7. Testing

After implementing the solution developed in the various domains into the target platforms, the system's behaviour is tested in several levels of granularity: at the subsystem level — unit testing, and system level — integrated testing. The idea is progressively test the behaviour of each subsystem and its integration into the system. In this chapter are presented the unit testing and integrated testing for the RFCAR.

7.1. Unit testing

The unit tests are briefly described in this section.

7.1.1. Navigation Virtual Subsystem

The NVS susbsystem tests are presented next. The stack implementation tests were mainly focused on the interactions established between the packages at play and the surrounding systems both lower-tiered and high-tiered.

7.1.1.1. IO: Input/Output Package

The IO package testing (figure 7.1) consisted in the creation of a GPIO object with which one could simulate motor PWM outputs that would generate a file, through which one would simulate the IR sensor readings or motor pulse readings. The first GPIO test relied on a configuration of an object in OUTPUT_PWM mode. With the latter test, it was possible to write three 32 bit integers to a binary file and observe the results with notepad++. The second test hinged on reading from the forementioned binary file, expecting to retrieve the corresponding values. Should the latter be as expected the test could be considered a success.

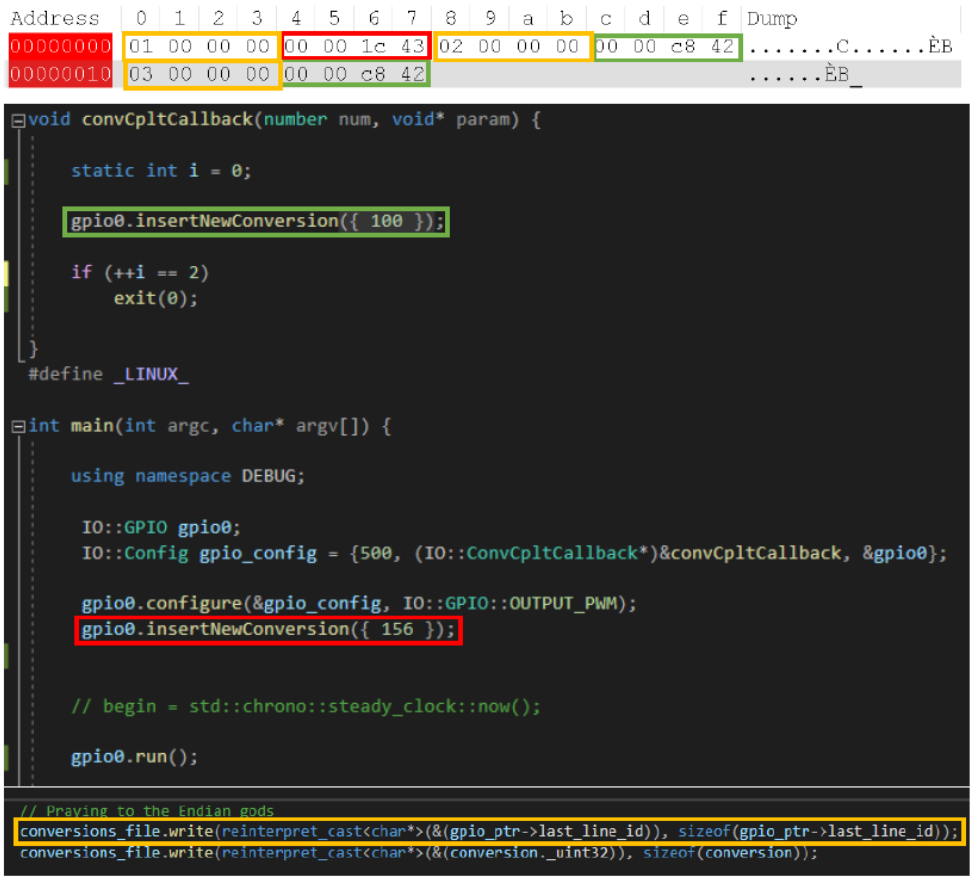

Figure 7.1.: IO package OUTPUT_PMW mode tests

7.1.1.2. COM: Communications Package

The COM package testing involved effectuating some experiments between the NVS and the smartphone (using Bluetooth) and the former and the RVVS (using RS232). These experiments will be performed in section 7.1.1.5.

7.1.1.3. OS: Scheduler Package

The OS package testing was based on the creation of two types of Threads:

- **Producer Thread (P)**
- **Consumer Thread (C)**

The OS package testing was based on the creation of two types of Threads. These tests were made to simulate the way how the operating system handles multi-thread contexts. The Producer Thread type

publishes values in a list while the Consumer Thread type removes values from the latter. The introduction of an initial delay in the Producer Thread function allows the OS to assign the CPU more evenly. The test is represented in figure 7.2.

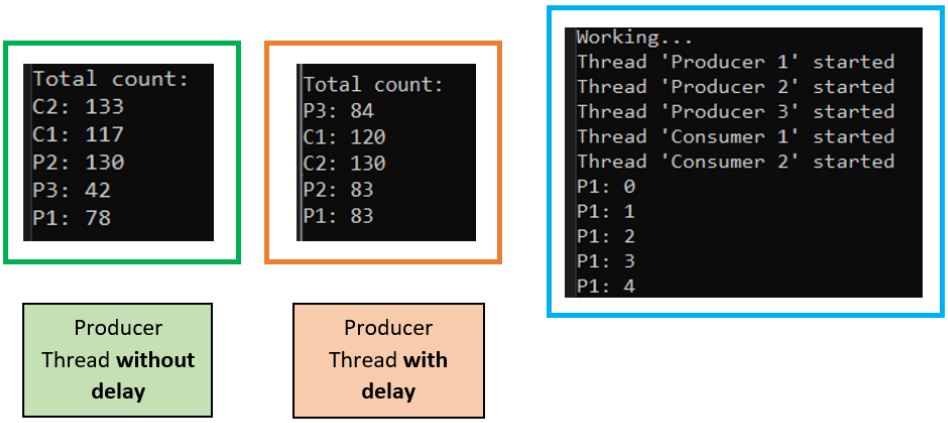

Figure 7.2.: OS package tests

7.1.1.4. MEM: Memory Structures Package

The MEM Package testing consisted of creating two projects, one that used linked lists and another that used a circular buffer. In the former, one simply pushed numerical characters and verified that the appropriate memory spaces were filled. Afterwards, those characters were popped and the result was as expected, the memory spaces were removed from the list. This test is depicted in figure **??**.

```
char_list.push(buff2[0], Position::BACK);
char_list.push(buff2[1], Position::BACK);
char_list.push(buff2[2], Position::FRONT);
char_list.push(buff2[3], Position::FRONT);

char_list.pop(buff2[2]);
char_list.pop(buff2[3]);
char_list.pop(buff2[0]);
```

Figure 7.3.: MEM package linked list tests excerpt

7.1.1.5. CLK: Timing Package

The CLK package testing (figure 7.4) included the creation of a Timer object, associating it to a callback and proceed to verify if the callback was called at the specified time preemptively defined in the Config object configuration.

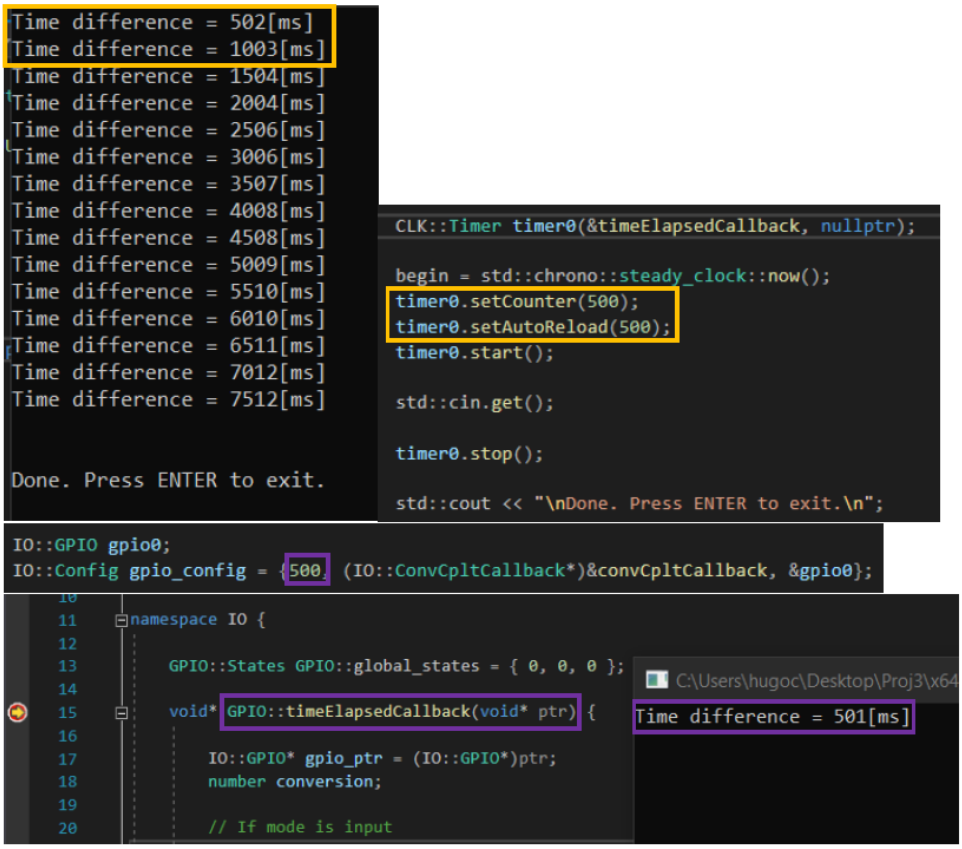

Figure 7.4.: CLK package tests

7.1.1.6. System response

The implementation of the control module was made using the modified speed algorithm mentioned and explained in 5.1.1.3.

The tests for this module consists in the comparison between the results obtained and simulated in 5.1.1.6. Starting with speed reference = 1 m/s and $\theta = 0$ rad:

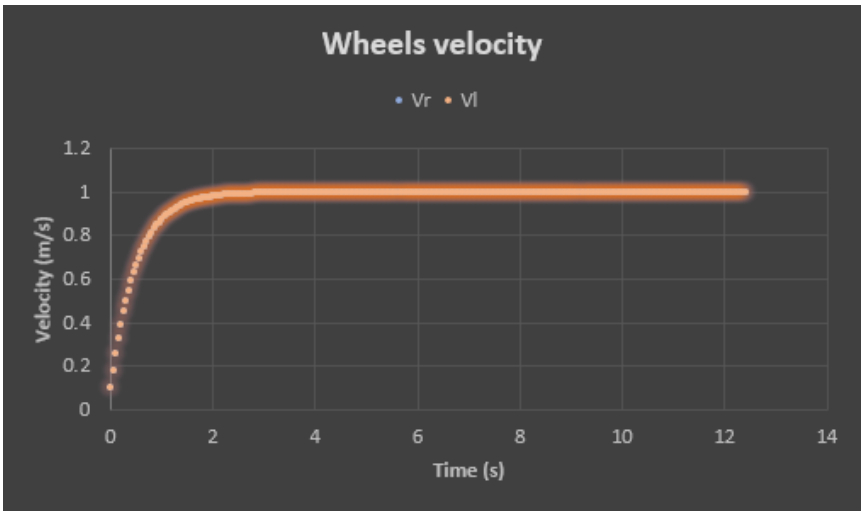

Figure 7.5.: Wheels velocity v=1m/s,$\theta = 0$ rad

In the figure 7.5 shows that the velocity of both wheels reaches the reference value in nearly 1.5 seconds, while in the simulation 5.8 it takes about 3 seconds. This discrepancy is due to the controller in use. The simulations were made using the PID block provided by Simulink, while in the implementation, the controller algorithm used was as mentioned before, the modified speed algorithm. With this algorithm the behavior of the car is the same as with the traditional PID, but faster.

