

Master Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Control Engineering

Simulator of wireless tire-pressure sensors

Bc. Martin Pešek

Supervisor: Ing. Michal Sojka, Ph.D.
Field of study: Cybernetics and Robotics
Subfield: Cybernetics and Robotics
May 2020

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Pešek** Jméno: **Martin** Osobní číslo: **456926**
Fakulta/ústav: **Fakulta elektrotechnická**
Zadávající katedra/ústav: **Katedra řídicí techniky**
Studijní program: **Kybernetika a robotika**
Studijní obor: **Kybernetika a robotika**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Simulátor bezdrátových senzorů pro měření tlaku v pneumatikách

Název diplomové práce anglicky:

Simulator of wireless tire-pressure sensors

Pokyny pro vypracování:

1. Make yourself familiar with methods for automatic measurements of tire pressure in cars.
2. Analyze the functionality of tire-pressure monitoring sensors planned for use in future Skoda Auto cars.
3. Propose a hardware solution for simulating those sensors at wireless protocol level and build a hardware prototype, which can be controlled via USB-based serial line or CAN bus.
4. Design and develop software for controlling the simulator. The software will consist of a GUI part running on a PC and device firmware.
5. After successful testing and evaluation, update the hardware design for small series production.
6. Document the results.

Seznam doporučené literatury:

- [1] Volkswagen AG, Diagnosedokumentation RDK BERU 30, 2017
[2] Volkswagen AG, Reifendrucküberwachungssysteme – Konstruktion und Funktion

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Michal Sojka, Ph.D., vestavěné systémy CIIRC

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **17.01.2020**

Termín odevzdání diplomové práce: _____

Platnost zadání diplomové práce: **30.09.2021**

Ing. Michal Sojka, Ph.D.
podpis vedoucí(ho) práce

prof. Ing. Michael Šebek, DrSc.
podpis vedoucí(ho) ústavu/katedry

prof. Mgr. Petr Páta, Ph.D.
podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

Datum převzetí zadání

Podpis studenta

Acknowledgements

A great many thanks belong to my supervisor for leadership, help, and trust in my abilities even during the hard-looking times.

Thanks as big belongs to colleagues who helped me during the initial phases as a part of a team project. I am also grateful to friends and family for their long-term support with my demanding studies.

Given the difficult times of the pandemic when writing this thesis, I wish to express my gratitude to everyone who spent their time (including extra hours) and those who sacrificed the little of leisure time they had left to provide support in any form for our hospitals, elderly and everyone who needed it. As an owner of a small company, I hope many of us who had to attenuate or end their business and were left alone to struggle, will soon be able to rise from the ashes with a new outlook on things and path to walk on.

Declaration

I hereby declare that I have completed this thesis individually and that I have listed all used information sources in accordance with “Metodický pokyn o držování etických principů při přípravě vysokoškolských závěrečných prací”.

In Trutnov, 8 May 2020

Abstract

The aim of this thesis is to create a simulator of a set of smart wireless valves for tire pressure monitoring systems (TPMS). The simulator is planned for use in the testing department of ŠKODA AUTO.

The thesis describes the methods of analysis of ŠKODA AUTO vehicles' TPMS protocol, and also the design and the process of realizing the TPMS Simulator based on the *CC1101* radiofrequency integrated circuit and *STM32F446* microcontroller. The TPMS protocol specification is generally undisclosed by the manufacturers, and due to the intellectual property protection, we describe it only partly. We have also developed a *Qt*-based Graphical User Interface to accompany the physical simulator device (further referred to as the Transmitter unit) with the possibility of configuration over a computer.

The implementation comprises both a prototype device as well as a version suitable for production in smaller series. The device was successfully tested in the ŠKODA AUTO laboratories on a prototype of a new ŠKODA KODIAQ car, and has been handed over for further use.

Keywords: TPMS, Tire Pressure Monitoring System, RDK, STM, CAN

Supervisor: Ing. Michal Sojka, Ph.D.

Abstrakt

Cílem této práce bylo vytvořit simulátor sady chytrých bezdrátových ventilků systému pro měření tlaku v pneumatikách auta (TPMS). Tento simulátor se plánuje použít pro potřeby testovacího oddělení ŠKODA AUTO.

Tato práce popisuje způsoby analýzy TPMS protokolu ve vozech ŠKODA AUTO a také návrh a průběh realizace TPMS simulátoru založeného na radiofrekvenčním integrovaném obvodu *CC1101* a mikrokontroléru *STM32F446*. Specifikace TPMS protokolu nejsou všeobecně zveřejňovány výrobcí a kvůli ochraně duševního vlastnictví je popisujeme pouze částečně. Také jsme vyvinuli se základem v knihovnách *Qt* vlastní grafické uživatelské prostředí, které doplňuje fyzické zařízení simulátoru o možnost konfigurace z počítače.

Implementace se skládá jak z prototypu, tak finální verze vhodné pro menší sériovou výrobu. Naše zařízení bylo úspěšně otestováno v laboratořích ŠKODA AUTO na prototypu nového vozu ŠKODA KODIAQ a bylo předáno k dalšímu užívání.

Klíčová slova: TPMS, Systém měření tlaku v pneumatikách, RDK, STM, CAN

Překlad názvu: Simulátor bezdrátových senzorů pro měření tlaku v pneumatikách

Contents

1 Introduction and motivation	1	4.4.2 Initialization	38
2 Theoretical background	3	4.4.3 Main loop	39
2.1 TPMS	3	4.4.4 Modules	40
2.1.1 Susceptibility to misuse	5	4.5 GUI	41
2.2 Digital communication	5	4.5.1 Front-end	41
2.2.1 Shannon model of digital communication	5	4.5.2 Back-end	43
2.2.2 Software-Defined Radio	8	5 Implementation process	47
2.2.3 IQ modulation and demodulation	8	5.1 Prototype	47
3 TPMS protocol analysis	11	5.2 Small series device	48
3.1 SDR data recording	11	5.2.1 Hardware	49
3.1.1 TPMS smart valves' transmission activation	12	5.2.2 Firmware	51
3.2 Transmission parameters identification	13	5.2.3 CAN Command-line configurator	53
3.2.1 Carrier frequency	14	5.3 Design verification	54
3.2.2 Modulation	14	6 Conclusion	57
3.2.3 Baud rate	14	A List of used acronyms	59
3.2.4 Frame encoding	15	B TPMS Simulator communication protocol description	63
3.2.5 Other parameters	16	B.1 Query structure	64
3.3 Data interpretation	16	B.1.1 Commands definition	64
3.3.1 Checksum cracking	17	B.1.2 Query examples	66
3.3.2 Identification of a TPMS valve	18	C Complete schematics and PCB layout	69
3.3.3 Pressure measurements	18	D IAR EW & Mbed vs GNU Arm Toolchain & HAL comparison	73
3.3.4 Rotational velocity measurements	19	D.1 IAR EW vs GNU Arm Toolchain	73
3.3.5 Temperature measurements	20	D.1.1 Licensing	73
3.3.6 Miscellaneous data	21	D.1.2 Functional safety	73
3.4 CAN analysis	21	D.1.3 IDE functionality	73
4 Simulator design concept	23	D.1.4 Miscellaneous	74
4.1 Hardware	24	D.1.5 Conclusion	74
4.1.1 Power supply	25	D.2 Mbed vs HAL libraries usage	74
4.1.2 Microcontroller unit	27	D.2.1 Abstraction	74
4.1.3 Radiofrequency integrated circuit	30	D.2.2 Miscellaneous	75
4.1.4 EEPROM	31	D.2.3 Conclusion	75
4.1.5 LEDs	31	E Bibliography	77
4.1.6 USB	32		
4.1.7 CAN	33		
4.1.8 FTDI converter	34		
4.2 PCB design	34		
4.3 Mechanics	36		
4.4 Firmware	37		
4.4.1 Structure	38		

Figures

1.1 TPMS smart valve	1	4.1 <i>Interactive</i> mode - connection of the Transmitter unit to a computer with the configuration software over the USB or CAN	23
1.2 TPMS ECU	2	4.2 <i>Standalone</i> mode - connection of the Transmitter unit to a source of electrical power	24
2.1 Scheme of a possible TPMS configuration	3	4.3 Block diagram of the Transmitter unit	25
2.2 TPMS warning icon based on Docket No. NHTSA 2004-19054 [1]	4	4.4 Power jack circuitry	25
2.3 Shannon model of the digital communication system [2]	6	4.5 Power supply mixing and the overvoltage/overcurrent protection	26
2.4 BFSK constellation diagram	7	4.6 DC/DC converter circuitry	27
2.5 BFSK modulated signal example	7	4.7 MCU circuitry	28
2.6 Simplified SDR schematic based on [3, p. 7]	8	4.8 Pin header connector for Serial Wire Debug (SWD) programming	29
2.7 IQ modulator	9	4.9 CC1101 RF IC pluggable module	30
2.8 IQ demodulator	9	4.10 Connector used for CC1101 module	31
3.1 <i>Gqrx</i> interface	12	4.11 EEPROM circuitry	31
3.2 Attempt at activating the TPMS transmission with an embedded valve	13	4.12 LEDs circuitry	32
3.3 Welded wheel holder with a rotatable plate	13	4.13 USB circuitry	32
3.4 IQ data of a located TPMS messages surrounded by noise visualised in <i>Audacity</i>	14	4.14 CAN circuitry	33
3.5 Example of <i>rtl_433</i> tool decoding raw IQ data into bits (respectively hexadecimal) sequences	15	4.15 FTDI converter connector	34
3.6 <i>CRC RevEng</i> tool example – input messages with appended checksum yield the used CRC polynomial along with its properties	18	4.16 PCB block scheme	35
3.7 Pressure measurements in an enclosed container	19	4.17 PCB visualisation	36
3.8 Rotational velocity measurements on a valve attached to a drill	19	4.18 <i>Hammond</i> 1455N1201 enclosure [4]	36
3.9 Rotational velocity measurements on a valve externally attached to a wheel of a car	20	4.19 Front panel graphics and milling marks	37
3.10 Temperature measurements in warm water in home environment	21	4.20 Back panel graphics and milling marks	37
3.11 TPMS ECU with an adapter attachable to a CAN interface device	22	4.21 FW start-up diagram	39
		4.22 FW main loop diagram	40
		4.23 FW <i>Main</i> module dependency graph	41
		4.24 GUI <i>Main</i> window	42
		4.25 GUI <i>Settings</i> window	43
		4.26 GUI <i>Main</i> module dependency graph	44
		4.27 <i>CommExchange</i> class inheritance diagram	44
		4.28 GUI class diagram	45
		5.1 Prototype internals – NUCLEO-F401RE and the CC1101 module	47

5.2 Finished prototype – front view .	48
5.3 Finished prototype – back view .	48
5.4 Finished small series device – front view	49
5.5 Finished small series device – back view	49
5.6 Assembled and working PCB...	50
5.7 CAN analysis using the <i>Little Embedded Oscilloscope</i>	51
5.8 <i>STM32CubeProgrammer</i> – a possible software solution for managing the DFU of the MCU ..	52
5.9 CAN TPMS Simulator Configurator	53
5.10 Infotainment TPMS screen [5] .	54
C.1 Power supply and USB schematics	69
C.2 MCU and peripherals schematics	70
C.3 Top PCB layer	71
C.4 Bottom PCB layer	72

Tables

5.1 Power consumption measurements	50
B.1 Control commands set	64
B.2 Wheel index values	65

Chapter 1

Introduction and motivation

In order to increase road safety, comfort and decrease the gas consumption, the car tires pressure should be periodically monitored to see, if they have the correct prescribed values. Tire pressure monitoring system (TPMS) primarily allows the on-line measuring of the pressure inside the car's tires and ultimately passes this information to the car's Electronic Control Unit (ECU). However, it may also measure the temperature or other physical quantities.

The main benefits consist of increased safety, comfort, and ecological savings. However, apart from these arguments, there are also some downsides. In general, these include increased costs [6] both for the production and the after-sale services (ultimately burdening the customer) [7, p. 2-8][6, p. 214][8], the possibility of the malicious misuse and increased difficulty of the production testing. The last two topics are also the reasons why this thesis was written.

Nowadays, the TPMS has to be compulsorily implemented in newly produced vehicles in many significant countries (such as the USA and the states of the EU) [9] or such legislation is in preparation or will most likely be in the following years [6, p. 214][10, p. 5][11, 12, 13]. For these reasons, the boom of TPMS market is expected [14, 15].



Figure 1.1: TPMS smart valve

Our task was to create a simulator capable of forcing the user-selected values of wheels' pressure and temperature to the car. The device has to masquerade as the TPMS sensors (as seen in Figure 1.1), hence providing the

TPMS ECU (displayed in Figure 1.2) with fake measurements. Ultimately, this inserts the bogus data as real to the car, which takes further action.



Figure 1.2: TPMS ECU

The primary motivation was the ŠKODA AUTO's wish to abstract the tests that include tires onto a higher level because until now, a physical change of tire pressure had to be done. With our device, however, the user may simply input any desired values, and the TPMS Simulator will effectively transfer this information to the car's ECU.

The sensors principle of operation and communication parameters were unknown and had to be analyzed first. This fact leads us to the essential secondary motivation – the reliability of the TPMS against hackers, susceptibility to misuse, and severity of possible end-effect of these attacks.

This thesis is structured as follows. Chapter 2 describes the theoretical background behind the TPMS and the necessary principles of digital communication. In Chapter 3, the process of TPMS protocol analysis is described – only partially due to the Intellectual Property (IP) protection. In Chapter 4, the TPMS Simulator design concept is introduced and in Chapter 5 its implementation is described. We provide our conclusions in Chapter 6.

Chapter 2

Theoretical background

Let us now introduce the theory behind the TPMS, its use, functionality, and components. Firstly, we will describe the system itself from the practical point of view, secondly we shall analyze and bring up the theory behind its elements.

2.1 TPMS

TPMS (also often abbreviated as RDK from German *Reifendruckkontrolle*) is a system that can measure and evaluate physical quantities inside a car's wheel tire, mainly pressure but one may also often see the temperature, wheel's angular velocity, battery status measurements and more.

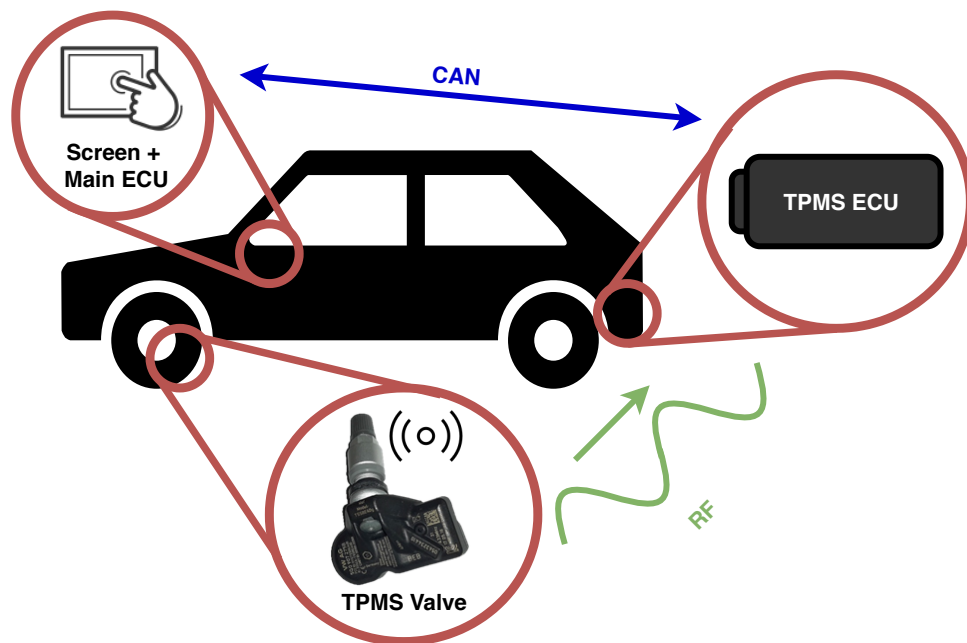


Figure 2.1: Scheme of a possible TPMS configuration

As illustrated in Figure 2.1, the values measured by the smart valves (end-sensors partially fitted inside the tire) are sent to the TPMS ECU, which

further processes them and finally provides them (or some of them) to the car. In our case, the values from the valves are transferred over radio link and from the TPMS ECU to the rest of the car using the Controller Area Network (CAN) bus [16] with its specific protocol [17].

The history of the TPMS dates back to 1986 when it was first fitted to a Porsche 959 [18, p. 124] which also interestingly used Bridgestone's first run-flat tires where the user may not easily tell the tires are flat [19]. During the decade of 1990, more car manufacturers have implemented this system. Upon a great recall of Firestone tires, a Tire Recall Enhancement, Accountability, and Documentation (TREAD) Act has been passed by the USA Congress effectively mandating the use of the TPMS in the new series of cars in the USA [20]. A similar law has been legislated in the European Union, taking effect in 2014. Many other major countries are taking similar actions [6, p. 214][10, p. 5][11, 12, 13]. The TPMS is often signaled by the symbol in Figure 2.2.



Figure 2.2: TPMS warning icon based on Docket No. NHTSA 2004-19054 [1]

The main benefits of the TPMS are the following [7, p. 2-8][6, p. 214][8]:

- increased road safety by warning the driver of incorrect pressure,
- greater user comfort of both checking the pressure and the drive itself,
- improved ecological and economical effect by using less fuel, producing less emission, and prolonging the life of tires.

However, there also come some downsides:

- greater costs for production, after-sale services [7, p. 2-8], and ultimately the customer,
- increased difficulty of production tests by adding another degree of freedom in terms of failures and their concurrence,
- susceptibility to malicious misuse.

TPMS is implemented with one of the two technologies. One is called the direct TPMS (dTPMS) since it directly measures the pressure inside the tire. The second is the indirect one (iTPMS), the sensor is not placed inside the wheel and it detects the magnitude of pressure by other means (wheels' different angular velocity taken from the Anti-lock Brake System (ABS) sensors [21], using the spectrum analysis [22, 23], and also other solutions are being proposed [24]). The selection of either one is a trade-off between purchase costs and sensor accuracy or precision with the latter being less expensive and less accurate and precise and vice versa. Also, the dTPMS

requires an internal power supply (such as a battery) and provides greater modularity – one may buy a system not connected directly to the car but rather an aftermarket portable device with a display that shows the measured values. At the same time, the pressure measurements with a non-moving car may not be possible with conventional iTPMS. Also, relative measurements may pose another inaccuracy dangers.

The device developed in this thesis acts as dTPMS smart valves simulator.

■ 2.1.1 Susceptibility to misuse

Another question that is raised is that of the general susceptibility of the TPMS to malicious attackers. If the wireless communication between the smart valves and the TPMS ECU is ease to eavesdrop on, e.g., when not encrypted, it implies potential danger. After decoding the communication protocol, the potential attacker may understand it to such an extent that the replication of the original messages sent from the smart valves is possible by masquerading the custom messages as the legit messages.

This fact implies that it would be possible to find out the identification numbers (IDs) of smart valves of a selected car and send messages that pretend to come from these valves with any (!) legal values of measured physical entities. This means that the car under test may interpret that, e.g., the pressure of a tire is of any value, that the hacker desires, and takes appropriate action – warns the car user. It is, e.g., possible for the hacker to make the driver think that his wheel or wheels are defective or with decreasing pressure at an arbitrary rate. This fact may result in forcing him to stop his ride or not initiate it at all.

It may also be possible to follow a movement of a specific person or car by creating a receiver network and tracking the TPMS signals with unique ID.

The essential solution to this problem however exists. It is possible to implement the encryption of the radio frequency (RF) communication between the smart valves and the TPMS ECU. Without the knowledge of the private encryption keys, the hacker's options to decode the protocol would become substantially thinner and much more complex if not impossible (methods such as brute-force cracking the keys, smart valve's processor memory readout or code injections). Even if all the communication is unidirectional and would be based on a single encryption key, it would still be better than none.

■ 2.2 Digital communication

Analyzing our TPMS protocol requires some background knowledge of methods used for transmitting information in our case over the RF link.

■ 2.2.1 Shannon model of digital communication

We may conveniently utilize the Shannon model of the digital communication system to segment our single analysis tasks [25], illustrated in Figure 2.3.

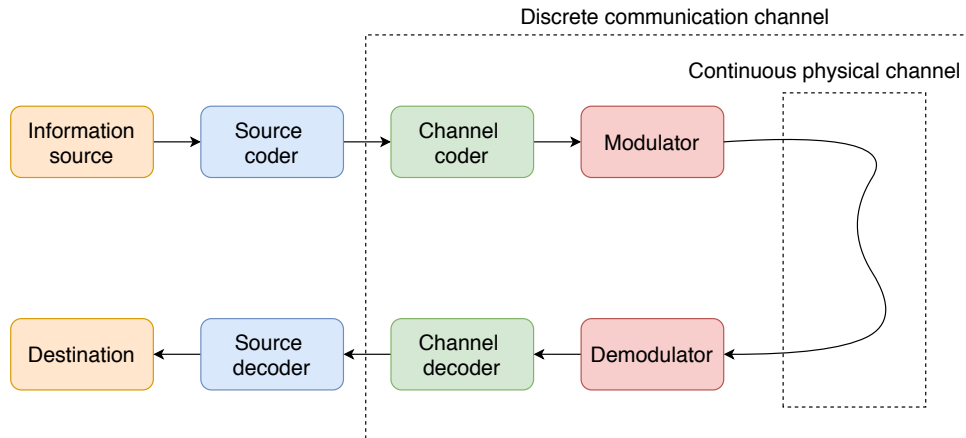


Figure 2.3: Shannon model of the digital communication system [2]

The information source generates the messages and it may be characterized by its alphabet, the symbol rate, and the bit rate.

The source coding is transforming the message to be sent into a representation consisting of symbols suitable for further processing or beneficial for transferring. In digital communications, the alphabet is often binary. We may differentiate between no encoding and the compression, which may be either lossless or lossy. The compression removes the redundant data, and lossy one also removes the irrelevant data.

The channel coding is a means for detecting or correcting erroneous messages. The process consists of inserting extra information into the message that ultimately serves for this purpose. Often used methods for detection are checksums, specifically Cyclic Redundancy Checks (CRCs).

The modulator further transforms the signal for it to be transferable over the continuous physical channel. While in a solid medium over a short distance, no modulation may be necessary, communicating over RF requires a form of modulation. Amongst some widespread modulations for digital data, one may find Amplitude Shift Keying (ASK), Frequency Shift Keying (FSK) or Phase Shift Keying (PSK).

The continuous physical channel is a medium for data transfer. It may be typically over a solid wire, over the RF but also, e.g., acoustic [26] or over the light, e.g., LiFi or Infrared (IrDA).

