#FF,
70     FrameControl = #frame_control{ack_req = ?ENABLED, pan_id_compr
    = ?ENABLED},
71     MacHeader = #mac_header{seqnum = Seqnum, dest_pan = <<16#BEEF
   :16>>, dest_addr = <<16#0002:16>>, src_addr = <<SrcAddr:16>>},
72     % io:format("Sending message #~w with seqnum ~w~n", [Cnt,
   Seqnum]),
73     mac_layer:mac_send_data(FrameControl, MacHeader, Data, #
   tx_opts{wait4resp = ?ENABLED, w4r_tim = 0}),
74     case  mac_layer:mac_receive(true) of
75         {#frame_control{frame_type = ?FTYPE_ACK} = _RxFrameControl
   , #mac_header{seqnum = Seqnum} = _RxMacHeader, _RxData} -> % io
   :format("ACK received for frame seqnum ~w~n", [_RxMacHeader#
   mac_header.seqnum]),
76
   send_data_wait_ack(Cnt+1, Max, SrcAddr, TotTrialsAllowed,
   TotTrialsAllowed, Data, {Success+1, Error, TotalFrameSent+1});
77         {_RxFrameControl, _RxMacHeader, RxData} -> io:format("
   Received MAC frame but not ACK: ~w~n", [RxData]),
78
   send_data_wait_ack(Cnt, Max, SrcAddr, TrialsLeft,
   TotTrialsAllowed, Data, {Success, Error, TotalFrameSent});
79         _ -> io:format("Reception error. Trying again...~n"),
80                 pmod_uwb:write(sys_status, #{rxfto => 2#1}), %
   reset rxfto to avoid false t.o.
81                 send_data_wait_ack(Cnt, Max, SrcAddr, TrialsLeft-1,
    TotTrialsAllowed, Data, {Success, Error+1, TotalFrameSent+1})
82     end.
83 %--- Callbacks
   ------------------------------------------------------------------

84
```

```
85 % @private
86 start(_Type, _Args) ->
87     {ok, Supervisor} = robot_sup:start_link(),
88     grisp:add_device(spi2, pmod_uwb),
89     % Res = pmod_uwb:read(dev_id),
90     {ok, Supervisor}.
91
92 % @private
93 stop(_State) -> ok.
```

Listing C.1: main code for the example ack_no_jitter

C.2 ack_jitter

```
1  % @doc robot public API.
2  -module(robot).
3
4  -behavior(application).
5
6  -include("mac_layer.hrl").
7
8  % Callbacks
9  -export([test_receiver_ack/0]).
10 -export([test_sender_ack/2]).
11 -export([start/2]).
12 -export([stop/1]).
13
14
15 %--- API ----------------------------------------------------------------

16 test_receiver_ack() ->
17     pmod_uwb:write(panadr, #{pan_id => 16#BEEF, short_addr =>
       16#0002}),
18     pmod_uwb:write(sys_cfg, #{ffad => 2#1, autoack => 2#1}), %
       allow ACK and data frame reception and enable autoack
19     pmod_uwb:write(sys_cfg, #{ffen => 2#1}), % enable frame
       filtering and allow ACK frame reception and enable autoack
20     receive_data_jitter().
21
22 % @doc Test the sender
23 % @param NbrFrames => The number of frames to send
24 % @param FrameSize => The size of the frame to send in bytes
25 test_sender_ack(NbrFrames, FrameSize) ->
26     pmod_uwb:write(panadr, #{pan_id => 16#BEEF, short_addr =>
       16#0001}),
27     pmod_uwb:write(rx_fwto, #{rxfwto => 16#FFFF}),
```

```
28      pmod_uwb:write(sys_cfg, #{rxwtoe => 2#1}),
29      #{short_addr := SrcAddr} = pmod_uwb:read(panadr),
30      pmod_uwb:write(sys_cfg, #{ffaa => 2#1, autoack => 2#1}), % allow ACK and data frame reception and enable autoack
31      pmod_uwb:write(sys_cfg, #{ffen => 2#1}), % enable frame filtering and allow ACK frame reception and enable autoack
32      Data = <<0:(FrameSize*8)>>,
33      Start = os:timestamp(),
34      {Success, Error, Total} = send_data_wait_ack(0, NbrFrames, SrcAddr, 10, 10, Data, {0, 0, 0}),
35      End = os:timestamp(),
36      Time = timer:now_diff(End, Start)/1000000,
37      io:format("------------------- Report --------------------~n"),
38      io:format("Sent ~w frames - Success rate ~.3f (~w/~w) - Error rate ~.3f (~w/~w)~n", [Total, Success/Total, Success, Total, Error/Total, Error, Total]),
39      io:format("Data rate ~.1f b/s - In ~w s ~n", [(bit_size(Data)*NbrFrames)/Time, Time]),
40      io:format("-----------------------------------------------~n").
41
42  %--- Private ----------------------------------------------------------------
43  receive_data_jitter() ->
44      case mac_layer:mac_receive(false) of
45          {#frame_control{frame_type = ?FTYPE_DATA, pan_id_compr = ?ENABLED} = _FrameControl, MacHeader, Data} ->
46              io:format("Received data from ~w with seqnum ~w~n", [MacHeader#mac_header.src_addr, MacHeader#mac_header.seqnum]),
47              % Simulates some delay in the network for every frame out of 4
48              case rand:uniform(4) of
49                  1 -> timer:sleep(200);
50                  _ -> ok
51              end,
52              pmod_uwb:wait_for_transmission();
53              % pmod_uwb:write(sys_status, #{txfrs => 2#1});
54          {_, _, _} -> io:format("Received unexpected frame~n");
55          Err -> io:format("Reception error: ~w~n", [Err])
56      end,
57      receive_data_jitter().
58
59  %% ----------------------------------------------------------------------------
60  %% @private
61  %% @param Cnt: number of MAC message already sent
62  %% @param Max: total number of MAC message to send
63  %% @param SrcAddr: the address of the device sending the MAC
```