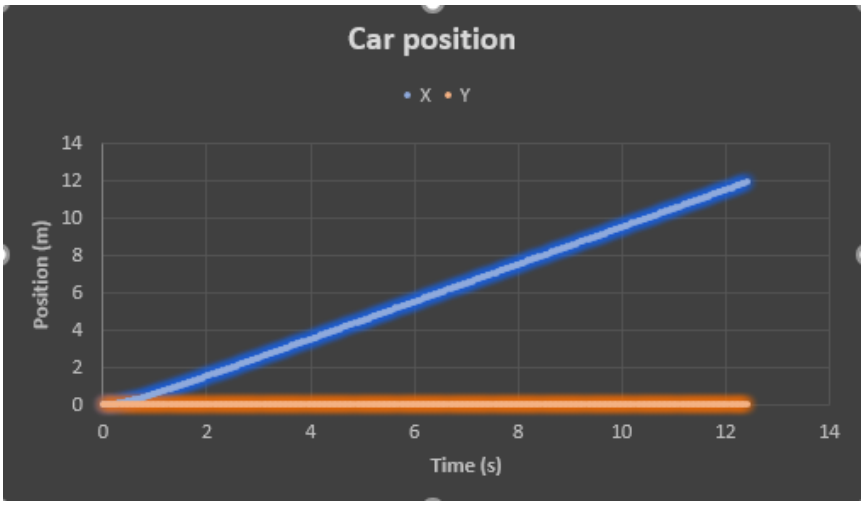

Figure 7.6.: Position v=1m/s,$\theta = 0$ rad

In the figure 7.6 it can be observed that the position of the car is the same as in the simulation 5.9. With speed reference = 1m/s and $\theta = 0.1$ rad:

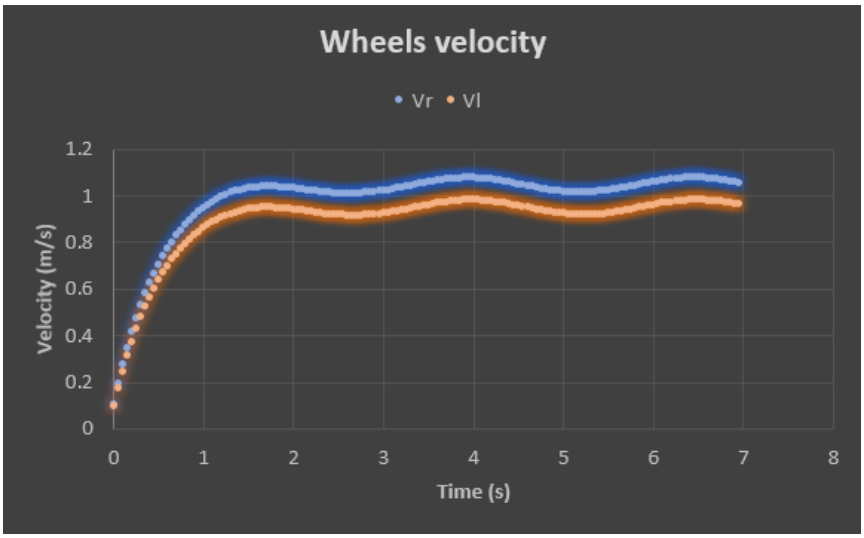

Figure 7.7.: Wheels velocity v=1m/s,$\theta = 0.1$ rad

In the figure 7.7 comparing to 5.10 it's possible to see that once again the time it takes to reach steady state is smaller for the reason mention before. In the implemented algorithm it's also possible to see that the steady value of the velocity of both wheels has a ripple around the reference value.

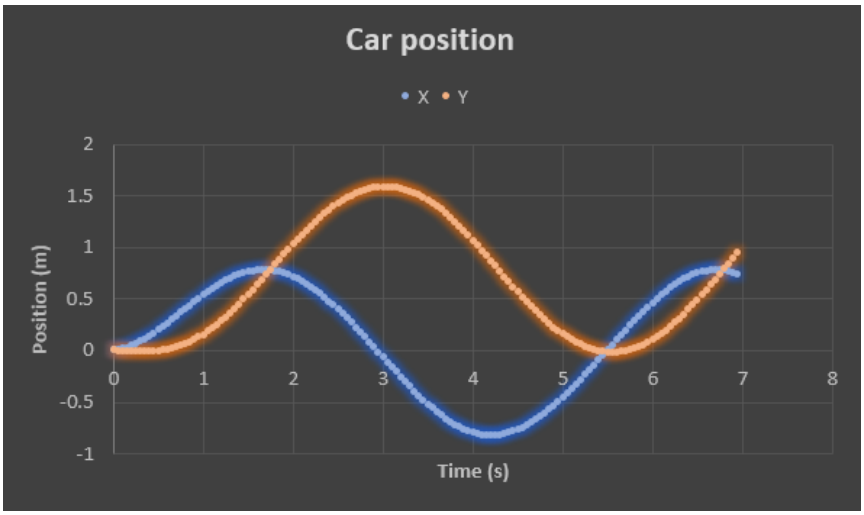

Figure 7.8.: Position v=1m/s,$\theta = 0.1$ rad

In the figure 7.8 it is present the position of the car. In comparison to the simulated 5.11 it is possible to see that the results are the same, thus validating the implementation.

7.1.1.7. Obstacle Avoidance Through Odometric Sensors

The car composed by 9 odometric sensors radially distributed, 40 degrees apart.
In order to test if the car can avoid obstacles, the values for the sensors were generated, and the behavior of the car was as follows:

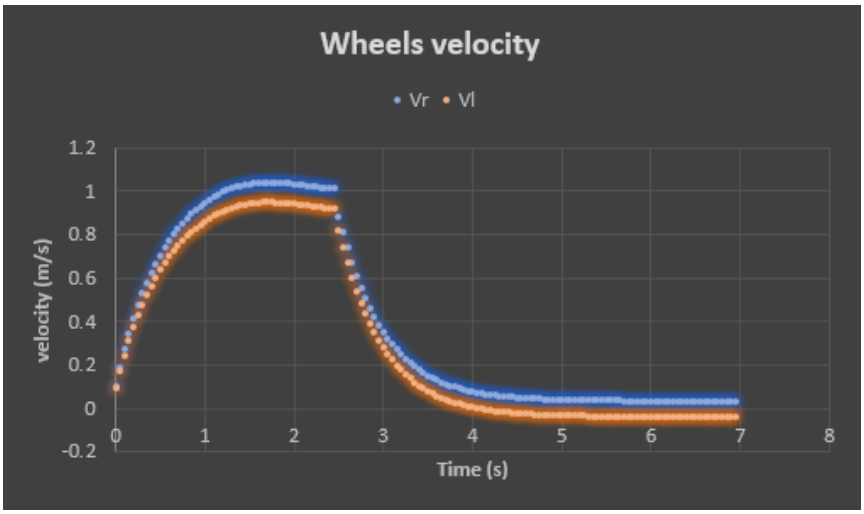

Figure 7.9.: Wheels velocity

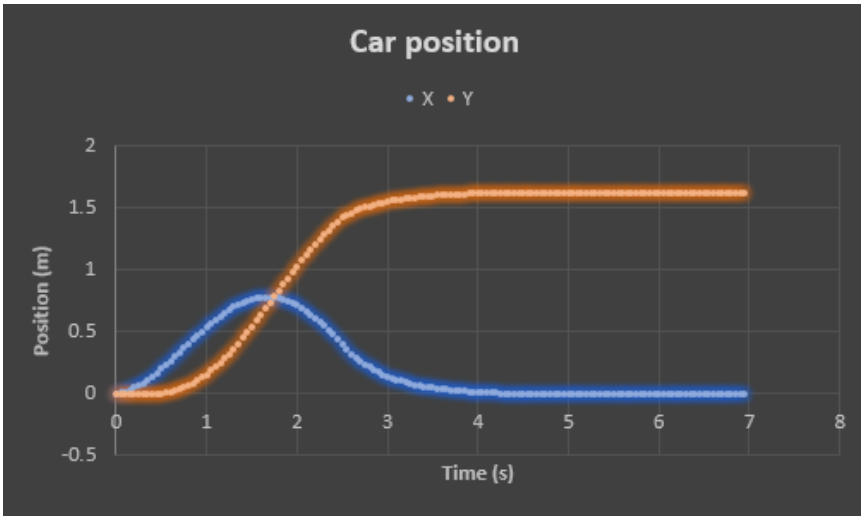

Figure 7.10.: Car position

In the figure 7.9 it is possible to see that at t=2.5 seconds the velocity of both wheels starts decreasing until they reach a value near 0 m/s. Even though the velocity of both wheels is not exactly 0 m/s, the values

are too small to break the inertia of the wheels, so the car effectively stops. In the figure 7.10 it is possible to see that the position of the car remains the same after the decrease of the velocity of the wheels, proving that the car is not moving.

7.1.2. Remote Vision Virtual Subsystem

In this section are presented the tests performed on the RVVS subsystem in the relevant domains.

7.1.2.1. Image Acquisition

The image acquisition system, implemented in Section 6.2.2.1, was tested out to assess its behaviour.

Firstly, it was selected the built-in webcam from host and attached to the guest VM. However, and despite extensive troubleshooting, the bypass suggested for web cameras[17] was not possible for a Mac OS host. Thus, a quick alternative was to flash a Linux OS onto a Storage Disk (SD) card and plug it to a Raspberry Pi 3 (available) and connect an external Universal Serial Bus (USB) camera to it (Fig. 7.11).

