UCLouvain

École polytechnique de Louvain

Protocol stack for 802.15.4 based personal network (6LoWPAN) [GRiSP project with Stritzinger]

Author: **Soukéina Bojabza**
Supervisor: **Peter Van Roy**
Readers: **Ramin Sadre, Peer Stritzinger**
Academic year 2020–2021
Master [120] in Computer Science and Engineering

Abstract

The GRiSP-Base board is a compact circuit board running Erlang applications and that can be used as an Internet of Things device, thanks to modules called Pmods that can be connected to it an provide various sensors and actuators. The Pmod RF2 is the module allowing radio-frequency communication through the Microchip® MRF24J40 IEEE 802.15.4™ 2.4GHz RF transceiver module.

IoT devices communicate with each other through low-power wireless personal area network protocols, such as 6LoWPAN and IEEE 802.15.4. The GRiSP-Base board can communicate through Wi-Fi but does not have a working implementation of the IEEE 802.15.4 and 6LoWPAN protocols.

In this master's thesis, an implementation of an MRF24J40 Microchip driver is made, in order to enable GRiSP-Base boards to communicate as defined by the IEEE 802.15.4 standard. This driver was developed by following the official Microchip datasheet and by translating an existing implementation used by the Contiki operating system. Some parts of a previously attempted driver implementation were also reused.

Despite some remaining issues, the driver works. The boards can communicate not only with each other, but also with other devices running Contiki.

Acknowledgments

First, I would like to thank my advisor, Professor Peter Van Roy, for his availability, the regular meetings and his precious advice.

I also would like to thank Professor Ramin Sadre for first, teaching the excellent course of LINGI2146 - Mobile and Embedded Computing, which made me interested in the subject of my thesis, then for his excellent advice concerning Contiki and the IEEE 802.15.4 protocol, and finally for lending me his Contiki devices which allowed me to test the good functioning of my driver.

The next person I would like to thank is Peer Stritzinger, for giving me advice and helping me debug my driver despite his busy schedule.

And finally, I want to thank my family and my friends for their encouragements and support.

Contents

List of Figures

Chapter 1

Introduction

The Internet of Things (or IoT) is a domain in perpetual expansion, connecting more and more devices over wireless networks every years. By 2021, 35.82 billion IoT devices will be installed all over the world, and 75.44 billion by 2025 [2].

IoT devices are generally small pieces of hardware with low resources. They therefore need appropriate network protocols in order to communicate with other devices. This is why the IEEE 802.15.4 standard and the 6LoWPAN protocol are used, since they are low-power wireless personal area networks protocols.

The GRiSP-Base board is a compact circuit board running Erlang applications that can be used as an IoT device. It can be used for multiple purposes since it can be expanded by connecting different modules, allowing it to play the role of a sensor for example. For the time being, it can only connect to Wi-Fi. To be qualified as a true IoT device, it should also be able to communicate through the low-power wireless personal area networks protocols just mentioned.

The goal of this master's thesis is to make those types of communication possible. The first step towards this aim is to make two GRiSP-Base boards communicate with the IEEE 802.15.4 protocol. The Pmod RF2 is the expansion module allowing radio-frequency communication, by using the MRF24J40 Microchip that offers support for the IEEE 802.15.4 standard. To be more precise, my task will be to implement a working driver for the MRF24J40 Microchip.

At the end of this work, the GRiSP boards should be able to communicate as defined by the IEEE 802.15.4, thanks to the MRF24J40 Microchip and Pmod RF2, which offer support for this RF communication. They should be able to communicate with each other, but also with other devices running the Contiki operating system, which will be used as a reference since it contains a working MRF24J40 Microchip driver. This work will serve as a base for upper network layers implementation, like 6LoWPAN.

This document is organized as follows. The chapter 2 describes the hardware

used in this work. Chapter 3 gives an explanation of the different network protocols involved in this thesis. Then, chapter 4 presents the gathered knowledge needed to implement the driver. In chapter 5, I explain the design and the implementation of the MRF24J40 driver. I then test my implementation in the chapter 6, along with a description of the current remaining bugs and possible solutions to address them. Chapter 7 presents the results of a small performance evaluation. And finally, the last chapter concludes this document. The code written for this work can be found in the appendix.

Chapter 2

Hardware description

The purpose of this chapter is to describe the hardware used in this thesis. It will first detail what the GRiSP-Base board is. Then, it will describe the Pmod RF2 and then present the MRF24J40 Microchip.

2.1 GRiSP-Base board

The embedded device used in this thesis is the GRiSP-Base board and is shown on figure 2.1. It is a compact circuit board developed by Peer Stritzinger GmbH, a German company and leading provider of Erlang solutions for embedded systems [3]. It has built-in Wi-Fi, USB connectivity and other ports[1].

The GRiSP-base board uses a hard real-time embedded "operating system" called RTEMS (= Real-time executive for multiprocessor systems) and the applications running on it are written in Erlang. This allows to create more high-level applications, avoiding to drop to low-level languages close to the hardware like C [4].

The board contains many connectors in which expansion modules can be plugged. Those modules are called Pmods and provide a large range of sensors and actuators. In order to use them, high-level Erlang drivers have been created. The Pmods allow the GRiSP board to be modular and to be used for a large variety of projects.

2.1.1 How to use it ?

Tutorials for setting up a development environment, creating and deploying applications on the board and accessing to the boards Erlang shells are provided[2].

[1] The complete board specification can be found on the GRiSP website: `https://www.grisp.org/specs/`

[2] `https://github.com/grisp/grisp/wiki`

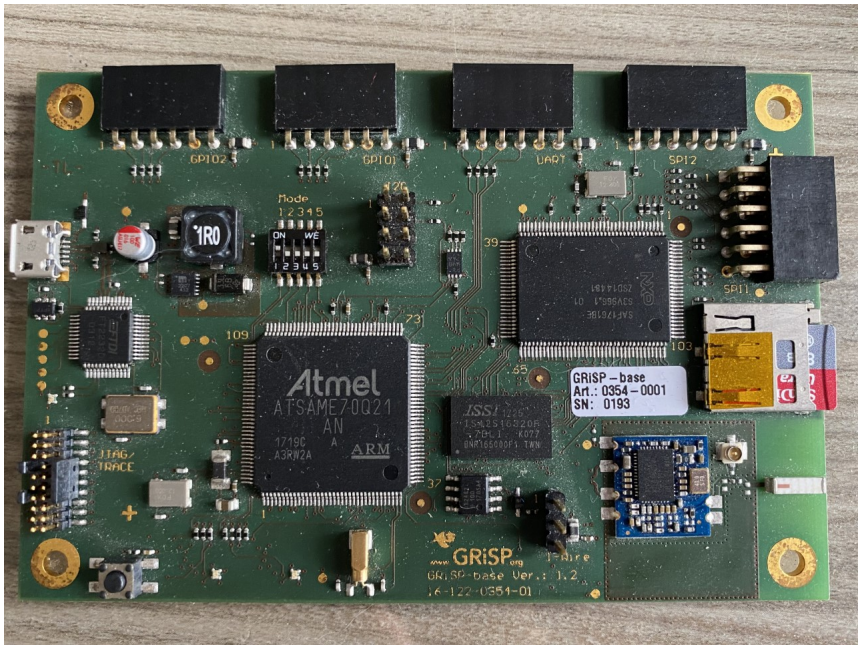


Figure 2.1: GRiSP-Base board

2.2 Pmod RF2

The Pmod used in this thesis is the Pmod RF2, which allows radio-frequency (RF) communication through the Microchip® MRF24J40 IEEE 802.15.4™ 2.4GHz RF transceiver module [5]. The Pmod is shown on figure 2.2.

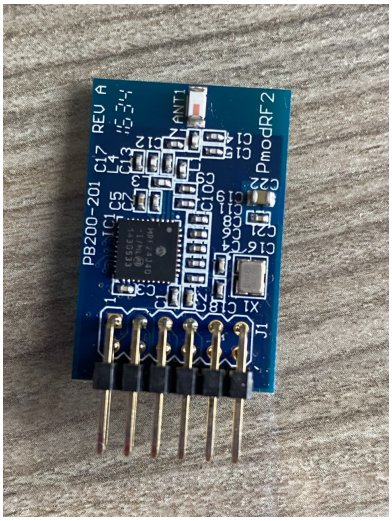
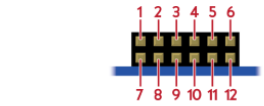


Figure 2.2: Pmod RF2

The GRiSP-Base board communicates with the Pmod through the SPI (Serial Peripheral Interface). The Pmod RF2 pins are shown on figure 2.3. The figure 2.6 shows the Pmod connected on the SPI1 connector (the SPI1 pin names are shown on the figure 2.4 [6]. In order to set or unset a certain pin, the grisp_gpio[3] library is used.

An example of Erlang code setting the `Wake` pin to 0 is shown on figure 2.5.



Pin 1	~CS
Pin 2	SDI
Pin 3	SDO
Pin 4	SCK
Pin 5	GND
Pin 6	VCC
Pin 7	INT
Pin 8	~RST
Pin 9	WAKE
Pin 10	NC
Pin 11	GND
Pin 12	VCC

Figure 2.3: Pmod RF2 pins

SPI1

Pin	Name 1
1	ss1
2	
3	
4	
5	
6	
7	spi1_pin7
8	spi1_pin8
9	spi1_pin9
10	spi1_pin10
11	
12	

Figure 2.4: SPI1 pin names

```
grisp_gpio:configure(spi1_pin9, output_0), % set WAKE pin to 0
```

Figure 2.5: Example of setting a pin with the grisp_gpio library

[3]Go to `https://hexdocs.pm/grisp/` an then choose the grisp_gpio module to see its API.

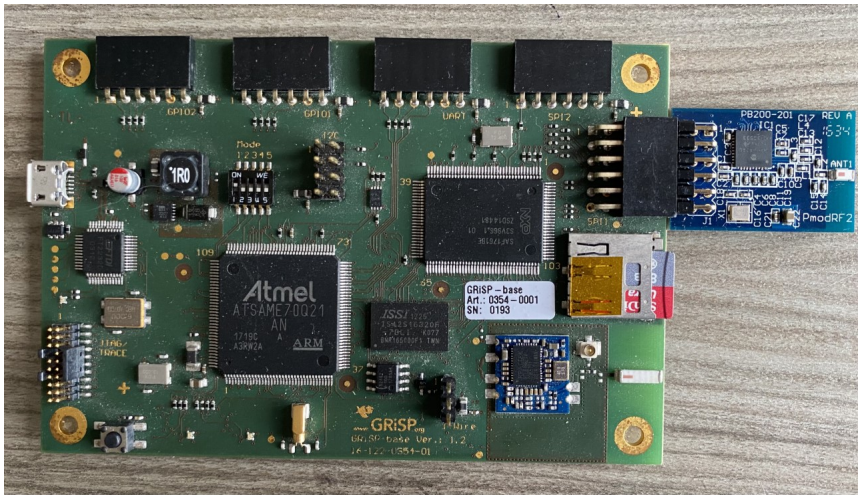


Figure 2.6: Pmod RF2 connected to a GRiSP-Base board

2.3 MRF24J40 Microchip

As said in 2.2, the Microchip® MRF24J40 IEEE 802.15.4™ 2.4GHz RF transceiver module, which will be shortened as MRF24J40 Microchip for the sake of readability, is the component allowing the RF communication. It contains the hardware needed to support the physical and MAC layers of the IEEE 802.15.4 standard.

In order to make the MRF24J40 Microchip work, the control registers it contains have to be set to appropriate values. Its functioning is thoroughly described in the datasheet [7].

The registers are written through the SPI. On figure 2.7, the functions to use in order to read and write the registers are shown. There are two types of registers: the short address registers and the long address ones. This is why there are two setter and two getter functions. The `Addr` argument is the address of the register we want to read or write, and the `Value` argument is the value we want to write in it. The `write` and `read` functions called inside them are the ones using the SPI and were implemented by Zofia (more details on this in the chapter 4).

An example of use is shown on figure 2.8, where the interruption register is read. The figure 2.9 shows a writing example: the value `2#00000001` is written in the `?TXCON` register. Note that `?INTSTAT` and `?TXCON` are the addresses[4].

[4]They are defined in the header file called reg.h (see Appendix).

```
82  % ---------------- Read and write registers ----------------------
83
    Used 35 times | Cannot extract specs (check logs for details)
84  set_short_add_mem(Addr, Val) ->
85      grisp_gpio:configure(ss1, output_0), % set CSn pin low (pin 1 of spi1 is called ss1)
86      write(short, Addr, Val),
87      grisp_gpio:configure(ss1, output_1). % set CSn pin high
88
    Used 12 times | Cannot extract specs (check logs for details)
89  set_long_add_mem(Addr, Val) ->
90      grisp_gpio:configure(ss1, output_0), % set CSn pin low
91      write(long, Addr, Val),
92      grisp_gpio:configure(ss1, output_1). % set CSn pin high
93
    Used 16 times | Cannot extract specs (check logs for details)
94  get_short_add_mem(Addr) ->
95      grisp_gpio:configure(ss1, output_0), % set CSn pin low
96      Ret = read(short, Addr),
97      grisp_gpio:configure(ss1, output_1), % set CSn pin high
98      Ret.
99
    Used 8 times | Cannot extract specs (check logs for details)
100 get_long_add_mem(Addr) ->
101     grisp_gpio:configure(ss1, output_0), % set CSn pin low
102     Ret = read(long, Addr),
103     grisp_gpio:configure(ss1, output_1), % set CSn pin high
104     Ret.
105
106 % ----------------------------------------------------------------
```

Figure 2.7: Functions reading and writing the registers

```
Int_status = get_short_add_mem(?INTSTAT),
```

Figure 2.8: Reading the interruption register

```
set_short_add_mem(?TXNCON, 2#00000001),
```

Figure 2.9: Writing the `?TXCON` register

Chapter 3

6LoWPAN

In an Internet of Things network, small devices are exchanging data over a wireless network without human intervention. These IoT devices are in general small pieces of hardware with useful features such as sensors, actuators, etc., in order to be used for various applications. However, they generally have limited resources: they are low-power and have small amounts of memory. Therefore, the network they are communicating through needed to be adapted.

3.1 IEEE 802.15.4 protocol

The IEEE 802.15.4 protocol defines a standard for the lower layers for low-power wireless personal area networks, ideal for the IoT devices. These lower layers are the physical layer and the medium access control (MAC) layer.

3.1.1 The physical layer

This layer manages the physical radio communication between a transmitter and a receiver. It allows, among other features:

- Channel selection with energy detection: 16 channel frequencies (from 11 to 26) can be chosen in the 2.4 [GHz] band. The energy can be measured to see if the channel is busy or idle, that is, if a communication is happening in the channel or not.
- Link quality indications: two metrics, RSSI and LQI, can be used in order to measure the quality of the link on which a packet is received. While RSSI (Received Signal Strength Indicator) measures the received signal power, LQI (Link Quality Indication) gives an indication of the link quality, and therefore of the received packet quality.

- Transmission and reception of packets.

3.1.2 The MAC layer

The medium access control is the layer managing the MAC frames. It allows, among other features :

- Frames transmission through the physical layer and validation of received frames. It can also manage the acknowledgements sending.
- Clear Channel Assessment: with the help of the physical layer indicators, it is possible to know if the channel is busy or idle before sending a frame.
- Beaconing: beacon frames are sent periodically to announce the presence of a network and to synchronize the nodes of a network.

3.1.3 Next layer ?

This protocol does not define any upper layers above the MAC layer. However, network protocols have been developed and designed to use the IEEE 802.15.4 layers. This is the case of 6LoWPAN.

3.2 6LoWPAN

The 6LoWPAN network protocol is a protocol adapting IPv6 packets in order to be se sent on the small frames of the IEEE 802.15.4 standard. Its name stands for IPv**6** over **Lo**w-Power **W**ireless **P**ersonal **A**rea **N**etworks. The figure 3.1 shows the protocol stack of 6LoWPAN: we can clearly see that the 6LoWPAN layer is defined above the layers defined by the IEEE 802.15.4 standard.

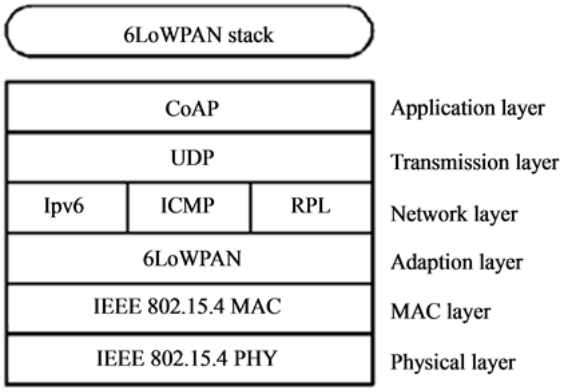


Figure 3.1: 6LoWPAN Protocol Stack [1]

An example of network using 6LoWPAN is shown on the figure 3.2 [8]. On the left, the 6LoWPAN edge router is connected to both the Internet and an IoT network. It therefore needs to adapt the IPv6 packets from the Internet in order to send them to the IoT devices.

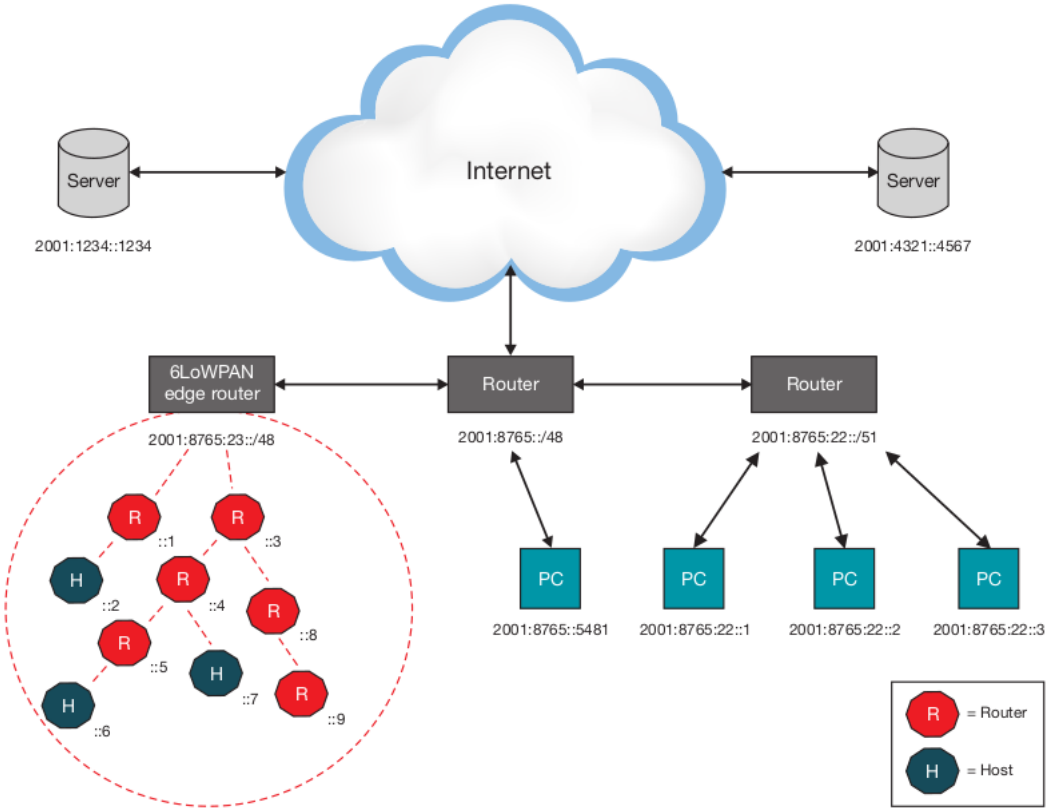


Figure 3.2: Example of IPv6 network with a 6LoWPAN mesh network

3.2.1 IPv6 packets adaptations

The adaptations consists in:

- Header compression: the IPv6 header being quite long, some fields can be omitted. They are generally fields that can be reconstructed with the link layer.
- Packets fragmentation and reassembly: the packets being too large, they need to be cut into smaller pieces. Of course, they need to be reassembled later on and the fragmented packets contain the required information to do so.