■ Frequency Shift Keying

A very frequently used method of modulation is FSK. Symbols are encoded by the change of signal frequency. We may consider a central frequency with a positive and negative deviation. This may be easily presented using a constellation diagram – a binary FSK (BSFK) is the simplest example, illustrated in Figure 2.4. In short terms, a bit 0 of a signal is modulated by the lower frequency of the signal and the bit 1 by the higher frequency (in the special case of BFSK, symbols may be equated to bits). FSK is naturally not

limited to binary modulation but may use an arbitrary number of frequencies (limited by transmitter and receiver capabilities).

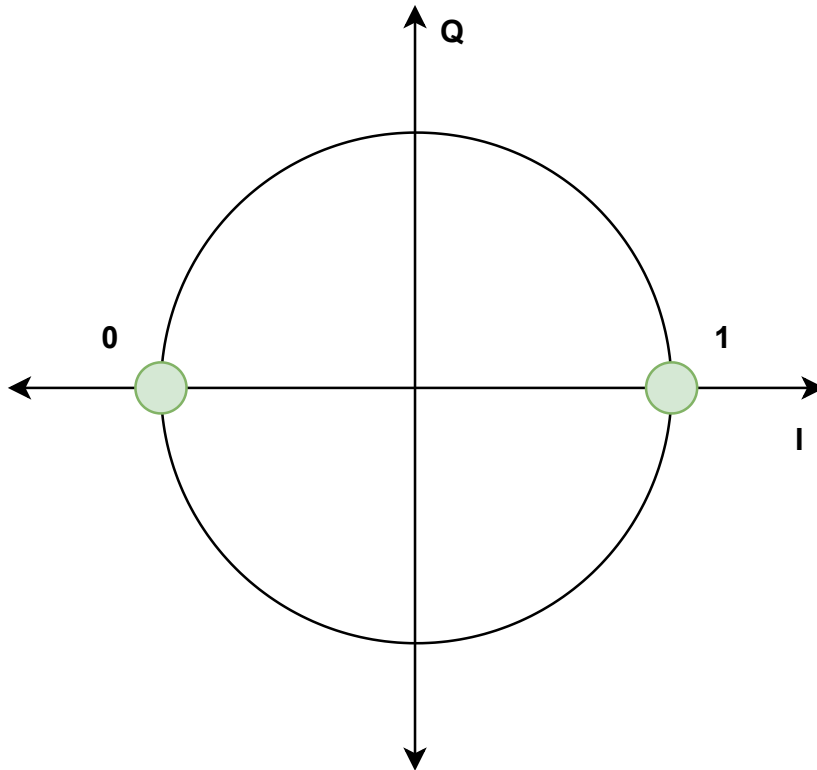


Figure 2.4: BFSK constellation diagram

An example of a typical BFSK signal in time may be seen in Figure 2.5.

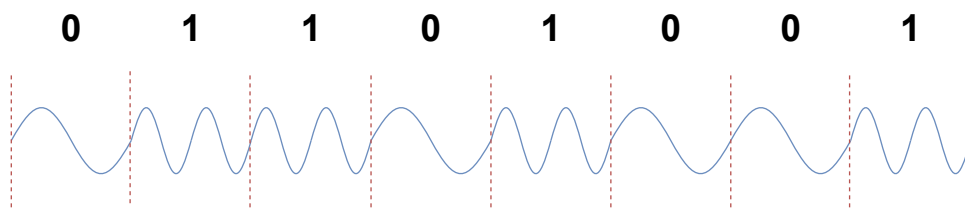


Figure 2.5: BFSK modulated signal example

Its main benefits comprise a relative simplicity and a constant envelope which allows for the use of efficient non-linear power amplifiers (such as class C or E [27, p. 1]).

Nonetheless, the power spectrum efficiency of such modulation is always considered lower than that of the non-constant envelope [28, p. 304].

2.2.2 Software-Defined Radio

For our signal recording needs, we utilize a Software-Defined Radio (SDR). It is a device capable of receiving or transmitting the RF data based on the settings done by software. The SDR, in our case, allows for an easy and fast receiver configuration (such as the centre frequency, the filter width, the sampling rate), low purchase costs, and a direct digital output in the in-phase, quadrature (IQ) data form that is convenient for us for further processing.

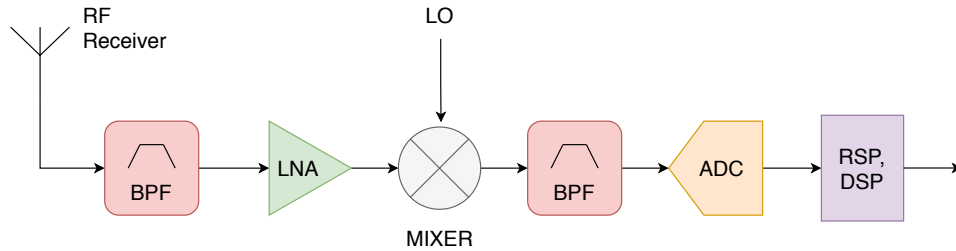


Figure 2.6: Simplified SDR schematic based on [3, p. 7]

The SDR schematic is illustrated in Figure 2.6. The received signal is filtered through a band-pass filter to only include signals from a specific desired bandwidth, amplified through a low-noise amplifier, mixed with the output of the local oscillator, filtered from intermodulation products and ultimately converted to the digital signal. Further operations are done by a receive-signal processor and digital signal processor. The desired output bandwidth is digitized (not only channels) and therefore, the most signal processing can be done digitally, avoiding the need for the complex analogue circuitry [3, p. 7][29].

2.2.3 IQ modulation and demodulation

A very important technique used in SDRs and radio-technologies in general is the IQ modulation and demodulation. The I stands for *in-phase* and the Q for *quadrature*.

In general, a sine wave can be represented by a summation of two components – I and Q. These two components are harmonic waves with a difference in phase of 90° . Interestingly, one may modulate both the amplitude or the phase of the composed signal by changing the amplitude of the I and Q signals. Since these signals are orthogonal, their summation or separation may be performed with relative ease. Using the IQ data is heavily implied by practical hardware designs where a direct phase manipulation may prove difficult [30].

The IQ modulator (also called quadrature modulator), illustrated in Figure 2.7, is a device that performs the described task. It is often utilized for Quadrature Amplitude Modulation (QAM), PSK, or other modulations [31, p. 693]. Both analogue and digital implementations are possible. The input signal is firstly converted using a modulator (in a way depending on the type of modulation) into two signals – the I and Q. These are then mixed with

the output of the Local Oscillator (LO). Precisely, the Q component with the LO output delayed in phase by 90° . The resulting signals are summed and the result is the final output.

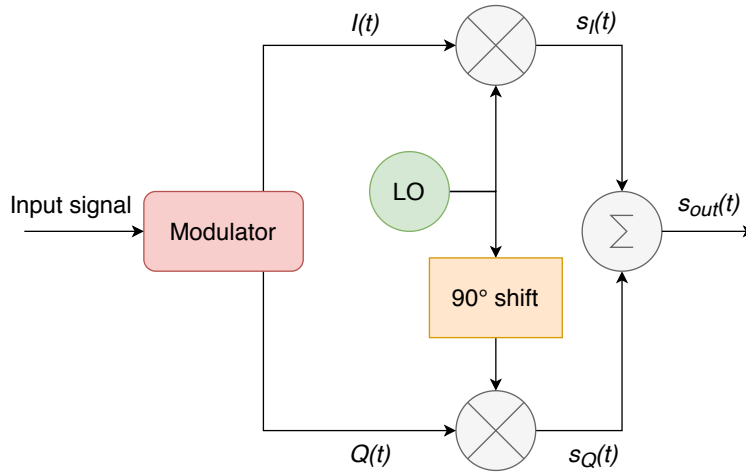


Figure 2.7: IQ modulator

The IQ demodulation is a very similar process described in Figure 2.8. In our simple SDR diagram in Figure 2.6, the quadrature demodulator may be implemented in the Digital Signal Processor (DSP). A necessary feature for the demodulator is the low pass filter that eliminates the intermodulation components produced by non-linearities, i.e., the combinations of the LO's higher harmonics and their differences or sums with the input signal and its harmonics.

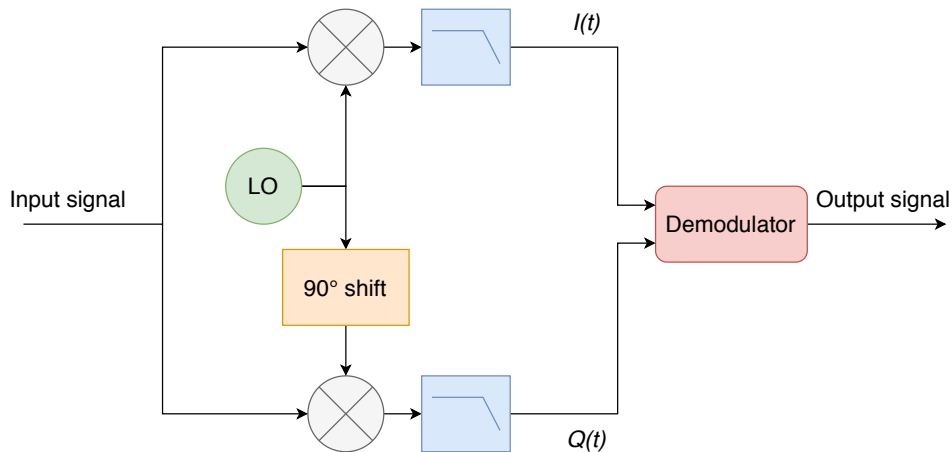


Figure 2.8: IQ demodulator

A mathematical interpretation of the modulation is the following. The LO oscillates at the carrier frequency f_c . Let us consider the desired output signal s_{out} :

$$s_{out} = A \cdot \sin(2\pi f_c t + \phi(t)) \quad (2.1)$$

Using the trigonometric identities we may write:

$$s_{out} = A \cdot \sin(2\pi f_c t) \cos(\phi(t)) + A \cdot \cos(2\pi f_c t) \sin(\phi(t)) \quad (2.2)$$

Accordingly to the Figure 2.7 let us denote the I-component as $I(t)$ and the Q-component as $Q(t)$:

$$I(t) = A \cdot \sin(\phi(t)) \quad (2.3)$$

$$Q(t) = A \cdot \cos(\phi(t)) \quad (2.4)$$

We can finally conclude:

$$s_I(t) = A \cdot \sin(\phi(t)) \cos(2\pi f_c t) \quad (2.5)$$

$$s_Q(t) = A \cdot \cos(\phi(t)) \sin(2\pi f_c t) \quad (2.6)$$

$$s_{out} = s_I(t) + s_Q(t) \quad (2.7)$$

It can be seen that the output signal may be fully modulated based on the $I(t)$ and $Q(t)$ amplitudes, frequency, and phase.

Chapter 3

TPMS protocol analysis

The critical and the first part of this thesis was the analysis of the ŠKODA AUTO's TPMS protocol. We had absolutely no prior information on its functionality since the TPMS design is the intellectual property of a company that manufactures these systems.

All we knew at first was that the communication is wireless. Luckily, not all the information had to be extracted by ourselves because it was expected that many aftermarket manufacturers and amateur hobbyists might have tackled a similar issue already. Therefore research had to be conducted first.

The procedure of the analysis consisted of the following steps:

- recording the IQ radio data using the SDR capture,
- identifying the parameters of TPMS valves transmission,
- interpretation of the data,
 - assigning the correct units, offset and gain of encoded transmitted physical entities,
 - determining the metadata (such as checksum etc.),
- CAN communication analysis.

3.1 SDR data recording

We made records using a relatively cheap, available, and community-supported *SDR RTL2832U*. The captured data were in the form of IQ components, as explained earlier in Subsection 2.2.3.

An often-used software tool to cooperate (not only) with this SDR is the *Gqrx*¹. It allows the user to record the IQ data and set many parameters such as a sampling rate, Automatic Gain Control (AGC) settings, a DC component removal, filter types and widths, Fast Fourier Transform (FFT) settings, and many more. It is also able to demodulate Amplitude Modulation (AM) and Frequency Modulation (FM) radio stations transmissions and directly play

¹<https://gqrx.dk/>

the result as a sound but this is not viable for our case of encoded and modulated digital data. The screenshot of this application is in Figure 3.1.

The first step was to determine the carrier frequency at which the data are transmitted over radio waves. Luckily, this frequency is publicly known as it is also written on the smart valves themselves. This frequency is located inside the Industrial, Scientific and Medical (ISM) band with the centre frequency of 433.92 MHz.

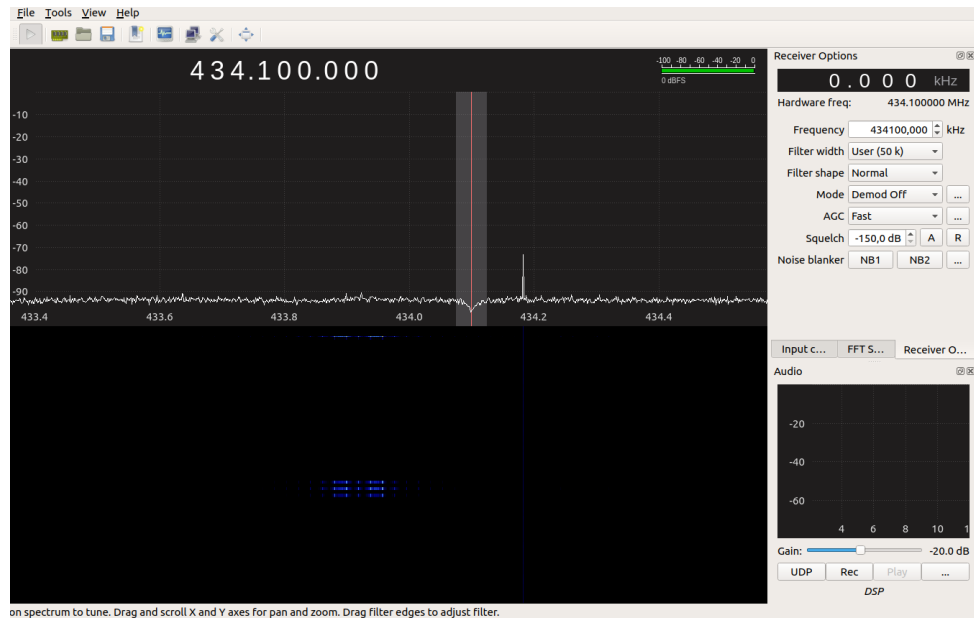


Figure 3.1: *Gqrx* interface

3.1.1 TPMS smart valves' transmission activation

It has to be noted that the TPMS smart valves' transmission seemed to be motion-activated. This triggering is either by a swift movement after a period of motionlessness or during a periodic, constant movement. However, it is possible that the smart valves also transmit sporadically without an external incentive in more significant periods (such as tens of minutes or hours) in order to save battery.

This activation is an easy task when the smart valve itself is located outside the tire and can be achieved by plainly shaking it with one's hands. It is, however, more challenging when the valve is correctly mounted on a wheel which one, therefore, has to accelerate. Although we had both a wheel with an embedded valve and extracted valves, the expected credibility of the received data was higher with the tire, because the valve is located in its desired environment with expected pressure and motion type. That was because of the natural rotational movement and high pressure. The first tries of activation of the valve embedded in the tire consisted of various rotating of the wheel using very rough manual methods such as accelerating by kicking, as seen in Figure 3.2, and spinning on a chair.



Figure 3.2: Attempt at activating the TPMS transmission with an embedded valve

However, a more sophisticated apparatus with a rotatable plate, made by welding, had to be developed, as it would enable much easier manipulation. It can be seen in Figure 3.3.



Figure 3.3: Welded wheel holder with a rotatable plate

3.2 Transmission parameters identification

Initially, we had to obtain some radio wave transmission parameters to be able to capture and decode the data.

3.2.1 Carrier frequency

The carrier frequency is inside the ISM band with a centre frequency of 433.92 MHz. Once we knew the rough carrier frequency, we could record the IQ data using the SDR and use it for further investigations. The IQ data sometimes had to undergo digital filtering first depending on the amount of noise present in captured data, which was very dependent on the location of recording, SDR's Universal Serial Bus (USB) connection quality and other relevant conditions.

3.2.2 Modulation

We determined the type of modulation the following way – we first recorded the IQ data when the TPMS valve was transmitting and then viewed them in *Audacity*², as seen in Figure 3.4. Carefully analyzing the phase, the amplitude, and the frequency of the captured waveforms in time easily yields the used modulation which, as expected, is a rather primitive one given by this simple system (primitive in comparison to the television signals for example).

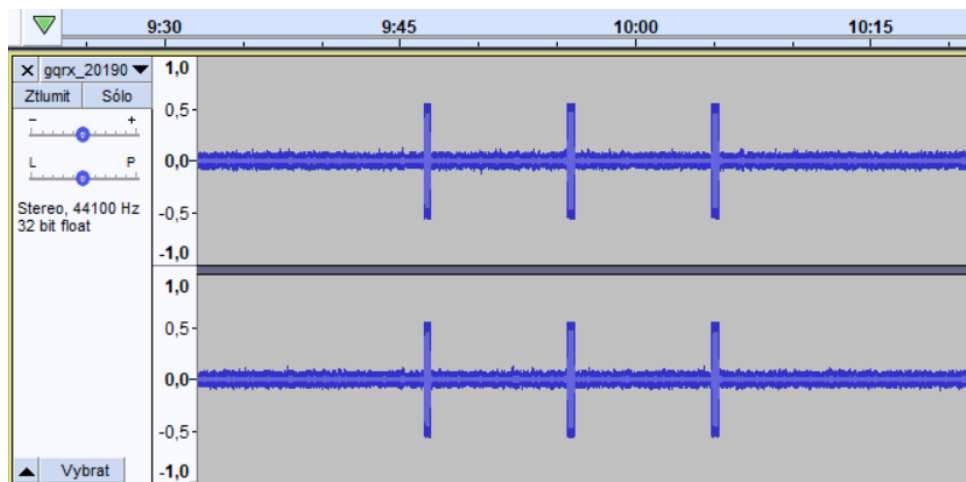


Figure 3.4: IQ data of a located TPMS messages surrounded by noise visualised in *Audacity*

3.2.3 Baud rate

Determining the baud rate (BR) has been done straightforwardly. We simply measured the time duration of a single message (t_m) and the baud count of the message (B_m), then the calculation was trivial:

$$BR = \frac{B_m}{t_m} \quad (3.1)$$

Although the final calculation was taken as an average of more measurements, the exact knowledge of the baud rate was expected not to be necessary.

²<https://www.audacityteam.org>

This is because, in various types of communication, where an externally provided clock is unavailable, synchronization bytes consisting of alternating bits are put at the beginning of the transmitted message [32, p. 732][33, p. 207][34, p. 1-3], thus initially specifying the baud rate. The receiver may adapt its baud rate accordingly to this synchronization, therefore eliminating a possible clocking instability.

3.2.4 Frame encoding

Identifying of the frame encoding (e.g., checksum) was not directly feasible. Therefore, finding the right encoding consisted of three main parts – research, hints given by the communication structure, and trial and error. In general, we started from the most likely options and continued to less likely ones.

However, the only way to indeed verify the correctness of the selected encoding was by a valid data interpretation (in the sense of Section 3.3), which we have ultimately successfully carried out. Furthermore, this created the first degree of freedom in our decoding task. One way would also be to dissect the TPMS hardware which, after studying its internals, also reveals a limited set of possible encodings.

Extracting binary data from messages

A necessary part of our work was to utilize the methods of demodulating the received IQ data and extract the sequences of bits from them, which we could use for further interpretation. The first method we used was a MATLAB script that would load the IQ data and output a sequence of bits based on our algorithm. This approach, however, had some issues. Notably, it was slow and often required manual checks for correct logic level ranges. It was nevertheless sufficient for the first necessary tries.

```

marty@marty: ~/Desktop/rtl_433
File Edit View Search Terminal Help

Consider using "-M newmodel" to transition to new model keys. This will become the default someday.
A table of changes and discussion is at https://github.com/merbanan/rtl_433/pull/986.

Registered 1 out of 123 device decoding protocols [ ]
Test mode active. Reading samples from file: cf32:tlak_1bar.raw

-----
time      : @72.822594s
model    : RDK      count   : 1          num_rows  : 2
rows     :
len      : 23      data    : 269a2c,
len      : 1       data    : 0
codes    : {23}269a2c, {1}0
Modulation: Freq1   : 0.1 MHz   Freq2    : 0.0 MHz
RSSI     : -0.2 dB  SNR     : 1.9 dB   Noise    : -2.2 dB
-----

time      : @437.326996s
model    : RDK      count   : 1          num_rows  : 2
rows     :
len      : 65      data    : 00086125559a26848,
len      : 1       data    : 0
codes    : {65}00086125559a26848, {1}0
Modulation: Freq1   : 0.0 MHz   Freq2    : 0.0 MHz
RSSI     : -0.2 dB  SNR     : 2.3 dB   Noise    : -2.5 dB
marty@marty:~/Desktop/rtl_433$

```

Figure 3.5: Example of *rtl_433* tool decoding raw IQ data into bits (respectively hexadecimal) sequences

In the long run, a search for a more efficient method has been conducted. This revealed a tool called *rtl_433*³, its usage may be seen in Figure 3.5. Although it required non-trivial tampering of its code, inputting some parameters that may vary with each IQ record (that had to be sometimes digitally filtered first), it provided us with a much faster response. This fact eased further interpretation significantly for us. We also used it in the next phases of the project, specifically when verifying the design of our simulator (Section 5.3).

■ 3.2.5 Other parameters

There are, of course, more parameters to consider – such as transmitting power and in dependence on other properties also channel spacing and frequency deviation. These were roughly measured when analyzing the IQ records and were then adjusted during the progress described in Section 5.3.

■ 3.3 Data interpretation

After establishing at least some parameters of the communication channel and converting the IQ data to the sequences of bits, we were free to try to interpret the received data. However, this problem was very complicated, because it introduced many new problems and degrees of freedom, specifically:

- Is the communication encrypted?
- Is the communication bidirectional?
- Is the frame encoding using an error detection algorithm?
 - What data from the transmitted frame is this encoding based on?
 - What method is used? What are its parameters?
 - Where in the message is the result located?
- Where does the data frame start? Is there a start bit?
- Where does the data frame end? Is there an end bit?
- Are the data separated into bytes?

The methodology was to start with what we consider to be most likely. If this approach kept failing, then a more systematic stance would have to be taken. Practically, we started with the following assumptions:

- Communication is not encrypted.
- Communication is not bidirectional.

³https://github.com/merbanan/rtl_433

- A CRC algorithm is implemented.
 - The synchronization bytes will not take part in the calculation, the rest of the preamble however may.
 - The CRC will most likely be a widely used 8-bit or 16-bit polynomial with no initial seed (resp. the value would be 0).
 - The CRC will be naturally appended at the end of the message.
- The data will be segmented into bytes. Bits that do not fit are either start or end bits.

