TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology

ITI40LT

Aleks-Daniel Jakimenko-Aleksejev 134502IABB

LIBRERVAC: FREE FIRMWARE FOR ROBOTIC VACUUM CLEANERS

Bachelor's thesis

Supervisor: Gert Kanter
Master of Science
Lecturer

Tallinn 2016

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

ITI40LT

Aleks-Daniel Jakimenko-Aleksejev 134502IABB

LIBRERVAC: ROBOTTOLMUIMEJATE VABA PÜSIVARA

Bakalaureusetöö

Juhendaja: Gert Kanter
Magister
Lektor

Tallinn 2016

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the materials used, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Aleks-Daniel Jakimenko-Aleksejev

2016-05-23

Abstract

Robotic vacuum cleaners have proprietary firmware which the users cannot modify and improve.

Instead of trying to work around these problems, it was decided to write a complete firmware replacement.

During this project a firmware replacement has been developed. This document offers a comprehensive walkthrough of methods, design decisions and implementation details. Results of this project are published under free licenses.

All of the groundwork has been done and the project is ready to transition to community-driven development.

This thesis is written in English and is 51 pages long, including 7 chapters, 13 figures and 3 tables.

Annotatsioon

LibreRVAC: Robottolmuimejate vaba püsivara

Robottolmuimejates kasutatava suletud lähtekoodiga püsivara tõttu ei saa kasutajad robotite käitumist muuta ega parendada.

Selle asemel, et püüda olemasolevat lahendust täiendada, otsustati luua täiesti uus püsivara, mis oleks enamiku tolmuimejatega kokkusobiv.

Käesolevas dokumendis on kirjeldatud rakendatud meetodid, disaini otsused ja implementatsiooni detailid. Kõik projekti tulemused on avalikustatud vaba litsentsi all.

Projekti kõige olulisemad osad realiseeriti ja edaspidi arendatakse seda vaba tarkvara projektina.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 51 leheküljel, 7 peatükki, 13 joonist, 3 tabelit.

List of Abbreviations and Terms

ARM	*Advanced RISC Machine* – proprietary family of architectures for computer processors
Bit banging	Using software instead of hardware to perform various tasks
bps	Bits per second
CPU	*Central processing unit* – in this document it is used to refer to ARM processors on SBCs
CV-SLAM	*Ceiling vision-based SLAM* – a variation of SLAM that uses a top-mounted camera
Dock	*Docking station* – used for charging the robotic vacuum cleaners. Robots are usually capable of finding it and connecting to it automatically.
Falling edge	Transition in a digital signal from logic high to logic low
FPU	*Floating-point unit* – part of a microcontroller designed to do floating point operations
Free	In this document the word “free” is used to speak about freedom, not price
Flashing	The process of overwriting existing firmware
Linux	Refers to Linux kernel. When referring to the whole operating system, term *GNU/Linux* is used.
GNU/Linux	Unix-like operating system
HAL	*Hardware Abstraction Layer* or *Hardware Abstraction Library* – software that is used to abstract away hardware details and to make it possible to run the same software on different hardware
IC	*Integrated circuit* or just *Chip* – a component with electronic circuit embedded in it
Interrupt	A signal to the processor emitted by hardware (or software), usually causes the execution of an ISR
IPC	*Inter-process communication* – refers to mechanisms of communication between different processes on one operating system
ISP	*In-system programming* – the process of programming (flashing) a microcontroller when it is already installed into the system

ISR	*Interrupt service routine* or *Interrupt handler* – a function that is executed in response to an interrupt
JIT	*Just-in-time compilation* – a compilation that is performed at run time. It has some benefits over ahead-of-time compilation (AOT), mainly because of the extra information that is available at run time (such techniques include adaptive optimization and dynamic recompilation). JIT is an important part of many dynamic programming languages.
JTAG	A standard that is used for debugging and programming of various ICs
MCU	*Microcontroller unit* – a small computer on a single IC
Motherboard	In this document it is used to refer to a main PCB on robotic vacuum cleaners
Opamp	*Operational amplifier* – a component that is commonly used in analog circuitry
PCB	*Printed Circuit Board* – is a board that mechanically supports various components and has traces to electrically connect various points together
Programmer	In this document it is used to refer to a physical device that performs ISP
Rising edge	Transition in a digital signal from logic low to logic high
RX	“receive”, “receiver” or “reception”
𝕊	*Samsung Navibot SR8730* – robotic vacuum cleaner
SBC	*Single-board computer* – a complete computer built on a single PCB. Sometimes also referred to as *GNU/Linux board*. In this thesis it is considered to be one of the most important parts that does most of the high-level processing.
SLAM	*Simultaneous localization and mapping* – a process of constructing a map of the surrounding environment while keeping track of the location of the robotic vacuum cleaner
Trace	Conductive track on a PCB. Among other things it is used to electrically connect pins of different ICs.
TX	“transmit”, “transmitter” or “transmission”
Unix socket	Or *Unix domain socket*, or *IPC socket*. Used for data communication between two processes.
𝕍	*Vileda Relax* – robotic vacuum cleaner

Table of Contents

List of Figures

List of Tables

1 Introduction

Robotic vacuum cleaners are a common part of many modern households, yet many questions have recently appeared which make it clear that these devices have a limited potential due to the proprietary nature of their firmware. During the writing of this thesis, free firmware was designed and implemented, demonstrating that such a possibility is viable.

The author describes what has been done and the reasoning behind some of the design decisions. After reading this thesis the reader should be able to build upon the provided work, or, if required, to build his own solution.

In chapter 2, it is discussed why this project is needed and what kind of problems it may solve. Chapter 3 gives an overview of how the implementation process was approached. Chapter 4 describes some of the design decisions and implementation details. In chapter 5, the author covers things that were not implemented yet, and discusses different possibilities and the project future. Chapter 6 provides, where possible, a comparison between firmware provided by the manufacturer and the results of this thesis.

All of the source code is published. Different licenses are used for different parts:

- Most of the source code is released under GNU General Public License version 3 or later [1].
- Control panel is under GNU Affero General Public License version 3 or later [2].
- Wiki content and this document is under Creative Commons Attribution 4.0 International License (CC-BY 4.0) [3].
- Schematics and CAD files are also under GNU General Public License version 3 or later [1].

Links to the source code can be found in Appendix 1.

2 Problem Statement

Firmware for most existing robotic vacuum cleaners has several flaws:

- In some cases, the device may still be on the market, but the manufacturer is no longer interested in improving the software. This means that either there are no firmware updates at all, or these updates are very rare and are only issued during a short period of time after the product release. The result is that there are bugs that will never be fixed. For example, Neato Robotics have stopped providing any firmware updates [4][5].
- Most robotic vacuum cleaners do not have a control panel of any sort. That is, the user cannot change any settings, but it is often required since all environments are different. Furthermore, the user cannot control the robot manually, which may be required in certain circumstances.
- The user has no control over the software and no way to make sure that it is secure. Some vacuum cleaners are connected to the internet, which may lead to serious security issues.
- Due to vendor lock-in, cooperation between multiple robotic vacuum cleaners is not possible (at this moment, no cleaners are able to talk to each other even if they are the same model, let alone allow cooperation between different models and brands).

These issues may be addressed by creating a free alternative to the existing proprietary firmware.

The main goal is to develop a set of libraries and tools which, once installed, will control robotic vacuum cleaners.

While known issues of stock firmware will be addressed as much as possible, the main priority is to create a foundation that will allow further growth of the project.

2.1 Motivation

This project was inspired by other hardware-oriented projects like OpenTx [6], OpenWRT [7], Libreboot [8], Sigrok [9]. These projects seem to suggest that community-based alternatives to proprietary products are indeed possible, even though these projects face problems that pure software projects do not have to deal with. Such problems include:

- Lack of documentation (a lot of information has to be gathered by reverse engineering).
- Development environment may require complex setup or special hardware.
- Some libraries, compilers or HAL-s are non-free. This complicates the distribution of the resulting software or the source code, which may result in problems with integrity of the resulted software (e.g. version mismatch).
- Cost of the hardware. Only the users who already have the supported hardware may try out the project and help with the development.

Despite above-mentioned difficulties, these projects are actively maintained.

Considering existing efforts associated with the robotic vacuum cleaners (attempts to build robotic vacuum cleaners from scratch [10], reverse engineering of existing devices [11], attempts to gather a team to work on such projects [12], etc. [13][14][15]), I conclude there is also a significant demand for free firmware for robotic vacuum cleaners. As an owner of a robotic vacuum cleaner, I understand where this need is coming from.

2.2 Given Hardware

Even though the long-term plan is to add support for as many devices as possible, as a starting point only a few devices can be added. Therefore, two robotic vacuum cleaners were chosen. The choice is based solely upon the fact that the author already had these devices.

These devices are:

- Samsung Navibot SR8730 (shown in Figure 1)
- Vileda Relax (shown in Figure 2)

For convenience purposes I will be using 𝕊 and 𝕍 characters respectively to refer to these particular devices.

