ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCHOOL OF ENGINEERING AND ARCHITECTURE

DEPARTMENT of
ELECTRICAL, ELECTRONIC AND INFORMATION ENGINEERING
"Guglielmo Marconi"
DEI

MASTER'S DEGREE IN
Advanced Automotive Electronic Engineering

DEGREE THESIS
in
Hardware-software design of embedded systems m i.c.

An Embedded Data Logger for In-Vehicle Testing

CANDIDATE

Manuel Cintura

SUPERVISOR

Chiar.mo Prof. Francesco Conti

CO-SUPERVISORS

Chiar.mo Prof. Manuele Rusci

HOST COMPANY SUPERVISOR

Chiar.ma Paola Conti

Academic Year
2020/2021

Session
I

INDEX

1. INTRODUCTION

Lightweight, small, and smart electrification is becoming increasingly popular in the automobile industry. As compared to other markets in the automotive ecosystem, this has accelerated the growth of the automotive electronics industry. Automobile makers have committed to putting more effort into improving the performance of their automobiles. OEMs (Original Equipment Manufacturers) are working hard to ensure that vehicles meet the most stringent regulatory standards. As a result, automakers are adding more technological components in their automobiles. Electronic providers have risen to prominence in the automotive market for a multitude of scenarios, including ADAS and safety, electric vehicles, connected vehicles, and autonomous cars, which are among the fastest-growing technology. With the potential for these automotive trends to expand, automotive data loggers are estimated to expand quickly, as the rising electrical complexity in automobiles demands higher levels of vehicle testing, diagnosis, and maintenance. Automotive OEMs prefer on-road testing to lab testing because it allows them to better understand the real-world scenario of the performance of various components in their cars and also see how the vehicle performs in a real-world situation. However, if the instrument employed does not get a signal from the car network, a vehicle diagnostic becomes impossible. This is known as non-communication, and it can happen for a variety of reasons, such as when the OBD-II (second generation On-board diagnostics) port is broken. It is possible that system installers deliberately or unwittingly tamper the wiring harness of the OBD-II connector, thus hindering communication between the testing tool and OBD.

In general, a **data logger is an electrical device that uses a built-in or external instrument and sensor to record data over time or in relation to location**. They are typically tiny and portable, having a CPU, internal memory for data storage, and a variety of sensors. Some data recorders connect to a computer through software to activate the data logger and display and analyse the gathered data, while others can

operate autonomously. Data loggers range from general-purpose devices that may be used for a wide range of measurement applications to highly specialised devices that can only be used in one environment or application type. General-purpose types are frequently programmable. Data loggers are generally placed and left unattended for the duration of the monitoring period after they have been activated. This provides a complete and accurate picture of the observed environmental conditions. As technology advances and costs decrease, the cost of data loggers has decreased over time. Simple single-channel data recorders are inexpensive, but more complex ones can cost hundreds or thousands of dollars.

The goal of the thesis is the realization of a custom prototype of a data logger which fulfils the requirements and implements functionalities necessary for the company. The company is Maserati S.p.A. and the main goal is to build a data logger that can be used during daily tests on vehicles. The main activities done to achieve the final result expected for Maserati company are:

- Analyze requirements and documentations that were provided by the company. They include manuals for the development environment and documents regarding the structure of the programming language.
- Implementation of the Software modules targeting a Raspberry Pi in C++ language.
- Realize the data logger prototype for in-vehicle experiments, including the wiring of the prototype.
- Make the connections between wires and board.
- Test the final prototype board on the vehicle.

The device I will talk about later, in the next chapters, differs from the others currently on the market. Compared to other devices available on the market, in this project we have developed and implemented some special features which, combined together, make our data logger unique. What makes this device exclusive is the possibility for monitoring the ADAS and radio features with two videos and, at the same time, sniff the CAN bus and serial port to retrieve data with synchronized timestamps. These are

features that other data logger do not have and is an added value for tests that are carried out daily. As an example, until now the engineers who work in the company have always taken the logs bringing in the car at least one laptop, one CANcase and the related cables and above all they needed a second person to help them during the tests. Because, while the driver is driving the vehicle, the co-pilot has to run a tool inside its pc and coordinating with the other person. Leave in the part about the technical aspects, an important advantage that comes from our device is the on-board space reduction during tests since the data logger can be mounted or place in a hide position that does not interfere with the normal behaviour of the vehicle. Because the prototype of the logger has the dimension of a Raspberry Pi and once all peripherals are properly connected, when an issue is detected by the driver tester, he only should press a HW button. Then correlated SW trigger event is detected by the device and all logs (Serial, CAN) will be stored with the corresponding video records in the SD Card or inside in a common pendrive. Since this button is placed close to him, he can do this action according to its needs and timing. All logger device settings shall be decided and set before to put it inside the vehicle under test. In this way the data logger can be used independently on the screen, laptop or other external hardware.

The next Chapter 2 gives a general view of the whole project, focusing on the team management, the design of the device taking account of the requirements and the project tasks divided by stages.

2. PROJECT OVERVIEW

In this chapter I will explain, in general terms, the purpose of this project and I will give an overview of the main aspects concerning the realization of the data logger.

2.1. General purpose

The device is designed to work as a logger for in-vehicle testing and to help the driver tester to record all useful logs for his activities. It can be also used to record the user experience as well. Since it has different type of peripheral and since they can be set according to multiple configurations, the data logger must be flexible for many purposes, or it can be reused and reconfigured to be adapted to new requirements or activities.

2.2. Requirements & Functionalities

The device must respect some requirements and it must have some functionalities. So, it is designed to fulfil all of them specifics. First of all, it is required that it supports simultaneous recording with two USB Webcams at 20fps with a minimum resolution of 480x640. This is needed because, otherwise, it will be very difficult to analyse the output videos to detect malfunctions, bugs and problems on the radio and infotainment or to monitor the correct behaviour of ADAS systems. Another important aspect to consider is the setup of the CAN-bus because it required to work at a specific speed, phase-segment and sample-point according to the data given by the datasheet. At the end the serial port, which is a connector by which a vehicle that sends data one bit at a time it is connected to a data logger. It needs just to work at a proper speed of 9600 or 115200 bit/s.

From the point of view of the design, another important requirement is the length of the cables because the driver tester should be able to press the button easily, so it must be very close to his hands.

The device must feature press button for user interaction. Normally, the device acquires data and stores them into an SDcard. One pressed, the devices start saving data and any other pressing on that button will be ignored to avoid problems (e.g. unwanted data overwriting).

Also the temperature is a critical aspect to be considered and it must be managed by a cooling fan on the top of the device case. Datasheet provides a range of temperature from 0°C to 50°C. If the temperature exceeds the maximum value, then the microprocessor may start to slow down and consequently the performance of the entire device decreases.

The requirements state that CAN and serial logs must be saved as text files to be ready for post test processing. Videos, indeed, must have an *avi* file extension, so that they can be open with most common media player software.

2.3. Device design

The design of the prototype must be “modular”. This is because it is composed of a Raspberry Pi on the bottom, connecting by a pin headers 40 pin to a CAN-BUS Shield on the top and thanks to the settings file, the tester can configure the device depending on the foreseen usage. There is the possibility to use all the USB ports to record two videos and to save two log files from the serial interfaces while both CAN channels are active. The driver can choose how many and which peripheral use of each single test. All this aspect will be deeper discussed in the next chapter. The whole system is depicted in figure 1.

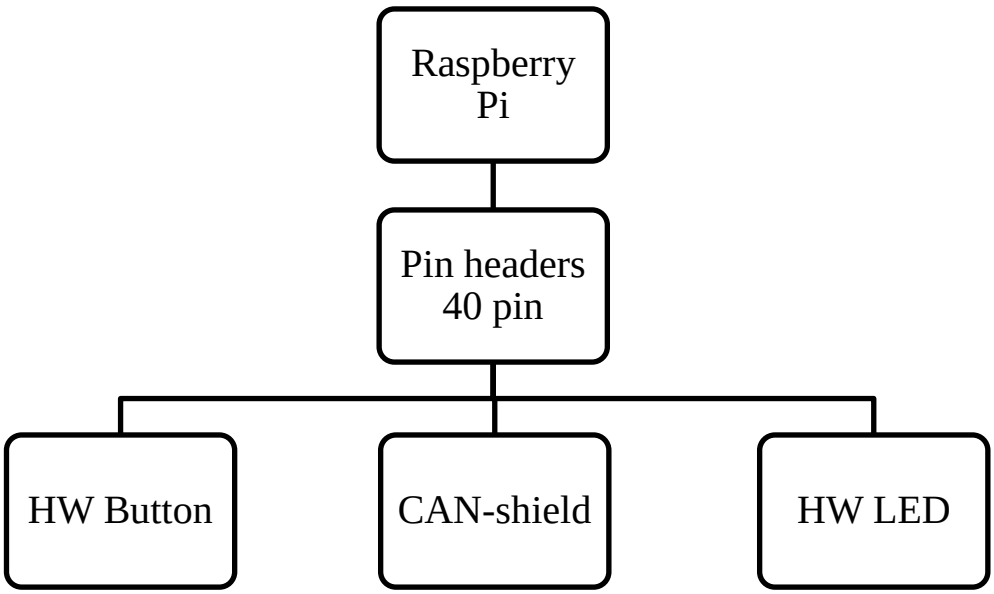


Figure 1. Block diagram of the expected system

2.4. Executional model

Fundamentally, there are two scenarios where the data logger is tested and used. The first where I start the development is the company laboratory where, with the help of the instrumentation at my disposal, I made a small bench in which there were an electronic control unit of the vehicle to simulate the CAN communication signals and also the radio (these components are exactly the same which are mounted on the real car. In this way I made all tests I need to test every new part of the written code. The other scenario is inside the car, where the device is installed and tested. Here the two USB webcams are mounted on the internal side of the windscreen by a suction cup and the other one is installed in front of the radio display by a flexible arm with a proper support.

2.5. Software milestones

The following milestones are referred to the development and testing of the software, written in C++, for the prototype of the data logger.

Task 0 – at week 2

- Define software architecture.
- Collaboration with LATAM region to define HW architecture.
- Identify and build using third part software.

Task 1 – at week 9

- Webcams management, code writing and debugging.
- Task 1 integration in the main code.

Task 2 – at week 11

- CAN management, code writing and debugging.
- Task 2 integration in the main code.

Task 3 – at week 12

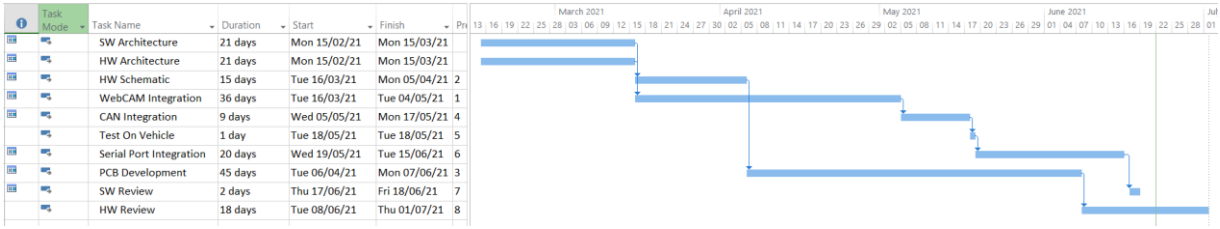
- First test on field (inside car).

Task 4 – at week 15

- Serial management, code writing and debugging.
- Task 4 integration in the main code.

These milestones were defined at the beginning of the project in order to have a general idea of the timing of the completion of the project.