```
    message
% @param TrialsLeft: the number of reception attempts left
% @param TotTrialsAllowed: the maximum number of times we will try to receive a frame after a bad reception
%
    ----------------------------------------------------------------------

-spec send_data_wait_ack(Cnt :: integer(), Max :: integer(), SrcAddr :: integer(), TrialsLeft :: integer(), TotTrialsAllowed :: integer(), Data :: bitstring(), _Stats) -> ok | {error, any()}.
send_data_wait_ack(_, _, _, 0, _, _, Stats) -> error({reception_error, "Max trials reached", Stats});
send_data_wait_ack(Max, Max, _, _, _, _, Stats) -> Stats;
send_data_wait_ack(Cnt, Max, SrcAddr, TrialsLeft, TotTrialsAllowed, Data, {Success, Error, TotalFrameSent}) ->
    Seqnum = Cnt rem 16#FF,
    FrameControl = #frame_control{ack_req = ?ENABLED, pan_id_compr = ?ENABLED},
    MacHeader = #mac_header{seqnum = Seqnum, dest_pan = <<16#BEEF:16>>, dest_addr = <<16#0002:16>>, src_addr = <<SrcAddr:16>>},
    % io:format("Sending message #~w with seqnum ~w~n", [Cnt, Seqnum]),
    mac_layer:mac_send_data(FrameControl, MacHeader, Data, #tx_opts{wait4resp = ?ENABLED, w4r_tim = 0}),
    case  mac_layer:mac_receive(true) of
        {#frame_control{frame_type = ?FTYPE_ACK} = _RxFrameControl, #mac_header{seqnum = Seqnum} = _RxMacHeader, _RxData} -> % io:format("ACK received for frame seqnum ~w~n", [_RxMacHeader#mac_header.seqnum]),

    send_data_wait_ack(Cnt+1, Max, SrcAddr, TotTrialsAllowed, TotTrialsAllowed, Data, {Success+1, Error, TotalFrameSent+1});
        {_RxFrameControl, _RxMacHeader, RxData} -> io:format("Received MAC frame but not ACK: ~w~n", [RxData]),

    send_data_wait_ack(Cnt, Max, SrcAddr, TrialsLeft, TotTrialsAllowed, Data, {Success, Error, TotalFrameSent});
        _ -> io:format("Reception error. Trying again...~n"),
             pmod_uwb:write(sys_status, #{rxfto => 2#1}), % reset rxfto to avoid false t.o.
             send_data_wait_ack(Cnt, Max, SrcAddr, TrialsLeft-1, TotTrialsAllowed, Data, {Success, Error+1, TotalFrameSent+1})
    end.

%--- Callbacks
    ----------------------------------------------------------------------
```

```
87
88 % @private
89 start(_Type, _Args) ->
90     {ok, Supervisor} = robot_sup:start_link(),
91     grisp:add_device(spi2, pmod_uwb),
92     % Res = pmod_uwb:read(dev_id),
93     {ok, Supervisor}.
94
95 % @private
96 stop(_State) -> ok.
```

Listing C.2: main code for the example ack_jitter

C.3 ack_fast_tx

```
1  % @doc robot public API.
2  -module(robot).
3
4  -behavior(application).
5
6  -include("mac_layer.hrl").
7
8  % Callbacks
9  -export([test_receiver_ack/0]).
10 -export([test_sender_ack/2]).
11 -export([start/2]).
12 -export([stop/1]).
13
14
15 %--- API ----------------------------------------------------------------

16 test_receiver_ack() ->
17     pmod_uwb:write(panadr, #{pan_id => 16#BEEF, short_addr =>
       16#0002}),
18     pmod_uwb:write(sys_cfg, #{ffad => 2#1, autoack => 2#1}), %
       allow ACK and data frame reception and enable autoack
19     pmod_uwb:write(sys_cfg, #{ffen => 2#1}), % enable frame
       filtering and allow ACK frame reception and enable autoack
20     receive_data_jitter().
21
22 % @doc Test the sender
23 % @param NbrFrames => The number of frames to send
24 % @param FrameSize => The size of the frame to send in bytes
25 test_sender_ack(NbrFrames, FrameSize) ->
26     pmod_uwb:write(panadr, #{pan_id => 16#BEEF, short_addr =>
       16#0001}),
```