Figure 1. Samsung Navibot SR8730.

Figure 2. Vileda Relax.

2.3 Hardware Details

Even though the support was only added for two devices, two completely different microcontrollers have to be supported:

- 𝕊 – PIC32MX360F256L microcontroller (32-bit, 256kB flash, 32 kB RAM) [16]
- 𝕍 – STC90C58RD+ microcontroller (8-bit, 32kB flash, 1280 bytes RAM) [17]

Microcontrollers may be observed in Figure 3 and Figure 4 (biggest ICs in the middle of the board).

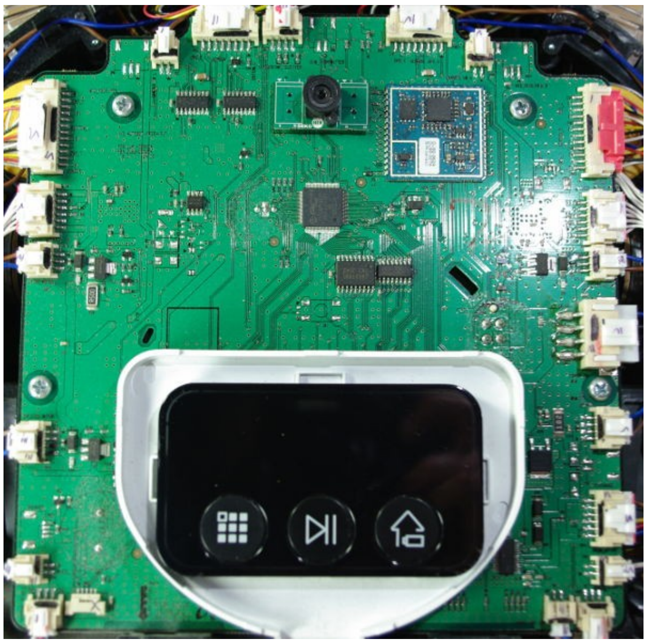


Figure 3. Samsung Navibot SR8730 motherboard.

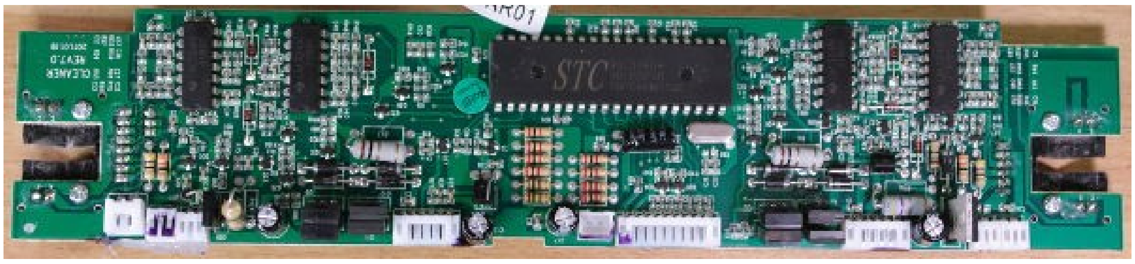


Figure 4. Vileda Relax motherboard.

Not only does the project have to support low-end microcontrollers, but it is also desirable to utilize features and extra memory provided by high-end devices. This raises a couple of challenges for the project. These challenges will be explained in section 4.4.

2.4 Reverse Engineering

Support for any device has to start from building a preliminary pinout table where each pin of a microcontroller has an associated feature. Such table is essential for device evaluation (determining whether support for this device is desired at all) and for early stages of writing the firmware. This table does not have to be entirely correct, mainly because many caveats and details will be figured out during the implementation, and also because there is no way to guarantee completely correct results at this stage. This table is only required to give the basic understanding of pin functions.

Aleksandr Boldin contributed preliminary pinout tables for three devices: 𝕊, 𝕍 and Severin RB7025. During the evaluation it was decided that Severin RB7025 hardware is inferior in too many ways, and custom firmware is not going to improve the situation.

Pinout tables for 𝕊 and 𝕍 were then refined, improved and corrected by me during the development process.

Table 1 is an excerpt from the full pinout table for 𝕊. The full pinout table may be found on the project wiki [18]. This excerpt is provided to demonstrate what a pinout table looks like.

Table 1. Excerpt from the pinout table for Samsung Navibot SR8730

Pin	**Type**	**Function**	**Pullup / Pulldown**	**Active low / high**	**Comment**
7	out	main brush			DRV8840 (pin 20 – PHASE)
8	out	main brush			DRV8840 (pin 19 – DECAY)
9	out	ic16 – C			14051BG ic16 (pin 9 – C)
10	out	motor right			A3950ST (pin 2 – MODE)
11	out	motor right			A3950ST (pin 3 – PHASE)
12	in	MID_RECEIVER_R	+3.3V	?	not connected
13	RESET	reset		low	
14	in	bumper switch left	+3.3V	low	
15	VSS				
16	VDD				
17	JTAG	mode select			

Pin	Type	Function	Pullup / Pulldown	Active low / high	Comment
18	in	rear right IR receiver	+3.3V	low	VIRTUAL_REAL_R (typo on silkscreen, has to be VIRTUAL_REAR_R)
19	in	button interrupt	+3.3V	low	all three buttons
20	in	main brush encoder	+3.3V	low	
21	ADC	battery thermistor			

During the reverse engineering (both by Aleksandr Boldin and by me) the firmware binary was not used. That is, all information was gathered by visual inspection and various maintenance methods (continuity tests, oscillograms, etc.). One of the reasons for not using any parts of the original firmware is to avoid legal issues that may exists under some jurisdictions.

Basically, to understand the function of some pin, all you have to do is follow the trace (e.g. visually) and see what is connected to it. On robotic vacuum cleaners, most things are connected to the motherboard via connectors. Therefore, by finding a pin that is wired to some port, you can figure out its function. Of course, it should not rely on just visual inspection, since any digital multimeter is capable of confirming whether two points are electrically connected (continuity test).

This is as easy as it sounds, except for cases that are complicated by multiplexers, current drivers, opamps, traces that branch under ICs, etc. However, all you have to do is repeat this process about 80 times (on 𝕊, the microcontroller has 100 pins, most of them are used) and you will get a good enough pinout table with which the firmware development process is rather easy.

Some pins, however, have to be studied in more detail. For example, on 𝕊, pin 24 was known to be associated with charging, but some details were missing. The easiest way to go about this is to look at the behavior of the original firmware. Figure 5 illustrates one of the ways of doing that.

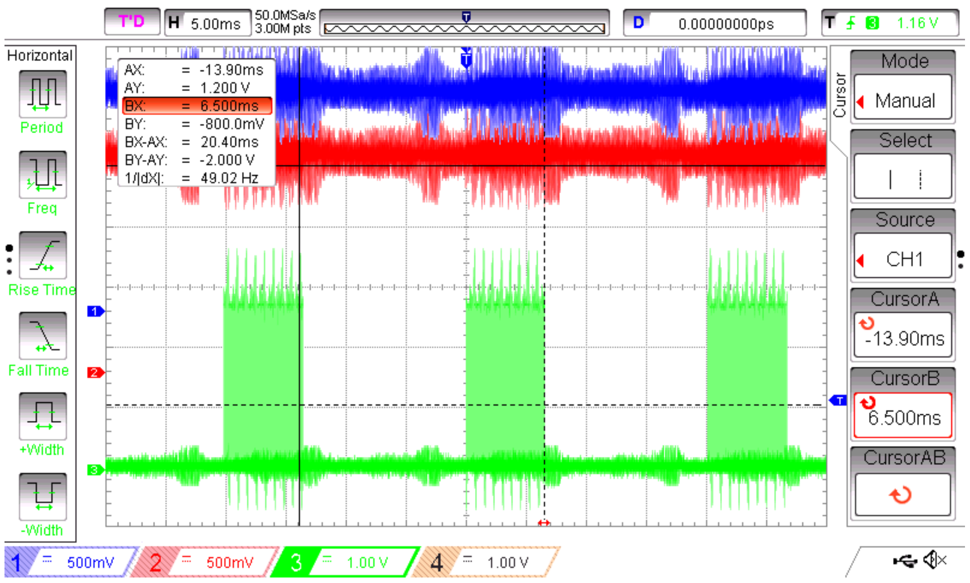

```
AX:     = -13.90ms
AY:     = 1.200 V
BX:     = 6.500ms
BY:     = -800.0mV
BX-AX:  = 20.40ms
BY-AY:  = -2.000 V
1/|dX|: = 49.02 Hz
```

Figure 5. Charging process (original firmware), captured with an oscilloscope.

Channel 3 (green) of the oscilloscope is connected to pin 24. This particular screenshot does not demonstrate the charging process in detail, but it gives an idea on how the device may be studied.

Appendix 3 is an excerpt from the source code that demonstrates the struggle that sometimes happens during the reverse engineering (the comments are explaining what happens in Figure 5).

