



**Politecnico
di Torino**

POLITECNICO DI TORINO

Master's degree
in Mechatronics Engineering

A.y. 2021/2022

Degree session: July 2022

Hardware and Software development of a remote interface to program Embedded Systems

Supervisor:

Maurizio Martina

Candidate:

Carlo Fiori

Abstract

In recent years, the need of remote control of various electronic devices has grown more and more, especially in the automotive sector. Some of the main reasons are: to speed up design times, to facilitate the cooperation between companies (by testing in real-time various alternatives and/or changes in the projects), to limit the physical presence of employees in a laboratory or in a company, also due to the pandemic situation.

This project intends to give a specific method to develop Hardware and Software design of a particular electronic device, that permits to program through a remote interface an embedded system, like an electronic control unit. The work was divided into two main parts: hardware design and programming/web design, both made with the help of a device called Raspberry Pi, in particular the Raspberry-Pi 3 model B+ was used, on which the Raspbian Operating System was installed. By means of this electronic device it was possible, through some simple programs, to manage its own GPIOs (General-Purpose-Input-Output pins), to which various electronic devices have been connected. In the first part of the hardware design, various electronic components, mainly produced by companies such as RS, Farnell and Digi-key, were carefully selected and integrated together, first mounted on a breadboard (base used to create circuit prototypes) to test and verify the functionality of each individual component, and then welded together in a multi-hole plate to reduce the amount of wiring. In the second part dedicated to programming/web design, the web application was created, with the development of a part called “front-end” (web interface with

which the user can interact) written in HTML and JavaScript languages, and a “back-end” part (not accessible to the user and that communicates with the front-end) built with the help of NGINX and some servers.

During the hardware design phase, Hardware-In-the-Loop tests were performed in order to verify the correct functionality of all the components which constitute the PCB: this procedure consists of connect external modules, which can be switched on/off from the Raspberry-Pi device, to the PCB and then test if they are detected from the RPi itself. At the end, the overall system, PCB and Application Web, were tested connecting the RPi with an ECU.

This electronic device, which can be called Smarthub, was created ad hoc according to the needs of the Abinsula company where the work was carried out, to manage and program an electronic control unit. Anyway, it can be useful for many companies, in particular the multinational ones, which want to control and manage remotely in an efficient way various devices (telematic boxes, motherboards, production machines, etc..) from different sites and in any moment.

Contents

1	Introduction	9
1.1	Thesis scheme	10
2	Hardware design	13
2.1	System description	14
2.2	Rapsberry Pi	16
2.2.1	Model Used	17
2.2.2	Setup	18
2.3	PCB design	19
2.3.1	HIL (Hardware-In-the-Loop) test	26
2.4	3D MODEL of the PCB	31
2.4.1	KiCad	31
2.4.2	Electric Scheme	32
3	Software design	37
3.1	What is an Application Web?	37
3.2	NGINX	41
3.2.1	NGINX configuration	41
3.3	Back-end	44
3.3.1	OPEN-API	45
3.3.2	Node.js	48
3.3.3	STREAMING.py	50
3.4	Front-end	52
3.4.1	index.html	53

3.4.2	index.css	55
3.4.3	button.js	56
3.4.4	coding.js	57
3.4.5	serial.js	57
4	Validation Test	59
4.0.1	Test case	60
5	Conclusions	63
	Nomenclature	64
	List of figures	66
	Bibliography	68

Chapter 1

Introduction

This thesis gives a detailed procedure on how to design, from both Hardware and Software point of view, a specific PCB (electronic device called Printed Circuit Board), able to manage various external devices and useful to program an Electronic Control Unit, or, in general, an electronic board (e.g a Motherboard, Telematic box, Device Under Test, etc...).

This part provides a general information about objectives obtained and a description, at the end, of the content of various chapter, providing to the reader the followed steps.

The aim of this project is to create a simple extension board for the Raspberry Pi device, called "*Shield*": this component allows to group in a specially sized space other electronic modules that the basic model of Raspberry does not possess. This Shield was created for interface a computer to an Electronic Control Unit to be programmed remotely: for this purpose, many peripherals (such as Programmer, USB Pen drives, Camera, USB2ETH, USB2CAN, SERIAL USBs) must be managed remotely, for example switching them on/off during the programming phase. This work makes possible the connection of an external computer to the Raspberry, through an Application Web interface. In this way everybody can control and perform the programming of any board (like Motherboards, Device Under Test, and much more) remotely.

In particular, the main followed steps are:

- Setup of Raspberry-PI 3 (B+ model) in order to manage the external peripherals through the PCB: the Operating System Raspbian (OS of Linux Debian distribution) is installed;
- Research and selection of components able to switch on/off USBs, Power Supply, USB2CAN, USB2ETH, SERIAL USB modules;
- Check functionality of each component mounting them over a bread-board;
- Weld together the components in a multy-hole plate to reduce the amount of wirings;
- Install Camera device useful to check the correct functionality remotely;
- Design of Electronic scheme of the PCB and its 3D model with KiCad 6.0 software;
- Develop an Application web in order to interface an external computer to the Raspberry, connected to the designed PCB;
- Test the overall system.

1.1 Thesis scheme

Right away, a brief description of the content of each chapters, in order to understand better the followed procedure:

- Chapter 2: Hardware design

This chapter fully describes the Hardware design of the PCB, starting

from the selection of various components, of which will be given the relatives Datasheets and functionalities, until the description of the overall Electric circuit, with a section dedicated to the 3D realization through KiCad 6.0 software;

- Chapter 3: Software design

This section is dedicated to the Software design: from the compilation of the back-end part, written in Python and Javascript languages, that deals with the creation of web servers for the remote managing of the external components linked to Raspberry Pi, until the styling of the front-end part (webpage), written in HTML, CSS and Javascript languages.

- Chapter 4: Test and Validation

This section is about test and validation phases, useful to check the right functionality of the PCB and the correctness of Application web.

- Chapter 5: Conclusions

The aim of this chapter is to give general informations on what is done during the overall design, on the results and on the possible improvements to perform in order to adapt this work to various scenarios.

Chapter 2

Hardware design

This chapter is dedicated to the accurate description of the procedures followed to build the Printed Circuit Board (PCB) as a Raspberry-Pi shield. This device, at which various external devices will be connected, is managed and driven by the Raspberry Pi. In particular, this PCB must be capable of:

- Switch on/off at most eight USB modules, among which: USB Pen-drives, Programmer (USB side), USB2-ETHERNET cable, Camera Video, USB2-CAN cable and SERIAL USBs for serial communication. For this aim, two types of components must be selected: one deals with the VCC and the other with the Data BUS D+ and D- of the USB modules;
- Power on/off the power supply of the Electronic Control Unit that has to be programmed;
- Detach and reattach the ARM-JTAG (10 pin cable), through which the ECU is linked to the Programmer;
- Short-circuit two pins mounted on the ECU used for its Boot (set of processes used for the load of *kernel* of the ECU).

These requirements are fundamental for the programming of the ECU, since, for example, to flash the memory of its processor, there is the need to switch off the power supply before connect the Programmer. Once the Programmer is accessible from the Raspberry pi, the Power Supply of the ECU can be switched on and then the USB ports related to the Serial Communication can be activated.

In the past, many software problems on the ECU were found after the ECU itself was installed in the vehicle: this was a huge problem since the ECU had to be replaced with another one. Nowadays instead, OEMs (Original Equipment Manufacturers) offer the possibility of using *flashing* procedures: a set of software and standards to install on the ECU without remove it from vehicle. Without enter into detail of programming phases since it is not part of the project, there exist some programming operations executed during the ECU boot, which ensure that the firmware is updated and check the functionality of the application (for example, one of main the application installed on any ECU is the ABS system, that does not lock the wheels during the vehicle brake) installed on the ECU.

2.1 System description

Before going into the details of the PCB design, it is important to understand how the overall system is composed, from the remote computer to the ECU (or DUT) to program. The next figure explains better the structure:

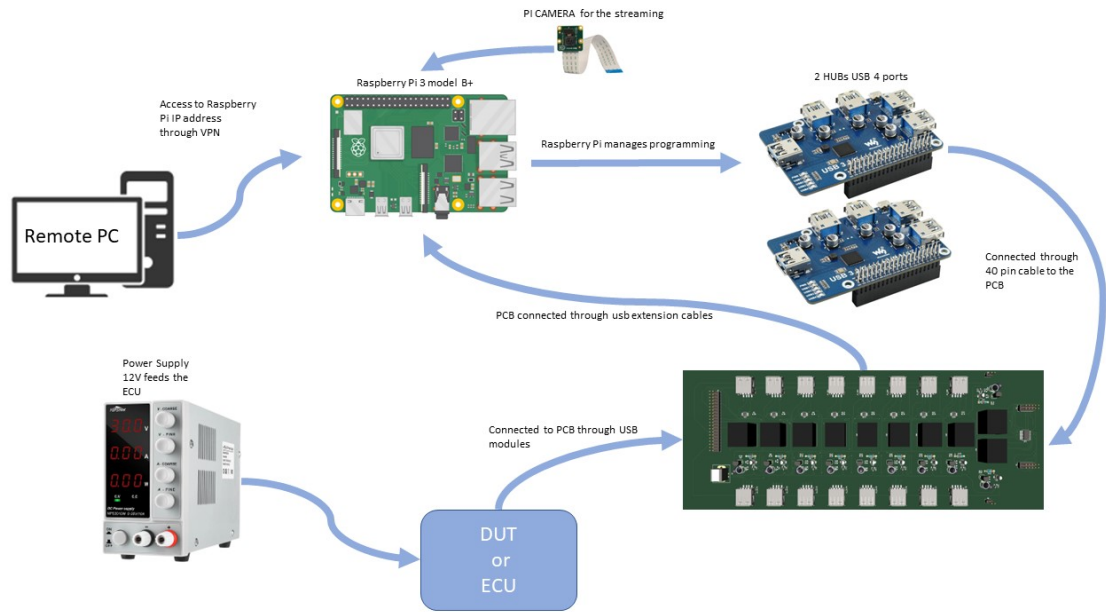


Figure 2.1: Structure of the system.

As shown in figure 2.1, it is possible in every moment with a remote PC to connect, through an IP address, to a simple website in which you can interact with the Raspberry Pi, where the Application Web is compiled. To make it possible, it is important to connect the computer, by way of VPN, to the IP address of the Raspberry Pi. At this point, the Raspberry device, equipped with two HUBs USB, is now linked to the designed PCB through a 40 pin cable. This cable is very important because allows to link GPIOs with PCB components that have to be managed. The ECU (or DUT), the last component, is powered by 12 V Power Supply and it is connected to the Raspberry Pi (passing through the PCB) with different cables (Serial Cable, ARM-JTAG cable, USB2ETH cable, USB2CAN cable and Programmer). The overall connections will be widely explained in the chapter 4.

