MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

Firmware for RoFICoM

Bachelor's Thesis

PETR BÉM

Brno, Spring 2023

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Firmware for RoFICoM

Bachelor's Thesis

PETR BÉM

Advisor: RNDr. Jan Mrázek

Brno, Spring 2023

FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Petr Bém

Advisor: RNDr. Jan Mrázek

Acknowledgements

I would like to thank the ParaDiSe laboratory for a welcoming place to work, Jan Mrázek for introduction to the RoFI platform, and everybody who supported me during the writing of this thesis.

Abstract

The RoFI platform is an open-software and open-hardware platform for modular robots. The RoFI platform uses the RoFI Communication Mechanism, RoFICoM for short, for connecting and communicating between two RoFI modules.

This thesis presents an implementation of firmware for RoFiCoM. It is built on top an existing prototype implementation and transforms it into a production-ready state. We primarily introduce a board support package to support various hardware revisions and implement missing functionality, such as utilizing all on-board sensors and reworking motor control. Additionally, we enhance the communication protocol with support for these features and extend implementation for the Universal module's RoFICoM driver to support these changes.

Keywords

firmware, embedded, RoFI, robots, c++, lidar

Contents

1 Introduction **3**

2 RoFI **5**
2.1 Universal Module 6
2.1.1 Architecture 6
2.2 RoFI Communication Mechanism 8
2.2.1 Mechanical structure 8
2.2.2 Connector-to-Connector Interface 10
2.2.3 Connector-to-Module Interface 10
2.2.4 Hardware details 11

3 RoFICoM — prototype version and current implementation **13**
3.1 RoFICoM's Firmware 13
3.1.1 Connector-to-Module Protocol 13
3.2 Stm32 Drivers 15
3.3 RoFI HAL 16

4 Distance Measurement — LiDAR **17**
4.1 Communication 18
4.1.1 Inter-Integrated Circuit Protocol 18
4.1.2 Addressing VL53L1X's Registers 19
4.2 VL53L1X Interface 20
4.3 API of the Ultra Lite Driver 21
4.3.1 Error Handling 21
4.3.2 Distance Modes 21
4.3.3 Timing Budget 22
4.3.4 Inter-Measurement Period 23
4.3.5 Result Data 23
4.4 LiDAR Diver 24
4.4.1 Error handling 25
4.4.2 Usage of the driver 26
4.4.3 Autonomous Distance Mode 28

5 Board Support Package **29**
5.1 Gpio::Pin Initialization 30

6 Implementation — Design Decisions **31**
6.1 Changes to the Build System 31
6.2 I2C driver 31
6.2.1 Error handling 32
6.3 Motor driver 33
6.4 Power-sharing 34
6.5 Protocol Implementation in RoFICoM 36

7 RoFI Hardware Abstraction Layer **38**
7.1 Interface 38
7.2 Implementation — esp32 39

8 Conclusion **41**

A Building the Project **43**

Bibliography **44**

Abbreviations

BSP Board Support Package 4, 29, 30, 41

GPIO General-purpose input/output 16, 30

HAL Hardware Abstraction Layer 4, 16, 38, 40, 41

I^{2}C Inter-Integrated Circuit 4, 11, 12, 15, 16, 17, 18, 19, 20, 21, 25, 29, 31, 32, 39, 41

LiDAR Light Detection and Ranging 2, 4, 6, 11, 14, 15, 16, 17, 21, 22, 23, 24, 25, 26, 27, 28, 32, 36, 37, 38, 39, 40, 41

MCU Microcontroller unit 11, 12, 15, 20, 29, 30, 35

RoFI modular Robots developed at Faculty of Informatics iv, v, 2, 3, 4, 5, 6, 13, 16, 20, 25, 38, 40, 41

RoFICoM RoFI Communication Mechanism , i, v, 2, 3, 4, 6, 8, 9, 10, 11, 13, 14, 15, 17, 20, 21, 23, 24, 26, 29, 30, 31, 33, 34, 35, 36, 39, 40, 41, 43

SPI Serial Peripheral Interface 7, 10, 11, 12, 16, 28, 36

UART Universal asynchronous receiver/transmitter 6, 7, 10, 11, 12, 16

ULD ultra lite driver 20, 21, 22, 23, 28, 32

Glossary

RoFIbot is a several interconnected RoFI modules. 5, 6

Universal module is a versatile, already-built instance of the RoFI module. 3, 5, 6, 7, 16, 38, 40, 41, 43

VL53L1X is a LiDAR used in RoFICoM 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 31, 37, 41

1 Introduction

The RoFI platform is an open software and open-hardware platform for modular robots. The RoFI platform utilizes the RoFI Communication Mechanism, RoFICoM for short, for connecting and communicating between two RoFI modules.

This thesis improves the currently implemented firmware for the connector (RoFICoM) of the RoFI platform.

Figure 1.1: The RoFI platform logo with Universal module [1]

The RoFI platform is established in Jan Mrázek's thesis [2]. His thesis focuses on modeling the platform as well as specifying the requirements for modules and even the docking system. In the case of the docking system, his thesis introduces connector-to-connector and connector-to-module communication and the mechanical structure, which was improved over the proposed version.