Figure 7.11.: Raspberry Pi 3 + Webcam testing setup

The deployment was performed using the Secure Shell (SSH) protocol to connect to the Raspberry Pi and execute the necessary commands on it, and using the Secure Copy (SCP) command that uses the Secure File Transfer Protocol (SFTP) protocol to transfer files between host and guest (Listing 7.1).

```
ssh pi@192.168.1.10 -v
scp -r webcam pi@192.168.1.10/Documents/
```

Listing 7.1: Deployment commands targetting Raspberry Pi

The web camera was then tested using the **lsusb** command and piped the output to a **.txt** file for further examination. Listing 7.2 contains an excerpt of this output where it can be observed the webcam model (**Cubeternet WebCam**) and the USB2.0 interface.

```
Bus 001 Device 007: ID 1e4e:0100 Cubeternet WebCam
Device Descriptor:
  bLength                18
  bDescriptorType         1
  bcdUSB               2.00
  bDeviceClass          239 Miscellaneous Device
  bDeviceSubClass         2
  bDeviceProtocol         1 Interface Association
  bMaxPacketSize0        64
  idVendor           0x1e4e Cubeternet
  idProduct          0x0100 WebCam
  bcdDevice            0.02
  iManufacturer           1 Etron Technologies
  iProduct                2 USB2.0 Camera
  iSerial                 0
  bNumConfigurations      1
```

Listing 7.2: Webcam information obtained via **lsusb** (excerpt)

However, this information by itself is not so useful. Additionally, and much more importantly, one wants to check the capabilities of the device and the supported formats. For that purpose it was used the **v4l2-ctl** utility (Listing 7.3). It can be observed that the only format supported is YUYV, which is a colour space based on one luminance component (Y') and two chrominance components, called U(blue) projection and V(red projection), respectively. The framerate is also fixed (30 fps), but with different resolutions. This is very limitative, as the newer webcams support MJPEG formats, easing video capture.

```
# determing devices connected and its nodes
raspberrypi:~$ v4l2-ctl --list-devices
bcm2835-codec-decode (platform:bcm2835-codec):
  /dev/video10
  /dev/video11
  /dev/video12

USB2.0 Camera: USB2.0 Camera (usb-3f980000.usb-1.4):
  /dev/video0
  /dev/video1

# determing device parameters for /dev/video0
```

```
pi@raspberrypi:~$ sudo v4l2-ctl -d /dev/video0 --get-parm
Streaming Parameters Video Capture:
  Capabilities     : timeperframe
  Frames per second: 30.000 (30/1)
  Read buffers     : 0

# determing image formats using v4l2-ctl
pi@raspberrypi:~$ v4l2-ctl --list-formats-ext
ioctl: VIDIOC_ENUM_FMT
  Type: Video Capture
  [0]: 'YUYV' (YUYV 4:2:2)
    Size: Discrete 640x480
      Interval: Discrete 0.033s (30.000 fps)
    Size: Discrete 352x288
      Interval: Discrete 0.033s (30.000 fps)
    Size: Discrete 320x240
      Interval: Discrete 0.033s (30.000 fps)
    Size: Discrete 176x144
      Interval: Discrete 0.033s (30.000 fps)
    Size: Discrete 160x120
      Interval: Discrete 0.033s (30.000 fps)
```

Listing 7.3: Webcam supported image formats, resolutions, and framerates

Effectively, it was executed the driver program for the webcam interface (Listing 6.20), but it reported the unsupported format as expected (Listing 7.4).

```
pi@raspberrypi:~/Documents/webcam$ ./webcam-main
Unable to set image format: Input/output error
```

Listing 7.4: Webcam driver program error: unsupported image format

Then, it was tried out the **UYVY422** format by modifying the following lines into Listing 6.20 (Listing 7.5)

```
/* Set format to 320x240 and Motion JPEG */
    wbc.setFormat(320, 240, V4L2_PIX_FMT_UYVY);
/* Start stream and write to file */
    wbc.startStream("output.yuvy");
```

Listing 7.5: Webcam driver program error: modifications to support **UYVY422** format

Although an image in the **UYVY422** format was obtained, **ffmpeg** requires tedious conversions between the packed format (**UYVY422**) to planar one (**JPG**). Thus, an online tool was used to convert this image [18] into **jpg** format. In Fig. 7.12 can be seen that an image was successfully acquired.

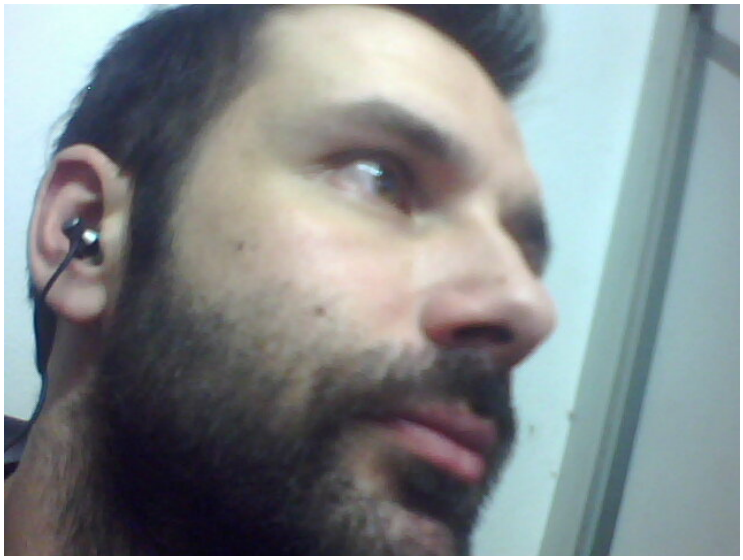

Figure 7.12.: Raspberry Pi 3 + Webcam test: success

However, this is a tedious process that could be avoided by using a different, newer, web camera supporting additional formats and resolutions, which limited severely the implementation and testing.

7.1.2.2. Wi-Fi

The Wi-Fi communications workflow, implemented in Section 6.2.2.2, was tested out to assess its behaviour, based on the client/server architecture in a concurrent execution scenario.

For this purpose, a small driver program was used (7.6), encapsulating each role on the network — client and server — in the respective thread and then execute them. The server thread creates a socket, binds to a port and address and listens for incomming connections. If this happens, it tries to accept the connection from a client, and tries to read any incoming data, prints it out and then exits. The client thread, on the other hand, tries to connect to the **localhost**, on IP address **127.0.0.1**, and if sucessful, sends a message to the server.

```
1 #include <iostream>
  #include <string>
  #include <list>
  #include <utility>
  #include <cstdio>
```

```
#include <map>

#include "main.hpp"
#include "Thread.hpp"
#include "WCOM_LL.hpp"

#define PORT_TEST 4545
#define BUFFER_SZ ((uint32_t) 64)

void tcp_socketServerTest(OS::Thread*) {
    WCOM::Error error;
    std::string error_str;
    WCOM::LL<WCOM::Protocol::TCP, WCOM::Role::SERVER> server(PORT_TEST);
    char buffer[30];

    // Serves to demonstrate that a pointer to the generic class can be used to access
        methods methods from the specialized classes
    WCOM::LL<>* server_ptr = &server;
    WCOM::LL<WCOM::Protocol::TCP, WCOM::Role::SERVER>* server_ptr2 = \
        static_cast<WCOM::LL<WCOM::Protocol::TCP,
                             WCOM::Role::SERVER>*>(server_ptr);

    std::cout << "FUN[TCP socketServerTest]: STARTED\n";

    // Create socket, bind and listen for connection
    error = server_ptr2->listenConnection();
    server_ptr->getLastError(error_str);
    std::cout << "SERVER   " << error_str;
    if (error) return;

    // Accept connection
    error = server.acceptConnection();
    server_ptr->getLastError(error_str);
    std::cout << "SERVER   " << error_str;

    // If there was an error during acceptance, return immediately.
    // The rest has been taken care of by the socket.
    if (error) return;

    while(1) {

        char temp_buffer[BUFFER_SZ];
```

```
        // Read string into temporary buffer to demonstrate how a message could be read
            continuously
        server_ptr->readStr(temp_buffer, BUFFER_SZ - 1);
        error = server_ptr->getLastError(error_str);
        std::cout << "SERVER    " << error_str;

        if (error)
            break;

        // Concatenate temporary string with permanent one
        strcat(buffer, (const char*)temp_buffer);

        // Check for reception of the message termination character
        if (strchr(buffer, '\n') != NULL) {
            // When finished, ptint the message and close the connection
            std::cout << "FUN[TCP socketServerTest]: Message received: "
                      << buffer;
            break;
        }
    }
    server.closeConnection();
    return;
}


void tcp_socketClientTest(OS::Thread*) {
    WCOM::Error error;
    std::string error_str;
    WCOM::LL<WCOM::Protocol::TCP, WCOM::Role::CLIENT> client(PORT_TEST);
    WCOM::LL<>* client_ptr = &client;

    char buffer[BUFFER_SZ] = "This is a message\n";

    std::cout << "FUN[TCP socketClientTest]: STARTED\n";

    std::string serv_addr = "127.0.0.1"; /**< localhost */

    // Open connection
    client.Connect(serv_addr, PORT_TEST);
    error = client_ptr->getLastError(error_str);
    std::cout << "CLIENT    " << error_str;

    if (error) return;
```

```
91      // Send message
        client_ptr->writeStr(buffer, sizeof(buffer));
        error = client_ptr->getLastError(error_str);
        std::cout << "CLIENT    " << error_str;
    }

    int main(int argc, char* argv[]) {

        using namespace std::chrono_literals;
        OS::Thread server_thread("Socket server test", tcp_socketServerTest);
101     OS::Thread client_thread("Socket client test", tcp_socketClientTest);

        std::cout << "\n---------------------------\n";
        std::cout << "Working...\n\n";

106     // Start server
        server_thread.run();
        // Wait for the server to configure. Anothr option would be to make
        // the client try to establish a connection indefinitely.
        std::this_thread::sleep_for(1s);
111     client_thread.run();

        // Synchronize the tasks
        server_thread.join();
        client_thread.join();
116     std::cout << "Joined!" << std::endl;

        return 0;
    }
```

Listing 7.6: Wi-Fi client/server driver program

Listing 7.7 presents the previous program's output, running on the Raspberry Pi. It can be observed that client and server exchange a message, thus, validating the implementation of the client/server model for the Wi-Fi communication.

```
1  pi@raspberrypi:~/Documents/wifi/sockets$ ./wcom

   ---------------------------
   Working...
```

```
FUN[TCP socketServerTest]: STARTED
SERVER   OK: Listening for connection
FUN[TCP socketClientTest]: STARTED
CLIENT   OK: Open connection
SERVER   OK: Acepted connection
SERVER   OK: Read string
SERVER   OK: Read string
SERVER   OK: Read string
SERVER   OK: Read string
FUN[TCP socketServerTest]: Message received: This is a message
Joined!
pi@raspberrypi:~/Documents/wifi/sockets$
```

Listing 7.7: Wi-Fi client/server driver program

7.1.3. Smartphone

7.1.3.1. Sensor Interaction

As referred in section 6.3.1.2, a ball movement application was made to test the accuracy of the linear acceleration values attained from the phone's accelerometer. One can observe the results in figures 7.13 and 7.14. In the **first case** (figure 7.13), the phone is raised on the left side but slightly downwards so the ball moves to the bottom right corner, as expected. On the **second case** presented (figure 7.14), the phone is also raised on the left side but this time slightly upwards making the ball to move from the initial position to somewhere near the top right corner. The linear acceleration values can be later used for generating the control commands of the rover since the **values were proven to be accurate**. As a way to test the code of the interaction with the rotation sensor, the values calculated were displayed on screen to perform a quick analysis, depicted in figure 7.15. One must notice the method used based in control percentage values since it allows the use of different rover components maintaining the way these values are calculated.

7.1.3.2. Bluetooth

After effectively changing the direction of a ball according to the accelerometer's state in the section 7.1.3.1, one could only need to merge its code with the Bluetooth connection setup, so that, instead of sending plain text messages, one can transfer the acceleration values.