```
27      pmod_uwb:write(rx_fwto, #{rxfwto => 16#FFFF}),
28      pmod_uwb:write(sys_cfg, #{rxwtoe => 2#1}),
29      #{short_addr := SrcAddr} = pmod_uwb:read(panadr),
30      pmod_uwb:write(sys_cfg, #{ffaa => 2#1, autoack => 2#1}), %
        allow ACK and data frame reception and enable autoack
31      pmod_uwb:write(sys_cfg, #{ffen => 2#1}), % enable frame
        filtering and allow ACK frame reception and enable autoack
32      Data = <<0:(FrameSize*8)>>,
33      FrameControl = #frame_control{ack_req = ?ENABLED, pan_id_compr
        = ?ENABLED},
34      MacHeader = #mac_header{seqnum = 254, dest_pan = <<16#BEEF
        :16>>, dest_addr = <<16#0002:16>>, src_addr = <<SrcAddr:16>>},
35      MacFrame = mac_layer:mac_frame(FrameControl, MacHeader, Data),
36      pmod_uwb:write_tx_data(MacFrame),
37      Start = os:timestamp(),
38      {Success, Error, Total} = send_data_wait_ack(0, NbrFrames,
        SrcAddr, 10, 10, byte_size(MacFrame), {0, 0, 0}),
39      End = os:timestamp(),
40      Time = timer:now_diff(End, Start)/1000000,
41      io:format("------------------- Report --------------------~n"),
42      io:format("Sent ~w frames - Success rate ~.3f (~w/~w) - Error
        rate ~.3f (~w/~w)~n", [Total, Success/Total, Success, Total,
        Error/Total, Error, Total]),
43      io:format("Data rate ~.1f b/s - In ~w s ~n", [(bit_size(Data)*
        NbrFrames)/Time, Time]),
44      io:format("-----------------------------------------------~n").
45
46  %--- Private
        ----------------------------------------------------------------

47  receive_data_jitter() ->
48      case mac_layer:mac_receive(false) of
49          {#frame_control{frame_type = ?FTYPE_DATA, pan_id_compr = ?
        ENABLED} = _FrameControl, MacHeader, _Data} ->
50              pmod_uwb:wait_for_transmission();
51              % pmod_uwb:write(sys_status, #{txfrs => 2#1});
52          {_, _, _} -> io:format("Received unexpected frame~n");
53          Err -> io:format("Reception error: ~w~n", [Err])
54      end,
55      receive_data_jitter().
56
57  %
        ----------------------------------------------------------------------------

58  % @private
59  % @param Cnt: number of MAC message already sent
60  % @param Max: total number of MAC message to send
61  % @param SrcAddr: the address of the device sending the MAC
        message
```

```
% @param TrialsLeft: the number of reception attempts left
% @param TotTrialsAllowed: the maximum number of times we will try to receive a frame after a bad reception
% @TODO use RXAUTR later on
%
%%%---------------------------------------------------------------------------

-spec send_data_wait_ack(Cnt :: integer(), Max :: integer(), SrcAddr :: integer(), TrialsLeft :: integer(), TotTrialsAllowed :: integer(), Data :: bitstring(), _Stats) -> ok | {error, any()}.
send_data_wait_ack(_, _, _, 0, _, _, Stats) -> error({reception_error, "Max trials reached", Stats});
send_data_wait_ack(Max, Max, _, _, _, _, Stats) -> Stats;
send_data_wait_ack(Cnt, Max, SrcAddr, TrialsLeft, TotTrialsAllowed, DataLength, {Success, Error, TotalFrameSent}) ->
    % Starting the transmission
    pmod_uwb:write(tx_fctrl, #{txboffs => 2#0, tr => 2#0, tflen => DataLength}),
    pmod_uwb:write(sys_ctrl, #{txstrt => 2#1, wait4resp => ?ENABLED, txdlys => 0}), % start transmission and some options
    case  mac_layer:mac_receive(true) of
        {#frame_control{frame_type = ?FTYPE_ACK} = _RxFrameControl, #mac_header{seqnum = Seqnum} = _RxMacHeader, _RxData} ->
    send_data_wait_ack(Cnt+1, Max, SrcAddr, TotTrialsAllowed, TotTrialsAllowed, DataLength, {Success+1, Error, TotalFrameSent+1});
        {_RxFrameControl, _RxMacHeader, RxData} ->
    send_data_wait_ack(Cnt, Max, SrcAddr, TrialsLeft, TotTrialsAllowed, DataLength, {Success, Error, TotalFrameSent})
    ;
        _ -> io:format("Reception error. Trying again...~n"),
             pmod_uwb:write(sys_status, #{rxfto => 2#1}), % reset rxfto to avoid false t.o.
             send_data_wait_ack(Cnt, Max, SrcAddr, TrialsLeft-1, TotTrialsAllowed, DataLength, {Success, Error+1, TotalFrameSent+1})
    end.