In particular, the designed PCB can be described as a composition of 3 sub-systems: the first sub-system is composed by eight couple of FSUSB30MUX/Relay-Groove, with the aim of switching the eight USB modules described before; the second one is assembled with two Relays,

of which one deals with the Power Supply and one of the Boot; the last subsystem includes only the 10-BIT Bus SWitch, for the ARM-JTAG cable.

2.2 Rapsberry Pi

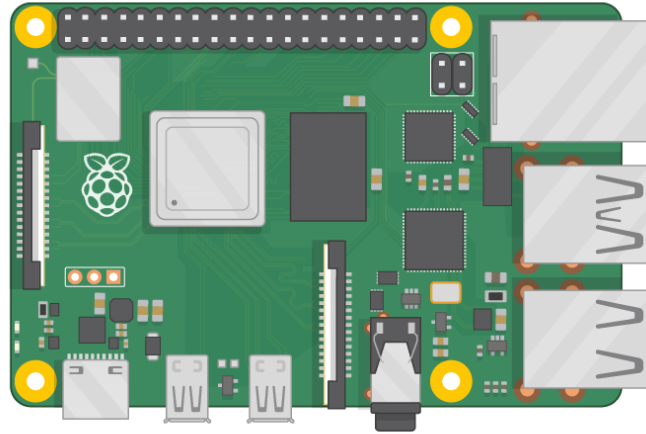


Figure 2.2: Raspberry Pi.

The very first step was to interface with the Raspberry Pi's world. This very small version of a contemporary computer, with a very little dimension, was first delivered in 2012 by the United Kingdom Raspberry Pi foundation, with the aim of promoting subjects such as computer Science and Internet of Things (IoT) to a varied audience, in a faster and more immediate way. Equipped with Raspbian OS, Linux-Debian Operating System based, it behaves like a real computer: for this reason it is accessible to everyone and perform a lot of basic operations, from simplest (search on Internet, send emails, manage document, etc...) to most difficult ones. It can be described with the concept of *"Single Board Computer"* (SBC): in fact it does not have the need of external hardware to work, but only few peripherals (RAM, CPU and GPU) are needed to interact with this device. The most important portion of Raspberry Pi are the GPIOs (General-Purpose-Input-

Output pins), through which it can interact with external components, such as sensors, display LED, Voltage Regulators, Relays, Integrated Circuits (ICs) and much more. Hence, thanks to its simple configuration and to its small price, this very useful device can be adopted in projects for personal use and in general for Embedded Systems. Furthermore it has various fields of applications, such as Domotic, Automotive, Automation, Education and so on [7].

2.2.1 Model Used

The first choice was the Raspberry Pi model B+ V1.2, with a RAM memory of only 512MB. However, its processor (ARM11) isn't compatible with the tools of the Programmer needed for flashing the memory of the ECU. Hence, a more suitable choice, is the usage for the entire project of the Raspberry Pi 3 model B+, with the following specifications [2]:

- **Processor:** Broadcom BCM2387, 1.2GHz Quad-Core ARM Cortex-A53
- **Memory:** 1GB LPDDR2 SDRAM;
- **GPU:** Dual Core VideoCore IV® Multimedia Co-Processor;
- **MEMORY:** 512MB SDRAM;
- **OS:** Raspbian (LINUX-BASED OS) Boots from Micro SD card;
- **POWER:** Micro USB socket 5V/2A;
- **CONNECTORS:** Ethernet, Video Output(HDMI), Audio Output(3.5mm JACK), 4 x USB 2.0, 40-pin 2.54mm pitch, Camera Connector(15-pin MIPI Camera-Seria-Interface), Display COnnector (15-pin Display-Seria-Interface), Memory Card Slot

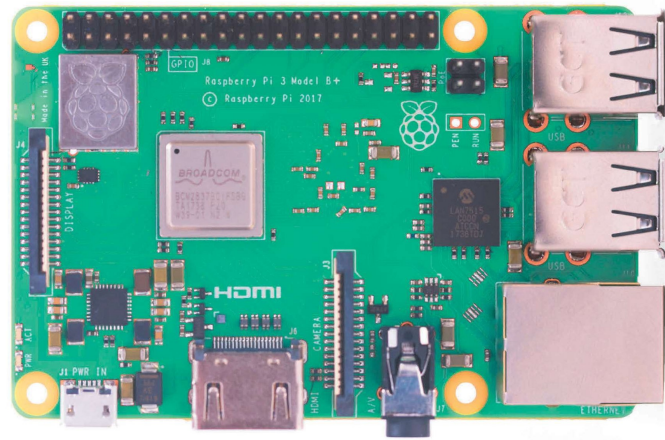


Figure 2.3: Raspberry Pi 3 model B+ [2].

2.2.2 Setup

The very first operation was to install the Operating System on the Raspberry-pi device: for this purpose, with the help of an external PC equipped with Windows 10 OS, it was possible to install into a SD card (32GB memory) the Rapsbian-OS Desktop Version (LINUX-BASED Operating System) through the software Raspberry Pi Imager, downloadable for free from Google.



Figure 2.4: RaspberryPi Imager [15].

Once the OS is installed, the SD card was insert into the Raspberry Pi appropriate slot, and the Raspian OS was accessible to begin its setups. At this point, two main operations was performed: the installation of packages and the configuration for the ARM Core and Linux Kernel.

- **PACKAGES**

Through Raspbian shell it was possible to install many packages: not including those installed for dependencies, the most important ones are: *"python3"*, that allows to program into python language; *"nodejs"*, essential for the Application Web design (JavaScript environment) and *"nginx"*, for the creation of various servers that will be explained in the chapter 3.

- **INTERFACE OPTIONS CONFIGURATIONS**

This operation is very important to interface the Raspberry with external devices that interact with different communication protocols. Through the command line *"sudo raspi-config"* it was possible to access the Raspberry Pi BIOS, where in particular were enabled the Camera (for the streaming), the SSH (to consent remote connection), the SDA/I2C (for the communication with the components used for switch on/off the USB peripherals) and the Serial.

2.3 PCB design

Nowadays, Printed Circuit Boards are the main part of electronic devices in several applications, such as Biomedical (defibrillators, electrocardiogram, pacemakers,...), automotive (ECUs, navigation systems, sensors, actuators,...), aerospace, industrial (CNC machines, electrical equipment,...), or in general, in everyday life (Cellphones, washing machines, microwave ovens, Personal Computers,...). The need of building and designing PCBs was born out of the necessity of link together ICs and other electric devices in a stable, fast and simple way: formerly, this interconnection was done

by hand using cables. Today, PCBs have a simple structure, composed by a insulating material and conductor material: they can be organized in two types of configurations, one structured by only one copper layer on top and the other by two copper layers, top and bottom, separated by an insulating material [14]. The design process of the PCBs is a fundamental sensitive step, that starts from the motivation for using the PCB until its conclusive production. Hence, the first operation is to analyze requirements and research of electronic components: these needs are translated into a concept and schematic phase, which deals with functionality, size and operating work of the PCB. Subsequently, components are assembled and the connection are routed. At the end, the overall PCB is tested, and, in case of success, a report is compiled and PCB can be produced [19]. For this specific Project, not all the phases were followed: from requirements mentioned before, components were chosen and weld together in a multi-plate hole, and then a model of the PCB was performed, without any reporting phase and production. The requirements led to the choice of the following components:

FSUSB30MUX



Figure 2.5: FSUSB30MUX [13].

This component is a DPDT (double-pole-double-throw) low-power high speed USB 2.0 switch: it is able to switch between the data coming from two high-speed (480 Mbps) USB input ports. This small IC, is designed for many applications such as cellphones, TV, printers, digital cameras and so on. Main features are [13]:

- Maximum Supply Voltage V_{cc} : +5.5 V;
- Control Input Voltage: +5.5 V
- DC Output Current: 50 mA;
- Operating temperature: 85 C°

The scheme of this component can be summarized as follows:

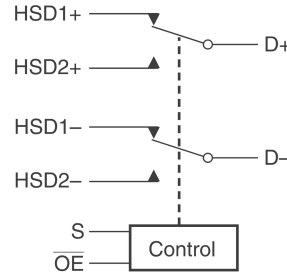


Figure 2.6: Scheme of FSUSB30MUX [13].

In the image above: D+ and D- are the data plus and minus respectively of the output USB; the HSD1+ and HSD1- are the data plus and minus of the first input USB; the HSD2+ and HSD2- are the ones of the second input USB; S is the Control Signal (connected with one of the Raspberry Pi GPIO) and \overline{OE} is a Control Signal used for enable/disable the switch. In this specific project, there is no need to switch the data between two input USB ports, but the aim is only to switch on/off the data coming from the same port. So one of the couple of input (in particular HSD1+ and HSD1-) are left always unconnected. Since eight of this multiplexers are needed, the truth table played a fundamental role to reduce the usage of Raspberry Pi GPIOs. In fact:

Truth Table		
S	\overline{OE}	Function
X	HIGH	Disconnect
LOW	LOW	D+, D- = HSD1 _n
HIGH	LOW	D+, D- = HSD2 _n

Figure 2.7: Truth Table of FSUSB30MUX [13].

the GPIO 2 of Raspberry Pi was connected to all the \overline{OE} s of the eight FSUSB30MUXes and it has been set to LOW (the overline in the symbol OE means that the signal has to be set in the reverse state); at this point the HSD1+ and HSD1- of each FSUSB30MUX were always left unconnected and the external input USB is connected only with the HSD2 port. Hence, with only one GPIO (set to LOW) it was possible to connect all the switches, and with other eight GPIOs every single USB was managed and controlled separately.

To work with this very small component, with only 0.5 mm pitch between its pins, was important to weld them with an appropriate package, the MSOP10 package. The following images show the physical dimension of the component and the package:

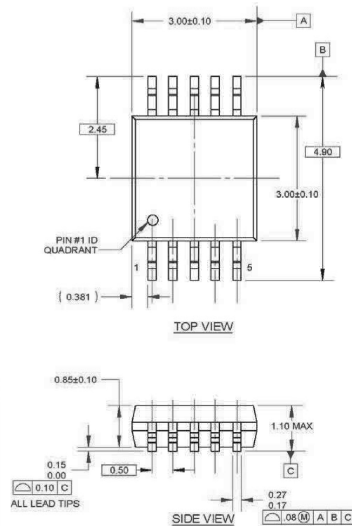


Figure 2.8: Physical dimension of FSUSB30MUX [13].