One important point to note is that, when in doubt, original behavior may be replicated. It is unclear whether original implementation is actually the best solution, but by doing the same thing I can at least guarantee that my implementation is not worse. After all, battery charging is a process where special care should be taken to avoid battery degradation or fires.

However, the struggle does not end there. The docking station also has a microcontroller (which, by the way, is also reflashable, meaning that we can write a custom firmware for that too), and it does not allow the charging process to work properly unless you send a specific code via infrared LEDs. In Figure 6 such transmission can be seen.

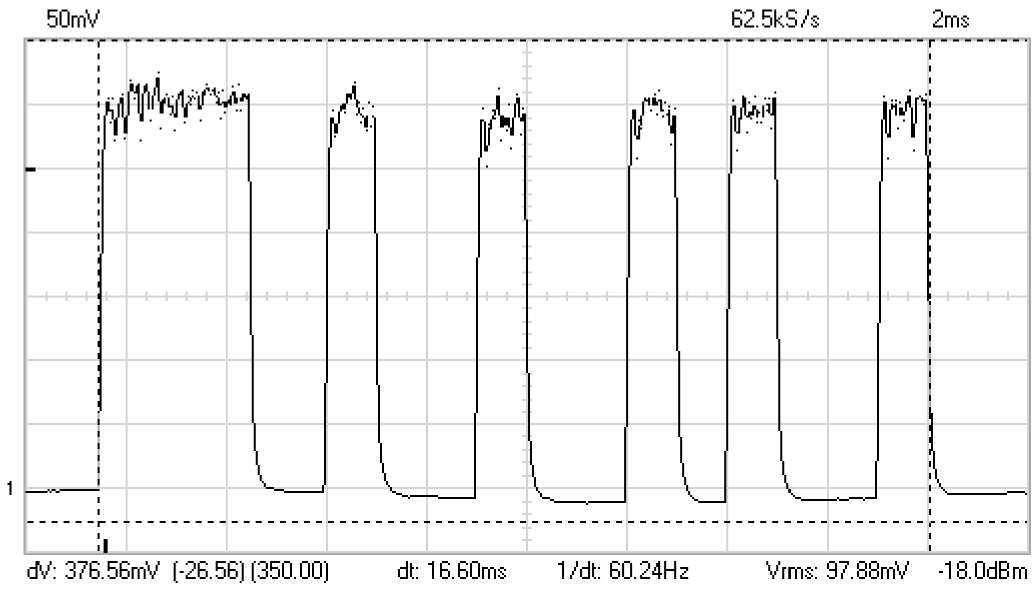


Figure 6. Signal sent to the dock (original firmware), captured with an oscilloscope.

In Figure 6, the signal is slightly distorted due to 38kHz modulation (docking station has a standard infrared receiver which works at this frequency, this is required for noise rejection).

The signal in Figure 6 may be represented by a 33-bit binary value *111111000110000110000110011000011*. This value is probably non-random (some protocol was used), but for writing our own implementation this is irrelevant. The important part is that if we keep sending this code periodically (which is what original firmware does), the dock will allow the robot to charge properly. There are some issues with the idea that the robot should send something in order to charge, but this will not be discussed here because at the moment of writing it is not planned to write a custom firmware for the docking station. In other words, it should be considered as hardware limitation for now.

To sum it up, while most parts can be easily reverse engineered, some particular things require significant amounts of effort.

2.5 Flashing

Most devices on the market are reflashable without hardware modifications (e.g. without replacing the chip). Manufacturers leave a way to change the firmware in case there is any problem with it, and we can happily use this.

As an example, in Figure 7 we can see maintenance ports on 𝕊.

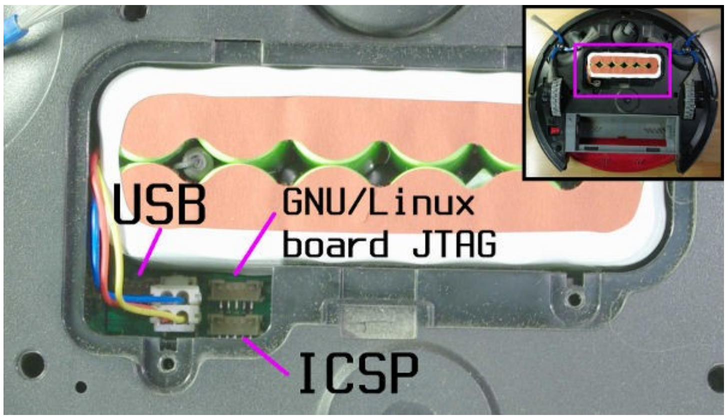


Figure 7. Battery compartment on Samsung Navibot SR8730.

ICSP port is falsely marked as JTAG on the board, however it is actually connected to ICSP pins on the microcontroller. On this microcontroller, there is a separate set of pins for JTAG, so the marking seems to be erroneous. The exact function of the USB port is currently unclear.

In case of 𝕊, there is also a way to read the firmware. This means that the user does not have to give up the chance to revert to original firmware when he wants to try out an alternative. On 𝕍, there is no such possibility, the user can only write the firmware. The only way to keep the original firmware is to replace the microcontroller with a similar chip. This, however, is adverse to project goals and therefore will not be described in details or even endorsed by this project.

One of the important features of this project is that there should be an easy way to revert to the original firmware if something goes wrong. This means that the additional SBC (this is described in more detail in chapter 3) should be able to read and write flash memory of the microcontroller without requiring the user to connect external programmer. This is possible in case of 𝕊, but it is complicated by the fact that not many free software projects can emulate PIC programmer via bit banging, so it was not implemented (usually the flashing process is accomplished by connecting a special programmer, but in this project it is much more convenient to reflash the microcontroller without user intervention and without using any dedicated hardware). The only project that was found is *pic32prog*. Pic32prog is a flash programming utility for pic32 microcontrollers, it supports many different adapters (programmers). The bit banging implementation is present in the source tree [19], but the documentation states that it is incomplete and therefore it is unknown whether any code from that project will help or if something else should be attempted.

Unlike other microcontroller manufacturers, the manufacturer of STC microcontrollers does not offer a way of reading the flash memory. There is a chance that it will be possible to extract the firmware by using non-official methods, however, these are usually not simple, and therefore do not bring any useful result for this project. Legally only the copyright holders of the firmware are allowed to distribute it, which means that every user has to read the firmware himself. In case of $\mathbb{S}$, it can be automated so that the user is not required to take any actions, but in case of $\mathbb{V}$ that would mean more sophisticated hardware hacks, if it is possible at all.

3 Approach

Robotic vacuum cleaners usually have many peripherals, which means that the microcontroller does not have sufficient resources to do mapping or planning. Therefore, it is reasonable to use a separate unit for all of the high-level processing. I decided to install a SBC into each robotic vacuum cleaner. Two different boards were chosen:

- Olimex A13-SOM [20]
- Rasperry Pi Zero [21]

Which board to use is a completely irrelevant question. As long as it runs GNU/Linux and has a UART port, the board is considered suitable for this project. The only important criteria is dimensions of the board, simply because the board has to be physically small enough in order to fit into the vacuum cleaner. Computer boards that are too big are not very practical [22].

- On $\mathbb{V}$, there is a space of about 14.7 x 5.4 x 8 cm. This is enough to hold both a SBC and an adapter board (adapter board is described later in this chapter).
- On $\mathbb{S}$, there is no easily accessible open space. There is about 4.4 x 3.0 x 1.9 cm in the battery compartment, but at the time of writing this is considered too small to hold any SBC. There is enough space right above the motherboard, but special care has to be taken in order to prevent accidental short circuits or overheating issues.

An example setup can be seen in Figure 8.

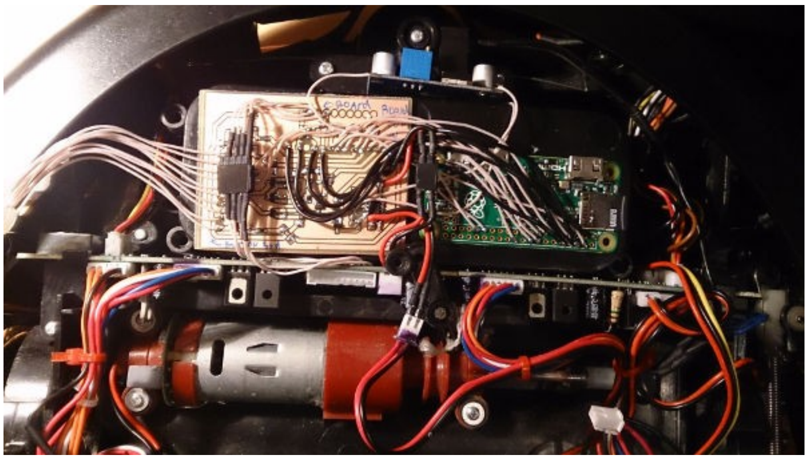


Figure 8. Example setup on Vileda Relax.

