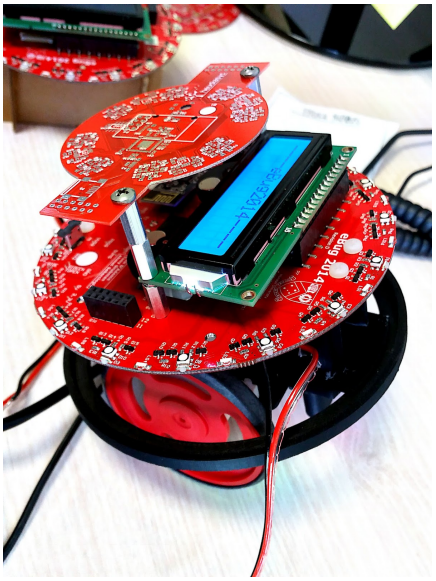

Research Project
Robots eBugs swarm

Di Vita Rémi - Morgan Pfister

TX

Spring 2017

Supervisors :
M. Ahmet SEKERCIOGLU
Mrs Ariane SPAENLEHAUER

utc
Université de Technologie
Compiègne

Contents

List of Figures

Chapter 1

Introduction

The goal of this course is to realize a technical and practical project involving an engineering approach.

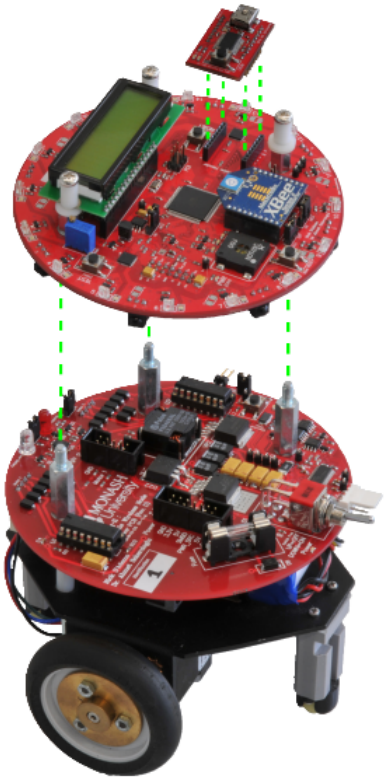

Figure 1.1: eBug Exploded View

The project we chose to participate in deals with real time algorithms in a network composed of a robots swarm. We chose this subject because it perfectly matches with our studies. Both of us are currently studying computer science in the field of real time systems and embedded systems. We thought that having a concrete experience was a perfect way of applying what we learned in other courses.

1.1 The eBugs

Our work is based on the *eBugs*, which are low-cost (around $500 each) and open robotics platforms designed for undergraduate teaching and academic research in multiple fields. This type of robot was first designed in 2008 at the Monash University in Australia, and they were conceived so that they are highly modular and can be used in several applications. On the figure 1.1, we can see the different components of the robot. Different layers can be attached to the robots depending on the desired application.

At first, they were presented to us as *dancing robots*. The goal was to turn on many of them and send them

commands to make them move on a specific area surrounded by cameras. Also, we imagined a lot of possible industrial applications. They could for example go where man can't go in the core of a nuclear plant, or they could be used as night watchmen in supermarkets... Thanks to their long battery life and their ability to travel on various surfaces, they are perfectly adaptable to any kind of application.

1.2 The subject

In order to be able to send commands to the eBugs, we first need to know exactly and all the time where they are in a specified area. To do so, a camera is placed over the robots and is able to detect them. This information is sent to the computer, where a map of the area is calculated along with the position of the eBugs. For now, there is only one camera, and it must be placed above the robots, but in the future, we can imagine a larger area covered by several cameras to track many eBugs.

The devices are all equipped with the same chip **nrf2401**[1]. This chip is a radio transceiver with a very low current consumption ($\simeq 10.5mA$). It is used as a wireless data communication system : data can be exchange in real time in the form of *packets* between the different devices, and we will explain later the exact functioning of the communication protocol. The problem we have to tackle with in this project lies in the communication between the robots, the camera(s) and the computer. As the communication is wireless, the main problem is the loss of packets, which can cause the eBug detection to work very bad. Our job is to look for optimized algorithms to manage the exchange of those packets in the network. The development has to be made in Python, under a Linux environment. A first program is already existing, and it constitutes the starting point of our job.

1.3 Conduct of the project

As we were 5 students interested in working on that project, we made 3 different groups :

- Gauthier Barbereau and Tanguy Le Flock are both working the development of a visualization tool software to display the position of the robots in real time.
- Lucas Yining Cai works on a multicamera visual sensor tracking system
- Our group, **Rémi Di Vita and Morgan Pfister**

As we were working on the same project, we had to share our work with the other groups, and especially with Gauthier and Tanguy as they sometimes had to work on the

[1] `https://www.sparkfun.com/datasheets/RF/nRF2401rev1_1.pdf`

same files as us. In order to share our work, we used *GitHub* which is a file versioning software. This allowed the different groups to work on the same project on different branches, and then merge what we achieved to do. The whole project and resources can be found here : `https://github.com/monash-wsrn/nrf-bridge`.