3.2.2 Above 6LoWPAN

Routing protocols are running above 6LoWPAN. The most widely used is RPL (Routing Protocol for Low-Power and Lossy Networks).

3.3 Link with the GRiSP board

As said in the previous chapter, the GRiSP board is the embedded device used in this thesis. It can be used as an IoT device since it can connect to the Wi-Fi. But ideally, it should communicate by using 6LoWPAN, with the Pmod RF2 and MRF24J40 Microchip giving the IEEE 802.15.4 support.

This is the ultimate goal of this master's thesis. But before climbing up to 6LoWPAN, a driver implementation needs to be made first in order to allow RF communication as defined by the IEEE 802.15.4 standard, by using the MRF24J40 Microchip. This is what my master's thesis is about.

Chapter 4

Resources and goal

Before starting this thesis, some knowledge needed to be gathered, in order to define a clear goal and to have the required background to reach it.

This is what this chapter is about. It will present, in the first section, the resources used to accomplish the goal described in the second section.

4.1 Resources

4.1.1 Erlang

Erlang is a functional programming language, but not only: it was designed to support concurrency, distribution and fault-tolerance. It was developed by Ericsson at the end of the 80's for telecommunication purposes [9].

Figure 4.1: Erlang logo

The Erlang language is essential to build applications destined to be run on the GRiSP-Base boards. Therefore, learning Erlang was a priority in order to implement the driver. An excellent book [10] allowed me to learn it pretty quickly. It is even easier to get started if you have some knowledge of functional programming.

4.1.2 Contiki

Contiki is an open source operating system dedicated to small Internet of Things devices. The targeted devices are generally low-powered, have small memory amounts (= memory constrained) and are connected to a network to communicate. This is why Contiki is lightweight, manages memory efficiently and supports multiple network protocols, such as IPv4 and IPv6 and also the low-power wireless protocols like 6LoWPAN, RPL and CoAP.

Having followed the LINGI2146-Mobile and Embedded Computing course[1], teached by Pr. Ramin Sadre, I had heard about Contiki and decided to ask him some advice. By chance, it appears that Contiki has a working implementation [11] of an MRF24J40 driver that I could use as a reference for my implementation. Contiki uses the C language so it would need to be translated to Erlang.

4.1.3 Bachelor's thesis of Zofia Polkowska

A previous work implementing the IEEE 802.15.4 physical and MAC layers on the GRiSP boards has been done by Zofia Polkowska for her bachelor's thesis [12].

In her work, she first implemented what was required in order for the GRiSP board to communicate with the Pmod RF2. In other words, the low-level hardware communication through the Serial Peripheral Interface (SPI). This is necessary to perform read and write operations on the MRF24J40 Microchip registers.

Then, she implemented the RF communication, by setting the different control registers as indicated in the MRF24J40 Microchip datasheet. She also provided a way to build frames with different parameters.

Her SPI driver implementation works well and could be used for this master's thesis. However, her RF communication has some issues and could not be kept for this work. These issues mainly concern the reception:

- The interruption that should occur when a packet is received is not triggered, making it difficult to know if a packet arrived.
- The length of the received packet is not the right one: the register always indicates a length of 0.

She also did not have any way to see how the transmitted packet looked like in the air, and since the reception is not working correctly, the transmission could not truly be verified.

Nonetheless, this code still served as reference, to see how to use the SPI driver.

[1] `https://sites.uclouvain.be/archives-portail/cdc2020/en-cours-2020-LINGI2146`

4.2 Goal

After gathering the needed knowledge, the goal of this work became clear: implementing the MRF24J40 driver by translating and adapting the Contiki code into Erlang and by using Zofia's SPI driver for the registers operations. The MRF24J40 Microchip datasheet would also be consulted regularly to understand the Contiki code.

At the end of this work, the RF communication should work, that is, it should be possible to send and receive messages between two GRiSP boards. Ideally, it should also be possible to have a working communication between a GRiSP board and an IoT device using Contiki, to ensure the interoperability of the driver.

Chapter 5

Design and implementation

This chapter will describe the way the MRF24J40 driver code was designed and implemented. It will first present the parts of Zofia's code that were kept. Then it will explain the general implementation, by describing the main functions and the processes. Finally, the last section will expose the different kinds of Contiki code adaptation that were made.

5.1 Reused code

Some code previously implemented by Zofia in her work has been kept for this implementation:

- `reg.hrl`: this header file contains the registers definitions, matching the registers names with their addresses. Additional definitions have been added since some needed ones were missing.
- `mrf24j40_reg.erl`, implementing the reads and writes of the registers through the SPI.
- `mac.hrl`: the header file defining records and types supposed to be used to create frames[1].

5.2 Implementation

The implementation mainly consisted in translating the Contiki code written in C into Erlang, with the help of the MRF24J40 Microchip datasheet in order to

[1]It was not used though, because the frame creation was not done in this thesis. But it was kept for eventual future work.

understand what was done in `mrf24j40.c`[2]. Therefore, it would be useless to describe all the code here. Thus, only the functions needed to use the driver and the processes that needed to be created in order to ensure the communication will be described. Note that the complete code can be found in the appendix.

5.2.1 Functions

The functions to call in order to use the driver are defined in `mrf24j40.erl` and are:

- `init()`: initializes the driver. After this function call, both the transmission and reception are enabled. It consists in writing proper values in certain control registers. It also starts the different processes, which are essential for the good functioning of the communication.
- `set_panid(Id)`, `set_short_mac_addr(Addr)`, `get_short_mac_addr()`, `set_extended_mac_addr(Addr)` and `get_extended_mac_addr()`: as indicated by their name, these functions are used to set the MAC PAN ID and the short and extended MAC addresses. The get functions simply gets them.
- `write(Data, Len)`: this is the function to call in order to send a packet (sends `Data` whose length is `Len`). It first writes the packet in `TXFIFO`, which is the transmission buffer, and then triggers the transmission. It then has to wait for the transmission to complete, by waiting for the `wait_status` process to inform it when it is the case. Finally, the function returns a value depending on the transmission status.
- `read()`: this function is called in order to read a received packet from `RXFIFO`, the reception buffer. This function is never called manually since it is automatically done by the `receiver_process` when a packet is received.
- `cca()`: the Clear Channel Assessment (CCA) indicates whether the channel is busy or not, that is, if an RF transmission is happening in it.
- `on()` and `off()`: respectively turn on and off the receiving mode.

Note that `cca()`, `on()` and `off()` were not tested.

[2] `mrf24j40.c` is the driver code. But `mrf24j40.h` and `mrf24j40_arch.h`, which are its header files and are located in the same folder, were also needed.

5.2.2 Processes

Five processes were created in order to make the communication work. Three of them will be described in this section and the remaining two will be described in the section 5.3, since they were created because Erlang is different than C.

- `interruption_process`: this is the most essential process since it is the one allowing the transmission and the reception. It looks at the interruption register, `INTSTAT`, which is 0 if nothing happened and different than 0 if an interruption occurred.

  An interruption occurs in two cases: when a packet is transmitted and when a packet is received. The type of interruption is known by looking at the `INTSTAT` bits. If it is a reception, the process sends a message to `receiver_process`, telling that it can read the packet. If it is a transmission, the transmission status is read from the `TXSTAT` register and the process tells to `wait_status` that the transmission is over.

  The code is shown on the figure 5.1.

- `receiver_process`: this process is the one that allows the automatic reading of the received packets. It waits for `interruption_process` to allow it to read the packet and informs it when the reading is done. The read packet is then sent to `mailbox`.

  The code is shown on the figure 5.2.

- `wait_status`: when a packet is transmitted, the `write` function needs to wait for the transmission to be over. The role of this process is to inform it when it is the case; and it can do so when it receives a message from `interruption_process` telling that the transmission has finished.

  This process code is shown on figure 5.3.

```erlang
-module(interruption_process).
-include("include/reg.hrl").
-include("include/settings.hrl").
-import(mrf24j40, [read/0, get_short_add_mem/1, set_global/2]).

-export([start/0, interruption_process/0]).

%% When a packet is received, an interruption is issued and we can tell the
%% receiver_process that he can read the buffer.
%% When transmission is over, an interrupt (INTSTAT) is issued and we can
%% then look in TXSTAT if the transmission was successful or not
%% (ISR and PROCESS_THREAD in Contiki)
Used 2 times | Cannot extract specs (check logs for details)
interruption_process() ->
    Int_status = get_short_add_mem(?INTSTAT), % reads the interruption register
    Zero = <<0>>,
    case Int_status of
        Zero -> % if no interruption
            timer:sleep(1),
            interruption_process();
        _ -> % an interruption occured
            <<_:4, RXIF:1, _:2, TXNIF:1>> = Int_status,

            case RXIF of
                1 -> % a packet was received
                    io:format("Interruption_process: can read~n"),
                    set_global(set_pending, 1),
                    % tells to receiver_process that he can read
                    ReceiverPid = whereis(receiver_process),
                    ReceiverPid ! {self(), read},
                    receive
                        {ReceiverPid, done} -> ok
                    end;
                0 ->
                    ok
            end,

            case TXNIF of
                1 -> % a packet was transmitted
                    io:format("Interruption_process: transmission over~n"),
                    Tx_status = get_short_add_mem(?TXSTAT),
                    <<TXNRETRY:2, CCAFAIL:1, TXG2FNT:1, TXG1FNT:1, TXG2STAT:1,
                    TXG1STAT:1, TXNSTAT:1>> = Tx_status,
                    case TXNSTAT of % looking at the transmission status
                        1 ->
                            case CCAFAIL of
                                1 -> set_global(set_status, ?TX_ERR_COLLISION);
                                0 -> set_global(set_status, ?TX_ERR_MAXRETRY)
                            end;
                        0 -> set_global(set_status, ?TX_ERR_NONE)
                    end,

                    % tells to wait_status that the transmission is over
                    WaitStatusPid = whereis(wait_status),
                    WaitStatusPid ! {self(), transmission_over};
                0 -> ok
            end,
            timer:sleep(1),
            interruption_process()
    end.

Used 1 times | Cannot extract specs (check logs for details)
start() ->
    io:format("Starting interruption_process...~n"),
    Pid = spawn(?MODULE, interruption_process, []),
    register(interruption_process, Pid).
```

Figure 5.1: `interruption_process.erl`

```erlang
-module(receiver_process).
-import(mrf24j40, [get_global/1, read/0]).

-export([start/0, receiver_process/1]).

%% Reads the received packet in RX_FIFO only if the interruption process
%% told to do so
Used 2 times | Cannot extract specs (check logs for details)
receiver_process(MailboxPid) ->
    receive
        {InterruptionPid, read} -> % the receiver can read
            Packet = read(),

            InterruptionPid ! {self(), done}, % tells interrupt_proc. the reading is done
            MailboxPid ! {self(), new_packet, Packet}, % sends packet to mailbox

            TestPid = whereis(receiver_test), % the process testing the receiver
            case TestPid of
                undefined -> ok; % if we are not testing
                _ -> TestPid ! {self(), new_packet, Packet} % sends packet to receiver_test
            end,

            receiver_process(MailboxPid);
        _ ->
            receiver_process(MailboxPid)
    end.

Used 1 times | Cannot extract specs (check logs for details)
start() ->
    io:format("Starting receiver_process...~n"),
    MailboxPid = whereis(mailbox),
    Pid = spawn(?MODULE, receiver_process, [MailboxPid]),
    register(receiver_process, Pid).
```

Figure 5.2: `receiver_process.erl`

5.3 Adaptation of Contiki code

The driver implementation was mainly done by translating the Contiki code written in C into Erlang code. However, Erlang being different than C, and the GRiSP boards not having the same functioning as the Contiki devices, some parts could not be used as is and required some adaptations. This is what this section is about.

5.3.1 Single assignment and global variables

The first encountered issue is the use of global variables by the Contiki code: `Last_lqi`, `Last_rssi`, `Status_tx`, `Pending` and `Receive_on`. All along the code, these variables are used and modified according to where they are needed. The problem is that Erlang uses single assignment variables, so they can only be bound once. Therefore, a way to imitate the Contiki behaviour needed to be designed.

This why the `global_vars.erl` process was created. It plays the role of a "server" storing the global variables. It behaves like a recursive function, always calling itself with the same arguments: this is the way the variables value are kept. The process is looping only when it receives a message telling to:

- get the value of the specified variable: in this case, the process simply sends

```erlang
-module(wait_status).
-export([start/0, wait_status/4]).

% Manages the message passing to tell if the packet transmission is over
% Receives the status from the interruption process and tells it to
% "transmission_status()" who asked for it (in mrf24j40.erl).
Used 5 times | Cannot extract specs (check logs for details)
wait_status(Status, Need, Pid, PidI) ->
    receive
        {PidT, status} -> % status asked by transmission_status
            case Status of
                unset -> wait_status(Status, need, PidT, PidI);
                set ->
                    PidT ! {transmission_over},
                    wait_status(unset, no_need, 0, PidI)
            end;
        {PidI, transmission_over} -> % transmission is over
            case Need of
                no_need -> wait_status(set, Need, Pid, PidI);
                need ->
                    Pid ! {transmission_over},
                    wait_status(unset, no_need, 0, PidI)
            end;
        _ -> wait_status(Status, Need, Pid, PidI)
    end.

Used 1 times | Cannot extract specs (check logs for details)
start() ->
    io:format("Starting wait_status...~n"),
    InterruptionPid = whereis(interruption_process),
    Pid = spawn(?MODULE, wait_status, [unset, no_need, 0, InterruptionPid]),
    register(wait_status, Pid).
```

Figure 5.3: `wait_status.erl`

the value (so the corresponding argument) to the process asking for it. It then loops and waits for another message to arrive.

- set the specified variable to the provided value: in this case, the process simply calls itself with the corresponding argument replaced by the value. It then waits for another message to arrive.

The code is visible on the figure 5.4.

```erlang
-module(global_vars).
-include("include/settings.hrl").
-export([start/0, global_vars/5]).

% This module contains the process that manages the global variables
% used in mrf24j40.erl

Used 11 times | Cannot extract specs (check logs for details)
global_vars(Last_lqi, Last_rssi, Status_tx, Pending, Receive_on) ->
    receive
        {_, set_lqi, Value} ->
            global_vars(Value, Last_rssi, Status_tx, Pending, Receive_on);
        {_, set_rssi, Value} ->
            global_vars(Last_lqi, Value, Status_tx, Pending, Receive_on);
        {_, set_status, Value} ->
            global_vars(Last_lqi, Last_rssi, Value, Pending, Receive_on);
        {_, set_pending, Value} ->
            global_vars(Last_lqi, Last_rssi, Status_tx, Value, Receive_on);
        {_, set_receive, Value} ->
            global_vars(Last_lqi, Last_rssi, Status_tx, Pending, Value);

        {From, get_lqi} ->
            From ! {val, Last_lqi},
            global_vars(Last_lqi, Last_rssi, Status_tx, Pending, Receive_on);
        {From, get_rssi} ->
            From ! {val, Last_rssi},
            global_vars(Last_lqi, Last_rssi, Status_tx, Pending, Receive_on);
        {From, get_status} ->
            From ! {val, Status_tx},
            global_vars(Last_lqi, Last_rssi, Status_tx, Pending, Receive_on);
        {From, get_pending} ->
            From ! {val, Pending},
            global_vars(Last_lqi, Last_rssi, Status_tx, Pending, Receive_on);
        {From, get_receive} ->
            From ! {val, Receive_on},
            global_vars(Last_lqi, Last_rssi, Status_tx, Pending, Receive_on);

        _ ->
            global_vars(Last_lqi, Last_rssi, Status_tx, Pending, Receive_on)
    end.

Used 1 times | Cannot extract specs (check logs for details)
start() ->
    io:format("Starting global variables process...~n"),
    Pid = spawn(?MODULE, global_vars, [0, 0, ?TX_ERR_NONE, 0, 1]),
    register(global_vars, Pid).
```

Figure 5.4: `global_vars.erl`

In order to send messages to the `global_vars` process, the `set_global(Type, Value)` function is used to modify a global variable, and `get_global(Type)` is called to retrieve a value. These functions are defined in `mrf24j40.erl`, since this

is where they are needed. Their implementation is shown on the figure 5.5.

```
29  % ---------------- Global vars getter and setter ----------------
30
31  % Type is an atom : get_lqi, get_rssi, get_status, get_pending or get_receive
    Used 10 times | Cannot extract specs (check logs for details)
32  get_global(Type) ->
33      Pid = whereis(global_vars),
34      Pid ! {self(), Type},
35      receive
36          {val, Value} -> Value
37      end.
38
39  % Type is an atom : set_lqi, set_rssi, set_status, set_pending or set_receive
    Used 8 times | Cannot extract specs (check logs for details)
40  set_global(Type, Value) ->
41      Pid = whereis(global_vars),
42      Pid ! {self(), Type, Value}.
43
44  % ----------------------------------------------------------------
```

Figure 5.5: Global variables setter and getter

5.3.2 Single assignment and packets buffer

As said just above, Erlang single assignment is quite incompatible with the way the Contiki code is implemented. This time, it concerns the buffer in which the packet is stored, once it has been read from `RXFIFO`. In Contiki, the same buffer is used for all the messages and is cleared everytime a new message must be stored in it (as shown on figure 5.6).

```
913     packetbuf_clear();
914
915     ret = mrf24j40_read(packetbuf_dataptr(), PACKETBUF_SIZE);
916
917     packetbuf_set_datalen(ret);
```

Figure 5.6: Packet buffer in Contiki

Therefore a way to store the received packets was implemented: the `mailbox` process. It consists in a recursive function keeping a list of the received messages. The list is modified in the following situations:

- when a packet is received, the process reading `RXFIFO` sends it to `mailbox`, which adds it in the list.
- when the process receives a message asking for the oldest packet, it removes it from the list and sends it back to the process asking for it.

- when the process receives a message asking for all the received messages, it sends back all the list to the asking process, and calls itself with an empty list.