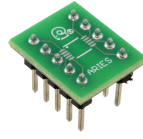


Figure 2.9: MSOP10 Package [1].

RELAY GROOVE v1.2



Figure 2.10: Relay Grove v1.2 [16].

This component is a normally-open and bistable relay piloted with an input signal coming from the Raspberry Pi GPIO: setting the GPIO to high, a led, mounted in the circuit, lights up red and the relay state is changed [16]. For this specific project is used to switch on/off the VCC of the USBs, to short-circuit the two pins of the ECU used for the BOOT and to switch on/off the Power Supply of the ECU (ten of these are used). This component, with regard to switch USBs state, is flanked with the FSUSB30MUX component: they are controlled by the same GPIO, since the data and Vcc of each USB has to be switched on/off at the same time (also in this case the usage of GPIOs is reduced).

SN74CBDT3384

Figure 2.11: SN74CBDT3384 [5].

This component is a 10 bit high-speed Bus Switch: it looks like a 2 separated 5-bit channels each piloted by two different \overline{OE} s and for this specific case is used to switch on/off the ARM-JTAG cable with which the ECU is connected with the Programmer. Most important features are [5]:

- Maximum Supply Voltage V_{CC} : +5.5 V;
- Control Input Voltage: +5.5 V
- DC channel current: 128 mA;
- Operating temperature: 85 C°

In the following image, pins description is shown:

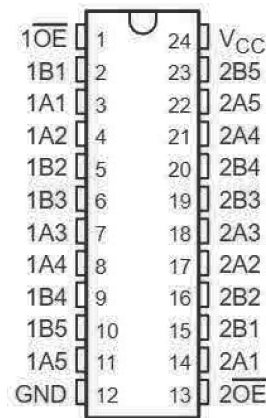


Figure 2.12: Pin description of 10 BIT BUS SWITCH [5].

As it can see, there are two separated channel, 1An/1Bn and 2An/2Bn, each piloted by two different control signals: the $\overline{1OE}$ control pin pi-

lots the left side bus connection and the the $\overline{2OE}$ pilots the right side bus connections (5 channels): for this specific application the left side and right side are piloted by the same pin at the same time, since there is the need of detach/reattach a 10 pin cable. As described for the FSUSB30MUX, also for this IC was important to chose a suitable package to be able to connect it to the Raspberry Pi GPIOs, since it has very small pins with 0.65 mm distance between each other.



Figure 2.13: TSSOP 24 [3].

2.3.1 HIL (Hardware-In-the-Loop) test

Hardware-in-the-Loop is an approach test used mostly in automotive fields: it is born in Aerospace and Defence industry in the 1950s, from the necessity of ensure security in human life during the executions of tests. Then it has expanded in automotive fields, since the usual design processes involved in control system development require a lot of validation phases repeated in loop. For example, the design process starts from the analysis of requirement, followed by tests and eventually, if test is not passed, by the redefinition of the requirements. Hence, before producing a embedded system, this test ensures costs and time saving [9]. It is widely used in Model-Based-design processes: in fact, it permits to validate a generic real-time application connecting them with the physical system [11]. In this part of the project, with HIL test we mean a procedure in which Raspberry Pi manages components visible from the device itself. The overall test, explained better in the chapter 4, is slightly different, since from a remote PC in laboratory, through the website, the correct functionality of the PCB and the Application Web is verified connecting them together with a physical ECU system.

The first test case, called *Local HIL*, is made with the usage of a breadboard:

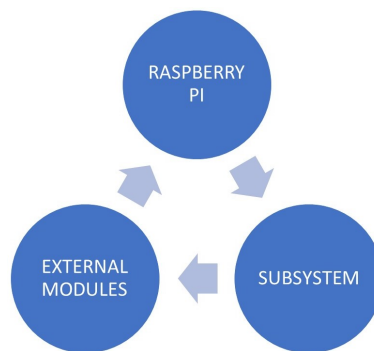


Figure 2.14: LOCAL HIL test.

This test consists of manage separately all components with Raspberry Pi device, in which we can see if an external component is switched on or switched off. Hence, with a simple program compiled in Python language, the components were piloted through GPIOs. The test was repeated in loop until the correct functionality of all components was verified. In the following image, the functionality of the couple FSUSB30MUX and Relay-groove is tested linking them together: this subsystem is able to switch on/off an external USB module (such as a Mouse, Keyboard or Pen Drive), visible from the Raspberry Pi itself.

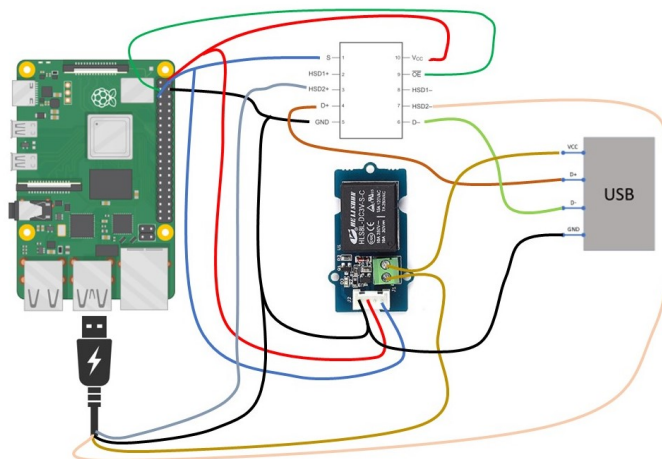


Figure 2.15: Connection between FSUSB30MUX and Relay-Groove.

Since the previous test is successfully passed, the other two sub-systems (two Relay-Grooves and the 10-bit Bus Switch) were tested with the help of a Multimeter. The following image shows the connection between Raspberry-Pi GPIOs and the electronic devices:

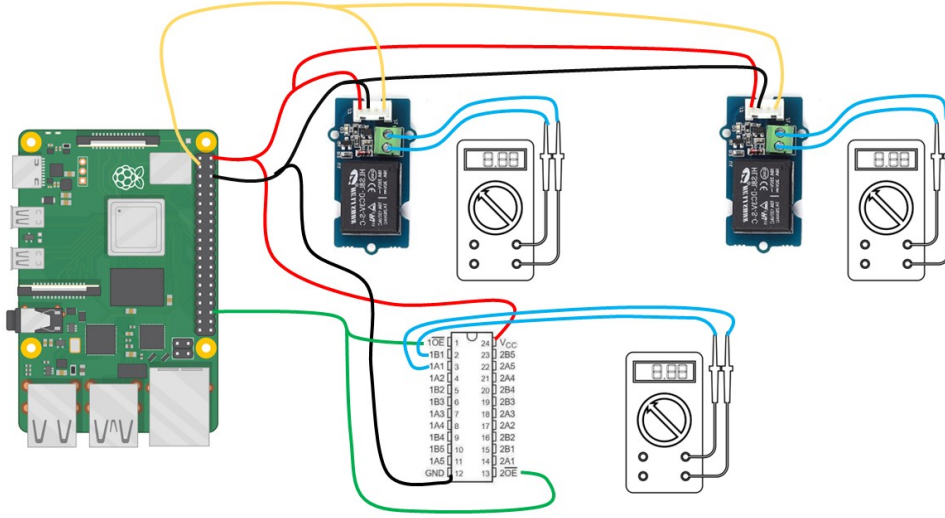


Figure 2.16: Connections of two Relay-Grooves and 10-BIT Bus Switch.

At this point, all components were melded together on a multi-hole plates: this step was essential in order to reduce the amount of wiring, since the cables came loose easily. Moreover, this operation allowed to group all the electronic devices into a organized space subdivided into two layers. For this purpose, other components were integrated in the circuit: in particular, one ARCELI RPi GPIO breakout expansion board type T with 40 pin (together with 20 cm FC40 flat ribbon cable) and a generic connector for the Power Supply, to feed all the subsystems. Since all the components are mounted in parallel (the next sub-chapter explains better the electric scheme of the PCB), a 5V-4A Power Supply is enough to feed the entire system.

It is also important to explain how the GPIOs usage is organized:

- GPIO 2 is used for the \overline{OE} s of the eight FSUSB30MUXes;
- GPIOs 3, 4, 17, 27, 22, 10, 9, 11 are used to control the eight couples previously described;
- GPIO 20 is used for the $\overline{OE}1$ and $\overline{OE}2$ of the 10-BIT Bus Switch;

has shown that not all the devices were read at the same time, but some of those, depending on order in which they was switched on or off, took a little longer to be read by the device. This is due to a simple limitation of the RPi. Since there is no need that all the modules have to be read within a certain time frame, the test can be considered as successfully passed. The next images show respectively the overall connection between components on the PCB prototype and a representation of the real one:

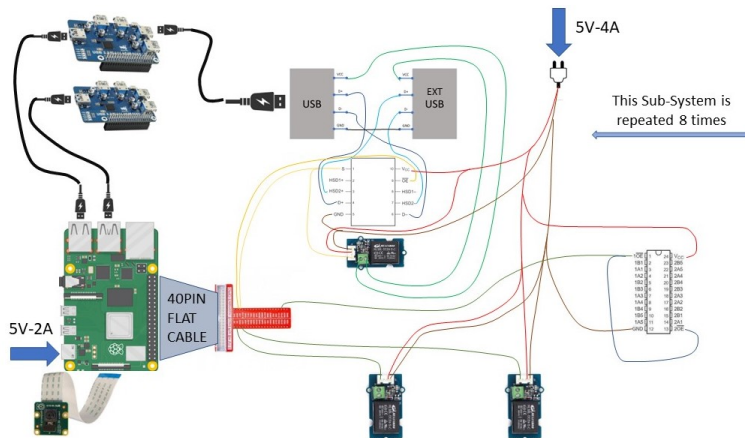


Figure 2.19: PCB prototype connections.



Figure 2.20: Real PCB prototype.