During the whole semester, we came every Tuesday afternoon in the Computer Science building to work on the project for 4 hours. We had access to a room with a computer in which we could use eBugs for our work. Every two weeks or so we had a meeting with Ahmet and Ariane to explain the progress of our work.

1.4 Acknowledgements

We would like to address a special thank you to Ahmet Sekercioglu for his precious help, availability and receptiveness during the whole project. His enthusiasm and the trust he placed in us made our work more efficient. We also would like to thank Ariane Spaenlehauer who allowed us to take part in this very promising project.

Chapter 2

Project

2.1 Learn about existing resources

As students of an engineering school, we are used to developing some projects from its beginning to its end. This practice is far from the project management methods used in larger companies. Being alone or in a reduced team for the development of a project imposes much less constraint. For example, it is rare for us to prepare a documentation for a school project. In that case, this is not a priority and we usually prefer to focus on the development itself. Moreover we don't have to think of the future of our project, when the grade is given, the project is over. That is why there is courses like the TX. Its allow the students to work in different condition with different requirements.

In this course which deal with the implementation of a wireless network for a robot fleet, we had to use an existing project base, and work on it in order to improve it. The eBugs project is larger than the ones we use to work on and when we started working on it, it was already well advanced. We had hundreds of line of code to understand and there wasn't a lot of documentation.

With the eBugs, the camera and the control station we had at our disposal, we started by testing the functions that we thought were the most important. Small scripts such as *discover.py* allowed us to have a better understanding of the global architecture of the project. A Python library named *ipdb* which work with the python interpreter *ipython* helped us a lot. This library is usually used to debug a Python script. It allows the user to place break points and open a Python interpreter with the scope of the script at the break point. For example, let's suppose we have the following program :

test.py

```
1  import random
2  import ipdb
3
4  def is_even_number(number):
5      if number % 2 == 0:
6          return True
7      return False
8
9  unknow_number = random.randrange(100)
10 ipdb.set_trace() # places a break point
11
12 if is_even_number(unknow_number):
13     print('action_1')
14 else:
15     print('action_2')
```

When we launch this program, it stops after the line *ipdb.set_trace()* and displays a command prompt of a Python interpreter.

```
remid-MPB:Desktop remid$ python3 test.py
> /Users/remid/Desktop/test.py(13)<module>()
     12
---> 13 if is_even_number(unknow_number):
     14     print('action_1')

ipdb>
```

Figure 2.1: Packets headers and commands

Then we can evaluate expressions or functions to debug the program :

```
ipdb> unknow_number
93
ipdb> is_even_number(unknow_number)
False
```

Figure 2.2: Packets headers and commands

Finally, it is possible to let the program continue its progress or watch it step by step :

```
ipdb> c
action_2
remid-MPB:Desktop remid$
```

Figure 2.3: Packets headers and commands

We mainly use this library in order to understand how the program was working, to follow function calls and analyze its returning data. However, it has been also really useful when we converted the whole project to Python 3. Indeed, a common decision was made to go through the code of the project and to make the necessary changes to support the recent updates on the Python language.

2.2 Communication Protocol

The first thing we had to do in this project was to understand the already existing code, and especially how the communication between the eBugs, the camera(s) and the control station exactly works. To do so, we first investigated the *nrf-bridge.py* file where a lot of functions were already written and seemed to correspond to what we were looking for. After running a few tests, we quickly discovered how to initiate the communication with the robots, and how to send them commands. Both the robots and the control station (the computer) are equipped with a wireless chip (**nrf2401**) that allows us to setup the communication. Every device connected with this chip can play the role of a transmitter or a receiver. The data transmitted on the wireless channel are in the form of a **packet**. Those packets are all composed in the same way, with hexadecimal numbers :

- The first two digits indicate the **packet type**, *i.e.* whether the packet targets one device (**unicast** packet), or every devices (**broadcast** or **multicast** packet).
- Then the following two digits correspond to the command we wish to send to the target. For example, if we want to print a message on the bottom line of the screen of an eBug, the code is **0x11**.
- Finally, we can add a **payload** to the packet which is optional information depending on the command sent. For example, if we want to print on the bottom line of the screen of an eBug, the payload is basically the message we wish to write.

After going through the firmware of the nrf2401 chip thanks to **PSoC Creator**, we extracted every commands that could be sent to either a camera or an eBug (see figure 4.1).

The communication protocol can be splitted into two different steps : (the figure 2.4 sums up the whole process)

- First, we have to initiate the communication with the robots. That means we need to determine which robots are turned on. To do so, we send a specific **broadcast** packet that checks every device equipped with the same chip around. Those devices then respond to this packet by sending back their MAC-addresses. A MAC-address is unique for each device and composed of a succession of 6 numbers. Those number are used when we want to send a command to one specific eBug. In order to lighten the size of the packets, a simple digit is associated to every detected MAC-addresses. Then, when we want to target a specific eBug, we no longer have to use a long and complicated address but only a key number.
- Once the global recognition is made, we can communicate to the powered-on ones. The first thing to do is to target the device we want to communicate with by sending

a specific packet, with the right key number. Then we can send the desired command to this device, which will only be processed by it and others will ignore it.