2.6. Project Gantt Chart



Task Mode	Task Name	Duration	Start	Finish	Pr
	SW Architecture	21 days	Mon 15/02/21	Mon 15/03/21	
	HW Architecture	21 days	Mon 15/02/21	Mon 15/03/21	
	HW Schematic	15 days	Tue 16/03/21	Mon 05/04/21	2
	WebCAM Integration	36 days	Tue 16/03/21	Tue 04/05/21	1
	CAN Integration	9 days	Wed 05/05/21	Mon 17/05/21	4
	Test On Vehicle	1 day	Tue 18/05/21	Tue 18/05/21	5
	Serial Port Integration	20 days	Wed 19/05/21	Tue 15/06/21	6
	PCB Development	45 days	Tue 06/04/21	Mon 07/06/21	3
	SW Review	2 days	Thu 17/06/21	Fri 18/06/21	7
	HW Review	18 days	Tue 08/06/21	Thu 01/07/21	8

Figure 2. Project Gantt Chart

This type of Gantt Chart is useful to understand and visualize the contribute of people that work on the project, underlining the timing and the task of each member.

2.7. Team organization

During the whole project I collaborate with LATAM to coordinate the hardware with the software to achieve full compatibility among the parts. For the activity scheduling and for the job organization, we have used different platforms powered by Google: Google Drive to organize and save in real time the daily work (software code, slides, schematics), Google Meet to make conferences for synchronize and parallelize the team activities and Google Calendar to schedule meetings and calls.

3. HARDWARE

3.1. Raspberry Pi 4 Model B

Raspberry Pi 4 Model B is the latest product in the popular Raspberry Pi range of computers. It offers groundbreaking increases in processor speed, multimedia performance, memory, and connectivity compared to the prior-generation Raspberry Pi 3 Model B+, while retaining backwards compatibility and similar power consumption. For the end user, Raspberry Pi 4 Model B provides desktop performance comparable to entry-level x86 PC systems. The dual-band wireless LAN and Bluetooth have modular compliance certification, allowing the board to be designed into end products with significantly reduced compliance testing, improving both cost and time to market.

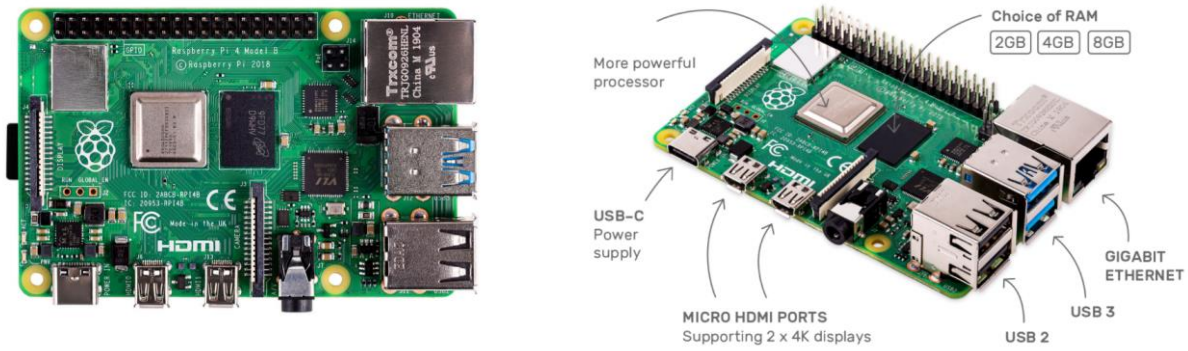


Figure 3. Raspberry Pi 4 model B

Going deeply on the hardware specification:

- **Processor**: Broadcom BCM2711, quad-core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz.
- **Memory**: 4GB LPDDR4.
- **Connectivity**: 2.4 GHz and 5.0 GHz IEEE 802.11b/g/n/ac wireless LAN, Bluetooth 5.0, BLE, Gigabit Ethernet, 2 × USB 3.0 ports, 2 × USB 2.0 ports.
- **GPIO**: Standard 40-pin GPIO header.
- **Video & sound**: 2 × micro-HDMI ports (up to 4Kp60 supported), 2-lane MIPI DSI display port, 2-lane MIPI CSI camera port, 4-pole stereo audio and composite video port.
- **Multimedia**: H.265 (4Kp60 decode), H.264 (1080p60 decode, 1080p30 encode), OpenGL ES, 3.0 graphics.
- **SD card support**: Micro SD card slot for loading operating system and data storage.
- **Input power**: 5V DC via USB-C connector (minimum 3A1), 5V DC via GPIO header (minimum 3A1), Power over Ethernet (PoE)–enabled (requires separate PoE HAT).
- **Environment**: Operating temperature 0–50ºC.

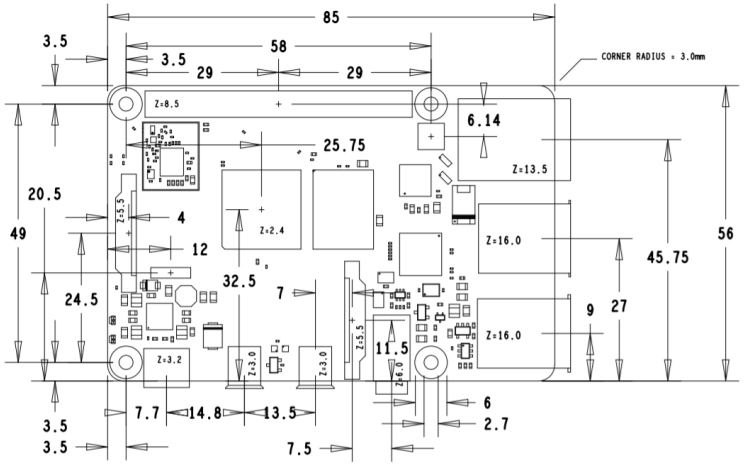


Figure 4. Dimension in mm of the Raspberry Pi

3.2. 2-Channel CAN-BUS (FD) Shield

The Controller Area Network protocol (CAN) ia a two-wire (twisted-pair), bidirectional serial bus communication method that allows electronic subsystems to be linked together and interact in a network. The physical layer uses differential transmission on a twisted pair wire and a non-destructive bit-wise arbitration is used to control access to the bus. The messages are small and are protected by a checksum, there is no explicit address in the messages, instead, each message carries a numeric value which controls its priority on the bus and may also serve as an identification of the contents of the message. It has an elaborate error handling scheme that results in retransmitted messages when they are not properly received. CAN Bus has a multi-master capability, meaning any node on the bus can initiate communication to any other node in a network and nodes on the CAN network must operate at the same nominal bit rate otherwise errors will occur on receiving side. Two or more nodes are required on the CAN Bus network to communicate. A message consists primarily of the ID (identifier), which represents the priority of the message, and up to eight data bytes. A CRC, acknowledge slot (ACK) and other overhead are also part of the message. CAN Bus message frame is shown on Figure 5.

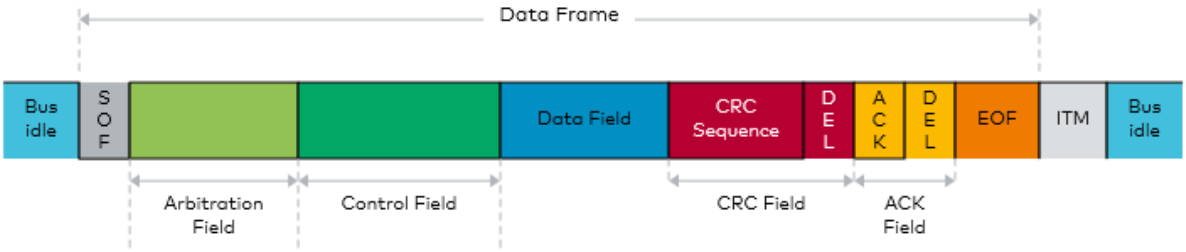


Figure 5. CAN data frame

This CAN-BUS Shield includes cost effective CAN FD controller. “FD” means it can not only support both CAN frames in the Classical format, but also CAN Flexible Data Rate format. CAN FD protocol is an upgraded version of the traditional CAN BUS that increases the CAN’s transmission rate from 1Mbps to 8Mbps. At the same time, CAN FD improves real-time performance and extends user data frames, providing higher efficiency. This shield lets Raspberry Pi become CAN master because there are 2-channel CAN-BUS on-board.

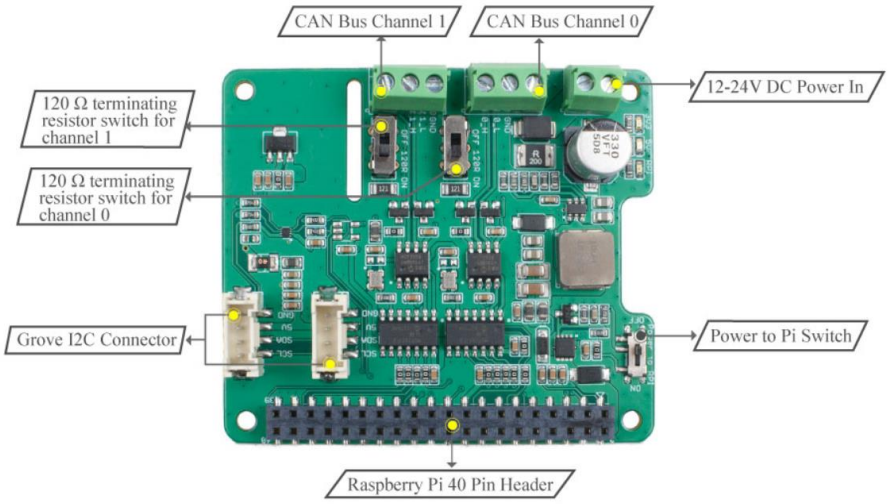


Figure 6. 2-Channel CAN-BUS (FD) Shield primary components

The Seeed 2-Channel CAN-BUS(FD) Shield for Raspberry Pi is based on MCP2517 CAN FD Controller and MCP557 CAN FD Transceiver which communicates with Raspberry Pi via the high-speed SPI interface. The shield can be powered via Raspberry Pi’s 40 pin header or using the 12 - 24V DC screw terminal to supply the power for the whole system. The devices at both ends of the CAN BUS need a 120Ω terminating resistor to avoid reflection. With the help of an on-board 120Ω resistor and its enable switch, the on-board resistor can be easily toggled on and off.

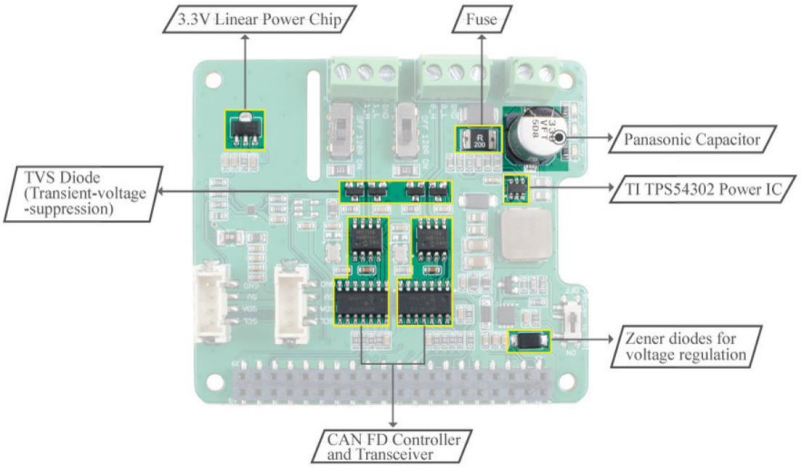


Figure 7. 2-Channel CAN-BUS (FD) Shield secondary components

To ensure the stability of communication is used the Panasonic capacitors and TI power chips, along with quality circuit design, to build a stable power management system, which provides stable 3.3V and 5V power supply for all parts of the system.

3.3. Power supply

Power supply to power up the raspberry Pi by KL30 is provided directly by OBD port. In order to create a link between 12V of the vehicle and 220V adapter plug into the cigar lighter it is used two pair of banana plug (female connected to the adapter and male connected to the KL30 and ground OBD pins). Then the device is normally connected to the electrical socket with its default plug.