In Figure 8 one can see Raspberry Pi Zero on the right (green board) and the adapter board on the left. Original motherboard can be seen mounted vertically below these boards. There is only one extra available pin on 𝕍 (at least two pins are required for UART), so we have to use existing ports that are originally used for a different purpose. On 𝕍, there is a 10 pin port which is connected to the button board (button board is located on top of the robot). Pins on that connector are used for controlling LEDs and getting the state of the buttons. The idea is to connect a SBC to this port (therefore getting MCU↔SBC communication to work), and connect the button board directly to SBC (therefore getting back the ability to control LEDs and detect button presses). This trick is likely to become a common practice with other robotic vacuum cleaners.

An adapter board is required because of the need to perform logic level conversion (3.3V↔5V) and power management (i.e. turning off the motherboard when it is not needed). Also, it helps to keep the setup being indifferent to which SBC is used (the idea is that it should be possible to use any SBC, not just Raspberry Pi Zero).

The schematic and PCB layout for the adapter board was done in KiCad [23]. KiCad is a free software suite for electronic design automation (EDA). Source files of the adapter board (e.g. schematic) are provided in Appendix 2.

3.1 Connection

A SBC may be connected to the MCU with any interface that is available, as long as it is capable of transferring data in half-duplex or full-duplex mode. On both $\mathbb{S}$ and $\mathbb{V}$ it happened to be UART. Besides UART there are two other viable options: SPI and I²C. Other interfaces are rarely supported in microcontrollers and SBCs, therefore, they will not be discussed here.

The following list describes these options in more detail. However, only the most common configurations are discussed, as more rare variants (like three-wire SPI) are rarely supported by hardware. In the context of this work, some details are also different (e.g. CLK line on SPI is not used).

- SPI – full-duplex, 3 lines required (besides ground), no additional overhead. This might look like the best option, but on many SBCs there are driver problems with SPI. On both chosen boards there is no support for SPI slave mode, which is rather limiting but can be solved by using a SBC as a master. This is slightly less convenient because on some devices we would have to implement SPI by bit banging, in which case controlling the clock line makes implementation much easier. Besides that, on Olimex A13-SOM, current SPI kernel module is blocking, meaning that CPU cannot work on other tasks during the data transmission. First attempt to get communication working on $\mathbb{S}$ was made with SPI, but it turned out to be impractical due to the lack of non-blocking SPI kernel module.
- I²C – may be a good option in some rare cases given that only 2 lines are required (besides ground). However, unnecessary overhead and the fact that it is half-duplex turns it into an unwanted option.
- UART – full-duplex, two lines required (besides ground), very widespread hardware support. There are some minor inconveniences, but overall it is a safe option (meaning that there is a very low chance of stumbling upon issues due to poor drivers or the lack of hardware support).

In other words, there is no definite choice. Each case should be examined independently and all possibilities should be considered.

On 𝕊, the connection is based upon the debug port (ICSP) which has two pins that can be configured as GPIO. The same pins are used for flashing and debugging, which is a good prerequisite for automatic firmware updates. However, there is no hardware UART port on these pins, so the interface is implemented via bit banging. This is much less efficient than having a hardware-implemented UART, so with the current implementation the baud rate cannot be higher than 230400 bps. Baud rate of 230400 is high enough and so far satisfies the requirements of the project. Current implementation is interrupt-based, meaning that the microcontroller is able to perform other processing during data transmission. However, just because the device is able to accept data at such rate does not mean that it will be capable of processing it as fast. This means that high baud rates may be used to decrease the latency, but not to transmit large amounts of data, in which case a software bandwidth limit should be used.

On 𝕍, the interface is implemented in hardware, so there is no need to deal with bit banging problems. It is also possible to reflash the microcontroller by using exactly the same pins (moreover, the flashing process is based on UART, so no need to reconfigure the pins on SBC side either), which means that the installation process is much easier. However, due to limitations of the hardware, the baud rate is limited to 38400 bps. There are ways to get other baud rates, but it is unlikely that a baud rate higher than 38400 bps will bring any profit. The microcontroller on 𝕍 is so slow that receiving the data faster than that is not very viable anyway.

3.2 Separation of Concerns

Since the hardware is separated into a SBC and a MCU, there has to be at least two distinct parts of the software.

In reality, there are even more parts, because it is very desirable to have a modular design. These parts are developed as separate projects, therefore there is a need for a name for every part. The naming convention is to call crucial parts analogously to nervous system of an animal, in a hope that it will be easier for new developers to understand the purpose of each component and get a clue of how things are working together. More specifically:

- The most basic piece of the project is a part that runs on the MCU, it is usually referred to as just "firmware". For this part, there is a separate project for each supported robotic vacuum cleaner. However, since the goal of the project is to have most of the code running on all supported devices, there is a common library called *Cordlib*.
- Cordlib[1] – implements common functions that are present on more than one device. The chosen name is not entirely correct, since besides acting as a library it also offers a main loop which then calls various hooks. This makes it more like a framework than a library, but the point remains that it is something that other firmware developers may or may not use during the development. Cordlib manages the data transmission and the protocol, communicating with *Vthal*.
- Vthal[2] – is responsible for relaying data between Cordlib and other components (e.g. web interface). It may also do some primitive processing. Since SLAM and other advanced problems were intentionally left out of the project, current simple implementation of autonomous cleaning is done in Vthal. This part does not care which particular model of the robotic vacuum cleaner it is talking to, as long as the protocols are compatible. This is also true for other connected components.
- Web interface – web-based control panel. Current implementation of the control panel connects to Vthal through the Unix socket, which also allows multiple instances of possibly different control panels. This is a good indication of low coupling.

There are also other parts that may be implemented in the future, for example a firmware updater. These can be connected either as modules (and will be executed together with vthal) or by talking through a unix socket. Another option is to use named pipes, although named pipes are less convenient because they only allow data to be transmitted in one direction. A named pipe works similarly to a regular pipe, except that it is implemented through a file.

[1]Named after the spinal cord

[2]Named after the thalamus

Figure 9 shows a simplified diagram of this design. Ellipsis indicates where other components may be implemented (e.g. a firmware updater).

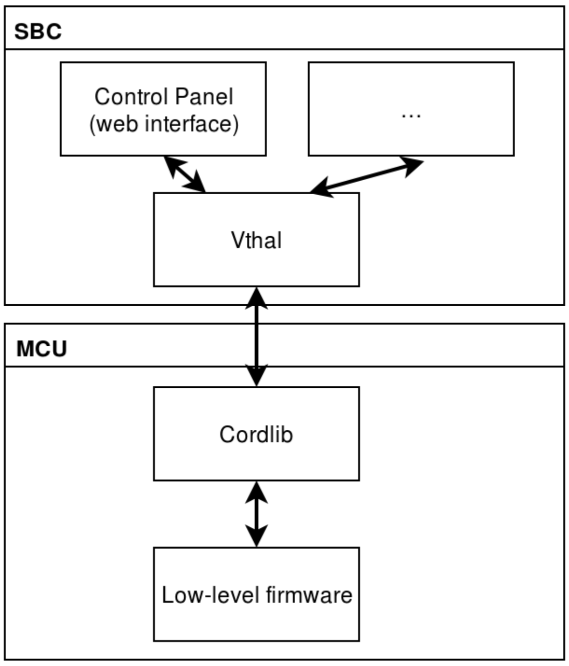


Figure 9. Simplified diagram of various parts of the project.

4 Implementation

This chapter focuses on the most important design decisions and implementation details.

4.1 Implementation on Samsung Navibot SR8730

Development tools include a Netbeans based development environment called MPLAB X [24], and a compiler that is based on gcc-mips C compiler (rebranded as "XC32 Compiler") [25], both provided by Microchip.

Side note: Since GCC is released under GPL license, everyone has a freedom to modify it, but access to the modified source code should also be provided if the software is being distributed. Microchip decided to take GCC and implement a feature that checks whether the user has bought "Standard" or "PRO" version of it, and if not, optimization levels above 1 are disabled [26]. This, obviously, does not prevent any user from removing these checks and rebuilding it from source [27], bypassing the attempt to make the user pay for the software (gcc itself is both free as in "freedom" and is distributed free of charge). However, Microchip not only does not provide instructions required to compile the compiler from source (which legally is a gray area), but they also deliberately excluded some of the files, therefore violating the GPL license. In order to not let this go to the court, they provide missing files one by one upon request. In the end, the easiest way to get rid of the check is to modify the binaries directly (which is legal given that you are allowed to do modifications to the software), yet users who are willing to pay are unable to get more recent versions of the compiler simply because it is too hard to go through the process of building it from source. This probably has some positive short-term impact on income for Microchip, but for free software projects it is an outstanding problem (even if the main developer eventually buys the software, most contributors will not, which leads to significantly fewer contributions). In other words, the easiest solution in this situation is to promote "cracking" the software (which, once again, is totally legal in this particular case).

4.2 Language Choice and Tooling

In this section I will explain which tools were chosen and why.