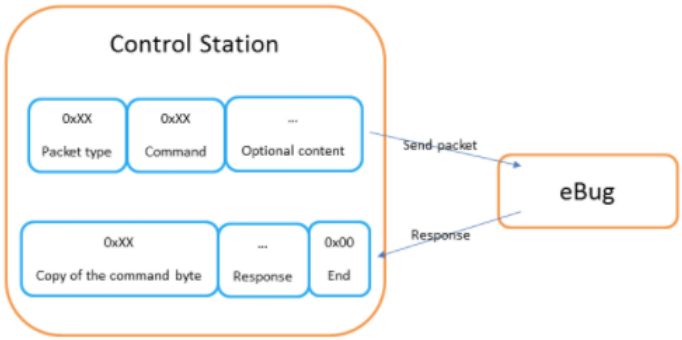

Figure 2.4: The communication protocol between the control station and an eBug

2.3 Light up eBugs

In order to be able to track the robots in a specific area, we first must be able to detect each robots separately. This is made thanks to a camera positioned above the robots. All the robots are equipped with a set of 16 LEDs arranged to form a circle on the edge of a specific layer. The firmware of the camera is designed so that it can properly detect those LEDs on the robots. Then, it sends the information of the position, colour and size of each perceived LEDs in the form of *blobs* (see section 2.5 for a more detailed explanations) to the control station.

One of the most important thing is the eBug recognition. We absolutely need to know which set of LEDs correspond to which eBug in order to know which robots are turned on, their real time position and to which robot we can send commands. The tracking of the robot is performed using this set of LEDs on each eBug. Each pattern is unique and correspond to one particular eBug. Moreover, the coloration is used to determine the overall orientation of the eBug.

Theory

Choosing a good pattern for the LEDs is the same as resolving a coloration problem, especially by using De Bruijn graphs. The principle is simple : we want a sequence of X adjacent LEDs to appear only on one of the eBugs. Indeed, even if the robots have a set of 16 LEDs, we don't have to perceive them all to determine which eBug it is.

Définition 2.1
A n-dimensional De Bruijn graph of m symbols is an directed graph representing overlaps between sequences of symbols. It has m^n vertices consisting of all possible length-n sequences of the given symbols.

The vertices of a de Bruijn graph dB(m, n) represent words of length m over an alphabet of size n.

The de Bruijn graphs are constructed following the instruction : considering **u** and **v** two vertices, draw an edge from **u** to **v** if the word obtained by removing the first letter of **u** is the same as the word obtained by removing the last letter of **v**.

Examples

- **n = 2, m = 1** :

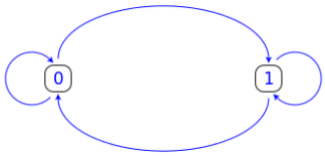

Figure 2.5: dB(2,1)

- **n = 2, m = 2** :

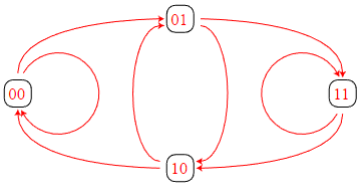

Figure 2.6: dB(2,2)

- **n = 2, m = 3** :

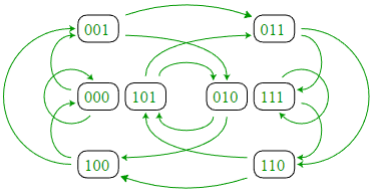

Figure 2.7: dB(2,3)

In our problem, we have an alphabet containing the following : **{R, G, B}** for Red, Blue and Green which are the three possible colors for one LED on the eBugs, and we want to define a sequence of 16 LEDs. The maths behind are quite complicated[1], but *in fine*, a valid colouring for our problem can be reduced to searching a *4*-cycle in the de Bruijn graph dB(3, 4) for example.

[1] `http://titania.ctie.monash.edu.au/papers/de_bruijn_cycles.pdf`

In practice

After we found some valid colouring for our eBugs, we need to light the LEDs up.

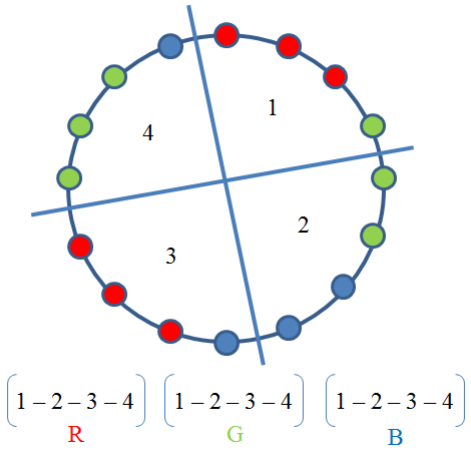

Figure 2.8: How to light up eBugs

As it is shown in the figure 2.8, the circle formed by the LEDs is splitted into 4 quarters. Thus each quarter represent a set of 4 consecutive LEDs. Now assume that we affect a bit to each LED, the binary number formed by the LEDs (clockwise direction) of the same color is a number from 0 to 15 and thus can be represented with a hexadecimal code. For each color, there are 4 hexadecimal number corresponding to the codes of the colour in each quarter.