%--- Callbacks
%%%---------------------------------------------------------------------------


% @private
start(_Type, _Args) ->
    {ok, Supervisor} = robot_sup:start_link(),
    grisp:add_device(spi2, pmod_uwb),
    % Res = pmod_uwb:read(dev_id),
    {ok, Supervisor}.
```

```
89
90 % @private
91 stop(_State) -> ok.
```

Listing C.3: main code for the ack_fast_tx example

C.4 ss_twr

```
1  % @doc robot public API.
2  -module(robot).
3
4  -behavior(application).
5
6  -include("mac_layer.hrl").
7
8  % Callbacks
9  -export([start/2]).
10 -export([stop/1]).
11 -export([ss_initiator/0, ss_responder/0]).
12
13 %-define(TU, 1.565444993393822e-11). % 1 t.u. is ~1.5654e-11 s
14 -define(TU, 15.65e-12).
15 -define(C, 299792458). % Speed of light
16 % https://forum.qorvo.com/t/sample-programs/788/3
17 -define(UUS_TO_DWT_TIME, 65536). % in one UWB  s , there are 65536
       t.u (UWB  s  are special  s  ???)
18 -define(FREQ_OFFSET_MULTIPLIER, 1/( 131072 * 2 * (1024/998.4e6))).
19 -define(HERTZ_TO_PPM_MUL, 1.0e-6/6489.6e6).
20
21 -define(NBR_MEASUREMENTS, 250).
22 -define(TX_ANTD, 23500).
23 -define(RX_ANTD, 23500).
24
25 %--- Single-sided two-way ranging
       ----------------------------------------------------------------------


26
27 ss_initiator() ->
28     pmod_uwb:write(tx_antd, #{tx_antd => ?TX_ANTD}), % ! this
       value is not correct - the devices should be calibrated
29     pmod_uwb:write(lde_if, #{lde_rxantd => ?RX_ANTD}),
30     ss_initiator(?NBR_MEASUREMENTS, []).
31
32 ss_initiator(0, Measurements) ->
33     Total = length(Measurements),
34     MeasureAVG = lists:sum(Measurements)/Total,
35     StdDev = std_dev(Measurements, MeasureAVG, Total, 0),
```

```
36      io:format("-------------------------------- Summary
        --------------------------------~n"),
37      io:format("Sent ~w request -~n",[Total]),
38      io:format("Average distance measured: ~w - standard deviation:
        ~w ~n", [MeasureAVG, StdDev]),
39      io:format("Min: ~w - Max ~w~n", [lists:min(Measurements),
        lists:max(Measurements)]),
40      io:format("
        ------------------------------------------------------------------------~
        n"),
41      Measurements;
42  ss_initiator(N, Measurements) ->
43      Measure = ss_initiator_loop(),
44      io:format("~w~n", [Measure]),
45      timer:sleep(100),
46      case Measure of
47          error -> ss_initiator(N-1, Measurements);
48          _ -> ss_initiator(N-1, [Measure | Measurements])
49      end.
50
51  ss_initiator_loop() ->
52      % Builds the MAC frame for Poll message
53      FrameControl = #frame_control{pan_id_compr = ?ENABLED,
        dest_addr_mode = ?SHORT_ADDR, src_addr_mode = ?SHORT_ADDR},
54      MacHeader = #mac_header{seqnum = 0, dest_pan = <<16#FFFF:16>>,
        dest_addr = <<16#FFFF:16>>, src_addr = <<16#FFFF:16>>},
55      Options = #tx_opts{wait4resp = ?ENABLED, w4r_tim = 0},
56
57      mac_layer:mac_send_data(FrameControl, MacHeader, <<"GRiSP">>,
        Options),
58
59      {_, _, Data} = mac_layer:mac_receive(true), % Reception of
        Resp
60      %io:format("Received data: ~w~n", [Data]),
61
62      % Getting the timestamps of the TX of Poll and of the RX of
        Resp
63      #{tx_stamp := PollTXTimestamp} = pmod_uwb:read(tx_time),
64      #{rx_stamp := RespRXTimestamp} = pmod_uwb:read(rx_time),
65
66      {PollRXTimestamp, RespTXTimestamp} = get_resp_ts(Data), %
        Getting the timestamps sent by the responder
67      % io:format("PollTX: ~w - PollRX: ~w - RespTX: ~w - RespRX: ~w
        ~n", [PollTXTimestamp, PollRXTimestamp, RespTXTimestamp,
        RespRXTimestamp]),
68
69      TRound = RespRXTimestamp - PollTXTimestamp,
70      TResp = RespTXTimestamp - PollRXTimestamp,
71
```