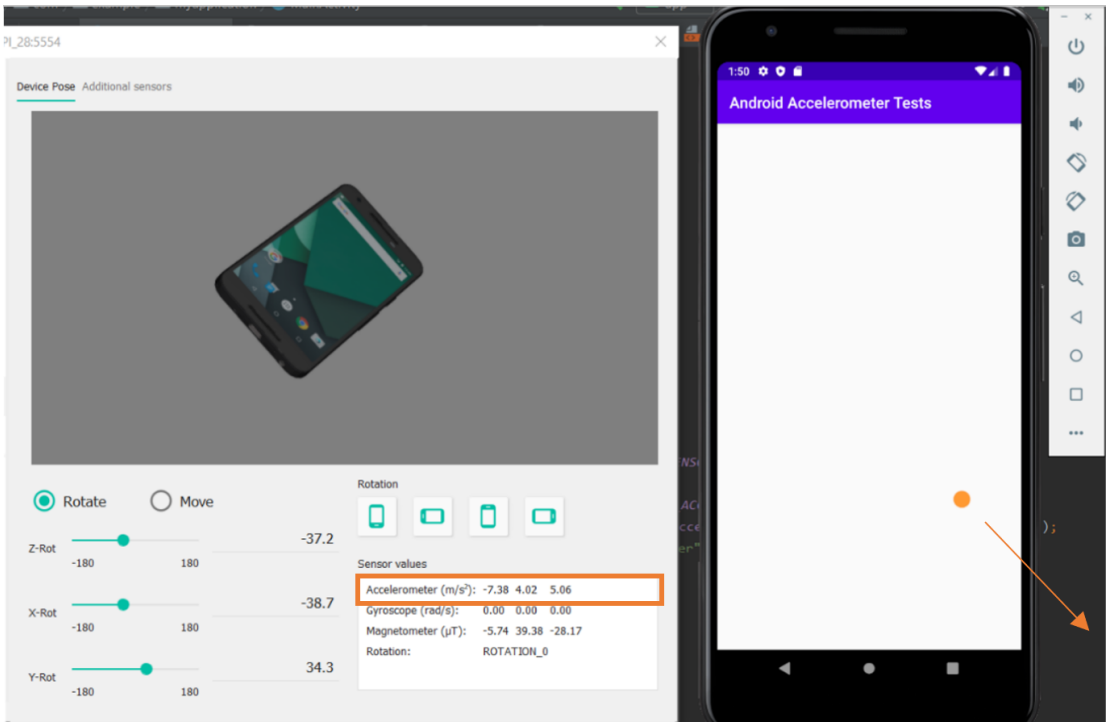

Figure 7.13.: Accelerometer ball movement tests - case 1

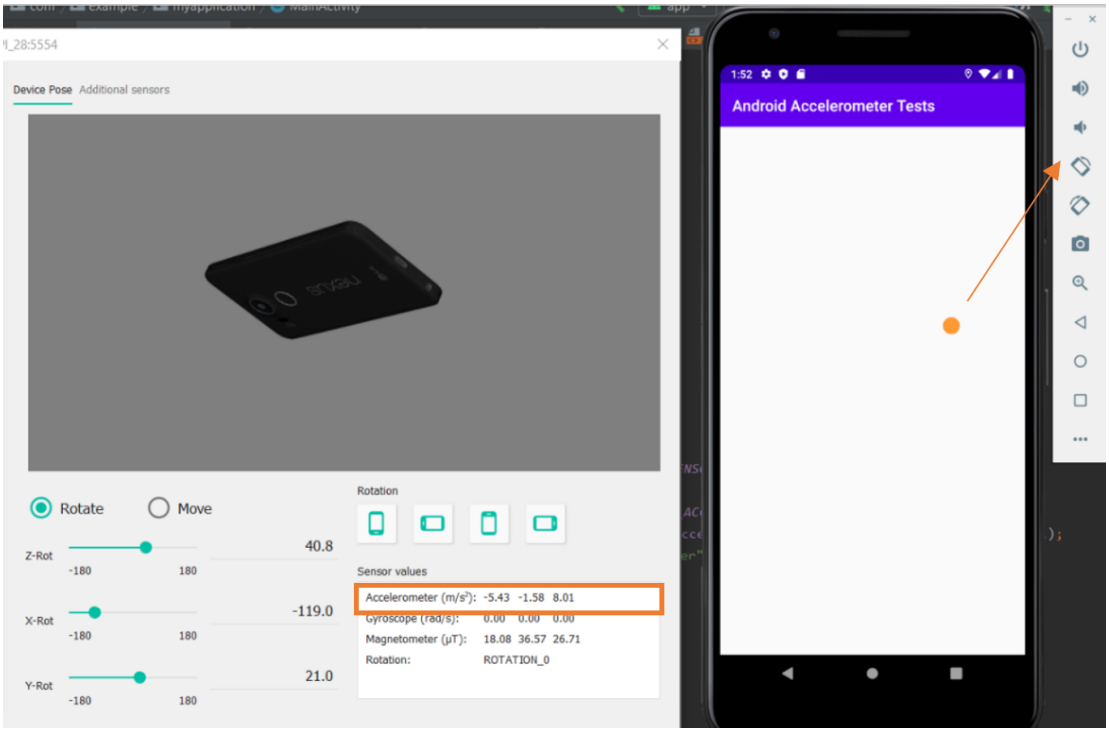

Figure 7.14.: Accelerometer ball movement tests - case 2

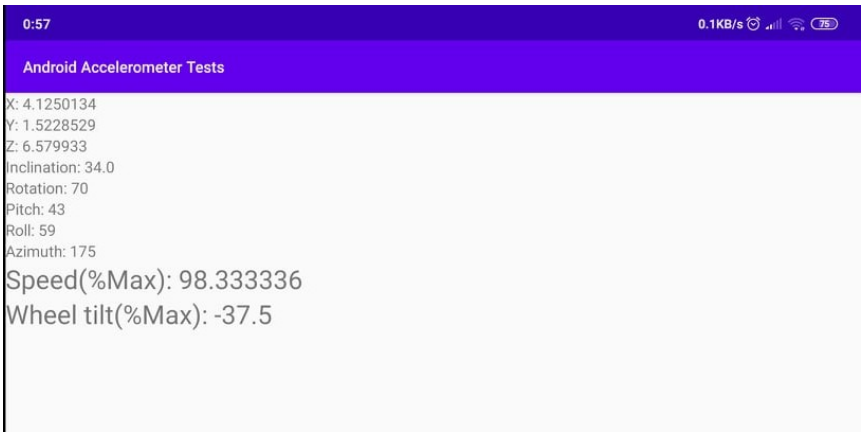

Figure 7.15.: Rotation sensor tests

7.1.3.3. Bluetooth: Smartphone-Smartphone

The first test was based on testing the Bluetooth connection setup when the interface was comprised of two distinct smartphones. With that in mind, the connection was successful and the devices were able to exchange messages, as expected. Since this test was performed using the code done in another course unit no further in-depth explanations will be presented in this section as the code was fully functional.

7.1.3.4. Wi-Fi

After the implementation of the Wi-Fi module forementioned in 6.3.3.1, one needed to perform tests in the connection between two smartphones and also between the smartphone and the RVVS. These tests will be discussed in the following sections.

7.1.3.5. Wi-Fi: Smartphone-Smartphone

For the tests related to the smartphone's wifi module, one tested the connection setup (figure 7.16) and the capability of message exchange (figure 7.17). Firstly, Wi-Fi must be enabled using the enable Wi-Fi button, accepting the request to turn it on. Next, one should press the Wifi Discover button, advancing to the subsequent screen, where a list of the available devices for connection is displayed. The green text on top of the screen refers to the current status of the application in terms of the wifi setup. When the user presses a list from the list, a request pop-up for Wi-Fi connection is presented. Accepting the request, the status changes to connected and the host and client are identified. Lastly, one tested sending and receiving messages from a smartphone to another smartphone by simply trying to press a WIFI MSG button to send a specific message, but due to deadline proximity, this feature wasn't functional and the Wi-Fi

smartphone-RVVS tests were abandoned.

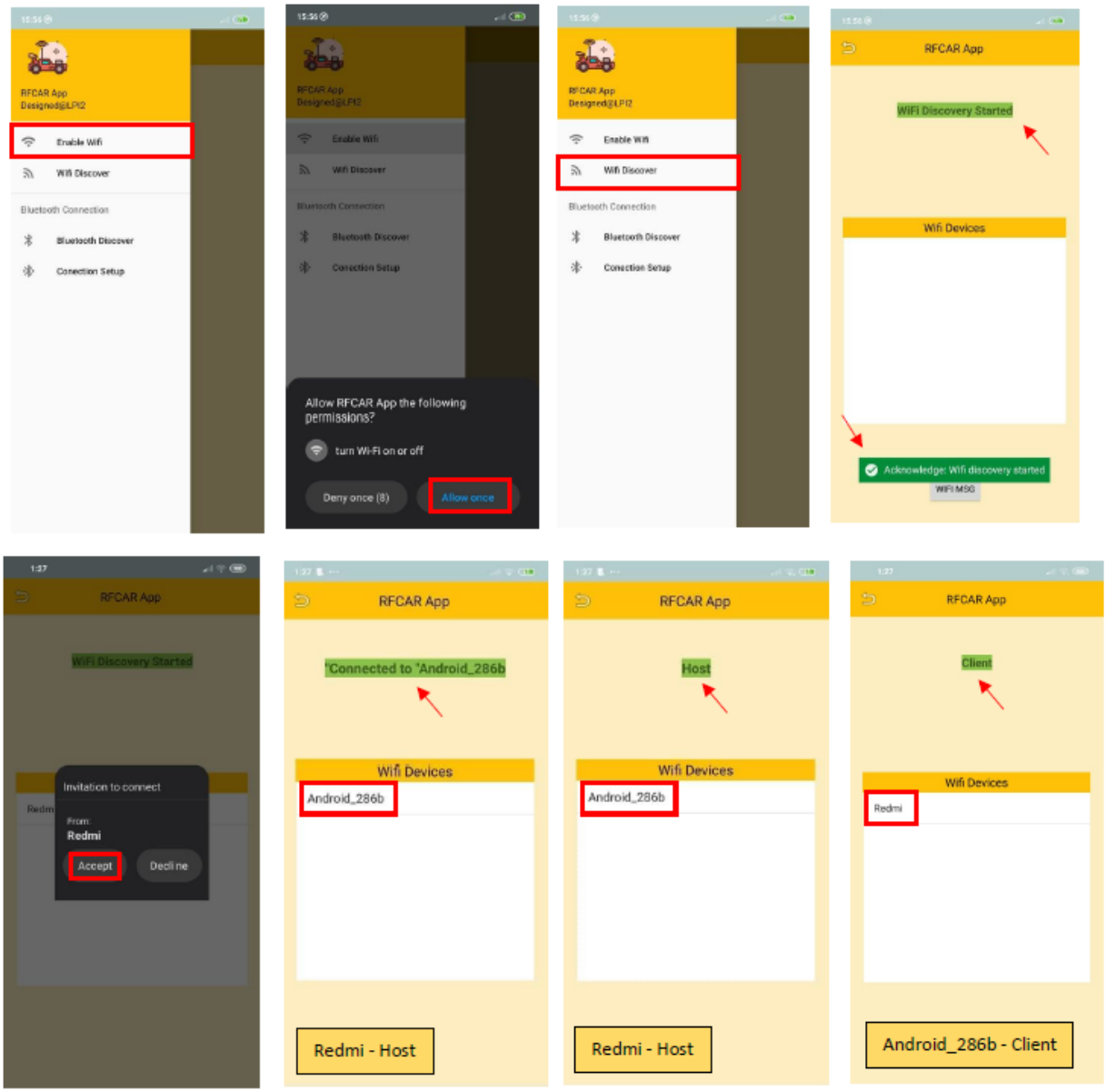

Figure 7.16.: Wi-Fi connection setup tests

7.1.3.6. User Interface

In the beginning, an app UI was initially thought out and made a first test design. When the user opens the program, the image of the RFCAR appears along with a built-in set of features. It is then explained the vehicle purpose as a navigation tool through inhospitable environments. Following that, on the top left

