**AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE**

**WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI, INFORMATYKI I INŻYNIERII BIOMEDYCZNEJ**

KATEDRA METROLOGII I ELEKTRONIKI

Projekt dyplomowy

# *A data acquisition system from the CAN bus and a GPS module using non-volatile mass storage.*

# *System akwizycji danych z magistrali CAN oraz modułu GPS wykorzystujący nieulotną pamięć masową.*

| | |
|---|---|
| Autor: | *Michał Kowalik* |
| Kierunek studiów: | *Mikroelektronika w Technice i Medycynie* |
| Opiekun pracy: | *dr inż. Piotr Otfinowski* |

Kraków, 2020

*Uprzedzony o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz.U. z 2018 r. poz. 1191, z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystycznego wykonania albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.", a także uprzedzony o odpowiedzialności dyscyplinarnej na podstawie art. 307 ust. 1 ustawy z dnia 20 lipca 2018 r. Prawo o szkolnictwie wyższym i nauce (t.j. Dz. U. z 2018 r. poz. 1668, z późn. zm.): „Student podlega odpowiedzialności dyscyplinarnej za naruszenie przepisów obowiązujących w uczelni oraz za czyn uchybiający godności studenta." niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.*

# Contents

# Abstract

The goal of this thesis is to present process of design, firmware and software implementation and tests of data acquisition system prepared to log data from CAN Bus and GNSS (including GPS) system using microSD card as storage mass. Logger is designed to be used in Formula Student racing car as a part of *AGH Racing Team* project.

## Acronyms

| | | |
|---|---|---|
| ACK | - | acknowledge |
| ADC | - | analog-digital converter |
| API | - | application programming interface |
| CAN | - | Controller Area Network |
| CSV | - | comma-separated values (file format) |
| DLC | - | data length code |
| DLL | - | dynamic link library |
| DMA | - | direct memory access |
| ECU | - | electronic control unit |
| ESD | - | electrostatic Discharge |
| FAT | - | file allocation table |
| FIFO | - | first in, first out (queue) |
| GND | - | **g**rou**nd**, reference point in an electrical circuit from which voltages are measured |
| GNSS | - | Global Navigation Satellite Systems |
| GPS | - | Global Positioning System |
| GUI | - | graphical user interface |
| HAL | - | hardware abstraction layer |
| ID | - | identifier |
| IDE | - | integrated development environment |
| ISR | - | interrupt service routine |
| Kb/s | - | kilo**bits** per second |
| KB/s | - | kilo**bytes** per second |
| LED | - | light-emitting diode |
| LiPo | - | lithium polymer battery |
| Mb/s | - | mega**bits** per second |
| MB/s | - | mega**bytes** per second |
| MCU | - | microcontroller unit |
| NMEA | - | National Marine Electronics Association |
| PC | - | personal computer |
| PCB | - | printed circuit board |
| PMU | - | power management unit |
| RPM | - | revolutions per minute |
| RTC | - | real-time clock |
| SD | - | secure disk |
| SMA | - | subminiature version A (connector) |
| SPI | - | serial peripheral interface |
| UART | - | universal asynchronous receiver-transmitter |

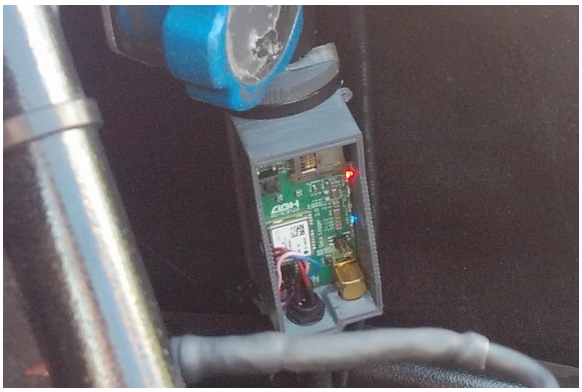
# 1. Introduction

The **Data Logger** described in this thesis is the device used in **racing cars** being built by AGH Racing Team.

*AGH Racing Team* is taking part in Formula Student competitions - Europe's most established educational engineering competition [1]. Aim of this contest is to build, test and race a single-seater formula racing car.

Car before being allowed to compete must pass scrutineering check to proof that is safe and comply with the regulations. After that, competition consists of dynamic and static events including **Design Event** which concept is to evaluate the engineering effort that went into the design of the car, prove that good engineering practises was used during design and manufacturing and present vehicle performance and overall value [2].

(a) AGH Racing combustion car during competition in the Netherlands, 2018 by KSAF AGH.



(b) Data Logger mounted in the car during tests with open case cover.

**Figure 1.1.** AGH Racing combustion car and Data Logger on the track

To be able to check performance and reliability of the car in the dynamic part of competition it is necessary to **collect measurements from sensors installed in the car during drive** (including race). This data is also needed to **validate** mechanical **simulations** (suspension, frame, engine, etc.) by **compering simulation results to the real data** from test drives. Natural solution to this problem is installing data logging system in the car, which should be easy-to-use also for engineers responsible for non-electronic parts of the car.

Because multiplicity of sensors in the car and the requirement of lightness of complete system caused necessity to introduce the approach used generally in automotive industry [3]: *CAN Bus* with nodes measuring data from chosen sensors and transferring them in digital form. Second main reason to choose the *CAN Bus* was easy integration with third-party elements of the system: *ECUMASTER EMU Black* (Engine Management Unit) and *ECUMASTER PMU* (Power Management Unit) containing built-in *CAN Bus* interface. [4]

## 1.1. Electronics system architecture

In this section simplified electronics system architecture is described. It includes key features for the project being described in this thesis.

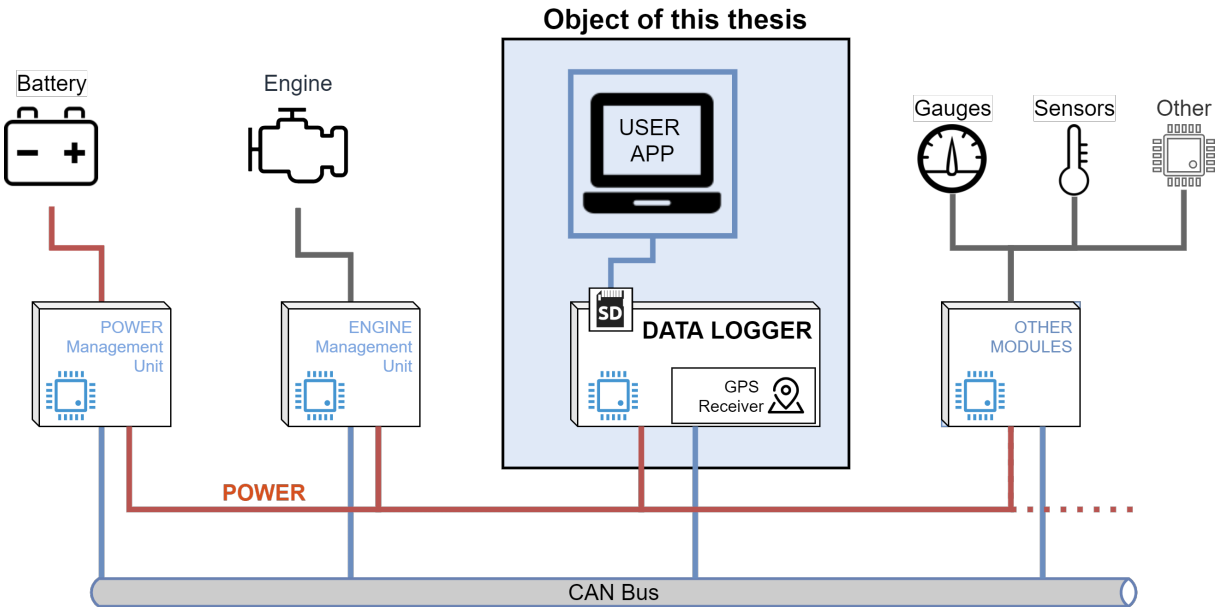



**Figure 1.2.** Electronic system architecture schema

All electronic units in the car are connected with *CAN Bus* distributed in the whole car. System consists of modules, in example:

- ECUMASTER EMU - Engine *Electronic Control Unit* (ECU) connected to all engine critical sensors
- ECUMASTER PMU - *Power Management Unit* (PMU)
- **Data Logger (described in this thesis)**
- Dashboard with gear display, Revolutions per minute (RPM) LEDs and alert LEDs
- Nodes collecting data from sensors
- etc...

Measurements from each of mentioned modules are transmitted via *CAN Bus* and (if configured) logged by described here data logger.

## 1.2. Goal of this thesis

The objective of this work is to design, implement and manufacture the data acquisition system, which allows logging data from *CAN Bus* and global localisation. System consists of two elements: embedded device (Data Logger) and User PC application.

Data Logger embedded device is designed to receive and store data frames from the *CAN Bus* (including Bus errors) and the GNSS System (Global Navigation Satellite Systems) to the defined binary file. Data Logger should be user configurable. Configuration should contain Can BUS baudrate, GNSS data frequency, CAN frames IDs to be saved/ignored and definition of signals (described in 4.1 chapter) within those frames.

PC application aims to convert data saved by the *Data Logger* to standard **Comma-separated value (CSV)** file format and provide user interface to prepare configuration for the logger.

## 1.3. Scope of work

To achieve goals of this work it is required to challenge design problems described below. Those includes process of design, manufacturing and tests of the device, firmware and software.

- Choosing microcontroller, components and modules
- Design of Data Logger PCB board
- Design of the case
- Manufacturing and start-up of the device
- Defining configuration file format
- Defining data log file format
- Implementation of Data Logger firmware
- Implementation of PC Software
- Debug and tests of the system

# 2. Used technologies

This chapter introduces to used technologies, protocols, etc.

## 2.1. CAN Bus

*CAN Bus* (next to GNSS data) is one of two sources delivering data to the logging system. Architecture of this protocol caused many choices in the design of the logger, including:

- format of configuration and logged data files,
- data storage size with a proper record speed,
- hardware design

Due to all of the above, it is crucial to understand key features (important due to design decisions) of this protocol.

### 2.1.1. Protocol introduction

CAN Bus (**C**ontroller **A**rea **N**etwork) is the most popular digital data transmission system in modern cars [3]. It was designed originally for automotive industry at *Robert Bosh GmbH* in 1986 [5]. Because of multitude of electronic control units in modern automobiles (over 13 in *AGH Racing* cars and even more than 70 in road cars) using this type of communication reduces number of direct-wire connections, simplifies electric bundle and gives easy way for extensions and modifications of transceiving data. Debugging any problems becomes easier (when possessing proper equipment and documentation).

In CAN Bus protocol there is no host - all nodes are equally important. CAN is message-based protocol, so messages are not send for particular recipients but with particular settled data distinguished by frame ID.

CAN Bus uses two wires (CANH[igh] and CANL[ow]) to send data using differential signal.

### 2.1.2. Frame format

CAN protocol defines precisely frame format sent on the bus. It is possible to use two frame formats: **standard** frame format, described by CAN 2.0A and CAN 2.0B standards, and **extended** frame format described only by CAN 2.0B. The difference between this two standards is the length of frame identifier

(ID) field described below. Standard frame contains identifier with **11-bit** length, while extended identifier has **29 bits**. Actually in automotive extended format is rarely used, so **for the purpose of this work it is assumed, that identifier (ID) has length of 11 bits**.

CAN defines four frame types: **data frame**, remote frame, **error frame** and overload frame. Remote frame and overload frame are not used in this project so will be omitted in the description.

#### 2.1.2.1. Data frame

**Data frame** is mainly used type of frame. Is is used for actual data transmission. Contains 44 to 108 bits. Below are described names, lengths and purposes of each bit:

| Field name | Length (bits) | Purpose |
|---|---|---|
| Start of frame | 1 | 1-bit with dominant level. Starts the transmission. |
| Identifier (ID) | 11 | Identifier unique for each message (not node!), also priority of the message |
| Remote transmission request (RTR) | 1 | Determines type of frame: dominant for data frame; recessive for remote request frame |
| Identifier extension bit (IDE) | 1 | Must be dominant for standard frames (11-bit ID) |
| Reserved bit (r0) | 1 | Reserved bit. Must be dominant. Recessive used for CAN FD protocol[1] |
| Data length code (DLC) | 4 | Number of bytes in Data Field. Value from 0 to 8 (values from 9 to 15 are treated as equal to 8) |
| Data field | 0 - 64 | Data aimed to be transmitted. Length in bytes specified in DLC field. |
| CRC | 15 | Cyclic redundancy check - error-detecting code detecting changes to row data. |
| CRC delimiter | 1 | Must be recessive. |
| ACK slot | 1 | Transmitter sets bus line to recessive state and receiver in this time sets line to dominant state what is detected by transmitter, so this acknowledges that message was correctly received. |
| ACK delimiter | 1 | Must be recessive. |
| End of frame (EOF) | 7 | Must be recessive. |
| Inter-frame space | 3+ | Must be recessive. |

[1] CAN FD (CAN with **F**lexible **D**ata-Rate) - extension to (described here) CAN protocol having backward compatibility with standard CAN 2.0 networks. For data field baud-rate is increased from 1Mbit/s to even 8Mbit/s and length of data field from 8 bytes to 64 bytes.

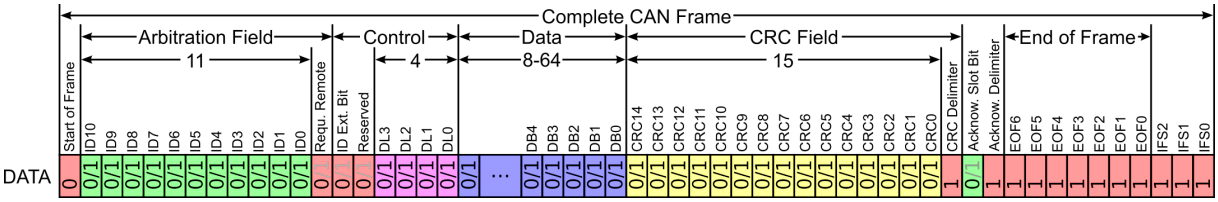



**Figure 2.1.** CAN Bus frame visualization. Based on the [6]

### 2.1.3. Data integration and errors detection

CAN Bus is designed to be used in vehicles, so data integration and errors detection is crucial for the safety of passengers. That implies that receiving no data is more beneficial than receiving tampered.

That is the reason why any frame can (or even should) be destroyed by any other node which finds out that frame integration is lost. It is implemented by sending **Error Frame** during data frame transmission. It interrupts actual transmission not letting corrupted frame to be received by any other node.

**Error types and error frames are logged** by described system.

### 2.1.4. Clock transferring, bit stuffing

Because in CAN bus standard there is no extra line for transferring clock and frame is very long (even 108 bodes) it is necessary to provide clock synchronization. Without extra clock line **bit-stuffing** mechanism is used to transfer enough rising or falling edges in main signal. When last 5 bits (bodes) in the row has the same level, artificially extra bode with opposite value is added what generates artificial signal edge helping receiver to synchronise its clock to transmitter clock. This operation is called bit-stuffing.

Maximum baudrate defined by CAN standard is 1 Mbit/s but typically in automotive 500 kbit/s speed is used.

### 2.1.5. Frames arbitration

Another aspect related to the safeness is significance of the transmitting data. Some of frames may contain critical safety data (e.g. information about crash incident from sensors to airbags system), so these should have been sent before others.