For the colouring in the figure 2.8, the results would be :

- Red : [0x70e0]
- Green : [0x8307]
- Blue : [0x0c18]

2.4 eBugs settings

In order to work with eBugs, it is important to detect and set up every devices that are currently powered-on and ready to communicate on the network.
The function *nrf.assign_adresses()* is in charge of these initialization. It sends a special packet readable by all eBugs. Then they respond with a packet which contains their MAC-addresses. The control station assigns a simpler ID called *Psoc_id* to each eBug in order to lighten the communication packets.

However, the unique Psoc_id of an eBug depends on its response time during this assignment. This is not resilient at all. It could be a problem if an error occurs during the execution of program, and by restarting the assignment, the eBugs don't keep their previous Psoc_id. To solve this problem, we decided to save all eBugs MAC-addresses in a file and then associate them to a unique Psoc_id.

We made a new function named *assign_static_addresses()* which takes the path to the settings file as an optional argument. At the beginning, this function used the library **Pickle** to read the settings file if it exists, and directly loads a setting variable from it. Then the function assigns Psoc_id according to this variable. If there is a new MAC-address, an interface requests a new Psoc_id for this MAC-address. Finally, we save the new settings variable updated by the user in the settings binary file. Although it was functional, this method was wrong because the interface to register a new device can interrupt the normal course of a higher-level program. Furthermore if an error is made when registering a new device, it's impossible to correct it. Indeed, Pickle saves the data in a binary file like shown in the figure 2.9, which is not human readable.

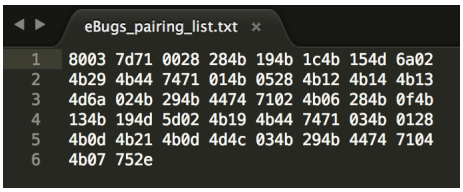

Figure 2.9: Binary eBugs settings file

To solve this new problem, we could have developed an interface to edit the binary file but it would have been long and not very relevant. We preferred to change the way of storing settings and instead of using binary file, we are now using JSON.

Définition 2.2
JavaScript Object Notation or JSON is an open-standard file format that uses human-readable text to transmit data objects consisting of attribute–value pairs and array data types (or any other serializable value). It is a very common data format used for asynchronous browser/server communication. It allows to represent structured information as XML allows for example.

Thanks to the *json* Python library, we made a human readable and editable file. This allows us to suppress the interface we made to add a new device and simplify this process. It is possible to open the file with a simple text editor such as *vim* to add or edit a device.

We later improved this setting file by adding all known data about each device : its type (*1* for camera or *0* eBug) and its LED sequence. Thus it would be possible in the future to add and save more settings in this file, and no python code modification will be needed on the existing code. The current setting file is presented on the figure 2.10.

eBugs_pairing_list.json

```
{
    "10":
    {
        "type": 1,
        "psoc_id": [15, 19, 25, 605, 25, 68]
    },
    "1":
    {
        "type": 0,
        "psoc_id": [17, 30, 13, 844, 41, 68],
        "led_sequence": ["0x1a02", "0x2091", "0xc56c"]
    },
    "3":
    {
        "type": 0,
        "psoc_id": [30, 20, 19, 618, 41, 68],
        "led_sequence": ["0x4381", "0x3c02", "0x807c"]
    },
    "5":
    {
        "type": 0,
        "psoc_id": [25, 28, 21, 618, 41, 68],
        "led_sequence": ["0x440", "0xd0a3", "0x2b1c"]
    },
    "6":
    {
        "type": 0,
        "psoc_id": [18, 20, 19, 618, 41, 68],
        "led_sequence": ["0x9061", "0x4c16", "0x2388"]
    },
    "7":
    {
        "type": 0,
        "psoc_id": [13, 33, 13, 844, 41, 68],
        "led_sequence": ["0x6483", "0x8308", "0x1874"]
    },
    "8":
    {
        "type": 0,
        "psoc_id": [21, 29, 13, 844, 41, 68],
        "led_sequence": ["0x14e1", "0x600a", "0x8b14"]
    }
}
```

Figure 2.10: JSON eBugs settings file

We modified our function *assign_static_addresses()* again in order to instantiate three public variable in the class *nrf.Bridge*:

- camera : contains all settings data about powered-on cameras.
- eBugs : contains all settings data about powered-on eBugs.
- unknown : contains the MAC-addresses of the unknown detected devices.

Lets consider a simple python script :

```
1 from libraries.nrf import Bridge
2
3 nrf = Bridge()
4 items1, items2, items3 = nrf.assign_static_addresses()
```

items1 contains Psoc_id and settings of all cameras, *items2* contains Psoc_id and settings of all eBugs and *items3* is a list of MAC-adresses.

It is also possible to have access to these data through this way :

```
5 print(nrf.eBugs[5].type)   # 0
6 print(nrf.camera[10].psoc_id)   # [15, 19, 25, 605, 25, 68]
7 print(nrf.unknow)   # ()
```

2.5 Test the network

A main aspect of our project was to test the limit of the network. Indeed, we need to know its limits in order to avoid overcharge and unwanted or uncontrolled results.