```
72      % Getting the clock offset ratio
73      #{drx_car_int := DRX_CAR_INT} = pmod_uwb:read(drx_conf),
74      ClockOffsetRatio = (DRX_CAR_INT * ?FREQ_OFFSET_MULTIPLIER *  ?
        HERTZ_TO_PPM_MUL),
75
76      io:format("PollTX ~w - RespRX ~w - PollRX ~w - RespTX ~w~n", [
        PollTXTimestamp, RespRXTimestamp, PollRXTimestamp,
        RespTXTimestamp]),
77      TimeOfFlight = ( (TRound - TResp) * ((1-ClockOffsetRatio)/2) )
         * ?TU,
78      if
79          (RespRXTimestamp >= PollTXTimestamp) and (RespTXTimestamp
        >= PollRXTimestamp) -> TimeOfFlight * ?C;
80          true -> error
81      end.
82
83  get_resp_ts(Data) ->
84      <<PollRXTimestamp:40, RespTXTimestamp:40>> = Data,
85      {PollRXTimestamp, RespTXTimestamp}.
86
87  ss_responder() ->
88      pmod_uwb:write(tx_antd, #{tx_antd => ?TX_ANTD}),
89      pmod_uwb:write(lde_if, #{lde_rxantd => ?RX_ANTD}),
90      #{pan_id := PANID, short_addr := Addr} = pmod_uwb:read(panadr)
        ,
91      ss_responder_loop(?NBR_MEASUREMENTS, PANID, Addr).
92
93  ss_responder_loop(0, _, _) -> ok;
94  ss_responder_loop(N, PANID, Addr) ->
95      {FrameControl, MacHeader, _} = mac_layer:mac_receive(),
96      #{rx_stamp := PollRXTimestamp} = pmod_uwb:read(rx_time),
97
98      RespTXTimestamp_ = PollRXTimestamp + (20000 * ?UUS_TO_DWT_TIME
        ),
99      pmod_uwb:write(dx_time, #{dx_time => RespTXTimestamp_}),
100
101     RespTXTimestamp = RespTXTimestamp_ + ?TX_ANTD,
102     TXData = <<PollRXTimestamp:40, RespTXTimestamp:40>>,
103     TXMacHeader = #mac_header{seqnum = MacHeader#mac_header.seqnum
        +1, dest_pan = MacHeader#mac_header.src_pan, dest_addr =
        MacHeader#mac_header.src_addr, src_pan = <<PANID:16>>, src_addr
         = <<Addr:16>>},
104     Options = #tx_opts{txdlys = ?ENABLED, tx_delay =
        RespTXTimestamp},
105
106     mac_layer:mac_send_data(FrameControl, TXMacHeader, TXData,
        Options),
107     ss_responder_loop(N-1, PANID, Addr).
108
```

```
%--- Tool functions for stats
    ------------------------------------------------------------------


-spec std_dev(Measures :: list(), Mean :: number(), N :: number(),
    Acc :: number()) -> number().
std_dev([], _, N, Acc) ->
    math:sqrt(Acc/N);
std_dev([H | T], Mean, N, Acc) ->
    std_dev(T, Mean, N, Acc + math:pow(H-Mean, 2)).


%--- Callbacks
    ------------------------------------------------------------------


% @private
start(_Type, _Args) ->
    {ok, Supervisor} = robot_sup:start_link(),
    grisp:add_device(spi2, pmod_uwb),
    % Res = pmod_uwb:read(dev_id),
    {ok, Supervisor}.

% @private
stop(_State) -> ok.
```

Listing C.4: main code for the example ss_twr

C.5 ds_twr

```
% @doc robot public API.
-module(robot).

-behavior(application).

-include("mac_layer.hrl").

-export([ds_initiator/0, ds_responder/0]).
%
% Callbacks
-export([start/2]).
-export([stop/1]).

%-define(TU, 1.565444993393822e-11). % 1 t.u. is ~1.5654e-11 s
-define(TU, 15.65e-12).
-define(C, 299792458). % Speed of light
% https://forum.qorvo.com/t/sample-programs/788/3
```

```
18 -define(UUS_TO_DWT_TIME, 65536). % in one UWB  s , there are 65536
       t.u (UWB  s  are special  s  ???)
19
20 -define(NBR_MEASUREMENTS, 250).
21 -define(TX_ANTD, 16450).
22 -define(RX_ANTD, 16450).
23
24 %--- Double-sided two-way ranging
       ----------------------------------------------------------------------