The rationale is the following – the encrypted communication would be extremely challenging to crack. There may be methods, ranging from the brute-force, reading out program assembly code out of the TPMS hardware to various hacking solutions such as injecting custom code. However, without a known algorithm, private encryption key, with locked Microcontroller Unit (MCU), this would be a challenging problem. Therefore, counting on the fact that the encryption and the security had been considered an unnecessary extra step during the TPMS development (costing more effort – i.e., time and money), we supposed that the encryption is simply not used.

The communication is unidirectional because otherwise a more complex circuitry would be required, and seemingly, there is no need for the smart valve also to receive the RF data.

The CRC is a very often utilized and effective algorithm for error detection in communication media [35]. We also suppose it will be a multiple of 8 bits and it will not be too big (not over 16 bits) – out of simplicity and the fact that a transmission of every extra byte is a substantial payload from the view of battery life. In the ultra-low-power systems, the radio transmission may be critical from the energy consumption view considering the electronics are very likely to be in sleep modes most of the time with minimal current consumption.

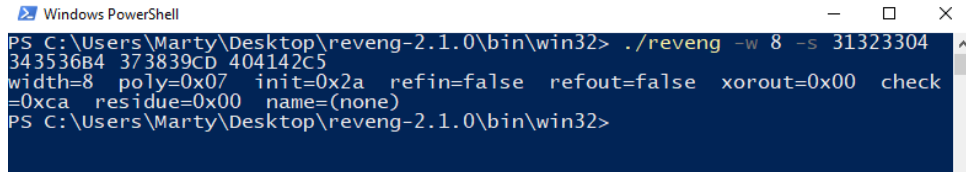
The segmentation of data into bytes is a natural supposition since it eases the interpretation of data in a digital system and also the developer's work. Appending the checksum at the end of the message is a very usual practice, easing the message processing.

■ 3.3.1 Checksum cracking

As mentioned, the supposed method of the checksum was the CRC. It has to be noted that once the CRC and its parameters had been successfully verified to be used in the message, it meant a significant breakthrough for our progress because it confirms many assumptions we had (relevant start of data; end, start and stop bits; CRC location and calculation method).

The first method of cracking the used CRC was a very naive one, half-automated computation of CRC based on various parts of demodulated messages. This method was carried out using a custom MATLAB script with some predefined CRC polynomials that we considered likely to be used. After

trying out many combinations of strings of input data (with varying start and end bits for the CRC calculation) and CRC polynomials with no success, it became apparent that a more systematic approach was necessary.



```

Windows PowerShell
PS C:\Users\Marty\Desktop\reveng-2.1.0\bin\win32> ./reveng -w 8 -s 31323304
343536B4 373839CD 404142C5
width=8 poly=0x07 init=0x2a refin=false refout=false xorout=0x00 check
=0xca residue=0x00 name=(none)
PS C:\Users\Marty\Desktop\reveng-2.1.0\bin\win32>

```

Figure 3.6: *CRC RevEng* tool example – input messages with appended checksum yield the used CRC polynomial along with its properties

At this point, more thorough research on the Internet has been carried out and it revealed some interesting points. Concretely, we have decided to try out a tool called *CRC RevEng*⁴, a command-line application that can reverse-engineer numerous CRC algorithms based on input byte sequences. The more sequences one inputs, the less ambiguity is achieved. The main benefit in contrast to our naive algorithm is the ability to try more CRC polynomials with any initial seeds. Its use may be seen in Figure 3.6.

With various splicing of input byte sequences and more trial and error, we were able to confirm that a specific CRC polynomial is indeed used, however with a non-zero initial seed. This discovery was critical, as it confirmed the correct selection of the channel decoding, message’s start and end, byte segmentation, and importantly – the absence of any encryption seemed very likely at that point.

3.3.2 Identification of a TPMS valve

Since we expected a unidirectional communication, the next natural guess what to look for in the received messages was the smart valve’s ID. We supposed that the TPMS ECU would know the relation of the smart valve to the respective position of the wheel by a unique ID. We had some initial clues – the ID would always be the same number located in any message sent by the same valve. We also expected that the valve’s ID might be printed somewhere on the valve itself.

Both of these suppositions were correct, and the analysis of numerous messages of one valve allowed us to find the ID in the message. This discovery also practically confirmed that indeed **the communication is not encrypted (!)**. This fact may be critical from the safety and the security point of view, where susceptibility to malicious misuse by hackers may arise as mentioned in Section 2.1.1.

3.3.3 Pressure measurements

Naturally, we were confident the messages would include information about the pressure. The methodology was quite simple – obtain the messages from

⁴<http://reveng.sourceforge.net/>

the smart valve with no physical entities changing but the pressure. The first idea was to take the wheel with an embedded valve, change the pressure inside using a compressor, and then force it to send messages. The last step, unfortunately, proved very impractical because of the problematic movement of tire and demanding activation of transmission (as described in 3.1.1). We were, however, able to extract a few messages for further analysis.



Figure 3.7: Pressure measurements in an enclosed container

The second and more efficient idea was to put the pressure sensor of the smart valve into an enclosed container (a glass can) where we could control the pressure using a compressor and activate the transmission in a much easier way, as Figure 3.7 shows. This approach allowed us to reliably find the pressure data inside the messages and calibrate based on the values. Once again, however, a precise unit system was found later, thanks to the CAN analysis (Section 3.4).

■ 3.3.4 Rotational velocity measurements



Figure 3.8: Rotational velocity measurements on a valve attached to a drill

Based on the TPMS hardware (HW) analysis, we believed that the data of rotational velocity might also be measured and transmitted. We expected the possibility of a 2-axis accelerometer or gyroscope. Initial attempts consisted of using a simple cordless drill that would rotate the smart valve, as seen in Figure 3.8. Based on direction and magnitude, the messages would differ. This assumption proved correct.

The second methodology we used was externally attaching the valve to a powered wheel on a car (Figure 3.9). Rotational velocity was indirectly measured using the car's speedometer. This method made the previous results more accurate and credible.



Figure 3.9: Rotational velocity measurements on a valve externally attached to a wheel of a car

The last method we used was directly with a test car in ŠKODA AUTO when we captured data sent by the valve inside a moving tire on a car along with supplemental car velocity data from CAN communication.

■ 3.3.5 Temperature measurements

The next value we were expecting to find inside the messages was the temperature. The methodology was the same as in previous steps, change temperature only. We were able to perform calibration in three points with the valve located in a bowl of water of room temperature, warm temperature, and a refrigerator temperature (around 8 °C). The measurement process can be seen in Figure 3.10.

The exact calibration was not necessary at that moment since we in the first place wanted to confirm in the home environment our idea, that the messages contain the temperature data. The precise measurements were planned for later, but thanks to the CAN analysis (Section 3.4), we were able to extract the unit system precisely with minimal effort.



Figure 3.10: Temperature measurements in warm water in home environment

■ 3.3.6 Miscellaneous data

Although the previously analyzed data was already somewhat sufficient for designing and constructing a simulator, a long-run analysis of the messages also revealed more information transmitted from the valves. Often the frame had some pieces of data missing. Those were given by a number at the beginning designating the type of message.

Data from the same valve over time revealed a slowly decreasing byte we were somewhat anticipating – the internal battery status.

A small fraction of the data has remained undisclosed and mostly the same with any actions we took. This data is interpreted perhaps as various flags, version numbers, or reserved bits for future use.

■ 3.4 CAN analysis

The analysis of CAN communication between the car's ECU and the TPMS ECU has been tackled as a parallel task to the previously mentioned decoding and also to the prototype development described in Section 5.1. This part of

the project made us understand a little bit more about the internal evaluation of the TPMS and allowed us to adjust many transmission parameters of our TPMS Simulator later on.

The idea of the inquiry was the following – take the TPMS ECU, connect to it over a CAN interface device (both shown in Figure 3.11) and try to find out what are its options – such as whether it has diagnostics or what kind of information from the TPMS valves it passes on. We have also had the option to use the commercial *CANoe* software, which was able to interpret many messages when correctly configured.

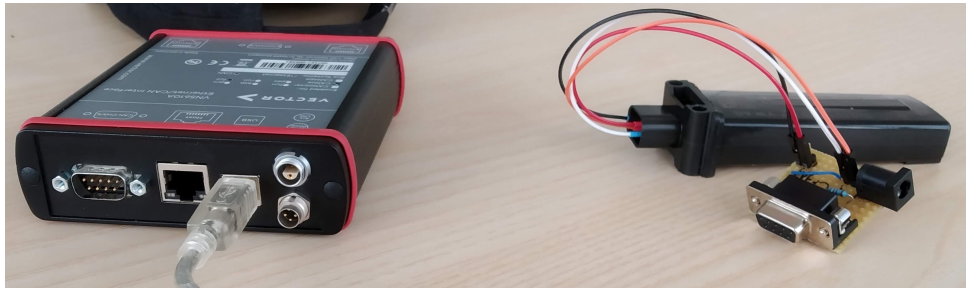


Figure 3.11: TPMS ECU with an adapter attachable to a CAN interface device

As it turned out, a lot of data exchange from the TPMS ECU required some sort of authentication that was unavailable for us. We were, however, able to interpret some data received from the TPMS valves. This was vital knowledge because it served as a verification mechanism for our TPMS Simulator as we could see if our device is successfully transferring the data messages to the TPMS ECU. Furthermore, it allowed us to precisely calibrate some physical entities measured by valves and processed by TPMS ECU as described later on in Section 5.3.

In the end, the CAN analysis served more of a supplemental role in this project from the point of communication decoding but its contributions in debugging the simulator itself were critically helpful.

Chapter 4

Simulator design concept

This chapter describes the concept of the TPMS Simulator design. Generally the term TPMS Simulator used in this thesis refers to both the physical device, called Transmitter unit, and the graphical user interface (GUI). The requirements for the final version were:

- physical portability,
- USB connectivity,
- CAN connectivity,
- USB and 12 V power input.

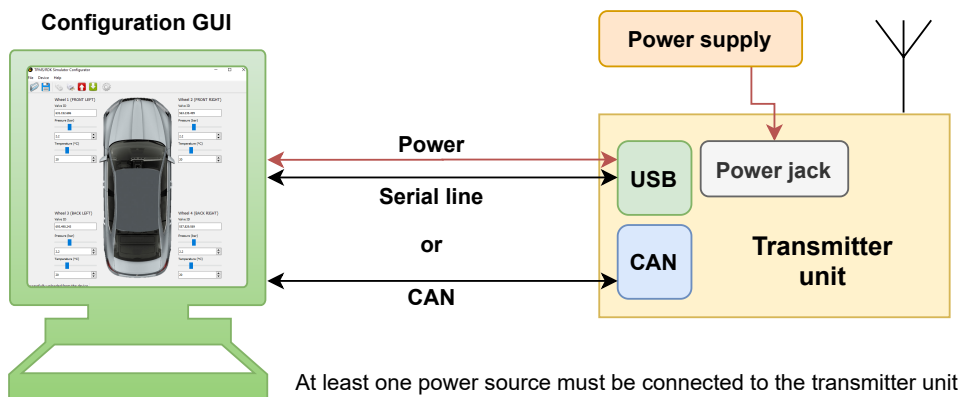


Figure 4.1: *Interactive* mode - connection of the Transmitter unit to a computer with the configuration software over the USB or CAN

Initially, a prototype has been developed (which did not include CAN or 12 V power input) and after that, the first version of the final product – a device suitable for small series production. The simulator in itself consists of the following:

- Transmitter unit,
 - hardware and printed circuit board (PCB) layout,

- mechanical realization,
- firmware,
- configuration GUI.

The physical device can send the TPMS messages to the TPMS ECU and force it into interpreting them as messages sent by the actual TPMS smart valves located in the car's wheels.

The Transmitter unit may work in two modes – *interactive* and *standalone*. In *interactive* mode, illustrated in Figure 4.1, the Transmitter unit is connected to the computer over USB or CAN, where it can also be configured with the respective software. In this mode, the device is both transmitting and open for any commands coming from the software on PC.

Alternatively in *standalone* mode, illustrated in Figure 4.2, it may be connected directly to a source of electrical power (such as a power supply or external power bank), thus running with the last saved configuration.

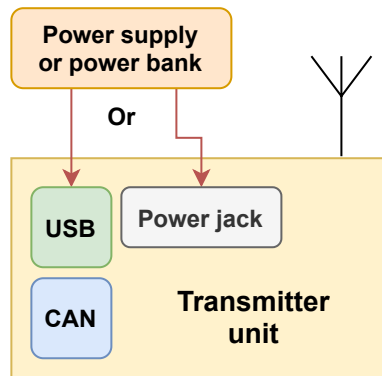


Figure 4.2: *Standalone* mode - connection of the Transmitter unit to a source of electrical power

The prototype version has been based on an STM32F401RE NUCLEO development kit with limited connectivity and power options (USB only).

The final version, while principally fulfilling the same purpose, is a custom PCB driven by a standalone STM32F446RC MCU with more options for both the connectivity and the power input (USB, CAN, and a power jack input).

4.1 Hardware

Initially, research of available SDRs with the capability of transmission has been conducted, leading to several results. It was deduced that they are usually rather complex structures with a relatively high price but a great extent of capabilities. We have therefore concluded to try a more light-weight solution first – a radiofrequency integrated circuit (RF IC) driven by an MCU.

A block diagram of the final version of the Transmitter unit can be seen in Figure 4.3, with power lines represented by red colour. Both the schematics

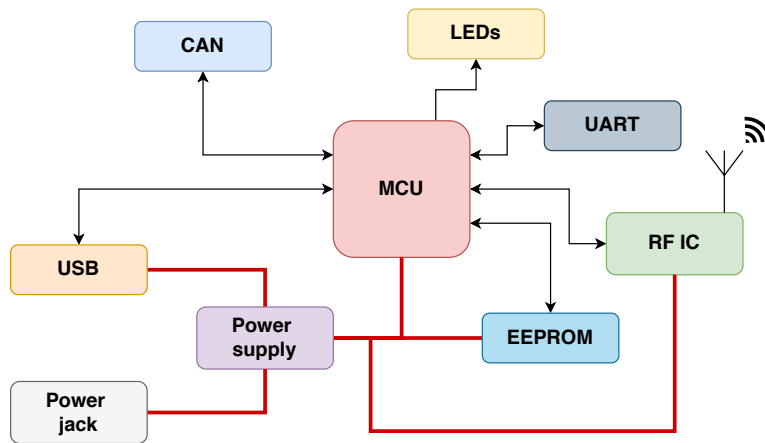


Figure 4.3: Block diagram of the Transmitter unit

and PCB layout were drawn using the KiCAD Electronic Design Automation (EDA)¹.

4.1.1 Power supply

While the prototype of the Transmitter unit utilized a development kit that included linear regulators and a USB power input only, the final device had broader requirements as mentioned at the start of this chapter. It was necessary to build a custom power supply circuitry that would allow the power input both from the USB and a power jack with the nominal voltage of 12 V.

Power input

As mentioned, the Transmitter unit may be powered either from the power jack or directly from the USB, which enables it to be a portable device in combination with a standard USB power bank.

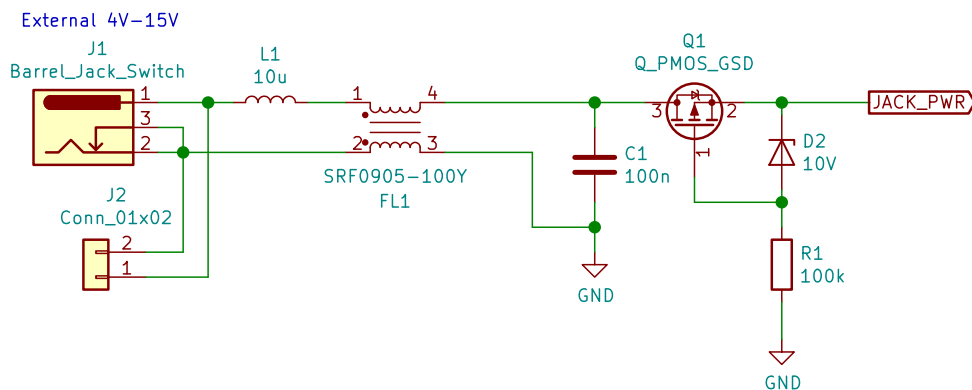


Figure 4.4: Power jack circuitry

¹<https://kicad-pcb.org/>

As seen in Figure 4.4, the power jack input is expected to come from a standard 12 V DC supply connected to the 230 V power line. A simple LC along with a common-mode Electromagnetic Interference (EMI) filtration is therefore added. There is also reverse-voltage input protection using a power MOSFET that switches into a fully closed state when the voltage of a correct polarity is applied and stays open in the reverse case. This solution is greatly more power-saving than compared to using a single diode in series. While the nominal voltage is 12 V, our device is also capable of being powered by a voltage from the interval of 4 V to 15 V.

For development purposes, it is also possible to connect the voltage directly to the pins of the J2 jumper.

The USB power input voltage is usually rather stable and is therefore filtered using a simple LC element (L2, C2) as seen in Figure 4.5. The filtered input voltage from either USB or power-jack is then mixed in a straightforward way using the low-voltage drop Schottky diodes where the power loss is not critical due to the used low voltages. There exist more smart ways to choose between the primary and the secondary power source, but the respective circuitry is relatively complex, and the available ICs were evaluated as way too expensive to weigh out the downsides. A discrete solution is also demanding because back powering caused by imperfect timing of opening and closing of the MOSFET switches is very dangerous and in USB standard strictly prohibited [36, p. 113][37, p. 171].

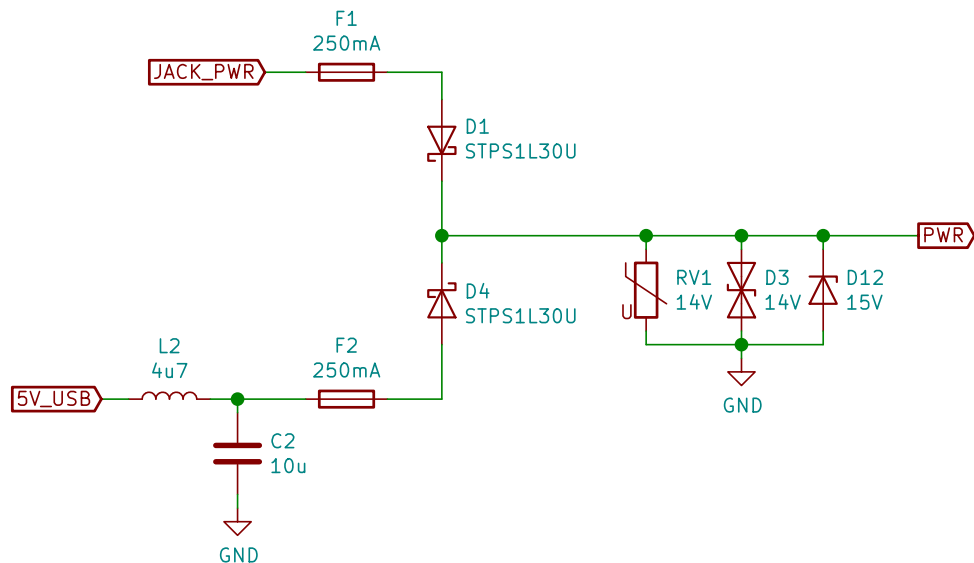


Figure 4.5: Power supply mixing and the overvoltage/overcurrent protection

A combination of fuses, together with elements that short on specified voltage, forms a protective circuit both against overcurrent and overvoltage. Transient Voltage Suppression (TVS) diodes tend to have a fast response but are less capable in terms of heat absorption in contrast to the used varistor. Zener diode is an added safety element that should respond to a smaller overvoltage caused mainly by the user supplying the device with the

wrong voltage as it slowly starts to conduct above 15 V and draw current. The output is then directed to the DC/DC converter, covered in the next subsection.

■ DC/DC converter

To avoid more power losses, we have decided to use a DC/DC converter instead of a linear regulator, this is shown in the schematic in Figure 4.6. While this generally means more components, more occupied space on PCB, and higher costs, it also means greater efficiency with correctly selected components and designed PCB layout.

The circuitry is connected mostly according to the respective datasheet [38] with some extra filtration on the input using a ferrite bead with an added electrolytic capacitor for stability purposes.

The output voltage of this exact series of the DC/DC converter is 3.3 V, a level necessary for all the used IC, also used as a high logic level. The output voltage is not adjustable. This is on purpose since there is no need for it and also the number of components would have to increase. The converter's current capabilities are sufficient for our low-power application (up to 500 mA [38]), the Transmitter unit's power consumption measurements are described in Subsection 5.2.1.

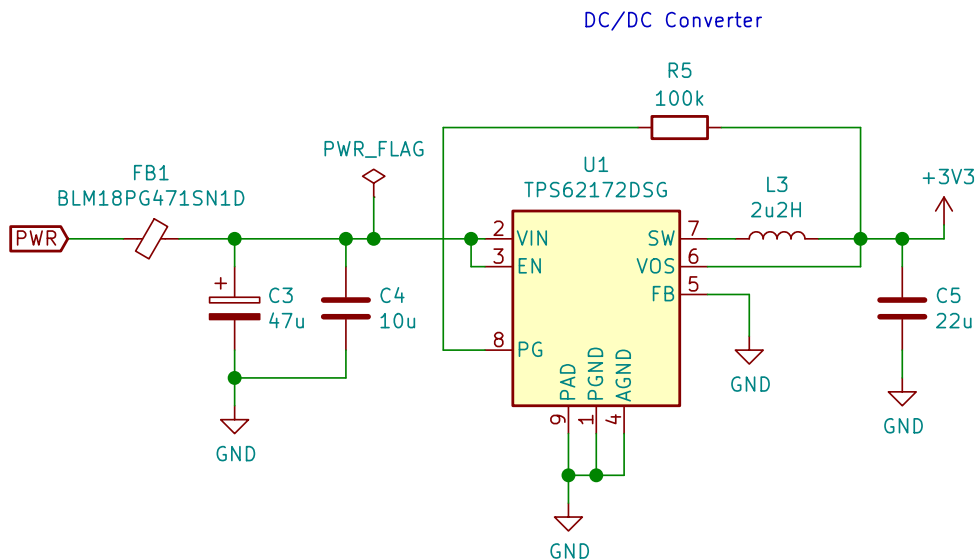


Figure 4.6: DC/DC converter circuitry

■ 4.1.2 Microcontroller unit

The MCU, specifically an STM32F446RC, is a central element that drives the RF IC and the other parts of the hardware. For the prototype, we utilized a NUCLEO development kit (with a similar but different STM32F401RE). The NUCLEO incorporates amongst other features, an embedded programmer, linear regulators, and pin headers connected to MCU's output pins.