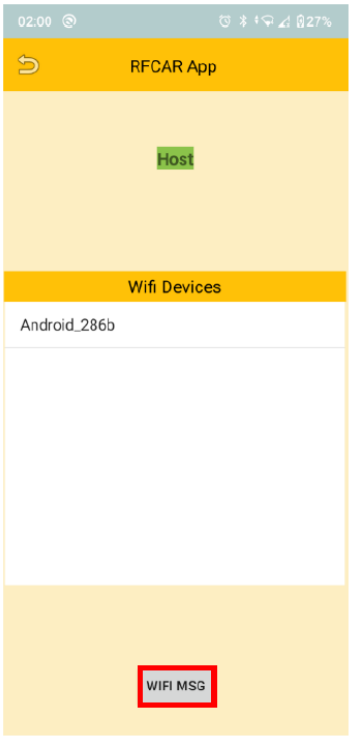

Figure 7.17.: Wi-Fi message exchange tests

corner, there's a button that when pressed it redirects to the main app activity allowing first-person vision on the controlled car just like a simulated car game, only this time in a real-life scenario. Some statistics, such as the transport position and its velocity, appear in front of full-screen video capture. From that point on, one can also choose to see the map and possibly the route traced along with some more detailed statistics of the remote control car. This first UI idea is depicted in figure 7.18. Later, some UI tests were made in terms of app functionality. One had to make sure the application behaved properly without crashing when, for example, the user intends to rotate the screen. Some activities' (app screens) orientation was fixed to ensure correct behaviour. The ones where that wasn't an issue, a version for landscape was created and tested (figure 7.19).

7.2. Integrated testing

Having tested every subsystem, comes the time to test the whole system as one.
An initial test was made with the **HC-05 Bluetooth module** inserted on the STM board. That approach only allows system engineers who have access to that module to fully test both Bluetooth client and server. However, to prove this new compound functionality, the device to be paired should also have Bluetooth

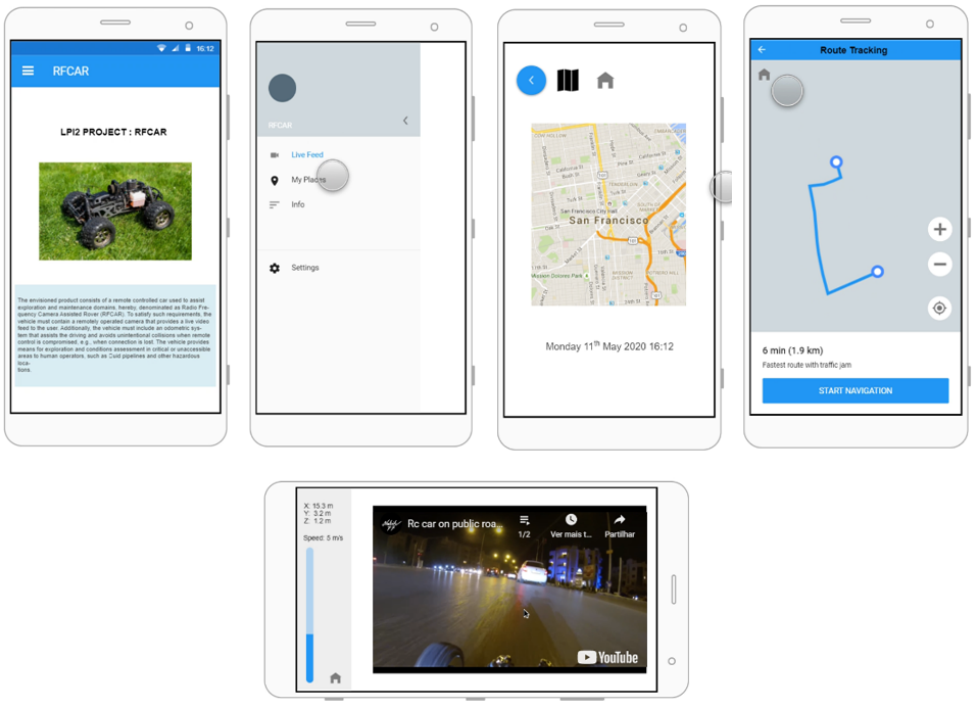

Figure 7.18.: First UI idea

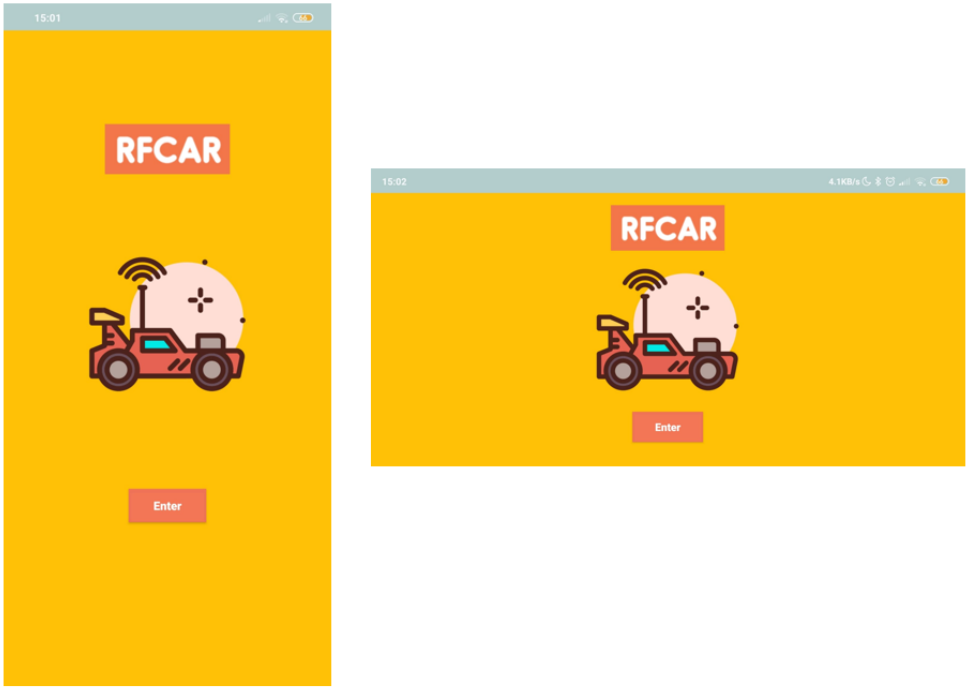

Figure 7.19.: UI orientation test example

drivers (setup) and specific hardware to use that protocol. However, despite both devices having the forementioned drivers and hardware for the connection to succeed, that didn't happen. After some research on the topic, one can assume that the problem was caused by **android version mismatching**. Meaning that the HC-05 only recognised older android versions (prior to 4.2). Fortunately, a new solution was found where one could run the server Bluetooth app in a virtual machine using, for example, COM6 port to communicate, and then redirect that port to another one (COM9) establishing the connection with the phone. In other words, the android app and bluetooth server app can be connected to different ports and still comunicate, due to the foresaid redirection represented in figure 7.20. As an advantage, this method requires less hardware resources.

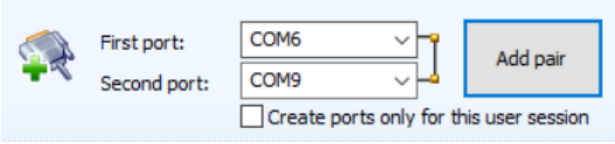

Figure 7.20.: Port redirection

After turning on Bluetooth on both android (ram) and PC devices (JOAO F), as depicted in figure 7.21, one could then open the android app and see in main menu its initial functionalities, figure 7.22. The app manually requests Bluetooth enabling if it isn't in the beginning.

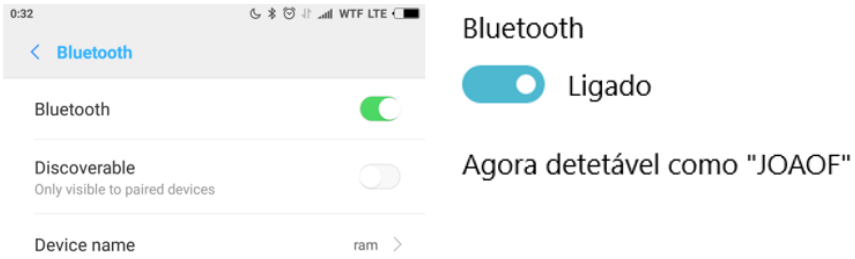

Figure 7.21.: Smartphone as "ram" and PC as "JOAOF" with Bluetooth enabled

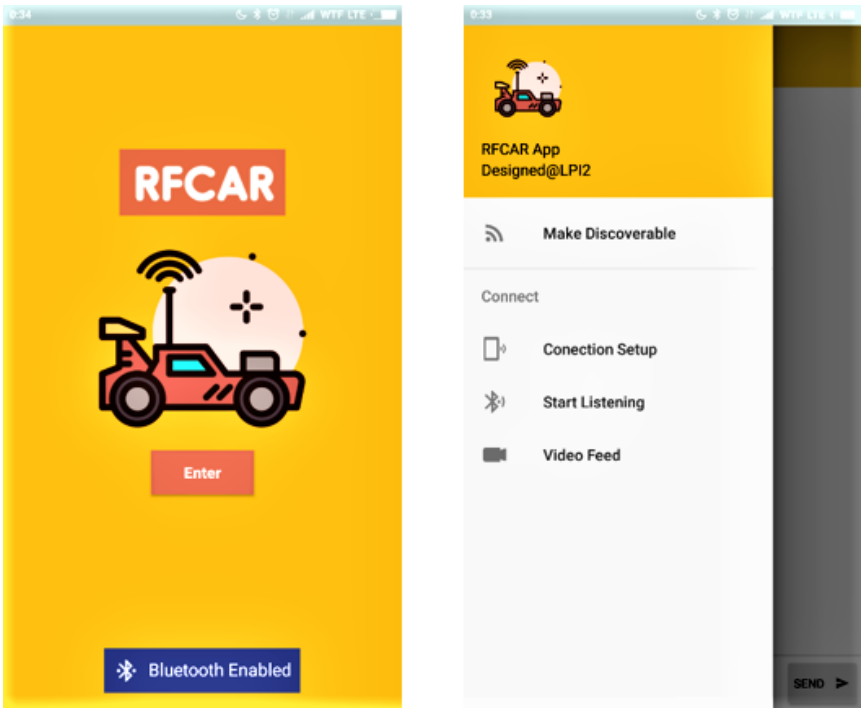

Figure 7.22.: RFCAR app: Initial and main activities, accordingly (**Test UI**)

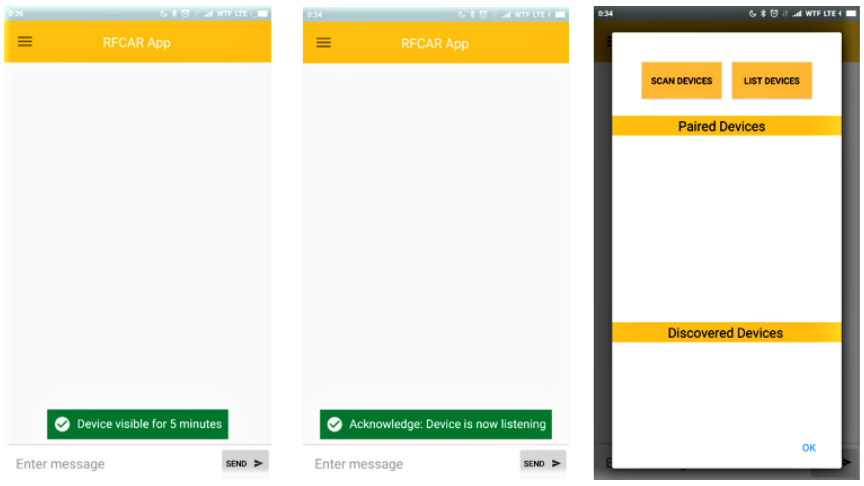

Figure 7.23.: RFCAR app: Discoverable, Start Listening and Connection Setup activities, respectively

To make an initial connection from square one, the smartphone should be discoverable and then be able to start listening, so that one could configure its connection setup. To do that, the options available in the main menu were clicked in that sequence, and when making it discoverable (even if only for 300 seconds) one should accept, figure 7.23.