Figure 9. Positive and negative poles of KL30 (in red and black)

Figure 8. Electrical socket and cigar lighter with custom wires

3.4. Push button & LED

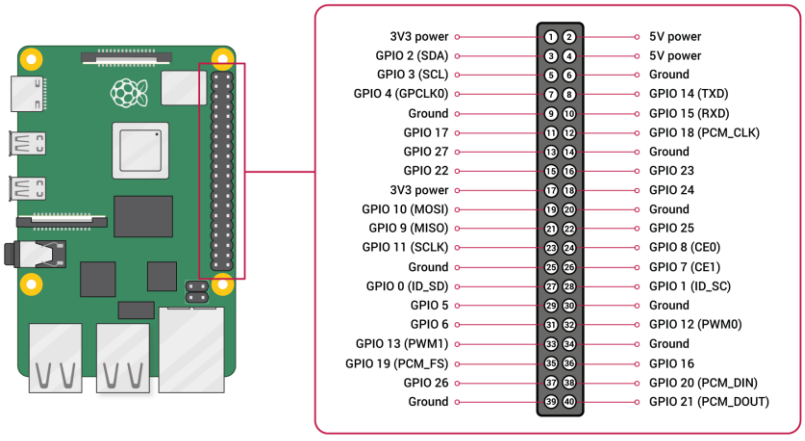


Figure 10. Pinout of Raspberry Pi

The colour of the LED is red and it is set as active-high, so it is on pin 1 (3V3) and on pin 1 (GPIO 4). The push button is a classic mechanical click button and it is set as active-high, so it is on pin 17 (3V3) and on pin 11 (GPIO 17). The pins have been selected in such a way that they can be welded close to each other on the board. All of these pins are managed and configured in the C++ software.

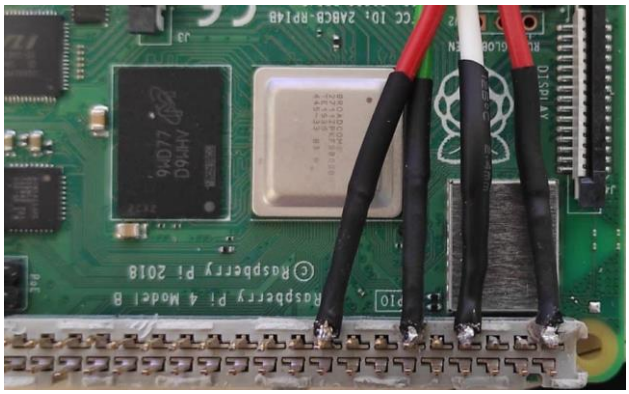


Figure 11. LED and button configuration

These two components are inserted inside a closed case to protect them during in-vehicle tests and the wires are blocked with a secure strap to make sure that they do not move, as it is shown in figure 12.

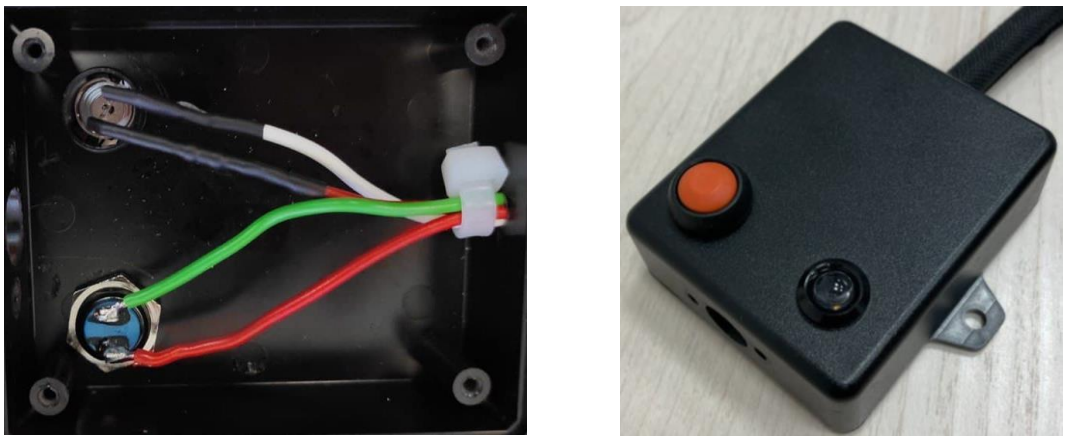


Figure 12. Case containing red LED and button with hidden cables

The installed LED has a very important practical function because it advises, by a fixed red light, the driver that the device is saving the recorded data. In this situation, if the tester tries to press the button when the light is on, then the software prevents the occurrence of another trigger event.

3.5. OBD2 cable

The OBD cable has two main functions:

- Power the Raspberry Pi.
- Establish a link between CAN lines of vehicle and CAN-shield of the board.

The OBD cable was made to be versatile for both bench and in-vehicle tests and it is approximately two meters. In the figure 13 it can be noticed that there are two pair of DB9 connectors, the shortest one design for bench tests and the longest one implemented for in-vehicle tests. The red and black wires are the KL30 and ground of OBD2 connector and they were configured according to the specifications of the vehicle where the tests are made. In this way, even if the driver performs a “Key

Cycle” which means that it turns off the vehicle for few seconds and then turn on it, the data logger keeps logging without any problems. At the end, a protective sock is used for the entire length of the twisted wires for safety reasons.

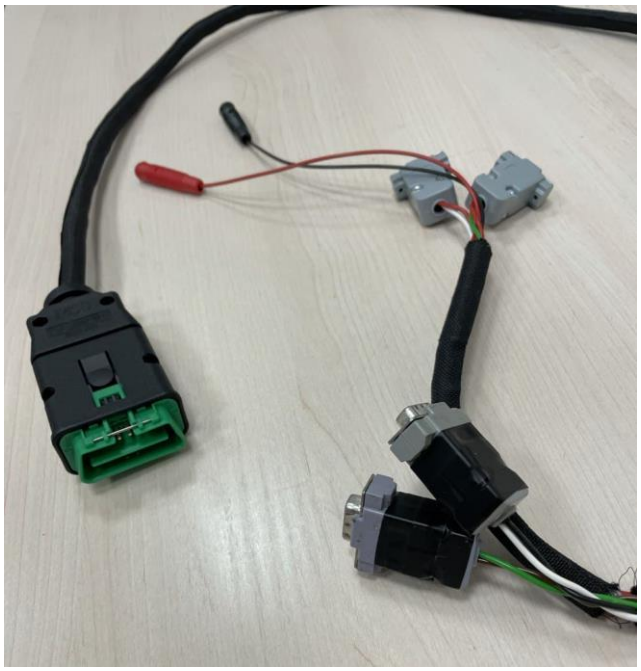


Figure 13. OBD2 custom cable with DB9 connectors and power supply wires

The DB9 connectors are used to connect the CAN-bus of the car with the one of the board. The used pins are 2 and 7 as standard of CAN-bus lines. On the side of the cable there are DB9 of female type and on the side of the board the connectors are of male type.

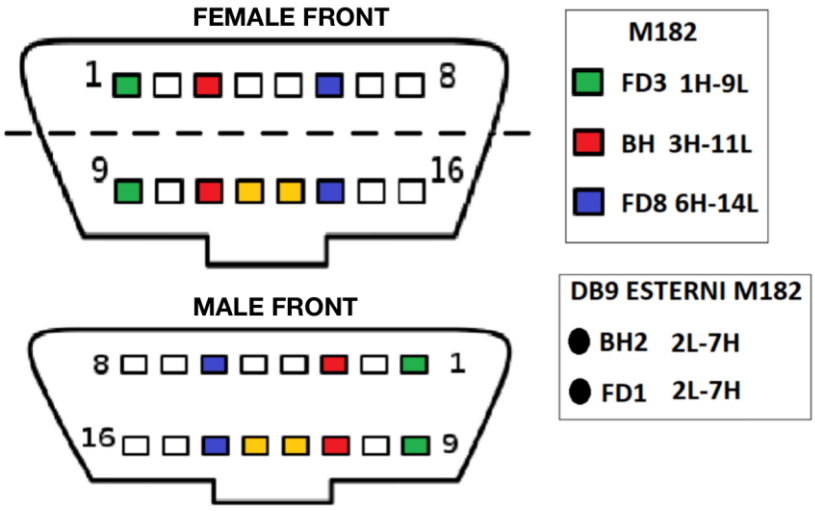


Figure 14. OBD and DB9 schematics

In the figure 14 it is shown the schematic of the pins for the OBD and DB9 CAN-bus configuration for M182 vehicle. The OBD cable was configured with BH and FD8 lines, but it can be modified if need be.

3.6. Cooling fan

For the correct behaviour of the device, the temperature has an important role to play here. In fact, as said before in the previous chapters, the Raspbery Pi has a limited range of temperature which goes from 0°C to 50°C to ensure the best performance as possible. So, the fan is placed on the top of the case in order to maintain the temperature as constant as possible. It can work both at 3V or 5V and is physically connected to the pin-header of the board by two pins, one for the power supply and the other for the ground.

Figure 15. Cooling fan on the top of the data logger

3.7. Data Storage

The place to store the recorded data can be chosen in the setting file before to start the test and consequently the software will place properly the correct saving path inside the code. The tester can choose if save the files directly inside the SD card, containing the operating system, or inside an external pendrive connected to one of the available USB ports. About the size of the storage capacity, it was taken 128GB as reference. This size allows to record a lot of data until the saturation. Since the user can set also the duration of each log, it can choose a smaller size memory according to his needs.

4. PERIPHERALS

This chapter will explain which peripherals have been used and how they have been integrated at software level.

4.1. Webcams

The data logger is able to record at maximum two videos simultaneously, but the tester can decide, as appropriate, whether to enable both webcams, only one or none. The device will automatically detect how many and which webcams have been connected to the USB ports and the software will set them as input devices for audio and video capture. The engineer, before starting the test, will have to decide which webcams to use and how to use them. The basic idea is to have two types of settings: external or internal record mode.

Figure 16. Logitech C920

Figure 17. Logitech C615

External mode: with this type of setting only the video is captured because to have also the audio will be a waste of hardware resources and it is not useful to the successive analysis in the laboratory. The purpose of the external webcams is to record what happen to the ADAS car system while driving.

Internal mode: if this mode is selected, both audio and video are captured by the webcam and it is useful for further analysis because the internal webcam is pointed on the radio interface so also the audio is an important source of information.

On the software side there are nested *if else conditions* to guarantee that all combinations of webcam configurations will be covered. For each webcam it is fundamental to specify the device number (*/dev/video*) and the audio card (*plughw:CARD*) intended for tests.

Unfortunately, we have to deal with the hardware resource limit of the Raspberry Pi and it means that we can not set both webcams in external mode because the system will freeze. For the same reason, the maximum resolution that the device can achieve is 480x640p with a framerate of 20fps. These results and considerations come from various tests on the system.

The algorithm to implement the recording strategy is based on the logic of circular buffer as represented in the figure 18.

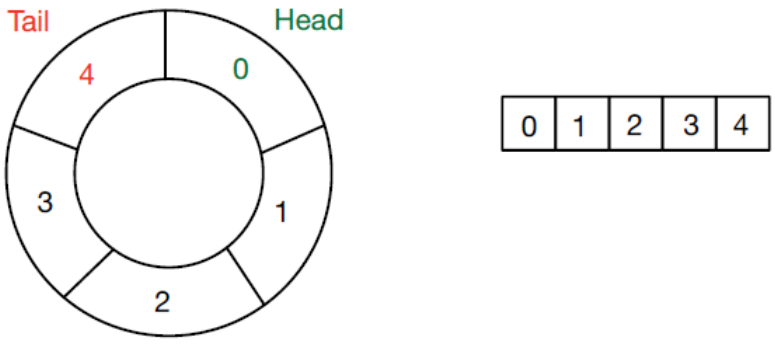


Figure 18. Circular buffer log representation

In general, a linear buffer is a data structure that implements the FIFO (First-In-First-Out) ordering. This means that the first item added to your queue is the first one out. For this project, it was used a circular buffer which is essentially a queue with a maximum size or capacity which will continue to loop back over itself in a circular motion.