The manufacturer, *STMicroelectronics* (often abbreviated as *ST*), describes it the following way: “The STM32F446xC/E devices are based on the high-performance Arm® Cortex®-M4 32-bit RISC core operating at a frequency of up to 180 MHz. The Cortex-M4 core features a floating point unit (FPU) single precision supporting all Arm® single-precision data-processing instructions and data types. It also implements a full set of DSP instructions and a memory protection unit (MPU) that enhances application security.” [39, p. 10]. RISC stands for Reduced Instruction Set Computer.

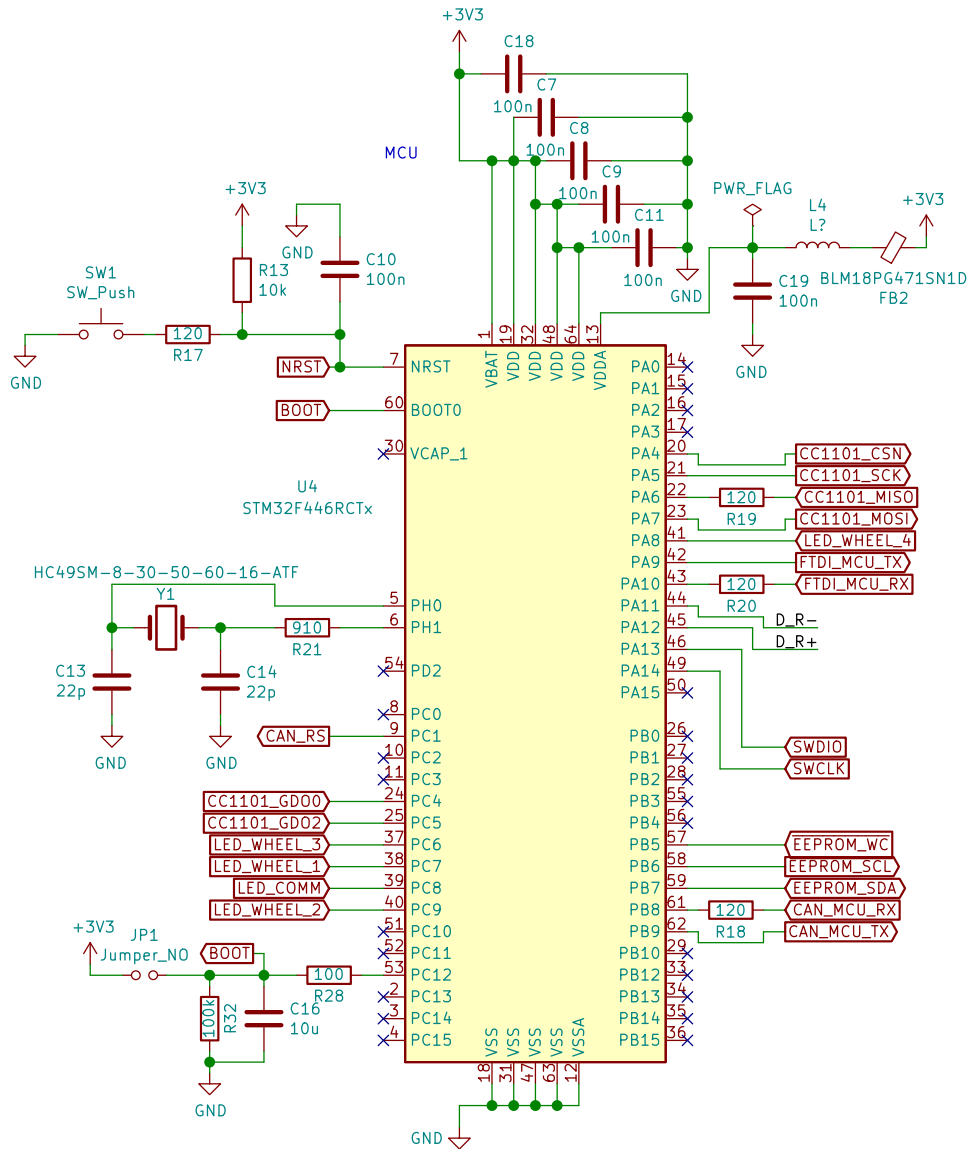


Figure 4.7: MCU circuitry

For our use, some of its peripherals are absolutely vital, namely:

- Serial Peripheral Interface (SPI),
- Inter-Integrated Circuit (I²C),

- Universal Asynchronous Receiver and Transmitter (UART),
- USB interface,
- CAN controller.

The MCU runs the firmware described in Section 4.4 and its circuitry is in Figure 4.7. It lights up the informative Light Emitting Diodes (LEDs), drives the RF IC, controls the external volatile memory (EEPROM), manages the USB connection with the computer, communication with a CAN transceiver and with the debugging FTDI UART-Communication (COM) Port module.

There is a possibility of resetting the MCU using an external button and also a circuitry that enables it to self-enter a Device Firmware Upgrade (DFU) mode on boot. The principle is trivial, the capacitor C16 is firstly charged using the General Purpose Input Output (GPIO) pin PC12 in output mode, then the MCU resets itself. Upon booting, it senses a high logic level on the BOOT pin and therefore enters the DFU mode, where one may update its firmware over the USB. There is also an external jumper prepared for DFU testing purposes.

In the end, however, a better solution was found, and that is to perform the jump to the system memory using the firmware (FW) only. This actions consists of internal de-initialization of clock-related modules, memory remapping, setting the stack pointer and jumping to memory. This approach implies that the R28, R32 and C16 components do not need to be assembled. Since there is enough space on the PCB, however, the footprints for the mentioned elements are left in the layout for possible future use (if the former approach would, e.g., prove to be problematic or troublesome for debugging).

The use of external crystal is necessary due to the precision required by USB standards. Resistors used in series on data lines prevent possible signal reflections. Standard blocking capacitors of value 100 nF are used near the MCU's power supply pins, and analogue voltage is connected to 3.3 V through inductor and ferrite bead filters. Their exact value is not critical, inductor L4 (of nH to μ H order) may even be omitted (shorted).

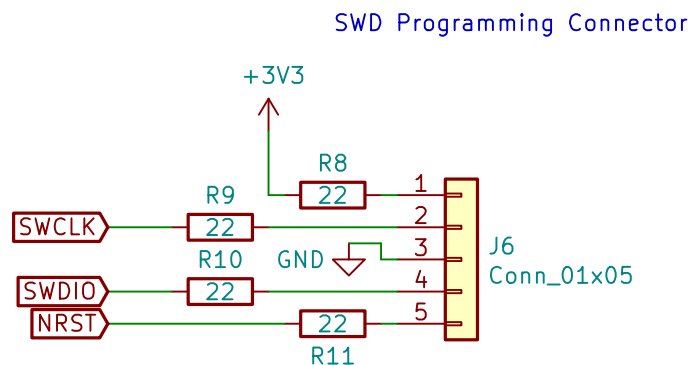


Figure 4.8: Pin header connector for Serial Wire Debug (SWD) programming

Programming of *ST*'s MCUs is possible via the SWD, as seen in Figure 4.8. It allows both for programming and debugging. A simple pin header

connector may be utilized by a respective programmer such as *ST-LINK*.

4.1.3 Radiofrequency integrated circuit

The primary rationale for choosing an RF IC over a transmission-capable SDR is significantly reducing costs, development time and the final device size, since the RF IC can be easily integrated into an embedded solution with a low-cost MCU. The cost-cutting is naturally even more prominent with the small series production.

The criteria were straightforward. The IC would have to transmit around the ISM band with the centre frequency of 433.92 MHz and be as configurable as possible – i.e., modulation, power, baud rate, centre frequency and deviation, channel spacing, ideally also channel encoding and adjustable preamble.

The first pick was a small module with a chip called *CC1101*, manufactured by *Texas Instruments Inc.*, that seemed to fulfil our needs of replicating the TPMS smart valves transmission. Quoting its datasheet, “CC1101 is a low-cost sub-1 GHz transceiver designed for very low-power wireless applications. The circuit is mainly intended for the ISM (Industrial, Scientific and Medical) and SRD (Short Range Device) frequency bands at 315, 433, 868, and 915 MHz ...” [40, p. 1].

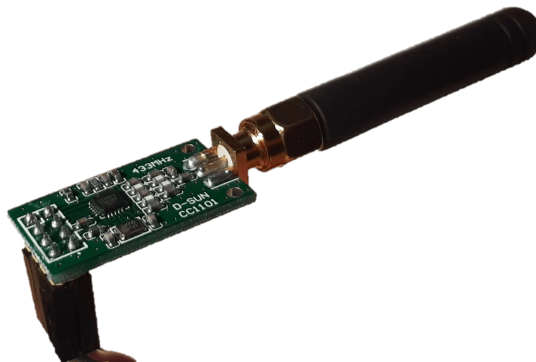


Figure 4.9: CC1101 RF IC pluggable module

The particular module we have selected was out-of-box ready for connection to an MCU over the SPI bus and with the connected circuitry optimized for the 433 MHz operation. Its photo is in Figure 4.9. Apart from the power supply, it also includes some GPIO pins that may serve various functions, such as a notification of a sent packet.

The main advantage of using a female pin socket connector to plug in the CC1101 module, seen in Figure 4.10, is the fact that any other module may be used in the future regardless of the RF IC chip choice. Thus, no change in the PCB layout is necessary. As long as the connected module uses SPI and possibly some GPIO pins, this connector serves universally. Only firmware changes would have to be done.

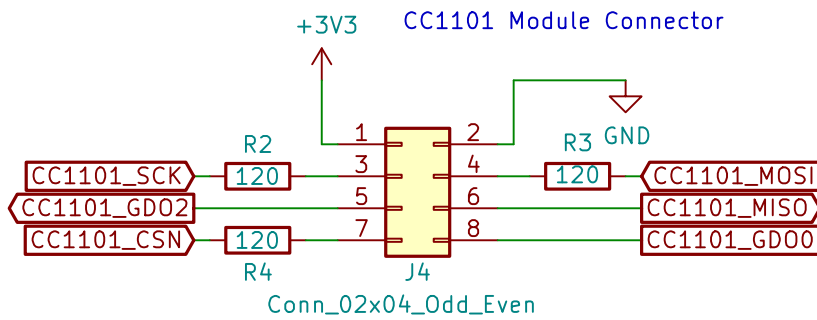


Figure 4.10: Connector used for CC1101 module

4.1.4 EEPROM

An external memory is used in our design. This is because the MCU has no internal Electrically Erasable Programmable Read-Only Memory (EEPROM), only Flash. While the Flash may be sufficient for an extensive period, e.g., when using the EEPROM emulation techniques [41], such as in the prototype, it is still worse than the lifespan of an EEPROM. Given its low cost, we have decided to implement it into our design.

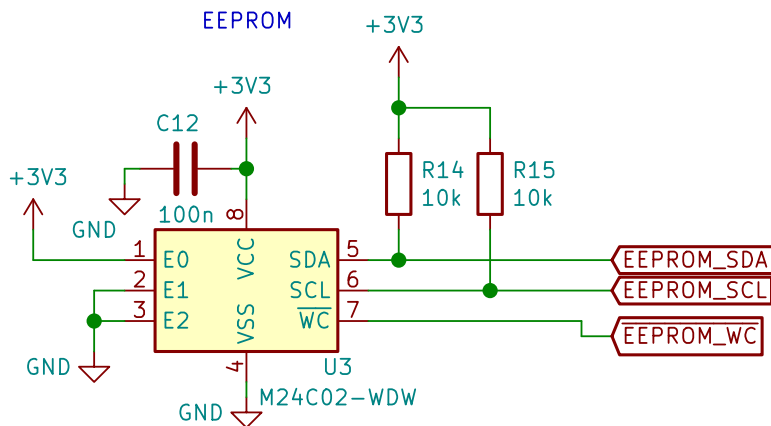


Figure 4.11: EEPROM circuitry

As seen in Figure 4.11, the EEPROM uses a connection over I²C. Part of its address is set using the E0, E1, and E2 pins. There are also the pull-up resistors for I²C lines and a blocking capacitor for the power supply pin.

4.1.5 LEDs

The connection of LEDs is trivial, although they are not connected to the GPIO pins but rather timers' Pulse Width Modulation (PWM) outputs, practically meaning that the MCU can directly drive the amount of the emitted light by setting a variable duty cycle of PWM. There are six LEDs in total as shown in Figure 4.12.

- One lights up when the DC/DC converter provides the voltage of 3.3 V on its output.

- The second LED signals active communication between the Transmitter unit and other devices connected over USB or CAN.
- The remaining four LEDs signify by blinking a currently sent message from the respective virtual wheel of the Transmitter unit.

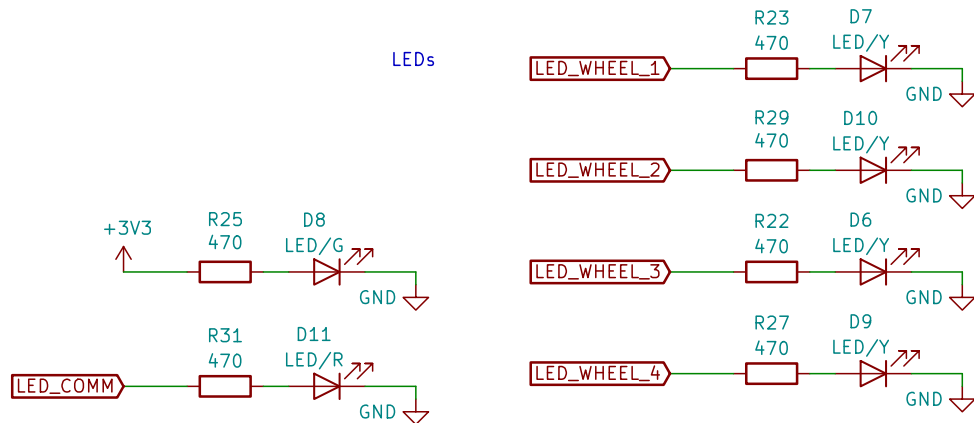


Figure 4.12: LEDs circuitry

4.1.6 USB

USB may be used to connect to the Transmitter unit and configure it. Its respective schematics are in Figure 4.13. The used USB connector is type B. The shield is connected to the ground using a parallel RC element (R12 and C6), as suggested by *Cypress* [42].

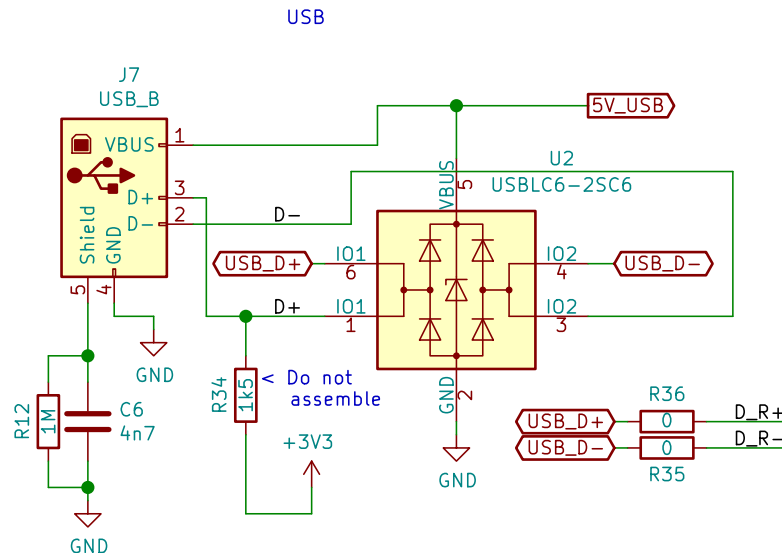


Figure 4.13: USB circuitry

While the datasheet of the used MCU does not mention the pull-up resistor on D+ line and explicitly says that the impedance matching is embedded

and not necessary to solve externally [39, p. 137], the decision was to prepare footprints for elements that would provide this functionality externally – R35, R36 may be assembled as impedance-matching resistors and R34 may be a pull-up on D+ line. This feature might be needed if another MCU with a similar pinout is chosen in the future or if the written USB specifications are not satisfyingly met. However, for the current configuration the R34 resistor should not be assembled and R35, R36 should be shorted (assembled with shorting chips).

U2, a protective IC against various transients, overvoltage, or undervoltage effects is also used. The IC consists of fast-acting TVS diodes.

4.1.7 CAN

The CAN is an alternative to connecting over the USB for configuration since it may be more frequent for usage in ŠKODA AUTO or similar automotive environment.

As seen in Figure 4.14, a CAN transceiver is used in combination with a standard DB9 connector. The transceiver communicates with the MCU using the MCU's internal CAN controller. As seen in Figure 4.14, the resistor R30 on Rs pin selects the mode of CAN between a high-speed mode and a slope control mode. A blocking capacitor C15 for the power supply is used.

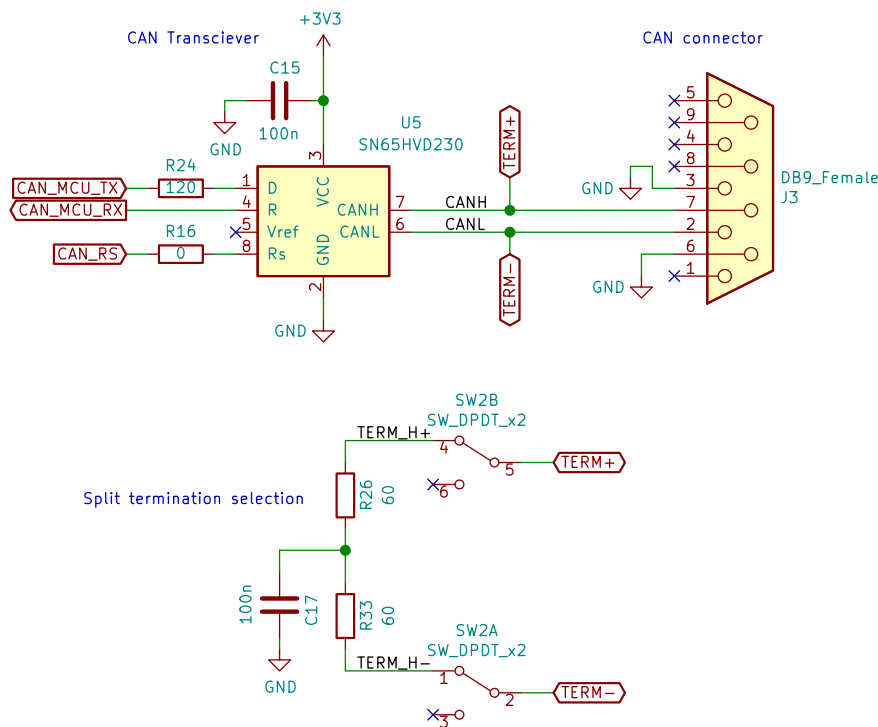


Figure 4.14: CAN circuitry

Termination of the CAN bus is manually selectable with the use of a switch. Specifically, a method of split termination is used that although requiring more components than a standard single resistor of 120 Ω value,

also beneficially serves as a low-pass filter [43] decreasing the common-mode noise.

4.1.8 FTDI converter

For debugging purposes, a pin header (in Figure 4.15) for the UART-USB COM Port converter is used, specifically the FTDI converter TTL-232R-3V3. The MCU may, for instance, easily transmit various information about the current state of the program or receive commands.

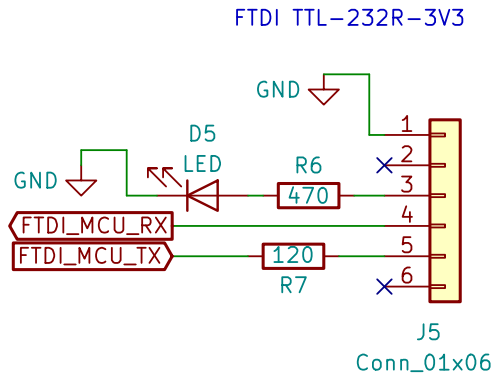


Figure 4.15: FTDI converter connector

The reason for this is that the hardware debugging influences the internal timing of the MCU and we may want to avoid issues caused by this phenomenon and instead execute the code without any debugging interrupts (perhaps even in a release build with optimizations taking effect). E.g., in real-time systems the debugging may cause a disruptive behaviour.

4.2 PCB design

The final version of the Transmitter unit was realized on a custom PCB. As there is no noise-sensitive analogue circuitry, it was sufficient to choose a two-layer board. As Záhlava and Montrose suggest [44, p. 56][45, p. 41], the top layer is filled with the ground plane and the bottom layer with the voltage plane. Also, both layers are used for conducting the signals with the top layer being the primary. The overall design can be seen in Figure 4.16.

The selected dimensions of the PCB are 100 x 120 mm, in order to fit the selected PCB enclosure as mentioned further in Section 4.3. While the size of the PCB is excessive to the number of used components, there were pragmatic reasons for selecting such size. One side of the PCB is joined to the “connectivity panel”, where all the connectors are located. The other side is the “functional panel”, where the antenna and informative LEDs are. More rationale can be found in Section 4.3.

The PCB was designed with the intention of a small series of manual production, meaning that while there was a motivation for cost-cutting, it was sidelined due to the primary goal being an easy assembly by hand. Thus

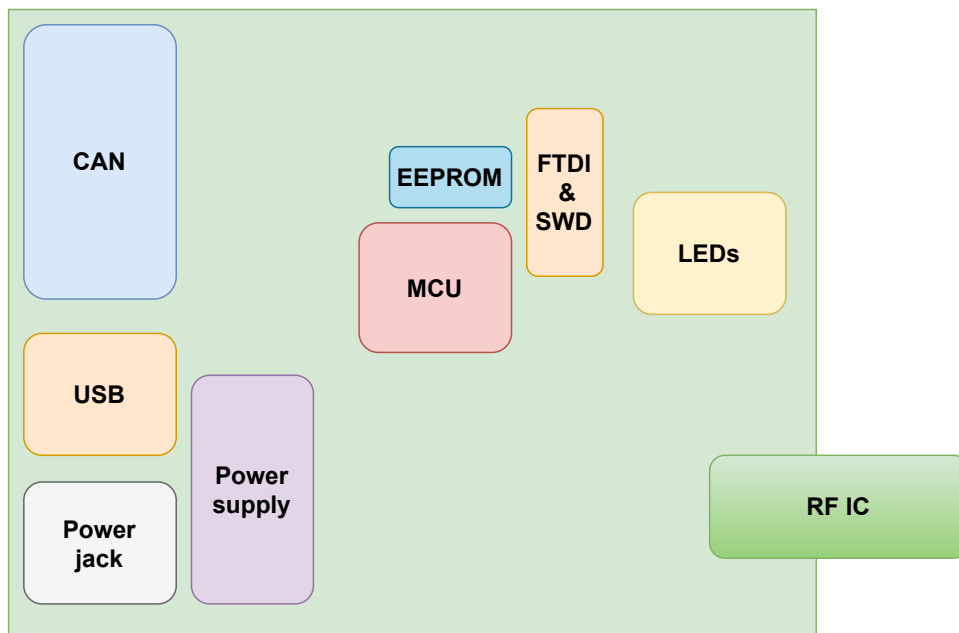


Figure 4.16: PCB block scheme

at the cost of the PCB dimensions, no internal cables are necessary which should make the assembly both more effective and less error-prone. The vastly preferred choice of technology was the Surface Mount Technology (SMT) – usual chip types of passive components were 0805 and 0603. While these are somewhat outdated due to downsizing [46, 47], they are still vastly supported and chosen thanks to the much easier hand assembly.