In CAN Bus can be found mechanism prioritizing frames. Frames with lower ID has higher priority than this with higher ID and are sent in prior to the second ones. It is related to the fact, that CAN has open drain/collector output, and zero (0) is dominant state [7].

### 2.1.6. State of the node

CAN Bus node (ie. described here Data Logger) may be in 3 states:

– Error Active - normal operation

– Error Passive - error counter (errors detected by this node) raises above 127. Node in this state is not allowed to send Error Frame.
– Bus Off - Node is not allowed to transmit anything on the bus.

Node error counter is increased by 8 for sending error frame and is decreased by 1 for correctly detecting error on the bus or by receiving correct frame [8].

**Changes of node state are logged** by described system.

## 2.2. Global navigation satellite system (GNSS)

Satellite positioning/navigation systems are used for estimating coordinates of the user in each place on the globe. 4 separate positioning systems exists worldwide:

– GPS (Global Positioning System) - operating since 1978, available since 1994. Developed and owned by by United States government
– GLONASS (**GLO**bal **NA**vigation **S**atellite **S**ystem) - Russian developed navigation system
– Galileo - European Union and European Space Agency alternative to GPS
– BeiDou - Chinese satellite navigation system

The receiver used in the Data Logger being described here is configured to use GPS and GLONASS systems.

### 2.2.1. General system description

GNSS systems consists of three segments: satellites placed in the outer space, control segment placed in known position on the globe and on-the-ground receivers being hold by end users.

### 2.2.2. Space segment

Space segment consist of at least 24 operational satellites in GPS system [9]. The same is for GLONASS and Galileo. Satellites are moving around the globe on one of 6 orbits. The user with probability 99,9% will have in line-of-sight minimum 5 satellites [10].

Satellites broadcasts radio signals which includes: location, status and precise time. Time is being measured by on-board atomic clock. Signals transmitted by satellites travel to the devices being in the user segment with speed of light [10].

### 2.2.3. User segment

This segment consists of every device which contains GPS receiver. This part of the system is passive - user only receive signal from satellite. Does not transmit any signal other way.

Each of satellites sends message which contains primarily precise time (when the message was sent) and precise position of the satellite. Moreover contains information about the rest of the system.

When the receiver receive messages from minimum 4 satellites, using received positions of satellites and distance to the satellites (deducted by the time difference and time correction - based on the message received from fourth satellite) the receiver may calculate accurate position on the globe.

### 2.2.4. Control (on-the-ground) segment

Each of the GPS satellites is being tracked by control stations, monitor antennas and monitor stations - signals are being received from the satellites and compared to actual position with the on-the ground station/antenna. Difference is sent to Master Control Station which communicates with each satellite and makes corrections to the time, position and taken orbit based on calculated differences.

### 2.2.5. NMEA protocol

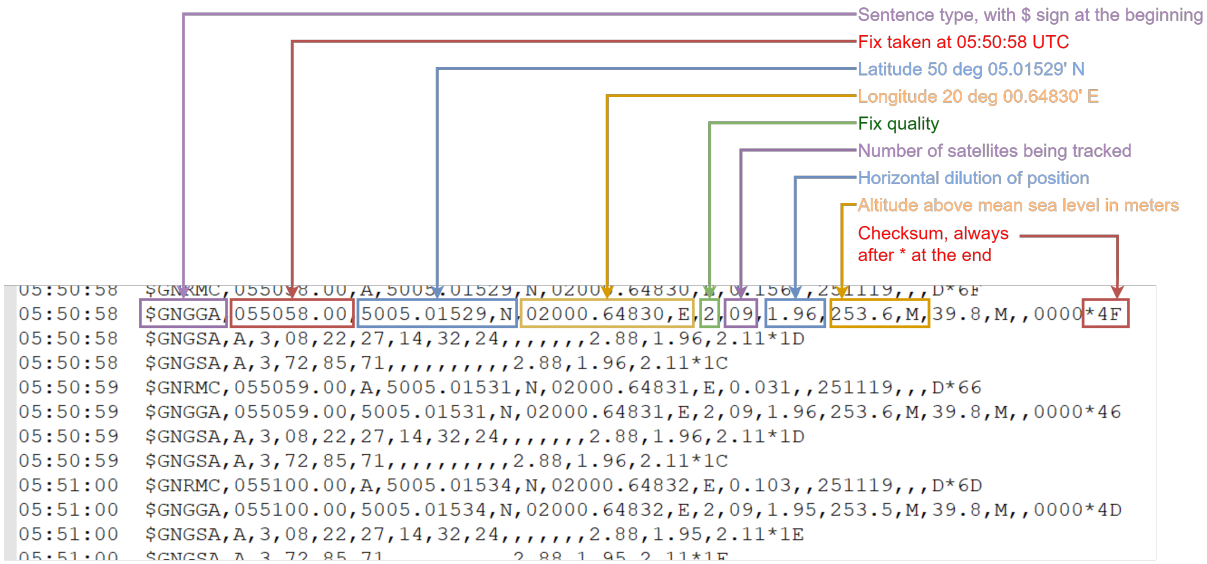



**Figure 2.2.** GNGGA NMEA sentences example explanation

NMEA (National Marine Electronics Association) protocol is used to transmit positioning data **from GNSS receiver** chip **to the MCU** using universal asynchronous receiver-transmitter (UART) protocol on the data link layer. Messages itself are sent via text/string sentences. Format of this messages is defined by NMEA standard [11]. This protocol was defined primarily for communication between navigation and marine electronics equipment.

Data sent by NMEA sentences includes complete position, velocity and time computed by GNSS receiver by itself. Each sentence is independent to another, begins with '$' sign and ends with end-of-line sign. Data parts are separated by commas in a strictly defined way. What type of data is included is defined by each of NMEA sentences. In this project sentences below are used:

– **$GNGGA** - containing among others: time, latitude, longitude, number of satellites being tracked, dilution of precision, altitude

– **$GNGSA** - containing among others: type of fix (2D or 3D), satellites used for fix, and dilution of precision in both directions
– **$GNRMC** - containing among others: time, latitude, longitude, speed over the ground, date

## 2.3. On-board communication protocols

To communicate with peripheral chips not being embedded into MCU, extra communication protocols are necessary. Those are used to communicate with GNSS receiver and microSD memory card.

### 2.3.1. UART

Universal asynchronous receiver-transmitter (UART) is asynchronous serial communication protocol. It is used for point-to-point communication. In the project described here it is used for communication with GNSS receiver.

Two lines are used, where TX pin of one device is connected to RX pin of the second device and TX of the second one is connected to RX of the first one. Two (high and low) voltage levels are used, where for used STM32 microcontroller voltage below $0.3V_{DD}$ is treated as low state, and above $0.7V_{DD}$ is high state [12]. Single frame consists of start bit, data bits of configurable length (7, 8 or 9 bits), optional parity bit and 1 or 2 stop bits. Buadrate is also configurable by the system designer.

In this project 8-bit data length, no parity and 1 stop bit is used. Baudrate is set to 115000 bits per second. This value is fast enough to handle 10Hz GNSS data receiving, what is highest frequency of positioning data used in this system.

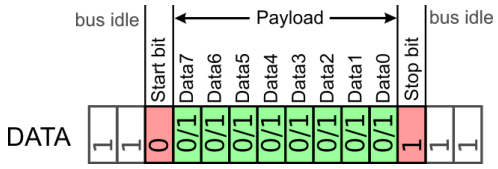



**Figure 2.3.** UART frame format used in this project

### 2.3.2. SD-Card interface

Communication with SD/microSD card is based on dedicated nine-pin interface. Two modes: **SPI** and **SD Bus mode** are defined to communicate with SD cards [13]. Moreover, SD Bus mode defines two variants: with **single data line** (DAT0) and **four data lines** (DAT0-DAT3). The second variant has 4 times higher transfer rate - even up to 100 Mb/s, according to protocol definition [13].Maximum speed is different for save and read operations and depends on the used card model. In the project described in this thesis SD-Card interface with single data line variant is used.

Motivation for using SD-Card interface instead of SPI in described here Data Logger:

1. Native support for **FatFs library** [14] used with STM32 libraries - code modules prepared by the configuration tool (STM32CubeMX) provided by microcontroller manufacturer
2. Possible very simple expanding of layout between single data line and four data line if speed would have found out too slow. Only minor changes in the code are necessary. No such possibility for SPI mode.

Card necessary connections are:

– Supply voltage (3.3V)
– Ground
– Clock
– Data Line
– Command/response

This protocol gives direct access to the memory blocks on the card. For files definition in the memory any file system is necessary. FAT32 filesystem have been chosen for this project and is described in the following section.

## 2.4. File Allocation Table (FAT) file system

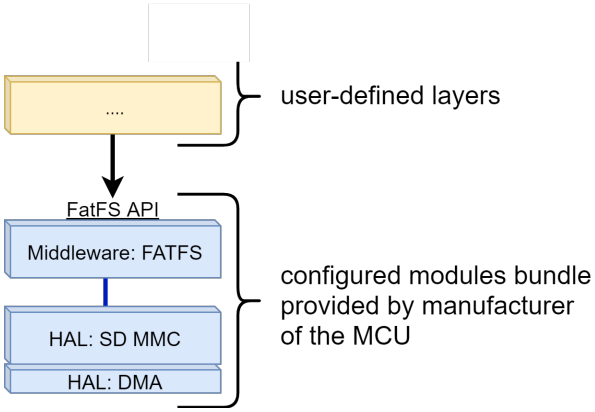



**Figure 2.4.** FatFs as middleware layer

File system defines how data is organised into files on given storage medium (microSD memory card in this case) and how to retrieve files from storage medium. File system is independent of used platform or story media [14]. File system is middleware layer between application (where access to files and directories may be used by filesystem API) and hardware drivers (where raw memory management is necessary). If file system did not exists it would be impossible to manage hierarchy, names and actually know where each of the files is placed in the memory [15].

One of filesystems widely used in embedded solutions is **File Allocation Table (FAT)**. It is ideal for removable Flash media such microSD cards [16]. Thanks to fact that **FatFs** [14] open-source library have been written. This library is independent of used platform. Low-level memory drivers should be provided by the user. In project described in this theses the manufacturer of used MCU provided this

library with pre-configured and implemented low-level SD-Card interface drivers. Configuration of both is possible from hardware configuration tool - *STM32 CubeMX*.

FAT system contains information about partitions on given storage media, starting and ending sectors numbers of each partition, number of bytes per sector, number of sectors per cluster, total count of sectors on the partition, number of root directory entries, etc.

# 3. Hardware

For designing schematics and PCB layout **Altium Designer 19** software was used. Components libraries and device sheets used in the project was prepared by the author of this thesis or members of AGH Racing Team. PCB Layout was prepared entirely by the author of this thesis.

## 3.1. Technical requirements

Requirements for Data Logger device are:

- Standalone embedded device
- Logging time - minimum 30 minutes from start to end of one file
- Keeping track of actual date and time after switching off the power.
- Support for one *CAN* bus connection
- Logging data on removable microSD card
- Electrical requirements:
  - Supply voltage from 5.0V to 18V (automotive environment)
  - 4-pin connector for power supply and *CAN Bus* connection (pins: 12V, GND, CANH, CANL)
  - Device protection for power supply over-voltage or opposite polarisation
  - *CAN Bus* and power supply ESD protection
- Physical and PCB requirements:
  - Minimal PCB size with width equal to 26mm (must fit to the standard team case design)
  - 2-layers PCB, minimum 8 mils tracks width, possibility of manual assembly
  - Case allowing the device to be mounted in the AGH Racing car

## 3.2. Used components

Choosing microcontroller and other project components is included in scope of work. Let's present modules used in the project.

### 3.2.1. Microcontroller

Choosing proper MCU was crucial decision for whole project. *AGH Racing Team* decided to use two microcontroller models mainly to minimise necessary amount of spare parts for all project developed in the team.

First model was low-cost and simple MCU from STM32 F0 family with core operating at up to 48MHz. Second one (used in this project) has high core clock frequency (max 216 MHz) and higher amount of available peripherals.

Reason for choosing STM32 series was experience with this MCU family gained in previous seasons and available code from last-year projects. Moreover HAL library [17] provided by the *ST Microelectonics* company was big advantage to other manufacturers, where low level libraries was necessary to be implemented by the programmer himself.

Necessary embedded peripherals available directly in MCU was:

- CAN controller
- UART or SPI controller (for communication with GNSS receiver)
- real-time clock (RTC) controller with battery backup
- SD-Bus controller with DMA support
- ARM 32-bit core.
- real-time architecture
- available evaluation board with chosen MCU
- RAM size letting to buffer reasonable amount of CAN frames and GNSS NMEA sentences

All above requirements and requirements from other projects in AGH Racing using the same MCU let to be decided to use **STM32F767ZIT6** microcontroller. This device meets all above requirements.

Chosen MCU has ARM 32-bit Cortex-M7 CPU with clock up to 216MHz and 512 KBytes of SRAM and 2MBytes of Flash. Contains four UART, three CAN and 2 SDMMC interfaces. NUCLEO-F746ZG is ready-to-go evaluation board with embedded SWO programmer and all peripherals available on external pins.

MCU is being programmed with St-Link programmer using Serial Wire Debug (SWD) protocol.

### 3.2.2. GNSS Receiver

GNSS receiver is one of keystones of whole system. In previous revision of this system SIM28 GPS receiver was used. Main problem with this chip was very poor support and documentation what led to really long development time of drivers. Availability of described chips on the market was also limited, so author decided to change used GNSS receiver. Moreover, previous chip supported only GPS system, any of GLONASS, BeiDou or Galileo was not supported by SIM28 chip.

After market research author found out that very good support is for *U-Blox* devices. *UBlox NEO-M8N* has support for three localisation systems and has the same dimensions and pinout to *SIM28*. Above arguments convinced author to use this chip in the project.

*UBlox NEO-M8* is versatile GNSS module. It has ability to concurrent reception of up to 3 systems from list of: GPS, Galileo, GLONASS or BeiDou. The device analyses and merges results from those navigation systems and provides data to the MCU using one of protocols: UART, USB, SPI or I2C. UART connection is used in the application described here. 115.2 Kb/s baudrate is used. Supply voltage is equivalent to used for MCU (3.3V) and external passive and active antenna support (for higher accuracy) is available.

### 3.2.3. Power supply

Data logger is meant to be used in *AGH Racing* cars environment, what implies 12V nominal power supply. It causes necessity to use dedicated power-supply electronics circuit.

#### 3.2.3.1. Main power supply

In the *AGH Racing* cars light-weight **3S and 4S LiPo batteries** are used. That implies that maximum voltage in the system may be even $4.2V \cdot 4 = 16.8V$ [18], so absolute maximum value of power supply voltage, including safety factor, may not be lover than 18V. On LiPo batteries voltage may not be lower than 2.7V [18] on each cell, so for the system with **LiPo 3S** batteries mounted, power supply voltage may not be lower than $2.7 \cdot 3 = 8.1V$. Expecting voltage drops and safety factor in the system, Data Logger power supply minimum voltage should not be higher than **5.0V**.

STM32 MCU, UBlox GNSS module, microSD card and other peripherals have 3,3V nominal power supply voltage [12] [13] [19]. Not having any high-power peripherals but having limitations for used space for PCB, low drop-out voltage regulator (LDO) found out to be satisfying solution.

Author of this project decided to use **NCP1117DT33G** LDO, which input voltage may vary from 4.5 to 20V, what meets the requirements. Maximum current for this LDO is 1.0A, but maximum temperature is limited to 150°C [20]. Let's consider max temperature which may be reached by the device.