4.2.1 Firmware

The language choice is between C, C++ and Assembly. While C++ seems like a reasonable choice, Microchip distributes the C++ compiler separately, requiring every user to register his MAC address in order to download it. The author considers that as an unacceptable step for contributors, so C was chosen. Assembly is not a valuable option because the resulting firmware is rather complex (due to high amount of various peripherals on the vacuum cleaner), but a small amount of inline assembly in hot paths is acceptable.

Note that this particular choice does not affect the choice for other devices.

4.2.2 Cordlib

Since Cordlib will be used on several platforms simultaneously, there are certain limitations on what C features may be used.

- For 𝕊, C11 is the last C standard supported by the latest XC32 Compiler.
- For 𝕍, SDCC [28] is the only free C compiler available. The latest supported version of the standard is C11, yet there is some minor incompleteness [29].

JSMN [30] is used for JSON parsing. JSMN is a lightweight JSON parser designed for embedded use.

4.2.3 Vthal

There are no specific requirements on language, any language will do. On early stages *Perl 6* was chosen due to the ease of interaction with other languages, but also because I am very familiar with it. The only concern was that *Rakudo* (which is a Perl 6 implementation) was not quite up to the speed of other languages. Since the first release of Perl 6 was in December 2015, there was not enough time for it to get mature in terms of performance. However, the most major problem for this project was the lack of JIT in Rakudo on ARM builds (not yet implemented).

At some point it became apparent that Rakudo does not provide enough performance to handle basic data transmission, so it was decided to switch temporarily to *Perl 5*. The syntax is very similar, so porting Perl 6 code to Perl 5 did not take much effort (comparing to porting from languages that are not in perl family). After the switch, the CPU consumption dropped from near 100% during active data transmission to about 5%.

This was expected, and the option to switch to Perl 5 for performance reasons has been a backup plan for exactly this case. However, Perl 6 is still on the map for the distant future of the project.

4.2.4 Web Interface

In order to make it easier for further contributors to understand the code base, it was decided that the less the number languages is, the better. Therefore, the control panel is also written in Perl 5.

4.3 Implementation Details

Next sections will describe in details how to get the most basic system running.

The process starts from getting all parts of the system up and running on $\mathbb{S}$ (porting to $\mathbb{V}$ will be described separately).

4.3.1 UART Bit Banging

On $\mathbb{S}$, two hardware UART interfaces are already used for communication with the built-in SBC and the gyroscope module. Although within this project we are not going to use the built-in SBC (we install our own), there is no easy way to use this UART interface. It is not wired to any external port, and the given microcontroller does not support interface remapping.

In order to get the UART working, we either have to make some hardware modifications, or implement it in software on any of the available pins.

There are several reasons not to introduce custom hardware modifications:

- The original firmware will not work on different hardware. That is, if at some point the user decides to go back to original firmware, he will have to revert the hardware back to its original state.
- Most users will not be willing to do any modifications. Without hardware modifications the installation process is mostly plug'n'play, but it is still rather complicated and only advanced users will be capable of doing that. The need to resolder something will raise the bar too high even for most of advanced users.

Therefore, the UART interface is implemented with a technique usually referred to as "bit banging", which means that instead of relying on hardware implementation of some interface, we will be driving the pins high or low manually.

This is also beneficial in some way. For example, I implemented an automatic baud rate detection, which is not possible with built-in hardware UART interface on the given pic32 microcontroller.

In order for the automatic baud rate detection to work, the other side (in this case it is SBC) has to transmit a byte that has at least one short pulse (e.g. 101 or 010 in binary). The length of this shortest pulse is measured, and one of the timers is configured to fire with exactly this period. This results in consistent transmission performance (interrupt priority of this timer is set above other interrupts). It is recommended to send ASCII character 'U' to the microcontroller before starting any data transmission. When 'U' is sent, it results in (0)10101010(1) being transmitted (start bit and stop bit in parentheses, no parity is used). This gives the other side enough high-low transitions to measure the length of a single bit. Also, if for some reason connection was suddenly restarted, it should expect exactly 10 edges so that there is a bigger chance of getting the shortest high-low-high or low-high-low pulse.

Data reception is done by using *Change Notification* feature of the given pic32 microcontroller. Change notification is similar to external interrupts, but completely different. Instead of having a dedicated interrupt on [rising edge]/[falling edge]/[both]

on particular pin, change notification interrupt fires up on any change on any of the configured pins. This means that in order to find out why the interrupt was fired, developer has to check the state of all configured pins in the ISR, and all extra processing of such events will have to be done in the ISR itself. Since it is unacceptable to miss such events on Rx pin, it is unlikely that change notification feature will be used for anything besides UART Rx pin. Because of that, some things that could have been done with interrupts will have to be continuously polled in the main loop. This is slightly unfortunate, but it is reasonable tradeoff.

4.3.2 Protocol

Unfortunately there is no convention that would describe a protocol that could be used in a situation like that. In other hardware projects, the communication is considered stable (most of the time it is used for intra-board communication), so there is no need to implement complicated error checking techniques. In this project the connection is between two different boards, and the wiring may be potentially faulty (we cannot guarantee that the user did everything right), so the protocol must have some additional properties. For example, due to stack overruns, hardware resets or other unexpected behavior, one of the sides may start listening the line in the middle of the transmission. In such a case, the receiving part either has to ignore half of the received packet, or ask to transmit it again. Another important problem is that the packets may be corrupted during transmission, in which case the error has to be detected. After all, it is controlling real hardware that is moving. A command that drives the motor incorrectly may end in catastrophic failure for the robot or its surroundings.

The protocol must also be easily extendable, because at any point the version of the flashed firmware and the version of the software running on SBC may differ significantly. In this case, both the board and the microcontroller may be sending packets that are not recognized by the other side, or these packets may contain parameters that are not understood. This means that the structure of the packet has to be understandable even when the contents are not.

Another concern, that is often of a bigger importance, is that the user is also expected to debug or to improve the software. Working with a binary protocol is sometimes very complicated.

Therefore, I decided that it would be a plain text protocol, in which JSON objects will be sent as packets. Plain text protocols have several drawbacks, but overall this solution works very well.

Plain text protocols are less efficient, both in terms of the overhead data that has to be sent and in terms of processing that is required on both sides. However, according to preliminary calculations and tests, the amount of data that has to be transmitted is not high and the processing overhead is not noticeable. This is a tradeoff between performance and ease of debugging and development.

However, it also makes sense to think through the case if at some point the throughput becomes the bottleneck of the system. In this case, the packets that require high throughput may get binary equivalents (that is, it would be possible to send the packet in its JSON form, or to send it as an optimized binary version that is hardcoded on both sides). It is important to at least start the communication with the plaintext protocol (which is well understood with software and hardware of any versions), and then automatically “upgrade” the communication if both sides support it. Such need actually exists on $\mathbb{V}$, where the amount of available memory is small and the MCU itself does not provide enough performance.

Each packet is separated by 0x1F (*INFORMATION SEPARATOR ONE;* control code defined in ASCII as *Unit Separator*) character, which means that both sides can simply split the stream by 0x1F characters to get actual packets. These packets are then processed by cordlib or vthal, depending on which way the packet has been sent.

4.3.3 Cordlib Callbacks and Tests

Cordlib may be compiled as a library, which is used for automatic tests. In some cases instead of just providing a set of functions it also has to call some functions in the firmware. For example, when cordlib receives a command from the SBC that controls one of the motors (e.g. wheels), it should call a function in the firmware itself that is responsible for that. This is implemented via callbacks.

Callback is a well-known technique in C, so this is not going to be covered here. However, since the tests for Cordlib are written in Perl 6, it is worth looking at an example of running C code from Perl 6.

Let’s say we have a function *get_time* that returns a *double*. Below we will see an example that demonstrates how to use a callback with this function from Perl 6.

As a side note, notice that “double” data type is used for time values. As a convention in cordlib, all non-discrete values (like time, which is continuous) should also have a proper type that supports continuous data to the maximum extent. This is done to prevent the situation when more precision is required, yet the given function can only supply numbers up to some arbitrary precision.

However, a lot of commonly used microcontrollers do not have a FPU, which means that all operations with floating point numbers will be much slower than similar operations with integers. This leads to a few things to keep in mind:

- Such functions should only be used in non-hot high-level code. The code is considered to be “hot” if it is executed very frequently. For example, if it is inside a loop, or if it is part of an ISR.
- It is acceptable to pass such data around, but if it has to be processed, then the firmware may convert it into another format that is faster on the given system.
- Although the floating point number may be very precise, it does not mean that the firmware is obligated to use this to its full potential. The firmware may decide to truncate it due to performance reasons or due to the hardware (e.g. a counter) not being able to produce more precise values.
- If the function is commonly used in hot code, or if original value is commonly discrete, then cordlib must provide an optimized version. For example, besides **get_time** there is also **get_time_ms.**

Function **get_time** will use a callback if it is set. This callback should be set in firmware by calling **set_callback_get_time**. In the following examples, the test suite will be acting as a firmware. Example of such declaration can be seen in Figure 10.

```
void set_callback_get_time(double (*callback)(void));
```

Figure 10. Example declaration of a function that sets a callback.