Some general suggestions are followed, such as minimizing the lengths of traces, keeping the blocking capacitors as close to the power pins as possible, reduction of current loops. Power paths are realized as wide as possible, ideally, in planes to reach a low impedance [44, p. 12], signal traces may be thinner, ground via stitching is performed to reduce EMI [44, p. 56].

The DC/DC converter circuitry is designed as per the datasheet, minimizing loop tracks and partially separating analogue ground that is ultimately connected to the power ground at a specific place [38, p. 25]. Tracks with differential signals are also matched in their length by forming various trace meanders.

The overall design is relatively compact also for possible future extensions. The connectors naturally take the most space. The LEDs are Surface Mount Devices (SMDs), but they are mechanically prepared for the placement of light guide pipes, making the design more robust. The only used Through Hole Technology (THT) components are some connectors and pin headers or sockets due to the reason of the better mechanical endurance (when plugging and unplugging cables, pushing or pulling).

The visualisation of the PCB can be seen in Figure 4.17 and the complete PCB layouts may be found in appendix C.

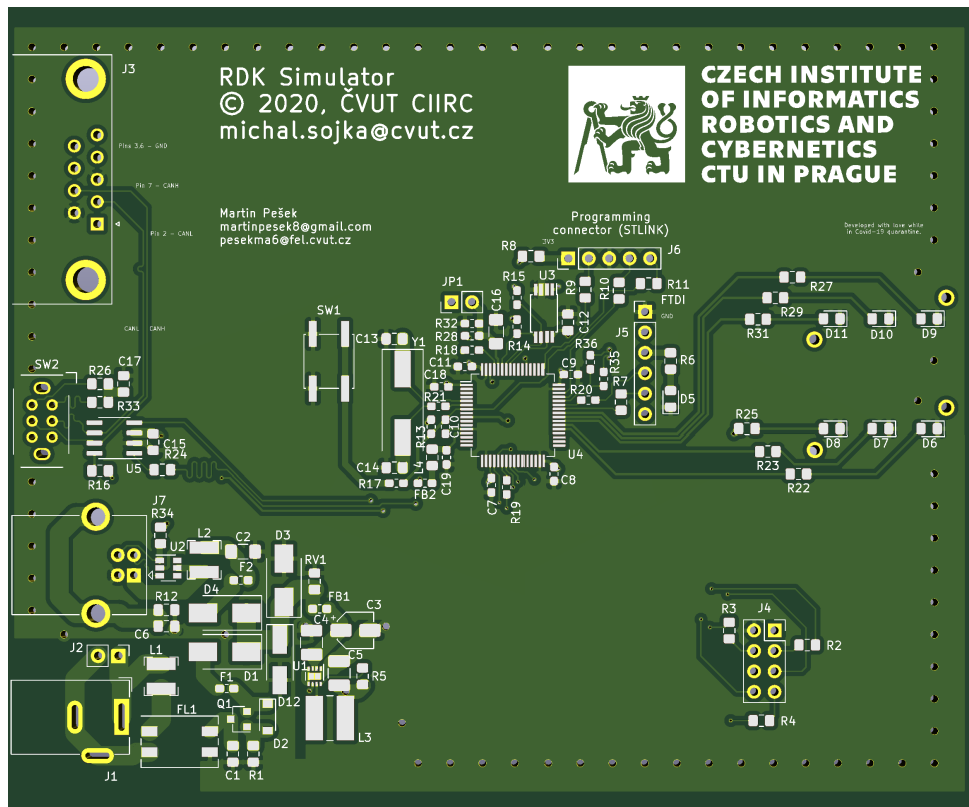


Figure 4.17: PCB visualisation

4.3 Mechanics

A PCB enclosure was required for the Transmitter unit and we have decided to choose *Hammond's* aluminium ones. For the prototype, the *1455K1201* (43 x 78 x 120 mm) was selected and for the final version a similar but a slightly bigger *1455N1201* (53 x 103 x 120 mm), illustrated in Figure 4.18.

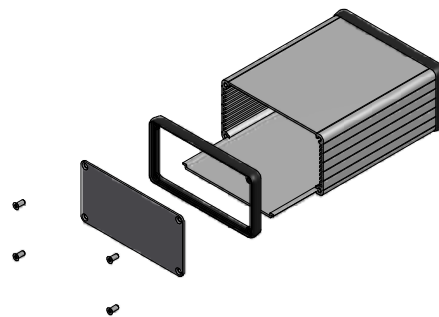


Figure 4.18: *Hammond* 1455N1201 enclosure [4]

To fit the connectors through the plates, milling was required. We have also created a printable graphics design for the front (functional) and the

back (connectivity) panels. On the front panel (Figure 4.19), there are six LEDs, one for the device being powered up, one for active CAN or USB communication, and four blinking ones – symbolizing a message sent from the respective wheel of the virtual car illustrated between the four LEDs. Holes for antenna and screws are included, along with the Czech version of the logo of CTU. The back panel (Figure 4.20) provides USB, CAN, and power supply connection along with a switch for toggling the CAN termination. The back panel (Figure 4.20) provides USB, CAN, and power supply connection along with a switch for toggling the CAN termination.

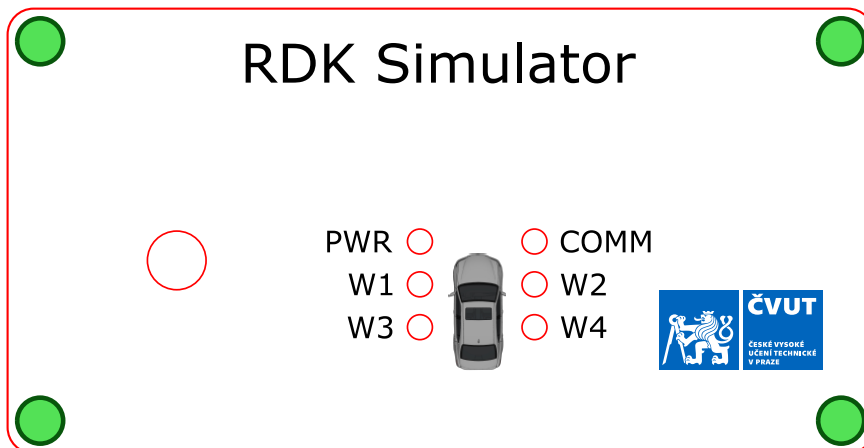


Figure 4.19: Front panel graphics and milling marks

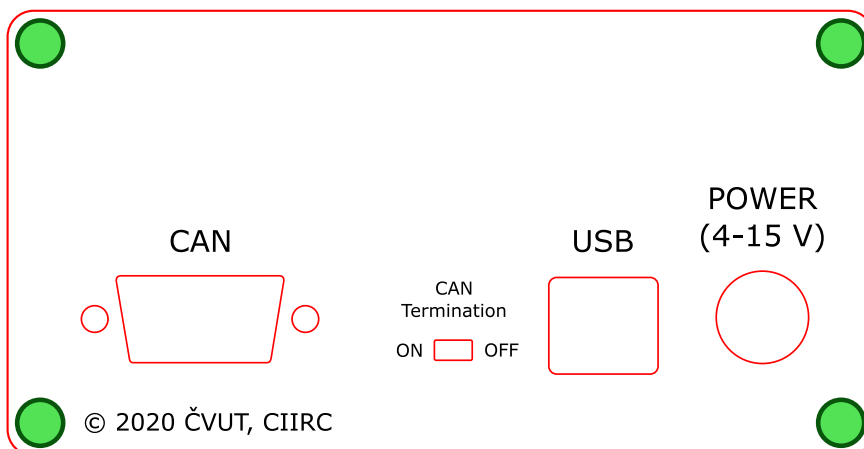


Figure 4.20: Back panel graphics and milling marks

4.4 Firmware

The Transmitter unit is driven by an MCU that naturally requires a firmware containing the machine instructions to be executed. Typical programming language to use for such application is C, but we have decided to implement some convenient object-oriented features as well. Thus we have also used C++.

The firmware has to take care of controlling the RF IC, lighting up the LEDs, saving to and loading data from the EEPROM over I²C, and providing communication interfaces for USB, CAN and UART. The firmware also enables the utilization of the DFU feature over USB.

The entire code is thoroughly described using Doxygen, and therefore the extensive documentation may be found in the same location as the source code.

4.4.1 Structure

The basic idea for this firmware was not complicated. Only a relatively few peripherals needs to be controlled, no high-speed circuitry is necessary, and the system is not safety-critical nor hard real-time. Also, there are not many events that disrupt the usual flow of the program. Only a few interrupts may be triggered, pertaining mainly to the data transmission and the Direct Memory Access (DMA). Therefore, we have decided not to implement any operating system (OS) as a simple main loop calling sub-tasks would suffice.

The code includes *Hardware Abstraction Layer (HAL)* libraries² provided by *STMicroelectronics*. Although they occupy more memory than *Low Level (LL)* libraries (accessible through the same URL as *HAL* libraries), which are of course still more expensive than accessing the registers directly, they provide a very comprehensive and friendly interface for developers.

While the prototype worked with the *Mbed* libraries³ and was compiled using the *IAR EWARM*⁴ compiler, the final version was ported to the *HAL* libraries and is fully compilable with the *GNU Arm Toolchain*⁵. The reasons for this were mainly the more advanced hardware control using the *HAL* and the freely licensed compilation. The complete rationale may be found in the appendix D.

4.4.2 Initialization

At the power-up, the MCU first checks to see the voltage level on the BOOT0 pin. Then it decides from what memory to execute the code. If the voltage level is low, the instructions from the Flash memory are executed, otherwise the bootloader code from the System memory provided by *STMicroelectronics*. Thanks to this, it is possible to use the DFU feature over such peripherals as USB, UART or CAN using special software.

The code from the Flash memory is the firmware created by us. At first the startup assembly code routines are executed, such as setting vector tables, configuring interrupts and setting the start of the program. Then the *main()* function is called and it starts by the initialization of MCU's internals, i.e., as in Figure 4.21, *HAL* libraries, clocks, peripherals and control objects, structures and variables initialization.

²<https://github.com/STMicroelectronics/STM32CubeF4/>

³<https://os.mbed.com/>

⁴<https://www.iar.com/iar-embedded-workbench>

⁵<https://developer.arm.com/tools-and-software/open-source-software/>

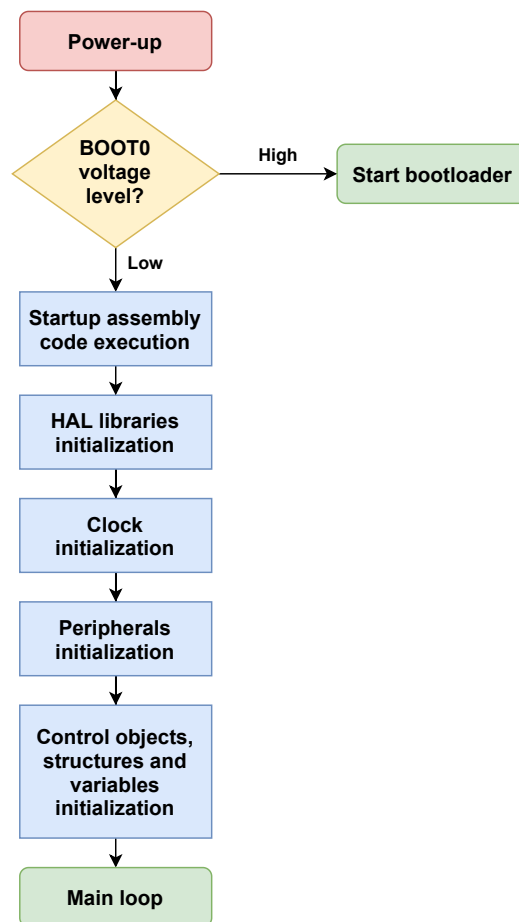


Figure 4.21: FW start-up diagram

4.4.3 Main loop

The main loop may be divided into a few, further decomposable, routines, as seen in Figure 4.22.

The communication routine checks for any available messages inside the buffers of various sources (USB, CAN, UART) and if any data is available, it further processes them, such as parsing the line of data or checking for its validity (by checking the CRC result and the syntax correctness). If all received data are valid, then the appropriate actions are taken, such as configuring the virtual wheels or responding to queries. If any changes to the configuration had to be made, they are saved into the EEPROM.

The LEDs routine simply checks and enables or disables the timeout-based LEDs. This means that whenever a flag is set for a LED to turn on, a countdown is started. It is then checked and acted upon with every run of this routine.

The RF transmission routine directly manages the communication with the RF IC and the transmission of the appropriate messages with the respective timing.

At the end of the main loop, various internal receive buffers are checked to avoid overflows and respective reset routines are carried out if needed and if the respective communication channel is enabled.

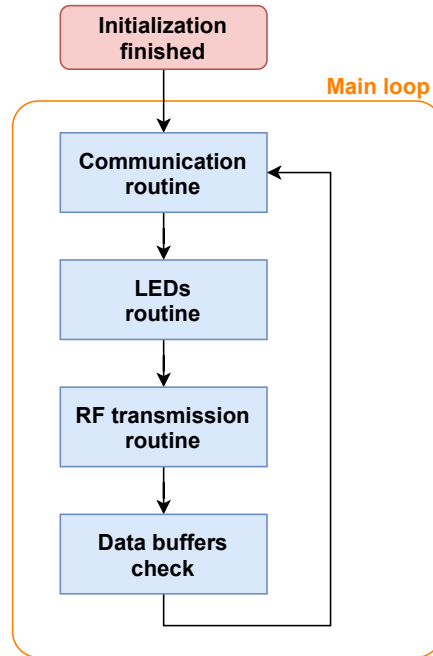


Figure 4.22: FW main loop diagram

The main loop is also accompanied by a few interrupts and a DMA execution. The DMA is employed for the FTDI (UART) transmission, and in other cases, it was deduced that the resulting overhead would be worse because the DMA reception of the data of an unknown length is not a trivial problem and also because often the execution is blocked or skipped nevertheless until confirmation of completion is obtained.

Apart from the system ones, the communication interrupts are utilized (USART, USB, and DMA global interrupts) to help process the communication data (such as putting the received data into buffers and setting execution flags for further actions).

■ 4.4.4 Modules

The module with the main loop function connects all the other modules, as described in Figure 4.23.

The main module serves as the highest layer that defines and runs the functions that often utilize lower-layer modules, as previously described in Figure 4.22.

The modules then access the peripherals utilizing the commonly used module *cmn_utils* that provides several utilities (such as *typedefs*, *enums*) that are shared across all of the components.

The *RDK TX Service* is an exception as it is an abstraction of the *CC1101*

module, which directly communicates with the selected RF IC. The *RDKit TX Service* generalizes many functionalities of interlinking between the RF IC and the main module, further easing the possible module reusability and portability.

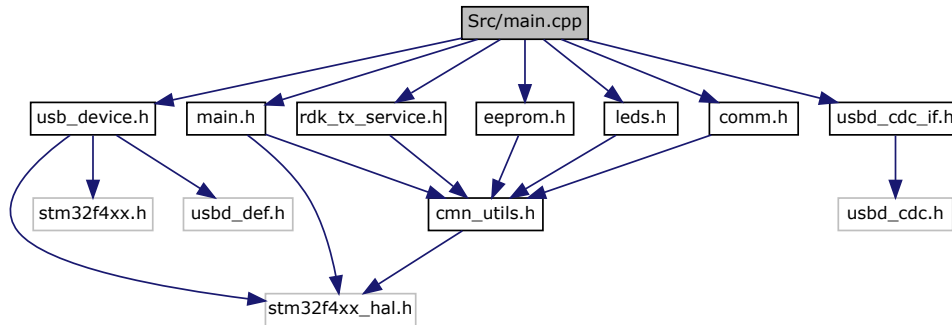


Figure 4.23: FW *Main* module dependency graph

Also, the *Comm* module defines a class suitable as a unifying abstraction of the communication (USB, UART, CAN), adding a layer above the *HAL*. It serves as an interface for derived concrete classes (*Usb*, *Can*, *Uart*) that provide methods for other modules, e.g. buffers processing or transmission. The actual low-level data reception is interrupt-based.

4.5 GUI

Apart from the Transmitter unit itself, an application called the Configurator was necessary for editing its settings. We have therefore decided to create a GUI that serves as a higher layer for the Transmitter unit configuration over the USB serial line (or FTDI). This GUI is based on *Qt* libraries⁶, which has ensured multi-platform portability.

4.5.1 Front-end

Excluding various dialogue windows such as file saving or opening, the front-end comprises effectively of the *Main* window and the *Settings* window.

Main window

The *Main* window serves for the configuration of the Transmitter unit itself once connected. It is also a starting point for the user as all the software's features and windows are accessible from it. Apart from the *About* informative window in the *Help* section are all rest accessible both from the top bar with text aids or from the informative panel with icons. The design of the *Main* window may be seen in Figure 4.24.

⁶<https://www.qt.io/>

It is possible to save the TPMS configuration into and load from the files with a custom *rsc* extension with the effective internal structure of an Extensible Markup Language (XML) file.

The *Main* window further provides the possibility to connect to or disconnect from the Transmitter unit and either set up its internal configuration with the one created in this software or retrieve the one currently used in the physical device.

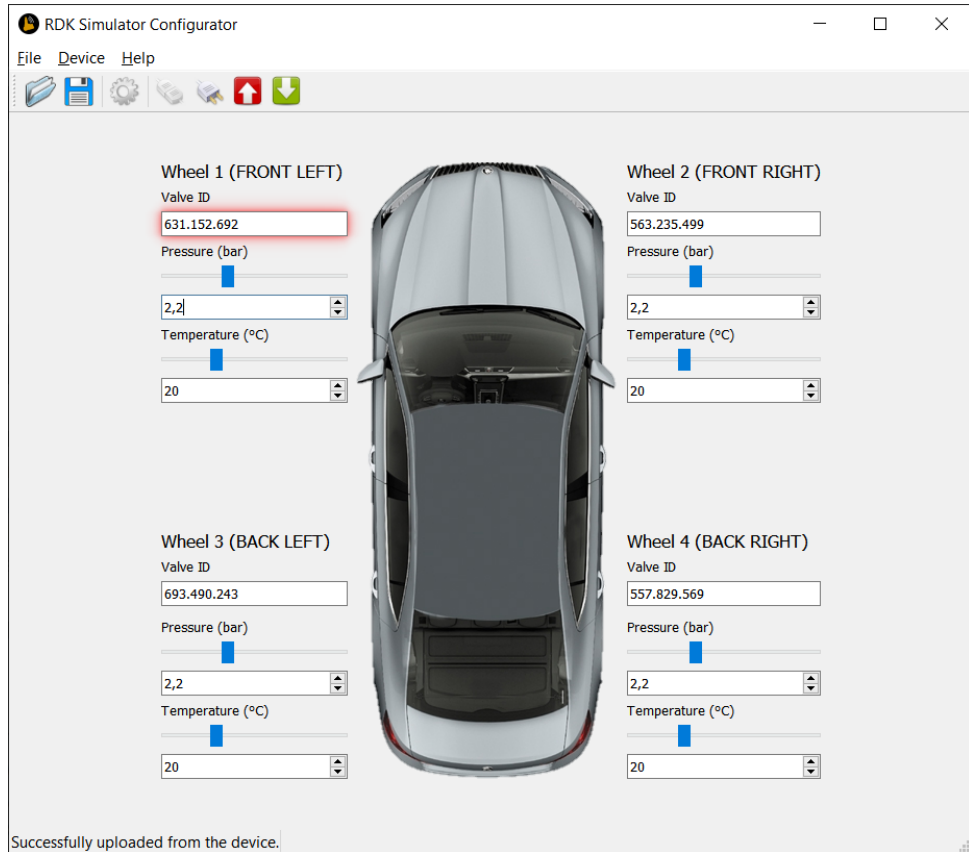


Figure 4.24: GUI *Main* window

Functionally, the user may change the following properties of the configuration:

- TPMS smart valve ID (decimal format),
- pressure [bar],
- temperature [°C].

However, before successfully changing the parameters, the user must at least once correctly set up the USB connection to the Transmitter unit in the *Settings* window as described in the next subsection 4.5.1.

The user is also notified of the parameters he has changed differently from the Transmitter unit's current settings by a red border around the respective graphical field.

■ Settings window

For the user to connect to the Transmitter unit, the connection must be configured first in the *Settings* window. The design of it is displayed in Figure 4.25. Next time when starting the software, setting this up is no longer necessary unless there has been a change of port for some reason.

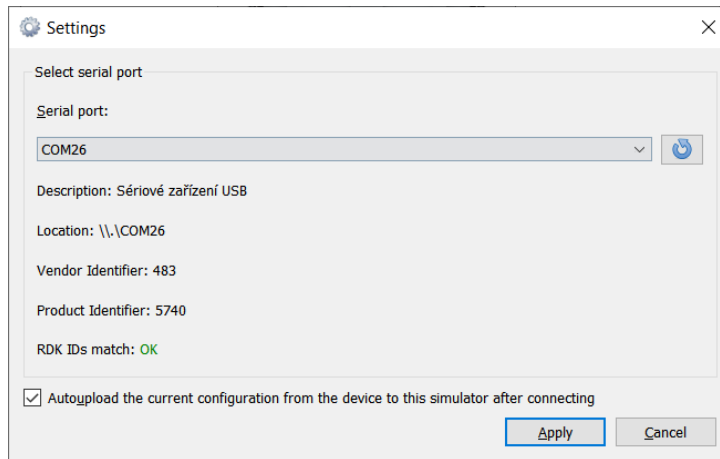


Figure 4.25: GUI *Settings* window

The user must select the Serial port that is occupied by the Transmitter unit, and if the “RDK IDs match” item reports OK, the USB Vendor and Product identifiers are valid then the port selection may be confirmed and applied. There is also the default possibility to load the configuration from the Transmitter unit to the Configurator after a successful connection, which must be initiated in the main window after applying the settings in this *Settings* window.

■ 4.5.2 Back-end

The software is written using the C++ language with the *Qt* libraries which effectively enable a build both on the *Windows* and *Linux*-based systems. The code is object-oriented and we will, therefore, provide a description of classes.