During experiments maximum current consumption did not exceed **111mA** for whole voltage range. Let's compute maximum calculate power dissipation:

$$P_{DMAX} = (U_{in} - U_{out}) \cdot I = (18 - 3.3)[V] \cdot 0,110[A] = 1.617[W]$$

Junction-to-Ambient thermal resistance is equal **67°C/W** for this LDO [20]. Let's compute if the LDO would not overheat for worst-case scenario. Barcelona is city with highest temperatures where Formula Student competition is taking place. Let's assume 38°C ambient temperature [21]:

$$T_J = (\theta_{JA} \cdot P_D + T_{air} = 67\frac{^\circ C}{W} \cdot 1.617 + 38^\circ C = 146,4^\circ C$$

This implies that calculated value is lower than maximum temperature possible for used LDO.

#### 3.2.3.2. Backup battery power supply

For *MCU RTC module* and *GNSS chip* extra LR54 backup battery is used. Thanks to that calendar and time of MCU is not reset after power off.

Backup battery is also connected to GNSS module what improves chip performance making possible to run hot start or warm start speeding up fixing position. If no backup battery is provided module always has to be started in cold startup mode.

## 3.3. Protection

Protection circuits in consumer electronics is used to avoid serious damage after any sudden mishappening and unwitting wrong usage of the device electrical interface.

## 3.4. Over-voltage and reverse polarity protection

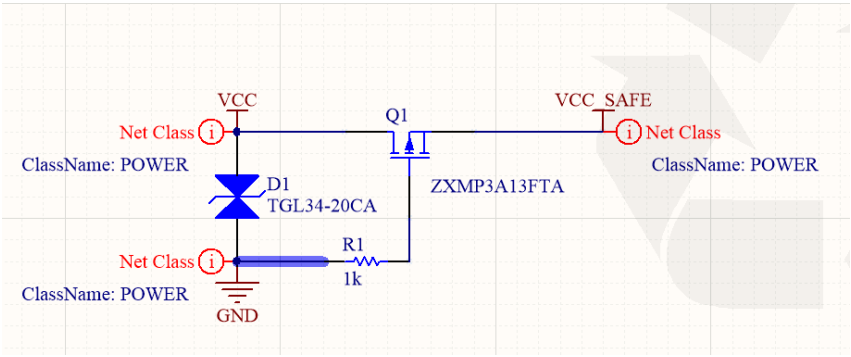



**Figure 3.1.** Over-voltage and reverse polarity power supply protection - implementation schematic

The device is protected by using transil TGL34-20CA which shortcuts VCC with GND pins if the difference of potentials on those pins is higher than 20V [22]. After that happen over-current protection in the car should shut-off the device.

Moreover P-MOFSET transistor is used for protecting the device against reverse polarity. If the GND voltage potential will be higher than VCC, transistor Q1 will not conduct and the rest of the system will be safe.

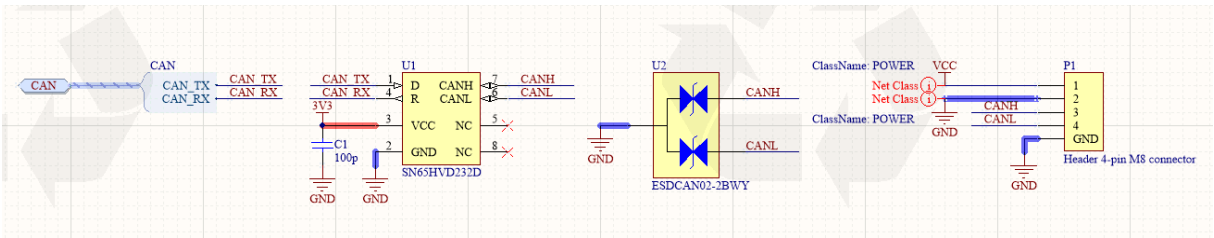



**Figure 3.2.** CAN Bus transceiver and CAN ESD protection

## 3.5. ESD CAN bus protection

For protecting CAN transceiver from ESD discharge dedicated chip was used. It is dual-line transil which can discharge up to 30kV pulse with power up to 250W.

## 3.6. Main schematics

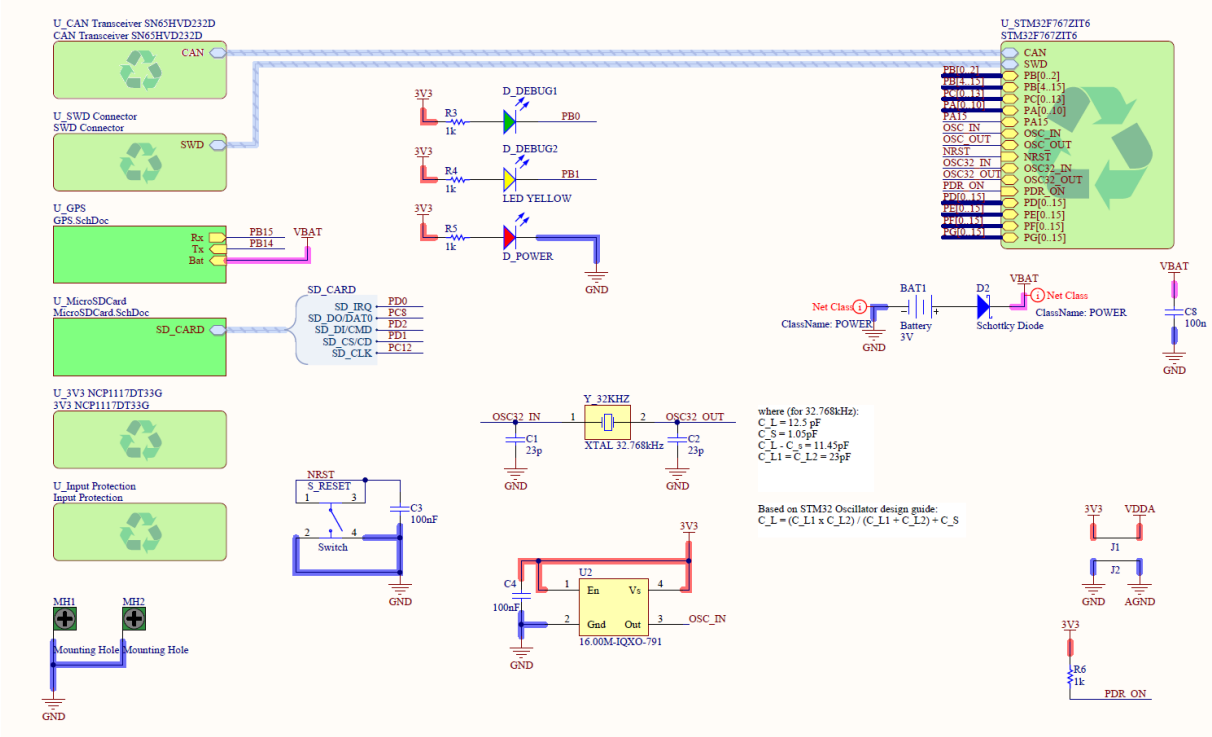



**Figure 3.3.** Main schematic of Data Logger PCB board

Whole Altium project was divided to pre-prepared *device sheets* - standard schematic sheets to be re-used in many projects, ie. MCU with decoupling capacitors, power supply with protection or standard CAN transceiver. Non-standard parts was prepared by the author of this thesis. Those was GNSS connections, miscellaneous LEDs, chips, extra SMA (GNSS) connector, microSD card slot and main sheet connecting all the schematic together. This one is showed in the figure 3.3:

## 3.7. PCB Layout

Layout of the PCB is prepared to be manufactured in **8 mils** technology, on **1.0mm thick**, **2-layer** board with **35um copper** thickness. During layout design it was necessary to meet requirement of **26mm maximum PCB width**. This requirement is dictated by necessity of fitting the PCB to the standard case design of *AGH Racing team*. Quite large MCU case (24x24mm LQFP176) also required efficient paths routing using all available space on the PCB. Manual soldering process forced to use passive elements

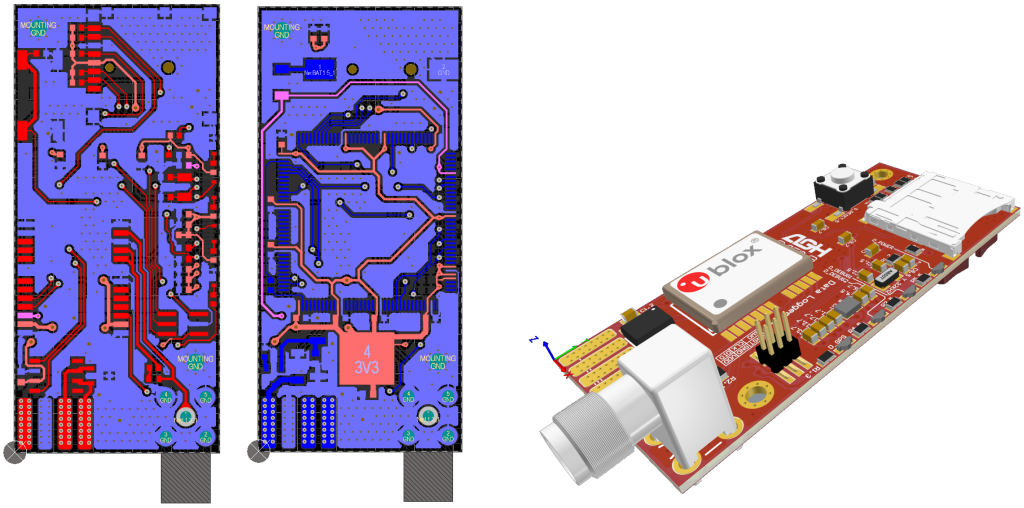



**Figure 3.4.** PCB layout - top(left), bottom(midle) and 3D(right) view

not smaller than **0603 case** and chips with **pin raster** not smaller than **0.5mm** admitted as the smallest possible to solder by manual soldering.

## 3.8. Case design and manufacturing

The device is meant to be used in automotive environment what implies hard operating conditions. That implies the need for a housing. Because low volume of devices is going to be assembled **3D print technology** for case manufacturing seems to be natural choice. Case was designed by the author of the thesis in **SolidWorks CAD** application based on *AGH Racing Team* standard case template. STEP file exported from Altium Designed software helped to prepare case fitting properly the PCB. The case was printed with usage of Fused Deposition Modeling 3D printer. It is designed to be simply mountable on the frame pipe of the *AGH Racing* car frame.

## 3.9. Manufacturing process

Whole PCB was soldered manually by the author. Soldering was performed with solder paste and Hot-Air station for all components except GNSS receiver which was soldered manually with soldering iron.

After soldering visual inspection with microscope was performed for checking of shortcuts, "cold solder" joint or any other soldering defects.

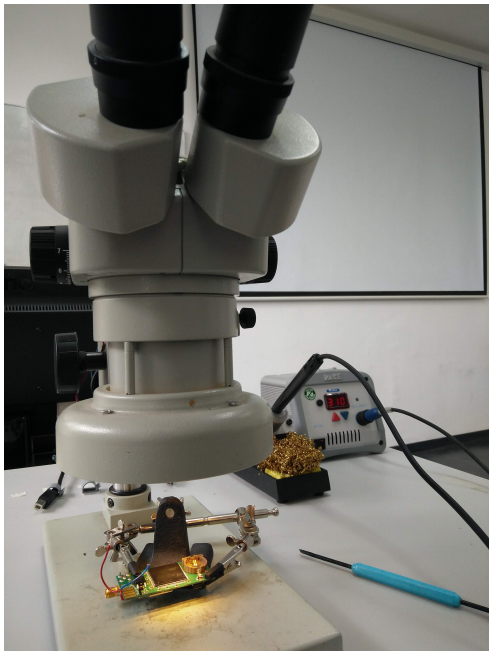


**Figure 3.5.** Visual inspection of soldered PCB board

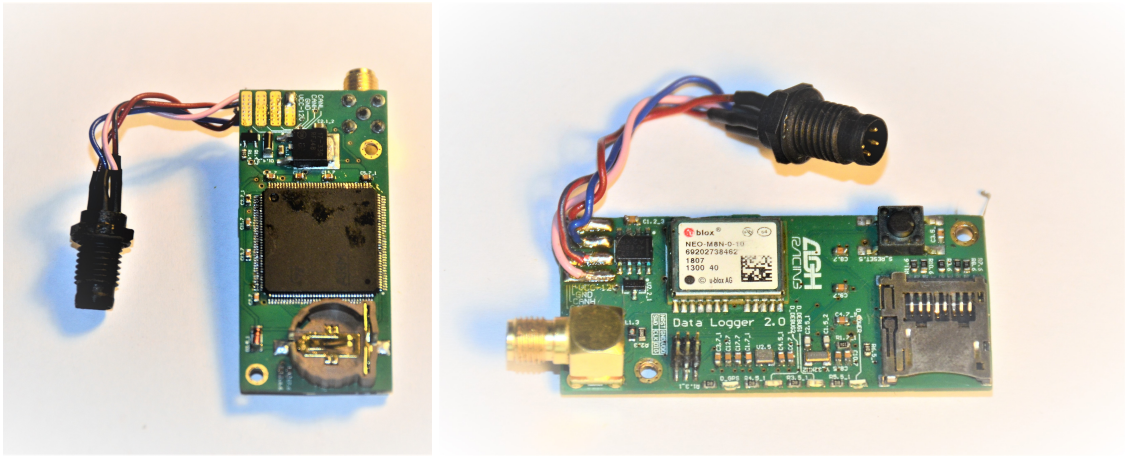



**Figure 3.6.** Data Logger PCB board - front and bottom view without case

# 4. Configuration and log files structure

Taking into consideration requirements of the system and mark-up of ASCII representation (size of text file may be much higher than for binary type) of all typical files formats like XML, JSON or INI, author of this thesis decided to invent custom binary data types: *.aghconf* for configuration of Data Logger and *.aghlog* for data log.

Let's consider 16-bit unsigned integer which range is from $0$ to $2^{16} = 65\ 536$. For 1-digit values (from 0 to 9) two bytes are necessary to write it in ASCII form (one byte for writing digit using ASCII coding and one byte for any delimiter sign). It is the same number of bytes like for binary representation. But for higher numbers things are getting worse because for 5-digit value (above $65\ 000$) 6 bytes of data are necessary what is 3 times more than in binary form.

Moreover *Vector Informatik* company (respected producer of professional simulation environments for automotive buses) recommends using binary (*.BLF*) file format instead of ASCII (*.ASC*) format for high data rates [23]. Unfortunately *.BLF* is closed format so using it in this project was not possible.

During 30 minutes test described in chapter 7, *.ASC* file had over 14 times higher size than *.BLF* reaching 387MB(*.ASC*) instead of 26MB(*.BLF*).

All parameters values in *.aghconf* and *.aghlog* file are stored using as small as possible data type, which fits all possible values of parameters from one of types listed below:

- **uint8_t** - 1-byte unsigned value. Used for parameters which value is known to be non-negative and and not greater than $2^8$.
- **uint16_t** - 2-byte unsigned value. Used for parameters which value is known to be non-negative and and not greater than $2^{16}$.
- **uint32_t** - 4-byte unsigned value. Used for parameters which value is known to be non-negative and and not greater than $2^{32}$.
- **int32_t** - 4-byte signed value. Used for parameters which value is known to be not less than $(-1) \cdot 2^{31} - 1$ and not greater than $2^{31}$. Two's complement representation is used for negative numbers.
- **char** - 1-byte value interpreted as ASCII char

## 4.1. Configuration file

This file contains all necessary parameters for configuring all modules in Data Logger device. It includes drivers parameters but also frames which should be logged.