In Perl 6 we have to declare it first. Such declaration is shown in Figure 11.

```
sub set_callback_get_time(&callback ( --> num64)) is native('cordlib') {…}
```

Figure 11. Declaration of a function in Perl 6 that sets a callback.

After that you can call this function just like a regular Perl 6 subroutine.

Full example[1] of using callbacks is shown on Figure 12.

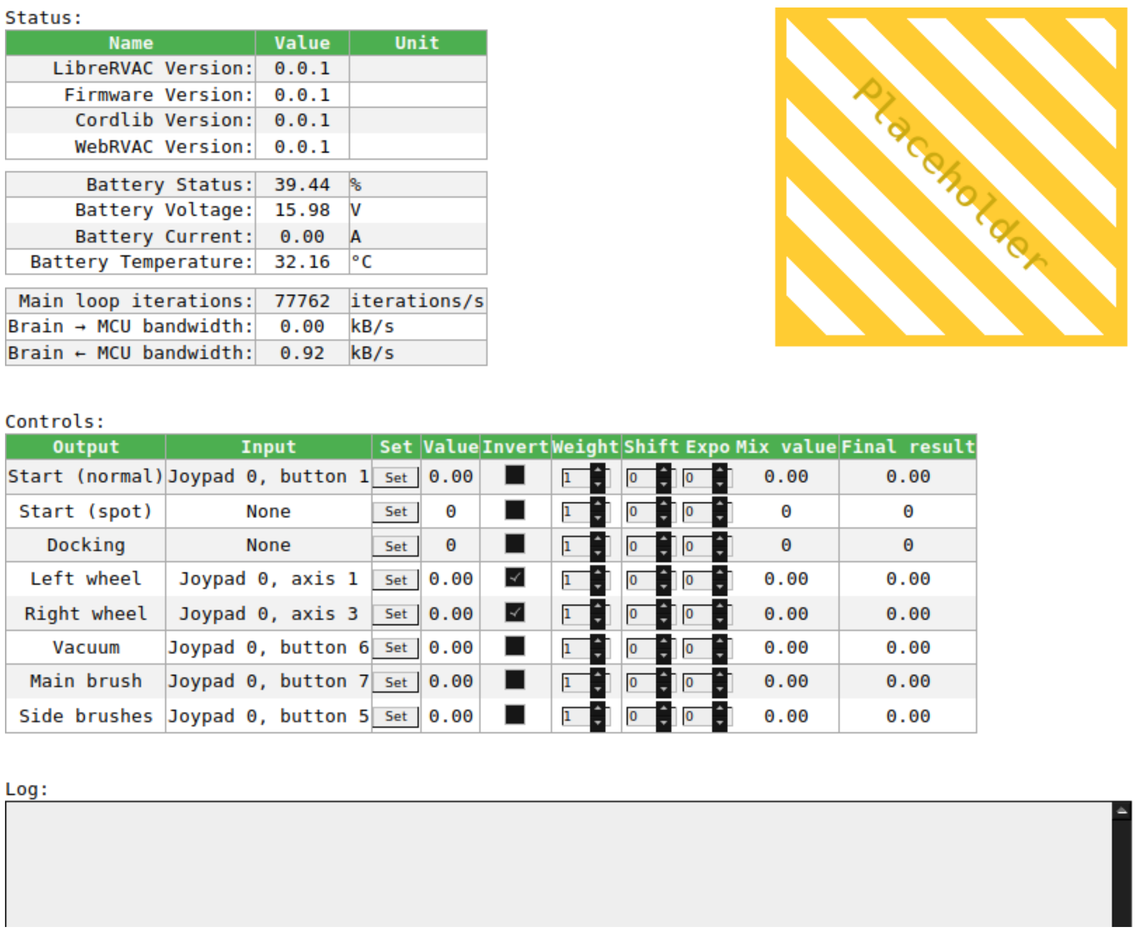
```
sub set_callback_get_time(&callback ( --> num64)) is native('cordlib') {…}
sub hw_get_time() returns num64                   is native('cordlib') {…}
set_callback_get_time sub { 4567.8.Num };
is-approx hw_get_time(), 4567.8.Num, 'get_time() callback works';
```

Figure 12. Full example of using callbacks from Perl 6 code.

4.3.4 Web Interface

In Figure 13, a screenshot of the current web interface can be seen.

[1]In Perl 6, curly quotes (' ') can be used similarly to straight quotation marks (' '). Provided example was not mangled by the text processor and is valid in Perl 6 as is.



Status:

Name	Value	Unit
LibreRVAC Version:	0.0.1	
Firmware Version:	0.0.1	
Cordlib Version:	0.0.1	
WebRVAC Version:	0.0.1	

Battery Status:	39.44	%
Battery Voltage:	15.98	V
Battery Current:	0.00	A
Battery Temperature:	32.16	°C

Main loop iterations:	77762	iterations/s
Brain → MCU bandwidth:	0.00	kB/s
Brain ← MCU bandwidth:	0.92	kB/s

Controls:

Output	Input	Set	Value	Invert	Weight	Shift	Expo	Mix value	Final result
Start (normal)	Joypad 0, button 1	Set	0.00	☐	1	0	0	0.00	0.00
Start (spot)	None	Set	0	☐	1	0	0	0	0
Docking	None	Set	0	☐	1	0	0	0	0
Left wheel	Joypad 0, axis 1	Set	0.00	☑	1	0	0	0.00	0.00
Right wheel	Joypad 0, axis 3	Set	0.00	☑	1	0	0	0.00	0.00
Vacuum	Joypad 0, button 6	Set	0.00	☐	1	0	0	0.00	0.00
Main brush	Joypad 0, button 7	Set	0.00	☐	1	0	0	0.00	0.00
Side brushes	Joypad 0, button 5	Set	0.00	☐	1	0	0	0.00	0.00

Log:

Figure 13. Web interface.

The control panel is a simple single-page application that has several sections:

- Device information – this section lists all available information (version numbers, battery status, etc.).
- Control mixes – this section is a simplified version of the mixes that are usually found on remote control transmitters. Each row in the table is a channel, and the user may attach any control (e.g. joystick button or analog stick) to control it. This is slightly more advanced than most users expect, but as long as good default values are chosen there is no problem.
- Log – this is a textual log that is intended to show errors either from the device itself or from other parts of the software.
- Map – at the moment of writing this is a placeholder that does nothing. The reason why it was not implemented is discussed in section 5.3.

4.4 Implementation on Vileda Relax

Implementation on 𝕍 is a proof of concept. One of the goals of the project is that cordlib should be robust enough to be used on different MCUs. This may be verified by adding support (even a very basic one) for another robotic vacuum cleaner.

Before getting into details of the firmware on 𝕍, it is important to look at the summary of others parts of the software. At this point, these parts are given:

- Cordlib – a library for the firmware that takes care of the main loop, communication, protocol, and some other basics.
- Vthal – software that runs on a SBC and "talks" to the microcontroller and other components like a control panel.
- Control panel – software that runs on GNU/Linux and is connected to Vthal. It displays device info and allows the user to control the device manually.
- Also, we have a preliminary pinout table for 𝕍.

Looking at this, it is easy to realize that the amount of work required to get the firmware working on 𝕍 is not high (assuming that other parts are implemented properly).

This is a good sign, because in the long run the project needs to support as many devices as possible. It will be easier to achieve such broad device support if adding support for more devices is easy.

There are, however, some challenges related to the MCU. Most of these challenges come from the fact that it only has 1280 bytes of RAM, which is very limiting.

- There are different types of memory on 8051 architecture. The memory model that fits this project is "large", which uses 1024 bytes of memory as RAM and 256 bytes for other purposes (e.g. stack). Because of such a small stack, some code paths require minor tweaks in order to fit into the memory.
- Non-blocking IO is of course preferred, but it means that there must be two buffers (for input and output). The buffers and the overhead can easily sum up into more than half of the available memory. Either these buffers have to be

really small, or some tricks have to be implemented in order to save some memory (e.g. parsing the packet on the fly).

- Cordlib is using C11 features freely (which is a design decision), but some of these occurrences have to be changed with *#ifdef*'s in case if the code is compiled with SDCC. This does not introduce any significant problems to the development process, but it makes the implementation on 𝕍 slightly less robust.
- printf and its cousins are extremely slow. This makes it unsuitable for the implementation of a plaintext protocol. On 𝕊, the main loop is typically run more than 60 000 times every second (which is more than enough), while on 𝕍 the same code even without some "heavy" functions results in the main loop frequency of 5 Hz.

These issues raise a question of whether support for such low-end microcontrollers should be added at all, given how many problems such decision may introduce. The goal of confirming the portability of Cordlib was achieved, but further research is required for a fully functional implementation on 𝕍.

5 Possible Improvements and Project Future

This chapter describes long-term plans for the project.