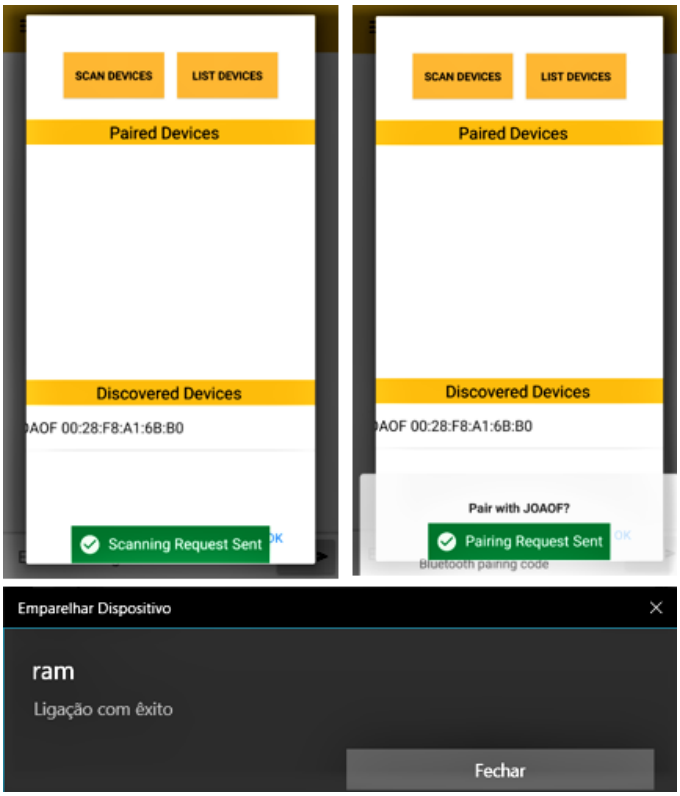

Figure 7.24.: RFCAR app and PC: Pressed Scan Devices button, target device pressed and paired successfull message on PC. A PIN number given by the PC must be inserted on app to conclude the pair phase

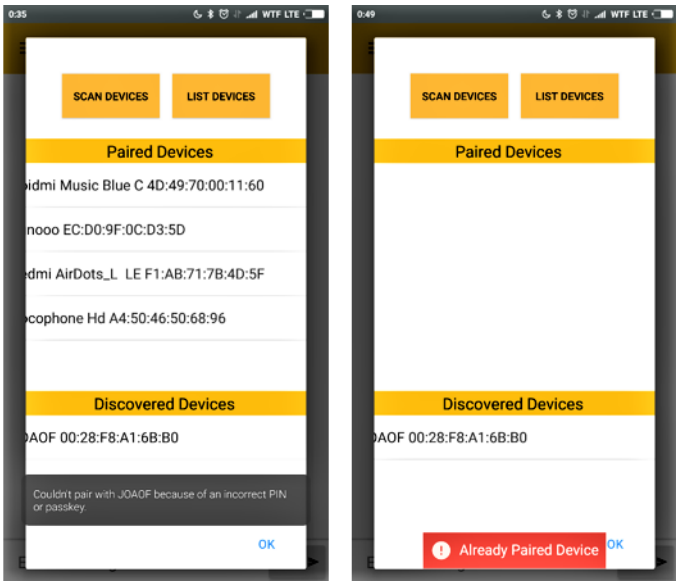

Figure 7.25.: RFCAR app: Pair failed examples: Wrong pin number or already paired device

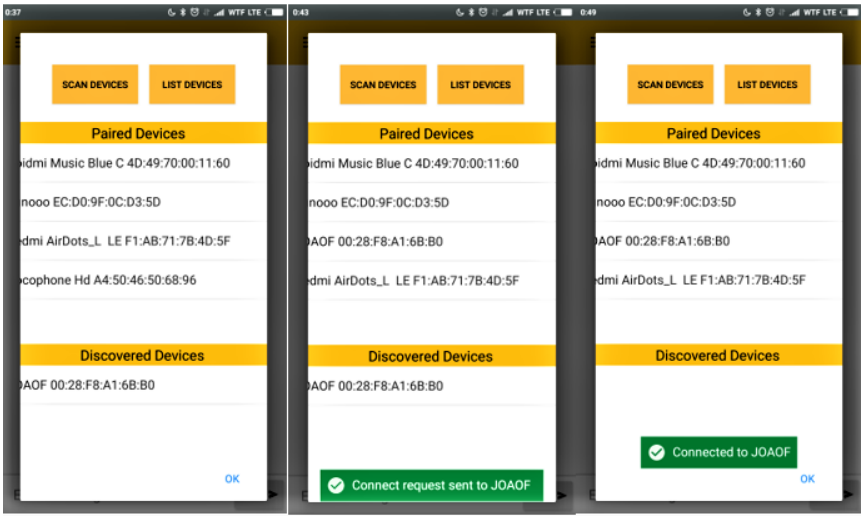

Figure 7.26.: RFCAR app: Connect phases: List devices pressed, target deviced pressed and successfull display of connection establish toast

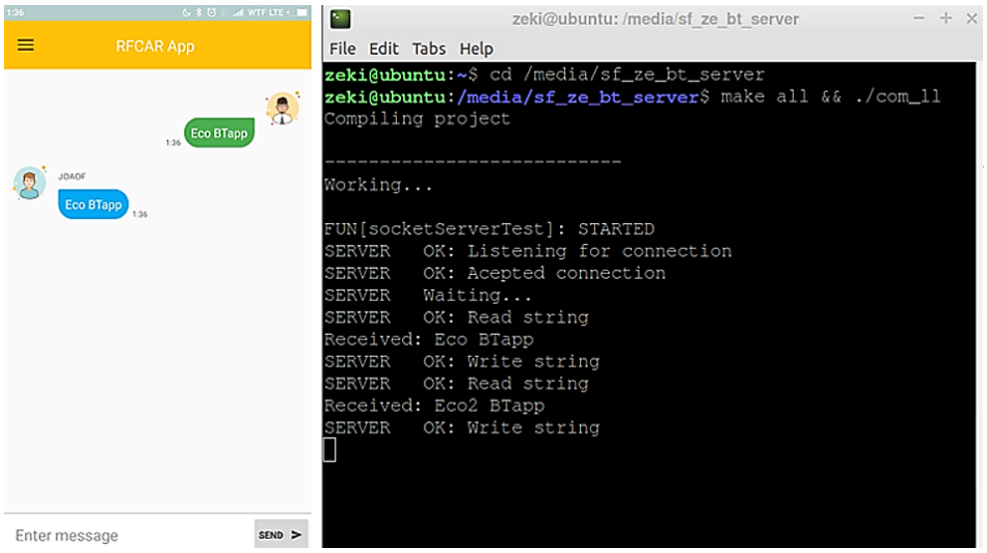

Figure 7.27.: RFCAR app and PC BT server: Successfull message exchanged from android to pc and android echo

When on the connection setup phase, the scan button was clicked to show the PC to connect as a discoverable device and following that, that same device detected was clicked to pair up with the smartphone. When the pairing phase is completed, one out of two outputs happen. The PC device pairs, as proven in last capture of figure 7.24, or doesn't as depicted in figure 7.25, as an example. Error messages would pop up whether the PIN introduced on the PC was mismatched or it was already paired in the first place, accordingly. On the former alternative, the connection was possible after pressing list devices button and clicking on the same target device as when pairing, sequence of events displayed in figure 7.26.

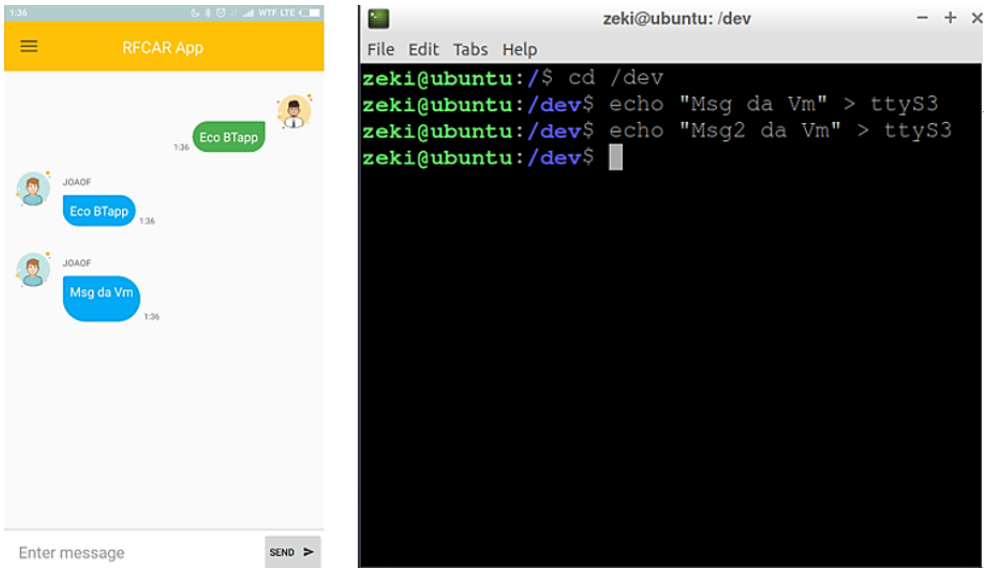

Figure 7.28.: RFCAR app and PC BT server: Successfull message exchanged from pc to android

Figure 7.29.: RFCAR app: It's possible to resend and receive more than once

Lastly, after opening output log terminals to receive app messages on the personal computer virtual machine guest operative system, on the principal menu, in the Enter message section, the messages were typed and then sent via clicking the adjacent send button. The message was then echoed on the phone and conveyed to the target (JOAO F) device, where they were displayed. Figure 7.27 show app and PC terminal points of view. Regarding figure 7.28, the message was written on the terminal and received on the smartphone, respectively. Figure 7.29 display final state on the app log after resending a different message as well as obtaining another one via the same source. Note that the PC terminal images already depicts all messages exchanged and show the initial command configuration needed for the test to happen.