### 4.1.1. Hardware configuration

First field in configuration file is its (configuration file) **version** and **subversion**. Thanks to that MCU and PC Application can recognise if the definition of file format is not out-dated.

Second field is string (chars array) with prefix for **log file name**. It may determine name of the car, test name or whatever user finds useful.

Next very important parameter available to be configured is **CAN Bus baudrate**. Few standard values are available to be set: 50kb/s, 125kb/s, 250kb/s, 500kb/s and 1Mb/s.

Next important parameter is **frequency of GNSS data** - how often GNSS receiver should send via *UART bus* NMEA sentences to the MCU. That implies how often localisation data will be saved in the log file. One of listed options is available: GPS OFF, 0.5Hz, 1Hz, 2Hz, 5Hz, 10Hz.

Next thing is **frame id** which is used **for setting-up RTC time**. Using pre-defined format is possible to configure actual RTC date and time. It may be used to synchronise clocks between many devices on the bus.

Second option for synchronising date and time is **GNSS/GPS synchronisation. Flag** in configuration file determines if time received from GPS satellites should be only save to the log file, or RTC of MCU should also be synchronised. If set to true **time zone** may be also configured.

Next is the definition of frames willing to be logged.

### 4.1.2. Frames including signals conception

As long as frame definition is identical with *CAN Bus frame* definition, signal concept is not a part of that.

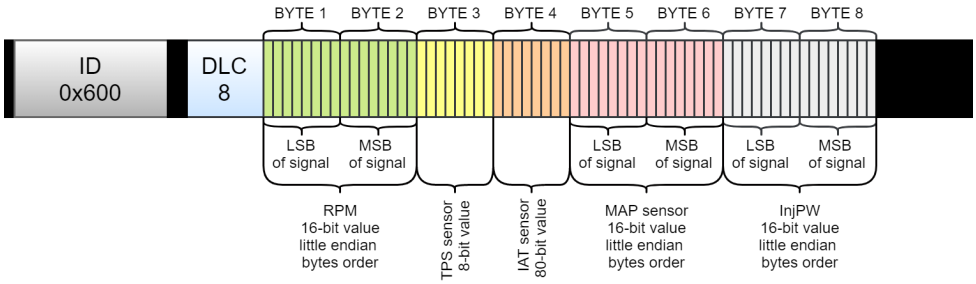



**Figure 4.1.** Graphic representation of frame with indicated signals definitions. Example of one of frames used in *AGH Racing* combustion car.

Signal is an consistent subset of $n$ frame payload bits, which may be interpreted as $n$-bit integer value. On those bits may be transferred value of any parameter of the engine or value of any sensor in the

car. One signal is associated with only one frame, but one frame may contain many signals. Separate bits subsets may be interpreted differently. Example of frame with defined signals is presented in graphical form on figure 4.1.

### 4.1.3. CAN Frames definition

Let's remind that ID of the frame determines content of the frame, not the sender node itself. Thanks to that fact the user can define which frames should be logged and which not. Thanks to that log file has smaller size and is easier to analyse. Each frame being logged is determined by its **ID** and contains properties as **expected DLC** and **user-defined name**.

After definition of those parameters, **number of associated signals** with this frame is defined. Then definitions of those signals are placed in the form described below.

### 4.1.4. Signals definition

Signal position is determined by **start bit** in frame payload and **length in bits**. Let's notice that $([\textbf{start bit}] + [\textbf{length in bits}])$ must not be greater than $[\textbf{expected DLC of frame}] \cdot 8$ in which the signal is defined.

Because range of integer type may not fit range and accuracy of physical value, conversion formula is used for interpreting data. Let's call integer value of signal bits as they are sent on the bus as **raw value** and **physical value** define as converted **raw value** of signal to range being interpreted as actual physical value using conversion formula as follows:

$$[\textbf{physical value}] = \frac{[\textbf{raw value}] \cdot [\textbf{multiplier}]}{[\textbf{divider}]} + [\textbf{offset}]$$

All of **raw value**, **multiplier**, **divider** and **offset** are integer values, but **physical value** is decimal value which is physical representation of data being sent on bits defined for given signal.

Two more parameters must be defined to correctly interpret given value. First one is **byte order**. While bit order is defined by the *CAN Bus* standard, until byte order may be defined by the user. So if the signal contains more than 8 bits it is necessary to define if *little endian* or *big endian* notation is used.

Second thing is the information if the **raw value** should be interpreted as **unsigned value** or **signed value** (with Two's complement representation)

So in described here implementation signal definition consists of:

- **signal id** (with frame id forms a unique key for further identifying)
- **start bit index**
- **length of signal** in bits number
- **value type** - bit-fields consisting of: little(0)/big(1) endian flag, unsigned(0)/signed(1) value
- **multiplier** to conversion formula
- **divider** to conversion formula
- **offset** to conversion formula

- **channel name** string
- **channel unit** string
- **user comment** string

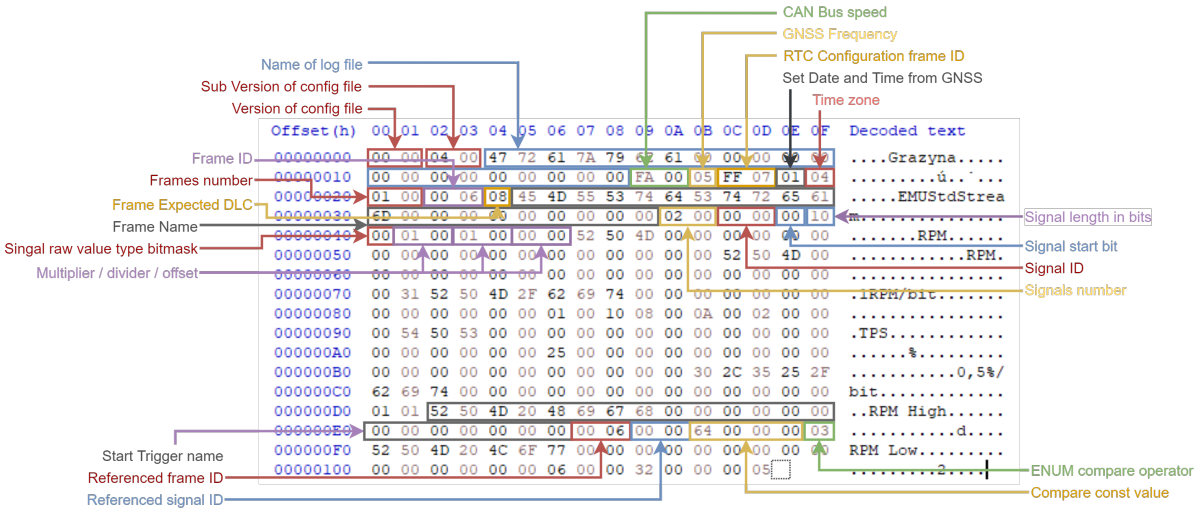



**Figure 4.2.** Simple *.aghconf* file in binary form with some values configuration values indicated

### 4.1.5. Start/stop log trigger definitions

One of system requirement is possibility to *start* and *stop* of log as reaction to values of signals within received frames. That is the reason of defining start and stop triggers. Trigger definition consists of:

- **trigger name** - for the user usage
- **frame id** of the frame to which the trigger applies (or signal within this frame)
- **signal id** within frame above, if compare operator requires signal value to compare
- **compare operator** to compare selected signal value (or any frame property itself) with constant value. This property has enumumeration type with value equal to one of following:
  - *equal*, *not equal*, *grater*, *greater or equal*, *less*, *less or equal* - compare operators comparing denoted signal value with constant compare value
  - *bitwise and*, *bitwise or*, *bitwise xor* operators performing logical operation of *raw* denoted signal *value* with constant compare value
  - *frame occurred* - condition is met when frame with given id occurs
  - *frame timeout* - condition is met when frame with given id does not occur on the bus with given constant timeout time
- **constant integer value** to be **compared** with denoted signal values

## 4.2. Log file

Log file contains records which corresponds to events with received data. Three type of events may be logged: received *CAN Bus frame*, *GNSS data* and *CAN Bus errors* events. Each of this events is bearing *timestamp in milliseconds* timed from start of the log.

### 4.2.1. CAN Frame event

First type of record may be *CAN Bus frame*. Event (after **timestamp**) contains information about **frame ID**, frame **DLC** and bytes with **payload**.

### 4.2.2. GNSS data event

GNSS data event (after timestamp) contain word with special **GNSS data ID** (0x800 value - 12 bits must be used to save this number, so CAN frame ID is not possible) and then following fields:

- 2-byte **yaer**
- 1-byte **month**
- 1-byte **day of month**
- 1-byte **hour**
- 1-byte **minute**
- 1-byte **second**
- 4-byte fixed-point representation of **longitude**
- 4-byte fixed-point representation of **latitude**
- 1-byte **number of satellites**
- 4-byte fixed-point representation of **altitude**
- 4-byte fixed-point representation of **speed over ground**
- 4-byte fixed-point representation of **track angle**
- 1-byte ENUM for **fix type**
- 4-byte fixed-point representation of **horizontal dilution of precision**
- 4-byte fixed-point representation of **vertical dilution of precision**

Fixed-point data type is explained in chapter 5.9.4.

### 4.2.3. CAN Error event

CAN Error event (after **timestamp**) contains word with special **CAN Error data ID** (0x801 value) and **type of the event** coded on 1-byte integer. That may be on of the following:

- *Protocol Error* - general protocol error
- *Passive mode* - when CAN controller goes into *Error Passive mode* described in chapter 2.1.
- *Bus Off mode* - when CAN controller goes into *Bus Off mode* described in chapter 2.1
- *Bit Stuffing Error* - when bit stuffing rule is not met for one of frames on the bus

- *Form Error* - when one of bits of frame with fixed value is incorrect
- *ACK Error* - when ACK bit is not set
- *Bit Recessive Error* - when recessive bit was sent to the bus and is read-back as dominant
- *Bit Dominant Error* - when dominant bit was sent to the bus and is read-back as recessive
- *CRC Error* - when CRC value is incorrect for any received frame
- *Transceiver Error* - when any transceiver error occurs
- *Other Error* - when any other type of error occurs

# 5. Firmware

Developing good quality firmware was main goal of author of this thesis. Components of **layered architecture** was thought to be used by other members of *AGH Racing Team* in other projects. Because STM32 MCUs architecture is mainly used in the team, mostly low level drivers should be easy to adapt to be used with other projects.

Whole firmware project is implemented in **C language**, in C11 standard, compiled with **arm-gcc** complier. Project was developed in **Atollic TrueSTUDIO for STM32** integrated development environment (IDE).

## 5.1. Technical requirements

- Start-up time less than 2 seconds
- Logging time range greater than 30 minutes for one file
- Real-time system - timestamp of received data frame with accuracy no less then 1ms
- Keeping track of actual time and date after switching off the power.
- Synchronising RTC (Real-time clock) of the MCU (Micro-controller unit) with GNSS/GPS time
- Possibility to start and stop logging as reaction for change of frame payload, occurrence or timeout of particular frame.
- Removable microSD card - file system format readable by standard PC-computer
- CAN Bus:
  - Logging *CAN 2.0A* and *CAN 2.0B* frames functionality with 11-bit identifier. (29-bit identifiers not supported.)
  - possible baud-rate set-ups: 50Kb/s, 125 Kb/s, 250 Kb/s, 500 Kb/s, 1 Mb/s
  - fully functional (logging all frames) with *CAN Bus* working with maximum baudrate 1Mb/s and over 90% bus load
  - correct module behavior according to *CAN Bus* defined **Error Active/Error Passive** modes.
  - Logging only frames defined in the configuration file. Other frames should not be logged.
  - Supporting start/stop triggers definitions according to definition in .aghconf file chapter 4.1
- Data storage:
  - Logging data on removable **microSD** card
  - PC-computer readable file format system on microSD card (like **FAT32**)

  - Main record speed allowing to log CAN Bus and GNSS data in max baudrate/frequency of both
- Data Logger configuration in *.aghconf* format:
  - Interpreter of configuration file described in chapter 4.1.
- *.aghlog* file format for logged data:
  - logged data should be saved in format described in chapter 4.2
- GNSS:
  - Logging *GNSS* data functionality with frequency no less than 10Hz

## 5.2. Estimation of bytes number to save per second

Before preparing code and choosing proper storage mass it was necessary to estimate maximum bytes amount to be logged per second. Values will be calculated for two sources of data: CAN Bus and GNSS and then added together. Assumptions are using .aghlog file format in version 0.4.

### 5.2.1. CAN Bus estimation

Maximum speed (or baud-rate) of CAN Bus is $1$ Mbit/s. Because, as mentioned, data field has $0$ to $64$ bits, while complete frame has $47$ to $111$ bits (maximum when $64$ payload bits).

Let's calculate number of complete frames per second for each of DLC from 0 to 8 for 100% bus load. DLC may be in range from 0 to 8. Consider only frames with $0$ bits stuffed (because those does not carry any information and extends frame transmission time). To calculate number of frames per second for each DLC it will be used formula below:

$$\frac{1\ 000\ 000\ \frac{bits}{s}}{\text{bits in frame}} = \left[\frac{\text{frames}}{\text{second}}\right]$$

Each of this frames has *DLC* bytes of payload, 11-bit ID (saved as 2-byte value), DLC itself (saved as 1-byte value) and timestamp (saved as 4-byte value). So bytes number to be save be saved per second will be calculated using formula below:

$$\left[\frac{\text{frames}}{\text{second}}\right] \cdot (\text{DLC} + 2 + 1 + 4)\ \text{bytes} = \left[\frac{\text{bytes}}{\text{second}}\right]$$

For each of possible *DLC* amount of data per second is calculate in Table 5.1.

So actual maximum payload size to be logged per second is $1191.5$ Kb/s $= 1.19$Mb/s.

Converting to bytes per second it gives: $1191.5$ Kb/s$/8 = 145.5$ KB/s.

### 5.2.2. GNSS estimation

Maximum frequency of GNSS data may be set to 10Hz. Each of GNSS record consists of fields defined in chapter 4.2.2. That gives sum of **43 bytes of data** for one GNSS data bundle. Let's compute amount of data to be saved per second:

| DLC | data bits | Frames/second | Bytes/second | Kb/s | Mb/s |
|---|---|---|---|---|---|
| **0** | **47** | **21276.6** | **148936.2** | **1191.5** | **1.2** |
| 1 | 55 | 18181.8 | 145454.5 | 1163.6 | 1.2 |
| 2 | 63 | 15873.0 | 142857.1 | 1142.9 | 1.1 |
| 3 | 71 | 14084.5 | 140845.1 | 1126.8 | 1.1 |
| 4 | 79 | 12658.2 | 139240.5 | 1113.9 | 1.1 |
| 5 | 87 | 11494.3 | 137931.0 | 1103.4 | 1.1 |
| 6 | 95 | 10526.3 | 136842.1 | 1094.7 | 1.1 |
| 7 | 103 | 9708.7 | 135922.3 | 1087.4 | 1.1 |
| 8 | 111 | 9009.0 | 135135.1 | 1081.1 | 1.1 |

**Table 5.1.** Estimation of amount of data to be saved per second for all possible frame DLC values

$$10\frac{\text{frames}}{\text{second}} \cdot 43 \text{ bytes} = 430\frac{\text{bytes}}{\text{second}}.$$

### 5.2.3. CAN Errors

Logged CAN errors may be omitted in this estimation treated as exceptional situation.