The algorithm works in this way: once the user has decided, trough the setting file, how long want to take the video files, the software automatically splits the entire length of the video in five parts. Each part is managed using different timing calculations, trying to reduce useless operations in order to speed up the execution of the program. The software is written to record in loop 5 videos which are coded as numbered *segments* to be recognizable during the saving stage. Depending on when the driver pushes the button the saving function will read a specific segment from the record function and the algorithm makes some steps:

- It copies the parts we are interested in from the circular buffer into different location.
- It cuts the involved footage with a resolution of seconds.
- It pastes all footages together.
- It gives the final video on the output folder.

To perform all these media operations, it is used a dedicated framework called FFmpeg, which is able to decode, encode, transcode, mux, demux, stream, filter and play pretty much anything that humans and machines have created. It supports the most obscure ancient formats up to the cutting edge. No matter if they were designed by some standards committee, the community or a corporation. It is also highly portable. On the operating system, there were installed all *FFmpeg* libraries for developers as follow:

- Libavutil: is a library containing functions for simplifying programming, including random number generators, data structures, mathematics routines, core multimedia utilities, and much more.

- Libavcodec: is a library containing decoders and encoders for audio/video codecs.
- Libavformat: is a library containing demuxers and muxers for multimedia container formats.
- Libavdevice: is a library containing input and output devices for grabbing from and rendering to many common multimedia input/output software frameworks, including Video4Linux, Video4Linux2, VfW, and ALSA.
- Libavfilter: is a library containing media filters.
- Libswscale: is a library performing highly optimized image scaling and color space/pixel format conversion operations.
- Libswresample: is a library performing highly optimized audio resampling, rematrixing and sample format conversion operations.

The FFmpeg command lines, used to record videos by webcams, contains in particular Video4Linux2 for video and ALSA for audio.

4.2. CAN-BUS

Data logger exploits the CAN-shield to make the connection between vehicle CAN signals and the mounted shield. From the prototype coming out two pairs of twisted wires terminating with two DB9 connectors as shown in the figure 19. These connectors were marked with two different labels: CAN0 and CAN1. In this way the tester can clearly distinguish the CAN channels. It is important to say that each channel can be set as FD or BH type.

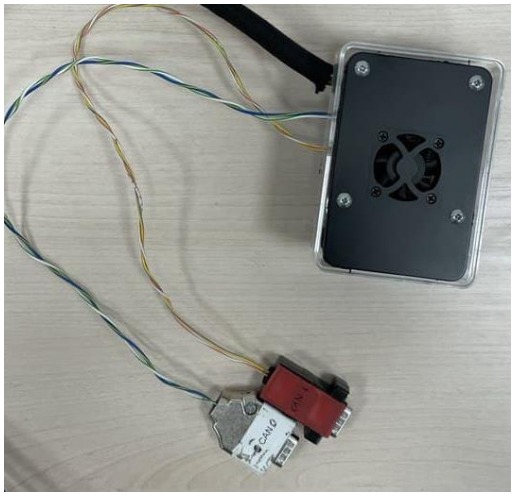


Figure 19. DB9 connectors for CAN1 and CAN2

From settings file the driver can decide to have both channel active, just one or nothing. Another important aspect is that each channel can be of any kind among FD and BH. It is important to analyse the possible configuration for each CAN type in terms of bitrate, sample-point, and phase-segment:

- **Bitrate**: bitrate describes the rate at which bits are transferred from one point to another. So, it measures how much data is transmitted in a given amount of time. Bitrate is commonly measured in bits per second (bps), kilobits per second (Kbps), or megabits per second (Mbps).
- **Sample-point**: the sample point is the location, typically given as a percent value, inside each bit period where the CAN controller looks at the state of the bus and determines if it is a logic zero (dominant) or logic one (recessive). All CAN controllers allow this point to be configured and it is always specified as a percentage from the start of the bit period. The location of the sample point is a trade-off. An early sample point decreases the sensitivity to oscillator tolerances and allows lower-quality oscillators. A late sample point allows for a longer signal propagation time and therefore a longer bus. A later sample point is useful for non-ideal bus topologies.
- **Phase-segment**: it is divided into segment 1 and segment 2. The first is programmable to be 1,2, ... 8 Time Quanta long. It is used to compensate for

edge phase errors and may be lengthened during resynchronization. The second one is the maximum of segment 1 and the information processing time long. It is also used to compensate edge phase errors and may be shortened during resynchronization. Information Processing Time is less than or equal to 2 Time Quanta long. The total number of Time Quanta has to be from 8 to 25.

It is important to notice that CAN BH channel has only bitrate as parameters and it is set to 125000 as standard says. On the other hand, CAN FD channel needs to have all of these parameters. Firstly, the bit rate set to 500000, sample-point to 0.8, phase-segment 1 to 7 and phase-segment 2 to 2.

The algorithm to implement CAN operations is based on three main phases: initialization, acquisition and save.

- The first is the initialization is made in the main part of the code and it is checked which CAN channel is used and what type it is. Only after this check, the channel is initialized using the command line "*system*" to do the operation with console logic.
- The second phase is about acquisition of data by CAN-bus and it is performed in a dedicated function that works in loop, as the record function for the webcams do. This task needs a specific command called "candump" to start to read data from the CAN and to save them into specific file in *.asc* format that stand ASCII. In this way the testrs can check the file directly on CANalyzer.
- The last phase is the save of data, but the sniffed data in not conform to the CANalyzer data format, so the file must be re-arranged to be compliant with the tool. For this purpose, it is used a dedicated function which starts at the beginning of the saving phase and by the use of *awk* command the input file is modified column by column. These operations are like remove parenthesis, swap one column to another and add new columns.

Once CAN files are stored on SD card or on external USB, the tester can take them and he can import these files into CANalyzer tool to verify the acquired signals.

4.3. Serial ports

The serial port peripheral is needed in this data logger because not all vehicles to test output the same data via CAN-bus, for some vehicle to get data from the radio the driver needs to connect to it via USB-FTDI.

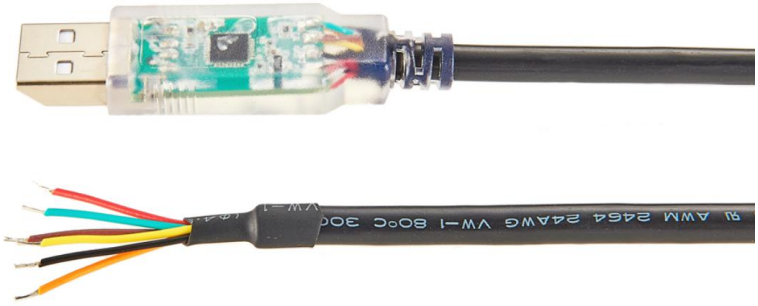


Figure 20. USB-FTDI cable

The USB-RS422 cable uses FTDI's FT232RQ USB to serial UART IC device. It contains a small internal electronic circuit board, utilising the FT232R, which is encapsulated into the USB connector end of the cable. The integrated electronics also include the RS422 transceiver plus Tx and Rx LEDs which give a visual indication of traffic on the cable (if transparent USB connector mould specified). The other end of the cable is bare, tinned wire ended connections by default, but for this project, it was customised with the same USB-FTDI interface. The cable has an internal EEPROM used to store USB Vendor ID (VID), Product ID (PID), device serial number, product description string and various other USB configuration descriptors. The cable is supplied with the internal EEPROM pre-programmed. The FT232R has a low 15mA operating supply current and a very low USB suspend current of approximately 70µA. The USB interface of the FT232R uses as little as possible of the total USB bandwidth

available from the USB host controller. The FT232R includes the FTDIChip-ID security dongle feature. This chip feature allows a unique number to be burnt into each cable during manufacture. This number cannot be reprogrammed. This number is only readable over USB can be used to form the basis of a security dongle which can be used to protect any customer application software being copied. The USB-RS422 cables are capable of operating over an extended temperature range of -40º to +85º C thus allowing them to be used in commercial or industrial applications. To use and to test the serial port connection between laptop and Raspberry Pi, I had to install the driver that enables communication between two devices.

There are three main functions used to implement this peripheral: one for configuration of the port, so here there is the main initialization where it is specified the device name "/dev/ttyUSB" because all serial devices connected to the Raspberry Pi is recognized as ttyUSBx. After some error checks this function sets all parameters for the serial communication such as parity bit, stop bit, data size, CS, hardware flow control. There are also two important parameters to set for timely decision, VTIME which wait for x in deciseconds returning as soon as any data is received and VMIN which wait for x char until return. Inside the function the Baudrate is set according to which is written in the setting file. It can assume two possible values: 9600 or 115200. It is very important that that the Raspberry Pi and serial source have the same baudrate, otherwise the message could be unreadable. Another one to receive in loop the serial messages saving them into a char buffer, adding also the timestamps. The last one to save into a file what is received and to filter the messages by time using the *awk* command.

5. SOFTWARE

5.1. Architecture flowchart

The software has many functionalities that allow the device to work properly as specifications state. These functionalities are:

- Reading lines from text file.
- Management of the set peripherals in a synchronous way.
- Implementation of parallel threads.
- Execution of reading and saving functions inside threads.
- Handling possible error occurrences.
- Management of input/outputs GPIO.

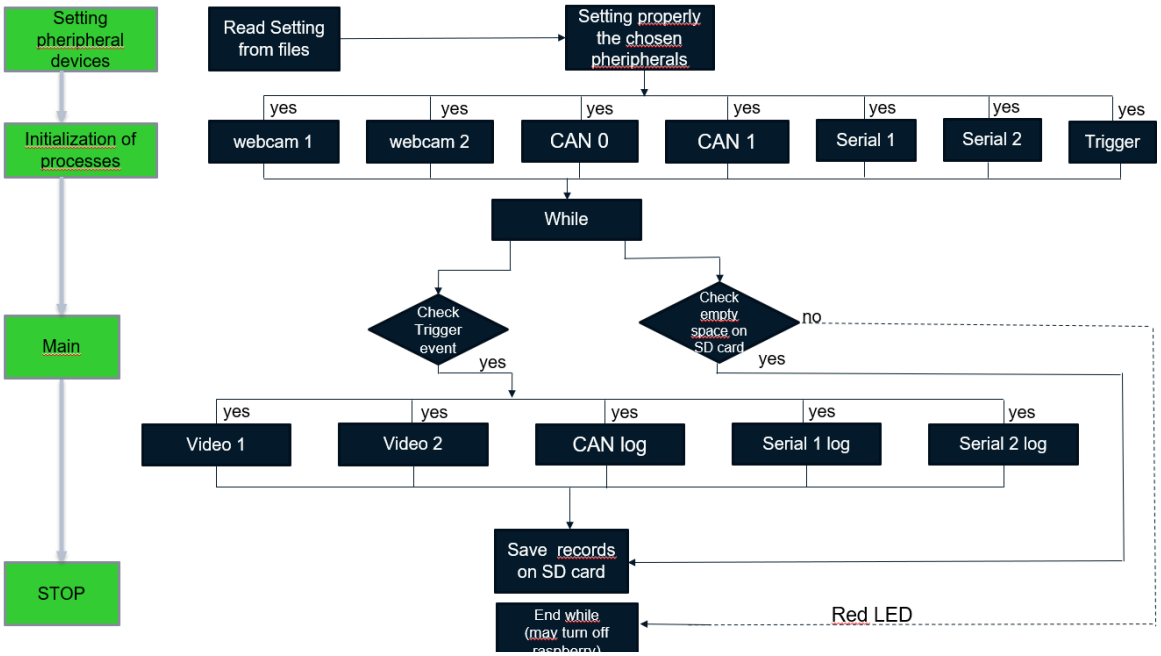


Figure 21. Flowchart of software design

The figure 21 describes the whole software architecture divided in green and dark blue blocks. The green blocks represent the macro functions containing many micro frunctions which compose the whole logic.