■ Description of classes

The program is run from the *Main* module. Its dependencies are shown in the Figure 4.26. The *Main* window core logic is then located in the *MainWindow* class and the *Settings* window’s logic in the *SettingsDialog* class.

The core classes are described by a method of a class diagram in Figure 4.28. This description does not include the automatically generated files by the *Qt Creator* Integrated Development Environment (IDE), the visual look of the software was developed using the *UI Design* tool that provides a graphical interface for programmers to create the GUI elements layout. Let us now describe the classes in more detail.

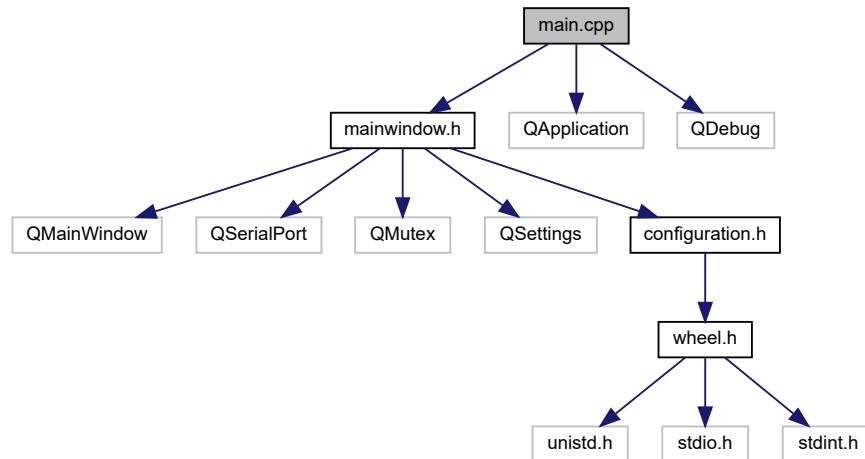


Figure 4.26: GUI *Main* module dependency graph

The *Wheel* class is a simple class abstracting a real wheel from the view of the TPMS. This means such a virtual wheel is fully described by parameters such as position on the car, smart valve’s ID, pressure.

Multiple (four in our case) *Wheel* objects are then included in the *Configuration* class that provides complete information of all the wheels from the point of view of the TPMS.

The *MainWindow* class includes the core logic of the entire program as well as the graphical backbone for the main window. This practically means that the *MainWindow* owns a *Configuration* object and that the communication with the Transmitter unit device is driven from this class, i.e., various messages over the serial port (USB or UART for debugging) are sent, received and evaluated. The sent messages may include variable data that are generated based on the user-selected values of the physical entities, as described in Subsection 4.5.1. The reception is done via a signal triggered when data are ready to be read in the serial port. Evaluation of the messages is done via a pointer to a callback function of the *CommExchange* object. The *MainWindow* also manages the possible repetitions of sending messages on no or invalid responses and takes care of locking and unlocking mutexes to avoid possible problems such as deadlocks or conflicts in the received data expectancy.

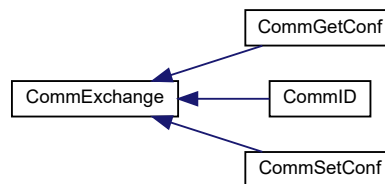


Figure 4.27: *CommExchange* class inheritance diagram

The *CommExchange* abstracts a simple communication between the GUI and the Transmitter unit. This process can be sufficiently described by a few parameters based on the communication type – what should be sent, what

should be received, and what should be done on successful execution or failure. It is functionally connected to the *MainWindows* by keeping a reference to it. *CommExchange* itself provides an interface of virtual functions that have to be implemented by the inheriting classes *CommID*, *CommGetConf* and *CommSetConf* as illustrated in Figure 4.27.

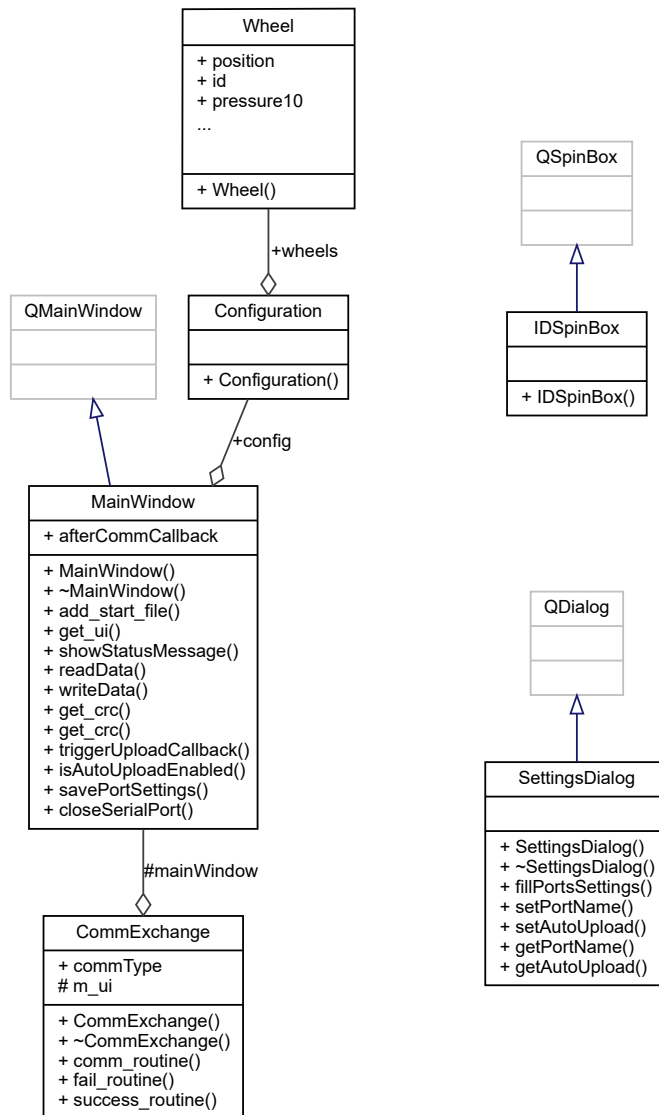


Figure 4.28: GUI class diagram

These classes are concrete implementations for some communication processes – *CommID* is when the identification of the connected Transmitter unit over the serial port needs to be obtained, *CommGetConf* is for the *Configuration* to be constructed based on the real settings of the Transmitter unit and *CommSetConf* for the opposite case, i.e., transferring the user configuration to the device.

The concrete *CommExchange* classes always implement the communication process based on the respective commands sent over the serial line as described

in appendix B, table B.1.

The *IDSpinBox* is an own re-implementation of the *QSpinBox*, necessary for a custom behaviour in the GUI. It is used by the *UI Design* tool.

The *SettingDialog* class is very analogical to the *MainWindow* but for the case of the *Settings* window. It allows the user to set up the parameters of the connection to the Transmitter unit. The correct Serial port must be selected from the enumeration of all the available ones.

Chapter 5

Implementation process

The first device we made was a prototype based on a development kit by *ST*, NUCLEO-F401RE. The final version includes a custom PCB, a slightly different MCU (STM32F446RC), many improvements and is ready for small series production. We have been developing the simulator since September 2019.

5.1 Prototype

The prototype was being developed back-to-back along with the GUI (Section 4.5) and CAN analysis (Section 3.4). The used library for firmware was the *Mbed* that while providing a great layer of abstraction over low-level programming, is insufficient as, e.g., remapping various peripherals may require non-trivial tampering the library's internals.

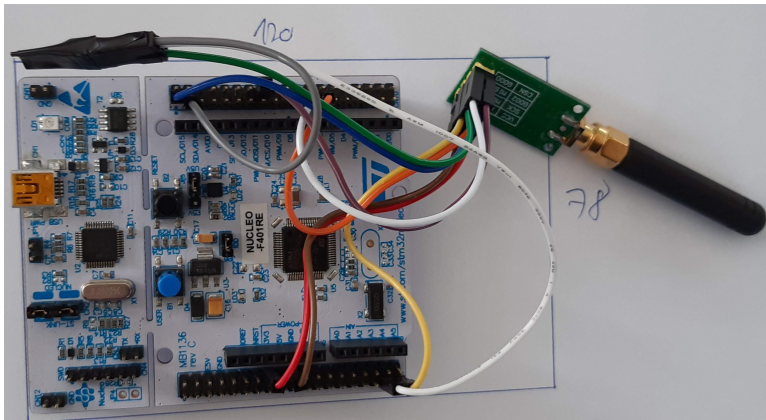


Figure 5.1: Prototype internals – NUCLEO-F401RE and the CC1101 module

The core of the prototype can be seen in Figure 5.1. It served mostly as a sandbox for testing the possibilities of the RF IC. Once we had reached the state where the communication with the TPMS ECU was successful (more in Section 5.3), we have developed the GUI, done some debugging, performed design improvements and then released the prototype for testing in ŠKODA AUTO. Then the development of the final product has begun based on various

feedback.



Figure 5.2: Finished prototype – front view



Figure 5.3: Finished prototype – back view

The prototype hardware was fastened inside the enclosure using a supportive prototype PCB. The LEDs were held in place using the special through-hole connectors, and the antenna is mounted from the outside. The final product can be seen in Figure 5.2 and Figure 5.3.

■ 5.2 Small series device

The prototype served as a good base for functional and user testing. It also validated the choice of the modules such as the MCU, the RF IC, the LEDs, or the USB. However, in principle, almost no components remained the same.

Newly added features include the input jack for power supply, CAN connectivity, and EEPROM for longer memory life. A new fully custom design has been developed as described in Chapter 4. The final product can be seen in Figure 5.4 and Figure 5.5.



Figure 5.4: Finished small series device – front view



Figure 5.5: Finished small series device – back view

■ 5.2.1 Hardware

Two PCBs were ordered from the Czech manufacturer *Pragoboard*. Further manual soldering and assembly was required.

The initial assembly was without greater problems and mostly confirmed the correctness of the design concept. The most significant realization issues included some traces destroyed due to the heat caused by multiple soldering and desoldering due to testing. Also a minimal MCU pinout change was performed. These issues were fixed using the microwires. The assembled PCB can be seen in Figure 5.6.

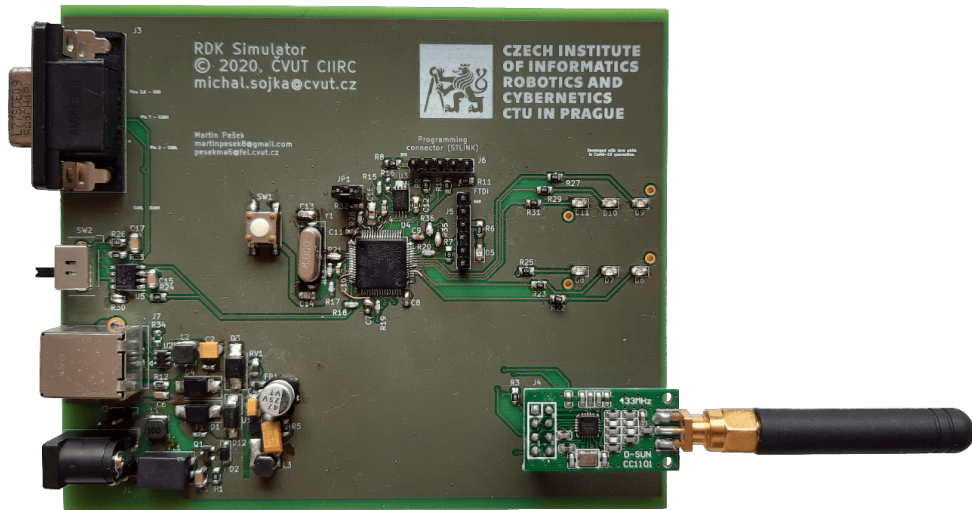


Figure 5.6: Assembled and working PCB

Power consumption measurements

A crucial parameter of a mobile device is its power consumption. Special care during design has been taken when addressing this by utilizing a DC/DC step down converter instead of a simple linear regulator. We were thus able to easily comply with USB power specifications (less than 100 mA at 5 V without further configuration [37, p. 178]). Measurements of Transitter unit power consumption were done for two cases – a peak and average power consumption. All of the measured values are shown in Table 5.1 with the most significant values marked bold.

V_{in} [V]	I_{max} [mA]	I [mA]	P_{max} [mW]	P [mW]
4	43.7	37.15	174.80	148.60
5	35.01	29.46	175.05	147.28
6	29.42	24.79	176.52	148.74
7	25.36	21.39	177.52	149.70
8	22.39	18.91	179.12	151.28
9	20.02	16.92	180.18	152.28
10	18.17	15.37	181.70	153.65
11	16.64	14.09	183.04	154.94
12	15.4	13.01	184.80	156.12
13	14.33	12.14	186.29	157.76
14	13.48	11.43	188.72	159.95
15	12.66	10.74	189.90	161.03

Table 5.1: Power consumption measurements

The large portion of power (about 30 %) drawn is used only to light up the LEDs, which shows that the rest of the electronics is low-power suitable for mobile devices. The measurements were carried out at room temperature.

5.2.2 Firmware

The firmware was completely ported due to many issues with the previously used *Mbed* libraries in combination with the used commercial *IAR EW* compiler. Comprehensive analysis on this matter can be found in the appendix D but in short terms, it was necessary to gain more low-level control over the MCU and port the code to a compiler that is easily usable with a less restrictive non-commercial license. The natural choice was *GNU Arm Toolchain*, the used components of which have licenses that are open-source and free [48].

While porting from the combination of *IAR EW/Mbed* fortunately preserves a lot of reusable C/C++ source code, some components had to be changed, mainly the ones with a low-level character. Many *Mbed* modules were ported to *ST's HAL* libraries that allow for a more advanced MCU hardware control. From the point of changing the compiler, various precompiler directives, linker scripts, and start-up assembly codes had to be modified. The core of the firmware (the higher layer) has stayed mostly the same thanks to the general code decomposition.

With useful debugging methods, including the oscilloscope observation and logic analyzers utilization, we were able to port the code.

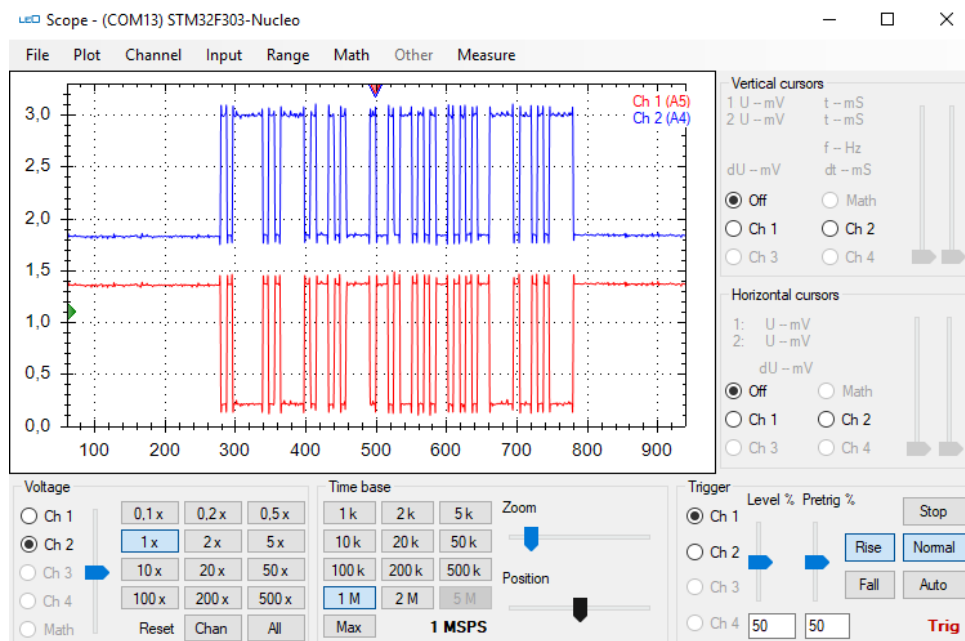


Figure 5.7: CAN analysis using the *Little Embedded Oscilloscope*

Due to the issues caused by the Covid-19 pandemic, access to professional tools was limited. Fortunately, even in the home environment, we were able to use the *LEO – Little Embedded Oscilloscope*¹, a USB digital oscilloscope with GUI, running on NUCLEO-F303RE with no extra attachments needed.

¹<https://leo.fel.cvut.cz/>

This tool was developed by the CTU FEE, Department of measurement, and is publicly available. The screenshot of its GUI can be seen in Figure 5.7.

The most significant setbacks during firmware development included problems related to the DFU functionality and late exchange of MCU (formerly STM32F401RE) for another one from the same series (STM32F446RC). The MCU exchange was necessary due to an initial design flaw since the former MCU does not incorporate an embedded CAN controller, unlike the latter one. Fortunately, since they are both from the same F4 series, switching them was not an excessively demanding task as it consisted of altering linker scripts, start-up assembly code, CAN pinout and system clock configuration, which is now actually capable of extended options thanks to the greater possibilities of Phase-Locked Loop (PLL).

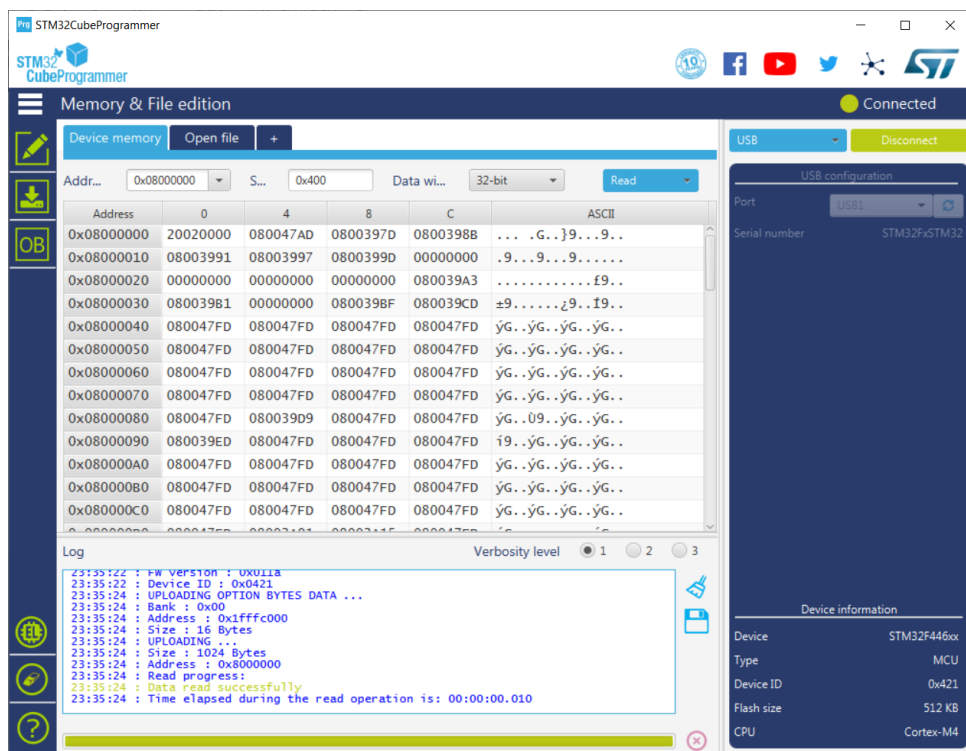


Figure 5.8: *STM32CubeProgrammer* – a possible software solution for managing the DFU of the MCU

At the same time, this has resolved problems with the DFU functionality. The issue was that the bootloader routine of the STM32F401 seemed to be failing in the phase of the detection of the external crystal oscillator where an error causes a system reset that makes the default code from the Flash memory be run on the next start-up (i.e., our firmware) [49, p. 122]. It was therefore impossible to connect to the DFU device on one's computer because it was never enumerated over the USB in the first place.

Although this issue was extensively discussed on the *ST* community forum²

²<https://community.st.com/s/question/0D53W0000041c5NSAQ/>

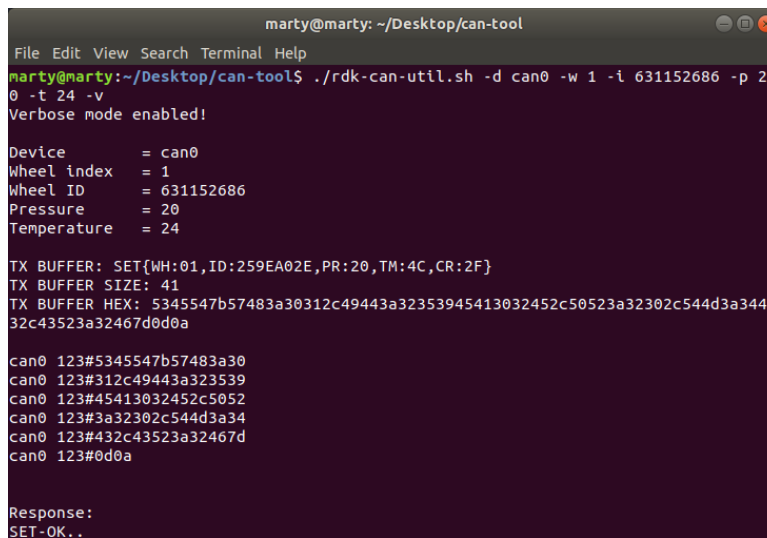
with many interesting suggestions, none of them were effective. Changing of external crystal oscillator for many different ones was tried, as well as resoldering or selecting different impedance-matching resistors, cables, connectors, computers with different operating systems with various drivers. The only resolution to this problem was ultimately a choice of a different MCU that did not display such behaviour.

The DFU functionality was not a necessary feature, but a very convenient one, since the programmer or even the user may update the Transmitter unit with, e.g., a provided binary file using only the USB cable and respective software. This feature naturally does not support debugging since it only loads the new firmware into the memory. We used the *STM32CubeProgrammer*³ software to perform the necessary DFU operations as seen in Figure 5.8.

5.2.3 CAN Command-line configurator

Mainly for the internal and future use, a command-line tool for configuring the Transmitter unit over the CAN bus has been developed. It is a simple *bash* script with a supportive executable file that needs to be built using the *Make* tool. It serves as a wrapper for the *cansend* command from the *can-utils*⁴ package.

This tool serves the same purpose as the previously described GUI – to set up the Transmitter unit with the user-configuration of virtual wheels but not over the USB but rather over the CAN. In contrast to the GUI, it provides a command-line interface instead of a graphical one, as seen in Figure 5.9.



```

marty@marty: ~/Desktop/can-tool
File Edit View Search Terminal Help
marty@marty:~/Desktop/can-tool$ ./rdk-can-util.sh -d can0 -w 1 -i 631152686 -p 2
0 -t 24 -v
Verbose mode enabled!

Device           = can0
Wheel index      = 1
Wheel ID         = 631152686
Pressure         = 20
Temperature      = 24

TX BUFFER: SET{WH:01, ID:259EA02E, PR:20, TM:4C, CR:2F}
TX BUFFER SIZE: 41
TX BUFFER HEX: 5345547b57483a30312c49443a32353945413032452c50523a32302c544d3a344
32c43523a32467d0d0a

can0 123#5345547b57483a30
can0 123#312c49443a323539
can0 123#45413032452c5052
can0 123#3a32302c544d3a34
can0 123#432c43523a32467d
can0 123#0d0a

Response:
SET-OK..