The camera is suppose to send picture to the control station each $\frac{1}{30}$ seconds, what could cause overcharging. To reduce the network charge, the camera doesn't really send a full picture, but only a collection of a maximum of 6 **blobs**. Each blob is composed of four elements. Put together, they represent information about one LED :

- **the X coordinate** : The position of the center of the LED on the X-axis of the camera.
- **the Y coordinate** : The position of the center of the LED on the Y-axis of the camera.
- **the LED's color** : 0 for red, 1 for green and 2 for blue.
- **the LED's radius** : The radius of the colored dot detected by the camera. If it is too small compared to the other, that probably mean that the colored dot is not a LED but stray light.

A *timestamp* is associated to each packet received so that it is possible to date the incoming data and then build up the frames. As a frame is composed of multiple blobs, it is possible to have more than one packet with the same timestamp.

This data structure is designed to send more information with less data. The camera usually detects 12 to 20 colored dot (LED + stray light) per eBugs and it can sends up to 6 blobs per packet. Therefore, the camera needs to send around 3 to 4 packets to the control station in order to represent the eBug's position on a user interface. The question that naturally came to our minds was : How many robots can be detected by a camera while guaranteeing a refresh rate of 30 frames per second?

At the beginning, we only had one eBug, the control station and no camera. EBugs aren't the most consuming device in term of network resource. They use the same network as the camera, but they only receive aperiodic commands from the user which takes a constant amount of packets, while the camera needs

$$number\_of\_eBugs * number\_of\_packets\_per\_eBug * refresh\_rate$$

This number can grow fast. For example, for three eBugs, and a 30 fps rate, the camera needs 270 packets per second to transmit the position of the eBugs.

Then we started testing we the material we had : one eBug and the control station. Our first tests were not very relevant. However, they gave us a pretty good idea of the network capacity because eBugs use the same chip as the camera.

Our first test was to send simple packets (we used the function *nrf.print_ top(string)*) during a certain amount of time and then calculate how much packets have been sent per second. The function we use (*nrf.print_ top(string)*) transmits an ascii string from the control station to the eBug and we noticed that the mean time between two packets depends on the size of each packet. Finally, we identified that we can send between $2 * 10^3$ and $5 * 10^3$ packets per second. That would mean that the camera could track 3 or 4 eBugs.

A few days after that test, we had access to more eBugs and a working camera. We decided to make more accurate tests. The camera network works just like the eBug's : by sending request to it and waiting for a response (see figure 2.11). The control station must request blobs from the camera before receiving data. We decided to get more statistics on the communication function between the camera and the control station : *nrf.get_ blobs()*.

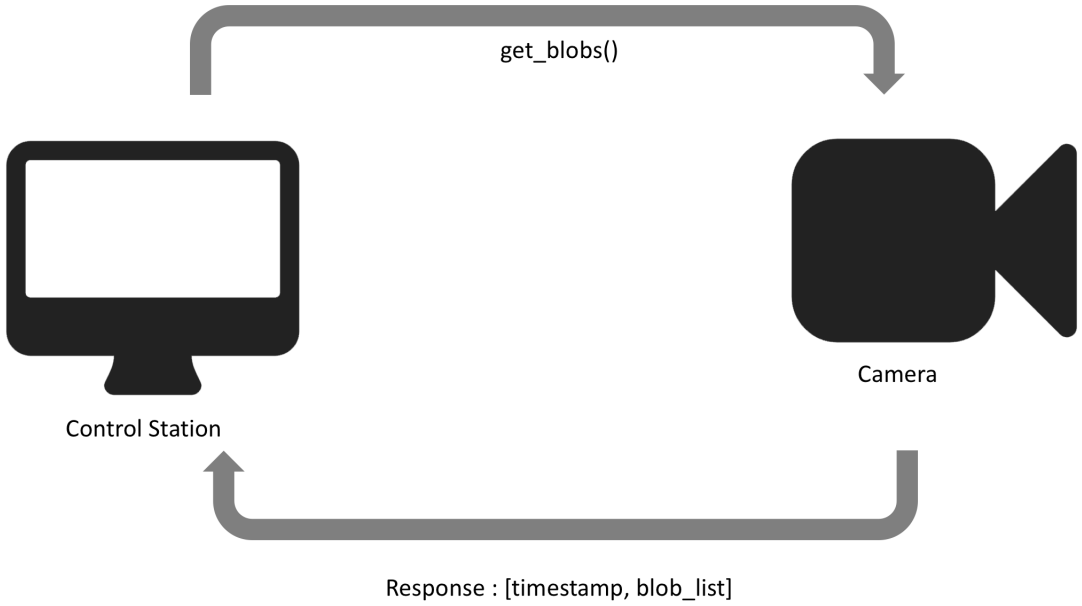

Figure 2.11: Camera communication

We decided to create a new script file named *blob_ test.py* to gather all the tests we made. This could be useful if we want to get measurement on improvement we will make in the future or just re-use piece of code or functions we made.