The code is shown in the figure 5.7. The messages are sent by the `get_packet(Type)` function, defined in `mrf24j40.erl` and shown on figure 5.8. The comments explain how to use it.

```
1  -module(mailbox).
2  -export([start/0, mailbox/1]).
3
4  %% Stores the received packets (read from rx_fifo) until they are retrieved.
5  %% This replaces the buffer used in contiki to store a packet.
   Used 6 times | Cannot extract specs (check logs for details)
6  mailbox(Mailbox) ->
7      receive
8          {_, new_packet, Packet} -> % receives new packet
9              io:format("Received packet: ~p~n", [Packet]),
10             mailbox([Packet | Mailbox]);
11         {From, get_next} -> % get the oldest message
12             case Mailbox of
13                 [] -> % if mailbox is empty
14                     From ! {next_packet, "No packet to get"},
15                     mailbox(Mailbox);
16                 _ ->
17                     [Next | Rest] = lists:reverse(Mailbox),
18                     From ! {next_packet, Next},
19                     mailbox(lists:reverse(Rest))
20             end;
21         {From, get_all} -> % get a list of all message in arrival order
22             Reverse = lists:reverse(Mailbox),
23             From ! {all_packets, Reverse},
24             mailbox([]);
25         {_, clear_all} ->
26             mailbox([]);
27         _ ->
28             mailbox(Mailbox)
29     end.
30
31
   Used 1 times | Cannot extract specs (check logs for details)
32 start() ->
33     io:format("Starting mailbox...~n"),
34     Pid = spawn(?MODULE, mailbox, [[]]),
35     register(mailbox, Pid).
```

Figure 5.7: `mailbox.erl`

```
437  % Getting the messages stored in the mailbox
438  % Type is an atom:
439  % - get_next for the oldest message
440  % - get_all for the list of all messages in arrival order
     Used 0 times | Cannot extract specs (check logs for details)
441  get_packet(Type) ->
442      Pid = whereis(mailbox),
443      Pid ! {self(), Type},
444      receive
445          {_, Packet} -> Packet
446      end.
```

Figure 5.8: `get_packet` function

5.3.3 Hardware related changes

Contiki being a complete OS, it has to manage things that are not necessary for an implementation on a GRiSP board. Therefore, some lines in the code were considered as specific to Contiki and were simply ignored. The figure 5.9 shows a first example of ignored lines.

```
498      ENERGEST_ON(ENERGEST_TYPE_LISTEN);
509      ENERGEST_OFF(ENERGEST_TYPE_LISTEN);
```

(a) These are maybe energy estimation needed by Contiki

```
924      NETSTACK_RDC.input();
```

(b) Radio duty cycle

Figure 5.9: Examples of ignored lines

Some interruptions were also avoided. First, because the GRiSP does not have a full interruption support so avoiding them would prevent weird behaviours; and next, because when looking in the `mrf24j40.h` header file, some pins seemed to be specific to Contiki. An other example of ignored lines is shown on the figure 5.10.

```
84    const uint8_t tmp = MRF24J40_INTERRUPT_ENABLE_STAT();
```

(a) Interrupt enabling

```
528   /* Set the IO pins direction */
529   MRF24J40_PINDIRECTION_INIT();
530
531   /* Set interrupt registers and reset flags */
532   MRF24J40_INTERRUPT_INIT(6, 3);
533
534   if(MRF24J40_SPI_PORT_INIT(10000000, SPI_DEFAULT) < 0)
535     return -1;
```

(b) Setting of some pins

```
688   #ifdef INT_POLARITY_HIGH
689     /* Set interrupt edge polarity high */
690     set_long_add_mem(MRF24J40_SLPCON0, 0b00000011);
691     PRINTF("MRF24J40 Init INT Polarity High\n");
692   #else
693     set_long_add_mem(MRF24J40_SLPCON0, 0b00000001);
694     PRINTF("MRF24J40 Init INT Polarity Low\n");
695   #endif
```

(c) Setting interrupt edge polarity

Figure 5.10: Other examples of ignored lines

Of course, if the driver was not behaving normally when ignoring these lines, then they would have been used. But as you will see in the next chapter, everything works well.

5.3.4 Interrupt Service Routine and threads

Interrupt Service Routine

The GRiSP boards do not have any Interrupt Service Routine (ISR), that is, a process reacting automatically to changes on the interrupt pin, by executing the appropriate actions related to the interruption. Unfortunately, this is a greatly needed functionality for the MRF24J40 driver implementation, as it can be seen at the line 859 of `mrf24j40.c`.

Therefore, a way to imitate the ISR behaviour needed to be found. This is why `interruption_process.erl`(showed in the figure 5.1) is a loop constantly polling the interruption register (`INTSTAT`):

- If the value in the register is 0, then `interruption_process` loops after sleeping 1 [ms] and checks again the `INTSTAT` register.
- If the value is different than 0, then an interruption occurred and the process can execute what needs to be done.

Threads

In Erlang, threads are in fact processes. Therefore, the thread beginning at line 900 in `mrf24j40.c`, whose role is reading `RX_FIFO` when a packet has arrived, has been implemented as `receiver_process.erl`.

5.3.5 Settings

The `settings.hrl` header file contains some parameters definitions. Some of them were set to a value different from the default values written in the Contiki code. This is the case for `DISABLE_AUTOMATIC_ACK` and `ACCEPT_WRONG_CRC_PKT`, which have been set to true because it does not matter if an ACK message is not automatically sent by the receiver and if the packet checksum is wrong: upper network layers generally send ACK and verify the checksum themselves.

Chapter 6

Tests and debugging

During all the implementation phases, the code has been tested gradually in order to make sure everything was working properly at every level. The tests are separated in two parts:

- the basic tests, which consist in "simply" testing if the code is correct. For example, if the different processes react correctly to the messages they receive, or if the different functions and loops have the expected behaviour.
- the communication tests, which test if the packets transmission and reception happen properly. They generally consist in generating a frame and sending it to the receiver, while observing the behaviour of the boards (with the help of appropriately placed prints) and how the sent packets look like in the air (by using a sniffer and the Wireshark software).

  Note that these tests will also act as a demonstration of the driver.

The first section will focus on the basic tests, while the second will be about the communication tests. Both will present the significant problems related to the aspect they cover and what was modified to resolve them.

The last section will give a summary of the achievements made and the remaining issues.

6.1 Basic tests

As said above, the basic tests check the correctness of some functions and processes. More precisely, it was mostly the code I had to "create" myself since I am relatively new to Erlang. This choice was also made because the operations on the registers were already tested by Zofia in her work and since I used her code, there was no point in testing them again. Also, the functions coming from Contiki were just a

translation from C to Erlang so the risk of incorrectness was low. Moreover, if they were wrong, they would have made the driver behave oddly (since they are directly involved with it), which would be spotted right away. However, some of them were still tested, in order to be more rigorous.

In this section, the tested functions and processes, as well as the major modifications they induced and the problems that remain will be presented.

6.1.1 Functions

The tested functions are the following:

The global variables getter and setter

`set_global(Type, Value)` and `get_global(Type)` were tested in pair with the `global_vars` process. The goal of this test was simply to see if the corresponding global variables were correctly set and gotten. The test consisted in calling `get_global(Type)` after `set_global(Type, Value)` with the same type and seeing if the values matched. This works perfectly.

The useful functions

The useful functions corresponds to the different recursive functions that were needed in order to replace the loops of the Contiki code. They are:

- `for_write(Addr, Buf, Buf_len)`
- `for_read(Addr, Len)`
- `while_short_and(Addr, Value, And_with, Expected)`
- `while_long_and(Addr, Value, And_with, Expected)`

They were tested in a simplified version: instead of reading and writing in registers, they were reading from a hard coded buffer and the writing was replaced by prints in the shell (with `io:format()`). All of them worked perfectly. Only one thing needed to be changed in the implementations: instead of using lists as they were in the beginning, they are now using binary, which is more convenient with the way the registers are read/written.

Some Contiki functions

The Contiki functions are the ones that were translated from the Contiki code, from C to Erlang. Some functions were not tested since they were not directly involved in the transmission/reception tests that were performed; but as said above, they are direct translations so the risk of them not working is really low.

The tested functions are the following:

- `set_channel(Channel)`: this function was indirectly tested when doing transmissions. Indeed, as it will be explained more in details with the communication tests, the sniffer needs to be configured on a certain channel to detect the packets sent on it. When packets were sent on different channels, the sniffer always detected them when set on the same one; and none were detected when set on a different one.

- `set_panid(Id)`: when reading the registers where the PAN ID was written, the correct values were returned.

- `set_short_mac_addr(Addr)` and `get_short_mac_addr()` were tested together by reading the address after writing it. The values were correct.

- `set_extended_mac_addr(Addr)` and `get_extended_mac_addr()` were tested in the same way. But here, the address returned was reversed because the get function was done in the wrong order. This has been corrected.

- `get_last_rssi()` and `get_last_lqi()` are just using `get_global(Type)` with the corresponding types. Therefore, they are correct.

- `set_txfifo(Buf, Buf_len)` was indirectly tested by sending packets. It is guaranteed to work since the sent packets are correct.

- `get_rxfifo()` was indirectly tested by receiving packets. It works since the received packets are correctly read.

- `init()`, `prepare(Data, Len)`, `transmit()`, `write(Data, Len)` and `read()` seem to work adequately since the transmission and reception work. Testing the transmission/reception as described in the communication tests (presented below) acts as the tests for these functions.

All the functions called by the above functions are assumed to work as well since the "callers" behave as expected.

However, there is an issue to mention: the `write(Data, Len)` function sometimes cannot complete and is stuck in a way that only pressing `Ctrl + C` or restarting the board can unblock it. A possible explanation is that the interruption that should be issued doesn't happen, which then prevents the `wait_status` process to give the answer `transmission_status()` needs to allow `write(Data, Len)` to complete. This hypothesis is quite hard to believe because the interruption is triggered the majority of the time.

An other explanation could be that the channel on which the packet should be sent is busy when the bug happens, that is, a communication is happening in it, preventing the transmitter to use it. In general, the channel is supposed to be checked first, by listening to it to see if it is in use. This is what we call Clear Channel Assessment and it is done by the `cca()` function in `mrf24j40.erl`. But as said in chapter 5, this function was not tested. Therefore, maybe calling it before sending a packet, and sending it only if the result is `true` would solve the bug, since the packet would never be sent on a busy channel.

There is still no solution to this "bug" since it works perfectly the majority of the time, but the second hypothesis seems more plausible than the first one, because there are a lot of interference with the Wi-Fi.

Other functions

The last functions tested are `transmission_status()` and `get_packet(Type)` who were tested in pair with the `wait_status` and `mailbox` processes respectively. Since these functions are simply asking something to their respective process (so they send a message to them) and then wait to receive an answer, it is obvious to see that they are correct.

6.1.2 Processes

The processes were generally tested just after their creation to make sure they behaved as expected. The tests generally consisted in sending all the types of messages, as well as the unexpected ones, to them and seeing if they reacted in the required way. In order to grasp what was happening, they were modified by adding some prints in them to display information.

The processes that were tested this way are the following :

- `global_vars`
- `mailbox`
- `wait_status`

The remaining processes, which are `interruption_process` and `receiver_process`, could only be tested during transmission and reception, since they are reading registers. Their behaviour is correct.

However, two issues happened: one has been corrected but the other one remains.

First problem

The first one only happened sometimes but lead to the crash of `receiver_process`: when a packet was received, an interruption was issued and `interruption_process` could tell `receiver_process` to read the reception buffer. At this point, a good behaviour would have been to loop until another packet was received. However, it immediately behaved as if a new interruption was there and therefore told again `receiver_process` to read the reception buffer. `receiver_process` would therefore read the buffer a first time **and** an other time; and this second time would read "nothing", causing `receiver_process` to crash.

Two hypothesis were emitted concerning this bug:

- A "concurrency" problem: `interruption_process` looped faster than the `receiver_process` could read the buffer, making it seem like two packets arrived.
- A "speed" problem: when `interruption_process` looped, the `INTSTAT` register was not 0 yet, causing `interruption_process` to re-read the same interruption value.

The correct hypothesis could not be identified but the solution that was found is a solution to both of them: when `interruption_process` tells to `receiver_process` to read the buffer, it waits for it to tell when the reading is done before doing its loop. Like this, the buffer is guaranteed to be empty and it also gives time to `INTSTAT` to reset. The code of this solution are on figures 6.1 and 6.2.

```
27                  % tells to receiver_process that he can read
28                  ReceiverPid = whereis(receiver_process),
29                  ReceiverPid ! {self(), read},
30                  receive
31                      {ReceiverPid, done} -> ok
32                  end;
```

Figure 6.1: Correction in `interruption_process`

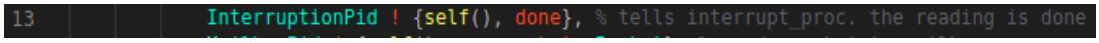


Figure 6.2: Correction in `receiver_process`

When testing again, the crash did not happen anymore. A critic that could be emitted concerning this solution would be that waiting for `receiver_process` to tell when the reading is done could slow down `interruption_process` a bit.

Second problem

The second problem is the one mentioned with the `write(Data, Len)` function: sometimes, when sending a packet, the sender is blocked. This is either because the interruption is not issued, or because the channel on which it is sent is busy. The second hypothesis seems more logical.

As said earlier, there is still no solution but the ways to unblock are either pressing `Ctrl + C` in the shell or restarting the board.

6.2 Communication

The communication tests simply consist in sending packets and observing the sender and receiver boards behaviour. The packets are observed in the air with the help of a sniffer and Wireshark.

The first section will explain what a sniffer is and how it works. In the second section, the communication between two GRiSP boards will be analysed. The transmission/reception between a GRiSP and a Contiki device will act as an interoperability test in the third section.

6.2.1 Sniffer and Wireshark

A sniffer is a device whose goal is to probe the network by capturing the packets it sees in the air and sending them to the host computer it is connected to. The sniffer used is the Open Sniffer, which is specialized in the Internet of Things protocols such as IEEE 802.15.4, Zigbee and 6LoWPAN. This makes it the ideal tool to use for this master's thesis.

The captured packets are then analyzed by a network protocol analyzer. The one used is Wireshark, which is the world's most famous one. It is also free and open-source, and this is the packet analyzer the Open Sniffer works with.

Figure 6.3: The sniffer

Figure 6.4: Wireshark logo

The Open Sniffer tutorial [13] is really well explained and complete and is enough to start using the sniffer (and Wireshark) right away. However, if like me you are using Linux instead of Windows, the host IP and network mask (of the Ethernet interface) settings are done differently. Here are two methods [1]:

Command line: simply use the command

```
ifconfig eth0 <ip_addr> netmask <netmask>
```

where `<ip_addr>` and `<netmask>` are the values in the tutorial (`10.10.10.1` and `255.255.255.0` respectively).

Manual setting: open the network settings of the computer and open the "filaire" settings [2]. Then, fill the IPv4 fields as shown on the figure 6.5.

[1] The first one did not work on my computer so I present a second method in case in does not work for the curious reader as well.

[2] Probably "wired" for an english computer.

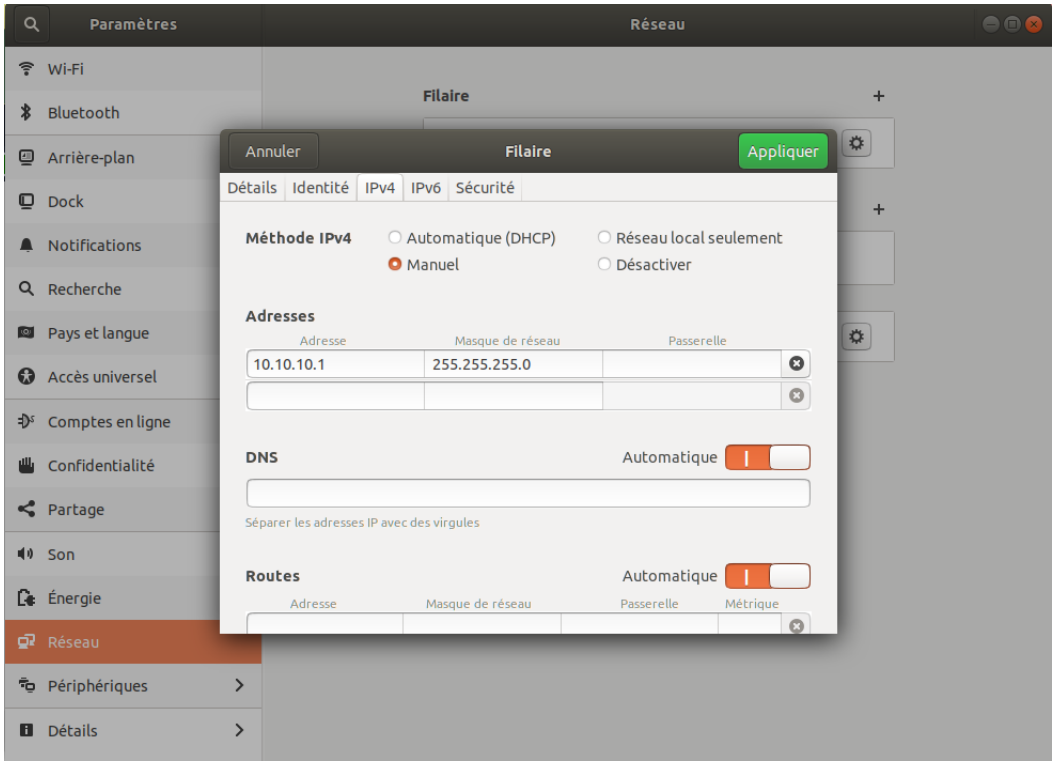


Figure 6.5: Network settings for the ethernet interface

The sniffer (and Wireshark) was used during the communication tests to always have an eye on what was happening in the air in parallel with what could be observed on the boards.

6.2.2 Between two GRiSP boards

This test is the true way to see if the driver implementation works correctly, since it tests both the sender and the receiver at the same time.

Set-up description

The test requirements are the following:

- The sniffer is plugged in the computer and the settings are correct (i.e. the channel it is listening matches with the one on which packets will be sent).
- Wireshark is running on the correct interface and the filter is set on "wpan", which is the IEEE 802.15.4 filter.

- Both GRiSP boards are plugged in the computer and they are both ready to work.

  The sender is initialized by writing the following lines in its Erlang shell:

  ```
  mrf24j40:init(),
  mrf24j40:set_panid(<<16#ABCD:16>>),
  mrf24j40:set_extended_mac_addr(<<16#ABCD:64>>).
  ```

  where the PAN ID and address are set to `«16#ABCD:64»`.

  The receiver is initialized as follows:

  ```
  mrf24j40:init(),
  mrf24j40:set_panid(<<16#DCBA:16>>),
  mrf24j40:set_extended_mac_addr(<<16#DCBA:64>>).
  ```

  where the PAN ID and address are set to `«16#DCBA:64»`.

Frame creation

A frame needed to be created in order to be sent. During the very first tests, the frame was simply hard coded by following the frame format of the IEEE 802.15.4 standard (figure 6.6 [14]) and looked like this:

```
Frame = <<34, 51, 2, 255, 255, 0, 0, 0, 0, 0, 0, 171,
   205, 255, 255, 0, 0, 0, 0, 0, 0, 205, 171, 4, 0, 15,
   30>>.
```

At this time, there were no `interruption_process` and `receiver_process` so the first tests simply consisted in sending the frame with `write(Data, Len)` and seeing if the packet was received by calling `read()` on the receiver board. At this point, the correctness of the `write(Data, Len)` and `read()` could be assumed since the read packet looked like the sent one (see figure 6.7).

This frame was still used when `interruption_process`, `receiver_process` and other processes were added. But even if the transmission was working well, the receiver was not on point yet. Indeed, it should normally trigger an interruption that would allow `receiver_process` to read; however, it was not the case. Therefore, the reception was still not automatized.

Having found nothing suspicious after carefully inspecting the code, the only remaining aspect to check was the frame format. With the help of the sniffer and Wireshark, it was clear that the packet format was wrong, as it can be seen on the Wireshark screenshot on the figure 6.8 [3].

[3]The frame control bytes 22 33 highlighted on the screenshot are the hexadecimals of 34 and 51 so they are the same frames.