25 ds_initiator() ->
26     % Set the antenna delay -> !! the values should be calibrated
27     pmod_uwb:write(tx_antd, #{tx_antd => ?TX_ANTD}),
28     pmod_uwb:write(lde_if, #{lde_rxantd => ?RX_ANTD}),
29     pmod_uwb:set_frame_timeout(16#FFFF),
30     ds_initiator_loop(?NBR_MEASUREMENTS, {0,0,[],0}).
31
32 ds_initiator_loop(0, {Succeeded, Errors, _Measures, Total}) ->
33     SuccessRate = Succeeded / Total,
34     ErrorRate = Errors / Total,
35     io:format("---------------------------------- Summary
       ----------------------------------~n"),
36     io:format("Sent ~w request - ratio: ~w/~w - Success rate: ~w -
       Error rate: ~w~n",[Total, Succeeded, Total, SuccessRate,
       ErrorRate]),
37     io:format("
       ----------------------------------------------------------------------------~
       n");
38 ds_initiator_loop(Left, {Succeeded, Errors, Measures, Total}) ->
39     case ds_initiator_protocol() of
40         ok -> ds_initiator_loop(Left - 1, {Succeeded+1, Errors,
       Measures, Total+1});
41         error -> ds_initiator_loop(Left-1, {Succeeded, Errors+1,
       Measures, Total+1}) % No response to Poll message -> try again
42     end.
43
44 ds_initiator_protocol() ->
45     % Sending the poll message
46     FrameControl = #frame_control{pan_id_compr = ?ENABLED},
47     MacHeader = #mac_header{seqnum = 0, dest_pan = <<16#FFFF:16>>,
       dest_addr = <<16#FFFF:16>>, src_addr = <<16#FFFF:16>>},
48     % enabling wait4resp to avoid an early timeout. Here we know
       that resp will take at least 10000  s
49     mac_layer:mac_send_data(FrameControl, MacHeader, <<"DS_INIT"
       >>, #tx_opts{wait4resp = ?ENABLED, w4r_tim = 20000}),
50     io:format("Poll message is sent~n"),
51
52     % Receiving the resp message
53     case  mac_layer:mac_receive(true) of
```

```
54          {_, _, <<"Resp_TX">>} -> #{tx_stamp := PollTXTimestamp} =
    pmod_uwb:read(tx_time),
55                          #{rx_stamp := RespRXTimestamp} = pmod_uwb
    :read(rx_time), % Getting the reception timestamp of the resp
    message
56
57                          % Setting up the final message
58                          FinalTXTime = RespRXTimestamp + (30000 *
    ?UUS_TO_DWT_TIME),
59                          pmod_uwb:write(dx_time, #{dx_time =>
    FinalTXTime}),
60                          FinalTXTimestamp = FinalTXTime + ?TX_ANTD
    ,
61                          Message = <<PollTXTimestamp:40,
    RespRXTimestamp:40, FinalTXTimestamp:40>>,
62                          pmod_uwb:write(sys_status, #{txfcg => 2#1
    }),
63                          % Sending the final message
64                          mac_layer:mac_send_data(FrameControl,
    MacHeader, Message, #tx_opts{txdlys = ?ENABLED, tx_delay =
    FinalTXTime}),
65                          io:format("Final message sent~n"),
66                          io:format("PollTX: ~w - RespRX ~w -
    FinalTX ~w~n", [PollTXTimestamp, RespRXTimestamp,
    FinalTXTimestamp]),
67                          io:format("Data sent~n"),
68                          timer:sleep(100);
69          Err -> io:format("Reception error: ~w~n", [Err]),
70                 error
71      end.
72
73
74  ds_responder() ->
75      % Set the antenna delay -> !! the values should be calibrated
76      pmod_uwb:write(tx_antd, #{tx_antd => ?TX_ANTD}),
77      pmod_uwb:write(lde_if, #{lde_rxantd => ?RX_ANTD}),
78      Measures = ds_responder_loop(?NBR_MEASUREMENTS, {0, 0, [], 0})
    ,
79      io:format("~w~n", [Measures]).
80
81  ds_responder_loop(0, {Succeeded, Errors, Measures, Total}) ->
82      SuccessRate = Succeeded/Total,
83      ErrorRate = Errors/Total,
84      MeasureAVG = lists:sum(Measures)/Succeeded,
85      StdDev = std_dev(Measures, MeasureAVG, Succeeded, 0),
86      io:format("-------------------------------- Summary
    --------------------------------~n"),
87      io:format("Received ~w request - ratio: ~w/~w - Success rate:
    ~w - Error rate: ~w~n",[Total, Succeeded, Total, SuccessRate,
```