Even though the thesis showcases the connector prototype, that prototype differs a lot from the current version. The more up-to-date version of the connector is described in the paper "RoFICoM — First Open-Hardware Connector for Metamorphic Robots" by Jan Mrázek and Jiří Barnat [3].

In the following text, the usage of "prototype" is used to address the project version before this thesis, and the usage of "current" (im-

plementation or version) to describe the resulting project after the thesis.

Even though the previous work creates the connector, the implementation is in a work-in-progress state. The aim of this thesis is to enhance the firmware of the RoFICoM to a production-ready state by improving the prototype implementation and adding new features.

The main objectives for this thesis are to:

- implement distance measuring into RoFICoM, which includes implementing an I^2C driver and integrating the driver for distance measuring,
- propagate the measurements from the RoFICoM to the RoFI module,
- port the prototype firmware for the RoFICoM to the production-ready version of the hardware board.
- change motor control in the firmware to use newly added magnetic sensors, and
- implement power-sharing — i.e., reading current and voltage on internal and external powerlines, charging accumulator from internal powerline, and discharging the accumulator to the internal powerline.

The structure of this paper starts with an introduction to the RoFI platform and a description of its components in the next chapter, which is followed by an overview of the prototype firmware implementation and proposes what changes were made to improve it. After that, this thesis focuses on changes made to the RoFICoM firmware. Those changes are distance measuring in the LiDAR chapter 4, Board Support Package for isolating board-specific code and adapting the firmware to changes in the board (chapter 5), and other interesting implementations and their reasoning (chapter 6). The last chapter (chapter 7) focuses on changes and updates to the RoFI Hardware Abstraction Layer, followed by the conclusion of this thesis.

2 RoFI

The RoFI is a platform of distributed self-reconfigurable modular robots developed at the Faculty of Informatics, Masaryk University.

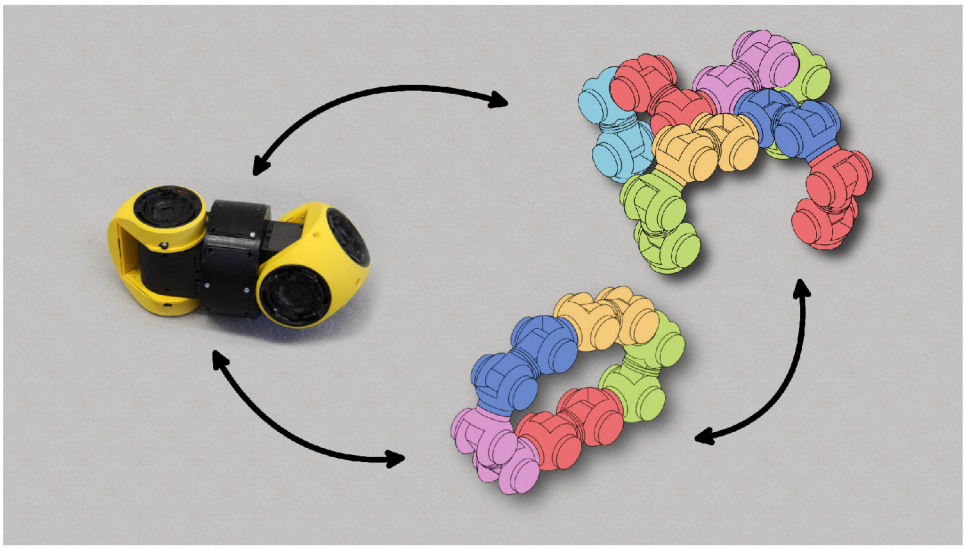

Figure 2.1: Illustration of possible RoFI robots transformation from [1]

The main goal of the RoFI platform is to provide a well-established open-source and open-hardware platform for metamorphic robots. Therefore, source code and hardware schematics are publicly provided in the GitHub repository [1]. The primary use case for such a platform is testing reconfiguring algorithms on a hardware platform and further researching this topic rather than instead of addressing real-world issues.

The basic building block is the RoFI module, which is an individual module that can connect and communicate with other modules. A group of connected RoFI modules makes a *RoFIbot*. A specific example of the RoFI module is the Universal module.

1. `https://github.com/paradise-fi/RoFI`

2.1 Universal module

This section provides only highlights of the functionality and design of the Universal module, the detailed description can be found in [2, chapter 3].

Universal module is a versatile, already-built instance of the RoFI module. The Universal module is not designed as an all-in-one module but rather as a module with minimal necessary features that can be used in nearly all systems. Thus, the Universal module uses only two sensors: an *inertial measurement unit* (IMU) and a *distance sensor* (LiDAR). If other specific sensors are needed, it is expected to use them as their own modules.

An inertial measurement device measures the unit's acceleration, angular velocity, and orientation. The measured data can be used in algorithms coordinating the modules.

The distance sensor (LiDAR) is placed in the middle of each dock and described in detail in chapter 4. The LiDAR measures the distance from the dock to an obstacle or a surface, which can help to align docks when connecting. The LiDAR can scan the environment, which can be useful when the RoFIbot needs to walk upstairs or needs to know about its surroundings.