On one hand, in the receiving terminal, the first command: cd /media/sf_ze_bt_server changed the directory to where the server app is located. The second: make all && ./com_ll compile all the source files and run com_ll.

On the other hand, in the sending terminal, the current directory was changed to /dev with the command: cd /dev to navigate to where virtual COM4 (ttyS3) so that messages could be sent through, as exemplified in the instruction: echo Msg da VM > ttyS3.

After properly testing the message exchange between the smartphone and the NVS one would simply need to start sending the control values (speed and wheel tilt percentages) periodically, receive the important alerts from the NVS and redirect them to the notification pop-ups created earlier. The results of these tests are represented in figure 7.30. Note that the video stream presented in figure 7.30 is merely used for test purposes as a way to envision a live rover feed.

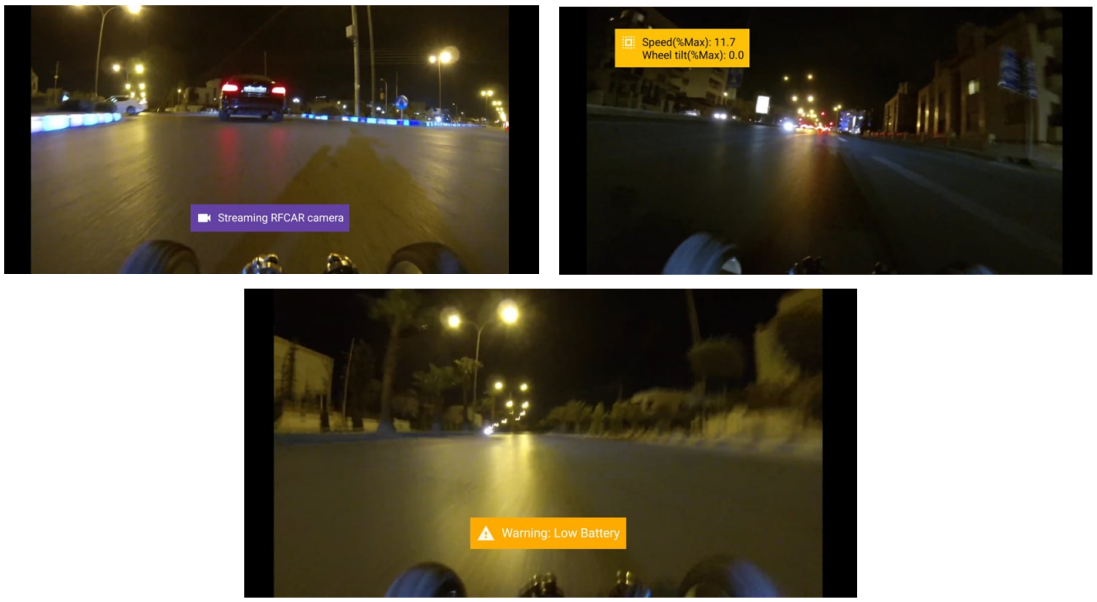

Figure 7.30.: Final product pop-up notification examples

7.3. Functionalities: Summary

The summary of the implemented functionalities is as follows:

- ☒ NVS [3/3]
  - ☒ Digital control of the vehicle: speed and direction
  - ☒ Obstacle avoidance
  - ☒ STM32 full-stack implementation
    - ☒ OS: threads, shared-memory, mutex.
    - ☒ IO: sensors, motors
    - ☒ Communication [2/2]:
      - ☒ Bluetooth: client and server
      - ☒ RS232: client and server
- ☒ Smartphone [3/3]
  - ☒ User Interface
    - ☒ Retrieving smartphone sensors data for simplified vehicle navigation (accelerometer/rotation sensor)
    - ☒ Video Feed
    - ☒ Notifications
  - ☒ Bluetooth: client and server
  - ☒ Wi-Fi: client and server
- ⊟ RVVS [2/3]
  - ☒ Communication [2/2]:
    - ☒ Wi-Fi: client and server
    - ☒ RS232: client and server
  - ⊟ Image Acquisition [1/2]
    - ☒ Image capture from webcam
    - ☐ Video capture and streaming
  - ☒ Telemetry

8. Verification and Validation

After testing the behaviour of the devised solution, its performance should now be tested and analysed in respect to the foreseen product specifications. Additionally, the product should be validated by an external agent to the development team to assess its overall suitability to its intended purpose. In this chapter, the verification and validation tests performed are presented and analysed.

8.1. Verification

In the verification phase, the product's performance is tested and verified its compliance to the foreseen specifications.

8.1.1. Correctness of the control algorithms

After the implementation of the control algorithms, the results obtained were compared with the output of the simulations.

In most cases, the output of the implementation and simulation are very similar, having only changed the stabilization time due to the use of the modified velocity algorithm. Even though in the implementation the car reaches the speed reference faster than in the simulations, the behavior of the car position wise is very similar, and both velocity graphics have the same wave form.

It was also possible to visualize that when the reference angle is not 0, in the implementation, the velocity of both wheels has a very small ripple at the same time. which did not occur in the simulations. These results are prensent in the test section 7.1.1.6.

In general, the implementation results were the expected, since the differences between what was obtained and what was simulated were very not significative.

8.1.2. Image Acquisition

In this section is verified the compliance between foreseen (see Section 3.1.5) and actual specifications (see Section 7.1.2.1).

As can be seen in this latter section, unfortunately, the available webcam is very old, supporting only the uyvy422 format (luminance + chrominance). This prevents the straightforward video capture and its streaming to a remote streaming platform (e.g. Youtube) where it's ubiquitously available through a shareable link. A more convenient format would be the MJPEG.

Concerning the framerate, it can be that the webcam supports 30 fps only. However, it's actual value in dependent on system's load, but, as video capture was not possible, consequently, it is left undetermined.

For the same reason, and although several resolutions are available (640x320, 352x288, 340x240, 176x144, and 160x120), it was only possible to test out image acquisition at these resolutions, with success, but lacking the video capture tests, which is not critical, as it does not depend on dynamic conditions, only on a statically defined capability of the webcam.

8.1.3. Functionality

The end goal in this stage was to create a main system composed of subsystems that interacted with protocols like Bluetooth, Wi-Fi, GPRS and RS232 to assure the speed reference and wheel tilt commands reached the (virtual) rover. Succeeding, the rover should move accordingly to those commands altering its virtual coordinates. After the phase of integrated testing, Section 7.2, one can affirm that this goal was half met since the commands reach the virtual vehicle but only through one communication source (Bluetooth). The latter can also communicate back sending the alerts related to the status of the rover, once again resorting to Bluetooth. The final virtual environment block diagram is depicted in Fig. 8.1.

8.1.4. Communication

Considering subsystem communication the two parameters to verify were Reliability and Redundancy, since the Communication Range was reliant on a physical prototype test.

8.1.4.1. Reliability

Concerning Reliability, one needed to implement a system where the Wi-Fi dropped packets vs total packets ratio was monitored. However, since the system lacks the control message exchange through Wi-fi this goal is impossible to fulfil. Despite that, the Bluetooth control message exchange was implemented and

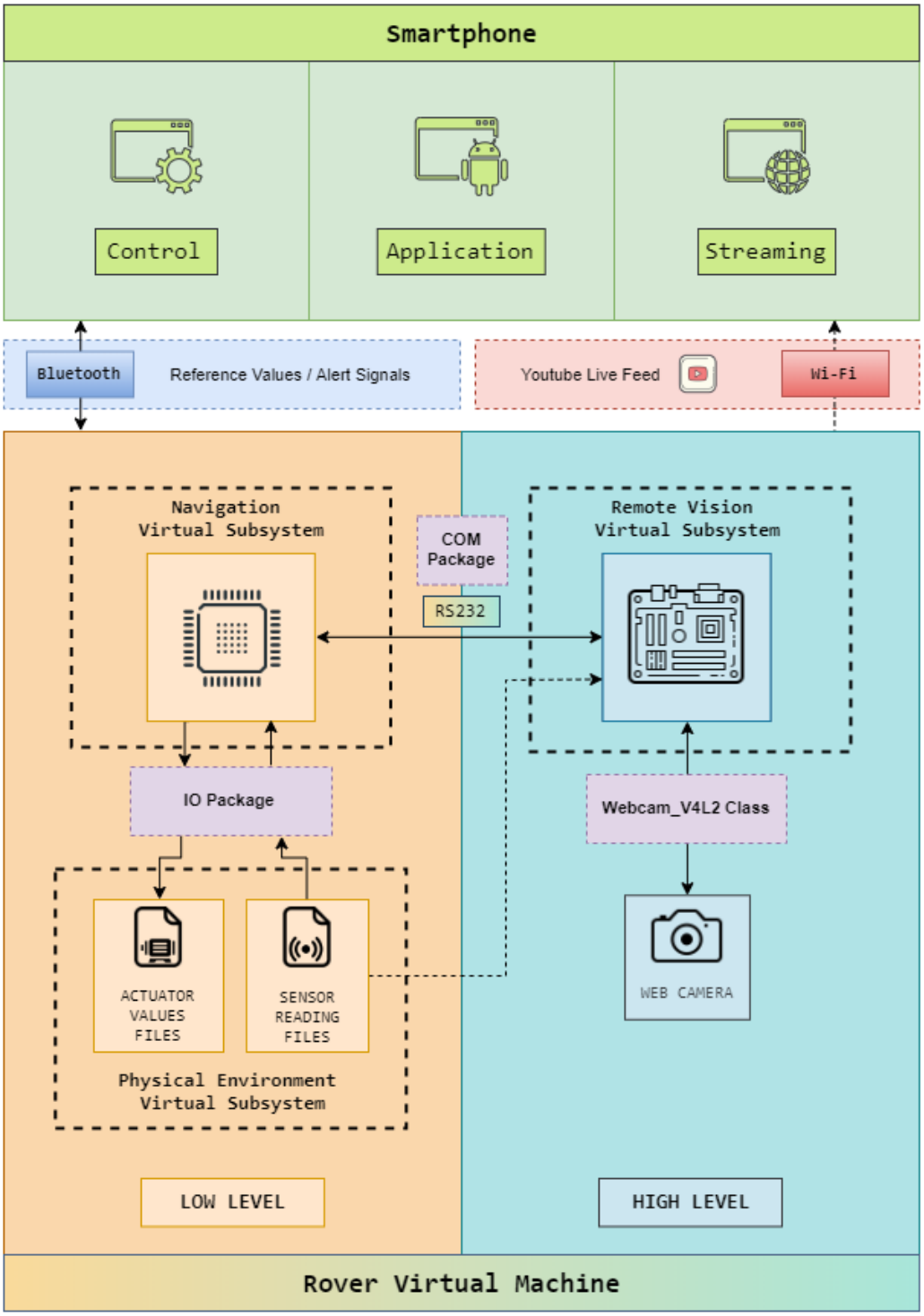

Figure 8.1.: Final virtual environment block diagram view

tested.As these tests were made the package loss was analysed and one can conclude there is little to no package loss when sending the commands from the smartphone to the NVS over Bluetooth since no message was lost. Despite not using a Wi-Fi, the Bluetooth protocol was still a reliable option.