```

Figure 5.9: CAN TPMS Simulator Configurator

This tool fully respects the communication protocol described in the appendix B.

³<https://www.st.com/en/development-tools/stm32cubeprog.html>

⁴<https://github.com/linux-can/can-utils>

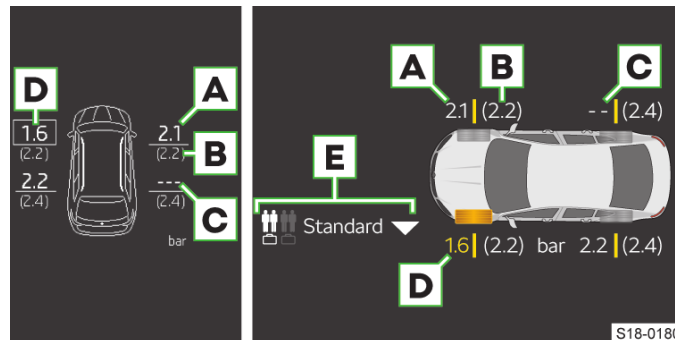
5.3 Design verification

The verification of the prototype design correctness was carried out in multiple phases.

Initially we only used the *Ggrx* mentioned in Section 3.1 to see if the RF IC transmits with the similar parameters as the TPMS smart valves. When it did, we tried decoding the messages sent by the Transmitter unit using the *rtl_433* in the same manner as we decoded the real TPMS smart valves messages.

Then we moved on to using the TPMS ECU. Small tampering of the transmission parameters of the Transmitter unit had to be done first. Once we knew that the TPMS ECU reported the data sent by our simulator, it was clear that the transmission parameters are correct (at least to a functional extent).

The subsequent process of the precise calibration was simple. Since we already had a rough estimate of the gain and offset constants of some measured physical entities as mentioned in Chapter 3, we sent some roughly-known data from the Transmitter unit to the TPMS ECU. Then we read the interpreted received messages and then we were able to modify the constants based on differences between the supposed values and the values interpreted by the TPMS ECU.



- A** Tyre pressure
- B** Recommended tyre pressure
- C** Tyre pressure not available
- D** Tyre pressure too low
- E** Adjust the vehicle's loading condition

Figure 5.10: Infotainment TPMS screen [5]

Afterwards, we have performed tests in laboratories of ŠKODA AUTO. We were provided with a car suspended in the air with its wheels moving freely. Firstly we used the SDR to capture the data coming out of the actual TPMS valves in real tires.

This way, we found out the IDs of real valves and were able to configure the Transmitter unit with these IDs and custom values. We were changing the

pressure of the Transmitter unit's virtual tires and observed the infotainment display, illustrated in Figure 5.10. The infotainment reported changes in wheels' pressure.

It was clear that our simulator has worked and has been able to replicate the actual TPMS valves as the vehicle's infotainment reported the pressure inside tires such that we had set up in the Transmitter unit.

Once the ID of the valves is discovered (by reading the physically written number on the valve itself, by decoding the RF transmission or by obtaining it over the CAN communication), it is sufficient to convince the TPMS ECU and masquerade as the real smart valves.

After the design verification, the release candidate of the GUI was created and given along with the prototype Transmitter unit to the responsible people of ŠKODA AUTO for user testing and feedback. After the handover, we have started to work on the more advanced device suitable as a small series device that would incorporate a custom PCB design and more features.



Chapter 6

Conclusion

The goal of this thesis was to create a TPMS Simulator for the needs of ŠKODA AUTO. That means a device capable of simulating the TPMS smart valves that measure pressure and temperature inside the car tires. The developed simulator can transmit the custom data to the TPMS ECU which are indistinguishable from the real valve messages. This ultimately means that the car receives this custom data and acts based on it – displays it in the infotainment and issues a warning.

We have fully succeeded in creating the TPMS Simulator consisting of a graphical user interface and a physical device – Transmitter unit, the settings of which are configurable either over USB using the GUI or over CAN. We proved it to be working in ŠKODA AUTO's laboratories. The Transmitter unit was designed with a focus on low power consumption and a low production price. We were able to do that by designing a schematic that incorporated a custom PCB with minimalistic solutions such as an MCU or an RF IC (unlike, e.g., a complex SDR with the ability to transmit).

The main benefit of the TPMS Simulator is the eased internal testing in ŠKODA AUTO by removing the need to manipulate the real physical entities concerning every single wheel. The user is informed of the runtime status by the use of LEDs with descriptions, and since the device may be powered by voltage in the range of 4 V to 15 V from a power jack or 5 V from the USB, it is fully portable by connecting, e.g., a USB power bank.



Appendix A

List of used acronyms

ABS	Anti-lock Brake System
AGC	Automatic Gain Control
AM	Amplitude modulation
ASK	Amplitude-Shift Keying
BPF	Band-Pass Filter
CAN	Controller Area Network
CRC	Cyclic redundancy check
DC	Direct current
DFU	Device Firmware Upgrade
DMA	Direct Memory Access
DSP	Digital Signal Processor
ECU	Electronic Control Unit
EDA	Electronic Design Automation
EEPROM	Electrically Erasable Programmable Read-Only Memory
EMI	Electromagnetic interference
FFT	Fast Fourier transform
FM	Frequency Modulation
FSK	Frequency-Shift Keying
FW	Firmware
GPIO	General Purpose Input Output
GUI	Graphical User Interface

A. List of used acronyms

HAL	Hardware Abstraction Layer
HW	Hardware
I2C	Inter-Integrated Circuit
ID	Identification
IDE	Integrated Development Environment
IEC	International Electrotechnical Commission
IP	Intellectual Property
IQ	In-Phase, Quadrature
IrDA	Infrared Data Association
ISM	Industrial, Scientific and Medical
LED	Light Emitting Diode
LL	Low Level
LNA	Low-Noise Amplifier
LO	Local Oscillator
OS	Operating System
PCB	Printed Circuit Board
PLL	Phase-Locked Loop
PSK	Phase-Shift keying
PWM	Pulse Width Modulation
QAM	Quadrature Amplitude Modulation
RDK	Reifendruckkontrolle
RF	Radio frequency
RISC	Reduced Instruction Set Computer
RSP	Receive-Signal Processor
SCPI	Standard Commands for Programmable Instruments
SDR	Software Defined Radio
SMD	Surface Mount Device
SMT	Surface Mount Technology

SPI	Serial Peripheral Interface
SWD	Serial Wire Debug
THT	Through Hole Technology
TPMS	Tire Pressure Monitoring System
TVS	Transient-Voltage Suppression
U(S)ART	Universal (Synchronous) Asynchronous Receiver Transmit
USB	Universal Serial Bus
XML	Extensible Markup Language

Appendix B

TPMS Simulator communication protocol description

The communication protocol between the Transmitter unit and its control is partially based on the philosophy, syntax, and semantics of Standard Commands for Programmable Instruments (SCPI), the extension of IEEE 488.2 [50]. However, it also introduces some different features. The communication protocol is carried out over the USB's serial line port, CAN, or for debugging purposes a simple UART. The physical layer parameters are the following for UART and USB serial lines:

- 8 data bits,
- no parity,
- 1 stop bit,
- 9600 bauds,
- no flow control.

And for the CAN:

- standard ID by default set to 0x123 – may be redefined in FW,
- no extended ID,
- data frame,
- bitrate of 500 kbps.

The basic description of the protocol itself is the following.

- Communication is executed by the transmission of queries to the Transmitter unit and receiving responses.
- The valid queries comprise one or more commands.
- Multiple commands inside a query are separated by a semi-colon symbol “;”. The semi-colon is not necessary after the last command in the query.

- Responses are for each respective commands, also separated by a semi-colon “;”.
- Received query is parsed as soon as a newline (“\r\n”) is received.

Variables and arguments in general described in this Appendix are signified by an enclosing left-pointing bracket “<” and right-pointing bracket “>” symbols. These brackets are not sent over the communication channel.

■ B.1 Query structure

The structure of a query is following:

<COMMAND 1>; ... ;<COMMAND N>\r\n

And the response:

<RESPONSE 1>; ... ;<RESPONSE N>\r\n

■ B.1.1 Commands definition

All possible commands that may be transmitted in a query are described in the following table. Some commands may include an additional argument/variable.

Command	Description	Response
*IDN?	Instrument identification	“RDK-SIMULATOR-<VERSION>” with the version number as suffix, e.g. “RDK-SIMULATOR-0.9”.
SET<CONF>	(Re)configures a single wheel with the wheel configuration subsystem as an argument.	“SET-OK” on success, “SET-ERR” on failure.
GET-CONFIG	Gets the configuration of all wheels	Four “STATUS<CONF>” messages with each wheel’s full description by the wheel configuration subsystem as an argument.
DBG	Toggles debug (verbose) mode	“DEBUG-ACTIVATED!” on the activation of debug mode, “DEBUG-DEACTIVATED!” otherwise.
DFU-MODE	Runs the DFU mode	“DFU mode will now initiate and this communication will be terminated!”.

Table B.1: Control commands set

The Transmitter unit debug (verbose) mode provides more internal information about the program runtime. This data is sent as an ASCII text

separated by newline characters and is useful for various debugging purposes and development.

If the DFU mode of the Transmitter unit is enabled, then the device is available for enumeration over USB as an *STM32 BOOTLOADER* device. The process must further be handled by a tool that can load the new FW over the DFU protocol, such as *STM32CubeProgrammer* or *dfu-util*.

■ The wheel configuration subsystem

It is a set of parameters describing the state of a single wheel. They must be enclosed in curly brackets “{” and “}”. This system may follow as an argument for the SET command sent to the Transmitter unit or for the STATUS command sent by the Transmitter unit.

Each parameter’s name inside this subsystem is defined by two upper-case ASCII letters and the value followed after a colon “:” written after the parameter’s name. The length of the value data is strictly given, as described below. The value data that are in hexadecimal format do not have any preceding characters as “0x”. Parameters are separated by commas “,”.

The *WH* (wheel index) and *CR* (CRC result) parameters with correct values must compulsorily be included at the start and the end respectively, otherwise the command shall be erroneous and not executed. The definition of parameters is described below. The subsystem itself may be written as:

{<PARAM 1>:<VALUE 1>, ... ,<PARAM N>:<VALUE N>} \r\n

Wheel index. The wheel index is a compulsory parameter that has to be included. It describes to which wheel is the whole wheel configuration subsystem related.

Parameter name: WH

Value size (bytes): 1

Value	Meaning
0	Front left wheel
1	Front right wheel
2	Back left wheel
3	Back right wheel

Table B.2: Wheel index values

TPMS valve ID. This parameter defines the ID of the respective TPMS valve. The value must be in a hexadecimal format!

Parameter name: ID

Value size (bytes): 4

Legal values: Any 4 byte number.

Pressure. The pressure inside the virtual tire. The format is decimal in [bar] multiplied by a gain of 10 (e.g., 20 $\hat{=}$ 2.0 bar).

Parameter name: PR

Value size (bytes): 1

Values: Any, in format 10·[bar].

Temperature. The temperature inside the virtual tire. The format is [°C] with a positive offset of 52 (decimal) to avoid negative numbers. The number to be sent must be converted to a hexadecimal format (e.g., 48 (hex) $\hat{=}$ 20 °C).

Parameter name: TM

Value size (bytes): 1

Values: Any, in format [°C] + 52.

CRC result. The second compulsory parameter that must be attached at the end of the subsystem. It is the hexadecimal result of a CRC calculated from the ASCII values of the whole subsystem string (preceding the *CR* parameter) to be sent (excluding enclosing curly brackets and directly preceding comma). The used CRC configuration is CRC-8-CCITT (0x07) with no initial seed.

Parameter name: CR

Value size (bytes): 1

Values: Any correctly calculated 1 byte CRC result.

Example: Let the wheel configuration subsystem string be:

{WH:00,ID:259EA02E,PR:22,TM:48}

The base for calculation is:

WH:00,ID:259EA02E,PR:22,TM:48

The CRC has to be therefore calculated from the following sequence of bytes in hexadecimal format. CRC configuration is CRC-8-CCITT (0x07) with no initial seed.

57, 48, 3a, 30, 30, 2c, 49, 44, 3a, 32, 35, 39, 45, 41, 30, 32, 45, 2c, 50, 52, 3a, 32, 32, 2c, 54, 4d, 3a, 34, 38

The calculated CRC is 0x02. The final string shall be:

{WH:00,ID:259EA02E,PR:22,TM:48,CR:02}

■ B.1.2 Query examples

Practical examples follow. The newline symbol (“\r\n”) terminating a query is always explicitly written when present in communication (at the end of query or response).

■ Example 1

Retrieve identification.

Query:

```
*IDN?\r\n
```

Response:

```
RDK-SIMULATOR-0.9\r\n
```

■ Example 2

Retrieve identification, get the configuration of all the wheels, and activate the debug mode.

Query:

```
*IDN?;GET-CONFIG;DBG\r\n
```

Response:

```
RDK-SIMULATOR-0.9;
STATUS{WH:00,ID:259EA02E,PR:22,TM:48,CR:02};
STATUS{WH:01,ID:21924AAB,PR:22,TM:48,CR:F6};
STATUS{WH:02,ID:2955D243,PR:22,TM:48,CR:7F};
STATUS{WH:03,ID:213FCDC1,PR:22,TM:48,CR:0D};
DEBUG-ACTIVATED!\r\n
```

■ Example 3

Set all the wheels with the respective IDs, the pressure of all of them to 2.2 bar and temperature of all of them to 20 °C with a faulty CRC in the case of the last wheel.

Query:

```
SET{WH:00,ID:259EA02E,PR:22,TM:48,CR:02};
SET{WH:01,ID:21924AAB,PR:22,TM:48,CR:F6};
SET{WH:02,ID:2955D243,PR:22,TM:48,CR:7F};
SET{WH:03,ID:213FCDC1,PR:22,TM:48,CR:FF}\r\n
```