2.4 3D MODEL of the PCB

2.4.1 KiCad

In recent years, grows the need to implement PCB for domestic or industrial use with a fast and simple way. There exist a lot of softwares, such us Eagle, Kicad, FreePCB and so on, that are on sale for the implementation of circuit diagrams and PCB design. In fact, a lot of libraries of various components are shared for users, and this makes the modelling phase an efficient and fast work. For this specific project, KiCad open-source software was used, but same procedures can be adopted on other software since the methodologies are the same. The work on this phase ends on the 3d modelling of the PCB, but a next implementation could be an *home-made* production with the help of a printer, a bromograph and photosensitive copper plate [6]. The KiCad software work-flow can be represented by the following flow-chart:

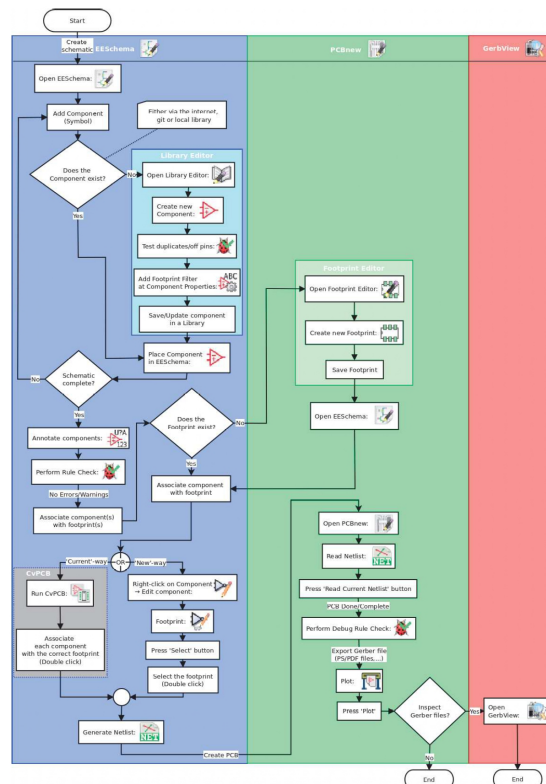


Figure 2.21: KiCad work-flow flowchart [6].

As it can see from the image, the first step is the creation of the scheme of the electric circuit: the symbols of the components are add from standard libraries, otherwise they are created or searched on internet suitable web pages. Once the component is put into the schematic, the relative footprint is added: at this point, the what is called *Netlist* was created and exported into PCBnew, a tool that permits to create the electric tracks between components, to choose the appropriate dimensions of the PCB and to visualize it through a 3D Visualizer.

2.4.2 Electric Scheme

The electric scheme of the PCB was created by adding the symbols of the components previously described:

Coaxial Power connector

This is a schematic of a common connector for the Power Supply input of coaxial shape:

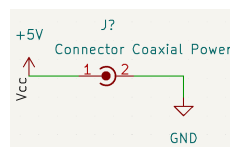


Figure 2.22: Coaxial Power connector electric scheme.

SN74CBDT3384 (10-Bit Bus Switch)

The schematic of the SN74CBDT3384 is coupled with the schematic of two ARM-JTAG 10 pin header, used for the switch of the ARM-JTAG cable with which the ECU is connected to the Programmer:

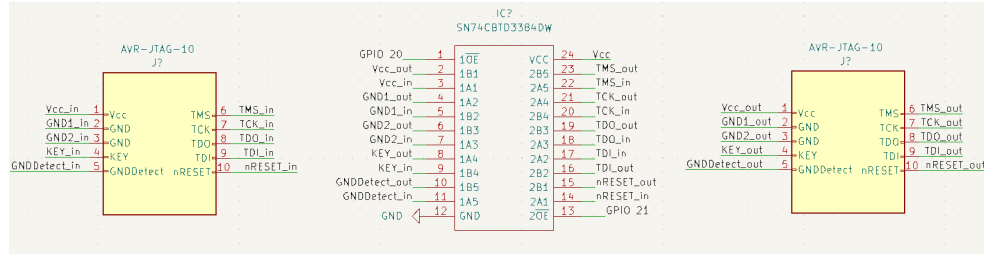


Figure 2.23: 10-Bit Bus Switch electric scheme.

40 PIN header

This component is able to replicate the 40 pins mounted in the Raspberry Pi device for the managing of the GPIOs:

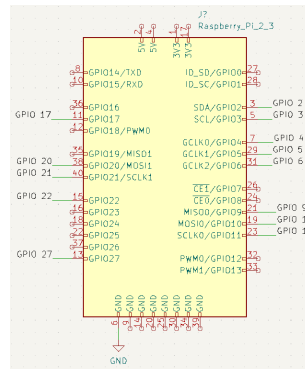


Figure 2.24: 40 PIN header scheme.

FSUSB30MUX

The schematic of the USB switches is represented as follows:

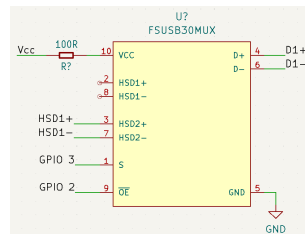


Figure 2.25: FSUSB30MUX electric scheme.

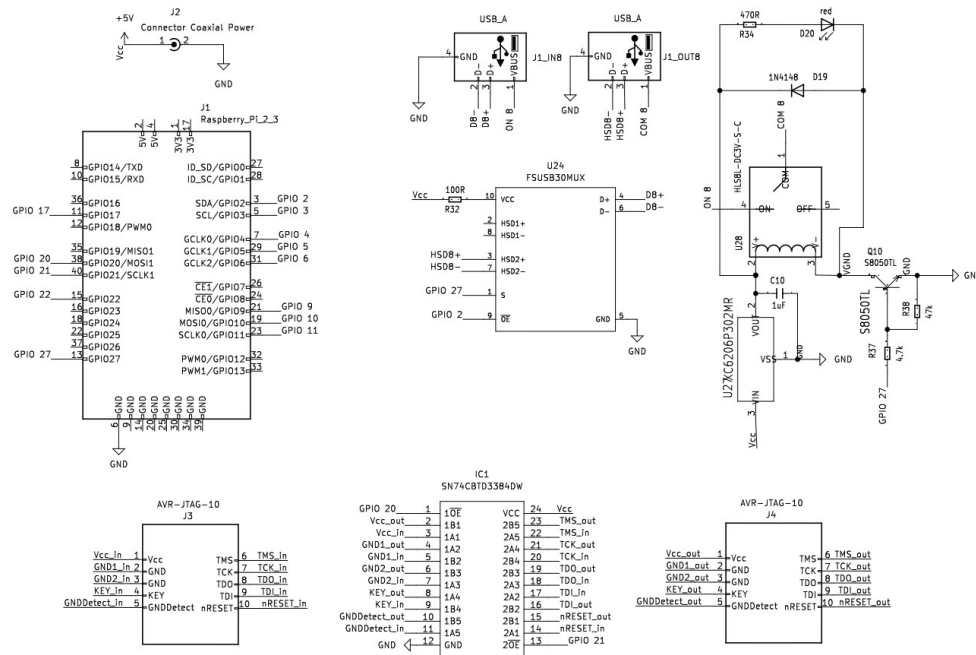


Figure 2.28: Overall Electric Circuit scheme.

Currently, since the electric scheme is correctly built, the *Netlist* is exported and subsequently imported in the PCBnew tool for the creation of the electric routes: this is a essential phase in which every connection was drawn in order to link all components together. For this project, the designed PCB has electric tracks in both bottom and top side:

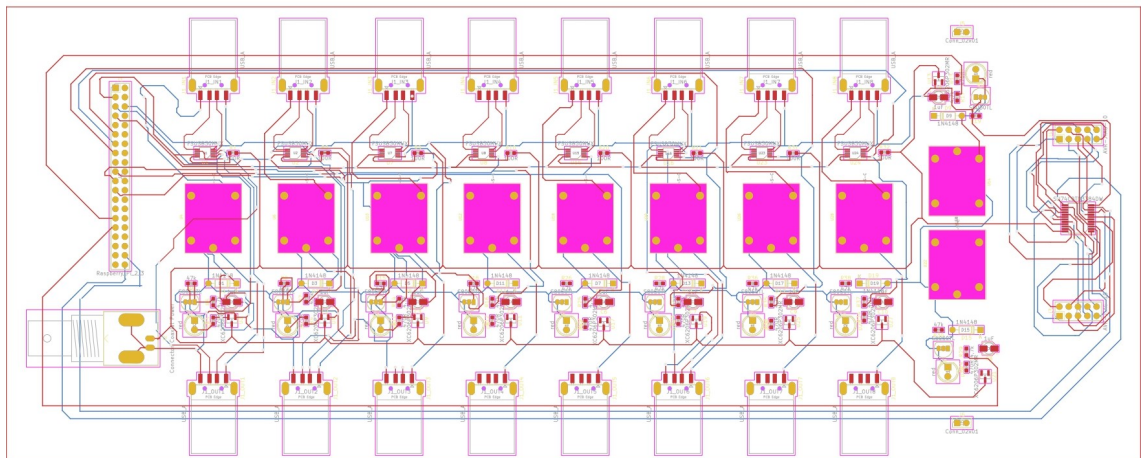


Figure 2.29: PCB electric routing.

Now, through a 3D visualizer, it is possible view the 3D model of the designed PCB:

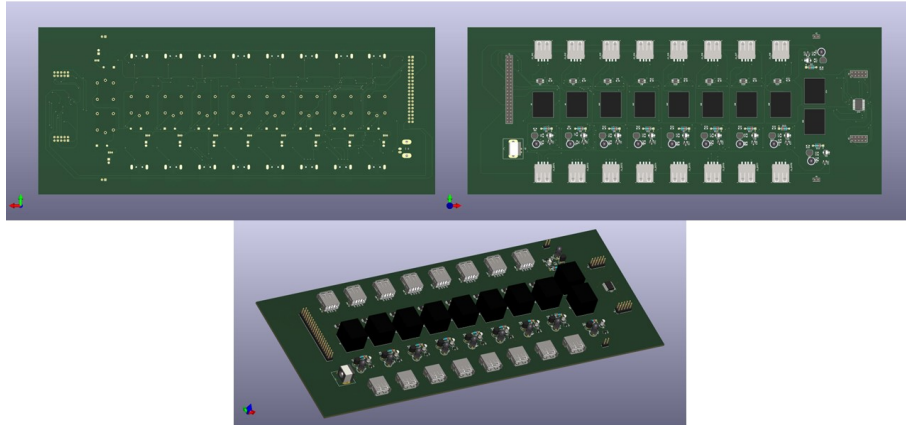


Figure 2.30: Top, Bottom and Side 3D view of the PCB.