### 5.2.4. Overall

Let's add CAN Bus estimation and GNSS estimation:

$$148\ 937\frac{\text{bytes}}{\text{second}} + 430\frac{\text{bytes}}{\text{second}} = 149\ 366\frac{\text{bytes}}{\text{second}}$$

In the **worst case** scenario system will have **149 KB/s of data to save** with 100% CAN Bus load and max frequency of GNSS receiver.

### 5.2.5. Memory card record speed experiments

Experiment of maximum speed of saving data to SD card was performed. Some (*loops count*) chunks of bytes containing (*bytes in data chunk*) bytes of random data was saved to the memory card. Time of this operation was measured and estimation of max record speed to the memory card was calculated. Results are presented in the Table 5.2;

In the Table 5.2 can be observed that for SDHC card it is possible to save all the data from CAN Bus (with 100% bus load) and GNSS (10Hz frequency), because minimum record speed is equal to 164KB/s, what is greater than 149 KB/s of data receiving.

For microSD 2GB card maximum performance of the system may not be reached, because minimum record speed is equal 108.1KB/s what is less than required 149 KB/s.

It is important to take it into account when installing the memory card in the real car.

| loops count | bytes in data chunk | card type | bytes number | time [s] | KB/s | Kb/s |
|---|---|---|---|---|---|---|
| 1000 | 1000 | SDHC 16GB | 1 MB | 5.119 | 195.4 | 1562.8 |
| 100 | 1000 | SDHC 16GB | 100 KB | 0.546 | 183.2 | 1465.2 |
| **1000** | **100** | **SDHC 16GB** | **100 KB** | **0.61** | **163.9** | **1311.5** |
| 1000 | 1000 | microSD 2GB | 1 MB | 8.243 | 121.3 | 970.5 |
| 100 | 1000 | microSD 2GB | 100 KB | 0.812 | 123.2 | 985.2 |
| **1000** | **100** | **microSD 2GB** | **100 KB** | **0.925** | **108.1** | **864.9** |

**Table 5.2.** Maximum speed of saving data to SD card experiment results.

## 5.3. Layered architecture

Whole firmware design was thought to be divided into modules communicating by each other interfaces - public functions defined in header files.

Five layers may be distinguished. All of them are introduced in graphic form on figure 5.1. Let's describe each of them:

### 5.3.1. Hardware Abstraction Layer (HAL)

This layer is provided by the manufacturer of the MCU. It is collection of libraries giving access to the peripherals of microcontroller, but without necessity of writing raw data to processor registers. It is very flexible piece of code what makes it less efficient than writing directly to the memory, but reduces time and effort of development

### 5.3.2. MCU peripheral drivers

This layer is actually abstraction layer between higher-level code implemented by the author and described above HAL. This layer is taking role of *facade design pattern* [24] interfacing HAL interface with quite complicated API and simplifying access methods for the user. Role of this layer is also to aggregate few functionalities and provide simple interface for higher layers.

### 5.3.3. Middleware layer

Code in this layer is used for aggregating some partial information from low-level layers, in example composing whole *NMEA sentences* from single bytes received from GNSS receiver by UART controller. This layer also contains *driver for GNSS receiver*. It implements interpreter of NMEA sentences aggregated by the *UartReceiverStartTerm* module (using start and termination sign). Described here splitting functionalities into separate modules implements *Single Responsibility Principle* from *S.O.L.I.D.* principles [25].

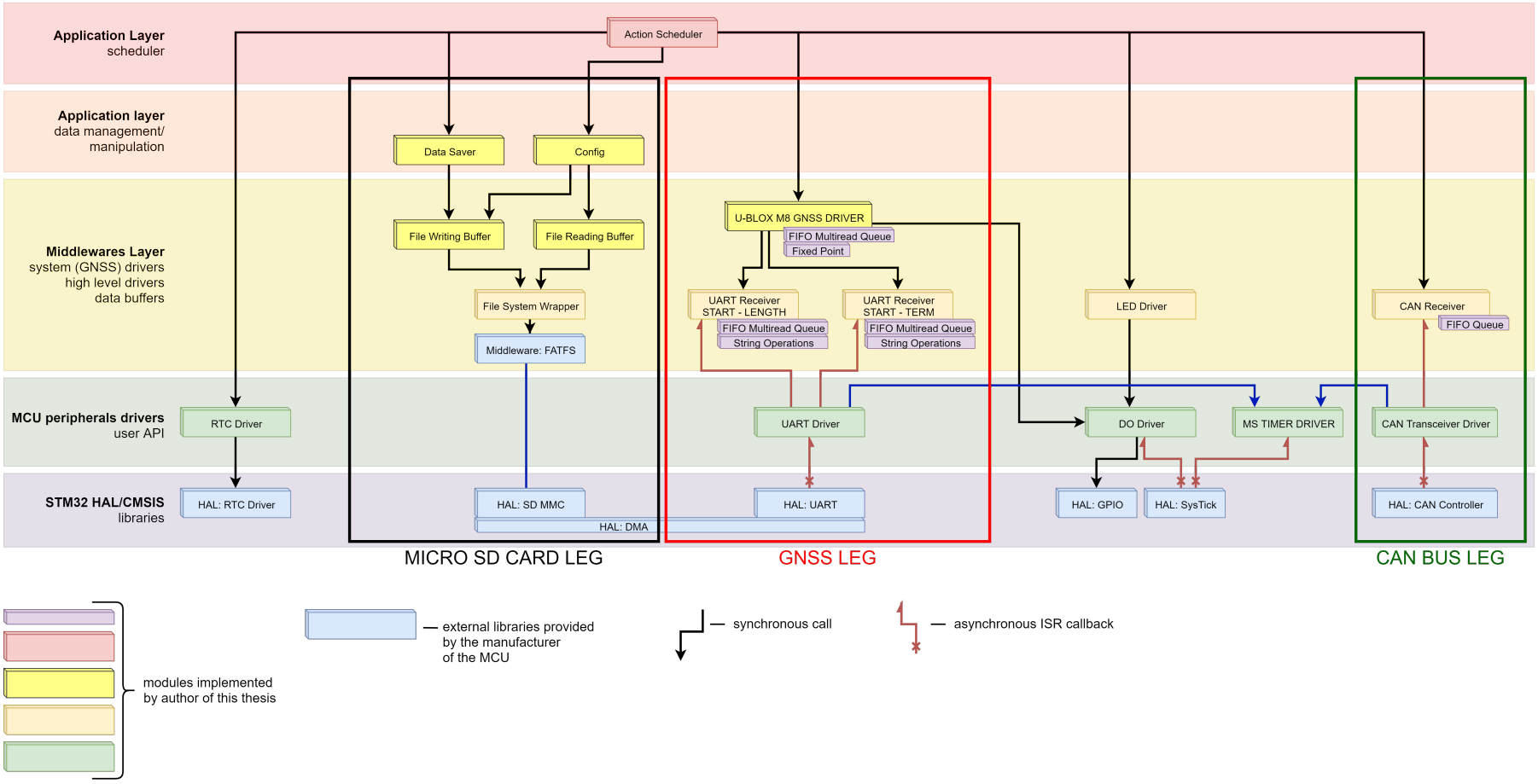



**Figure 5.1.** Functional layers of the firmware.

### 5.3.4. Application layer - data management manipulation

Code of this layer is more closely related to the application. Raw data received or buffered by lower layers are interpreted according to application requirements. First example is reading configuration file. This module processes bytes into *structure* fields according to *.aghconf* file definition. This *structure* is available for other code modules without necessity to read again raw config file. Second example is *Data Saver* module which obtains data from CAN and GNSS drivers, and converts them to *.aghlog* format byte-stream and sends to file system wrapper.

### 5.3.5. Application layer - scheduler

Because author of this thesis decided not to use any operating system, writing simple task scheduler as the top-level layer was necessary. This is never-ending loop running each task one-by-one. Because all resources dependencies are handled by low-level libraries, no synchronisation mechanisms (like mutexes or semaphores) was necessary in top-level layer. The scheduler may be in one of four states handled by state machine.

### 5.3.6. State machine

State machine behavioral design pattern [26] is implemented in the mentioned above Scheduler. States available for this state machine and transition graph is showed on the figure 5.2;

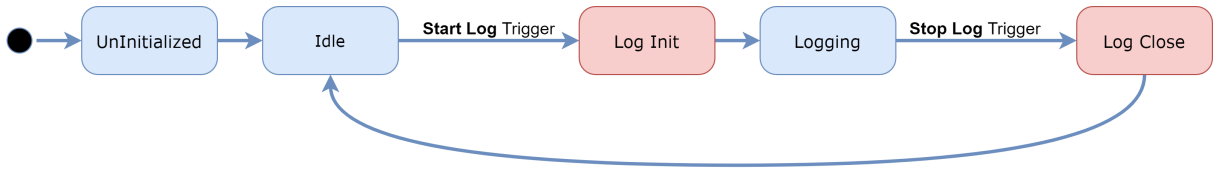



**Figure 5.2.** State machine of tasks scheduler

Figures displayed in red are only transition states for starting (*Log Init*) or finishing (*Log Close*) log. Transition from *UnInitialized* to *Idle* state is possible by running *Init()* function. Going from *Idle* to *Log Init* state and from *Logging* to *Log Close* state is possible as reaction for any *CAN Bus frame* or any value within any frame. It is possible to define five *start log triggers* and five *stop log triggers* which are saved in configuration file.

In example, for AGH Racing combustion car start log trigger occurs when RPM of engine (send in periodic frame from engine ECU) are higher than 100, and *stop log trigger* occurs when RPM is lover than 50. Thanks to that log is being saved only when engine is on.

## 5.4. Event driven architecture

In this system low-level layers reacts which ISR (interrupt service routine) for asynchronous events. This may be: *CAN Bus frame received*, *UART byte received* or *System tick* (set to 1 millisecond) timeout.

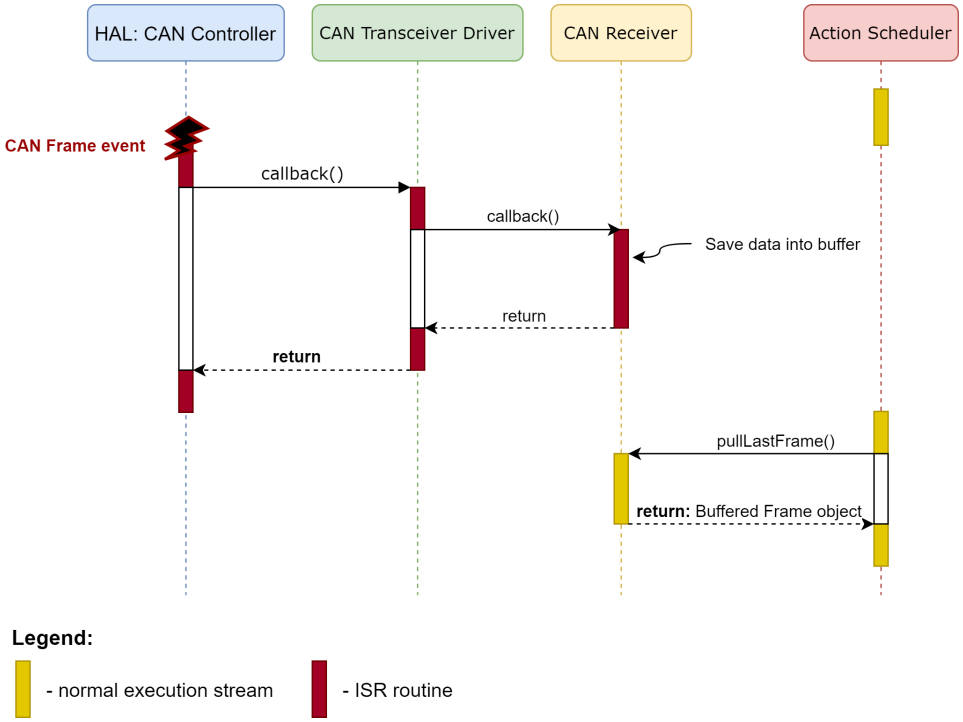



**Figure 5.3.** UML Sequence diagram for *CAN Bus frame* received event

That implies that time-consuming polling method is not necessary for low-level layers and system can react asynchronously (and gain precise timestamp) to the event. Data from ISR (including timestamp) is stored into the buffer, which than is processed in the normal execution stream. To make it possible higher layer modules can register for interrupt callbacks which are handled by modules in *MCU peripheral drivers* layer (figure 5.1). Events sources and propagation paths (with callbacks) are market with red arrows on figure 5.1. Example flow control of received CAN frame is showed on figure 5.3

## 5.5. Direct memory access (DMA)

For communication with microSD card *Direct Memory Access (DMA)* mechanism is used. It is technique where dedicated controller can copy some parts of memory directly into peripheral controller (in this case SD-Card interface controller) performing simple memory copying operations what gives extra time for MCU for more complex operations.

## 5.6. Object-oriented paradigm

It is known that C language is not object-oriented language. It is procedural programming language in its nature. Anyway object-oriented paradigm may be used even in C language [27].

Keeping information in objects being based on classes definitions does not imply necessity of using whole inheritance, delegation, polymorphism, etc. mechanisms. It may be possible to implement those

in C but main goal of author of this thesis was to keep information organised with classes reflecting real-world objects.

Each function is written as method of class. Keyword **this** is replaced by **self** parameter function being pointer to struct of the type being operated on representing object hold in **"this"** pointer in C++. Class constructor is substituted by **Init()** function/method which initialises fields of struct/object and sets state of the object to Initialised/Ready/etc.

```
typedef enum {
    CANReceiver_Status_OK = 0,
    CANReceiver_Status_Empty,
    CANReceiver_Status_NotInitialisedError,
    CANReceiver_Status_NullPointerError,
    CANReceiver_Status_AlreadyRunningError,
    CANReceiver_Status_NotRunningError,
    CANReceiver_Status_InitError,
    CANReceiver_Status_CANTransceiverDriverError,
    CANReceiver_Status_FullFIFOError,
    CANReceiver_Status_MSTimerError,
    CANReceiver_Status_FIFOError,
    CANReceiver_Status_Error
} CANReceiver_Status_TypeDef;

typedef enum {
    CANReceiver_State_NotInitialised = 0,
    CANReceiver_State_Initialised,
    CANReceiver_State_Running
} CANReceiver_State_TypeDef;

typedef struct {
    volatile CANData_TypeDef          aReceiverQueueBuffer [CAN_MSG_QUEUE_SIZE];
    volatile FIFOQueue_TypeDef        framesFIFO;
    volatile CANErrorData_TypeDef     aReceiverCANErrorsQueueBuffer [CAN_ERROR_QUEUE_SIZE];
    volatile FIFOQueue_TypeDef        canErrorsFIFO;
    volatile CANTransceiverDriver_TypeDef*  pCanTransceiverHandler;
    volatile MSTimerDriver_TypeDef* volatile  pMsTimerDriverHandler;
    Config_TypeDef*                   pConfig;
    CANReceiver_State_TypeDef         state;
} CANReceiver_TypeDef;

CANReceiver_Status_TypeDef CANReceiver_init(volatile CANReceiver_TypeDef* pSelf,
        Config_TypeDef* pConfig,
        volatile CANTransceiverDriver_TypeDef* pCanTransceiverHandler,
        volatile MSTimerDriver_TypeDef* pMsTimerDriverHandler);
CANReceiver_Status_TypeDef CANReceiver_start(volatile CANReceiver_TypeDef* pSelf);
CANReceiver_Status_TypeDef CANReceiver_pullLastFrame(volatile CANReceiver_TypeDef* pSelf, CANData_TypeDef* pRetMsg);
CANReceiver_Status_TypeDef CANReceiver_pullLastCANBusError(volatile CANReceiver_TypeDef* pSelf, CANErrorData_TypeDef* pRetErrorData);
CANReceiver_Status_TypeDef CANReceiver_clear(volatile CANReceiver_TypeDef* pSelf);

#endif /* CAN_RECEIVER_DRIVER_H_ */
```

**Figure 5.4.** Header file example presenting object-oriented approach and error handling technique in firmware code

## 5.7. Error handling

C does not provide error handling mechanism like exception handling implemented in C++. Because of that the author of the project decided to use approach where each function returns return defined in Enum type for each module. Error type may be interpreted by comparing with error enum definition. Thanks to that caller of this function is encourage to handle error in contrast to approach used by Linux Operating System, where error value is kept in errno variable [28].