8.1.4.2. Redundancy

Regarding redundancy, in the initial design (Fig. 4.1) it was established the Wi-Fi as the main communication and the Bluetooth as redundancy communication. Later, the main communication was changed to Bluetooth and, as aforementioned, there is no redundancy communication for sending commands to rover when the latter fails.

8.2. Validation

In this stage, the product is tested by an external agent to the development team to assess its overall suitability to its intended purpose. For that reason, the app was tested by a user outside of the workgroup. Instructions were given to him so that he could understand the concept to navigate through the user-friendly app and report any bugs or challenges faced when performing the following steps:

Step 1) Introduce the user to the last version of the android application: RFCAR app, after instructions given by one of the developers.

Step 2) When prompted by a system message pop up telling to turn on Bluetooth, turn on Bluetooth.

Step 3) Press Enter Button. Now, on the main menu, where a idle message, youtube icon and an arcade controller appears, press the button on the top left corner.

Step 4) Press the Bluetooth Discover button to allow the smartphone to be visible to other devices. Press Allow when prompted by the system message.

Step 5) Press the Connection Setup button on the lateral sliding tab.

Step 6) Press the Scan Devices button and wait until the Bluetooth enabled target device appears on Discovered Devices subsection and press it. If the target device to pair with doesn't appear, turn on its Bluetooth. A message to conclude the pair phase should appear on both device with a corresponding PIN. Certificate that the PINs match and then click allow and yes on both devices to conclude this stage.

Step 7) Press the List Devices button and wait again until the target device appears on Paired Devices subsection and press it to establish the connection.

Step 8) On the sliding tab, press Enable Wifi to turn it on. The device tries to automatically connect to a given Wifi network. If that happens, skip to step 10.

Figure 8.2.: The final version of the app is launched

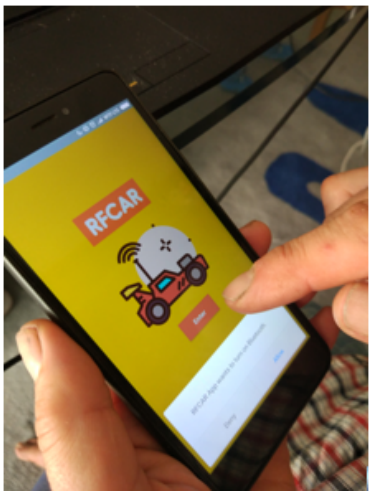

Figure 8.3.: Bluetooth is enabled, then Enter button is pressed

Step 9) Go to main menu and press the Wifi Discovery button and select the desired wireless network on the Wifi Devices section. Touch Wifi Msg to send an initial message to that network to initiate connection. Input the password if prompted.

Step 10) Touch the main menu. A full-screen youtube window and some accelerometer statistics emerges corresponding to speed and wheel tilt relative to its maximum values. Touch the play button and tilt the phone to see that the values are passing to the other Bluetooth device.

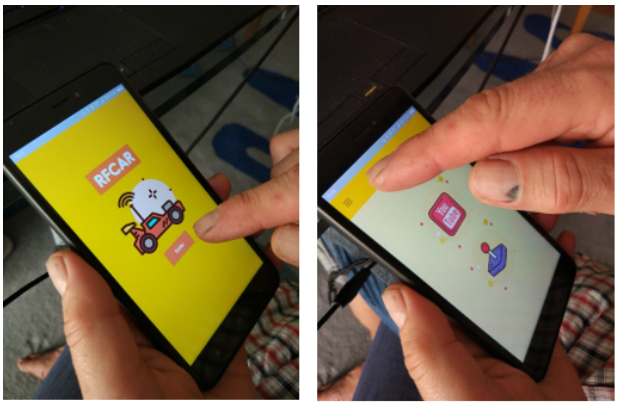

Figure 8.4.: On main menu, press the top left corner button to open options tab

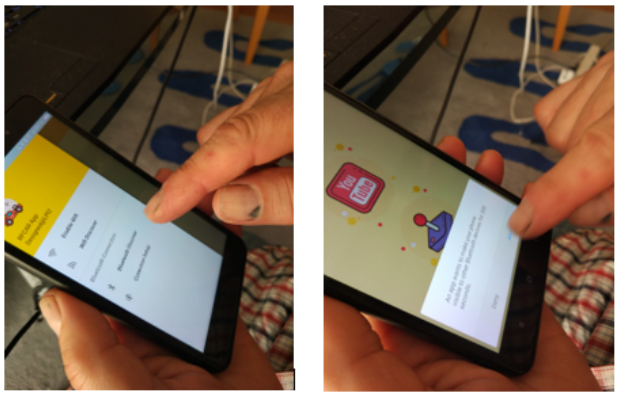

Figure 8.5.: Bluetooth Discover is pressed, turning the smartphone visible to other devices

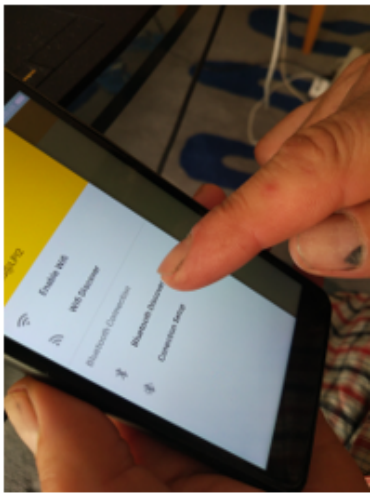

Figure 8.6.: Connection Setup is pressed to configure Bluetooth connection

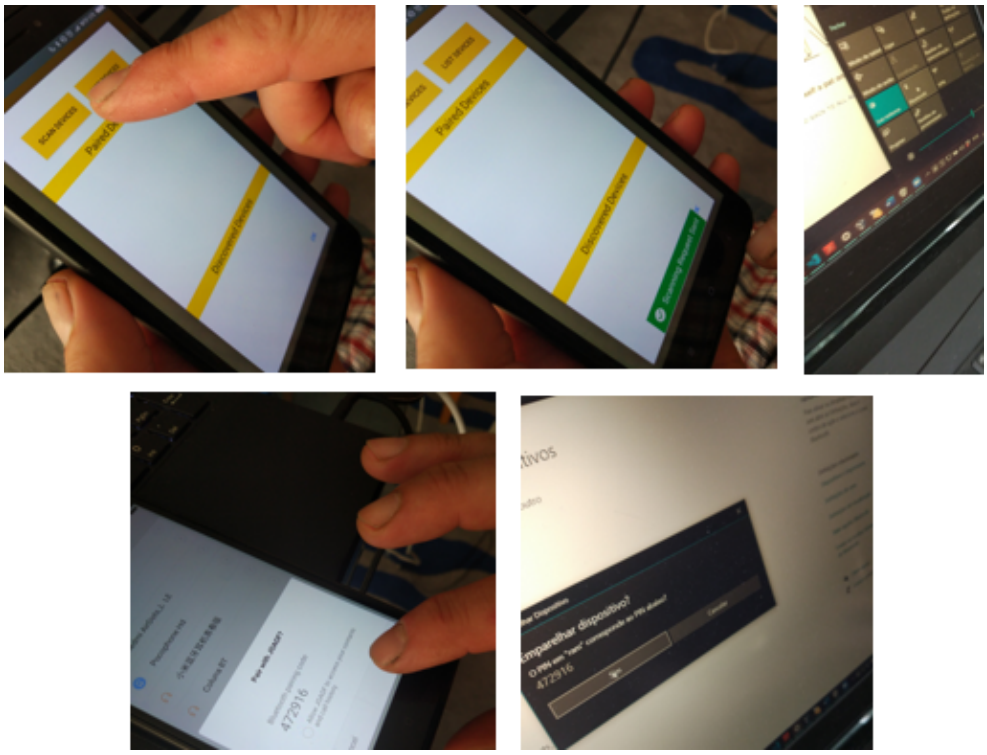

Figure 8.7.: Scan devices button is pressed, the desired target device to pair with is touched, and the pair phase concludes after the user verify that the PIN of the two devices matches and press yes on both messages displayed on each device

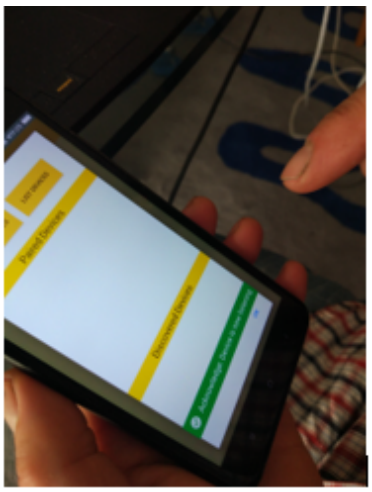

Figure 8.8.: A connect request is sent and shortly after accepted by the target device

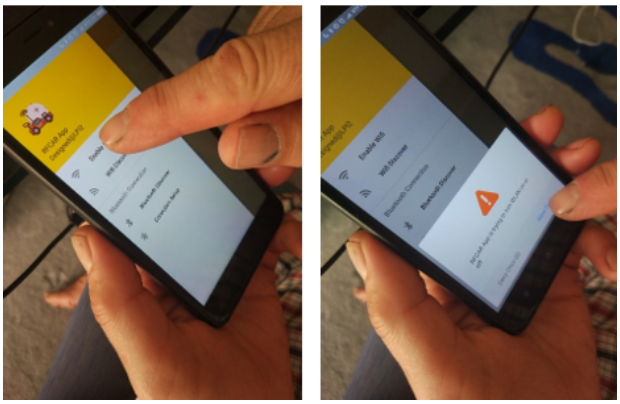

Figure 8.9.: Enable Wifi button is pressed

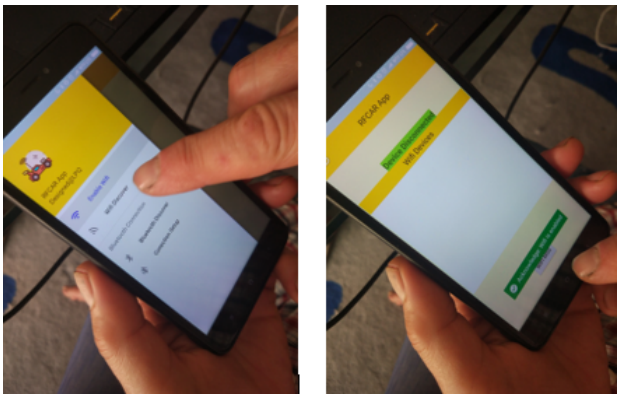

Figure 8.10.: The desired network is selected after pressing Wifi discover button

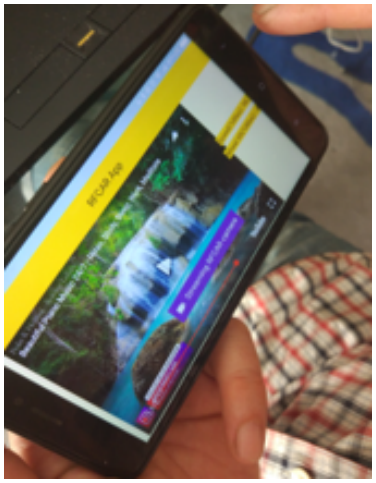

Figure 8.11.: The application sends accelerometer values to the desired device when phone tilts, and a video feed is enabled