Chapter 3

Software design

3.1 What is an Application Web?

Currently, does not exist a unique definition of *web application*, but with this term we mean systems based on client-server communication services. In the past, web services were born with the need of sharing files between users or to send emails, and then, with the growth of concepts such as IoT (Internet of Things), API (Application Programming Interfaces), SOA (Service Oriented Architecture), web services finalities evolved into dynamic structure used for entertainment, information, smart-working , e-commerce and so on. In fact, World Wide Web has modified its own nature: from a static entity done by a set of web pages it is improved into an vigorous architecture that communicates with servers and databases services. To understand better the notion of web application, it has to be compared with a desktop application: the first one can share and distribute data into a multi address environment, while a desktop application is run into the device in which the application is installed, and data are managed in a unique address space. Therefore, early, the web pages shared static documents placed into a server web: the user, who is connected to the web client application, through a suitable interface option, could request the relative document and wait for an answer from server. Then, this methodologies were improved

by involving together programming languages like JavaScript, PHP, Java, JQuery, Ajax and HTML, for both client and server side. Web application can manage a lot of users simultaneously, while the user, in the client side, has not particular software settings to perform: this makes the communication client-server an easy and fast process [4]. Essentially, the changes in terms of server-client communication can be summarized as follows[12]:

- First configuration involves only two computers: one computer-client accesses to the web site and through network can communicate with the second one in which web server is compiled [12].

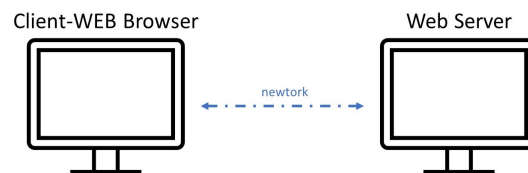


Figure 3.1: Old configuration web structure.

- Second configurations instead deals with several computers: again, the first one is used by the user that is served by the second computer that runs the web server. The last one is connected to N-computers in which various application servers are installed and run in parallel [12].

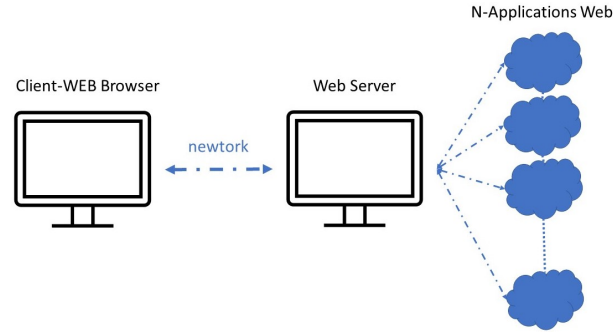


Figure 3.2: Recent configuration web structure.

The differences between the two configurations are clearly evident: from a weak structure you pass into a more secure, reliable and scalable one.

In this project case, an external PC accesses to a browser web page by inserting in the browser URL the IP-address of Raspberry-Pi, in which front-end and back-end of the application web is built. In particular, RPi executes Nginx proxy-server (explained better in the following section), in which client-side part is compiled, and other servers that deals with back-end part.

Front-end

This part is structured with four main files: "*index.html*", for the structure of the web page, "*index.css*", that deals with the style pages, and other three JavaScript files, "*coding.js*", "*buttons.js*" and "*serial.js*", which communicate with server side.

Back-end

The back-end part is made of four servers:

- *OPENApi*: it is an HTTP server for remote managing RPi GPIOs;
- *COMPILER.js*: web-socket server that manages an online compiler for the remote ECU programming;
- *STREAMING.py*: for the Video Streaming;

- *SERIAL.js*: web-socket server for the visualization of Serial Output coming from the ECU.

In other words, the Web Site makes available to the client interface structures that are managed remotely from servers, and that allows to play a lot of functionalities, from the streaming to the online programming. Nginx is only an intermediary element that makes possible the communication between client-side and server-side.

The overall structure can be described as follows:

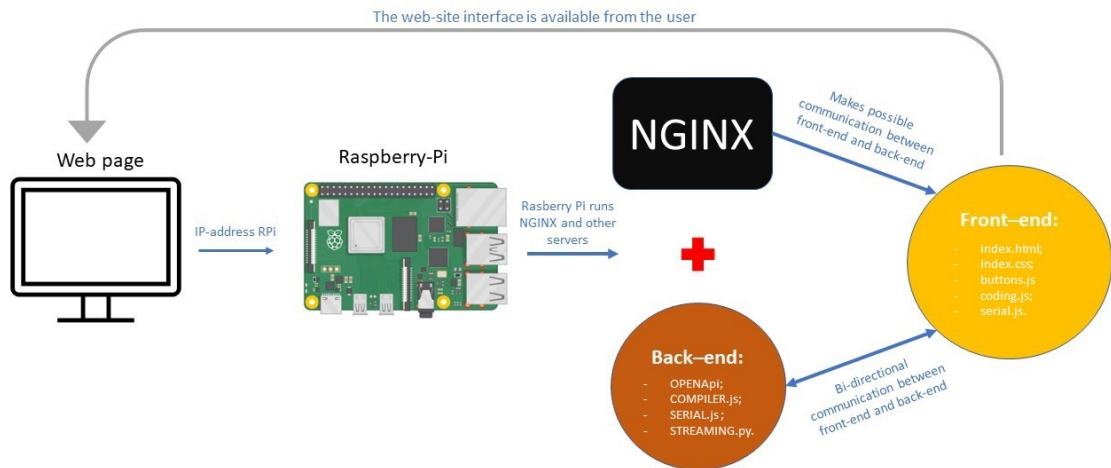


Figure 3.3: Application web structure.

3.2 NGINX



To increase efficiency in communication through internet it is important to evaluate the performances of Web servers. Nowadays, we are searching for criteria such as accuracy, velocity and efficiency in communication services through network world. There exist a lot of web server that have very good performances, like Apache, Ngnix, IIS and Lighttpd: Nginx, used also in this project, is one of the last developed web server which excels in terms of CPU utilization, service rate, memory usage and fast response time [8]. Nginx, that stands for "*Engine-X*", is an open-source web server that behaves like a reverse-proxy server: this means that, when a client requests any content from the web page, NGINX recovers these contents coming from different servers and sends them back to the client as if they came from the same server.

3.2.1 NGINX configuration

The configuration starts with the installation of this web server into the Raspberry device. Through the shell command `"sudo apt-get install nginx"` we begin the installation located into the path `/etc/nginx/` and `/var/www/`. In the first path there are several configuration files, while in the second one the HTML page (web interface), that is located together with the front-end part of the Application web, described in the section **3.4**. At this point, the main file, called `"nginx.conf"`, was modified to allow the communication between client-side and server-side. NGINX manages both master and worker processes: the master ones deal with decodification of the main configuration directives, while the seconds are responsible for managing the client requests. In particular, NGINX makes available to the client four upstream

servers through port 80 and the configuration includes the following code:

```
server {  
    listen 80;  
    server_name localhost;  
    root /var/www;  
    gzip_static always;  
    index index.html;  
    include nginx_locations.conf;  
}
```

This allows to access to the page web, served by the file *"index.html"* located into the path */var/www/*, by inserting on the browser URL the following line:

```
http://localhost:80/
```

At this point, the four servers are accessible, by defining on the configuration file *"nginx_http.conf"* four upstream servers (term upstream means that NGINX acts as balance loader between the servers):

```
upstream smarthub {  
    server 127.0.0.1:8080; # OPENApi server for GPIOs  
}  
upstream smarthub-compiler{  
    server 127.0.0.1:9001; # Online Compiler  
}  
upstream smarthub-serial{  
    server 127.0.0.1:9002; # Serial ECU output  
}  
upstream smarthub-streaming{  
    server 127.0.0.1:9003; # Streaming Video  
}
```

As it can be seen from the code above, each server communicates through different ports (8080, 9001, 9002 and 9003) and NGINX serves to the client all of these with an unique port, the 80 precisely. In the end, another important configuration file, located in the */etc/nginx/* path and called *"nginx_locations.conf"*, is considered. In this file are defined the paths which address the client requests to the right servers. This file is characterized by the following code:

```
location /v2/ {
    proxy_pass http://smarthub/v2/;
}

location /smarthub-compiler {
    proxy_pass http://smarthub-compiler;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection $connection_upgrade;
}

location /smarthub-serial {
    proxy_pass http://smarthub-serial;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection $connection_upgrade;
}

location /smarthub-streaming{
    proxy_pass http://smarthub-streaming
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection $connection_upgrade;
}
```

In particular, the location function defines, after the symbol `"/"`, the path of requests linked to relative servers, initialized with the upstream command. Hence, to make an example, when the client uses the online compiler (described in the following section), NGINX directs the requests towards the server called *smarthub-websocket*, and the same happens for the other servers.

3.3 Back-end

With **Back-end**, that deals with the server-side, we mean all the operations executed behind any application web that the client does not visualize, but with which he can interact. Many programming languages can be used to develop the back-end such as HTML, Python, SQL (in case of database development), JavaScript, nodejs (JavaScript runtime) and other more. For this specific project, the server-side handle the remote programming of the ECU. It must be capable of:

- manage GPIOs remotely (switch on/off devices described in the last Chapter) with suitable buttons. This is done by developing with Python language the what is called *"OPENApi"* HTTP server;
- carry out the real programming phase through the Programmer using the shell of the Raspberry-Pi: for this purpose a web-socket server, written in JavaScript, called *"COMPILER.js"* is built;
- visualize through a suitable area in the client side the Serial Output coming from the ECU: another web-socket server, called this time *"SERIAL.js"*, is developed;
- show through Streaming video the correct functionality of the ECU and other devices, with the implementation of a python web-socket server named *"STREAMING.py"*;

3.3.1 OPEN-API

Before enter into details on the building of the server, it is important to give the definition of what is called **API** (Application Programming Interfaces), namely a set of tools that consent to access to resources or data from a server: it behaves like a "courier" which mediates between request and response. For this project we will deals precisely with *RESTful* API, a particular type of API that returns resources and relative responses through HTTP protocol in JSON format, one of the most popular programming languages. More specifically, the client makes requests through the browser using URI and HTTP methods: the main are GET (get data from server), PUT (modify data into server), POST (send data to server) and DELETE (cancel data from server). The RESTful API acts as an intermediary between client and server and gives back to client a response and the relative resource [10]. But, what is OPENApi? OPENApi is a documentation for managing RESTful API services. Also called Swagger, it is an open-source tool that creates documentation for the API based on HTTP, using the YAML (used for this project) and JSON languages. Hence, trough Swagger.io site, a online compiler for API documentation, the yaml code, called "*smarthub.yaml*", is compiled. In this file program, are defined the resources and methods with which we want to access the data. In particular the following resources and methods were defined:

- The information of the USBs, JTAG, POWER SUPPLY and BOOT status (ON or OFF) through the GET method;
- Changing the USBs, JTAG, POWER SUPPLY and BOOT status through PUT method;
- The information of the ports used for the Serial output, using the GET method;

This means that the client, using the URL browser, can access to the resource and perform operations in order to know or to change remotely the

status of the USBs, the BOOT, the POWER SUPPLY, the JTAG and the serial ports, used for the Programming. The next image shows the visualization of the API:

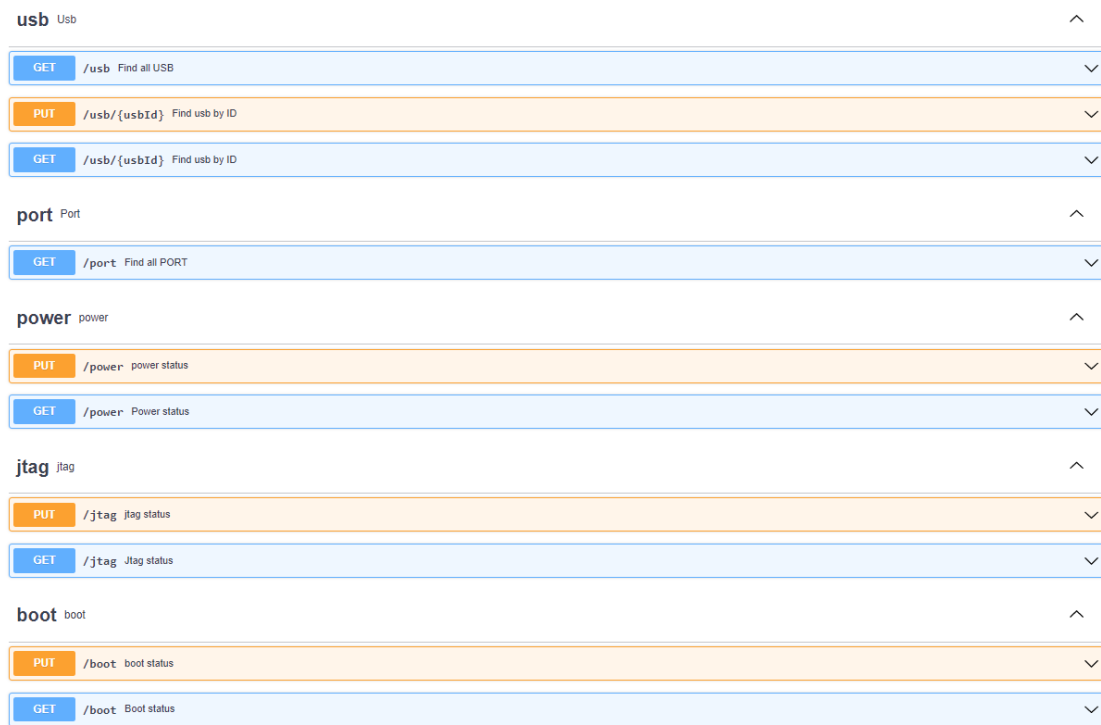


Figure 3.4: API visualization through Swagger Editor.

At this point we can build the OPENApi server that manages the API: through command prompt on Windows 10 we generate the server starting from the *smarthub.yaml* file by installing the what is called *OpenApi Generator* and subsequently by running the following command line:

```
npx @openapitools/openapi-generator-cli generate -i
smarthub.yaml -g python-flask -o D:\SmartHub
```

In a folder, called *SmartHub*, is created the source code of the OPENApi server compiled in Python. The main Python files are located into the folder *SmartHub/openapi_server/controllers* and are responsible for man-

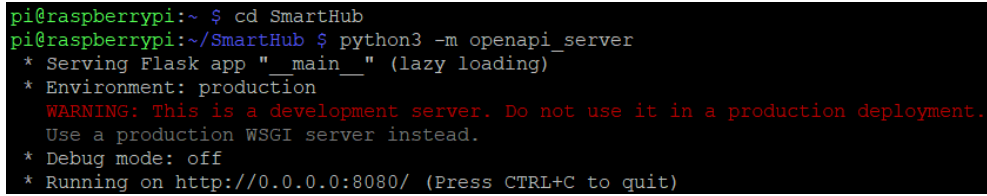
aging GPIOs connected to the external module described in the previous chapter:

- **usb_controller.py**: this file is responsible for managing GPIOs in which are connected electronic components for switching on/off the external USB modules. It returns the value (HIGH or LOW) of the pin and consequently the status of the relative USB;
- **jtag_controller.py**: this file pilots pin in which the JTAG module is connected;
- **boot_controller.py**: used for the BOOT of the ECU;
- **power_controller.py**: controls and manages the POWER SUPPLY of the ECU;
- **ports_controller.py**: returns the actual port in which an external serial port is connected.

Once the server is generated and its main code files are modified, the overall folder *SmartHub* is imported into Raspberry Pi device and it is run by writing on the shell the following commands

```
python3 -m openapi_server
```

The server is running on the 8080 port, but is served from NGINX to the client on port 80. Moreover, it is an HTTP static server: this means that it sends responses to the client if and only if the client makes a request; in the other cases the server remains in *"wait"* mode. To test the functionalities of this server the **RESTMAN** tool (an open-source online tool) is used: it is equipped with a suitable interface in which you can see the responses from the server and you can select the methods (GET or PUT in this case) with which make requests to the server.



```
pi@raspberrypi:~ $ cd SmartHub
pi@raspberrypi:~/SmartHub $ python3 -m openapi_server
* Serving Flask app "__main__" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://0.0.0.0:8080/ (Press CTRL+C to quit)
```

Figure 3.5: OPENApi Server shell output.

3.3.2 Node.js

Usually the development of a Web based application is split between two different professional figures: Front-end developer, who deals with HTML, CSS and JavaScript programming languages for the implementation of the web site design and the relative style and visual effects, and Back-end developer, which creates the algorithm of the application by using PHP, JAVA, JavaScript and eventually, in case of databases presence, SQL. Most of cases, both of the developers must have knowledge on both of back-end and front-end parts: in this case we speak about Full Stack developers. Node.js plays an important role for this purpose: it solves the problem related on the knowledge of different languages for Full Stack development about the implementation of an application in both server and client sides. In fact, one of the advantages of using Node.js, is that it require the only knowledge of JavaScript: the developer does not have to switch between multiple languages while it is developing client and server side of an application web. Node.js has many application fields, between which SPA (*Single Page Applications*), NodeOS (JavaScript based Operating System) and IoT (*Internet of Things*). The strong point of Node.js is that it is suitable for development of applications for embedded system, as in this case, since it makes available asynchronous and event-driven functions [17]. For this project two Node.js based web-socket servers were created.

COMPILER.js

This server is responsible for the management of the online compiler. Being a web-socket server, it remains always in "open" mode on port 9001: it receives as input from the client a script specially created for the ECU programming. This script language is made of several commands:

- **"USB num_usb mode_usb"**: "num_usb" is replaced with a number which identifies the USB that we want to switch and "mode_usb" is replaced with ON (or OFF);
- **"wait seconds"**: "second" is replaced with a number related to the interval time between one command and the next one;
- **"jtag mode_jtag"**: for switch on/off the JTAG module;
- **"power mode_power"**: for switch on/off the Power Supply of the ECU;
- **"boot mode_boot"**: for switch on/off the Boot of the ECU;
- **"from cmd:"** : this command allows the client to directly write a series of commands executed on the Raspberry Pi shell. In fact, the COMPILER.js takes as input this command and returns a response in term of what is called *"stdout"* and *"stderr"*, standard channels in which the operating system warns if the command was successful or if there is an error.

Once the script message is received from the client, the server perform the operations (get, put, wait or write on shell) specified from the script itself, and after the operation is ended, the server send back to the client a message: this message could be the output of the operation in case of successful, or an error message in case of failed operation. The output generated after the shell command *"node COMPILER.js"* is the following:

```
pi@raspberrypi:~ $ cd SmartHubCompiler
pi@raspberrypi:~/SmartHubCompiler $ node COMPILER.js
The Online Compiler server is running on port 9001
```

Figure 3.6: COMPILER.js Server shell output.

SERIAL.js

This server is a web-socket server always open on port 9002. It sends to the client, in a suitable textarea, the output of the serial communication coming from the ECU. In particular, the client, from the web interface, selects the port number in which the serial USB is connected and the baud rate (transmission speed) of the data coming from the serial of the ECU (usually 115200 bps is chosen); then it clicks on a specific button for the visualization of the output. Through the shell command `"node SERIAL.js"` the following output is generated:

```
pi@raspberrypi:~ $ cd SmartHubSerial
pi@raspberrypi:~/SmartHubSerial $ node SERIAL.js
The Serial Terminal server is running on port 9002
```

Figure 3.7: SERIAL.js Server shell output.

3.3.3 STREAMING.py

This is the last server, used for the Streaming Video, useful to check remotely the correct functionality of the entire system. Also in this case we speak about web-socket server, served on port 9003. Compiled in Python, it uses packages for the management of the PiCamera module connected to the Raspberry Pi: through a parameter in which is defined the IP address of the Raspberry Pi that makes available the HTML page, it creates a `"stream.mjpeg"` file that contains the output of the video recorded by the camera, with a certain frame rate defined in the code (30 fps is chosen). This mjpeg file is called back into the HTML page through the code for the streaming:

By executing the shell command *"python3 STREAMING.py"*, the result is the following:

```
pi@raspberrypi:~ $ cd SmartHubStreaming
pi@raspberrypi:~/SmartHubStreaming $ python3 STREAMING.py
Warning! No module named 'sounddevice'
Warning! No module named 'matplotlib'
Warning! No module named 'keras'
Server Started at http://169.254.229.242:9003
```

Figure 3.8: STREAMING.py Server shell output.

3.4 Front-end

The term "*Front-end*" stands for any graphical interface with which the client, through browser, can interact to perform any operation, from sharing data to requesting resources from servers or from databases. Any application web has its own interface, whose functionality depends on the purpose for which the application is implemented. Nowadays, is grown more and more the need of a well-organized graphical interface, in order to make the client experience pleasant and intuitive. In this regard, usually the web page is composed by a multiple file codes, written in different programming languages. In particular, with HTML5 language (the new updated version of HTML) the structure of the page is built, and with CSS the style elements are implemented into the structure. In the end, other files in other languages, such as PHP or JavaScript, are used in order to add graphical effects or to perform communication between client-side and server-side. In this project, the Front-end development is made of the following files:

- *"index.html"*;
- *"index.css"*;
- *"buttons.js"*;
- *"coding.js"*;
- *"serial.js"*.

This files are located into the Rapsberry Pi device in the path */var/www/*: this folder is served by NGINX to make accessible the interface to the client. The following part gives a more detailed description of the documents mentioned before, in order to make clarification on how the interface (client-side) is connected to the server-side.

3.4.1 index.html

This file, written on HTML5 language, contains the overall structure of the web interface and some JavaScript function. In particular, the structure contains:

- **Navigation bar:** it is always fixed in the top of the page through a JavaScript function called *"fixnavbar"* and it is used to access to a specific section.