## 5.8. Static memory allocation

Because embedded system has limited amount of memory, memory management is a matter that can be one of main factors for project stability and reliability. Because of that dynamic memory allocation is not recommended [29] or even sometimes forbidden for embedded projects.

Dynamic allocation is non deterministic what in real-time system is not acceptable. Moreover, it may cause memory fragmentation (free memory blocks are mixed with those in use what makes impossible allocating bigger block of memory if necessary) what in the end may cause disability to allocate necessary memory space. On more sophisticated machines (where memory management system is implemented) which can rearrange by itself memory blocks using memory mapping mechanism this problem may be unnoticed. For simple embedded system it may cause critical errors.

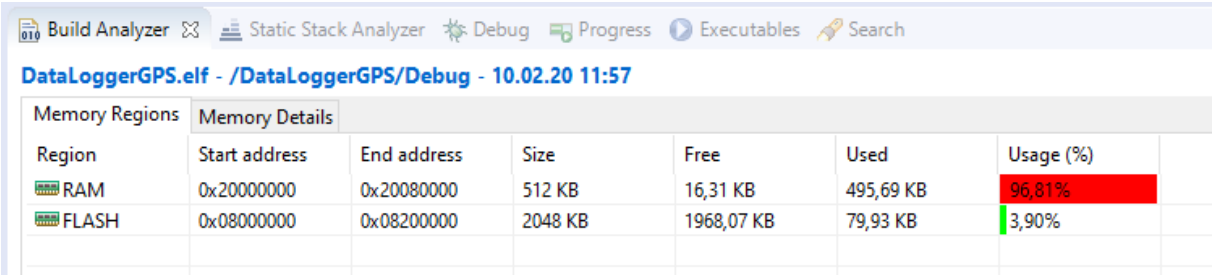



**Figure 5.5.** Used RAM estimation in Atollic TrueSUDIO for STM32 in the Data Logger project

In used IDE it is possible to estimate usage of RAM memory during execution. It is showed on figure 5.5. It would not be possible with dynamic allocation. In the project described here memory usage is about 96.8%.

## 5.9. Used data structures

For this project some of data structures was implemented by the author of this thesis. Let's describe briefly those non-standard.

### 5.9.1. FIFO Multiread

FIFO (First in, first out) Queue data structure is one basic one used in computer science. Concept of this container is to buffer elements in order which they arrive. Elements are being added using *emplace()* method. Element is added to the end of the queue. Removing elements from the queue is made in the same order which elements were emplaced (with *pop()* method) - elements arrived first are removed first (first in, first out). It implies that if element was read (with *pop()* method) by one reader/thread, it will not be read by another reader/thread.

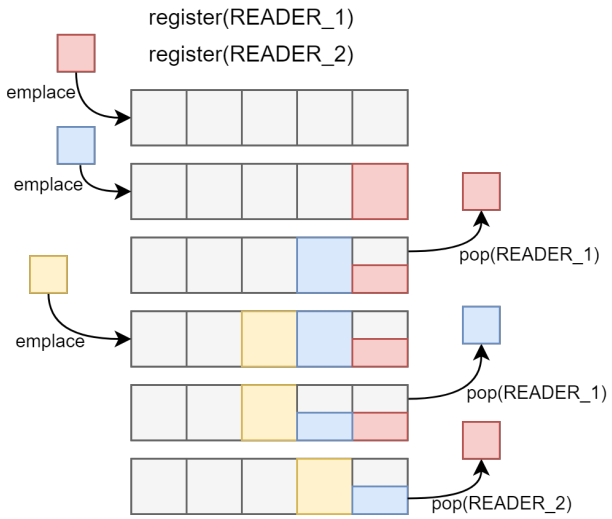



**Figure 5.6.** Example o usage of FIFO Multiread data structure for two readers.

In the project it was necessary for elements emplaced to the buffer (in ISR) was read in two contexts of code (in normal execution stream). Using multiple FIFOs structures would be very memory consuming. Author then decided to implement modified version of FIFO data structure which allowed multiple readers synchronisation.

If any thread wants to read from this queue, it must register at first as a reader. From this moment the Multiread FIFO module keeps track which elements was read by this reader. If this reader reads element from the structure, the module does not remove this element from the queue, but marks this element as read by this reader, but waits for the rest of readers to pop that element. When all readers read last element in the queue than it is removed from the structure. Figure 5.6 presents example of usage of this structure for two readers in a graphic way.

This structure uses cyclic-buffer method as a buffer for elements held in the structure. Structure is also prepared to be used in non-preemptive system where *emplace* operation is invoked from ISR routine and *pop* operation is called from normal execution stream.

### 5.9.2. Receiver Start-Term

NMEA sentences (described in chapter 2.2.5) may be easily recognised by start character ('$') and termination character (end-of-line '\n' ASCII char). It would be beneficial if any data structure would recognise whole sentences and provide them with API method as a whole sentence strings. The author of the thesis decided to write structure called *Receiver Start-Term*.

This module in initialisation function registers for callback from *UART Driver* when sign on UART bus is received. API provided for higher layer (GNSS Driver) let to get whole sentence starting with start sign (in this case '$') and ending with termination sign. Until that defined sentence will not be received this method will return status *Empty*.

Module counts start signs and terminations signs received and when both values are greater than 0 it means that complete sentence was received.

Correctness of *NMEA* sentence itself is checked by *GNSS Driver*.

### 5.9.3. Receiver Start-Length

For *UBX protocol* which is used for diagnostic communication with GNSS receiver other type of communication packages is used. Each message has known start character and its length. Thanks to that if the MCU send any diagnostic message to GNSS receiver answer for this would have known start character and known length.

Because of that author decided to implement data structure similar to *Receiver Start-Term* described previously, but with difference for recognising the end of the sequence: start sign (in UBX protocol it is byte equal to 0xB5) followed by registered number of bytes. If such defined sequence is received the user of the module can get the whole sentence using API function.

It is good to mention that UBX and GNSS message may be mixed on the bus. Because of that it was important to separate data structure handling both type of messages. One module handling both would break single-responsibility principle [25].

### 5.9.4. Fixed point

For storing fractional numbers in computer systems floating point is used most widely. This is solution finding golden mean between requirement of high range and good precision in the same time. Floating point numbers have very good resolution for small numbers but is getting lower when absolute value of the number grows. But range of the floating point may be much grater than for integer numbers. In embedded system used in automotive environment such trade-off is not acceptable.

Thanks to fact that range of measurements is limited (parameters from GNSS system) and accuracy is predefined, the accuracy and range of the fractional number is possible to be pre-defined.

Because of that facts better solution in this manner is to use fixed point number, where accuracy is determined by number of bits coding fractional part of the number and range defines number of bits in integer part [30]. Moreover, because binary point is known and must be the same for two number on which arithmetic operation is performed, integer numbers arithmetic may be used. That makes calculations much faster than using floating point operations, which are more complex.

Fixed point is used for recording latitude, longitude, horizontal and vertical dilution of precision, speed, track angle and altitude above mean sea level in GNSS data.

# 6. Software

Main part of the system is embedded device described in the above chapters. But to make it usable, an application being run on the PC computer is necessary. Main goals of this part of the system is preparing configuration file (in format described in details in chapter 4.1) and convert logged binary file described in 4.2 to standard *CSV* file format.

## 6.1. Technical requirements

- Graphical user interface (GUI) (in English language)
- **Configuration of the Data Logger**:
  - Interface for generating configuration file in *.aghconf* format for Data Logger device
  - Support for all functionalities of *.aghconf* file
  - Possibility of editing previously defined configurations
  - User can read/store this file from/to the file on computers disk or microSD card
- **Converting binary *.aghlog* file to *CSV* file format**:
  - Possibility to read and convert data from *.aghlog* format file to *CSV* format
  - Data should be possible to convert in two modes:
    - Raw data frames:
      Each logged CAN frame/GPS record/CAN Error in each row of CSV file. Row consists of timestamp, ID of frame (or GNSS or CAN Error row type note) and raw data fields:
      - for CAN frame - payload bytes, each in one column
      - for GPS record - each parameter of GPS package in one column
      - for CAN Bus Error - type of error type
    - With calculated CAN signals values (signals defined in 4.1 chapter) in each column. Raw CAN frame payload is not presented. Row consists of timestamp, ID of frame (or GNSS or CAN Error row type note) and raw data fields:
      - for CAN frame - calculated values of signals defined within received frame in each column
      - for GPS record - each parameter of GPS package in one column
      - for CAN Bus Error - type of error type
- Windows operating system support

## 6.2. Overall implementation description

Implemented application is stand-alone PC program. Whole application is implemented in C++ language (complied with g++ compiler, C++17 version) [31] using the *QT cross-platform library* for creating graphical user interface [32]. Project is written in *QT Creator* IDE.

Application is targeted to be mainly used by the mechanical engineers in *AGH Racing Team* wanted to log data mounted in racing car and analyse them using third-party tools like *Matlab* environment, *Microsoft Excel spreadsheets* or dedicated for motorsport software like *GEMS Data Analysis Professional* [33] used by the Team. That is the reason why the author decided to not to write any analysis functionalities within software part of the system but to write only a interface allowing users to export data in well-known format like CSV. Thanks to that users are able to import this data into mentioned applications which are very powerful.

## 6.3. User interface

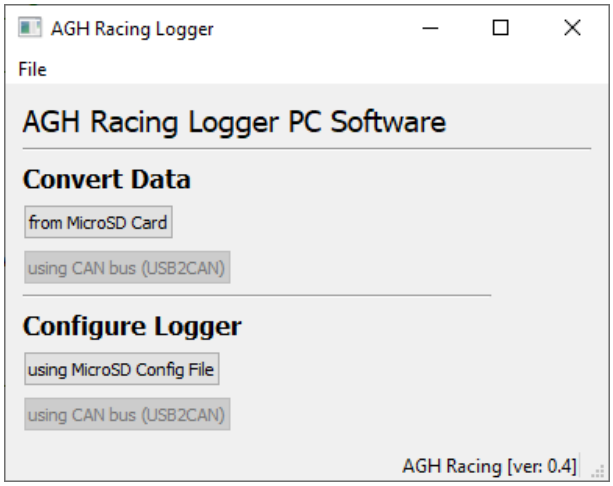



**Figure 6.1.** Main window of software application

After starting the application the main windows is displayed to the user. In this window, the user can choose which part of the application he wants to use: generating configuration file for the Data Logger device or converting *.aghlog* file to *CSV* type. Window is prepared to extend functionality of the application to implement both functionalities with *USB2CAN* [34] converter (without necessity of moving microSD card between PC computer and Data Logger device) in the future. Main window itself is presented on figure 6.1.

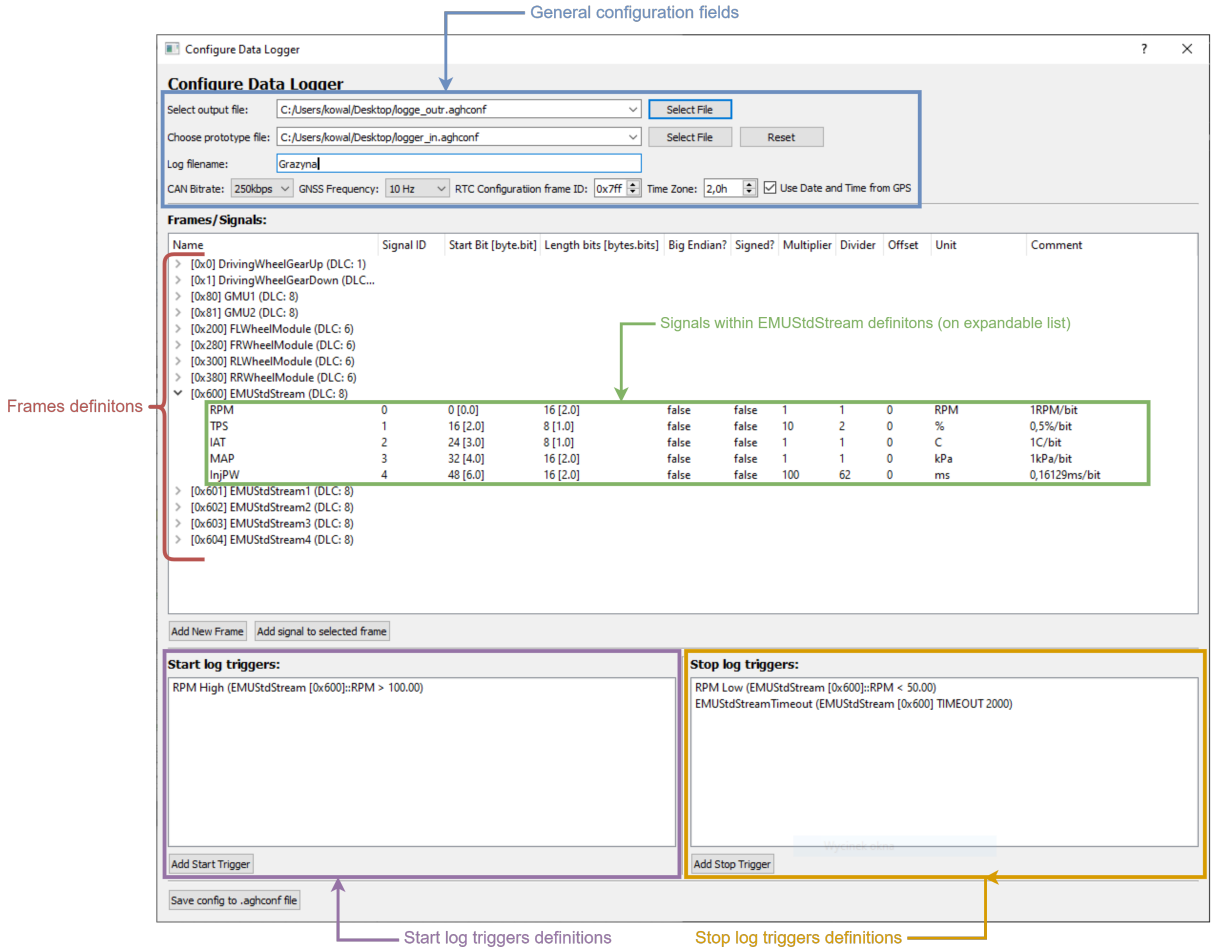



**Figure 6.2.** Configure windows with described each section

## 6.4. Configuration functionality

After clicking *using MicroSD Config File* in *Configure Logger* section widget with configuration options opens. Described here software provides interface for the user to configure all possible functionalities of the Data Logger. Window itself is presented on figure 6.2.