Octets: 2	1	0/2	0/2/8	0/2	0/2/8	variable	2
Frame control	Sequence number	Destination PAN identifier	Destination address	Source PAN identifier	Source address	Frame payload	FCS
		Addressing fields					
MHR						MAC payload	MFR

(a) Frame format

Bits: 0–2	3	4	5	6	7–9	10–11	12–13	14–15
Frame type	Security enabled	Frame pending	Ack. request	Intra-PAN	Reserved	Dest. addressing mode	Reserved	Source addressing mode

(b) Frame control field

Figure 6.6: General MAC frame format

```
Eshell V10.4  (abort with ^G)
1> mrf24j40:init().
0
2> mrf24j40:set_extended_mac_addr(<<16#ABCD:64>>).
<<>>
3> Frame = <<34, 51, 2, 255, 255, 0, 0, 0, 0, 0, 0, 171, 205, 255, 255
, 0, 0, 0, 0, 0, 0, 205, 171, 4, 0, 15, 30>>.
<<34,51,2,255,255,0,0,0,0,0,0,171,205,255,255,0,0,0,0,0,0,
  205,171,4,0,15,30>>
4> mrf24j40:write(Frame, 27).
0, 0, 0, 0, 1
0, 0, 0, 0, 1
0, 0, 0, 0, 1
0, 0, 0, 0, 1
0, 0, 0, 0, 1
0, 0, 4, 0, 1
1
5>
```

```
chdir(/media/mmcsd-0-0/)

erl_main: starting ...
getcwd: /media/mmcsd-0-0
hostname: defaulthostname
starting erlang runtime
Eshell V10.4  (abort with ^G)
1> mrf24j40:init().
0
2> mrf24j40:set_extended_mac_addr(<<16#DCBA:64>>).
<<>>
3> mrf24j40:read(27).
0, 0, 0, 0, 1
81, 0, 0, 0, 1
81, 78, 0, 0, 1
<<34,51,2,255,255,156,71,0,0,0,0,0,0,0,242,214,86,111,202,
  31,39,143,134,90,123,62,244>>
4>
```

Figure 6.7: First transmission test

⇒ The packet was so malformed that the receiver did not even identified it as a packet, which is why the interruption was not triggered on the receiver.

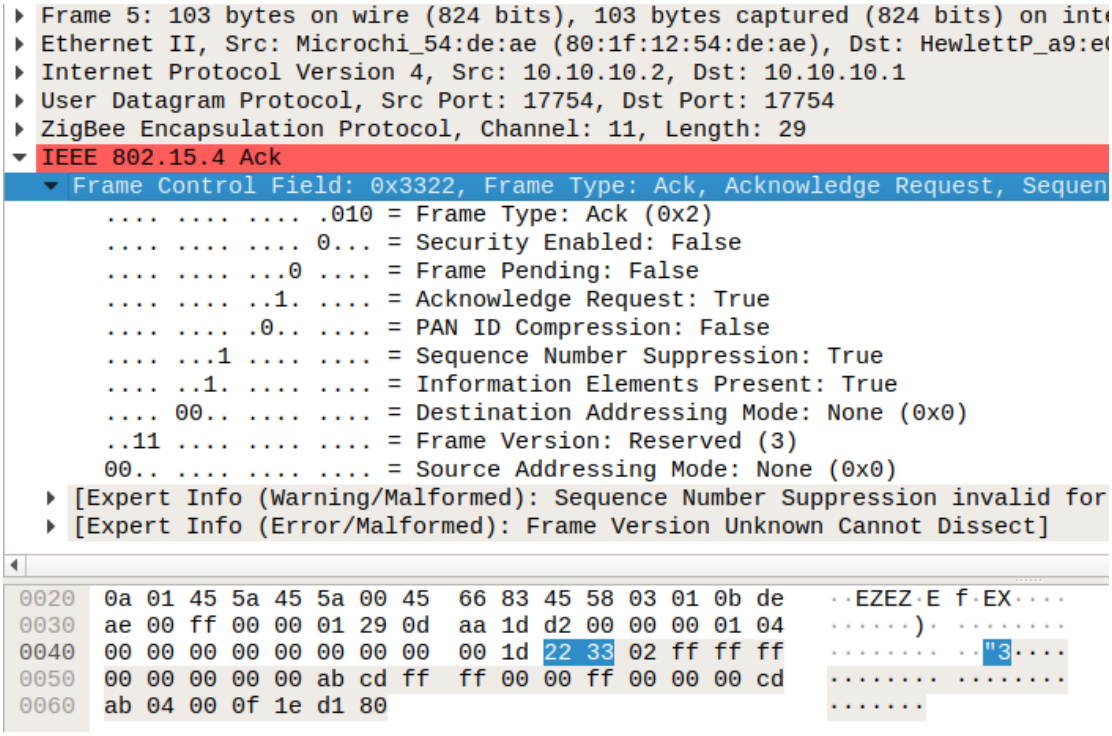


Figure 6.8: First frame captured on Wireshark

Hence, the frame needed to be corrected. A code was produced to hard code a new frame more correctly. This code is located in `frame.erl` and produces the following frame[4]:

```
Frame = <<1,204,1,255,255,186,220,0,0,0,0,0,0,205,171,205,171,0,0
,0,0,0,0,42,0,42>>,
Length = 26.
```

When looking again on Wireshark, the packet was finally correct (see figure 6.9) [5].

[4]In practice, the `makeFrame()` function is only called once to produce the frame, which was then copy-pasted for the tests. Feel free to look at the code in the appendix.

[5]They are the same packets since the 01 and cc frame control bytes are the hexadecimals of 1 and 204.

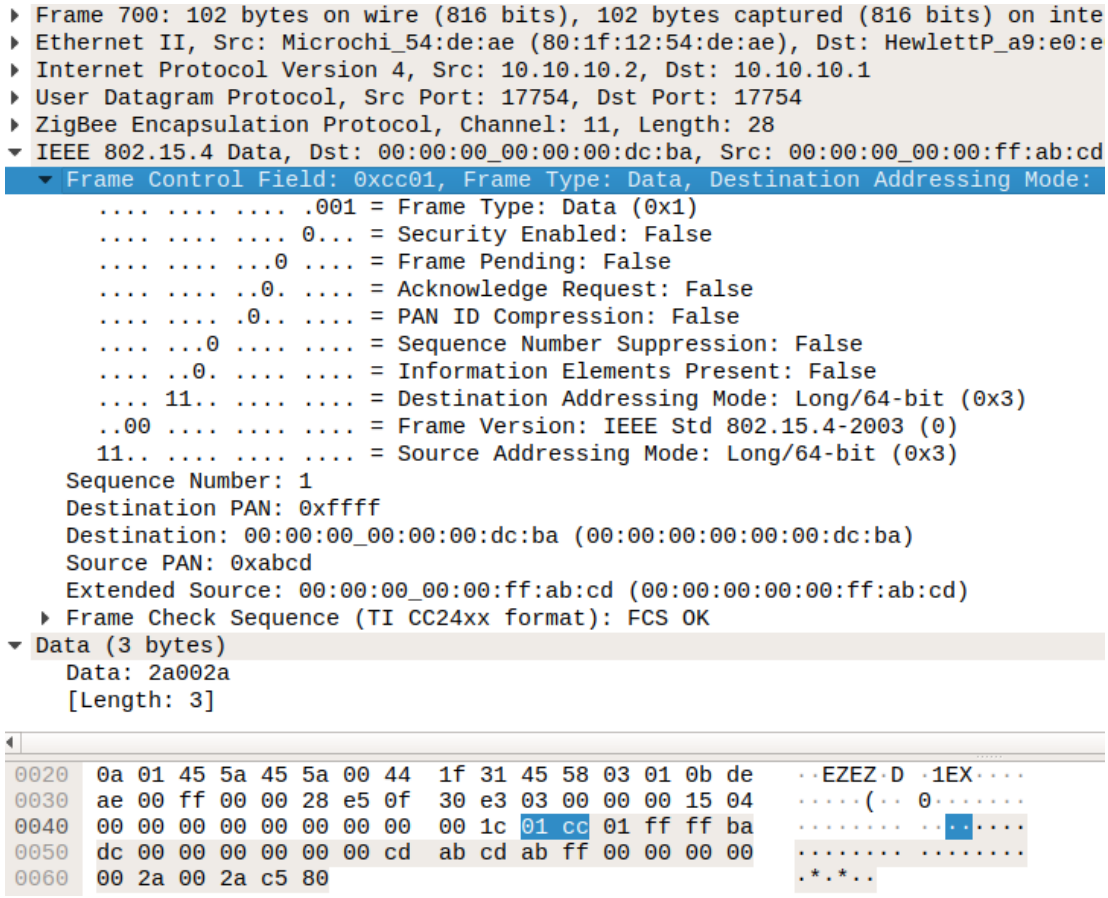


Figure 6.9: Correct frame captured on Wireshark

With this correct packet format, the transmission was tested again and the receiver now reacts correctly at reception. Note that the different packet fields values are displayed on the Wireshark screenshots.

Test

The way to test the transmission is described here.

After having correctly initialized the two boards and set the PAN ID and address,

- the sender needs the following lines to send a packet:

```
{Frame, Length} = frame:makeFrame().
mrf24j40:write(Frame, Length).
```

where the first line creates the frame (and gets its length) and the second one simply sends it. Of course, the frame and length can be set by simply copy-pasting the correct frame and its length shown above.

- the receiver does not need any additional command. During the tests, the packets were displayed in the shell so it was easy to follow them. However, the prints will be removed since their purpose was to help with the debugging. So, the command to see the received packets is

```
mrf24j40:get_packet(get_next).
```

This call returns the oldest received packet. To see all the packets in their order of arrival, the next command returns a list of them:

```
mrf24j40:get_packet(get_all).
```

Demonstration

The next screenshots show the boards shells during the tests in different situations.

The first one, on the figure 6.10, shows the situation when `receiver_process` was crashing, as described in the section 6.1.2. Here, the receiver is on the left of the screen and the sender, on the right.

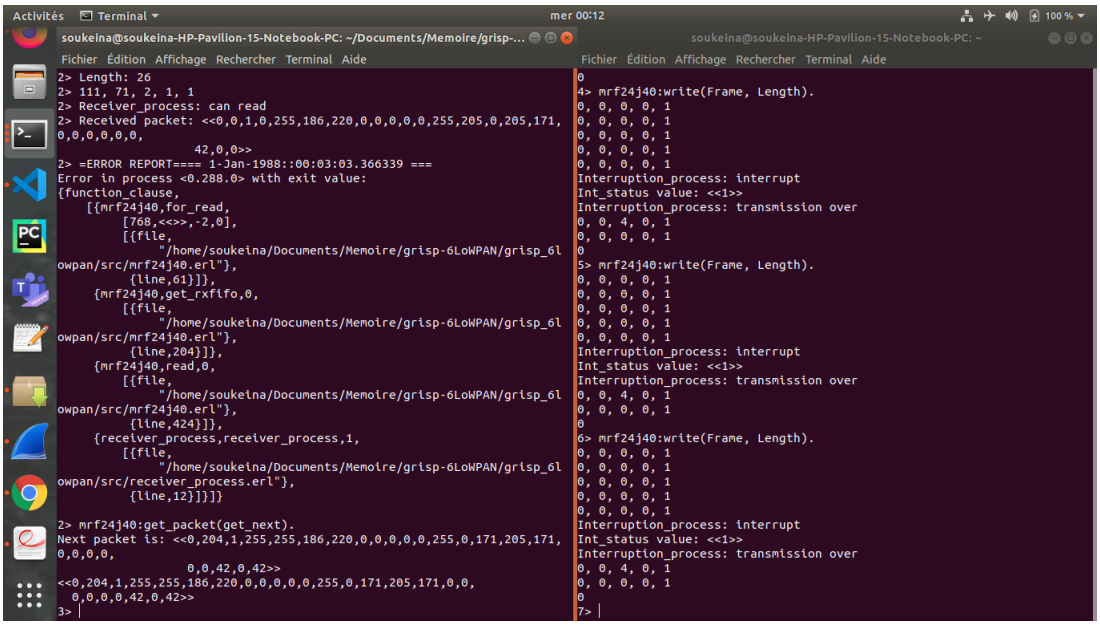


Figure 6.10: `receiver_process` crash

The second one, on the figure 6.11, shows a correct packet transmission and reception, after all the bugs were corrected. Here, the sender is on the left of the screen and the receiver, on the right.

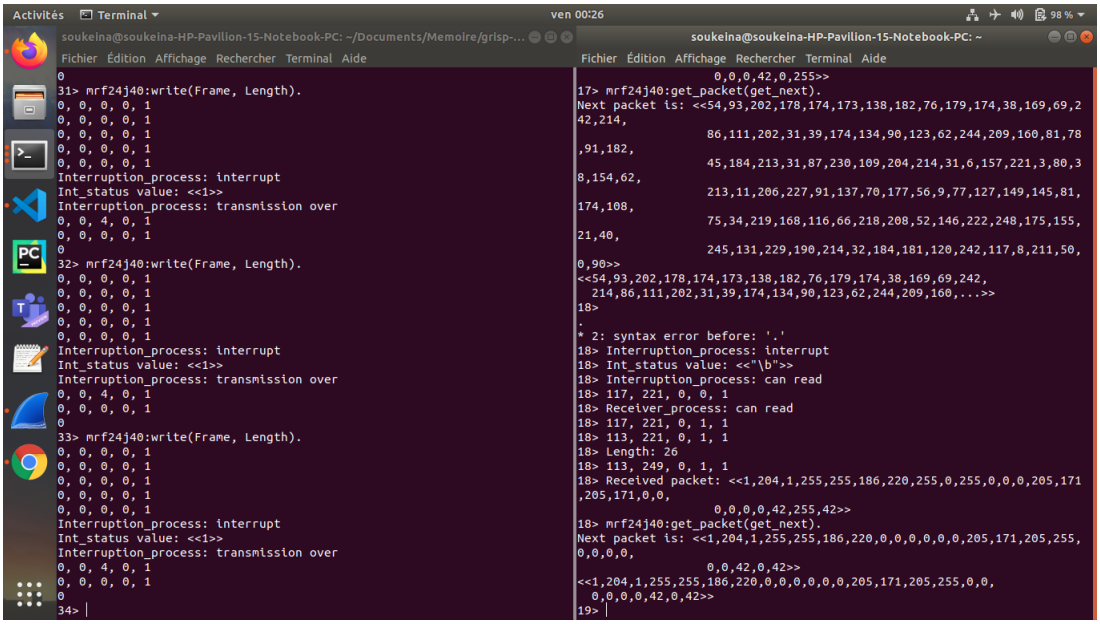


Figure 6.11: Correct transmission

Discussion

The attentive reader may have noticed that the received packet does not have a 100% match with the sent one. This can also be seen on the Wireshark screenshots. This phenomenon happens quite frequently.

The most plausible explication is that the packet distortion is caused by the interference with the surrounding Wi-Fi waves. Indeed, some IEEE 802.15.4 and Wi-Fi channels are overlapping (as shown on the figure 6.12) and since we live in a world full of connected devices, it is not surprising that the packets may be disturbed.

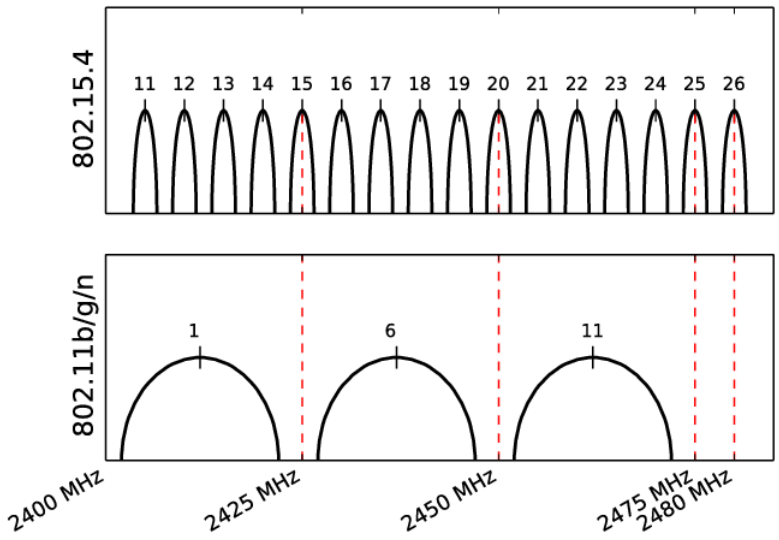


Figure 6.12: IEEE 802.15.4 and Wi-Fi channels

The tests were done using the channel 11 of IEEE 802.15.4 and as shown by the figure, it overlaps with the channel 1 of the Wi-Fi. Therefore, it is better to use the channels 15, 20, 25 and 26. Note that Contiki uses the channel 26 by default.

The default channel in the driver implementation has been changed to 26 and the transmission has been tested again. But once again, some packets were distorted. But since some packets are correct, I can affirm that this is not an implementation problem.

Two possible ways to investigate are:

- Testing in a completely disconnected environment: this would allow to confirm or exclude the interference hypothesis.
- Testing on a different computer or with brand new USB cables: maybe the power supply is too weak to guarantee a perfect functioning of the GRiSP boards and the Pmod RF2. My computer is 6 years old so I cannot affirm that its USB ports are perfectly working. And the cables are sometimes used for other purposes (like charging phones) so they may be a bit worn out.

An other issue to mention is the fact that sometimes, the receiver is not "starting" immediately. Several packets could be sent and none of them would be received (it is not a simple interruption triggering problem).

Three ways have been found in order to avoid that:

- After starting the board, waiting a certain time before initializing it has been observed to work. It is as if it needed a certain warm-up time. When doing this, the receiver reacts directly.
- Typing something in the shell, like `A = 1`, or calling `mrf24j40:read()` sometimes unblock the receiver. This a bit mysterious but it seems to wake the receiver up.
- If none of the above worked, then restarting the board is the last resort.

Note that since some packets, and sometimes several in a row, are wrongly sent, they therefore may not be considered as packets by the receiver, which may simply be ignoring them.

6.2.3 Interoperability with a Contiki device

This test was done to ensure that the receiver is working correctly and to make an interoperability test, which would give more credibility to the driver implementation.

It consisted in using a Contiki device, so an IoT device using Contiki, as a sender and the role of the receiver was played by a GRiSP board. The sniffer and Wireshark were again used to monitor the packets traffic. Only the case where the sender is the Contiki mote and the receiver is the GRiSP is interesting. The other way was not necessary because the transmission by the GRiSP is already tested with the sniffer.

Contiki device

The device used is the Zolertia RE-MOTE (see figure 6.13). In order to use it[6], Contiki-NG, the latest version of Contiki, first needed to be installed on my computer. It is running inside a Docker image.

Once the container is running, the device can be plugged in the computer. The code we want it to execute must first be compiled and then uploaded in it. The device then starts executing it right away and the outputs (the logs) can be seen on the console.

[6]Everything was explained in the tutorial given to me by Pr. Ramin Sadre when he gave me the devices.

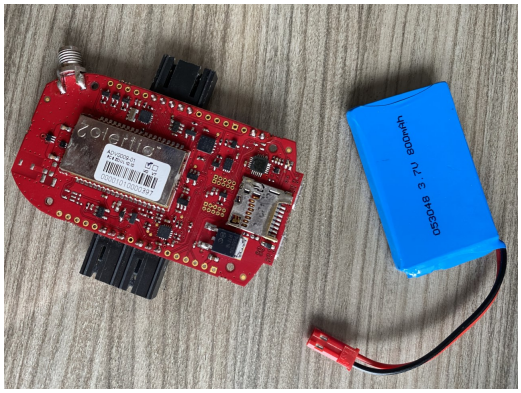


Figure 6.13: Contiki device: Zolertia RE-MOTE

The code needed to make the Contiki mote act as the sender is located in `contiki-ng/examples/rpl-udp/udp-client.c` [7]. In order to send packets to the GRiSP, the code needed to be modified:

- to set the destination address to the corresponding GRiSP address, and
- to set the channel on the right one. At that time, the GRiSP was still using the channel 11 so the Contiki mote (which uses the 26) needed to be set on the 11.