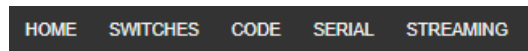


Figure 3.9: Navigation Bar of the Web site.

- **Home section:** this section contains the general information about the project, explaining the content of the web page.

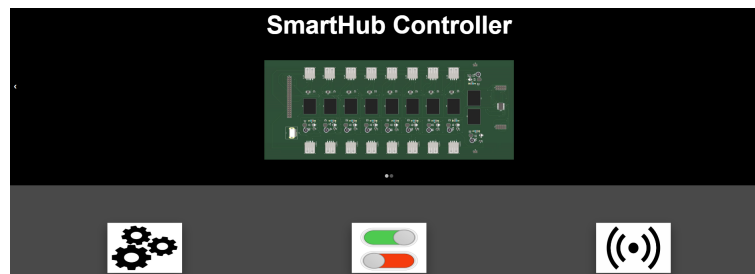


Figure 3.10: Home section of the Web site.

- **Buttons section:** contains all the buttons relatives to all the modules (Programmer, USB Pen Drives, Serial USBs, Power Supply, Boot, Jtag) to switch on/off. Each button, marked with a what is called *checkbox*, is identified by an id and is associated with a function, called SwitchUSB, that performs PUT requests to change each USB status and consequentially the relative button position.

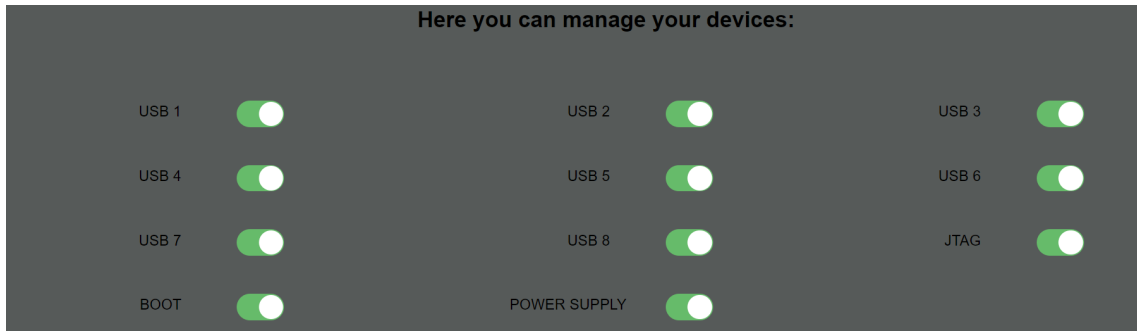


Figure 3.11: Buttons section of the Web site.

- **Compiler section:** contains two *textarea* elements. One of this textarea is dedicated to writing a script to be sent to the COMPILER.js server via a button marked with the word "COMPILE"; the other one textarea shows the output of the responses send back from the server.

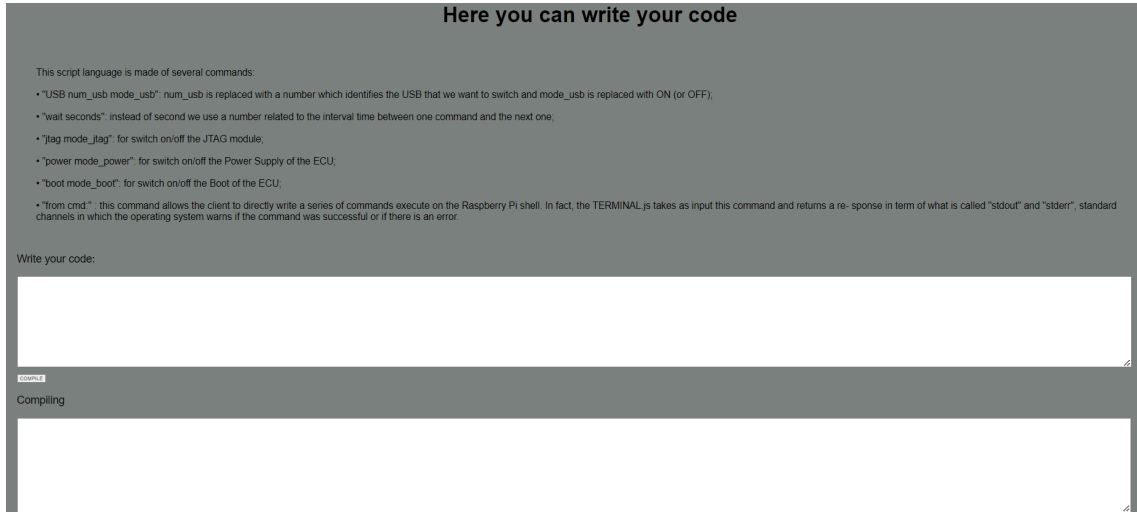


Figure 3.12: Compiler section of the Web site.

- **Serial section:** this section is dedicated to the output of the Serial communication coming from the ECU. In particular, there are five elements: two list elements in which you can choose the actual active

port for the serial communication and the baud rate speed; one button element, marked with "VIEW", that sends the data selected from the list elements to the SERIAL.js server; a textarea element for the visualization of the serial output; and another button, marked with the word "ADD", which allows to duplicate the previous elements to visualize different outputs coming from other ports and baud rates.

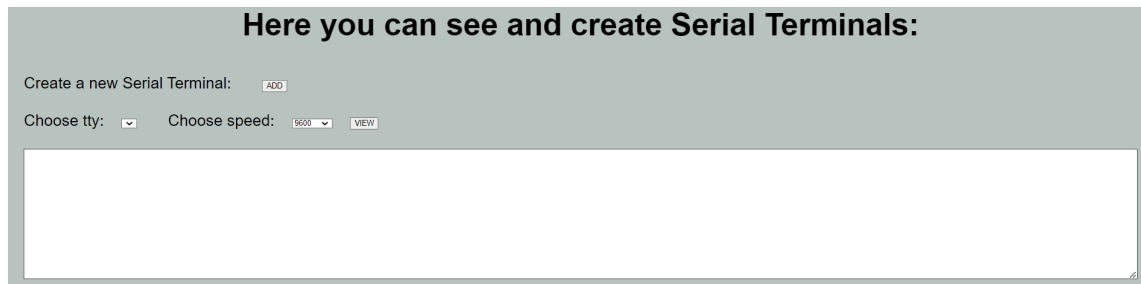


Figure 3.13: Serial section of the Web site.

- **Footer:** is the last section of the web page, located in the bottom side, in which are present personal information and contact and web site of the company in which the project was performed.



Figure 3.14: Footer of the Web site.

3.4.2 index.css

This is a CSS file, called in the index.html file, responsible for the style of the elements initialized on the HTML file: for example, the colors of the buttons at different positions, background of various sections, style and

color of the navigation bar and of the footer, position and width of the elements inside of the section, and so on.

3.4.3 button.js

This file contains JavaScript functions for the HTTP requests to the OPENApi server. These requests are executed in two ways: the GET request is done at the load of the web page, so that, every time you access the page, you have information about the actual status of the buttons, and therefore of the modules; instead the PUT request are performed by pressing the buttons, changing the status of the modules. Without reporting the content of all the functions, on the following code example the general structure is explained, in order to better understand how the client is connected to the server:

```
function (){  
    $.ajax({  
        method: "GET"  
        url: 'http://' + location.host + ':80/v2/jtag',  
        success: function(data, code, status) {  
            error: function() {console.log(arguments)},  
        })  
    };  
};
```

As it can be noticed from the code above, this function calls an AJAX (Asynchronous JavaScript and XML) function dedicated to the management of the requests coming from the client, forwarding them to the server. In particular, it specifies the methods with which make the requests and the URL in which write the IP address of the Raspberry Pi followed by the name of modules of which you want to modify or know the status. These modules are exactly the API previously defined. At this point, on the base

of the values present in the variable *"data"* (1 if ON or 0 if OFF), the position of the button is changed in the correct position.

3.4.4 coding.js

This file is characterized by JavaScript functions that interfaces the client with the COMPILER.js server (Online Compiler). In particular, it takes the script sentences written in the textarea by the client and, before send them to the server, it checks if there are errors on the script with the usage of a flag. When no error occurs, every sentence is forwarded one at a time to the server: the next operation can be performed only if previous operation, executed on the server-side, returns a successful output.

3.4.5 serial.js

This is a JavaScript file that interacts with the SERIAL.js server for the serial communication output. It takes the port and baud rate information from the list elements and saves them into a string variable called *"command"* to send to the server: at this point the server takes this string and creates a serial port variable in which are present serial data coming from the ECU. In particular, in the server side:

```
var PORT = command[0];
var BAUDRATE = Number(command[1]);
var port = new SerialPort({
    path: "/dev/"+PORT,
    baudRate: BAUDRATE,
});
```

The data saved in the *port* variable are forwarded to the client and visualized in a suitable textarea.

Chapter 4

Validation Test

The last phase of this project is related to the test of the entire system: the aim is to verify if the external components are readable by the Raspberry Pi through the PCB and visualize in the browser, in which we accessed from an external PC, the output of some programming operations done on the ECU. Before enter into the details on the setup and connection between ECU and Raspberry Pi device, it is important to analyze the TEST case performed on the ECU. The ECU under consideration is a Magneti Marelli prototype entrusted to the Abinsula company for what is called "*Smoke Test*" or "*Build Verification Testing*": these are software tests that allows to verify if some basic functions on the ECU software work properly. This particular test is performed during the Validation phase: the first step is the Pre-Integration test, in which the Development Team checks if the ECU software is developed according to the SW Requirements and Standards, the second step instead is the Integration Test, in which Integration Team performs Automatic Integration Tests, Smoke tests (checks of basic software functions such as installation and Power-on) and Performance Tests. The following image shows the System Integration Loop of the ECU.

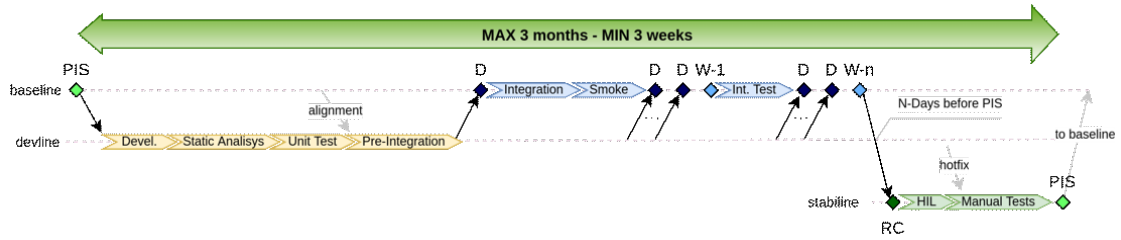


Figure 4.1: System Integration Loop.