5.1 Localization and Mapping

Localization and mapping with robotic vacuum cleaners is a good topic by itself, and it was deliberately excluded from the early steps of this project.

Most robotic vacuum cleaners do not have enough sensors to do proper mapping (no lidars or front-facing cameras), therefore localization and mapping is a very complex problem. In theory, if it is possible to get current position precisely, then the robot can gather information about the world around it by bumping into things. This is not as convenient as in solutions with advanced sensors, but since we are operating on a given hardware, there is not much we can do about it (possible solutions are discussed later in this chapter). However, given enough time, the robot will actually map the whole space by “touching” things around it, and the acquired data can be saved and then reused on the next run.

However, even getting the current position without external sources of information is a very complicated task.

Some expensive robotic vacuum cleaners often do not have good hardware, but compensate for that with interesting software tricks [31]. When hitting something a couple of times, the robot may decide that it is, for example, a wall, and according to that start moving along the wall, therefore cleaning an area alongside of it. With pure random movement some areas near to the wall may be left uncleaned.

The important thing to note here is that in such cases the robot does not build a map of the whole space. The algorithm is based on some heuristics that only use the latest data, for example the data acquired in the last 10 seconds.

However, there are other ways to tackle this problem. Since the user has to install additional hardware anyway, it is also acceptable to require some additional radio modules. That is, the solution to this problem may be based on radio beacons that are placed inside the rooms.

To sum it up, there are several possible solutions to the problem and this has to be studied. At the moment of writing, the software implements the simplest algorithm that uses no information about the surroundings. This algorithm is basically a working placeholder, which in the future will be replaced with better solutions.

5.2 ROS

ROS (Robot Operating System) [32] is a set of software libraries and tools for robot software development.

For the early stages of the project it was decided not to use ROS. However, many features that are already implemented in ROS will be required in the future.

As with anything, there are benefits and drawbacks. While ROS has a benefit of having a huge user base and a strong core development team, here are some reasons why ROS was not used in this thesis:

- The focus was mostly on a low-level implementation where ROS benefits do not even apply. The need for ROS will emerge over time, but at this point it is mostly irrelevant.
- Eventually ROS 2.0 [33] will replace ROS 1.x. For a long-term project it is much better to start with ROS 2.0 directly, but it is not mature enough. In other words, delaying the decision to use ROS is a good idea at this point.
- It will not be too hard to add support for ROS later.

5.3 Map in Control Panel

The map in the web interface has to display rather high amounts of visual data, possibly in 3D. This fact raises the bar for the implementation. That is, the implementation has to be based on WebGL [34], which requires a bit more thought put into it, rather than just drawing a couple of lines on the screen. WebGL is a low-level 3D graphics API based on OpenGL ES 2.0.

The map was not implemented in this project, but it has to be done.

5.4 Free Peripheral Library

On 𝕊, I decided to use peripheral library provided by Microchip to speed up the development process (instead of creating my own library). However, the licensing terms of this library are not very convenient for a free software project, even though the source code is available.

For some other microcontrollers, there is a project called libopencm3. Libopencm3 [35] is a free software library with sound licensing terms (GPLv3 or LGPLv3). However, no such project exists for pic32 microcontrollers yet.

5.5 Full Support of Hardware Features

As will be shown later in Table 2, there are some missing features. Mostly there are no difficulties, given that the devices are well studied and most of these features were already tested with some basic code. However, it takes time to implement these features correctly.

5.6 Perl 6

As it was described earlier, Perl 6 has a couple of useful properties that would be very handy in this project. However, during the implementation I had to switch to Perl 5 due to performance issues.

Since it is planned to switch back to Perl 6 at some point, instead of just waiting for it to become fast, it is a good idea to identify which parts are acting as a bottleneck and fix these myself. This, however, requires a functional JIT on ARM architecture in Rakudo.

5.7 Security

Since the control panel works over Wi-Fi, all security concerns related to wireless networks and local area networks apply. Basically, your robotic vacuum cleaner is as safe as your local area network is. In reality, most users do not pay enough attention to the security of their networks, so this becomes a real issue for this project.

The solution to this problem is unclear. In theory, any communication should be encrypted, yet in this case you cannot get HTTPS to work (no domain names on local area network means that even if you attempt to use a certificate, there is nothing to confirm its validity). Other possible solutions to this problem have different drawbacks. For example, requiring the user to generate and install certificates (both on robotic vacuum cleaners and on used computers) is perhaps the only secure solution, yet it complicates the installation process. This issue should be researched.

5.8 Communication Between Multiple Robots

Since all robots have different hardware (sensor types, dimensions, processing power, etc.), it makes sense for robots to cooperate. This potential feature may significantly improve the user experience in cases when one person owns multiple robotic vacuum cleaners. Another possible use case includes corporate environments. Further research is required, both on how robots should communicate with each other and what strategies may be used in such cases.

5.9 Broader Device Support

Even though only two robotic vacuum cleaners were discussed in this thesis, the actual number of supported devices is probably higher.

The motherboard on Samsung Navibot SR8730 suggests that Samsung is using exactly the same PCB for many devices in the series. For example, the board includes LEDs for a 7-segment display, which is not present on this particular device. Therefore, there is a high chance that the same firmware will work on more than one robotic cleaner, while on others some minor modifications will be required. It is much easier to add such per-device modifications than to implement a new firmware from scratch.

Vileda Relax Plus and *Vileda M-488A* are very similar to Vileda Relax, most likely these devices are just upgraded versions of it. This means that the firmware may be compatible too, at least partially. There is also *Eurolab Vacuum Cleaner* and *Hoover 5240*. It seems like some parts may be compatible across all of these devices, but it is currently unknown if the motherboard is same as well. It is also important to note that these devices are mostly clones of one of the earliest *iRobot Roomba* robotic vacuum cleaners.

However, there are still many devices that are not supported at all. The documentation for any device (supported or not) is kept on the wiki [36]. It is expected that the same wiki will keep documentation for robotic lawnmowers, robotic window cleaners, robotic snowblowers and other robots that are very similar to robotic vacuum cleaners in their nature.

Here is a simplified list of step-by-step instructions for adding support for a new device:

1. Check if any of the similar devices (e.g. devices in the same series) is already supported. If yes, try to base your efforts on existing code and documentation.
2. Reverse engineer the device. As a result, you should have a pinout table for the microcontroller.
3. Write a “hello world” firmware that sends one character over UART or any other interface. Check if it is working.
4. Add cordlib to your project. Use functions provided in *cord_connection.h* file to handle communication. Also, delegate your main loop to cordlib by using *cord_event_init* function.
5. At this point, the firmware will be transmitting various basic information over the chosen interface. Connect a SBC to your microcontroller and run vthal, the basics should be working already.
6. Keep adding remaining features one by one (e.g. motors, sensors, etc.). From this point the development process is rather straightforward.

6 Comparison With the Original Firmware

In this chapter, the results of this thesis will be compared to the original firmware.

6.1 Samsung Navibot SR8730

Table 2 shows the current status of the firmware.

Table 2. Table of low-level features on Samsung Navibot SR8730.

Feature	Legacy	LibreRVAC
Charging	yes	yes
Wheels	yes	yes
Main Brush	yes	yes
Side Brushes	yes	yes
Bumper	yes	yes
Beeper	yes	yes
LEDs	yes	yes
User Buttons	yes	yes
Wheel Buttons	yes	yes
Sensors	yes	planned
Cliff Sensors	yes	planned
Gyroscope	yes	planned
Camera	unknown	no

Most hardware features are supported. Other remaining ones can be implemented in the future. These are already documented and there are no known reasons that could prevent implementation of these features with the current software.

The only problematic feature is a built-in top camera. It is connected directly to the built-in SBC, but the documentation for that board is lacking. At the time of writing it is also unclear how to connect to it.

Also, it is unknown if the original firmware is using the camera or if it was added just as marketing tool. The robot with original firmware seems to behave exactly the same even when the camera is intentionally covered, which leads to conclusion that if it plays any role at all, it is pretty much irrelevant.

The usefulness of a camera that is pointing directly up is also questionable. The concept of CV-SLAM is not new [37], yet most robotic vacuum cleaner manufacturers who use CV-SLAM are still not relying on it as on the only source. For example, *LG Hom-Bot* has an optical flow sensor mounted on the bottom [38], and even 𝕊 itself was proven to work without the camera.

Samsung sends out sources of modified free software upon request. I did such request, and in return received a 222MB archive containing Linux kernel and some other patches [39]. They do not provide the source code of their own software, so it is hard to know if there is any image processing involved at all.

In other words, except for the camera, most hardware features are supported already and the remaining ones have no obstacles to the implementation. The camera, however, requires further research because not only there is little known about the built-in SBC, but also it is unclear if images should be retranslated to the main SBC through the microcontroller (or processed in place) and if so, how exactly this should be implemented without overloading the communication channel.

Table 3 gives an overview of the most basic high-level features.

Table 3. Table of high-level features on Samsung Navibot SR8730.