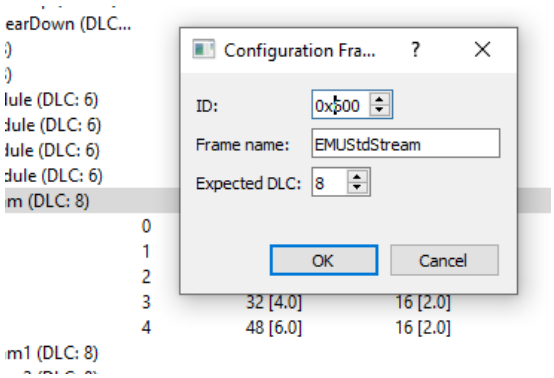



**Figure 6.3.** Widget for configuring parameters of CAN Bus frame

Using general configuration fields all fields not related to frames and signal may be configured, like CAN Bus baudrate, frequency of receiving GNSS data from GNSS module, or functionality of synchronising RTC with global time received in NEMA sentences.

In the center part of the window configuring definition of frames being received is possible. Frame definitions are displayed - each frame in each row. After clicking on that row using left mouse button it is possible to expand this frame definition with signals definitions defined within. When the user right click row with frame definition it is possible to delete or edit given frame. To add new frame "Add new frame" button is used. Window for editing/adding new frame is presented on figure 6.3

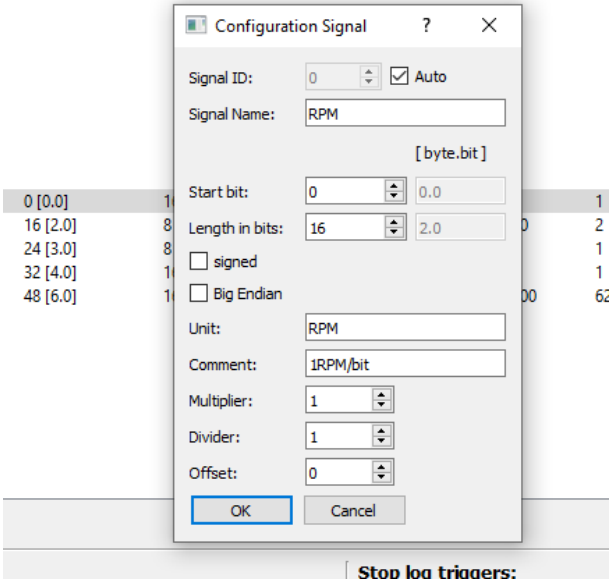



**Figure 6.4.** Widget for configuring parameters of Signal defined within CAN Bus frame

Signal, as described in 4.1 chapter is interpretation of bytes/bits within received CAN Bus frame payload. Parameters of the signal may be set using the window presented on figure 6.4.

Third widget possible to be run from configuration window is window for adding/editing start or stop trigger. This window displays definition of trigger referring to frames or signals (depending on chosen compare operator) defined in whole configuration. Using this window user can easily define condition which starts or stops logging.

## 6.5. Converting logged data functionality

Second main functionality of PC application is converting unreadable to humans binary *.aghlog* format to easy readable CSV format. *.aghlog* file format in its definition contains *.aghconf* file as prefix (metadata) thanks to that raw data saved may be easily interpreted.

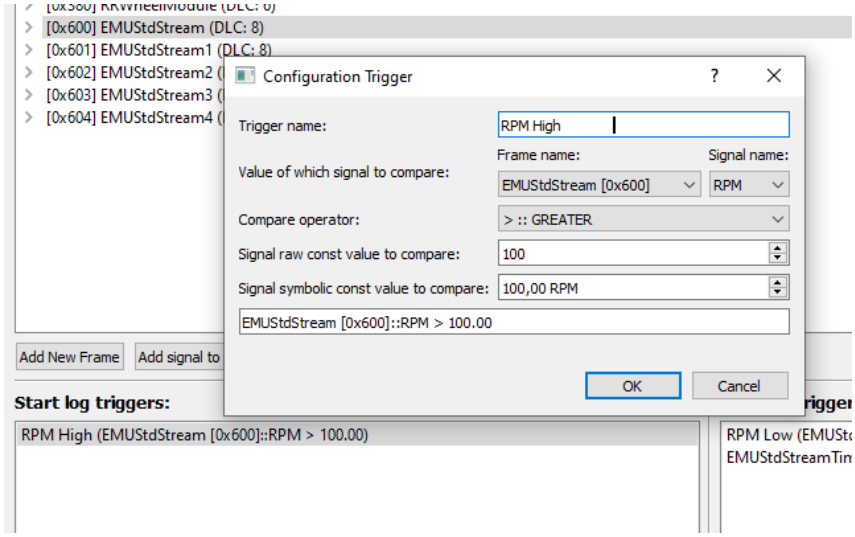



**Figure 6.5.** Widget for configuring start/stop trigger

Windows itself is very simple. On the top user can choose source directory (with *.aghlog* files - normally microSD card) and destination directory (where converted files should be saved). In the Advanced section mode of conversion may be chosen according to technical requirements:

- *Frame by frame mode* converts the file where each row (bearing timestamp and event type) contains:
  - for CAN frame - frame ID, DLC of frame and payload bytes, each byte in one column. Signals values are not calculated.
  - for GPS record - each parameter of GNSS package like latitude, longitude, speed, time, etc, each in one column in human readable form. That means if any value is represented by Fixed Point notation it is transformed to decimal number
  - for CAN Bus Error - type of error type in form of string description (bearing timestamp and event type) contains:

  Each of this row is saved with timestamp of occurred event timestamp.
- *Event Timing mode* converts the file where each row (bearing timestamp and event type) contains:
  - for CAN frame - each defined for given frame signal value, one channel in one column. That means that for received CAN frame value for each channel defined within this frame is calculated according to received frame payload and values of those signal are stored in CSV file. One signal value in one column
  - for GPS record - same as above
  - for CAN Bus Error - same as above

  The same as above, each of this row is saved with timestamp of occurred event timestamp.
- *Event Timing mode* converts the file in the same way as *Event Timing mode* but rows are stored for timestamps with static frequency: 10Hz, 100Hz 250Hz, 500Hz or 1000Hz. Let's call this timestamp values *sampling points*. That means if many events occurs between two sampling points, last

value is used. But if no event for given signal occurs between two sampling points, last value of signal is used.

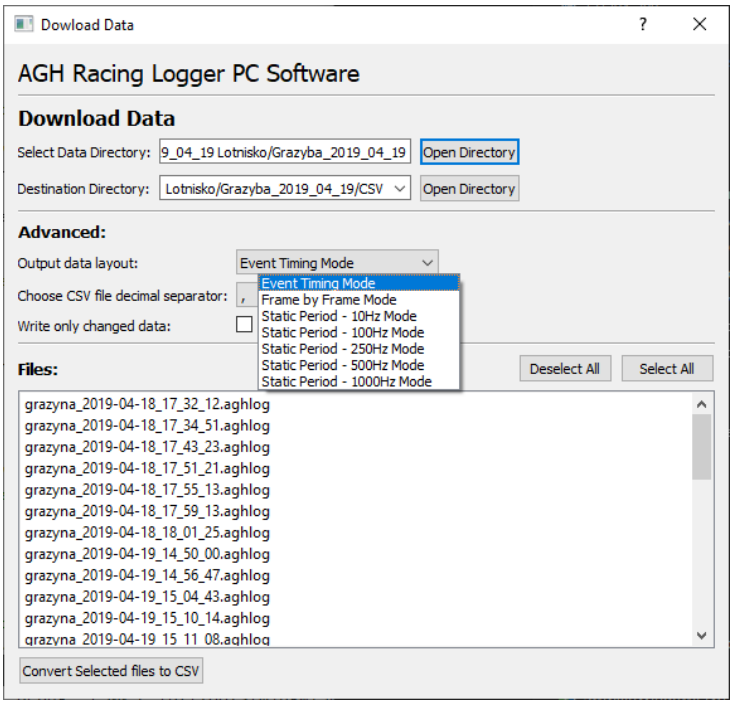



**Figure 6.6.** Widget for selecting files and method to conversion

Underneath two more setting are available: choosing between "." or "," decimal point separator and choice between saving all signals values for every row or only for signals with changed values. Second option reduces CSV file size letting many cell to be empty.

## 6.6. Event-based application

One of requirements for described here application is Graphical User Interface. In QT all events in GUI invokes callback methods. Because of that all functionalities are implemented in callback methods for events like changing value of widget.

Moreover, idea of the author was to separate model layer with user interface layer. Because of that object of Config class in all the time was holding actual values of each parameter configured by the user in GUI. That is one of min paradigms used in Model-View-Controller design pattern.Signal, as described in 4.1 chapter is interpretation of bytes/bits within received CAN Bus frame payload. Parameters of the signal may be set using the window presented on figure 6.4.

# 7. Tests

After implementation test of the Data Logger device was performed with with emphasis on CAN Bus frames logging functionality and GNSS data logging functionality.

## 7.1. CAN Bus logging functionality tests

The data logger was tested in the context of correctness of data logged on the CAN Bus and reliability of the system during test.

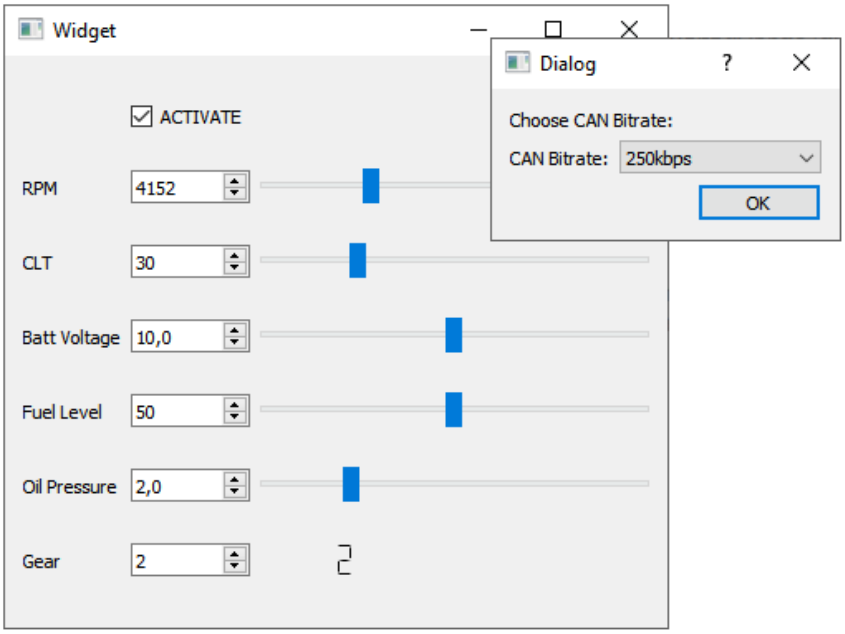



**Figure 7.1.** User interface of test application for sending CAN Bus frames with *ECUMASTER USB2CAN* device [34].

For stimulation of the Data Logger during development process *ECUMASTER USB2CAN* [34] device was used. Dynamic-Link Library (DLL) files provided by the manufacturer allowed to prepare simple test application which was sending frames to the bus. Thanks to using this device instead of real car bus made possible to test individual code modules without necessity of connecting Data Logger to real car CAN bus each time.

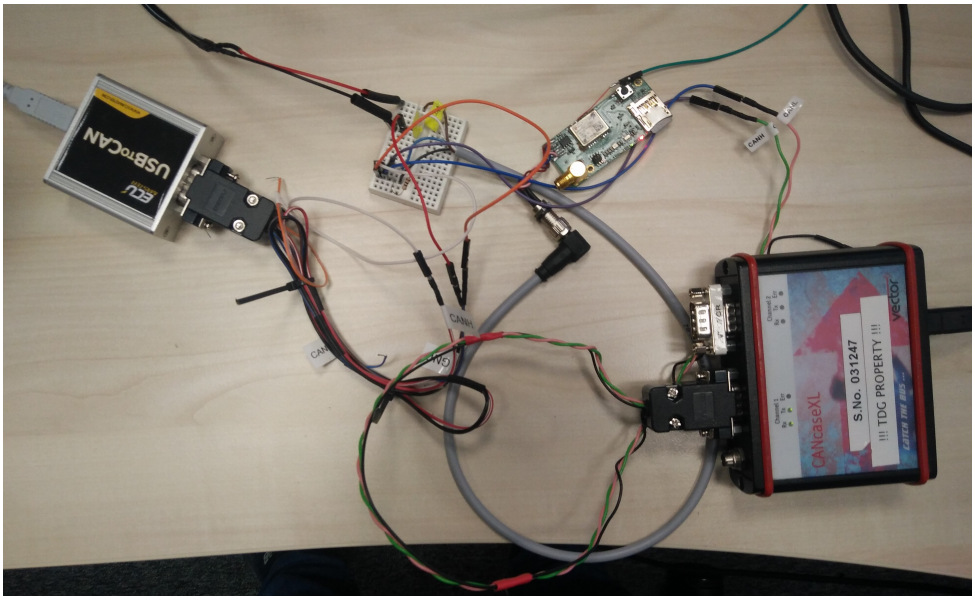



**Figure 7.2.** Test setup for validation tests

To validate correctness of frames being logged by the Data Logger and checking if requirement for ability to log data for more than 30 minutes **Vector CanCASE XL** device with **Vector CANoe** simulation environment was used. *USB2CAN* [34] device and the Data Logger was connected together with Vector CanCASE XL to the one bus.