Before executing testing function, we write some code that is executed each time the file is imported. It initializes the control station wireless chip, assigns simple addresses to all device and then sets different camera parameters to get a 30 fps rate for example.

blob_test.py

```
19 nrf = Bridge()
20
21 while True:
22     camera, eBugs, unknown = nrf.assign_static_addresses(path='../
           libraries/eBugs_pairing_list.json')
23     if camera:
24         break
25     else:
26         print(" -------- Waiting for camera --------")
27         time.sleep(0.03)
28
29 #set communication with the first camera detected
30 #to be changed when multiple cameras
31 nrf.set_TX_address(next(iter(camera)))
32
33 #settings camera
34 nrf.camera_write_reg(0x10, 20)
35 values = [0x79, 0x9a, 0xb1, 0x5a, 0xc6, 0x70, 0xa3, 0x82]
36 nrf.set_camera_thresholds(values)
```

After the initialization, we made a first testing function in order to get information about the duration of the *nrf.get_blobs()* function and its fail rate.

blob_test.py

```
38 def stat(maxINT):
39     '''
40     @param int maxINT : the number of get_blobs() calls
41     @return : displays various stats on the function get_blobs()
42     '''
43     array = list()
44     mean = 0
45     fail_number = 0
46     for i in range(maxINT):
47         sys.stdout.write('\r%d' % (i/maxINT*100))
48         sys.stdout.flush()
49         try:
50             start = time.time()
51             nrf.get_blobs()
52             end = time.time()
53             array.append((end-start)*1000)
54         except RuntimeError:
55             fail_number += 1
56             continue
57
58     mini = array[0]
```

```
59      maxi = array[0]
60      for value in array:
61          mean += value
62          if value > maxi:
63              maxi = value
64          elif value < mini:
65              mini = value
66      mean /= len(array)
67
68      print ("\nMini : %f ms, Maxi : %f ms, Mean : %f ms, Fail number :
            %d, Fail rate : %f \n" % (mini, maxi, mean, fail_number,
            fail_number/maxINT*100))
```

After getting these data, we wanted to observe the structure of the blobs and we made a function to record the result of the *nrf.get_blobs()* function without any treatment or modification of the data. The analysis of the blob records allowed us to get relevant figures on the network capacity and its key concepts. After running these test, we had a clear idea on how where blob organized and how much blob can be carried by a packet.

We have also made some supposition on the functioning of the camera. When the control station uses the *nrf.get_blobs()* function to request some information from the camera, this one responds with a first blob list. The camera keeps in a buffer all other blobs that couldn't fit in the first packet. The control station must run *nrf.get_blobs()* again until the camera blob buffer is empty. If the camera blobs buffer is empty and the control station requests more blobs, then the camera responds with an empty list (still associated with a timestamp) or newer blobs. This speculations were confirmed by Tony Grubman who wrote the firmware of the camera.

We noticed that the control station requests blobs faster than the camera can generate it with one or two eBugs. We wanted to make a better program for requesting blobs and lighten packets exchange between the camera and the control station. The first idea we had was to move the eBug recognition directly on the camera's firmware. Then we could have developed a function like *get_eBug_position()* which could have returned *[timestamp, [(eBug_id, x-coordinate, y-coordinate), ...]]*. Unfortunately, this function is handled in the control station by **OpenCV** (Open Source Computer Vision). This is a library of functions designed for computational efficiency and specialized in the picture processing. Moving it on the camera firmware would have been very complex for us given the fact that we lack expertise in firmware development, and the time constrain imposed by the University of technology of Compiègne.

An other idea was to stop requesting the camera when it returns an empty blob list, meaning that the frame is complete, and wait for a $\frac{1}{30}$ second timer. This will permit the application to save network and CPU resources for both the camera and the control station.

We wanted to develop a module, a function that returns a full frame : all the blobs with the same timestamp. Then this function could be easily exploitable by a the camera script.

We develop a second function to request blob until it get an empty or a blob with an other timestamp.

blob_test.py

```
89  def get_single_frame(first_blob = None):
90      timestamp = set()
91      frame = []
92      last_blob = None
93
94      while True:
95          try:
96              blob = nrf.get_blobs()
97          except RuntimeError:
98              continue
99          print(blob)
100         timestamp.add(blob[0])
101         if not blob[1]:
102             break
103         if len(timestamp) > 1:
104             last_blob = blob
105             break
106
107         frame.append(blob[1])
108
109     timestamp = sorted(timestamp)[0]
110     if first_blob and first_blob[0] == timestamp:
111         frame.append(first_blob[1])
112     return (timestamp, frame), last_blob
```

We made a simple function to write in a file. we did this action a lot while testing the network and it is easier to write it once than searching the syntax on internet every time we needed it.

```
115 def write_in_file(file, frame):
116     with open(file,'a+') as saving_file:
117         saving_file.write(str(frame) + '\n')
```

The timer is a simple *sleep* function that is triggered in a separated process.

```
120 def blob_timer():
121     time.sleep(1/30)
```

To be sure that writing in a file or applying other computing analysis or treatment won't affect the duration of the main function, we gather the frame treatment in a specific

function triggered in an other separated process. It is not possible to add longer treatment without delaying the main program.

```
123 def frame_treatment(frame):
124     if frame[1]:
125         #write_in_file('test.txt',time.time())
126         write_in_file('test.txt', frame[0])
```

Finally, we wrote the main function. It first launches the *blob_timer* and requests a new frame. If a blob from a newer frame (with a different timestamp) is detected, the program keeps it in memory to concatenate it to the next frame. Then, it waits until the timer expires and it loops back.

```
129 def get_frames():
130     next_blob = None
131     while(True):
132         try:
133             timer = Process(target = blob_timer)
134             timer.start()
135             frame, next_blob = get_single_frame(next_blob)
136             Process(target = frame_treatment, args=(frame,)).start()
137             timer.join()
138             if next_blob:
139                 print('frame was full')
140         except KeyboardInterrupt:
141             break
```

With a lot of eBugs (three for example), a lot of frames didn't end with an empty blob. Therefore, the timer was useless because we weren't able to properly identify the end of a frame. The communication between the control station and the timer was too slow to get 30 fps.