The code modifications are shown on the figure 6.14. The lines 55-56 are setting the channel to 11, and the lines 62-63 set the destination address to the one of the GRiSP.

Set-up description

The test requirements are the following:

- The sniffer is plugged in the computer and the settings are correct (i.e. the channel it is listening matches with the one on which packets will be sent).
- Wireshark is running on the correct interface and the filter is set on "wpan", which is the IEEE 802.15.4 filter.
- A GRiSP board is plugged in the computer and is ready to work. It is initialized as follows:

[7]The code can be seen in the Contiki-NG github repository (`https://github.com/contiki-ng/contiki-ng/blob/develop/examples/rpl-udp/udp-client.c`)

```
42  PROCESS_THREAD(udp_client_process, ev, data)
43  {
44    static struct etimer periodic_timer;
45    static unsigned count;
46    static char str[32];
47    uip_ipaddr_t dest_ipaddr;
48
49    PROCESS_BEGIN();
50
51    /* Initialize UDP connection */
52    simple_udp_register(&udp_conn, UDP_CLIENT_PORT, NULL,
53                        UDP_SERVER_PORT, udp_rx_callback);
54
55    int channel = 11;
56    NETSTACK_RADIO.set_value(RADIO_PARAM_CHANNEL, channel);
57
58    etimer_set(&periodic_timer, random_rand() % SEND_INTERVAL);
59    while(1) {
60      PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&periodic_timer));
61
62      uip_ip6addr(&dest_ipaddr, 0xfe80, 0x0000, 0x0000, 0x0000,
63                  0x0000, 0x0000, 0x0000, 0xabcd);
64
65      /* Send to DAG root */
66      LOG_INFO("Sending request %u to ", count);
67      LOG_INFO_6ADDR(&dest_ipaddr);
68      LOG_INFO_("\n");
69      snprintf(str, sizeof(str), "hello %d", count);
70      simple_udp_sendto(&udp_conn, str, strlen(str), &dest_ipaddr);
71      count++;
72
73      /* Add some jitter */
74      etimer_set(&periodic_timer, SEND_INTERVAL
75        - CLOCK_SECOND + (random_rand() % (2 * CLOCK_SECOND)));
76    }
77
78    PROCESS_END();
79  }
```

Figure 6.14: Changes made in `udp-client.c` in Contiki-NG

```
mrf24j40:init(),
mrf24j40:set_panid(<<16#ABCD:16>>),
mrf24j40:set_extended_mac_addr(<<16#ABCD:64>>).
```

where the PAN ID and address are set to `«16#ABCD:64»`.

- The Contiki device is plugged in the computer and is ready to work (i.e. the code it needs to execute has been uploaded and the logs are ready to be seen).

Test

The next screenshots will show what was displayed on the consoles during the tests.

The first one (the figure 6.15) shows the `receiver_process` crash mentioned in the section 6.1.2. Of course, this was before the bug was solved. On the left, this is the console displaying the execution outputs (the logs) of the Contiki device: it simply sends packets, one after an other with a random time between them. The GRiSP shell displaying the crash is on the right.

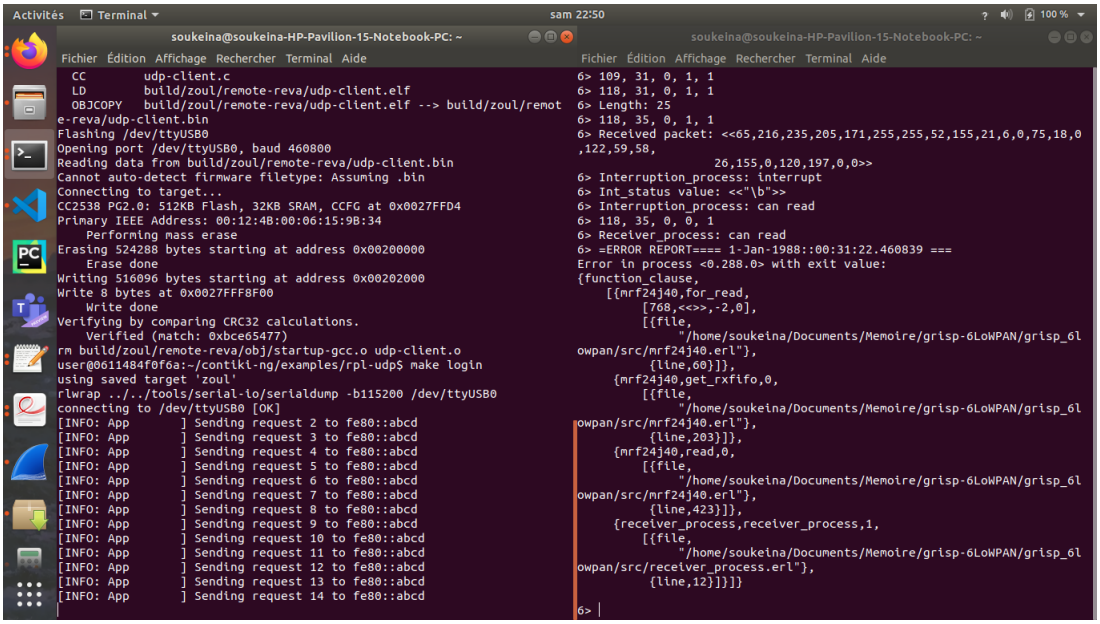


Figure 6.15: `receiver_process` crashing during interoperability test

The second one (the figure 6.16) shows a correct interoperability test.

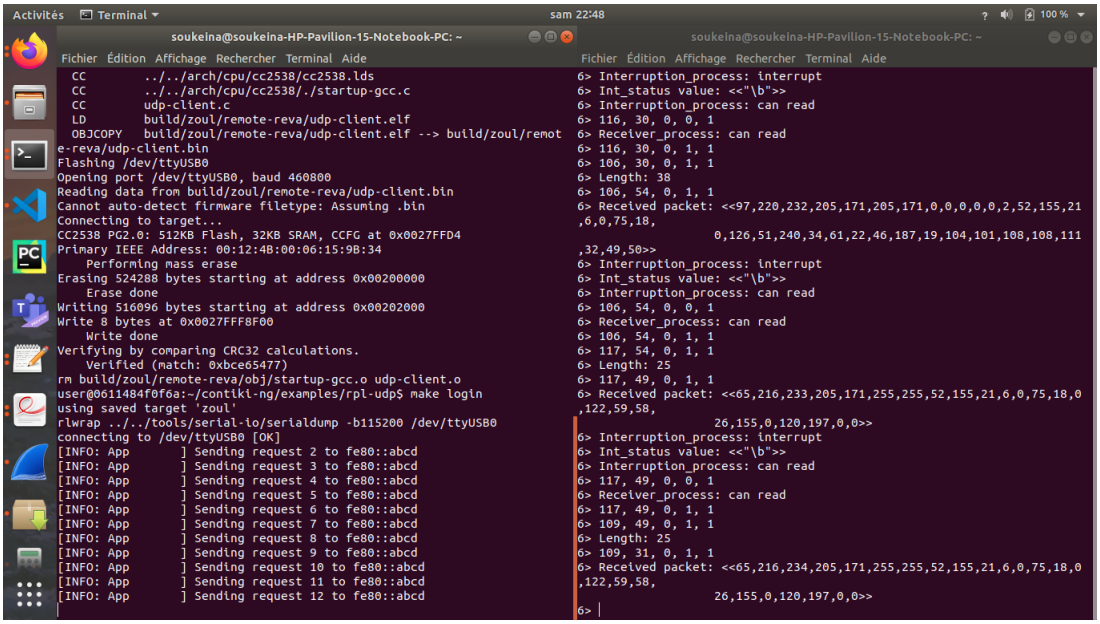


Figure 6.16: Correct interoperability test

A Wireshark capture is shown on the figure 6.17. By translating the frame control bytes, we can see that 61 and dc are the hexadecimals of 97 and 220, which are the first bytes of the received packet. Therefore, the reception works perfectly.

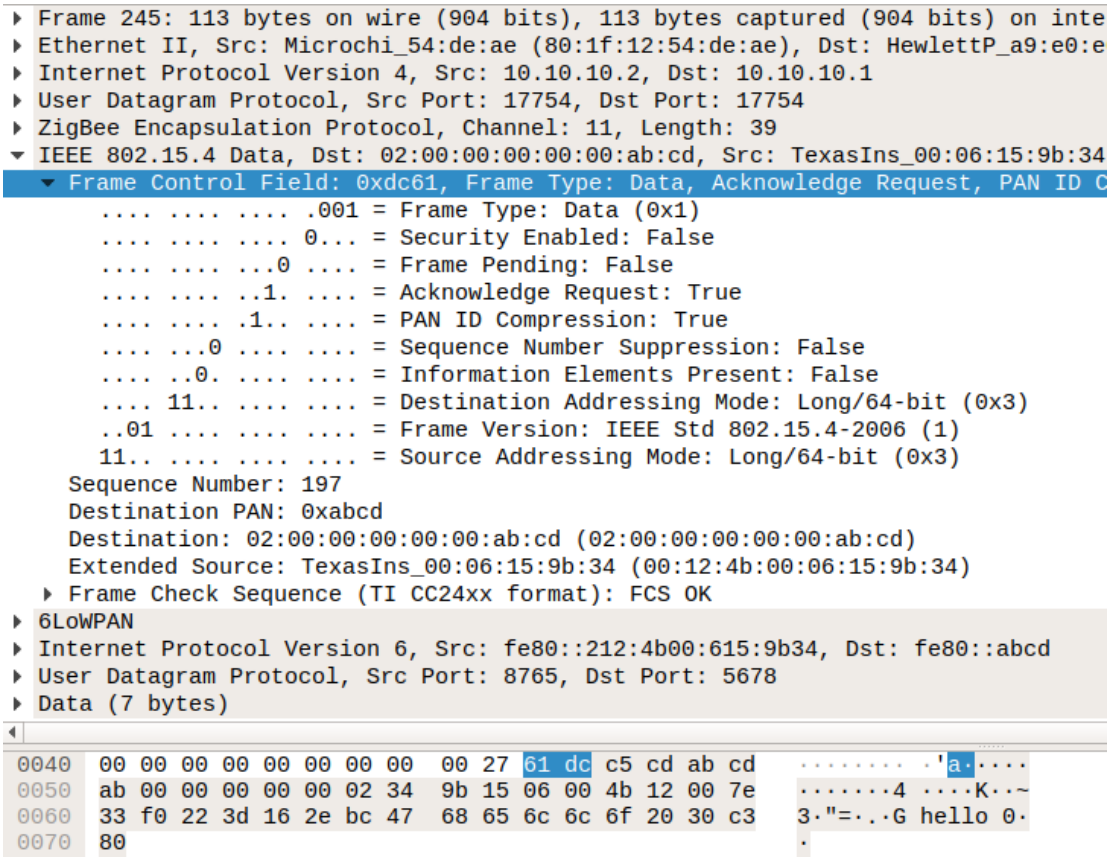
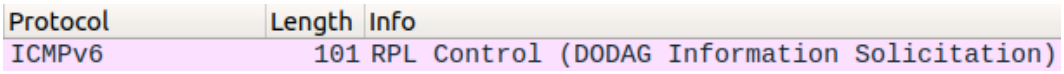


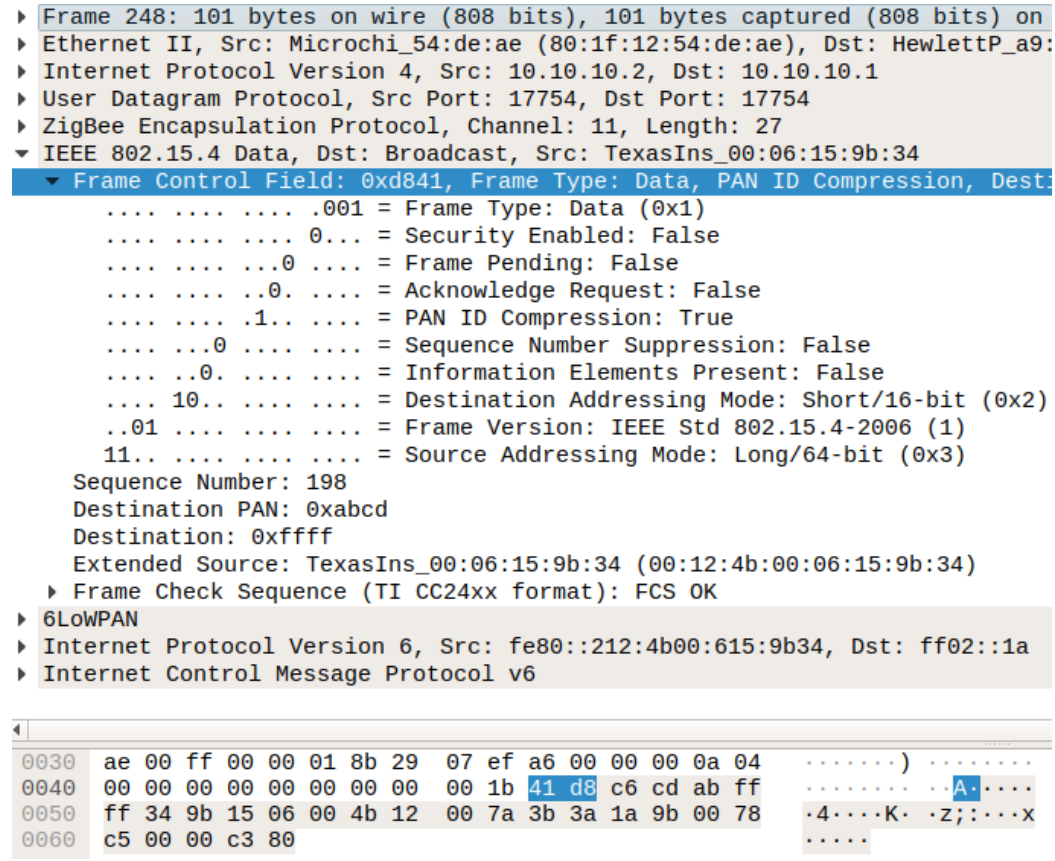
Figure 6.17: Normal Contiki device frame captured on Wireshark

Discussion

The curious reader may have noticed that there is an other type of packet received by the GRiSP. It may not be visible on the Contiki device console, but an other type of packet was also sent. This can be seen with Wireshark (figure 6.18).



(a) DODAG information solicitation captured on Wireshark



(b) Packet details

Figure 6.18: DODAG information solicitation captured on Wireshark

They are messages broadcasted by a node (a device) in order to join a network. The details are a bit out of this thesis scope, but in short, this is how a network is built when using the RPL routing protocol.

By translating the frame control bytes 41 and d8, we see that they are the hexadecimals of 65 and 216, which are exactly the two first bytes of the other received packet (on the GRiSP shell).

This allows to affirm that the driver implementation was correctly done, since it also received this kind of packet.

6.3 Achievements summary

All these tests allowed to ensure that the driver implementation was correct. The functions and processes were thoroughly tested, ensuring good bases for the implementation. Transmission and interoperability tests allowed to test the use of the driver and allowed to correct bugs that could not be found by simply inspecting the code.

With the exception of the three issues appearing sometimes and discussed above, everything works as expected. These issues are

- the never-ending transmission: when the transmission sometimes does not finish and blocks the next transmissions,
- the wrongly-formatted sent packets, probably caused by interference, and
- the late start of the receiver, but once started, it never stops working.

Hypotheses on how to solve them were formulated.

Chapter 7

Performance evaluation

The goal of this chapter is to discuss about the performance of the driver. In order to evaluate it, a test composed of two parts was designed: a part running on the sender (on the first GRiSP board) and a part on the receiver (the other board).

The performance test along with its codes, the issues related to it and the way to start testing are described in the first section. The last section presents and discusses the performance measurements.

7.1 Test description

The test consists in sending a specific frame a certain number of times and seeing how long it takes for all the frames to be sent and to be received, and how many were received correctly. The receiver knows the frame it is going to receive as well as their number in advance.

7.1.1 Sender test

The code is located in **`sender_test.erl`** and is shown on the figure 7.1.

The `Frame` argument is the frame to be sent, `Length` is the frame length and `N` is the number of frames to send. Before starting the loop that sends all the frames (with `mrf24j40:write(Frame, Length)`), `StartTime` is set to the current system time (in milliseconds). As the name indicates it, this is the test starting time. Once the loop is over, the current system time is used as the ending time (`EndTime`).

The time it took to send all the frames (`TotalTime`) is simply computed by subtracting `StartTime` to `EndTime`. The result is divided by 1000 in order to have seconds instead of milliseconds.

At the end, the function displays the starting time, which will be needed to compute the total time after reception, and the total time of the transmission.

```erlang
-module(sender_test).

-export([sender_test/3]).

% This function allows to test the performance of the driver in the sending part
% Sending the frame "Frame" of length "Length" N times
% Displays the time it took in seconds and the start time (needed with receiver_test)
Used 0 times | Cannot extract specs (check logs for details)
sender_test(Frame, Length, N) ->
    StartTime = os:system_time(millisecond), % get current time in millisecond
    sender_test(Frame, Length, N, 0), % loop
    EndTime = os:system_time(millisecond),
    TotalTime = (EndTime - StartTime) / 1000,
    io:format("--------------------Results--------------------~n"),
    io:format("StartTime: ~p, TotalTime: ~p~n", [StartTime, TotalTime]),
    io:format("-----------------------------------------------~n").

Used 4 times | Cannot extract specs (check logs for details)
sender_test(_, _, N, N) -> ok;
sender_test(Frame, Length, N, Count) when Count < N ->
    mrf24j40:write(Frame, Length),
    sender_test(Frame, Length, N, Count + 1).
```

Figure 7.1: `sender_test.erl`

7.1.2 Receiver test

The code is located in **`receiver_test.erl`** and the code is on the figure 7.2.

Compared to the sender test, this one is a process instead of a function to simply call. Therefore, it needs to be started by using the `start(Frame, ExpectedCount)` function, where `Frame` is the frame sent for the test (the one sent by the sender) and `ExpectedCount` is the number of frames to receive (the same as `N` in `sender_test`).

The process consists in a recursive function doing an iteration everytime a packet is received. At each iteration, a counter (the `Count` argument) is incremented by one and the received frame is compared to the expected one. If the frames match, then `CorrectCount`, which is the correct frames counter, is incremented by one; it doesn't change otherwise.

When the number of received frames is the expected one (`Count` = `ExpectedCount`), the current system time (in milliseconds) is taken as the end time of the frames reception, which is also the complete test ending time. The process then displays the results before exiting. The process will therefore need to be restarted (by using `start(Frame, ExpectedCount)`) if another test needs to be performed, because it would be inefficient to have a process running indefinitely just to be used a few times.