4.0.1 Test case

We connected the ECU to the Raspberry Pi through the PCB and we checked all the functionality executing some operations using the Application web. In particular, the following connections were performed:

- The ECU is feed by a 12 V Power Supply through an ODB cable, switched on/off by one Relay-Groove mounted on the PCB;
- Through a ARM-JTAG cable, we connected across the BUS-SWITCH the ECU to the Programmer, in turn linked to the Raspberry Pi device. In this case we can switch on/off both the JTAG and the Programmer (USB module);
- The ECU was linked to the Raspberry Pi device using one Serial USB, one USB2CAN cable and USB2ETH cable;
- One Pen Drive module is connected to RPi in order to install a *build* into the ECU software.

Once the connections were performed, with an external PC we logged into the web page by writing in the URL the Raspberry Pi IP address. The test consisted on writing on the Online Compiler a set of commands in order to perform some programming operations, and then to verify if the USB modules, that communicates with different communication protocols, were read correctly passing trough the electronic components installed in the PCB. Let's see an example of script:


```

power off
jtag on
usb programmer on
from cmd: /rfp-cli -d RH850 -t e2 -osc 16.0 -auth id
          FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF -a
          /media/data/dut_bootldr_app.hex

```

With this script command we performed a SW operation that allows to take from the Pen Drive a Bootloader program (that loads into the ECU processor the data of the operating system from secondary memory) using a tool of the E2 Programmer: two types of outputs were generated, one coming from the Programmer itself and one coming from the Serial communication, visible in the Serial Terminal of the web page. The output was the following:

```

Renesas Flash Programmer CLI V1.03
Module Version: V3.09.01.000
Connecting the tool (E2 emulator)
Tool: E2 emulator (OLS005082D)
Interface: 2 wire UART
Tool Firmware Version: V1.02.00.001
Emulator power supply: OFF
Connecting the target device
Enter Main clock [MHz] (8.0 - 24.0)? 16.0
Main Clock: 16 MHz
Speed: 500,000 bps
Enter ID Code (16 Bytes)? FFFFFFFFFFFFFFFFFFFFFFFFFF
Connected to R7F701602
Reading data from the device...

```

This output gives information about the tool used, the clock frequency and the ECU model showing through the Serial Port some basic instructions

installed in the ECU SW. This means that the Programmer and the ECU were correctly read from the Raspberry Pi device, passing through PCB components. Some different script codes were performed, observing that other communications protocols were readable crossing the PCB.

The results obtained in the previous tests are compared with the same operations performed using an external PC equipped with Ubuntu Operating System, to which we connected the ECU in the same way done for the Raspberry Pi. The outputs obtained after the shell commands were the same for both the PC and the Raspberry Pi device. The only difference between the two devices was that the RPi requires more time to print the outputs. This delays are caused not only by the difference in RAM and in speed of the processors computation, but also by the components installed into the PCB device: infact, the FSUSB30MUXes interface with external devices at 480 Mbit/s (typical speed of USB 2.0), unlike the PC that possesses USB 3.0 ports which manage data at 5Gbit/s. The same holds also for the JTAG cable that links the ECU to the Programmer through the 10-Bit BUS switch, that seems to slow down the stream of data: we noticed that between the input and output ports of the BUS Switch, the Multimeter revealed a resistance between 400 m Ω and 500 m Ω .

Chapter 5

Conclusions

In this thesis is discussed a design description of a Raspberry-Pi shield used to perform programming operations on the ECU, from both Hardware and Software (web-based) point of view. The design goes from the choice of different electronic components until the compilation of an Application Web. During the HW design process, various HIL tests were performed, with which the correct functionality of all the components chosen was verified, before going on into the SW design. After that, the Application web was build, first compiling the back-end part with NGINX, and then writing the Front-end part using HTML5 and Javascript languages. The last phase of the design process was to test the overall system by connecting it to an ECU. The results obtained led to improve the quality of programming an ECU, decreasing the time required to perform connections and disconnections between the ECU and various devices. Thanks to the possibility of a remote control, problems related to human errors or the risk of ruin contacts and cables are avoided. However, this component, as it has been built so far, still has limitations, partially due to Raspberry-Pi performances and partially due to the electronic components used. A possible improvement could be the choice of different components which constitute the PCB, for managing data stream with a faster speed or for managing the programming of multiple ECUs at the same time. Another improvement could be the choice of a

different model of Raspberry-Pi device, or directly the employment of an *embedded* PC. Obviously, these updates lead to HW and SW modifications, and also to higher costs.

Nomenclature

API Application Programming Interface

CPU Central Processing Unit

DPDT Double-Pole-Double-Throw

DUT Device Under Test

ECU Engine Control Unit

GPIO General Purpose Input Output

GPU Graphics Processing Unit

HIL Hardware-In-the-Loop

HW Hardware

IC Integrated Circuit

ODB On-Board Diagnostics

OE Output Enable

OEM Original Equipment Manufacturers

OS Operating System

PCB Printed Circuit Board

RPi Raspberry-Pi

SBC Single Board Computer

SW Software

URI Uniform resource identifier

URL Uniform Resource Locator

List of Figures

2.1	Structure of the system.	15
2.2	Raspberry Pi.	16
2.3	Raspberry Pi 3 model B+ [2].	18
2.4	RaspberryPi Imager [15].	18
2.5	FSUSB30MUX [13].	20
2.6	Scheme of FSUSB30MUX [13].	21
2.7	Truth Table of FSUSB30MUX [13].	22
2.8	Physical dimension of FSUSB30MUX [13].	22
2.9	MSOP10 Package [1].	23
2.10	Relay Grove v1.2 [16].	23
2.11	SN74CBDT3384 [5].	24
2.12	Pin description of 10 BIT BUS SWITCH [5].	24
2.13	TSSOP 24 [3].	25
2.14	LOCAL HIL test.	26
2.15	Connection between FSUSB30MUX and Relay-Groove. . . .	27
2.16	Connections of two Relay-Grooves and 10-BIT Bus Switch. .	28
2.17	GPIO description [18].	29
2.18	GLOBAL HIL test.	29
2.19	PCB prototype connections.	30
2.20	Real PCB prototype.	30
2.21	KiCad work-flow flowchart [6].	31
2.22	Coaxial Power connector electric scheme.	32
2.23	10-Bit Bus Switch electric scheme.	33

2.24	40 PIN header scheme.	33
2.25	FSUSB30MUX electric scheme.	33
2.26	Female USB electric scheme.	34
2.27	Relay-Groove electric scheme.	34
2.28	Overall Electric Circuit scheme.	35
2.29	PCB electric routing.	35
2.30	Top, Bottom and Side 3D view of the PCB.	36
3.1	Old configuration web structure.	38
3.2	Recent configuration web structure.	39
3.3	Application web structure.	40
3.4	API visualization through Swagger Editor.	46
3.5	OPENApi Server shell output.	48
3.6	COMPILER.js Server shell output.	50
3.7	SERIAL.js Server shell output.	50
3.8	STREAMING.py Server shell output.	51
3.9	Navigation Bar of the Web site.	53
3.10	Home section of the Web site.	53
3.11	Buttons section of the Web site.	54
3.12	Compiler section of the Web site.	54
3.13	Serial section of the Web site.	55
3.14	Footer of the Web site.	55
4.1	System Integration Loop.	60

Bibliography

- [1] Lcqt-msop10. <https://www.digikey.it/it/products/detail/aries-electronics/LCQT-MSOP10/4754589>.
- [2] Raspberry pi 3 model b+. <https://static.raspberrypi.org/files/product-briefs/Raspberry-Pi-Model-Bplus-Product-Brief.pdf>.
- [3] Ic adapter soic24 / tssop24 to dip24 pitch 15.24mm ups-so24. <https://www.gmelectronic.com/ups-so-24-tssop-24-dil24>.
- [4] Nalaka Ruwan Dissanayake and Kapila Asanga Dias. *Web-based Applications: Extending the General Perspective of the Service of Web*. August 2017.
- [5] Texas Instruments, editor. *SN74CBT3384A 10-BIT FET BUS SWITCH*. https://www.ti.com/lit/ds/symlink/sn74cbt3384a.pdf?HQS=dis-mous-null-mousermode-dsf-pf-null-ww&ts=1656080808744&ref_url=https.
- [6] David Jahshan, Phil Hutchinson, Fabrizio Tappero, Christina Jarron, and Melroy van den Berg. *Getting Started in KiCad*, May 2015.
- [7] Steven J Johnston and Simon J Cox. *The Raspberry Pi: A Technology Disrupter, and the Enabler of Dreams*. July 2017.
- [8] Douglas Kunda, Sipiwe Chihana, and Sinyinda Muwanei. *Web Server Performance of Apache and Nginx: A Systematic Literature Review*. November 2017.

- [9] Syed Nabi, Mahesh Balike, and Jace Allen. *An Overview of Hardware-In-the-Loop Testing Systems at Visteon*. March 2004.
- [10] Andy Neumann, Nuno Laranjeiro, and Sarang Noether. *An Analysis of Public REST Web Service APIs*. July 2021.
- [11] P.C. Nissimagoudar, Venkatesh Mane, Gireesha H M, and Nalini C. Iyer. *Hardware-in-the-loop (HIL) Simulation Technique for an Automotive Electronics Course*. 2019.
- [12] Jeff Offutt. *Web Software Applications Quality Attributes*.
- [13] onsemi / Fairchild, editor. *FSUSB30 Low-Power, Two-Port, High-Speed USB 2.0 (480 Mbps) Switch*. https://www.mouser.it/datasheet/2/308/1/FSUSB30_D-2314152.pdf.
- [14] Francisco Perdigones and José Manuel Quero. *Printed Circuit Boards: The Layers' Functions for Electronic and Biomedical Engineering*. March 2022.
- [15] Emma Roth. The raspberry pi image flasher receives an important update, MAR 2021. <https://www.makeuseof.com/raspberry-pi-image-flasher-receives-important-update/>.
- [16] seeedstudio. Grove - relay. <https://wiki.seeedstudio.com/Grove-Relay/>.
- [17] Hezbullah Shah and Tariq Rahim Soomro. *Node.js Challenges in Implementation*. May 2017.
- [18] TerryWarwick, Rsa Meser, Matt Wojo, and Sara Clay. Mapping dei pin raspberry pi 2 3. <https://docs.microsoft.com/it-it/windows/iot-core/learn-about-hardware/pinmappings/pinmappingsrpi>.
- [19] Nikola Zlatanov. *PCB Design Process and Fabrication Challenges*. September 2012.