We couldn't figure out the issue we had with this program and we decided to start over with a new one that would precisely record time data and write it in a log file.

blob_test.py

```
146 FRAMES = list()
147
148 def log_blobs(MAX_TIME, log_file):
149     '''
150     @params int MAX_TIME, string log_file
151     write in the @log_file file located in the folder /scripts/logs
            information about blobs
152     warninig : the logs folder must exist
153     '''
154     with open('logs/' + log_file, 'w+') as log_blobs:
155         frame = []
156         i = 0
```

```
157         #discard buffer by calling get_blobs() for a few sedconds
158         #the first blob recorded in the file can correspond to the
                middle of a frame, ie first frame may be uncomplete
159         while i < 3000:
160             try:
161                 blob = nrf.get_blobs()
162             except RuntimeError:
163                 pass
164             i += 1
```

We noticed that when we were started requesting the camera right after its initialization, we had some irrelevant time data. We made a simple loop to request blob from the camera during a fixed time interval so that we can get rid of the camera blobs buffer. You can notice that the function *nrf.get_ blobs()* is surrounded by a *try:* and *except RuntimeError:*. The function *nrf.get_ blobs()* can raise a *RuntimeError* to inform the user or the program that it encountered an error of communication. This often means that a group of blobs are lost. The camera won't send it again. The *try - except* clause allows the program to know that blobs could be missing and to react to this information.

```
164         frame_number = 1
165
166         print("Start log")
167
168         while True:
169             time_0 = time.time()
170             try:
171                 blob = nrf.get_blobs()
172             except RuntimeError:
173                 continue
174             time_1 = time.time()
175             break
176
177         frame.extend(blob[1])
178         RTT = time_1 - time_0  # depends of the size of the blob
179         camera_time_0 = camera_time = blob[0]
180
181         log_blobs.write("UNIX : %f ms, RTT : %f, Camera Timestamp : %d
                ms, Blob info : %s\n" % ((time_1 - time_0) * 1000, RTT,
               camera_time - camera_time_0, str(blob[1])) )
```

Then we instantiate variables representing the time reference of the camera and of the control station.

```
183         while time_1 - time_0 < MAX_TIME:
184             try:
185                 time_1 = time.time()
186                 try:
187                     blob = nrf.get_blobs()
188                 except RuntimeError as error:
```

```
189                     ex_type, ex, tb = sys.exc_info()
190                     traceback.print_tb(tb)
191                     log_blobs.write("RunTime Error : %s\n" % error)
192                     continue
193                 if camera_time != blob[0]:
194                     log_blobs.write("FRAME #%d : %s\n" % (frame_number
                            , str(frame)))
195                     frame_number += 1
196                     if frame:
197                         FRAMES.append(frame)
198                     frame = []
199                 if blob[1]:
200                     frame.extend(blob[1])
201                 camera_time = blob[0]
202                 time_2 = time.time()
203                 RTT = time_2 - time_1 #depends of the size of the blob
204                 log_blobs.write("UNIX : %f ms, RTT : %f, Camera
                        Timestamp : %d ms, Blob info : %s\n" % ((time_2 -
                        time_0) * 1000, RTT, camera_time - camera_time_0,
                        str(blob[1])) )
205             except KeyboardInterrupt:
206                 break
```

This program allowed us to have very precise data on blobs exchange. We figured out that some of blobs request (*nrf.get_blobs()*) were taking much more time than others, without obvious causes. That is why we investigated on the function *nrf.get_blobs()*. We notice that the function not only sends a simple packet and then waits for a response, but keeps trying sending the packet 10 times if it fails and waits 0.003 seconds between each attempt. This could be the reason why *nrf.get_blobs()* can sometimes take much more time. Moreover there is packet collision when the camera responds to this function and the control station doesn't get any response, it will keep asking the camera and silently lose blobs.

The previous function used *nrf.send_packet_check_response()* was not adapted when we communicate with the camera, so we made a similar function that tries to send a packet and checks if the response is corresponding, and it raises an error immediately if it fails.

nrf.py

```
79      def send_packet_check_response_without_retry(self, packet):
80          x = self.send_packet(packet)
81          if x is None:
82              raise RuntimeError('Empty Response')
83          if x[0] != packet[0]:
84              raise RuntimeError('Unexpected response: %s'%repr(list(
                    bytearray(x))))
85          return x[1:]
```

That new function give us much better result and we had no difficulty to get 30 fps from the camera.