- The first macro block is: **Setting peripheral devices**. It includes the entire setting procedure, starting from read setting from a file that is made by a specific function called read_file(). The type of this function is *VAL* and it plays a very key role for the code because it reads all settings from the file, compiled by the tester, and it properly fills a *VAL* structure that contains all the fields involved in peripheral initialization as it can be noticed on figure 22.

```
//struct used into read_file()
struct VAL {

        int error_code = 0;
        string path_to_store = "";

        string blocks_length = "";

        int CAN_0_ok = 0;
        string CAN_0_type = "";
        int CAN_0_baud = 0;

        int CAN_1_ok = 0;
        string CAN_1_type = "";
        int CAN_1_baud = 0;

        int CAM_1_ok = 0;
        string CAM_1_type = "";
        string CAM_1_card = "";
        string CAM_1_video = "";

        int CAM_2_ok = 0;
        string CAM_2_type = "";
        string CAM_2_card = "";
        string CAM_2_video = "";

        int SERIAL_1_ok = 0;
        int SERIAL_1_baud = 0;
        int SERIAL_1_usb = 0;

        int SERIAL_2_ok = 0;
        int SERIAL_2_baud = 0;
        int SERIAL_2_usb = 0;

        int serial_port1_obj = 0;
        int serial_port2_obj = 0;
    };
```

Figure 22. C++ struct of VAL type

In addition to peripherals, are also configured the desired length of the video and also in which place to store the final files. This function is composed by a nested if conditions which ensure that each line of the file is red and processed

correctly. This section will influence the rest of the code, including threads because if one peripheral is not used for the test, then the associated thread will not start. In this way we can save a lot of resources, also keeping down the temperature. The setting file section will be discussed more in detail in a next more detailed chapter.

- The second macro block is: **Initialization of processes**. In this section the software checks all peripheral states in order to create the needed threads for each enabled peripheral. In this part the created processes work in a loop from the turning on of the data logger and they start to record all data troughout the peripherals. All recorded data are stored in temporary files which will be processed later by the next blocks.
- The third and the fourth macro block are: **Main** and **Stop**. The main section of the code contains a while condition, including two important checks: if the tester has pressed the button (trigger) and if there is enough space on disk to save the output files. Analysing the check trigger event, we can have two possible results: if it is not present (button is not pressed), software just continue to record in loop without saving nothing or if there is a trigger event (button pressed) the software creates as many triggers as the processes used and at the same time it checks the available space on disk. If we have a positive answer, then the saving process can do its work, otherwise the code exits from the while condition and it stops the execution of the flow. This last event belongs to the stop block.

All of the micro blocks are managed with threads that are deeply discussed in the next section.

5.2. Threads

A thread is the shortest series of programmed instructions that may be controlled independently by a scheduler, which is usually part of the operating system, in

computer science. Threads and processes are implemented in some operating systems, but in most other situations, a thread is a component of a process as it is shown on figure 23. Within a single process, several threads can run concurrently and share resources like memory, however separate processes do not share these resources. At any one moment, a process' threads share its executable code as well as the values of its dynamically allocated variables and non-thread-local global variables.

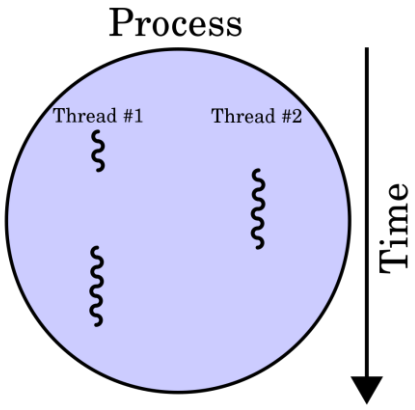


Figure 23. Threads as a part of a process

Multithreading allows an application to have multiple flows of control that are executed simultaneously. On the Raspberry Pi, this can be useful to control the various peripherals and input/output pins, for example to react to the button status change.

Inside the code there are two main distinctions between the background thread and the threads related to the trigger event:

- On the background there are the threads working continuously from the power on of the device and their job is to record constantly the videos from the webcams and the logs from CAN and serial busses. These threads are outsides from the main while cycle because they need to be created just one time and left to work until the driver decides to stop the test, turning off the data logger.

- On the other hand, there are the threads, inside the while cycle, related to the trigger and saving functions. These threads need to be synchronized in order to respond correctly and quickly to the button pressing (trigger event).

```
// Background threads
thread rec(loop_rec);
rec.detach();

thread can0(loop_CAN0);
can0.detach();

thread can1(loop_CAN1);
can1.detach();

thread serial1(loop_serial1);
serial1.detach();

thread serial2(loop_serial2);
serial2.detach();
```

Figure 24. Backgroung threads

```
while (true)
{
    // Threads creation
    thread btn([&] {button_status = button_pressed(); });
    thread th_save(save_CAMs, ref(button_status));
    thread can0_save(CAN0_save, ref(button_status));
    thread can1_save(CAN1_save, ref(button_status));
    thread s1_save(serial1_save, ref(button_status));
    thread s2_save(serial2_save, ref(button_status));

    // Threads detach
    btn.detach();
    th_save.detach();
    can0_save.detach();
    can1_save.detach();
    s1_save.detach();
    s2_save.detach();
}
```

Figure 25. Threads inside the while condition

Before to analyse images above, can be useful to explain what *detach()* function does and why it is preferable to use it instead of *join()* function. A C++ thread object generally represents a thread of execution, which is an Operating System or platform concept.

When *join()* is called, the calling thread will block until the thread of execution has completed. Basically, this is one mechanism that can be used to know when a thread has finished. When *join()* returns, the Operating System thread of execution has completed and the C++ thread object can be destroyed.

When the *detach()* is called, the thread of execution is "detached" from the thread object and is no longer represented by it so, they are two independent things. For example, the C++ thread object can be destroyed and the Operating System thread of execution can continue on without any problems. If the program needs to know when that thread of execution has completed, some other mechanism needs to be used, but *join()* can not be called on that thread object any more, since it is no longer associated with a thread of execution. The code of the data logger needs to have threads that are independent from the main thread, in this way can work without any blocks.

As the figure 24 and 25 show, each thread has as argument its related function, in this way we obtain the parallelization of those functions that need to work together. Starting to analyse the figure 24, it can be noticed that all threads are created executing their functions and soon after detached. So, once the device starts its execution, the background threads are created and they execute their own functions. On the other hand, figure 25 shows many threads created and detached inside the while cycle. The most important thread is the one named *btn*, because it contains and executes the function for the button pressing, so every time the user presses the button, this thread has the task of changing the variable related to the button status (1 if pressed, 0 if not pressed). In the line of the button thread is interesting to see that the return value of the function is passed and saved on button_status global variable so, everyone can access it, reading the value. Thanks to this mechanism, it is possible to pass a value among all threads. In fact, thanks to the syntax: (*[&] {variable = function(); })* thread can store the return value of its function inside a variable, so that all other concerned thread can

use this value by reference as input of their functions. The syntax to pass a value inside a thread in C++ is the following: *(function, ref(variable))*. Like this it is possible to share the same variable with the most updated value among all threads. As said in the previous chapter, threads for peripherals are created only if these peripherals were set in the file setting, saving resources.

There are a lot of threads inside the code and they need to be synchronized with the trigger event. To do that, threads have the same reference time to follow. In a real scenario, where testers want to study the logger results, all output files must have the same set time (e.g. x minutes of video, x minutes of CAN and x minutes of serial). Once the time is set in the setting file, then every function that works for saving peripheral result, is aligned with the same time. To achieve this goal, it was used the *<ctime>* library made available by C++, in particular *time_t* type of variable represented in figure 26.

```
#include <ctime>
time_t start = time(NULL); // to take the elapsed time from start to trigger
```

Figure 26. Use of <ctime> library

This type of variable is an alias of a fundamental arithmetic type capable of representing times, as those returned by function time. For historical reasons, it is generally implemented as an integral value representing the number of seconds elapsed since 00:00 hours, Jan 1, 1970 UTC. This function is fundamental inside the code also to generate timestamps used in CAN and serial logs.

Threads management is the most complicated part of the code since they are many and overlap at high frequency. In fact, many times during debugging the operating system crashed due to resource busy so, a lot of time was spent to create threads on the right place and to pass some variable between them.

5.3. Settings file

Customize settings file is the first step done by the driver in order to prepare the data logger for in-vehicle tests.

```
[CAN_0]
isUsed=1
type=FD
baudRate=500000

[CAN_1]
isUsed=0
type=BH
baudRate=125000

[CAM_1]
isUsed=1
type=internal
card=C615
video=0

[CAM_2]
isUsed=0
type=external
card=C920
video=1

[SERIAL_1]
isUsed=0
baudRate=9600
USB=0

[SERIAL_2]
isUsed=1
baudRate=9600
USB=1

[STORE]
SDcard

[VIDEO_BLOCKS_LENGTH]
time=45
```

Figure 27. Example of setting.txt

On figure 27 an example of settings.txt is visible. Proceeding in order it can be noticed how the different lines of the file are written, each section corresponds to one peripheral and each is separated from the other by an empty row. On the top of each block there is a header containing the name of the peripheral between square brackets. This header line is very important for the code to recognize in which section we are by *read_file()* function. It is based on *getline()* method that is used to read a string or a line from an input stream. It extracts characters from the input stream and appends it to the string object until the delimiting character is encountered. While doing so the previously stored value in the string object str will be replaced by the input string if any. It is a part of the <*string*> header. Now, going more deeper on the lines, after the header there are some fields:

- **isUsed**: this field represents if the peripheral is enabled (1) or not enabled (0) and it is present in every peripheral section.
- **type**: it is used in Webcams and CAN blocks to indicate, as the word itself says, the type of the webcam (internal or external) and the type of CAN-bus (BH or FD).
- **baudrate**: this field is present on CAN and Serial blocks and it indicates at which speed the ports have to communicate with the source. For the CAN the values are 125000 or 500000, instead, for the serial port the values are 9600 or 115200.
- **card**: it indicates the name of the sound card of the webcam set to internal.
- **video**: it is used to specify the label that the plugged webcam have to set as external driver.
- **USB**: it is just for serial ports and it indicates the number of USB at wich the serial cable is plugged.
- **SDcard**: in this field the tester can choose whether to set as saving place the SD card or an external USB pendrive writing **PenDrive** on this field. The correct path will be dynamically insert in the final path of the video.
- **Time**: user can set the length of the output files, choosing among pre-set values (3, 5, 10 minutes).

5.4. C++ as programming language

C++ is a statically typed, compiled, general-purpose, case-sensitive, free-form programming language that supports procedural, object-oriented, and generic programming. It is regarded as a middle-level language, as it comprises a combination of both high-level and low-level language features. C++ gives the programmer full control over the resources used and their allocation and de-allocation in memory, not providing any automatic dynamic management functionality. In addition, the basic semantics, so low level, allows direct control over how data is handled and managed at the hardware level by the machine on which you are running your code, for example, through the use of constructs such as pointers. The C++ is built to be a general purpose language, this makes it incredibly suitable in various application fields, and allows you to do almost anything. In short, it is a language that has proven to be able to ensure very high performance, unattainable by counterparts, as well as considerable versatility, this often makes it, the only feasible choice when you need to write software with very low latency, or you have to work with hardware that provide limited resources such as memory as in embedded systems. It allows the programmer complete control over the hardware he is programming as well as the data he is handling.

5.5. C++ Libraries

The library used in the code is divided into two categories: libraries used to control input/output peripherals and general C++ libraries.