```
    ErrorRate]),
88      io:format("Average distance measured: ~w - standard deviation:
        ~w ~n", [MeasureAVG, StdDev]),
89      io:format("Min: ~w - Max ~w~n", [lists:min(Measures), lists:
    max(Measures)]),
90      io:format("
    ----------------------------------------------------------------------~
    n"),
91      Measures;
92  ds_responder_loop(N, {Succeeded, Errors, Measures, Total}) ->
93      pmod_uwb:write(sys_cfg, #{rxwtoe  => 2#0}),
94      case ds_responder_protocol() of
95          error_rx_poll -> ds_responder_loop(N-1, {Succeeded, Errors
    +1, Measures, Total+1});
96          error -> ds_responder_loop(N-1, {Succeeded, Errors+1,
    Measures, Total+1});
97          Distance -> ds_responder_loop(N-1, {Succeeded+1, Errors, [
    Distance | Measures], Total+1})
98      end.
99
100 ds_responder_protocol() ->
101     % Receiving poll message
102     case mac_layer:mac_receive() of
103         {FrameControl, MacHeader, <<"DS_INIT">>} ->
104             io:format("Poll message received~n"),
105             pmod_uwb:set_frame_timeout(16#FFFF),
106             #{rx_stamp := PollRXTimestamp} = pmod_uwb:read(rx_time
    ), % Getting the the reception timestamp of the poll message
107
108             % Setting up and sending the resp message
109             RespTXTime = PollRXTimestamp + (30000 * ?
    UUS_TO_DWT_TIME),
110             pmod_uwb:write(dx_time, #{dx_time => RespTXTime}),
111             RespMacHeader = #mac_header{src_addr = <<16#FFFF:16>>,
     dest_pan = MacHeader#mac_header.src_pan, dest_addr = MacHeader
    #mac_header.src_addr, seqnum = MacHeader#mac_header.seqnum},
112             mac_layer:mac_send_data(FrameControl, RespMacHeader,
    <<"Resp_TX">>, #tx_opts{wait4resp = ?ENABLED, w4r_tim = 20000})
    ,
113             io:format("Response message sent~n"),
114
115             % Receiving the final message
116             case mac_layer:mac_receive(true)of
117                 {_, _, <<PollTXTimestamp:40, RespRXTimestamp:40,
    FinalTXTimestamp:40>>} ->
118                     #{tx_stamp := RespTXTimestamp} = pmod_uwb:read
    (tx_time), % Getting the tx timestamp of the resp message
119                     #{rx_stamp := FinalRXTimestamp} = pmod_uwb:
    read(rx_time), % Getting the rx timestamp of the final message
```

```
120
121                     TRound1 = RespRXTimestamp - PollTXTimestamp,
122                     TRound2 = FinalRXTimestamp - RespTXTimestamp,
123                     TReply1 = RespTXTimestamp - PollRXTimestamp,
124                     TReply2 = FinalTXTimestamp - RespRXTimestamp,
125
126                     TProp = (TRound1 * TRound2 - TReply1 * TReply2
    )/(TRound1 + TRound2 + TReply1 + TReply2),
127                     io:format("TProp: ~w~n", [TProp]),
128                     TOF = TProp * ?TU,
129                     Distance = TOF * ?C,
130
131                     io:format("PollRX: ~w - RespTX ~w - FinalRX ~w
    ~n", [PollRXTimestamp, RespTXTimestamp, FinalRXTimestamp]),
132                     io:format("TRound1: ~w - TRound2 ~w - TReply1
    ~w - TReply2 ~w ~n", [TRound1, TRound2, TReply1, TReply2]),
133                     io:format("Computed distance: ~w~n", [Distance
    ]),
134                     if
135                         (PollTXTimestamp =< RespRXTimestamp) and (
    RespRXTimestamp =< FinalTXTimestamp) and (PollRXTimestamp =<
    RespTXTimestamp) and (RespTXTimestamp =< FinalRXTimestamp) ->
    Distance;
136                         true -> io:format("Small error~n"), error
    % There was a wrap around in the clock of one of the GRIP -
    Throw away the result
137                     end;
138                 Err -> io:format("Reception error: ~w~n", [Err]),
139                        error
140             end;
141         Err -> io:format("Receiving error: ~w~n", [Err]),
142                error_rx_poll
143     end.
144
145 %--- Tool functions for stats
    ----------------------------------------------------------------------


146
147 -spec std_dev(Measures :: list(), Mean :: number(), N :: number(),
     Acc :: number()) -> number().
148 std_dev([], _, N, Acc) ->
149     math:sqrt(Acc/N);
150 std_dev([H | T], Mean, N, Acc) ->
151     std_dev(T, Mean, N, Acc + math:pow(H-Mean, 2)).
152
153 %--- Callbacks
    -----------------------------------------------------------------


154
```

```
155 % @private
156 start(_Type, _Args) ->
157     {ok, Supervisor} = robot_sup:start_link(),
158     grisp:add_device(spi2, pmod_uwb),
159     % Res = pmod_uwb:read(dev_id),
160     {ok, Supervisor}.
161
162 % @private
163 stop(_State) -> ok.
```

Listing C.5: main code for the example ds_twr

Appendix D

MAC layer unit tests

```
1  -module(mac_layer_tests).
2
3  -include_lib("eunit/include/eunit.hrl").
4
5  -include("../src/mac_layer.hrl").
6
7  %--- Setup
       ----------------------------------------------------------------------
8
9  %--- Tests
       ----------------------------------------------------------------------
10
11 mac_message_from_api_test() ->
12     FrameControl = #frame_control{ack_req = ?ENABLED, pan_id_compr
       = ?ENABLED, frame_version = 2#00},
13     MacHeader = #mac_header{seqnum = 0, dest_pan = <<16#DECA:16>>,
       dest_addr = <<"RX">>, src_addr = <<"TX">>},
14     ?assertEqual(<<16#6188:16, 0:8, 16#CADE:16, "XR", "XT", "Hello
       ">>,
15                  mac_layer:mac_message(FrameControl, MacHeader, <<
       "Hello">>)).
16
17 mac_message_pan_id_not_compressed_test() ->
18     FrameControl = #frame_control{ack_req = ?ENABLED, pan_id_compr
       = ?DISABLED, frame_version = 2#00},
19     MacHeader = #mac_header{seqnum = 0, dest_pan = <<16#DECA:16>>,
       dest_addr = <<"RX">>, src_pan = <<16#DECA:16>>, src_addr = <<"
       TX">>},
20     ?assertEqual(<<16#2188:16, 0:8, 16#CADE:16, "XR", 16#CADE:16,
       "XT", "Hello">>,
21                  mac_layer:mac_message(FrameControl, MacHeader, <<
```