All function that we defined in the file are reusable from other python script or python interpreter :

```
remid-MPB:nrf-bridge remid$ ipython3

Python 3.6.1 (default, Apr  4 2017, 09:40:21)
Type "copyright", "credits" or "license" for more information.

IPython 5.3.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: import scripts.blob_test as bt

Addr    PSoC ID                 Type        LED Sequence'
----    --------------------    ----------- ------------------------------
10      15-19-25-605-25-68      camera (0)
6       18-20-19-618-41-68      eBugs (1)   ["0x9061", "0x4c16", "0x2388"]

In [2]: bt.log_blobs(10, "1eBugs_10seconds")
```

While analyzing out blobs and our full frames, we had trouble with OpenCV. That's why we decided to make a very simple graphic interface to represent blob with the python library *tkinter*. The library only needs python and is easy to use and portable.

blob_test.py

```
209 master = tkinter.Tk()
210 w = tkinter.Canvas(master, width = 750, height = 750)
211
212 def _create_circle(self, x, y, r, **kwargs):
213     return self.create_oval(x-r, y-r, x+r, y+r, **kwargs)
214 tkinter.Canvas.create_circle = _create_circle
215
216 def print_frame(event):
217     frame = FRAMES.pop()
218     for dot in frame:
219         color = "red" if dot[2] == 0 else "green" if dot[2] == 1 else
              "blue"
```

```
220             w.create_circle(dot[0]*750/1000, dot[1]*750/1000, dot[3], fill
                    = color)
221
222     def display():
223             w.pack()
224             w.bind('<Button-1>', print_frame)
225             w.config(bg = 'white', borderwidth = 1)
226
227             message = tkinter.Label(master, text = "Click on your left
                    mouse button to see the next frame !")
228             message.pack(side=tkinter.BOTTOM)
229
230             tkinter.mainloop()
```

The function user should use is *display*. It can be used from any python interpreter as well but the *log_ blobs()* function must be run first.

```
remid-MPB:nrf-bridge remid$ ipython3
Python 3.6.1 (default, Apr  4 2017, 09:40:21)
Type "copyright", "credits" or "license" for more information.

IPython 5.3.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: import scripts.blob_test as bt

Addr    PSoC ID                 Type        LED Sequence'
----    --------------------    ----------- ------------------------------
10      15-19-25-605-25-68      camera (0)
6       18-20-19-618-41-68      eBugs (1)   ["0x9061", "0x4c16", "0x2388"]

In [2]: bt.log_blobs(10, "1eBugs_10seconds")

In [3]: bt.display()
```

Chapter 3

Conclusion

This project was a serious challenge for us, both in its technical and human aspects. Indeed, all our exchange with our supervising professors were done in English which is not our mother tongue. In an other hand, we discovered a lot of tools that will surely be useful in our future work of engineer. The quality of the supervision, and the good working conditions helped us a lot in our investment in this project. Even though we couldn't open the window while the heatwave was striking us, being able to work on a double screen computer in a dedicated office was very comfortable.

Furthermore, we learned a lot by conducting this project. Although we already new the Python coding language, we also discovered new possibilities such as the manipulation of serial connected devices. Being able to work in autonomy obliged us to be even more rigorous in our code, and resourceful as we had to discover and understand an already started project. We also had the opportunity to discover new software such as *OpenCV* which seems to be very powerful and could possibly offer a lot of possible improvements in the future. We regret not being able to use and learn more about that software.

This course allowed us to implement different concepts saw during other courses such as real time programming linked with embedded systems constraints with MI11 or MI12, or the graph theory with RO03. Finally, we hope that our job during this project will help the future students that will continue working on the eBugs.

Chapter 4

Annex

This table contains all the packets command we were able to find by investigating the **nrf2401** firmware with **PSoC Creator**.

multicast packet	0xb0	Neighbour discovery request only if no unicast address set
	0xb1	Neighbour discovery request
	0xb2	Set unicast address
	0xb3	Forget unicast address
unicast packet	0x00	Dummy packet to get pending ACKs
	0x01	Serial number and device type
	0x02	Firmware version
	0x03	JTAG ID
	0x10	print to LCD top line
	0x11	print to LCD Bottom line
	0x12	LCD backlight control
	0x20	Set motor speed and direction
	0x21	Adjust boost converter voltage
	0x22	Get motor feedback and reset counters
	0x23	Get motor feedback and don't reset counters
	0x24	Calibrate hall effect sensor
	0x25	Configure motor controller
	0x26	Set motor controller target
	0x27	Get motor speed
	0x30	Enable side top and center LEDs
	0x31	Set LEDs values
	0x32	Set LEDs brightness
	0x40	Get bump sensor state
	0x41	Get touch button state
	0x50	Get light sensor values
	0x51	Control light sensor sensitivity
	0x60	Battery voltage / current
	0x61	Calibrate delta-sigma ADC (for battery measurment)
	0x62	Increase charge current
	0xb3	Forget unicast address
	0xff	Reset to bootloader
	0xfe	Reset
Camera packet	0x91	Write reg
	0x93	Set Camera Tresholds
Motor packet	0x80	Get laser event
	0x81	Laser motor enable
	0x82	Set laser id
	0x83	Set laser motor speed
	0x84	Test laser
	0x90	Get laser motor feedback

Figure 4.1: Packets headers and commands