Response:

```
SET-OK;SET-OK;SET-OK;SET-ERR\r\n
```


Appendix C

Complete schematics and PCB layout

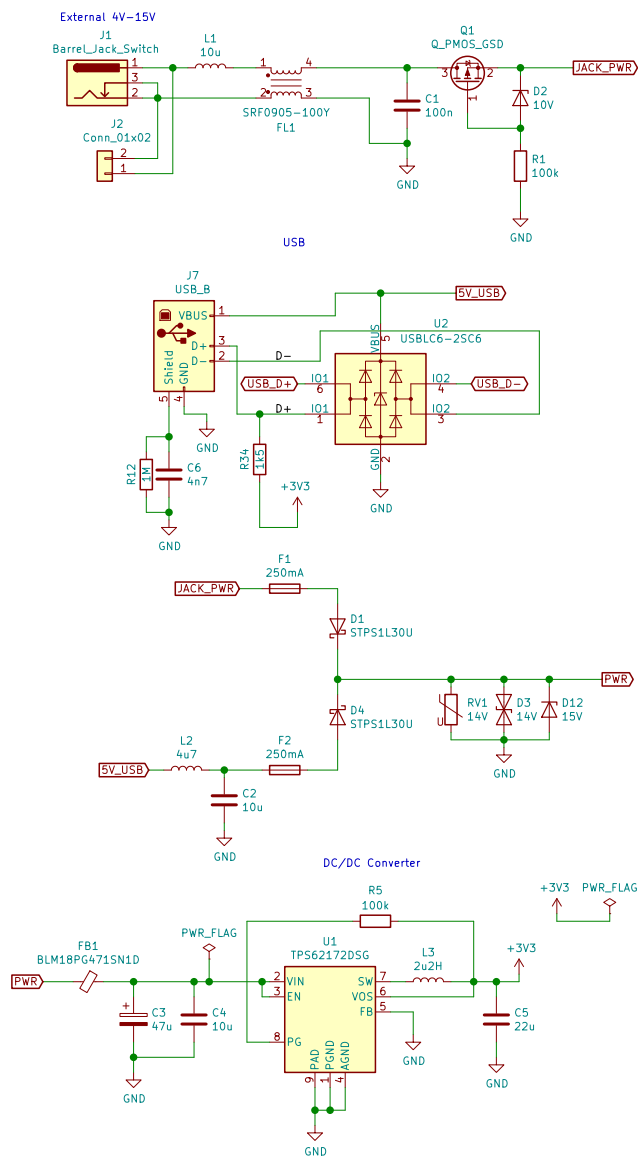


Figure C.1: Power supply and USB schematics

C. Complete schematics and PCB layout

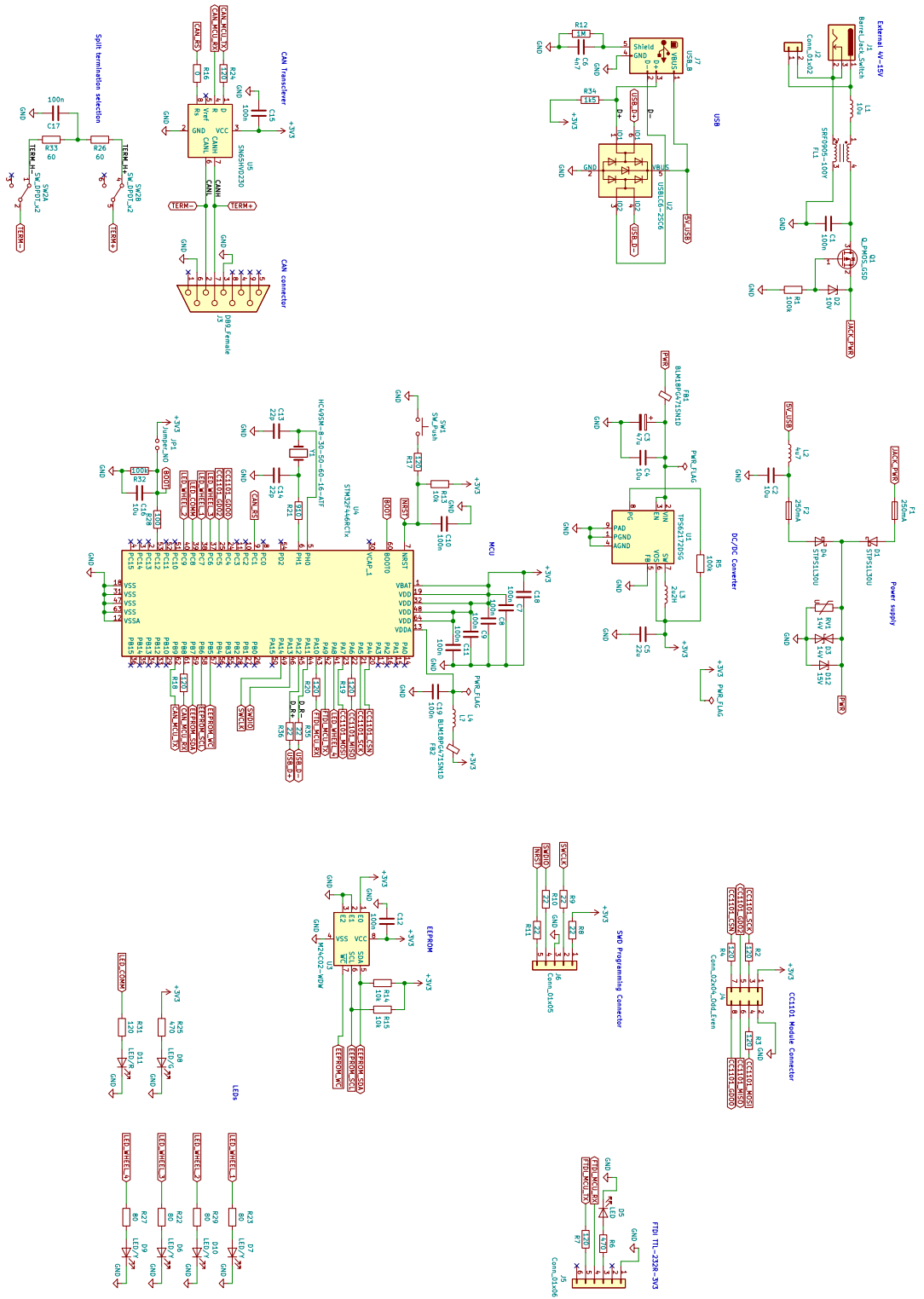


Figure C.2: MCU and peripherals schematics

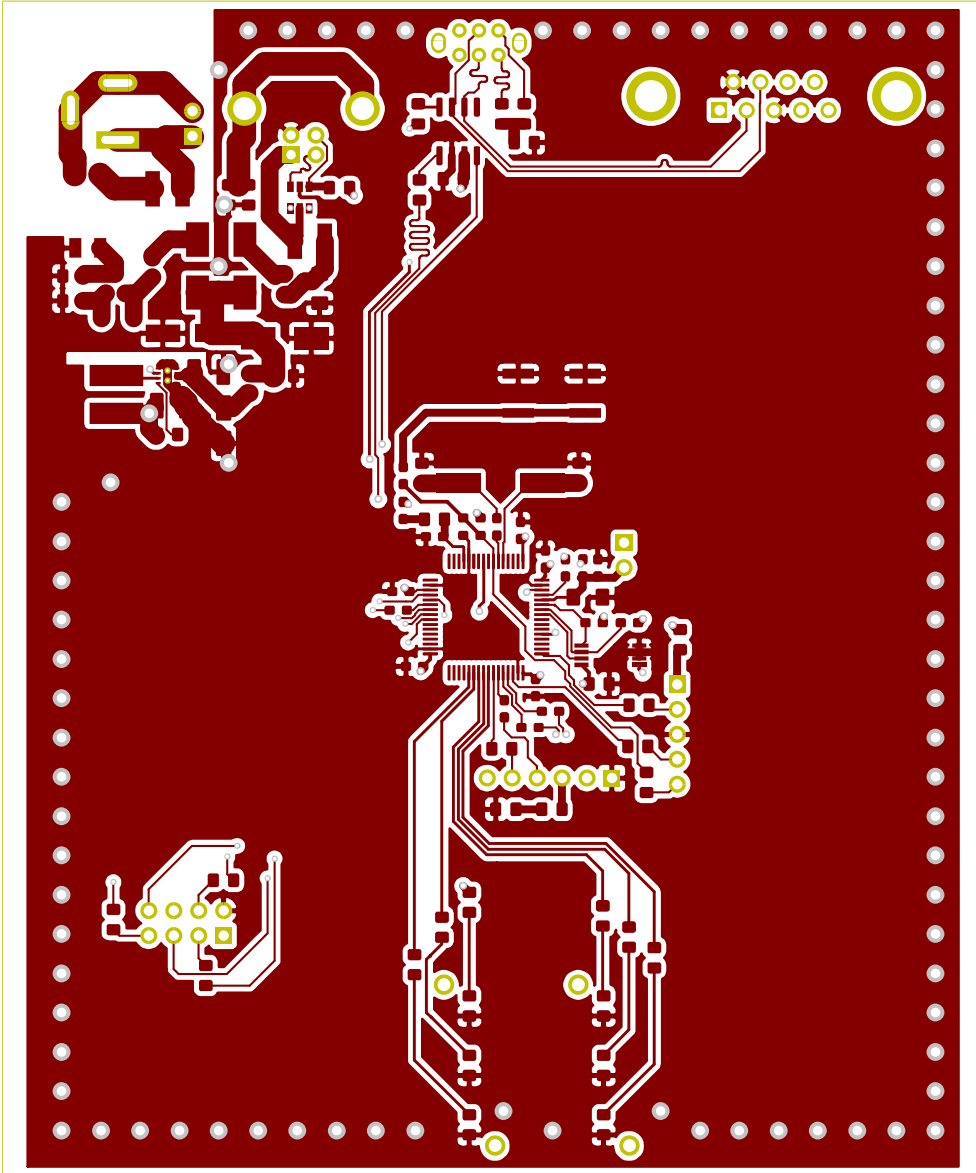


Figure C.3: Top PCB layer

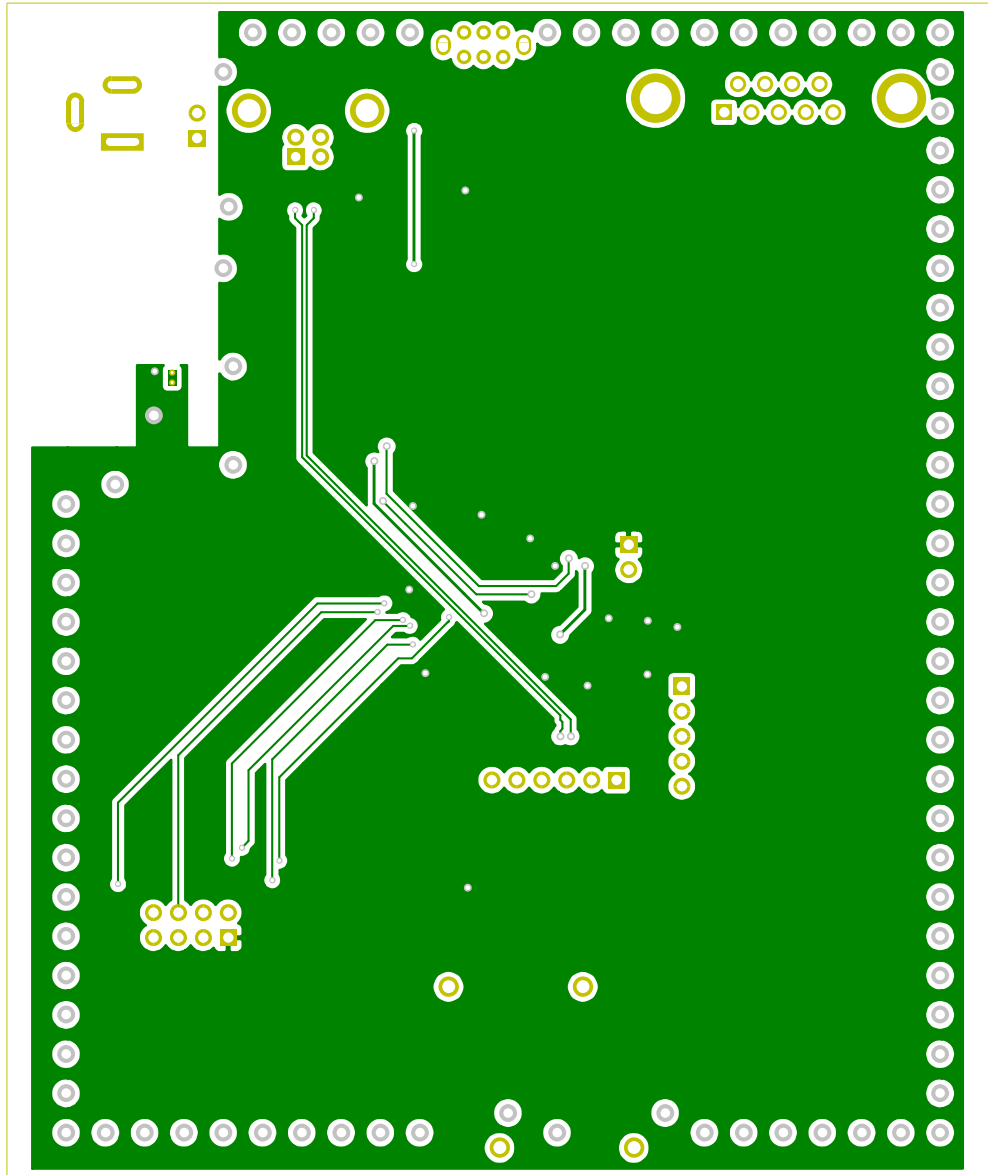


Figure C.4: Bottom PCB layer

Appendix D

IAR EW & Mbed vs GNU Arm Toolchain & HAL comparison

During the development of the prototype, the combination of *IAR EW IDE* with its toolchain along with *Mbed* libraries were used. Later in the project was the source code ported to the *GNU Arm Toolchain* with *HAL* libraries utilized. This chapter concisely describes the differences between the used methods.

D.1 IAR EW vs GNU Arm Toolchain

D.1.1 Licensing

The greatest difference between the two toolchains is their licensing. *IAR EW* is a professional set of tools with a restrictive custom commercial license [51]. *GNU Arm Toolchain* provides the possibility to work only with open-source components that are freely licensed by such as *GNU GPL* or *BSD* licenses [48].

The licensing also implies that there is official customer support available for the *IAR EW* unlike the *GNU Arm Toolchain* where the community substantially and extensively provides the support.

D.1.2 Functional safety

IAR EW has an integrated support for functional safety (norms IEC 61508, ISO 26262, EN 50128, EN 50657 and IEC 62304 [52]) as there are software versions certified by *TÜV* and there is also a source code *MISRA* compliance static analysis. *GNU Arm Toolchain* does not provide such functionality which may be critical for many automotive, medical, or industrial systems.

D.1.3 IDE functionality

The *IAR EW* is a complex all-in-one software. It includes not only all tools necessary for compiling, linking but also for debugging and code development. It is a full-fledged IDE with many standard features such as a source code indexer, formatter or an embedded debugger. While the installed binaries

take up space in order of GiB, it also universally supports a great number of any Arm-based processors.

The *GNU Arm Toolchain* is by default a command-line tool. However, there is a great amount of support available for, e.g., the *Eclipse*-based IDEs. This is the case of the IDE used by us, *STM32CubeIDE*¹, that has the core of the *Eclipse* and utilizes the *GNU Arm Toolchain*. Subjectively for our use, the *STM32CubeIDE* provided a vastly more user-friendly environment and more convenient features.

■ D.1.4 Miscellaneous

Apart from the points mentioned above, there is also one important difference and that is the cross-platform support. *IAR EW* provides no native support for any other OS than *Windows*, while the *GNU Arm Toolchain* may generally be run on *Windows*, *Linux* and *Mac OS X*.

■ D.1.5 Conclusion

As usual, both have their upsides and downsides but in our case the *GNU Arm Toolchain* is much more beneficial.

For various industrial, medical, or automotive applications, the *IAR EW* license (or other similar commercial software) may be a necessary solution due to the available safety certification, customer support, and an all-in-one solution.

However, for non-safety development, applications where little to no certification is necessary (e.g. when creating a device that is not sold to customers), where buying a commercial license is a substantial burden or where the cross-OS support is needed, *GNU Arm Toolchain* is a satisfactory choice, and that is our case.

■ D.2 Mbed vs HAL libraries usage

During the prototype development were the *Mbed* tools used but during the later phase we relied solely on *HAL* libraries. It should first be mentioned that the direct comparison *Mbed* vs *HAL* is not appropriate. The reason is that the *Mbed* is a large platform providing a great number of features. The developer's source code is run on top of the *Mbed OS*. This can also practically be interpreted as the *Mbed* libraries being a layer above *HAL*. In fact the *Mbed* actually incorporates the *HAL*. Both are freely-licensed and open-source.

■ D.2.1 Abstraction

A great level of abstraction is available in *Mbed* thanks to its structure. That naturally results in the following beneficial features:

¹<https://www.st.com/en/development-tools/stm32cubeide.html>

- code portability between various *Mbed-enabled* devices,
- no custom assembly start-up scripts needed,
- no extensive custom configuration of MCU's internals necessary,
- easy inclusion of various external libraries.

All of these features are granted by the fact that *Mbed* libraries are a layer above *HAL*. Nevertheless, the abstraction naturally brings the expected downsides:

- worsened code performance,
- increased code size,
- problematic pinout remapping – need to modify the *Mbed* libraries source code in an invasive way.

■ D.2.2 Miscellaneous

Mbed also introduces an online IDE. That is a handy feature as the user may compile for his device at any point when an internet browser is available. Expectedly debugging is not currently supported and the project has to be exported in format for another desktop tool first.

Subjectively analyzing the user experience with the *Mbed* community forums, the support appears to be somewhat insufficient at the time. That seems to be also caused by the abstraction, when the problem may often be that a *Mbed* module does not cooperate with specific hardware.

■ D.2.3 Conclusion

Mbed is a very ambitious project. At this time, we would suggest it for basic projects, where one does not need to utilize every bit of performance potential and where the memory size is not a critical issue.

The abstraction of *Mbed* is the key point that induces all the benefits and downsides. If the developer has to make ends meet with the MCU he has, *Mbed* is not an appropriate choice. Such cases may include, e.g., a need for fast code performance or small size, using various alternative pinouts or directly accessing registers of peripherals. While the *Mbed* includes the *HAL* libraries, they should not be accessed directly in order not to lose the portability. Of course, *HAL* itself is already an abstraction of the internals of series of *STM* MCU series (such as *F4*) and itself may be for many developers a too extensive or abstractive implementation. In some exceptional cases, the developer might even need to write the assembly code.

The need for a more advanced low-level access to the MCU was the primary motivation to ultimately keep using the *HAL* libraries.

Appendix E

Bibliography

- [1] Hydrargyrum. TPMS warning icon, August 2010. URL https://commons.wikimedia.org/wiki/File:TPMS_warning_icon.svg. [Online; accessed 3-March-2020].
- [2] Pavel Kovář. FUNDAMENTALS OF DIGITAL AND ANALOG RADIO COMMUNICATION, 2019.
- [3] Walt Kester. Which ADC architecture is right for your application. *Analog Dialogue*, 39(2), 07 2005.
- [4] Hammond Manufacturing. 1455N1201, 2014. URL <http://www.hammondmfg.com/pdf/1455N1201.pdf>. [Online; accessed 1-May-2020].
- [5] ŠKODA AUTO a.s. OWNER´S MANUAL ŠKODA KODIAQ, 2019. URL https://ws.skoda-auto.com/OwnersManualService/Data/en/Kodiaq_NS/11-2019/Manual/Kodiaq/A_SUV_Kodiaq_OwnersManual.pdf. [Online; accessed 1-May-2020].
- [6] Randy Frank. *Understanding Smart Sensors*. Artech House Remote Sensing Library. Artech House, 2013. ISBN 9781608075072.
- [7] Stephan van Zyl, Sam van Goethem, Stratis Kanarachos, Martin Rexeis, Stefan Hausberger, and Richard Smokers. Study on Tyre Pressure Monitoring Systems (TPMS) as a means to reduce Light-Commercial and Heavy-Duty Vehicles fuel consumption and CO₂ emissions. Technical report, TNO, July 2013. [Online; accessed 3-March-2020].
- [8] Ibrahim Mustafić, Fuad Klisura, and Sabahudin Jasarevic. INTRODUCTION AND APPLICATION OF TIRE PRESSURE MONITORING SYSTEM. In *3rd Conference MAINTENANCE 2014 Ženica, B&H*, 06 2014. doi: 10.13140/2.1.1829.7127.
- [9] Swindon Silicon Systems. The Global Adoption of TPMS and the ASIC within, July 2018. URL <https://www.swindonsilicon.com/the-global-adoption-of-tpms-and-the-asic-within>. [Online; accessed 3-March-2020].

- [10] NIRA Dynamics AB. TPMS Fitment and Tyre Inflation Pressures, 2018. URL <https://www.unece.org/fileadmin/DAM/trans/doc/2018/wp29grrf/GRRF-86-17e.pdf>. [Online; accessed 3-March-2020].
- [11] Swindon Silicon Systems. TYRE PRESSURE MONITORING SYSTEMS – TPMS PAST, PRESENT AND FUTURE PRODUCTION, April 2018. URL <https://www.swindonsilicon.com/tyre-pressure-monitoring-systems-tpms-past-present-and-future-production/>. [Online; accessed 3-March-2020].
- [12] Transparency Market Research. Tire Pressure Monitoring System Market, 2018. URL <https://www.transparencymarketresearch.com/tire-pressure-monitoring-system-automotive-market.html>. [Online; accessed 3-March-2020].
- [13] Research and Markets. Global and China Tire Pressure Monitoring System (TPMS) Market 2017-2021 - 8 Foreign, 25 Chinese Mainland, 6 Taiwanese TPMS Vendors and 5 TPMS Sensor Chip Companies, January 2018. URL <https://www.prnewswire.com/news-releases/global-and-china-tire-pressure-monitoring-system-tpms-market-2017-2021---8-foreign-25-chinese-mainland-6-taiwanese-tpms-vendors-and-5-tpms-sensor-chip-companies-300581937.html>. [Online; accessed 3-March-2020].
- [14] NIRA Dynamics AB. Boost in chinese tpms market: mandatory legislation from 2019, December 2016. URL <https://niradynamics.se/boost-in-chinese-tpms-market-mandatory-legislation-from-2019/>. [Online; accessed 3-March-2020].
- [15] MarketWatch. Automotive Tyre Pressure Monitoring System (TPMS) Market 2019 Global Industry Size, Future Trends, Market Size & Growth, Demand, Business Share, Sales & Income, Manufacture Players, Application, Scope, and Opportunities Analysis by Outlook – 2023, August 2019. URL <https://www.marketwatch.com/press-release/automotive-tyre-pressure-monitoring-system-tpms-market-2019-global-industry-size-future-trends-market-size-growth-demand-business-share-sales-income-manufacture-players-application-scope-and-opportunities-analysis-by-outlook-2023-2019-08-20>. [Online; accessed 3-March-2020].
- [16] Volkswagen AG. *Reifendrucküberwachungssysteme - Konstruktion und Funktion*, 2014.
- [17] Volkswagen AG. *Diagnosedokumentation RDK BERU 30*, 2017.
- [18] Gilbert Held. *Inter- and Intra-Vehicle Communications*. CRC Press, 2007. ISBN 9781420052220.

- [19] Bridgestone. The History of Run Flat Tyres and Bridgestone, September 2007. URL <https://www.bridgestone.co.za/news-article/579/the-history-of-run-flat-tyres-and-bridgestone>. [Online; accessed 3-March-2020].
- [20] National Highway Traffic Safety Administration. Review of the Office of Defects Investigation, 2002. URL <https://www.oig.dot.gov/sites/default/files/mh2002071.pdf>. [Online; accessed 3-March-2020].
- [21] Liqiang Wang, Zhe Zhang, Yanna Yao, Zongqi Han, and Wang Bin. Monitoring Method of Indirect TPMS Under Steering Situation. *DEStech Transactions on Engineering and Technology Research*, 05 2017. doi: 10.12783/dtetr/mime2016/10229.
- [22] Dirk Hammerschmidt. Indirect tire pressure monitoring systems and methods using multidimensional resonance frequency analysis, US9527352B2, 2013. URL <https://patents.google.com/patent/US9527352B2/en>. [Online; accessed 3-March-2020].
- [23] Martin Gotschlich. Indirect tire pressure monitoring systems and methods, US20140060170A1, 2011. URL <https://patents.google.com/patent/US20140060170A1/en>. [Online; accessed 3-March-2020].
- [24] Zhou Fuqiang, Wang Shaohong, Wei Yintao, and Zhichao Xu. Indirect tire pressure monitoring system based on tire vertical stiffness. In *2015 12th IEEE International Conference on Electronic Measurement & Instruments (ICEMI)*, pages 100–104, 07 2015. doi: 10.1109/ICEMI.2015.7494213.
- [25] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, July 1948. ISSN 0005-8580. doi: 10.1002/j.1538-7305.1948.tb01338.x.
- [26] Kiho Cho, Jae Choi, and Soo Kim. An acoustic data transmission system based on audio data hiding: method and performance evaluation. *EURASIP Journal on Audio, Speech, and Music Processing*, 2015, 12 2015. doi: 10.1186/s13636-015-0053-x.
- [27] Tom Hornak. *Efficient Linear RF Power Amplification with Constant Envelope Output Stages*. 1995.
- [28] W.C.Y. Lee. *Mobile Communications Engineering: Theory and Applications*. McGraw-Hill Education, 1997. ISBN 9780071500128.
- [29] Carlos Caicedo. Software Defined Radio and Software Radio Technology: Concepts and Applications. In *International Telecommunications Research and Education Association ITERA*, 01 2007.
- [30] National Instruments. What is I/Q Data?, October 2019. URL <http://www.ni.com/tutorial/4805/en/>. [Online; accessed 3-March-2020].

- [31] Mieczyslaw Kokar and Leszek Lechowicz. Language Issues for Cognitive Radio. *Proceedings of the IEEE*, 97(4):689 – 707, 05 2009. doi: 10.1109/JPROC.2009.2013028.
- [32] T. Vazão, T. Vazao, M.M. Freire, and I. Chong. *Information Networking. Towards Ubiquitous Networking and Services: International Conference, ICOIN 2007, Estoril, Portugal, January 23-25, 2007, Revised Selected Papers*. Computer Communication Networks and Telecommunications. Springer, 2008. ISBN 9783540895237.
- [33] S.C. Mukhopadhyay and H. Leung. *Advances in Wireless Sensors and Sensor Networks*. Lecture Notes in Electrical Engineering. Springer Berlin Heidelberg, 2010. ISBN 9783642127076.
- [34] R. Zurawski. *Industrial Communication Technology Handbook*. Industrial Information Technology. CRC Press, 2017. ISBN 9781482207330.
- [35] Williams Ross N. A PAINLESS GUIDE TO CRC ERROR DETECTION ALGORITHMS, 1993. URL https://www.zlib.net/crc_v3.txt. [Online; accessed 1-May-2020].
- [36] Universal Serial Bus Implementers Forum. Full and Low Speed Electrical and Interoperability Compliance Test Procedure, 2004. URL https://www.usb.org/sites/default/files/USB-IFTTestProc1_3.pdf. [Online; accessed 1-May-2020].
- [37] Compaq, Hewlett-Packard, Intel, Lucentand Microsoft, NEC, and Philips. Universal Serial Bus Specification, 2000. URL http://sdpha2.ucsd.edu/Lab_Equip_Manuals/usb_20.pdf. [Online; accessed 1-May-2020].
- [38] Texas Instruments. TPS6217x3-V to 17-V, 0.5-A Step-Down Converters with DCS-Control™, 2017. URL <http://www.ti.com/lit/ds/symlink/tps62172.pdf>. [Online; accessed 1-May-2020].
- [39] STMicroelectronics. STM32F446xC/E, 2019. URL <https://www.st.com/resource/en/datasheet/stm32f446re.pdf>. [Online; accessed 8-May-2020].
- [40] Texas Instruments. CC1101 Low-Power Sub-1 GHz RF Transceiver, 2013. URL <http://www.ti.com/lit/ds/symlink/cc1101.pdf>. [Online; accessed 1-May-2020].
- [41] STMicroelectronics. AN4894 Application Note, 2018. URL https://www.st.com/resource/en/application_note/dm00311483-EEPROM-emulation-techniques-and-software-for-stm3214-and-stm3214-series-microcontrollers-stmicroelectronics.pdf. [Online; accessed 1-May-2020].
- [42] Kolokowsky Steve and Davis Trevor. Common USB Development Mistakes – You Don’t Have To Make Them All Yourself!, 2006. URL

- <https://www.cypress.com/file/88486/download>. [Online; accessed 1-May-2020].
- [43] Corrigan Steve. Common-mode effects in CAN networks. *CAN Newsletter*, pages 52 – 54, 2012. URL <https://can-newsletter.org/uploads/media/raw/c190a7e99e71c41e1bf4692d5da58cd1.pdf>. [Online; accessed 1-May-2020].
- [44] Vít Záhlava. *Návrh a konstrukce desek plošných spojů: principy a pravidla praktického návrhu*. BEN - technická literatura, 2010. ISBN 9788073002664.
- [45] M.I. Montrose. *Printed Circuit Board Design Techniques for EMC Compliance: A Handbook for Designers*. IEEE Press Series on Electronics Technology. Wiley, 2000. ISBN 9780780353763.
- [46] Brent Fischthal and Mike Cieslinski. Beyond 0402M placement: Process considerations for 03015M microchip mounting., 2014. URL https://www.panasonicfa.com/sites/default/files/pdfs/Beyond0402MPlacement_ProcessConsiderationsfor03015M_Panasonic.pdf. [Online; accessed 1-May-2018].
- [47] Yuena Michael, Benedict Heather, Havlovitz Kris, Pitsch Tim, and Mackie Andy C. Tombstoning of 0402 and 0201 components: A study examining the effects of various process and design parameters on ultra-small passive devices., 2014. URL <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.594.655&rep=rep1&type=pdf>. [Online; accessed 1-May-2020].
- [48] Arm Developer. EULA, 2019. URL <https://developer.arm.com/GetEula?Id=c7f2905b-bb90-4f58-8d92-29c57529e789>. [Online; accessed 1-May-2020].
- [49] STMicroelectronics. AN2606 Application Note, 2019. URL https://www.st.com/resource/en/application_note/cd00167594-stm32-microcontroller-system-memory-boot-mode-stmicroelectronics.pdf. [Online; accessed 11-May-2020].
- [50] National Instruments. Documentation About the SCPI Specification, 2018. URL <https://knowledge.ni.com/KnowledgeArticleDetails?id=kA00Z0000019S48SAE>. [Online; accessed 30-April-2020].
- [51] IAR Systems. SOFTWARE LICENSE AGREEMENT, 2020. URL <https://www.iar.com/globalassets/pdf/softwarelicenseagreement-february-2020.pdf>. [Online; accessed 3-May-2020].
- [52] IAR Systems. Certified tools for functional safety, 2020. URL <https://www.iar.com/iar-embedded-workbench/certified-tools-for-functional-safety/>. [Online; accessed 3-May-2020].