In the first category, there are just three libraries for serial and gpio function, because CAN and webcams use the command line of the Linux environment.

```
#include <gpiod.h>
#include <termios.h>
#include <fcntl.h>
```

Figure 28. GPIO and Serial libraries

<gpiod.h> is part of libgpiod, a C library and tools for interacting with the Linux GPIO hardware. Gpiod provides a set of command line tools that are very useful for interactively exploring GPIO functions and can be used in shell scripts to avoid the need to write extra code if you only need to perform basic GPIO functions. The following commands are provided:

- **gpiodetect**: list all GPIO chips present on the system, their names, labels and number of GPIO lines.
- **gpioinfo**: list all lines of specified GPIO chips, their names, consumers, direction, active state and additional flags.
- **gpioget**: read values of specified GPIO lines.
- **gpioset**: set values of specified GPIO lines, and potentially keep the lines exported and wait until timeout, user input or signal.
- **gpiofind**: find the GPIO chip name and line offset given the line name.
- **gpiomon**: wait for events on GPIO lines, specifying which events to watch, how many events to process before exiting or if the events should be reported to the console.

The most used command during various debug sessions was *gpioinfo*, very useful to understand the behaviour of the pins used in the code. An example of console output is shown in the figure 29.

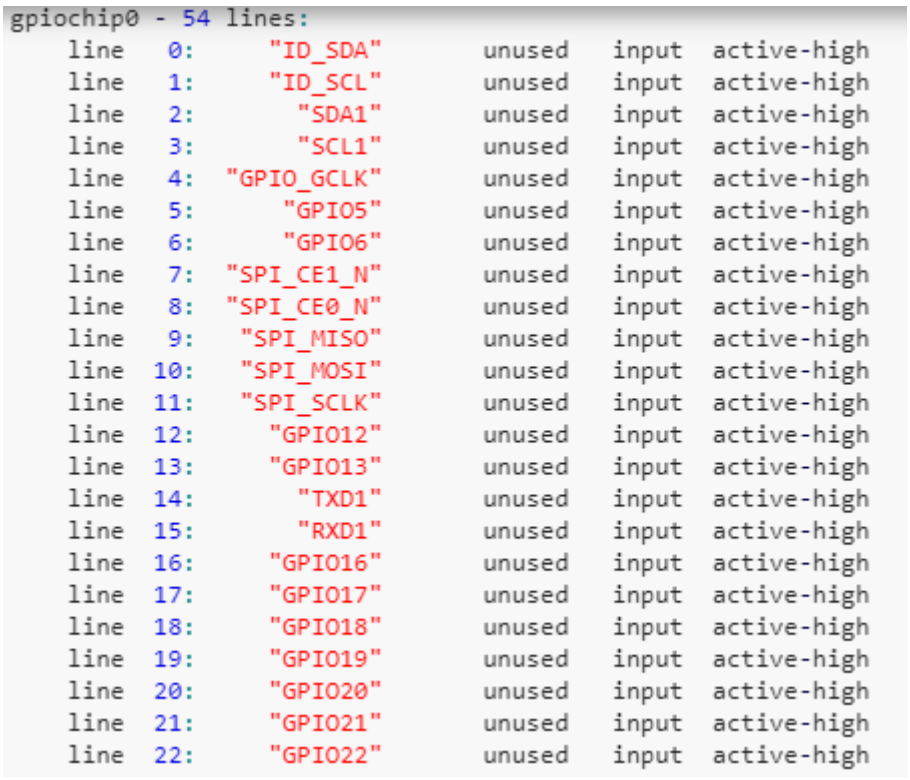


Figure 29. Available GPIO lines

If during debugging some pins are disabled unintentionally due to some random errors, then just by consulting the table on figure 29 I was able to take some fast decisions about the system (e.g. reboot Raspberry Pi to reset all pins or make some hardware changes on physical button and LED). The C API allows calling the gpiod library from C++ or languages that support C APIs. The function that recognizes the button pressing made by calling function from this library: firstly open the desired GPIO chip by calling one of the gpiod_chip_open functions returning a gpiod_chip structure which is used by subsequent API calls, then open the desired GPIO line by calling gpiod_chip_get_line() obtaining, in this way, a gpiod_line struct. The next step is to request the use of the line as an input/output with gpiod_line_request_input() or gpiod_line_request_output(), then just read the value of an input by calling gpiod_line_get_value() and at the end release the lines by calling gpiod_line_release() and chips by calling gpiod_chip_close().

<termios.h> and *<fcntl.h> are* used to describe a general terminal interface that is provided to control asynchronous communications ports like the serial ports. Many of its functions have a termios_p argument that is a pointer to a *termios* structure. The mentioned structure contains all parameters of the serial ports and to access it the syntax is *tty.c_cflag*. They also permit to choose if the device has to be opened for reading only, writing only, or both reading and writing. The device should be opened in blocking or non-blocking I/O. The user can also choose if the device has to be opened in exclusive mode so other programs cannot access the device once opened. For this project the device is opened for reading only because it does not transmit nothing through the serial port. This library permits also to set the device path and the file handle where, in case of error, "-1" is written.

In the second category appear all libraries needed to make complete the use of C++ on Linux and they are listed in figure 30.

```
#include <iostream>
#include <chrono>
#include <string>
#include <ctime>
#include <unistd.h>
#include <thread>
#include <cstdlib>
#include <stdlib.h>
#include <fstream>
#include <sys/statvfs.h>
#include <errno.h>
#include <stdio.h>
```

Figure 30. General system libraries

Many of them are used for common commands such as print on console, management of text files, use string operators, define threads and time.

5.6. Function prototypes and descriptions

The code is based on threads and consequently on functions. Each method has the task of returning or modifying a variable so that it can be exploited by the rest of the code, void functions just modify a global variable and functions of certain type return a precise variable. Starting from the methods that concern the main block of the code, there are:

- **int button_pressed()**: it is of integer type because the function returns an integer variable named button_status which can assume two values, 1 if the button is pressed or 0 if it is not pressed. Its body is enclosed within a while cycle and it contains the code to open, set and close the gpiochip that control the physical button.
- **void loop_rec()**: it records in loop the videos by webcams inside a while cycle. The function contains many if conditions to check which webcams are enabled and what type they are. Inside these conditions there are the definition of the block segments and the *ffmpeg* command to record videos.
- **void save_CAMs(int &result)**: firstly it takes as argument the value of the button status provided by the button_pressed() function. An important check is done at the beginning, in fact, if user presses the button while it was already pressed, the code will ignore this action. After that, there is a check of the available space on SDcard or pendrive and if we have a positive response, then the algorithm checks in which segment of video it is and it starts to save the videos also making some mux and demux operations by *ffmpeg* command line. At the end, after some operations, using complex filter and *concat* instructions, the final video is saved on output folder.
- **void loop_CAN0()**: this function has the task of starting the acquisition of the CAN0 line by *candump* command and saving files into a temporary folder. In the following chapter will be discussed how the system is configured to support CAN features. Obviously, the same operations are executed on the channel 1 of the CAN.

- **void CAN0_save(int &result)**: this method does some preliminary checks about available space on SDcard and then it preforms some changes on the recorded file that come from the loop_CAN0() function. This changes concern substituting and swapping columns, removing brackets and filtering based on timestamps, in order to fulfil synchronization requirements. These modifications are needed because the output files must be elaborated into CANalyzer tool which need a specific format input file.
- **bool configure_serial_port(int device_number, int baudrate, int &serial_port)**: the tasks of this method is to configure all parameters for ensuring the correct behaviour of the serial ports. Here are configured different field to establish correct serial communication between transmitter and receiver. In this phase some error checks are performed in order to cover all possible issues.
- **void loop_serial1()**: it does many things for the serial port features. It firstly initializes a 256-char buffer, it allocates to memory that buffer, setting the first value to 0 and using a pointer to the array. Then, after opening a text file to save the serial outputs, it writes into the file being careful in writing also the time stamp of the reading. At the end, the file and the port are closed. The same function is written for the second serial port.
- **void serial1_save()**: here time plays a key role because, since all saving functions are synchronized, it must know when start to keep serial lines from the trigger. Thanks to *awk* command line instruction it is possible to filter the received messages by a set time, knowing the interested interval. At the end inside the pre-processed temporary fiile is put and converted into the serial final file where there will be the final cropped output received messages. The same thing is done for the second port.

The other functions are used to ensure the correct functioning of the code and the behaviour of the peripheral methods. Because most of the following functions are used inside peripheral blocks.

- **string convertSecondsToHHMMSS(int value)**: this function just converts an integer, passed as argument, into HH:MM:SS format to use this format inside other functions.
- **int get_int64_value_from_ascii_string(string source_string, int char_index, int64_t *result)**: it takes and convert a string coming from the command line into an integer, to make it readable. The result of the check is saved in the last field as a reference.
- **string check_available_space(char* command)**: it is the first field of the previous functions, so the command string.

5.7. Operating system management

In this chapter there will be described how the environment was configured. We will go deeply into detail on the toolchain I created to operate the main system, from the installation of the library to the use of the specific command. Another important thing to view is the organization of the desktop. Starting from the Operating System, the written code is based on Debian O.S also known as Debian GNU/Linux, which is a Linux distribution composed of free and open-source software, developed by the community-supported Debian Project. Debian is one of the oldest operating systems based on the Linux kernel. The project is coordinated over the Internet by a team of volunteers and new distributions are updated continuously by them. To write and debug the code I used Visual Studio Code for Linux, where the code can be compiled and run in real time, with the possibility to see the output on the console space.

The configuration of Visual Studio is a critical part because to compile and run correctly che code, it need its C++ language packet installed and also the *CMake Tool*. *CMake* is an open-source, cross-platform tool that uses compiler and platform independent configuration files to generate native build tool files specific to your compiler and platform. The CMake Tools extension integrates Visual Studio Code and CMake to make it easy to configure, build, and debug the project. Once all components are installed, it is necessary to create a CMakeLists.txt file which contains

all instruction useful to the compiler to link used libraries to the code. An example is shown in figure 31.

```
cmake_minimum_required(VERSION 3.5)

project(LoggerDevice)

SET(CMAKE_CXX_FLAGS ${CMAKE_CXX_FLAGS} "-std=c++11 -pthread")
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)
set(CMAKE_EXE_LINKER_FLAGS "-lgpiod")

find_package(Threads REQUIRED)
add_executable(LoggerDevice main.cpp )
```

Figure 31. CMakeLists.txt file

As you can see, the writing of the code is done within the same device. So, there is no need of external notebook equipped with programmable cable because the software is directly written and tested on the O.S of the Raspberry Pi. Every time the code is compiled, Visual Studio makes a check of *CMakeLists.txt* to ensure that all requirements are fulfilled.

For what concern the management of the files, there was created many folders in order to contain, step by step, the peripheral files during the saving phase. The folders are visible in figure 32.

Figure 32. Folders for saving phase

There are three main stages in total from the start to the end of the saving phase:

- **Stage 1**: in the first step, after all peripherals are configured, all files coming from the vehicle are stored inside three specific folders: Temp, CAN_rec, serial_rec. These folders are in desktop. So, after a few seconds from the start, all received messages and recorded videos will be saved on these three folders. The files inside these folders are never deleted because, following the logic of the code, they are overwritten every cycle.
- **Stage 2**: after the tester have pressed the button to start saving data, all files that are contained in these folders are picked up, copied into the New, CAN_files and serial_files folders and processed. “Processed” means that specific functions for each peripheral elaborates the file inside the folder and modifies them in a suitable way in order to be ready for the final step. On this stage, the files are temporary and they are deleted after the saving operation is completed. In this way we can save space on disk.
- **Stage 3**: at the end of the process all files are moved to the Output folder, which is their final destination. On this stage all output files are enumerated because for each peripheral type there is a general name and so, every saving phase the code attaches an incremental number to each output file. It does this operation by using an integrated counter inside saving functions.

All these steps can be performed also if, inside the setting file, the tester configures an external pendrive as default saving space. In that case the files will be saved on the plugged-in USB.