A last thing to mention is the way the process receives the packet. The `receiver_process` has been slightly modified in order to send the read packet to `receiver_test` in addition to the mailbox. The code is shown on the figure

```erlang
-module(receiver_test).

-export([start/2, receiver_test/5]).

% This process allows to test the performance of the driver in the receiving part
% Frame is the expected packet sent by the transmitter,
% ExpectedCount is the number of packets expected to be received
% Returns: Count: the total number of received packets
%          CorrectCount: the number of correct packets received,
%          EndTime: the time in millisecond when all packets are received
Used 4 times | Cannot extract specs (check logs for details)
receiver_test(ReceiverPid, Frame, ExpectedCount, Count, CorrectCount) ->
    receive
        {ReceiverPid, new_packet, Packet} ->
            Cnt = Count + 1,
            case Packet of
                Frame -> CorrCount = CorrectCount + 1; % if the packet is the expected one
                _ -> CorrCount = CorrectCount
            end,

            case Cnt of
                ExpectedCount -> % all messages have been received
                    EndTime = os:system_time(millisecond), % the current time in millisecond
                    io:format("--------------------Results--------------------~n"),
                    io:format("Count: ~p, CorrectCount: ~p, EndTime: ~p~n", [Cnt, CorrCount, EndTime]),
                    io:format("-----------------------------------------------~n"),
                    exit(normal);
                _ ->
                    receiver_test(ReceiverPid, Frame, ExpectedCount, Cnt, CorrCount)
            end;
        _ -> receiver_test(ReceiverPid, Frame, ExpectedCount, Count, CorrectCount)
    end.

Used 0 times | Cannot extract specs (check logs for details)
start(Frame, ExpectedCount) ->
    io:format("Starting receiver_test...~n"),
    ReceiverPid = whereis(receiver_process),
    Pid = spawn(?MODULE, receiver_test, [ReceiverPid, Frame, ExpectedCount, 0, 0]),
    register(receiver_test, Pid).
```

Figure 7.2: `receiver_test.erl`

```erlang
            TestPid = whereis(receiver_test), % the process testing the receiver
            case TestPid of
                undefined -> ok; % if we are not testing
                _ -> TestPid ! {self(), new_packet, Packet} % sends packet to receiver_test
            end,
```

Figure 7.3: Code added in `receiver_process` to make the test work

7.3. Thus, when the driver receives a packet, `receiver_process` first checks if the testing process is running and if it is the case, it sends the packet to it.

7.1.3 Additional notes

There are small issues with this test and they are presented here with their solutions:

Timing issue: the test was designed assuming the system time on the two boards was the same (like on every computer). However, it is not the case. The two boards are not exactly on the same date and hour.

Solution: A way to compensate this time offset is to measure it beforehand. So, before starting the test, calling the `system_time()` function in both Erlang shell as quickly as possible (or with a known offset) and subtracting the results will give the correction factor to the time measurements between the receiver and the sender.

A better test would have been to measure the round-trip time by measuring the time between the packet transmission and an acknowledgment reception. However, the work achieved in this thesis does not include a way to build frames on demand (the frame used has been produced in a hard coded way), so it would have been less convenient.

Driver unreliability: when sending multiple packets in a row, there are times where a packet will not be sent and this will prevent the next packets to be transmitted. This problem and its hypothetical solutions were described in the chapter 6.

Therefore, testing with a too large number of frames would probably not lead to a successful test. An ideal number would be $1 \leq N \leq 10$.

7.1.4 Commands

In order to start testing, one needs to follow the next commands in the GRiSP boards Erlang shell. These commands need to be typed after having initialized the boards and set the addresses (as shown earlier in the chapter 6, section 6.2).

For measuring the time offset between the boards:

```
os:system_time(Format).
```

where `Format` is either `millisecond` or `second`. Subtract the results and keep it for later.

On the sender board shell, we need the frame (either copy-pasted or created with the `makeFrame` function), its length and the number of times to send:

```
{Frame, Length} = frame:makeFrame(),
N = 5,
sender_test:sender_test(Frame, Length, N).
```

On the receiver board, we also need the frame and their expected number:

```
{Frame, Length} = frame:makeFrame(),
N = 5,
receiver_test:start(Frame, N).
```

7.2 Results

Two tests were made, one with `N = 1` and the other with `N = 5`.

The results are presented on the following figures 7.4 and 7.5, where the sender is on the right and the receiver, on the left.

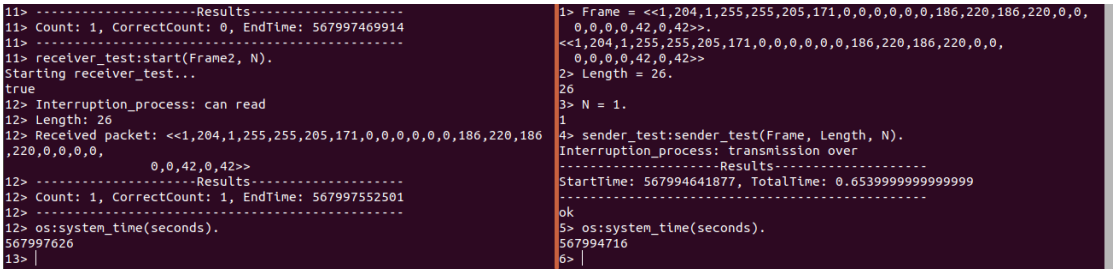


Figure 7.4: Performances with 1 packet sent

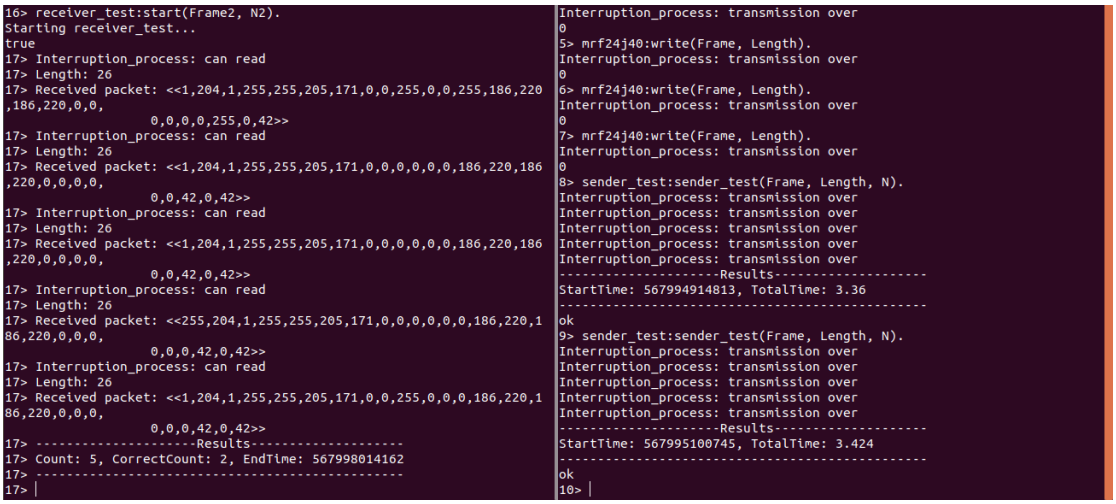


Figure 7.5: Performances with 5 packets sent

7.2.1 Time

First, the time difference between the boards is obtained by doing `Offset = 567997626 - 567994716 = 2910 [s]` (the values are displayed on the figure 7.4), so the receiver board is 2910 seconds in the future. This value will need to be subtracted from the total times.

For N=1, the sending takes 0.654[s]. The total transmission time is `EndTime - StartTime - Offset = 0.624[s]`.

The total transmission value is not relevant because the offset precision is really bad and measured in seconds[1], which makes it smaller than the sending time when in should be the reversed situation.

For N=5, the sending takes 3.424[s] and the total transmission time is 3.417[s].

Again the total transmission time is not relevant.

The time measurements really suffer from the GRiSP boards time difference. Unless a solution is found to set them to the same time, the results will never be accurate and no conclusion can be derived for them.

Note that the times may be smaller than the printed values in reality, because the prints done by `interruption_process` and `receiver_process`, which are useful to see what is happening, slow down the execution[2]. It is really visible when looking at the shell during the execution because the lines appear sequentially.

7.2.2 Correctness

With **N=1**, the packet correctness is 100%, and with **N=5**, the packet correctness is 40%.

These results illustrate perfectly the issue where the packets sent are not always correct, probably due to the interference (as said in the section 6.2.2 of chapter 6).

[1] Measuring it in milliseconds is nonsense because my reaction time can not be measured precisely.

[2] The io library is really slow, as said here: `https://stackoverflow.com/questions/5155337/erlang-ioformat-buffering-and-efficiency`

Chapter 8

Conclusion and Future work

In this thesis, an Erlang implementation of an MRF24J40 Microchip driver for the GRiSP-Base board has been made, by translating an existing implementation for the Contiki operating system.

After a thorough testing and debugging, the driver is working and an RF communication can be established between two GRiSP boards. Packets can be sent and received and the reception works automatically, since the receiver knows when a packet was received, thanks to the interruption process and its permanent polling of the interruption register.

Moreover, an interoperability test showed that the communication also works between a GRiSP board and another device using Contiki, which is the Zolertia RE-MOTE. More precisely, the GRiSP board receives what the Contiki mote is sending. The opposite direction was not tested but the good functioning of the transmission was ensured with the use of the sniffer and Wireshark, which show the packets sent in the air.

However, three issues were found in this implementation. They are not happening everytime but when they do, they prevent the driver to be used correctly.

The first one is the never-ending transmission issue, where the transmission does not finish and blocks the next ones. Two hypotheses have been formulated: either the interruption normally occurring when a packet is sent is not triggered, which is doubtful since it is done the majority of the time; or the packet cannot be sent because the channel is busy with another communication, which is more plausible since there are lots of Wi-Fi interference.

The second one is the wrongly-formatted packets sending problem, where the packets in the air and arriving to the receiver are not 100% identical to the sent one. This issue is probably caused by interference with surrounding Wi-Fi waves.

The third and last issue is the fact that the receiver sometimes does not always start directly after initializing the driver, maybe because it needs a warming-up

time.

Apart from these issues, which in my opinion are not related to the implementation itself, the driver works as expected. Thanks to it, communication between two GRiSP boards is possible.

8.1 Future work

If somebody were to continue this work, he would first need to solve the mentioned driver issues. Many hypotheses have been formulated in the chapter 6, and they should be accepted or excluded by testing them one after the other.

Another task, even if it was not explicitly said in this work, would be to replace the way the interruptions are detected; by replacing the interruption register polling principle by a true Interrupt Service Routine (ISR).

Once these issues are solved, the next task would be to implement the MAC layer of the IEEE 802.15.4 standard, that is, managing the frames.

After that, the lower layers would be operational. Therefore, the 6LoWPAN protocol implementation could be started.

Bibliography

[1] 6LoWPAN Protocol Stack (OSI Reference). https://www.researchgate.net/figure/6LoWPAN-Protocol-Stack-OSI-Reference_fig1_330049258, Nov 2020.

[2] 21+ Internet of Things Statistics, Facts & Trends for 2021. https://findstack.com/internet-of-things-statistics, Aug 2021.

[3] Peer Stritzinger GmbH — Stritzinger.com. https://www.stritzinger.com, Aug 2021.

[4] GRiSP Technical Specifications — GRiSP. https://www.grisp.org/specs, Jun 2021.

[5] Pmod RF2 - Digilent Reference. https://digilent.com/reference/pmod/pmodrf2/start, Aug 2021.

[6] Physical to atom pin mappings. https://github.com/grisp/grisp/wiki/Physical-to-atom-pin-mappings, Aug 2021.

[7] Microchip Technology Inc. MRF24J40 Data Sheet. IEEE 802.15.4™ 2.4 GHz RF Transceiver. DS39776C datasheet. http://ww1.microchip.com/downloads/en/DeviceDoc/39776C.pdf?_ga=2.149406454.1546766180.1613593420-2079461428.1602419288, 2010.

[8] Jonas Olsson. 6LoWPAN demystified, Oct 2014.

[9] Erlang – What is Erlang. http://erlang.org/faq/introduction.html, Aug 2021.

[10] Fred Hebert. *Learn You Some Erlang for Great Good!: A Beginner's Guide*. No Starch Press, Jan 2013. Freely available on https://learnyousomeerlang.com.

[11] contiki os. contiki. https://github.com/contiki-os/contiki/tree/master/platform/seedeye/dev/mrf24j40, Aug 2021.

[12] Zofia Polkowska. IEEE 802.15.4-based Mesh Networking Using Erlang and Embedded Devices, 2018.

[13] Open Sniffer Installation | Sewio RTLS. `https://www.sewio.net/open-sniffer/sniffer-installation`, Aug 2021.

[14] IEEE Computer Society. IEEE Std 802.15.4 TM -2003. IEEE Standard for Information technology—Telecommunications and information exchange between systems—Local and metropolitan area networks—Specific requirements, Oct 2003.

Appendix A

Implementation code

This appendix shows the driver implementation code. It contains the header files defining the different registers and parameters, the driver implementation and the performance evaluation code. The code is commented in order to be clear.

Note that the listing captions are above the code (not below).

A.1 Header files

Header files should be placed into a folder called "include".

Listing A.1: reg.hrl: defines the registers and some parameters.

```
% Partially implemented by Zofia Polkowska
-define(SHORT, 2#0).
-define(LONG, 2#1).

-define(W, 2#1).
-define(R, 2#0).

-define(Skip, 0).
%% REGISTERS
%%
-define(PANIDL, 16#01).
-define(PANIDH, 16#02).
-define(SADRL, 16#03).
-define(SADRH, 16#04).
-define(EADR0, 16#05).
-define(EADR1, 16#06).
-define(EADR2, 16#07).
-define(EADR3, 16#08).
-define(EADR4, 16#09).
-define(EADR5, 16#0A).
```

```
21 -define(EADR6, 16#0B).
22 -define(EADR7, 16#0C).
23 %% software reset
24 -define(SOFTRST, 16#2A).
25
26 %% POWER AMPLIFIER CONTROL REGISTERS
27 -define(PACON0, 16#16).
28 -define(PACON1, 16#17).
29 -define(PACON2, 16#18).
30
31 %% INTERFRAME SPACING
32 -define(TXPEND, 16#21).
33 -define(TXTIME, 16#27).
34 -define(TXSTBL, 16#2E). %% TX STABILIZATION REGISTER
35
36
37 %% table p.92 MRF24J40 Data Sheet
38
39 %% RF MODE CONTROL REGISTER
40 -define(RFCTL, 16#36).
41
42 %% RFCON[number] registers - RF CONTROL REGISTERS
43 %%
44 %% channel selection and frequency ( 2.4GHz band )
45 -define(RFCON0, 16#200). %%CHANNEL selection (bit 7-4), RF
       Optimize (bit 3-0)
46 -define(RFCON1, 16#201). %%Voltage-controlled oscillator
       optimization bits
47 -define(RFCON2, 16#202). %%PLL - Phase-locked loop (must be
       enabled for RF trans & rec)
48 -define(RFCON3, 16#203).
49 -define(RFCON6, 16#206). %% TX filter (7), 20MHx Clock Recovery
       (4), Batery monitor enable (3)
50 -define(RFCON7, 16#207). %% Sleep clock internal/external -
       selection (7-6)
51 -define(RFCON8, 16#208). %% Initialize VCO (4)
52 -define(RSSI, 16#210).
53 -define(SLPCON1, 16#220).
54 -define(BBREG2, 16#3A).
55 -define(CCAEDTH, 16#3F).
56 -define(BBREG6, 16#3E).
57
58 %%interrupts
59 -define(INTSTAT, 16#31).  %% flags
60 -define(INTCON, 16#32).   %% for enabling interrupts
61 -define(SLPCON0, 16#211). %% <1> bit determines the edge polarity
       of the INT pin
62                           %% match the interrupt pin polarity of
                                the host microcontroller!
```

```
63                              %%
64 %% RECEIVE MAC CONTROL REGISTER reception modes,
       pcoord/coord/device
65 -define(RXMCR, 16#00).
66 -define(RXFLUSH, 16#0D).
67 %% CSMA-CA MODE CONTROL REGISTER
68 -define(TXMCR, 16#11).
69 -define(ORDER, 16#10).
70
71 -define(RX_FIFO, 16#300).
72 -define(TX_FIFO, 16#000).
73
74 %%% Added by me %%%
75 -define(BBREG1, 16#39).
76 -define(RFSTATE, 16#20F).
77 -define(WAKECON, 16#22).
78 -define(SLPACK, 16#35).
79 -define(TESTMODE, 16#22F).
80 -define(TXNCON, 16#1B).
81 -define(TXSTAT, 16#24).
```

Listing A.2: settings.hrl: defines the settings needed for the implementation.

```
1 -define(ADD_RSSI_AND_LQI_TO_PACKET, true). % enable the packet
      RSSI
2
3 -define(DEFAULT_CHANNEL, 26). % better to use 15, 20, 25 or 26
4
5 % Activate or not the external amplifier needed by the MRF24J40MB
6 -define(MRF24J40MB, false).
7
8 % Transmission status
9 -define(TX_ERR_NONE, 0).
10 -define(TX_ERR_NOTSPECIFIED, 1).
11 -define(TX_ERR_COLLISION, 2).
12 -define(TX_ERR_MAXRETRY, 3).
13 -define(TX_WAIT, 4).
14
15 % Radio return values for the transmit() function
16 -define(RADIO_TX_OK, 0).
17 -define(RADIO_TX_ERR, 1).
18 -define(RADIO_TX_COLLISION, 2).
19 -define(RADIO_TX_NOACK, 3).
20
21 % Maximum buffer size
22 % see
       https://github.com/contiki-os/contiki/blob/master/core/net/packetbuf.h
23 -define(PACKETBUF_SIZE, 128).
24
```

```
25 -define(DISABLE_AUTOMATIC_ACK, true). % No automatic ACK
26 -define(PAN_COORDINATOR, false). % Device is not a PAN coordinator
27 -define(COORDINATOR, false). % Device is not a coordinator
28 -define(ACCEPT_WRONG_CRC_PKT, true). % Accept packets with wrong
       CRC
29
30 % Discard packet when there is a MAC address mismatch,
31 % illegal frame type, dPAN/sPAN or MAC short address mismatch.
32 -define(PROMISCUOUS_MODE, false).
33
34 % Default settings for init : (in Contiki code)
35 % -define(DISABLE_AUTOMATIC_ACK, false).
36 % -define(PAN_COORDINATOR, false).
37 % -define(COORDINATOR, false).
38 % -define(ACCEPT_WRONG_CRC_PKT, false).
39 % -define(PROMISCUOUS_MODE, false).
```

The next two header files were not used because the frame creation was not implemented. I still put them here for eventual future work.

Listing A.3: mac.hrl: defines the types and records for the frame creation.

```
1 % Implemented by Zofia Polkowska
2 -record(frame, {length :: integer(), mpdu :: mpdu(), fcs ::
       integer(), lqi :: integer()}).
3 -record(mpdu, {mhr :: mhr(), msdu :: msdu()}).
4 -record(mhr, {fc :: fc(), sn :: integer(), addr_fields ::
       addr_fields()}).
5 -record(fc, {t :: t(),
6              sec :: integer(),
7              pen :: integer(),
8              ack :: integer(),
9              intra :: integer(),
10             dst_m :: mode(),
11             src_m :: mode()}).
12 -record(addr_fields, {dst :: addr(), src :: addr()}).
13 -record(addr, {pan_id :: integer(), sadr :: binary(), eadr ::
       binary()}).
14
15
16 -type mpdu() :: #mpdu{}.
17
18 -type mhr() :: #mhr{}.
19
20 -type msdu() :: payload().
21
22 -type addr_fields() :: #addr_fields{}.
23
24 -type addr() :: #addr{}.
```

```
25
26 -type fc() :: #fc{}.
27
28 -type t() :: beacon | data | ack | command.
29
30 -type mode() :: integer().
31
32 -type payload() :: binary().
```

Listing A.4: frame.hrl: defines values to put into the frame control fields.

```
1 % this file contains some values to put in the FCF
2 %
    https://github.com/contiki-os/contiki/blob/master/core/net/mac/frame802154.h
3
4 -define(BEACONFRAME, 16#00).
5 -define(DATAFRAME, 16#01).
6 -define(ACKFRAME, 16#02).
7 -define(CMDFRAME, 16#03).
8
9 -define(BEACONREQ, 16#07).
10
11 -define(NOADDR, 16#00). % Only valid for ACK or Beacon frames.
12 -define(IEEERESERVED, 16#01).
13 -define(SHORTADDRMODE, 16#02).
14 -define(LONGADDRMODE, 16#03).
15
16 -define(NOBEACONS, 16#0F).
17
18 -define(BROADCASTADDR, 16#FFFF).
19 -define(BROADCASTPANDID, 16#FFFF).
20
21 -define(IEEE802154_2003, 16#00).
22 -define(IEEE802154_2006, 16#01).
23 -define(IEEE802154_2015, 16#02).
24
25 -define(SECURITY_LEVEL_NONE, 0).
26 -define(SECURITY_LEVEL_MIC_32, 1).
27 -define(SECURITY_LEVEL_MIC_64, 2).
28 -define(SECURITY_LEVEL_MIC_128, 3).
29 -define(SECURITY_LEVEL_ENC, 4).
30 -define(SECURITY_LEVEL_ENC_MIC_32, 5).
31 -define(SECURITY_LEVEL_ENC_MIC_64, 6).
32 -define(SECURITY_LEVEL_ENC_MIC_128, 7).
33
34 -define(IMPLICIT_KEY, 0).
35 -define(ONE_BYTE_KEY_ID_MODE, 1).
36 -define(FIVE_BYTE_KEY_ID_MODE, 2).
37 -define(NINE_BYTE_KEY_ID_MODE, 3).
```

A.2 Driver code

This code should be placed inside a folder called "src".