## 7.2. Tests with 1Mb/s baudrate

One of the firmware requirements was capability of full functionality at over 90% bus load and 1Mb/s baudrate. Unfortunately the device did not pass this test. The device was working correctly when the bus load was lower than 50% but when bus load exceeded value of around 50% the device went to error state. Reason of that was CAN FIFO buffer overflow. Events with CAN bus data was arriving more often than the MCU core was able to process and save them to the microSD card.

```
kowal@Thinkpad-T440s-Michal MINGW64 ~/Desktop/AGHRacingRepos/GitHub_DataLogger_CAN_GPS/ASCToCSVConverter (master)
$ python ascToCsv.py Logging2020-02-23_18-50-03.asc Vector.csv
Converted to Vector.csv file.

kowal@Thinkpad-T440s-Michal MINGW64 ~/Desktop/AGHRacingRepos/GitHub_DataLogger_CAN_GPS/ASCToCSVConverter (master)
$ python compareLogsFiles.py Vector.csv Grazyna_2020-02-23_18_50_52.csv
removig line: time [ms];ID;DLC;data[0];data[1];data[2];data[3];data[4];data[5];data[6];data[7];CAN Error type;gps date[YYYY-MM-DD];gps time[HH:MM:S
S];longitude;latitude;satelites available;altitude [m];speed [km/h];track angle [deg];fix type {No fix|2D|3D};horizontalPrecision;verticalPrecision
;


Compare start!

Log files are identical. Compared 3709172 lines
```

**Figure 7.3.** Result of comparing file logged by *Vector CANoe* with *.aghlog* file logged by the Data Logger in 30 minutes test, 250Kb/s CAN Bus baudrate

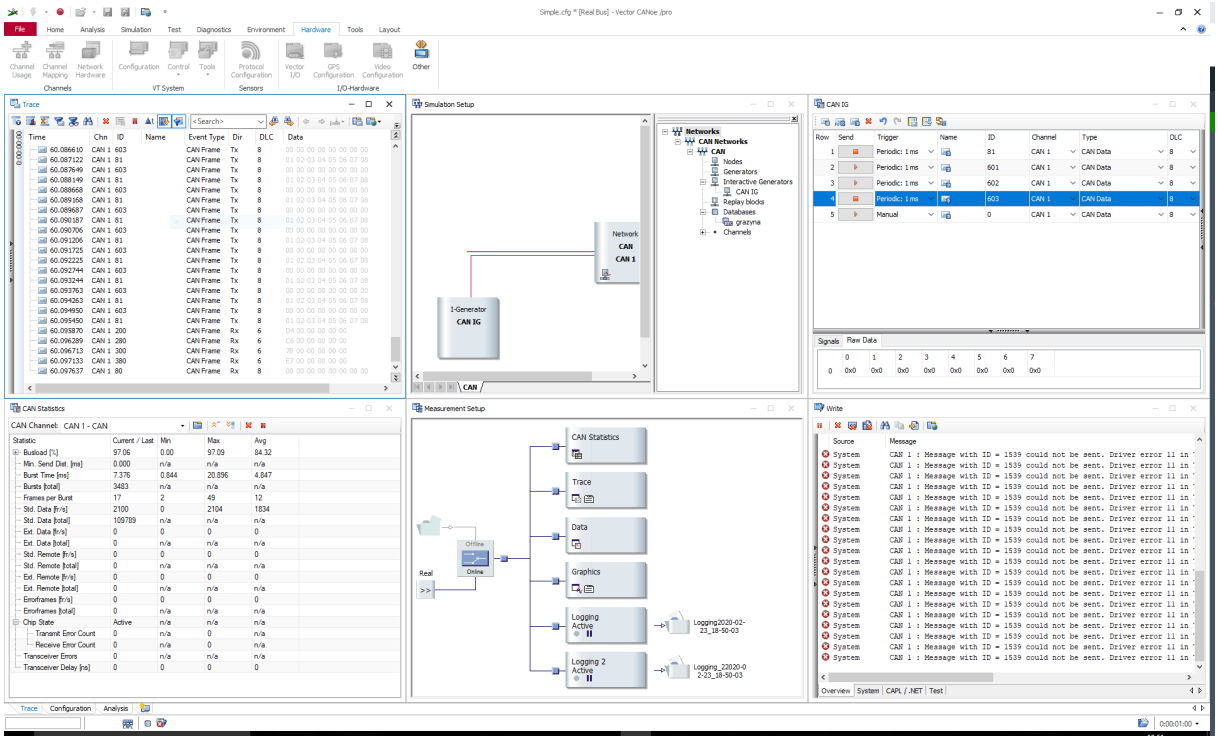


**Figure 7.4.** CANoe software configuration during the test

## 7.3. Tests with 250Kb/s baudrate

Because in the actual AGH Racing car 250Kb/s baudrate is used, author decided to perform the same test with lower baudrate of the bus.

During the test *USB2CAN* [34] device was sending all previously defined frames, what gave about 25% bus load. To reach bus load over 90% Vector simulation environment (CANoe + CanCASE XL) was sending (as fast as possible) frames with ID 0x81 and 0x603 what gave bus load about 97%.

Correctness of logged data was checked comparing *.ASC* log file saved by *Vector CANoe* environment with *.CSV* file converted from *.aghlog* file. Because formats are different, two special script in python programming language [35] was written.

First script extracts timestamp, ID, DLC and payload of following frames from *.ASC* file to match format of *.CSV* file converted from *.aghlog* file. Its main part is showed on listing 7.1.

Second script is comparing IDs, DLC and payload of following frames. Those parameters for following frames in both files must be exact the same to pass the test. Timestamps are ignored, sequence of frames matter. Script presented on listing 7.2.

Both files **did match** so the test was passed. The result is visible on figure fig:compareResult. Data Logger device was working as expected during the whole test.

Size of *.aghlog* file was 1.95 times higher than *.BLF* file. The *.aghlog* file size reached 51.8MB, when *.BLF* file had 26.6MB, but *.ASC* file (ASCII log file format from CANoe software) had 387MB size.

```
1 ...
```

```
for line in lines:
  splittedList = list(filter(None, line.split(' ')))
  try:
    DLC = int(splittedList[5])
  except:
    continue
  if int(splittedList[2],16) == 1024:
    continue
  fileOut.write(splittedList[0] + ";") # timestamp
  fileOut.write(str(int(splittedList[2],16)) + ";") # ID
  fileOut.write(str(int(splittedList[5],16)) + ";") # DLC
  for index in range(0, DLC):
    fileOut.write(str(int(splittedList[6+index],16)) + ";") # payload byte[index]
  fileOut.write("\n")

fileOut.close()
print ("Converted to " + fileOut.name + " file.")
```

**Listing 7.1.** Main part of Python ASC to CSV file converter script

```
...

print ("Compare start!\n")

for line1, line2 in zip(lines1, lines2):
  splittedLine1 = list(filter(None, line1.split(';')))
  splittedLine2 = list(filter(None, line2.split(';')))

  if splittedLine1[1] != splittedLine2[1]:
    print ("ID of frame in " + str(linesCounter) + " line is different\n")
    exit()

  if splittedLine1[2] != splittedLine2[2]:
    print ("DLC of frame in " + str(linesCounter) + " line is different\n")
    exit()

  for index in range(0, int(splittedLine1[2])):
    if splittedLine1[3+index] != splittedLine2[3+index]:
      print ("payload of frame in " + str(linesCounter) + " line is different\n")
      exit()

  linesCounter+=1;

print ("Log files are identical. Compared " + str(linesCounter) + " lines")
```

**Listing 7.2.** Main part of script comparing CSV raw log files

## 7.4. GNSS data logging functionality tests

For localisation logging functionality manual tests was performed. UART bus between GNSS U-blox receiver and MCU was sniffed by UART to USB converter and than *U-Blox U-Center* application was used to interpret sniffed NMEA sentences. Than results from mentioned application and converted *.aghlog* file was compared manually. Over a dozen of samples was checked and result of the test was positive.

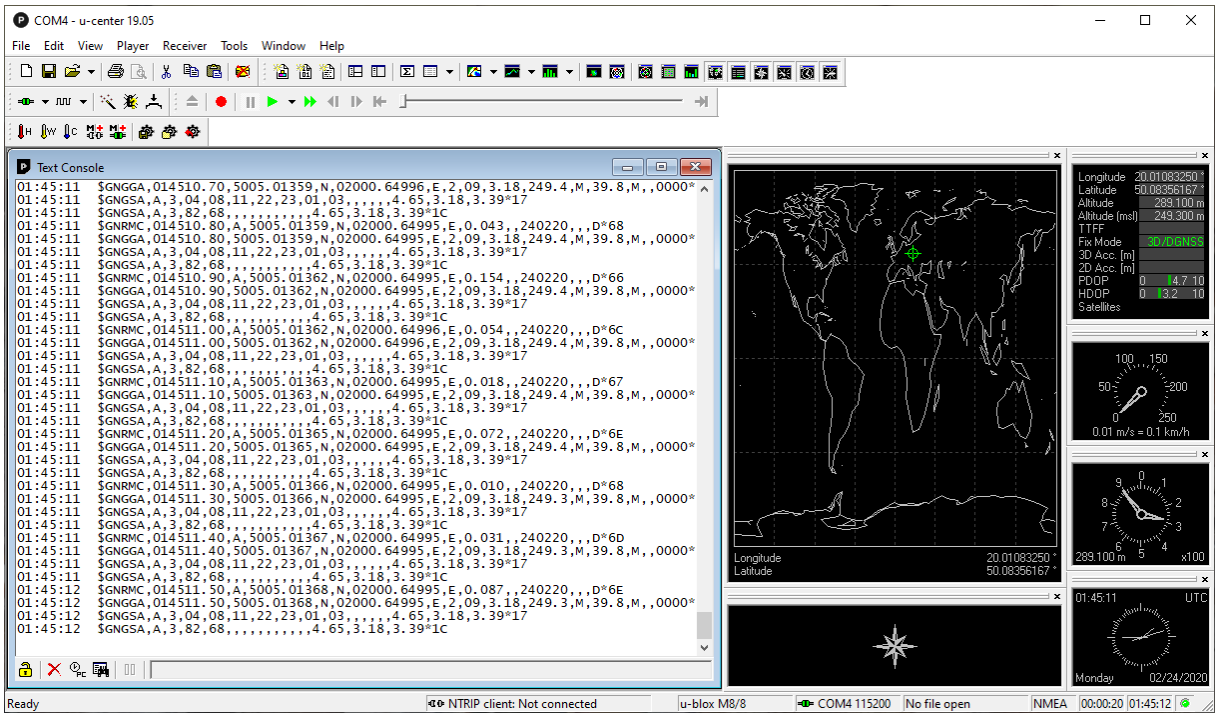



**Figure 7.5.** Overview of *U-Blox u-center* software

# 8. Summary

Main goal of this thesis was to design the data logging system for AGH Racing car. System is able to log data from the CAN Bus and GNSS system.

System consists of embedded Data Logger device and PC software for converting binary *.aghlog* custom file format to *CSV* file format. Configuration is prepared in the same PC application and is in custom *.aghconf* file format. In this file configuration for peripherals (mainly CAN Bus and GNSS receiver) is stored and list of frames with definitions of signal those contains.

Project was conducted in consecutive design stages: selection of electronic components (mainly MCU and GNSS receiver), PCB design and manufacturing, development of firmware for embedded device and development of GUI application for PC computer.

## 8.1. Conclusions

All the steps listed above have been completed. All required functionalities are implemented and works properly for normal working conditions. Moreover, project works properly in the worst case scenario for CAN Bus with baudrate equal to 250Kb/s. This baudrate is used in *AGH Racing* combustion car. For higher CAN bus baudrates improvements are necessary for worst-case scenarios.

Developing data logging device by yourself may reduce cost of that system repeatedly. Devices available on the market are very expensive but making this system reliable and ready to work properly in worst-case scenarios requires very high engineering effort and sophisticated testing methodology. Making fast and fault tolerant firmware is really time-consuming project. In this version firmware (excluding third-party libraries like HAL) contains over 9500 lines of code and PC application around 7000.

## 8.2. Possible improvements

First necessary improvement is refactoring firmware code to be able to log all frames with maximum CAN Bus speed and over 90% bus load. Unfortunately device reports an error (buffer overflow occurs) in these conditions. Increasing CPU clock frequency may be one of solutions, but this requires hardware design change. Changing arrangement of decoupling capacitors connected to the MCU is necessary for speeding up the MCU.

Another possible improvements in hardware design:

– adding poly-fuse protection for power supply line
– replacing LDO with step-down regulator what would reduce temperature of the board
– use SD-Card 4-bit protocol version instead of 1-bit version

Possible improvements in firmware and software:

– Adding handling 29-bit CAN frame identifiers
– Adding value-table functionality - each signal value might have assigned name
– Adding support for configuring the device and downloading data via CAN Bus
– Adding internal errors logger

# Bibliography


[1] Institution of Mechanical Engineers. *About Formula Student*. 2019. URL: *http://www.imeche.org/events/formula-student/about-formula-student* (visited on 2019-01-06).

[2] *2017-18 Formula SAE Rules*. SAE International. 2016.

[3] Herner A i Hans-Jürgen R. *Elektrotechnika i elektronika w pojazdach samochdowych*. Warszawa: Wydawnictwa Komunikacji i Łączności, 2012, s. 253–259.

[4] *ECUMASTER PMU-16/PMU-16DL Preliminary Manual*. 1.02. ECUMASTER. June 2018.

[5] *CAN bus, From Wikipedia, the free encyclopedia*. 2019. URL: *https://en.wikipedia.org/wiki/CAN_bus*.

[6] *CAN-Bus-frame in base format without stuffbits*. 2014. URL: *https://commons.wikimedia.org/w/index.php?curid=31571749*.

[7] *Controller Area Network (CAN Bus) - Bus Arbitration*. 2018. URL: *https://copperhilltech.com/blog/controller-area-network-can-bus-bus-arbitration/*.

[8] *CAN Bus Error Handling, Kvaser*. 2019. URL: *https://www.kvaser.com/about-can/the-can-protocol/can-error-handling/*.

[9] *Official U.S. government information about the Global Positioning System (GPS) and related topics*. 2020. URL: *https://www.gps.gov*.

[10] Paweł Gryboś. *Zintegrowane Czujniki Pomiarowe, Wybrane zagadnienia i przykłady*. Kraków: Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie, 2011, s. 115–135.

[11] *NMEA data*. 2020. URL: *https://www.gpsinformation.org/dale/nmea.htm*.

[12] *Reference manual, STM32F76xxx and STM32F77xxx advanced Arm®-based 32-bit MCUs*. RM0410 Rev 4. Mar. 2018.

[13] *SanDisk Secure Digital Card*. 1.9. Dec. 2003.

[14] *FatFs - Generic FAT Filesystem Module*. 2020. URL: *http://elm-chan.org/fsw/ff/00index_e.html*.

[15] *What is File System?* 2019. URL: *https://www.computerhope.com/jargon/f/filesyst.htm*.

[16] *Developing applications on STM32Cube™ with FatFs*. UM1721 Rev 3. Feb. 2019.

[17] *Description of STM32F7 HAL and Low-layer drivers*. DocID027932 Rev 3. Feb. 2017.

[18] *Lithium polymer battery, From Wikipedia, the free encyclopedia*. URL: *https://en.wikipedia.org/wiki/Lithium_polymer_battery*.

[19] *NEO-M8 series Versatile u-blox M8 GNSS modules*. URL: *https://www.u-blox.com/en/product/neo-m8-series*.

[20] *NCP1117, NCV1117 Datasheet, 1.0 A Low-Dropout Positive Fixed and Adjustable Voltage Regulators*. 28th ed. Jan. 2017.

[21] *Climate of Barcelona, from Wikipedia, the free encyclopedia*. URL: *https://en.wikipedia.org/wiki/Climate_of_Barcelona*.

[22] *TGL34-6.8 ... TGL34-200CA SMD Transient Voltage Suppressor Diodes*. 2018-02-01. Feb. 2018.

[23] *Vector Knowledge Base, Logging Formats*. URL: *https://kb.vector.com/entry/520/*.

[24] *Refactoring Guru, Facade Design Pattern*. URL: *https://refactoring.guru/design-patterns/facade*.

[25] *Principles of Object Oriented Design*. URL: *http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod*.

[26] *Refactoring Guru, State Behavioral Patterns*. URL: *https://refactoring.guru/design-patterns/state*.

[27] *Object-Oriented Programming (OOP) in C*. URL: *https://www.codementor.io/@michaelsafyan/object-oriented-programming-in-c-du1081gw2*.

[28] *Linux Programmer's Manual, ERRNO*. URL: *http://man7.org/linux/man-pages/man3/errno.3.html*.

[29] *Dynamic memory and heap contiguity*. URL: *https://www.embedded.com/dynamic-memory-and-heap-contiguity/*.

[30] Hayden So. *Introduction to Fixed Point Number Representation*. Feb. 2006. URL: *http://www-inst.eecs.berkeley.edu/~cs61c/sp06/handout/fixedpt.html*.

[31] *cppreference.com, C++17*. URL: *https://en.cppreference.com/w/cpp/17*.

[32] *Qt | Cross-platform software development for embedded desktop*. URL: *https://www.qt.io/*.

[33] *GEMS Data Analysis Software*. URL: *https://gems.co.uk/products/software/gda/*.

[34] *ECU Master USBtoCAN device*. URL: *https://www.ecumaster.com/products/usb-to-can/*.

[35] *Python programming language main page*. URL: *https://www.python.org/*.