The data logger device is designed to work autonomously, so no need to be connected to the monitor, keyboard and mouse. When it is used on-board, the code must start when the device is powered on and program need the operating system to work properly. So, once the device was tested and works correctly, it is configured in such a way that software is executed automatically. This operation is done by creating a bash file called *boot.sh* with the necessary instructions and adding a line before the end of "exit 0" in the *rc.local* file, located in /etc linux directory. These system files are shown of figure 33 and 34. We have to pay attention to a particular though, before starting the code, because we have to wait 10 seconds so that all the libraries and drivers of the operating system are fully loaded. Otherwise, the entire system will freeze. What is added on the rc.local file is a system default script executed at the end of each multiuser runlevel. By default, it does nothing, but in this case, adding *sudo sh /home/pi/Desktop/boot.sh* line, it executes *boot.sh* file with root permission. An important thing to say is that all operations concerning modification of system directory files need a special permission with the *sudo* command. In bash files the "#" char indicates a comment, so these lines will be ignored. The core lines for self boot of the code are contained in *boot.sh* script shown in figure 34, which makes the system waits 10 seconds before running the code contained in the directory written in the line below and in the last line there is the command "./" to start the executable.

```
#!/bin/sh -e
#
# rc.local
#
# This script is executed at the end of each multiuser runlevel.
# Make sure that the script will "exit 0" on success or any other
# value on error.
#
# In order to enable or disable this script just change the execution
# bits.
#
# By default this script does nothing.

# Print the IP address
_IP=$(hostname -I) || true
if [ "$_IP" ]; then
  printf "My IP address is %s\n" "$_IP"
fi

# sudo sh /home/pi/Desktop/boot.sh

exit 0
```

Figure 33. rc.local file for system self booting

Bash is a "Unix shell", meaning a command line interface that allows interaction with operating systems derived from Unix. It is available on most modern operating systems of this type, being the default shell for many GNU/Linux distributions. Its importance is also linked to the fact that it has been used on many other systems, not necessarily derived from Unix.

```
#!/bin/bash

sleep 10

cd /home/pi/Desktop/newtest_last/newtest/

sudo ./newtest
```

Figure 34. boot.sh script for system self booting

In order to use the libraries described in the previous chapter, the operating system needs some software packages installed on Linux. For example, to use the main libraries for videos, gpio and CAN I used the command line to download and install directly from the web-source the complete repositories:

- **ffmpeg command**: git clone https://git.ffmpeg.org/ffmpeg.git ffmpeg
- **gpio command**: sudo apt-get install gpiod libgpiod-dev libgpiod-doc
- **CAN-shield command**: git clone https://github.com/seeed-Studio/pi-hats

In particular, to use the CAN-shield features, a special system file must be modified adding a new line at the end by this command: *sudo nano /boot/config.txt* and the line is *dtoverlay=2xMCP2517FD-can*. In this way the the operating system can recognize the new installed component.

6. EXPERIMENTAL RESULTS

6.1. Bench tests

Before going inside the vehicle, various tests are performed from the start to the end of the project. Each test is intended to test the correct functioning of the program and for each block of written code a bench test has been done. A first approach was to bring a little piece of code into a separate space (e.g. command shell or temporary new project inside the IDE) to try its own functionality. After that, it was put and melted with the rest of the code. A common strategy used also here, is that print some useful variables to make you understand what is happening inside a specific function. So that, we have a readable output which can report the occurrence of possible anomalies. These *cout* lines can be commented and de-commented depending on the situation. For example, for the video blocks it is used different *cout* instruction to print all information about the chosen video settings and also some useful variable for debug, as shown in figure 35.

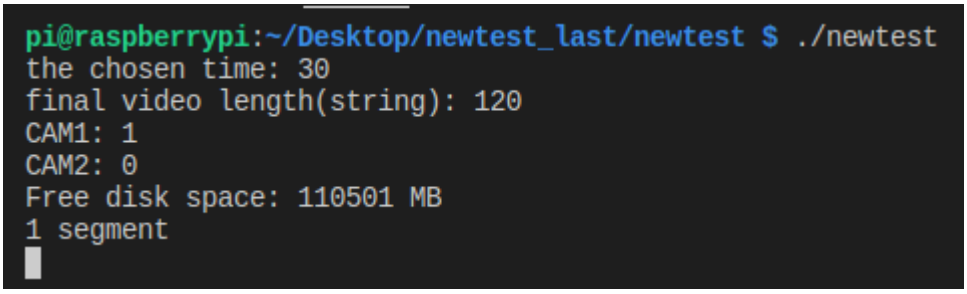


Figure 35. Printed information about video functions

The same speech is made for the CAN blocks, printing different parameters and variables as it is reported in figure 36.

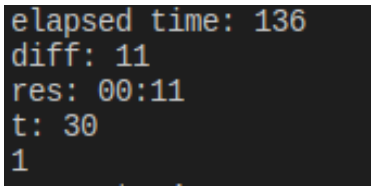


Figure 36. Printed information about CAN functions

In this way, for each simulation, you can find any errors or things do not go as they should. The software may give wrong results and to understand where the error is it is very useful to have some printed data to review instantly. Another important advantage of this technique is given by the addiction of *cout* line inside each thread where is printed a sort of phrase flag like “this is thread of video block”. By proceeding like this, it is very simple to recognize in which part of the code we are.

The bench used for internal tests simulates accurately the real behaviour of the vehicle and it is illustrated in figure 37, since it includes the control unit, the radio, all the CAN-bus lines and the on-board instrumentation. So, once the device OBD is connected to the bench, all tests about CAN lines can be performed in order to verify if the data logger works properly.

Figure 37. Maserati bench test

Since Maserati vehicles do not output serial messages from USB radio, it was needed to create a specific tool, write in C#, which send messages at a given baudrate to the Raspberry through the USB FTDI cable. So that, all peripherals present on vehicle are simulated faithfully by hardware and software on laboratory.

Not only the code is verified during the test phase, in fact, many tries were done also on a single component such as LED and button. For these tests I used a multimeter and a bench power supplier. More specifically the multimeter was used to check the connections between the various pins, especially after soldering because it is very important that all welds are done correctly and that all current and voltage of the components have the right values.

On the other hand, a bench power supplier was needed to test individually some single components such as LED and cooling fan. For example, giving an increasing current or voltage to the components to study their behaviour. On the figure 38 and 39 there are multimeter and bench power supplier used for these types of tests.

Figure 38. Digital multimeter

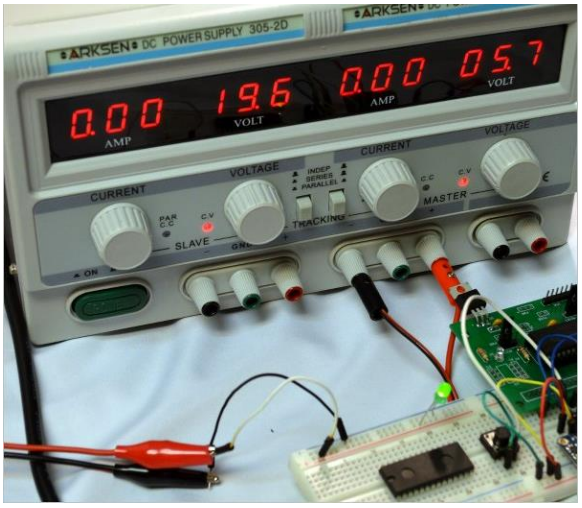


Figure 39. Bench power supplier

BENCH TESTS	GOAL	CONDITIONS	METER OF JUDGEMENT
Vehicle simulation log test.	Test, by simulation, if all required logs are acquired correctly.	Inside the laboratory, with device connected on test bench.	All logs must be complete and fully usable by software tools of company.
SW console output check.	Check if the software simulation gives correct console outputs, needed for future debug.	Inside the laboratory, with device connected on test bench.	On the console must be printed the right values of wanted code lines.
Compatibility test between CAN logs and company tool.	Check if the acquired CAN logs are suitable for CANalyze format tool.	Inside the laboratory, with CANalyzer tool and files.asc.	All logs must be loaded and read correctly by the tool.
Current and Voltage test.	Check if hardware can manage the current and voltage quantity of the input/output pins.	Inside the laboratory, picking single components.	Measured values must be compliant with respect to the datasheet specifications.

Figure 40. Table of performed bench tests

All tests present in the table, on figure 40 have given a positive result and they respect all the initial specifics illustrated in the first chapters.

6.2. In-vehicle tests

The first stage for the in-vehicle tests is the cable management, because it has to be done in a such way that it does not interfere with the tester while he is driving. The longest cable is the one for the OBD connection and it starts close to the brake pedal and it is passed behind the steering wheel. The final part of the cable is connected to the DB9 of the main device, which is placed on the right of the driver, more or less, on the center of the vehicle, as depicted in figure 41 and 42. Since there were some problems at the start of the tests in the car, to debug inside the vehicle was used an external display with mouse and keyboard to run all the debugging accessing the real components of the car. With the display it was possible to monitor the working temperature of the device, which is higher than the temperature it has inside the laboratory. This aspect is very important because, as it is specified in the hardware chapter, it has not to exceed the 50°C to work correctly inside the standard range.