```
    "Hello">>)).

mac_message_broadcast_test() ->
    FrameControl = #frame_control{ack_req = ?ENABLED, pan_id_compr
    = ?DISABLED, frame_version = 2#00},
    MacHeader = #mac_header{seqnum = 0, dest_pan = <<16#FFFF:16>>,
    dest_addr = <<16#FFFF:16>>, src_pan = <<16#FFFF:16>>, src_addr
    = <<16#FFFF:16>>},
    ?assertEqual(<<16#2188:16, 0:8, 16#FFFF:16, 16#FFFF:16, 16#
    FFFF:16, 16#FFFF:16, "Hello">>,
                 mac_layer:mac_message(FrameControl, MacHeader, <<
    "Hello">>)).

decode_mac_message_test() ->
    Message = <<16#6188:16, 0:8, 16#CADE:16, "XR", "XT", "Hello"
    >>,
    FrameControl = #frame_control{ack_req = ?ENABLED, pan_id_compr
    = ?ENABLED, frame_version = 2#00},
    MacHeader = #mac_header{seqnum = 0, dest_pan = <<16#DECA:16>>,
    dest_addr = <<"RX">>, src_pan = <<16#DECA:16>>, src_addr = <<"
    TX">>},
    ?assertEqual({FrameControl, MacHeader, <<"Hello">>},
                 mac_layer:mac_decode(Message)).

decode_mac_message_uncompressed_pan_id_test() ->
    Message = <<16#2188:16, 0:8, 16#CADE:16, "XR", 16#CADE:16, "XT
    ", "Hello">>,
    FrameControl = #frame_control{ack_req = ?ENABLED,
    frame_version = 2#00},
    MacHeader = #mac_header{seqnum = 0, dest_pan = <<16#DECA:16>>,
    dest_addr = <<"RX">>, src_pan = <<16#DECA:16>>, src_addr = <<"
    TX">>},
    ?assertEqual({FrameControl, MacHeader, <<"Hello">>},
                 mac_layer:mac_decode(Message)).

decode_ack_frame_from_device_test() ->
    Message = <<16#0200:16, 50:8>>,
    FrameControl = #frame_control{frame_type = ?FTYPE_ACK,
    src_addr_mode = ?NONE, dest_addr_mode = ?NONE},
    MacHeader = #mac_header{seqnum = 50},
    ?assertEqual({FrameControl, MacHeader, <<>>},
                 mac_layer:mac_decode(Message)).

% If Src address mode is zero and frame isn't an ACK. It implies
    that the frame comes from the PAN coordinator
decode_mac_message_no_src_test() ->
    Message = <<16#4108:16, 22:8, 16#CADE:16, 16#CDAB:16, "Test"
    >>,
    FrameControl = #frame_control{frame_type = ?FTYPE_DATA,
```

```
    pan_id_compr = ?ENABLED, dest_addr_mode = ?SHORT_ADDR,
    src_addr_mode = ?NONE},
54   % SRC addr set to zero because can't imply the addr of the PAN
     coordinator at this level
55   MacHeader = #mac_header{seqnum = 22, dest_pan = <<16#DECA
    :16>>, dest_addr = <<16#ABCD:16>>, src_pan = <<16#DECA:16>>,
    src_addr = <<>>},
56   ?assertEqual({FrameControl, MacHeader, <<"Test">>},
57                mac_layer:mac_decode(Message)).
58
59 decode_mac_message_no_src_no_compt_test() ->
60   Message = <<16#0108:16, 22:8, 16#CADE:16, 16#CDAB:16, "Test"
    >>,
61   FrameControl = #frame_control{frame_type = ?FTYPE_DATA,
    pan_id_compr = ?DISABLED, dest_addr_mode = ?SHORT_ADDR,
    src_addr_mode = ?NONE},
62   % SRC addr set to zero because can't imply the addr of the PAN
     coordinator at this level
63   MacHeader = #mac_header{seqnum = 22, dest_pan = <<16#DECA
    :16>>, dest_addr = <<16#ABCD:16>>, src_pan = <<16#DECA:16>>,
    src_addr = <<>>},
64   ?assertEqual({FrameControl, MacHeader, <<"Test">>},
65                mac_layer:mac_decode(Message)).
```

Listing D.1: unit tests performed for the encoding and decoding of a MCA frame

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**