Listing A.5: mrf24j40_reg.erl: this is SPI driver implemented by Zofia. It allows to do read and write operations on the registers.

```
1  -module(mrf24j40_reg).
2  %Implemented by Zofia Polkowska
3
4  -include("include/reg.hrl").
5
6  %% R/W operations
7  -export([read/2,
8           read/4,
9           write/3,
10          write/4,
11          read_integer/2,
12          set_bit/3,
13          unset_bit/3,
14          poll/3,
15          poll/4]).
16
17 %% Helpers
18 -export([decode/2, fifo/2]).
19
20 -define(MODE, #{cpol => low, cpha => leading}).
21
22 -type register_addr() :: integer().
23 -type register_type() :: short | long.
24
25 %% --- R/W operations
       ------------------------------------------------------------------------
26 -spec read(Type :: register_type(), Addr :: register_addr() |
       [register_addr()]) -> binary().
27 read(Type, Registers) when is_list(Registers) ->
28     Values = lists:map(fun(R) -> read_integer(Type, R) end,
           Registers),
29     RegHex = lists:map(fun(R) -> integer_to_list(R, 16) end,
           Registers),
30     lists:zip(RegHex, Values);
31 read(long, Addr) ->
32     Bits = <<?LONG:1, Addr:10, ?R:1, ?Skip:4>>,
33     grisp_spi:send_recv(spi1, ?MODE, Bits, 2, 1);
34 read(short, Addr) ->
35     Bits = <<?SHORT:1, Addr:6, ?R:1>>,
36     grisp_spi:send_recv(spi1, ?MODE, Bits, 1, 1).
37
```

```
38 -spec read_integer(Type :: register_type(), Addr ::
      register_addr()) -> integer().
39 read_integer(Type, Addr) ->
40     Res = read(Type, Addr),
41     <<Decimal:8>> = Res,
42     Decimal.
43
44 read(_, _, Result, 0) ->
45     Result;
46 read(Type, Addr, Read, Length) ->
47     Part = read_integer(Type, Addr),
48     read(Type, Addr+1, <<Read/binary, Part:8>>, Length-1).
49
50 -spec write(Type :: register_type(), Addr :: register_addr(),
      Command :: integer() | binary()) -> binary().
51 write(Type, Addr, Command) when is_integer(Command) ->
52     write(Type, Addr, <<Command:8>>);
53 write(long, Addr, Command) ->
54     Bits = <<?LONG:1, Addr:10, ?W:1, ?Skip:4, Command/binary>>,
55     grisp_spi:send_recv(spi1, ?MODE, Bits);
56 write(short, Addr, Command) ->
57     Bits = <<?SHORT:1, Addr:6, ?W:1, Command/binary>>,
58     grisp_spi:send_recv(spi1, ?MODE, Bits).
59
60 write(_, _, <<>>, Addr) ->
61     Addr;
62 write(Type, Addr, <<Part:8, Rest/binary>>, Result) ->
63     write(Type, Addr, Part),
64     write(Type, Addr+1, Rest, <<Part:8, Result/binary>>).
65
66 -spec set_bit(Type :: register_type(), Addr :: register_addr(),
      Pos :: integer()) -> binary().
67 set_bit(Type, Addr, Pos) ->
68     <<Bit>> = <<0:(8 - Pos - 1), 1:1, 0:(Pos)>>,
69     Resp = read_integer(Type, Addr),
70     Cmd = Bit bor Resp,
71     write(Type, Addr, Cmd).
72
73 -spec unset_bit(Type :: register_type(), Addr :: register_addr(),
      Pos :: integer()) -> binary().
74 unset_bit(Type, Addr, Pos) ->
75     <<Bit>> = <<1:(8 - Pos - 1), 0:1, 1:(Pos)>>,
76     Resp = read_integer(Type, Addr),
77     Cmd = Bit band Resp,
78     write(Type, Addr, Cmd).
79
80 poll(Type, Addr, Expected) ->
81     poll(Type, Addr, Expected, 1000).
82
```

```
83 poll(Type, Addr, Expected, Timeout) ->
84     timer:sleep(100),
85     case read_integer(Type, Addr) of
86         Expected ->
87             ok;
88         Other ->
89             case Timeout of
90                 0 ->
91                     Other;
92                 _ ->
93                     poll(Type, Addr, Expected, Timeout - 100)
94             end
95     end.
96
97 %% --- Helpers
       ------------------------------------------------------------------------
98
99 fifo(From, Count) ->
100    lists:seq(From, From + Count - 1).
101
102 decode(B, N) ->
103    integer_to_list(binary:decode_unsigned(B), N).
```

Listing A.6: mrf24j40.erl: this is the driver code. This is the translation of mrf24j40.c in Contiki.

```
1 -module(mrf24j40).
2
3 -include("include/reg.hrl").
4 -include("include/settings.hrl").
5 -import(mrf24j40_reg,
6         [read/2, write/3, read_integer/2, set_bit/3, unset_bit/3,
             poll/3, poll/4]).
7
8 -export([init/0, prepare/2, transmit/0, write/2, read/0, cca/0,
9         receiving_packet/0, pending_packet/0]).
10
11 -export([get_packet/1]).
12
13 -export([set_panid/1, set_short_mac_addr/1,
       set_extended_mac_addr/1,
14         get_short_mac_addr/0, get_extended_mac_addr/0]).
15
16 -export([get_status/0, get_last_rssi/0, get_last_lqi/0]).
17
18 -export([set_short_add_mem/2, get_short_add_mem/1,
       set_long_add_mem/2, get_long_add_mem/1]).
19
20 -export([get_global/1, set_global/2]).
```

```
21
22 -export([reset_rf_state_machine/0]).
23
24 % Implementation of the MRF24J40 driver
25 % Using mrf24j40.c file of the contiki repository.
26 % https://github.com/contiki-os/contiki/blob/master/platform/
27 %       seedeye/dev/mrf24j40/mrf24j40.c
28 % header file can be found here:
    https://github.com/contiki-ng/contiki-ng/blob/develop/os/dev/radio.h
29
30 % ---------------- Global vars getter and setter ----------------
31
32 % Type is an atom : get_lqi, get_rssi, get_status, get_pending or
    get_receive
33 get_global(Type) ->
34     Pid = whereis(global_vars),
35     Pid ! {self(), Type},
36     receive
37         {val, Value} -> Value
38     end.
39
40 % Type is an atom : set_lqi, set_rssi, set_status, set_pending or
    set_receive
41 set_global(Type, Value) ->
42     Pid = whereis(global_vars),
43     Pid ! {self(), Type, Value}.
44
45 % ---------------------------------------------------------------
46
47 % ---------------- Useful functions-------------------------------
48
49 % Write Buf_len bytes of the buffer Buf at the address Addr
50 for_write(Addr, Buf, Buf_len) -> for_write(Addr, Buf, Buf_len, 0).
51 for_write(_, _, Buf_len, Buf_len) -> 0;
52 for_write(Addr, <<First:8, Rest/binary>>, Buf_len, I) when I <
    Buf_len ->
53     set_long_add_mem(Addr + I, First),
54     for_write(Addr, Rest, Buf_len, I + 1).
55
56 % Read Len bytes form the address Addr into Buf
57 for_read(Addr, Len) -> for_read(Addr, <<>>, Len, 0).
58 for_read(_, Buf, Len, Len) -> Buf;
59 for_read(Addr, Buf, Len, I) when I < Len ->
60     <<B:8>> = get_long_add_mem(Addr + I + 1),
61     for_read(Addr, <<Buf/binary, B>>, Len, I + 1).
62
63 % While loops that read the address Addr, do the "and" operation
    with And_with
64 % and stops if the result is Expected
```

```
65 while_short_and(Addr, Value, And_with, Expected) ->
66     case (Value band And_with) of
67         Expected -> ok;
68         _ ->
69             <<NewValue:8>> = get_short_add_mem(Addr),
70             while_short_and(Addr, NewValue, And_with, Expected)
71     end.
72
73 while_long_and(Addr, Value, And_with, Expected) ->
74     case (Value band And_with) of
75         Expected -> ok;
76         _ ->
77             NewValue = get_long_add_mem(Addr),
78             while_long_and(Addr, NewValue, And_with, Expected)
79     end.
80
81 % ---------------------------------------------------------------------
82
83 % ---------------- Read and write registers ----------------------
84
85 set_short_add_mem(Addr, Val) ->
86     grisp_gpio:configure(ss1, output_0), % set CSn pin low (pin 1
        of spi1 is called ss1)
87     write(short, Addr, Val),
88     grisp_gpio:configure(ss1, output_1). % set CSn pin high
89
90 set_long_add_mem(Addr, Val) ->
91     grisp_gpio:configure(ss1, output_0), % set CSn pin low
92     write(long, Addr, Val),
93     grisp_gpio:configure(ss1, output_1). % set CSn pin high
94
95 get_short_add_mem(Addr) ->
96     grisp_gpio:configure(ss1, output_0), % set CSn pin low
97     Ret = read(short, Addr),
98     grisp_gpio:configure(ss1, output_1), % set CSn pin high
99     Ret.
100
101 get_long_add_mem(Addr) ->
102     grisp_gpio:configure(ss1, output_0), % set CSn pin low
103     Ret = read(long, Addr),
104     grisp_gpio:configure(ss1, output_1), % set CSn pin high
105     Ret.
106
107 % ---------------------------------------------------------------------
108
109 % RF State Machine Reset
110 reset_rf_state_machine() ->
111     <<Rfctl:8>> = get_short_add_mem(?RFCTL),
112     set_short_add_mem(?RFCTL, (Rfctl bor 2#00000100)), % put the
```

```
        value 0x04 in RFCTL
113     set_short_add_mem(?RFCTL, (Rfctl band 2#11111011)), % put the
            value 0x00 in RFCTL
114     timer:sleep(3). % sleep during 2500 [us] (=> 3 [ms])
115
116 % Flushing RX_FIFO: resets the RX_FIFO Address Pointer,
117 flush_rx_fifo() ->
118     <<Rxflush:8>> = get_short_add_mem(?RXFLUSH),
119     set_short_add_mem(?RXFLUSH, (Rxflush bor 2#00000001)). % the
            last bit is set to 1
120
121 % Setting the channel by setting the RFCON0 register (datasheet
      p.63 + p.92)
122 % the first four bits are the channels number
123 % the last bits are the recommended value of 0x3
124 set_channel(Channel) ->
125     Ch = ((Channel - 11) bsl 4) bor 2#00000011,
126     set_long_add_mem(?RFCON0, Ch),
127     reset_rf_state_machine().
128
129 % Setting the MAC PAN ID
130 % PANIDL register contains the low byte
131 % PANIDH register contains the high byte
132 set_panid(Id) ->
133     <<IdH:8, IdL:8>> = Id,
134     set_short_add_mem(?PANIDL, IdL),
135     set_short_add_mem(?PANIDH, IdH).
136
137 % Setting the short MAC address (16 bits)
138 % SADRL register contains the low byte
139 % SADRH register contains the high byte
140 set_short_mac_addr(Addr) ->
141     <<AddrH:8, AddrL:8>> = Addr,
142     set_short_add_mem(?SADRL, AddrL),
143     set_short_add_mem(?SADRH, AddrH).
144
145 % Setting the extended MAC address (64 bits)
146 % EADR0 register contains the higher byte
147 % ...
148 % EADR7 register contains the lower byte
149 set_extended_mac_addr(Addr) ->
150     <<Addr0:8, Addr1:8, Addr2:8, Addr3:8, Addr4:8, Addr5:8,
            Addr6:8, Addr7:8>> = Addr,
151     set_short_add_mem(?EADR7, Addr7),
152     set_short_add_mem(?EADR6, Addr6),
153     set_short_add_mem(?EADR5, Addr5),
154     set_short_add_mem(?EADR4, Addr4),
155     set_short_add_mem(?EADR3, Addr3),
156     set_short_add_mem(?EADR2, Addr2),
```

```
157     set_short_add_mem(?EADR1, Addr1),
158     set_short_add_mem(?EADR0, Addr0).
159
160 % Getting the stored short MAC address
161 get_short_mac_addr() ->
162     <<(get_short_add_mem(?SADRH))/binary,
            (get_short_add_mem(?SADRL))/binary>>.
163
164 % Setting the stored extended MAC address
165 get_extended_mac_addr() ->
166     <<(get_short_add_mem(?EADR0))/binary,
167     (get_short_add_mem(?EADR1))/binary,
168     (get_short_add_mem(?EADR2))/binary,
169     (get_short_add_mem(?EADR3))/binary,
170     (get_short_add_mem(?EADR4))/binary,
171     (get_short_add_mem(?EADR5))/binary,
172     (get_short_add_mem(?EADR6))/binary,
173     (get_short_add_mem(?EADR7))/binary
174     >>.
175
176 % Setting transmitter power
177 set_tx_power(Pwr) ->
178     set_long_add_mem(?RFCON3, Pwr).
179
180 % Returns the MRF24J40 status
181 get_status() ->
182     get_long_add_mem(?RFSTATE).
183
184 % Returns the measured rssi value
185 % (use table 3-8 of the datasheet (p.96) to convert it to dBm)
186 get_rssi() ->
187     <<Value:8>> = get_short_add_mem(?BBREG6),
188     set_short_add_mem(?BBREG6, Value bor 2#10000000), % init RSSI
            calculation
189     poll(short, ?BBREG6, 2#00000001), % wait for complete
            calculation
190     <<Rssi:8>> = get_long_add_mem(?RSSI), % get the averaged RSSI
            value
191     set_global(set_rssi, Rssi), % store the value in the global
            variable
192     Rssi.
193
194 % Getting last rssi value recorded in the global variable
195 % (call to global_vars process)
196 % (use table 3-8 of the datasheet (p.96) to convert it to dBm)
197 get_last_rssi() ->
198     get_global(get_rssi).
199
200 % Getting last lqi value recorded in the global variable
```

```
201 % (call to global_vars process)
202 get_last_lqi() ->
203     get_global(get_lqi).
204
205 % Stores a buffer of Buf_len bytes in the TX_FIFO buffer
206 set_txfifo(_, Buf_len) when Buf_len == 0; Buf_len > 128 ->
207     -1;
208 set_txfifo(Buf, Buf_len) ->
209     set_long_add_mem(?TX_FIFO, 0),
210     set_long_add_mem(?TX_FIFO + 1, Buf_len), % write the buffer
        length
211     for_write(?TX_FIFO + 2, Buf, Buf_len). % write the buffer
212
213 % Retrieves a message stored in the RX_FIFO
214 get_rxfifo() ->
215     set_short_add_mem(?BBREG1, 2#00000100), % disable packet
        reception
216     <<L:8>> = get_long_add_mem(?RX_FIFO), % length
217     Len_read = L - 2, % minus 2 bytes (remove RSSI and LQI)
218     case Len_read =< ?PACKETBUF_SIZE of
219         true -> % correct length
220             Buf = for_read(?RX_FIFO, Len_read), % get the packet
221             Len = Len_read,
222             case ?ADD_RSSI_AND_LQI_TO_PACKET of % get LQI and
                RSSI values
223                 true -> <<Last_lqi:8>> =
                    get_long_add_mem(?RX_FIFO + Len + 3),
224                         set_global(set_lqi, Last_lqi),
225                         <<Last_rssi:8>> =
                            get_long_add_mem(?RX_FIFO + Len + 4),
226                         set_global(set_rssi, Last_rssi);
227                 false -> ok
228             end;
229         false ->
230             Buf = [],
231             Len = 0
232     end,
233
234     set_short_add_mem(?BBREG1, 2#00000000), % enable packet
        reception
235     set_global(set_pending, 0),
236
237     case ?PROMISCUOUS_MODE of
238         true -> flush_rx_fifo();
239         false -> ok
240     end,
241
242     Return = if Len == 0 -> {-1, Buf};
243                 true -> {Len, Buf} % return buffer + its length
```

```
244             end,
245     Return.
246
247 % Puts the radio in sleep mode
248 put_to_sleep() ->
249     grisp_gpio:configure(spi1_pin9, output_0), % set WAKE pin to 0
250     % enable Immediate Wake-up mode
251     set_short_add_mem(?WAKECON, 2#10000000),
252     set_short_add_mem(?SOFTRST, 2#00000100),
253     set_short_add_mem(?RXFLUSH, 2#01100000),
254     % put to sleep
255     set_short_add_mem(?SLPACK, 2#10000000).
256
257 % Turns on and sets the radio on receiving mode
258 wake() ->
259     grisp_gpio:configure(spi1_pin9, output_1), % set WAKE pin to 1
260     % Reset RF State Machine
261     set_short_add_mem(?RFCTL, 2#00000100),
262     set_short_add_mem(?RFCTL, 2#00000000),
263     timer:sleep(3).
264
265 % Turns on receiving mode if it was off
266 on() ->
267     Receive_on = get_global(get_receive),
268     case Receive_on of
269         0 -> wake(),
270             set_global(set_receive, 1);
271         1 -> ok
272     end,
273     1.
274
275 % Turns off receiving mode if it was on
276 off() ->
277     Receive_on = get_global(get_receive),
278     case Receive_on of
279         1 -> set_global(set_receive, 0),
280             put_to_sleep();
281         0 -> ok
282     end,
283     1.
284
285 %
    ------------------------------------------------------------------------
286
287 % Initializes the radio tranceiver
288 % (more details in contiki code, following datasheet init
    procedure (p.90))
289 init() ->
290     % hard reset
```

```
291     grisp_gpio:configure(spi1_pin8, output_0), % set RSTn pin to 0
292     timer:sleep(3),
293     grisp_gpio:configure(spi1_pin8, output_1), % set RSTn pin to 1
294     timer:sleep(3),
295
296     set_short_add_mem(?SOFTRST, 2#00000111), %soft reset
297     % wait until reset is complete
298     <<Value:8>> = get_short_add_mem(?SOFTRST),
299     while_short_and(?SOFTRST, Value, 2#0000111, 0),
300     timer:sleep(3),
301
302     set_short_add_mem(?PACON2, 2#10011000),
303     set_channel(?DEFAULT_CHANNEL), % set channel
304     set_long_add_mem(?RFCON1, 2#00000010),
305     set_long_add_mem(?RFCON2, 2#10000000), % enable PLL
306     set_tx_power(2#00000000), % set power to 0
307     set_long_add_mem(?RFCON6, 2#10010000),
308     set_long_add_mem(?RFCON7, 2#10000000),
309     set_long_add_mem(?RFCON8, 2#00000010),
310     set_long_add_mem(?SLPCON1, 2#00100001),
311
312     set_short_add_mem(?BBREG2, 2#01111000),
313     set_short_add_mem(?CCAEDTH, 2#01100000),
314
315     % ifdef MRF24J40MB
316     case ?MRF24J40MB of
317         true -> set_long_add_mem(?TESTMODE, 2#0001111);
318         false -> ok
319     end,
320
321     % ifdef ADD_RSSI_AND_LQI_TO_PACKET
322     case ?ADD_RSSI_AND_LQI_TO_PACKET of
323         true -> set_short_add_mem(?BBREG6, 2#01000000);
324         false -> ok
325     end,
326
327     <<Value2:8>> = get_long_add_mem(?RFSTATE),
328     while_long_and(?RFSTATE, Value2, 16#A0, 16#A0),
329
330     % #ifdef DISABLE_AUTOMATIC_ACK
331     case ?DISABLE_AUTOMATIC_ACK  of
332         true -> I1 = 0 bor 2#00100000;
333         false -> I1 = 0
334     end,
335
336     % #ifdef PAN_COORDINATOR
337     case ?PAN_COORDINATOR of
338         true -> I2 = I1 bor 2#00001000,
339                 set_short_add_mem(?ORDER, 2#11111111);
```

```
340         false -> I2 = I1
341     end,
342 
343     % #ifdef COORDINATOR
344     case ?COORDINATOR of
345         true -> I3 = I2 bor 2#00000100;
346         false -> I3 = I2
347     end,
348 
349     % #ifdef ACCEPT_WRONG_CRC_PKT
350     case ?ACCEPT_WRONG_CRC_PKT of
351         true -> I4 = I3 bor 2#00000010;
352         false -> I4 = I3
353     end,
354 
355     %#ifdef PROMISCUOUS_MODE
356     case ?PROMISCUOUS_MODE of
357         true -> I5 = I4 bor 2#00000001;
358         false -> I5 = I4
359     end,
360 
361     set_short_add_mem(?RXMCR, I5),
362     set_short_add_mem(?TXMCR, 2#00011100),
363 
364     set_short_add_mem(?TXSTBL, 2#10010101),
365     set_short_add_mem(?TXTIME, 2#00110000),
366 
367     % starting the different processes
368     global_vars:start(),
369     mailbox:start(),
370     interruption_process:start(),
371     receiver_process:start(),
372     wait_status:start(),
373 
374     reset_rf_state_machine(),
375     flush_rx_fifo(),
376 
377     0.
378 
379 % Write Data of length Len in TX_FIFO
380 prepare(Data, Len) ->
381     Receive_was_on = get_global(get_receive),
382     on(),
383     set_txfifo(Data, Len),
384 
385     case Receive_was_on of
386         0 -> off();
387         1 -> ok
388     end,
```

```
389
390     0.
391
392 % Asks wait_status process if transmission is over and waits for
        an answer
393 % Function called by transmit().
394 transmission_status() ->
395     Pid = whereis(wait_status),
396     Pid ! {self(), status},
397     receive
398         {transmission_over} -> ok
399     end.
400
401 % Transmits the message waiting in TX_FIFO
402 transmit() ->
403     Receive_was_on = get_global(get_receive),
404     on(),
405     set_global(set_status, ?TX_WAIT),
406     set_short_add_mem(?TXNCON, 2#00000001), % transmit
407
408     transmission_status(), % wait until transmission is over
409
410     case Receive_was_on of
411         0 -> off();
412         1 -> ok
413     end,
414
415     % look at the transmission status
416     Status_tx = get_global(get_status),
417     case Status_tx of
418         ?TX_ERR_NONE -> ?RADIO_TX_OK;
419         ?TX_ERR_COLLISION -> ?RADIO_TX_COLLISION;
420         ?TX_ERR_MAXRETRY -> ?RADIO_TX_NOACK;
421         _ -> ?RADIO_TX_ERR
422     end.
423
424 % Prepares and transmits Data of length Len
425 write(Data, Len) ->
426     Prepare = prepare(Data, Len),
427     case Prepare of
428         0 -> transmit();
429         _ -> -1
430     end.
431
432 % Reads the received message
433 read() ->
434     {Len, Data} = get_rxfifo(),
435     io:format("Length: ~p~n", [Len]),
436     Data.
```

```
437
438 % Getting the messages stored in the mailbox
439 % Type is an atom:
440 % - get_next for the oldest message
441 % - get_all for the list of all messages in arrival order
442 get_packet(Type) ->
443     Pid = whereis(mailbox),
444     Pid ! {self(), Type},
445     receive
446         {_, Packet} -> Packet
447     end.
448
449 % Clear Channel Assessment
450 cca() ->
451     Receive_was_on = get_global(get_receive),
452     on(),
453     Ret = get_rssi() =< 95,
454     case Receive_was_on of
455         0 -> off();
456         1 -> ok
457     end,
458
459     Ret.
460
461 receiving_packet() ->
462     0.
463
464 % Getting the pending value (in the global_vars process)
465 pending_packet() ->
466     get_global(get_pending).
```