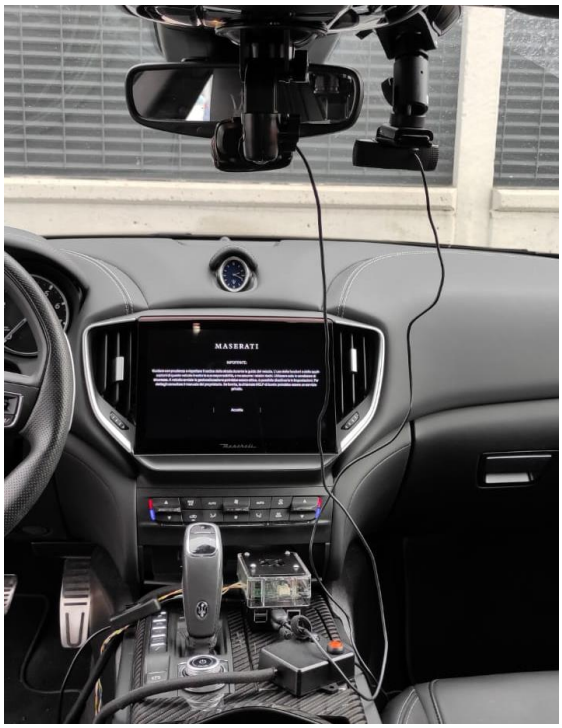


Figure 41. In-vehicle data logger configuration

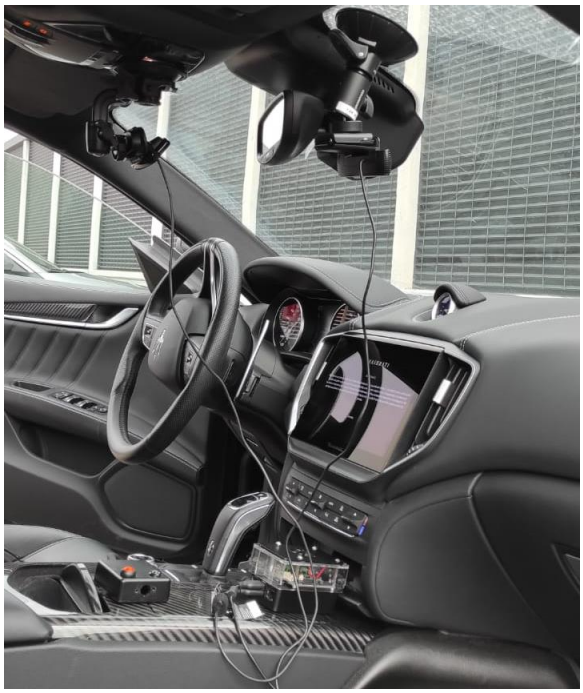


Figure 42. Side view of in-vehicle data logger configuration

In-vehicle tests were performed inside an open area driving M182 vehicle model. All results are shown in the following figures 43, 44, 45 and 46.

```
base hex timestamps absolute
000.000497 CANFD 2 RX 0FF        1 0 d 48 00 00 07 C0 01 EC 02 E4 01 F0 01 EC 01 EC 01 F0 00 00 00 00
000.002687 CANFD 2 RX 240        1 0 d 04 08 00 0C 70 000000 000 323000 00000000 50140850 50140250 2007030e
000.004483 CANFD 2 RX 118        1 0 d 32 01 00 00 00 00 00 00 00 00 00 00 60 00 00 FE 00 00 00 00 07 CF
000.005301 CANFD 2 RX 11C        1 0 d 64 00 00 00 00 02 00 08 F0 00 03 00 00 00 0F FC 00 00 00 00 00 00 00
000.005670 CANFD 2 RX 11A        1 0 d 64 FF FF FF FF FF FF 7F FF 7F FF 7F FF 7F FF 7F FF 00 00 00 FE 3F FE
000.006097 CANFD 2 RX 1DC        1 0 d 32 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000.006353 CANFD 2 RX 65A        1 0 d 32 FF FF FF FF FF FF FF FF FF FF C2 A6 7F FF FF FF FF FF FF FF FF FF
000.006764 CANFD 2 RX 102        1 0 d 64 00 00 0F FF 88 00 30 00 9F FF 80 00 01 F4 FC 00 64 00 00 00 00 00
000.006965 CANFD 2 RX 108        1 0 d 32 2B 7E 27 D0 17 CF 04 04 03 FF 32 20 00 00 AD F8 00 00 00 00 00 00
000.007185 CANFD 2 RX 103        1 0 d 32 00 00 00 00 00 00 00 00 00 00 03 FF 00 FF 80 FF 00 00 AC A2 65 0F
000.007726 CANFD 2 RX 116        1 0 d 64 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 D0 00 04 1F
000.007921 CANFD 2 RX 113        1 0 d 32 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03 FF
000.008156 CANFD 2 RX 201        1 0 d 32 00 00 00 00 18 00 00 00 00 00 00 00 00 20 18 00 00 00 00 00 00 00
000.008464 CANFD 2 RX 100        1 0 d 32 00 00 00 00 00 00 00 00 00 00 08 00 00 00 04 00 00 00 00 00 00 00
000.008696 CANFD 2 RX 2EE        1 0 d 32 00 00 00 00 00 00 03 23 91 00 00 00 00 00 00 00 00 00 00 00 00 00
000.008926 CANFD 2 RX 597        1 0 d 32 00 00 00 00 00 91 00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00
000.009428 CANFD 2 RX 402        1 0 d 32 00 00 C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000.009994 CANFD 2 RX 106        1 0 d 64 03 20 03 1F 03 20 40 00 00 00 80 00 00 00 04 17 00 00 61 80 00 30
000.010191 CANFD 2 RX 206        1 0 d 32 00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 00 00 30 00 00 00 00
000.010517 CANFD 2 RX 0FF        1 0 d 48 00 00 07 C0 01 EC 02 E5 01 F0 01 EC 01 EC 01 EC 01 F0 00 00 00 00
```

Figure 43. Extract of CAN FD file from vehicle

```
base hex timestamps absolute
000.000721 1 1E340017 RX d 2  00 00
000.001694 1 50C RX d 8  00 00 08 0B 00 00 01 00
000.002379 1 58E RX d 4  08 00 0C 70
000.003081 1 1E34009B RX d 2  00 1E
000.004070 1 0FA RX d 8  00 01 E0 00 00 00 06 7F
000.005028 1 2EB RX d 8  01 00 7F FE 01 EC 00 00
000.005829 1 51B RX d 6  00 06 F0 00 F0 00
000.006820 1 3DB RX d 8  00 00 00 00 00 00 00 00
000.007524 1 1E34008B RX d 2  00 1E
000.008241 1 1E34009C RX d 2  00 1E
000.009183 1 3DE RX d 8  65 0F AC A2 00 00 00 00
000.010961 1 50C RX d 8  00 00 08 0A 00 00 01 00
000.012940 1 0A0 RX d 2  80 00
000.013919 1 3E2 RX d 8  88 0D E4 00 0C 00 02 02
000.017506 1 1E340032 RX d 2  00 0E
000.018615 1 3E6 RX d 8  10 00 00 00 63 E0 91 00
000.021098 1 50C RX d 8  00 00 08 0B 00 00 01 00
000.023351 1 3E0 RX d 8  01 42 30 30 37 33 36 36
```

Figure 44. Extract of CAN BH file from vehicle

Figure 45. Internal webcam view of vehicle

Figure 46. External webcam view of vehicle

From figures 45 and 46 it is appreciable that both views are detailed and clearly interpretable by the tester. Together with the rest of the logs are very useful for finding problems and the device can be exploited for a lot of purpose since it is fully configurable and customizable.

IN-VEHICLE TESTS	GOAL	CONDITIONS	METER OF JUDGEMENT
HW stress test.	Test the solidity and robustness of the device.	Inside the vehicle, during driving without any anchorage.	Device must remain compact and it must work correctly during all test.
SW stress test.	Test the reliability of the software with repeated and close-up acquisition tests.	Inside the vehicle, during driving without any anchorage.	Device must acquire all set peripheral logs.

Comfort test.	Test the comfort and the ease of use of the system.	Inside the vehicle, during driving without any anchorage.	Driver must be able to perform all tests autonomously without any difficulty.

Figure 47. Table of performed in-vehicle tests

All tests present in the table, in figure 47 have given a positive result and they respect all the initial specifics illustrated in the first chapters.

The obtained files from the data logger displayed in figures 43, 44, 45 and 46 are very good and they are ready to be analysed so, the device works correctly as expected, giving correct data. This is an important point of arrival for the prototype because it is already usable for true tests.

7. VALIDATION

The concept of validation has been in vogue for centuries. Irrespective of the industry or products, validation ensures various critical aspects of a product and guarantees its success in the market as well as among the users.

7.1. Software validation

During the software development life cycle (SDLC), software validation plays a critical role in assisting the testing and development teams in producing a high-quality product. Software validation is the process of analysing a software product to verify that it fulfils pre-defined and stated business criteria, as well as the demands and expectations of end users/customers. It is mostly carried out with the goal of determining if the created software is designed according to pre-determined software requirement specifications (SRS) and if it meets the customers' actual demands in the real world.

On this software are performed usability tests and collection of driver feedbacks and since all answers are positive, it was defined that the device meets all initial requirements. Four of the most important things to consider software validated are:

- Software working correctly during all in-vehicle usage.
- Full compatibility between CAN files and CANalyzer tool.
- Clear videos where it can be possible to distinguish details.
- Good format of serial log to be integrated with the rest of the files.

All files given by the data logger were tested and interpreted and the results is that each of these four points is approved.

7.2. Hardware validation

Hardware validation proves that the specifications have created a product that meets the client requirements and needs and that the design and a correctly assembled system meet the specifications. So that, the device must:

- Works in real time.
- Be small and portable.
- Have synchronized log peripherals.
- Be suitable and integrable with company software (CANalyzer) and hardware (DB9 and OBD connectors).

The data logger works in real time because there is no latency between the actions of the driver and the response of the device.

The data logger is small and portable due to the dimension of the used board (Raspberry Pi), the precise size of the hardware board is depicted in the chapter 3.

The acquired logs are perfectly suitable and integrable with the company tools and hardware since they were tested in the laboratory by infotainment team.

On this hardware are performed usability tests and collection of driver feedbacks (e.g. comfort of position of the button, webcams and data logger) and since all answers are positive, it was defined that the device meets all initial requirements. Three of the most important things to consider hardware validated are:

- Hardware works correctly during all in-vehicle usage.
- Stability and robustness of the device located in-vechile.
- All single components have to respect the initial functional requirements.

Tests performed on the laboratory give some important results. Each critical component was tested with bench instrumentation, analysing current, voltage and temperature. All of them works in a proper range of functioning, so there will not be any electrical or mechanical issues on the board.

8. FUTURE STAGES

The next step for the project is to develop a Graphic Unit Interface using C# language. This will add an important feature to the software because instead of writing the settings manually using the text file, the user can select the various settings from a much more intuitive and fast graphic form. Even in terms of safety there are advantages because there is no risk of making mistakes by writing the configuration by hand.

Another step will be the introduction of an error code table where all types of errors that may occur during code execution are grouped and classified by number range, which identifies the different type of categories as it is shown in figure 48.

ERROR CODE	CATEGORY	DESCRIPTION
[1-100]	Storing	Issue during the video, serial log, CAN log, device log in SD or Pendrive
[200-299]	Settings	Setting file missed or corrupted
[300-399]	Generic	Generic system error
[400-499]	HW	Peripherals not recognized
[500-599]	Exceptions	Code exceptions

Figure 48. Error code table

In the future will be produced in series the final version of the prototype presented in this thesis. The figure 49 illustrates the render of the final PCB board. It has four layers due to the density of the main connectors and it is smaller, lighter and without some useless components loaded by default on the Raspberry Pi used for the prototype. In this way the company will save al lot of money on the production stage because it will

cost less than a Raspberry Pi board and this PCB can be enclosed in a very compact case, thus becoming very ergonomic and easy to handle.

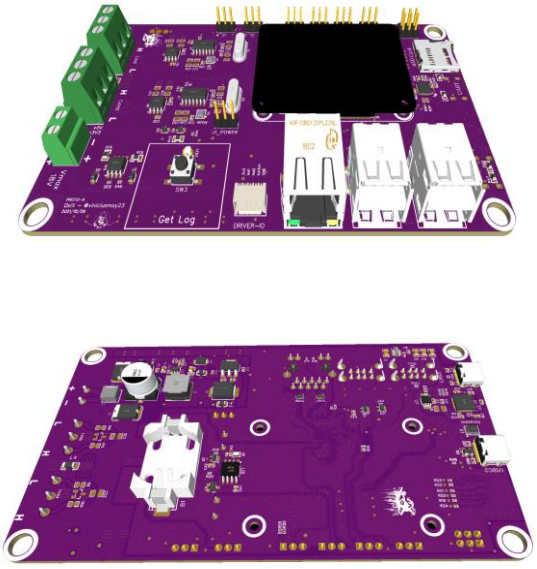


Figure 49. Final PCB render

9. CONCLUSION

This project aims to provide an alternative to current data logging systems. Until now, testers have used cumbersome and impractical devices to run logs inside the vehicle and very often do not have all the peripherals available in one device. During the tests they had to be accompanied by a second person who helped him during the acquisition. In this thesis the whole process of realization of the data logger has been shown, from the initial idea to the realization of the working final prototype. During the tests the engineers were able to try all the features it offers, experiencing several benefits including compactness, ease of use, the ability to log without the need to have a second person next door. One of the aspects that give more value to the device is the possibility of having more videos synchronized with the rest of the logs, in fact this function is not present in the other logger devices and was very appreciated during the tests on the road. The built prototype will be used in the company in vehicle tests until the production of the final PCB is started.

BIBLIOGRAPHY

- https://www.marketsandmarkets.com/Market-Reports/automotive-data-logger-market-88581029.html
- https://en.wikipedia.org/wiki/Data_logger
- https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/
- https://www.seeedstudio.com/2-Channel-CAN-BUS-FD-Shield-for-Raspberry-Pi-p-4072.html
- https://www.ffmpeg.org/
- https://www.ftdichip.com/Support/Documents/DataSheets/Cables/DS_USB_RS422_CABLES.pdf
- https://en.wikipedia.org/wiki/Thread_(computing)
- https://www.ics.com/blog/gpio-programming-exploring-libgpiod-library
- https://en.wikibooks.org/wiki/Serial_Programming/termios
- https://en.wikipedia.org/wiki/Debian
- https://www.esperimentanda.com/come-scegliere-un-alimentatore-da-banco-o-laboratorio-stabilizzato-duale-switching-migliore/
- https://www.typhoon-hil.com/documentation/typhoon-hil-software-manual/References/can_bus_protocol.html

ACKNOWLEDGEMENTS

I want to thank my family for all the support they have given to me in these years of study, my friends for helping me and all people who have believed in me. But I also want to thank myself for for making it and for never giving up.