Feature	Legacy	LibreRVAC
Web interface	no	yes
Scheduled start	no	yes, but no user interface yet
Wall following	no[1]	planned
Curved path	only when docking	planned
Room mapping	probably[2]	planned

6.2 Vileda Relax

As mentioned in section 4.4, the main purpose of the implementation on $\mathbb{V}$ is to confirm the portability of cordlib and other components. At this stage, it is unreasonable to make a comparison.

[1]In extremely small areas (3x1 meters) the robot may eventually start cleaning around the corners. This behavior has only been observed once.

[2]In legacy firmware the robot is moving in parallel lines, but collected info suggests that it is unable to build a map.

7 Summary

The goal of this project was to design and develop universal firmware for robotic vacuum cleaners. As a result, the firmware and other parts of the project were written, which resulted in a functional solution that runs on rather diverse platforms.

All of the groundwork is done, yet some details are missing. These details will be improved in the future.

The resulting combination of software does supersede original firmware in some cases, yet in other cases feature parity is not reached yet. These problematic cases were discussed and evaluated in terms of feasibility.

Most importantly, the groundwork for all parts of the solution was done and different parts can now be developed independently as a free / libre / open-source project (possibly by other developers).

References

[1] GNU General Public License version 3. [WWW] https://www.gnu.org/licenses/gpl.txt (2016-05-16)

[2] GNU Affero General Public License version 3. [WWW] http://www.gnu.org/licenses/agpl.txt (2016-05-16)

[3] Creative Commons Attribution 4.0 International License. [WWW] https://creativecommons.org/licenses/by/4.0/legalcode (2016-05-16)

[4] Empty download page. [WWW] https://www.neatorobotics.com/support/software-updates/ (2016-05-16)

[5] Semi-official response about discontinued firmware updates. [WWW] https://en.wikipedia.org/wiki/Neato_Robotics#Firmware (2016-05-16)

[6] OpenTx. [WWW] http://www.open-tx.org/ (2016-05-16)

[7] OpenWrt. [WWW] https://openwrt.org/ (2016-05-16)

[8] Libreboot. [WWW] https://libreboot.org/ (2016-05-16)

[9] Sigrok. [WWW] https://sigrok.org/ (2016-05-16)

[10] Abandoned attempt to build open-source vacuum cleaner. [WWW] https://www.youmagine.com/design_ideas/open-source-autonomous-vacuum-cleaner (2016-05-16)

[11] Pinout of one of the oldest Roomba vacuum cleaners. [WWW] http://blog.perquin.com/blog/roomba-uc/ (2016-05-16)

[12] Attempt to gather a team for working on a open-source robotic vacuum cleaner. [WWW] http://www.robotreviews.com/chat/viewtopic.php?f=4&t=14653 (2016-05-16)

[13] Custom web interface for LG HomBot robotic vacuum cleaner. [WWW] http://www.robvanhamersveld.nl/2015/03/13/lg-hombot-3-wifi-mod/ (2016-05-16)

[14] Community-based documentation for Neato XV-11 robotic vacuum cleaner. [WWW] http://xv11hacking.wikispaces.com/ (2016-05-16)

[15] Reverse engineering of Electrolux Trilobite robotic vacuum cleaner. [WWW] http://www.robotreviews.com/chat/viewtopic.php?f=4&t=13895&start=20 (2016-05-16)

[16] Datasheet for PIC32MX3XX/4XX family. [WWW] http://ww1.microchip.com/downloads/en/DeviceDoc/61143H.pdf (2016-05-16)

[17] Datasheet for STC90C58RD+ . [WWW] www.stcmcu.com/datasheet/stc/STC-AD-PDF/STC90C58RD+-english.pdf (2016-05-16)

[18] Project wiki page about Samsung Navibot SR8730. [WWW] https://librervac.org/sr8730 (2016-05-16)

[19] Source tree for ICSP bit banging with Arduino. [WWW] https://github.com/sergev/pic32prog/tree/master/bitbang (2016-05-16)

[20] Product page for Olimex A13-SOM. [WWW] https://www.olimex.com/Products/SOM/A13/A13-SOM-512/ (2016-05-16)

[21] Product page for Raspberry Pi Zero. [WWW] https://www.raspberrypi.org/products/pi-zero/ (2016-05-16)

[22] Webplayer-Roomba Integration. [WWW] https://web.archive.org/web/20060103093845/http://heathkit.mondrary.com/virgin (2016-05-16)

[23] KiCad website. [WWW] http://kicad-pcb.org/ (2016-05-16)

[24] Overview of MPLAB X development environment. [WWW] http://www.microchip.com/mplab/mplab-x-ide (2016-05-16)

[25] XC32 Compiler user guide. [WWW] http://www.microchip.com/mymicrochip/filehandler.aspx?ddocname=en557665 (2016-05-16)

[26] Difference between XC Compiler versions. [WWW] http://www.microchip.com/mplab/compilers (2016-05-16)

[27] Turning on optimizations in Microchip's XC32 compiler. [WWW] http://www.jubatian.com/articles/turning-on-optimizations-in-microchips-xc32/ (2016-05-16)

[28] SDCC - Small Device C Compiler. [WWW] http://sdcc.sourceforge.net/ (2016-05-16)

[29] Standard compliance of SDCC. [WWW] http://sdcc.sourceforge.net/mediawiki/index.php/Standard_compliance (2016-05-16)

[30] JSMN website. [WWW] http://zserge.com/jsmn.html (2016-05-16)

[31] Facts about iAdapt technology. [WWW] https://web.archive.org/web/20130511232754/http://www.irobot.com/EngineeringAwesome/images/iAdapt%20Fast%20Facts.pdf (2016-05-16)

[32] The Robot Operating System (ROS). [WWW] http://www.ros.org/ (2016-05-16)

[33] ROS2 Design. [WWW] http://design.ros2.org/articles/why_ros2.html (2016-05-16)

[34] WebGL. [WWW] https://www.khronos.org/webgl/ (2016-05-16)

[35] libopencm3 wiki. [WWW] http://libopencm3.org/wiki/Main_Page (2016-05-16)

[36] List of all documented devices. [WWW] https://librervac.org/Supported_Devices (2016-05-16)

[37] WooYeon Jeong, CV-SLAM: a new ceiling vision-based SLAM technique, 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems, 3195 – 3200, [Online] IEEE Xplore Digital Library (2016-05-16)

[38] Technical characteristics of LG HOM-BOT LrV5900 robot vacuum cleaner. [WWW] http://www.lg.com/us/vacuum-cleaners/lg-LrV5900-robot-vacuum#tech-specs (2016-05-16)

[39] Linux kernel and other sources for Samsung Navibot SR8730. [WWW] http://opensource.samsung.com/reception/receptionSub.do?method=downLoad&attach_id=8697 (2016-05-16)

Appendix 1 – Source Code

Source code for various parts of the project can be found on https://github.com/LibreRVAC.

More specifically:

- https://github.com/LibreRVAC/librervac-sr8730 – Firmware for Samsung Navibot SR8730.
- https://github.com/LibreRVAC/librervac-relax – Firmware for Vileda Relax.
- https://github.com/LibreRVAC/librervac-cordlib – Cordlib.
- https://github.com/LibreRVAC/librervac-vthal – Vthal.
- https://github.com/LibreRVAC/librervac-web – Web interface.

Appendix 2 – Schematic for the Adapter Board

KiCad sources for the adapter board for Vileda Relax can be found on https://github.com/LibreRVAC/librervac-relax-adapter.

Schematic in PDF format can be found on https://github.com/LibreRVAC/librervac-relax-adapter/releases/.

Appendix 3 – Reverse Engineering Struggle

This is an excerpt from the source code.

```
// When adapter is connected, it starts charging (PWM, 160 Hz ≈1.5% duty
// cycle, this results in about 250mA current).
// It is unclear why they do that. May even be a bug. This happens for
// around one minute.
// Then they change PWM frequency to 20kHz, duty cycle ≈3.33% but they are
// not doing that constantly. Instead, they have a 50Hz PWM on top of that.
// That is, it sends 20kHz wave for a while, then does nothing, then repeats
// this pattern with 50Hz frequency.
// They are slowly increasing the duty cycle of this 50Hz wave, which
// results in current going from almost zero to exactly 1A.
// Once charged, it attempts to compensate for leaks. In order
// to do that, every 2 minutes they charge the battery for 20 seconds with
// ≈40mA current.
// This trickle charge strategy is probably flawed. It seems like there is
// about 15mA discharge current when the device is sitting on a dock.
// 45mA charging for 20 seconds does not compensate 15mA discharge for
// 100 seconds, and more so if you add self-discharge.
// Interestingly, if you dock your robot while it is fully charged, then
// the device will start beeping annoyingly (“I am charged, get me out
// of here”). So they are not interested in keeping the battery charged,
// all they wanted to do is make sure that the battery does not go dead
// if you leave your cleaner on a charger for a couple of days.
```