Listing A.7: global_vars.erl: the process managing the global variables.

```
1  -module(global_vars).
2  -include("include/settings.hrl").
3  -export([start/0, global_vars/5]).
4
5  % This module contains the process that manages the global
       variables
6  % used in mrf24j40.erl
7
8  global_vars(Last_lqi, Last_rssi, Status_tx, Pending, Receive_on)
       ->
9      receive
10         {_, set_lqi, Value} ->
11             global_vars(Value, Last_rssi, Status_tx, Pending,
                   Receive_on);
12         {_, set_rssi, Value} ->
13             global_vars(Last_lqi, Value, Status_tx, Pending,
```

```
                Receive_on);
14          {_, set_status, Value} ->
15              global_vars(Last_lqi, Last_rssi, Value, Pending,
                    Receive_on);
16          {_, set_pending, Value} ->
17              global_vars(Last_lqi, Last_rssi, Status_tx, Value,
                    Receive_on);
18          {_, set_receive, Value} ->
19              global_vars(Last_lqi, Last_rssi, Status_tx, Pending,
                    Value);
20
21          {From, get_lqi} ->
22              From ! {val, Last_lqi},
23              global_vars(Last_lqi, Last_rssi, Status_tx, Pending,
                    Receive_on);
24          {From, get_rssi} ->
25              From ! {val, Last_rssi},
26              global_vars(Last_lqi, Last_rssi, Status_tx, Pending,
                    Receive_on);
27          {From, get_status} ->
28              From ! {val, Status_tx},
29              global_vars(Last_lqi, Last_rssi, Status_tx, Pending,
                    Receive_on);
30          {From, get_pending} ->
31              From ! {val, Pending},
32              global_vars(Last_lqi, Last_rssi, Status_tx, Pending,
                    Receive_on);
33          {From, get_receive} ->
34              From ! {val, Receive_on},
35              global_vars(Last_lqi, Last_rssi, Status_tx, Pending,
                    Receive_on);
36
37          _ ->
38              global_vars(Last_lqi, Last_rssi, Status_tx, Pending,
                    Receive_on)
39      end.
40
41  start() ->
42      io:format("Starting global variables process...~n"),
43      Pid = spawn(?MODULE, global_vars, [0, 0, ?TX_ERR_NONE, 0, 1]),
44      register(global_vars, Pid).
```

Listing A.8: interruption_process.erl: the process polling the interruption register and taking decisions accordingly.

```
1 -module(interruption_process).
2 -include("include/reg.hrl").
3 -include("include/settings.hrl").
4 -import(mrf24j40, [read/0, get_short_add_mem/1, set_global/2]).
```

```
5
6 -export([start/0, interruption_process/0]).
7
8 %% When a packet is received, an interruption is issued and we
      can tell the
9 %% receiver_process that he can read the buffer.
10 %% When transmission is over, an interrupt (INTSTAT) is issued
      and we can
11 %% then look in TXSTAT if the transmission was successful or not
12 %% (ISR and PROCESS_THREAD in Contiki)
13 interruption_process() ->
14     Int_status = get_short_add_mem(?INTSTAT), % reads the
          interruption register
15     Zero = <<0>>,
16     case Int_status of
17         Zero -> % if no interruption
18             timer:sleep(1),
19             interruption_process();
20         _ -> % an interruption occured
21             <<_:4, RXIF:1, _:2, TXNIF:1>> = Int_status,
22
23             case RXIF of
24                 1 -> % a packet was received
25                     io:format("Interruption_process: can read~n"),
26                     set_global(set_pending, 1),
27                     % tells to receiver_process that he can read
28                     ReceiverPid = whereis(receiver_process),
29                     ReceiverPid ! {self(), read},
30                     receive
31                         {ReceiverPid, done} -> ok
32                     end;
33                 0 ->
34                     ok
35             end,
36
37             case TXNIF of
38                 1 -> % a packet was transmitted
39                     io:format("Interruption_process: transmission
                          over~n"),
40                     Tx_status = get_short_add_mem(?TXSTAT),
41                     <<TXNRETRY:2, CCAFAIL:1, TXG2FNT:1,
                          TXG1FNT:1, TXG2STAT:1,
42                     TXG1STAT:1, TXNSTAT:1>> = Tx_status,
43                     case TXNSTAT of % looking at the transmission
                          status
44                         1 ->
45                             case CCAFAIL of
46                                 1 -> set_global(set_status,
                                        ?TX_ERR_COLLISION);
```

```
47                                      0 -> set_global(set_status,
                                            ?TX_ERR_MAXRETRY)
48                                  end;
49                              0 -> set_global(set_status, ?TX_ERR_NONE)
50                          end,
51
52                          % tells to wait_status that the transmission
                                is over
53                          WaitStatusPid = whereis(wait_status),
54                          WaitStatusPid ! {self(), transmission_over};
55                      0 -> ok
56                  end,
57                  timer:sleep(1),
58                  interruption_process()
59      end.
60
61  start() ->
62      io:format("Starting interruption_process...~n"),
63      Pid = spawn(?MODULE, interruption_process, []),
64      register(interruption_process, Pid).
```

Listing A.9: mailbox.erl: the process managing the mailbox (where the packets are put when they are received).

```
1   -module(mailbox).
2   -export([start/0, mailbox/1]).
3
4   %% Stores the received packets (read from rx_fifo) until they are
        retrieved.
5   %% This replaces the buffer used in contiki to store a packet.
6   mailbox(Mailbox) ->
7       receive
8           {_, new_packet, Packet} -> % receives new packet
9               io:format("Received packet: ~p~n", [Packet]),
10              mailbox([Packet | Mailbox]);
11          {From, get_next} -> % get the oldest message
12              case Mailbox of
13                  [] -> % if mailbox is empty
14                      From ! {next_packet, "No packet to get"},
15                      mailbox(Mailbox);
16                  _ ->
17                      [Next | Rest] = lists:reverse(Mailbox),
18                      From ! {next_packet, Next},
19                      mailbox(lists:reverse(Rest))
20              end;
21          {From, get_all} -> % get a list of all message in arrival
                order
22              Reverse = lists:reverse(Mailbox),
23              From ! {all_packets, Reverse},
```

```
24              mailbox([]);
25          {_, clear_all} ->
26              mailbox([]);
27          _ ->
28              mailbox(Mailbox)
29      end.
30
31
32  start() ->
33      io:format("Starting mailbox...~n"),
34      Pid = spawn(?MODULE, mailbox, [[]]),
35      register(mailbox, Pid).
```

Listing A.10: receiver_process.erl: the process reading the reception buffer when a packet has arrived.

```
1  -module(receiver_process).
2  -import(mrf24j40, [get_global/1, read/0]).
3
4  -export([start/0, receiver_process/1]).
5
6  %% Reads the received packet in RX_FIFO only if the interruption
       process
7  %% told to do so
8  receiver_process(MailboxPid) ->
9      receive
10         {InterruptionPid, read} -> % the receiver can read
11             Packet = read(),
12
13             InterruptionPid ! {self(), done}, % tells
                   interrupt_proc. the reading is done
14             MailboxPid ! {self(), new_packet, Packet}, % sends
                   packet to mailbox
15
16             TestPid = whereis(receiver_test), % the process
                   testing the receiver
17             case TestPid of
18                 undefined -> ok; % if we are not testing
19                 _ -> TestPid ! {self(), new_packet, Packet} %
                       sends packet to receiver_test
20             end,
21
22             receiver_process(MailboxPid);
23         _ ->
24             receiver_process(MailboxPid)
25     end.
26
27 start() ->
28     io:format("Starting receiver_process...~n"),
```

```
29      MailboxPid = whereis(mailbox),
30      Pid = spawn(?MODULE, receiver_process, [MailboxPid]),
31      register(receiver_process, Pid).
```

Listing A.11: wait_status.erl: the process waiting for the transmission to be over.

```
1  -module(wait_status).
2  -export([start/0, wait_status/4]).
3
4  % Manages the message passing to tell if the packet transmission
       is over
5  % Receives the status from the interruption process and tells it
       to
6  % "transmission_status()" who asked for it (in mrf24j40.erl).
7  wait_status(Status, Need, Pid, PidI) ->
8      receive
9          {PidT, status} -> % status asked by transmission_status
10             case Status of
11                 unset -> wait_status(Status, need, PidT, PidI);
12                 set ->
13                     PidT ! {transmission_over},
14                     wait_status(unset, no_need, 0, PidI)
15             end;
16         {PidI, transmission_over} -> % transmission is over
17             case Need of
18                 no_need -> wait_status(set, Need, Pid, PidI);
19                 need ->
20                     Pid ! {transmission_over},
21                     wait_status(unset, no_need, 0, PidI)
22             end;
23         _ -> wait_status(Status, Need, Pid, PidI)
24     end.
25
26 start() ->
27     io:format("Starting wait_status...~n"),
28     InterruptionPid = whereis(interruption_process),
29     Pid = spawn(?MODULE, wait_status, [unset, no_need, 0,
           InterruptionPid]),
30     register(wait_status, Pid).
```

Listing A.12: frame.erl: this is the hard coding of the packets used in the tests.

```
1  -module(frame).
2
3  -export([makeFrame/0]).
4
5  %-include("include/frame.hrl").
6
7  % Frame Control:
8  % FrameType = <<1:3>>, % data = 001
```

```
% SecurityEnabled = <<0:1>>, % no security
% FramePending = <<0:1>>, % no pending frame ?
% AckRequest = <<0:1>>, % no Ack requested
% IntraPAN = <<0:1>>, % source/dest addr + pan id must be present
% Reserved1 = <<0:3>>,
% DestAddrMode = <<3:2>>, % 64-bit extended address
% Reserved2 = <<0:2>>,
% SourceAddrMode = <<3:2>>. % 64-bit extended address

makeFrame() ->
  FrameType = 1,
  SecurityEnabled = 0,
  FramePending = 0,
  AckRequest = 0,
  IntraPAN = 0,
  Reserved1 = 0,
  DestAddrMode = 3,
  Reserved2 = 0,
  SourceAddrMode = 3,

  <<R1:2, R2:1>> = <<Reserved1:3>>,

  <<FrameControl:16>> = <<R2:1, IntraPAN:1, AckRequest:1,
   FramePending:1, SecurityEnabled:1, FrameType:3,
              SourceAddrMode:2, Reserved2:2, DestAddrMode:2,
   R1:2>>,

  SequenceNumber = 1, % 8 bits

  % DestPAN = 16#FFFF, % 16 bits, FFFF is for broadcasting
  % DestAddr = 16#DCBA, % DestAddrMode = 3 -> 64 bits
  % SourcePAN = 16#ABCD, % 16 bits
  % SourceAddr = 16#ABCD, % SourceAddrMode = 3 -> 64 bits

  % addresses need to be reversed ?? -> endianess
  DestPAN = 16#FFFF,
  DestAddr = 16#BADC000000000000,
  SourcePAN = 16#CDAB,
  SourceAddr = 16#CDAB000000000000,

  <<AddressingFields:160>> = <<DestPAN:16, DestAddr:64,
   SourcePAN:16, SourceAddr:64>>,

  Payload = 16#2A002A, % minimum 6 bits

  % FCS = , % 16 bits automatically computed by the chip ?

  Frame = <<FrameControl:16, SequenceNumber:8,
   AddressingFields:160, Payload:24>>,
```

```
54   Length = erlang:ceil(erlang:bit_size(Frame) / 8),
55   {Frame, Length}.
56
57 % Normal frame:
58 % <<1,204,1,255,255,186,220,0,0,0,0,0,0,
59 % 205,171,205,171,0,0,0,0,0,0,42,0,42>>
60 % Length = 26
```

A.3 Performance evaluation

Listing A.13: receiver_test.erl: the process testing the performance on the receiver side.

```
1 -module(receiver_test).
2
3 -export([start/2, receiver_test/5]).
4
5 % This process allows to test the performance of the driver in
      the receiving part
6 % Frame is the expected packet sent by the transmitter,
7 % ExpectedCount is the number of packets expected to be received
8 % Returns: Count: the total number of received packets
9 %      CorrectCount: the number of correct packets received,
10 %     EndTime: the time in millisecond when all packets are
      received
11 receiver_test(ReceiverPid, Frame, ExpectedCount, Count,
      CorrectCount) ->
12   receive
13     {ReceiverPid, new_packet, Packet} ->
14       Cnt = Count + 1,
15       case Packet of
16         Frame -> CorrCount = CorrectCount + 1; % if the packet is
      the expected one
17         _ -> CorrCount = CorrectCount
18       end,
19
20       case Cnt of
21         ExpectedCount -> % all messages have been received
22           EndTime = os:system_time(millisecond), % the current
      time in millisecond
23
      io:format("----------------------Results---------------------~n"),
24           io:format("Count: ~p, CorrectCount: ~p, EndTime: ~p~n",
      [Cnt, CorrCount, EndTime]),
25
      io:format("--------------------------------------------------~n"),
26           exit(normal);
```

```
27          _ ->
28            receiver_test(ReceiverPid, Frame, ExpectedCount, Cnt,
      CorrCount)
29        end;
30      _ -> receiver_test(ReceiverPid, Frame, ExpectedCount, Count,
      CorrectCount)
31    end.
32
33 start(Frame, ExpectedCount) ->
34    io:format("Starting receiver_test...~n"),
35    ReceiverPid = whereis(receiver_process),
36    Pid = spawn(?MODULE, receiver_test, [ReceiverPid, Frame,
       ExpectedCount, 0, 0]),
37    register(receiver_test, Pid).
```

Listing A.14: sender_test.erl: the process testing the performance on the sender side.

```
1 -module(sender_test).
2
3 -export([sender_test/3]).
4
5 % This function allows to test the performance of the driver in
      the sending part
6 % Sending the frame "Frame" of length "Length" N times
7 % Displays the time it took in seconds and the start time (needed
      with receiver_test)
8 sender_test(Frame, Length, N) ->
9     StartTime = os:system_time(millisecond), % get current time
          in millisecond
10    sender_test(Frame, Length, N, 0), % loop
11    EndTime = os:system_time(millisecond),
12    TotalTime = (EndTime - StartTime) / 1000,
13    io:format("----------------------Results---------------------~n"),
14    io:format("StartTime: ~p, TotalTime: ~p~n", [StartTime,
          TotalTime]),
15    io:format("--------------------------------------------------~n").
16
17 sender_test(_, _, N, N) -> ok;
18 sender_test(Frame, Length, N, Count) when Count < N ->
19    mrf24j40:write(Frame, Length),
20    sender_test(Frame, Length, N, Count + 1).
```

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