The module is divided into four components: two bodies and two shoes. The main purpose of shoes is to establish connections with other modules and provide movement. Each shoe contains three docks and one joint. In comparison, the body encloses actuators, electronics, and accumulators. The last joint is situated between the two bodies. In total, the module contains six docks, three joints (motors), and two types of sensors.

2.1.1 Architecture

The docks used in the Universal module are called RoFICoMs. The RoFICoM is described in detail and explained in the next section 2.2.

Each module has a single *control unit* that serves as a centralized control over the actions of the module and all the module's components. Control of the motors is accomplished with a UART (Universal

asynchronous receiver/transmitter) [2] protocol, whereas in the case of the docks, it is done with SPI (Serial Peripheral Interface) [3] protocol. Additionally, the control unit manages charging and power connections.

The architecture of the Universal module is shown in Figure 2.2. Each box represents a component of the Universal module, and each line represents a connection between components. All connections are done by wires, and connections with labels (such as UART, SPI) represent the used protocol or describe the wire (INT, EXT).

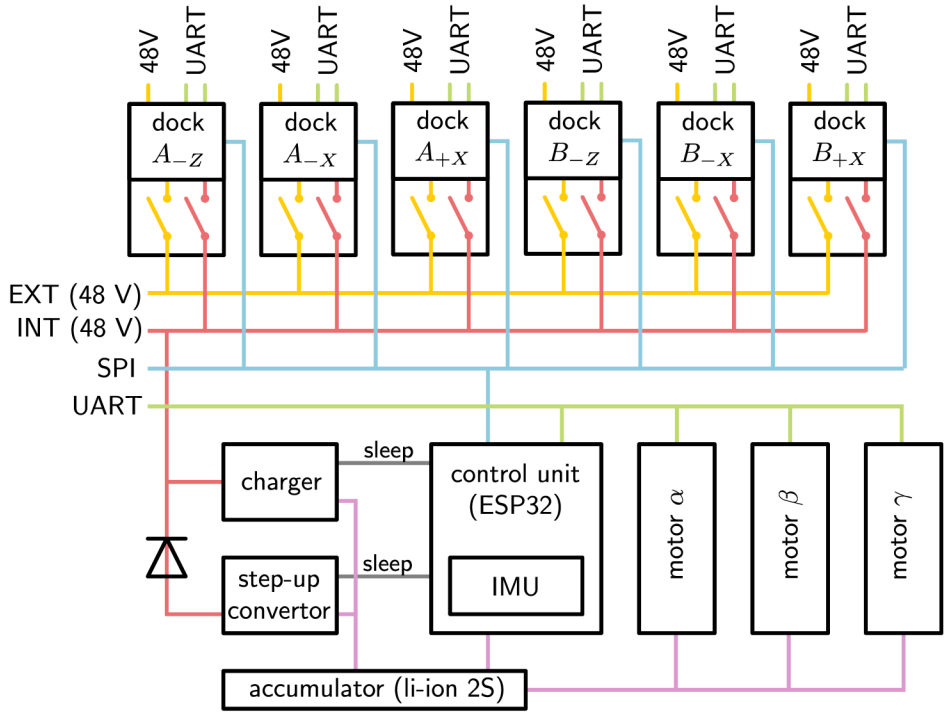

Figure 2.2: The architecture of the Universal module from [2, p. 35]

2. https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter
3. https://en.wikipedia.org/wiki/Serial_Peripheral_Interface

2.2 RoFI Communication Mechanism

RoFICoM focuses on reusability in order to make it easily integrable into any module for which RoFICoM provides communication to other modules. As the name suggests, its primary purpose is to (inter)connect modules. In order to achieve its purpose, the RoFICoM has a well-defined interface that can be divided into three parts:

- mechanical structure,
- connector-to-connector interface (communication between two RoFICoMs), and
- connector-to-module interface (communication with the integrated module).

2.2.1 Mechanical structure

This section describes the mechanical components, their functionality, and the mechanism used to extend and retract the connector. This section summarizes the previously mentioned paper "RoFICoM" [3], where the structure is explained more deeply.

RoFICoM is built from three main parts: *body*, *clip*, and *skirt*. These three parts are shown in Figure 2.3

The following text describes the movement of the RoFICoM's components in summary to provide an idea of the movement. The detailed description can be found at [2, p. 10–13].

The body is a non-movable part that is mounted into the module. The clip and skirt are essential for establishing strong mechanical connections between two RoFICoMs. The clip component consists of four latches attached to a ring. Those latches get attached while connecting with other RoFICoM. While expanding, the clip gets rotated and expanded upward by the motor. In the process, the latches on the mating side slide under each other, forming a mechanical connection that prevents connectors from being pulled apart. However, the latches themselves are insufficient for a mechanical connection, as a mutual rotation of the connectors releases the connection. This is why the skirt is needed, two connected skirts stop the rotation against each other. In order to help ease the process of connection, there are magnets placed in the holes of the skirt. This helps when two connecting sides are slightly misaligned.

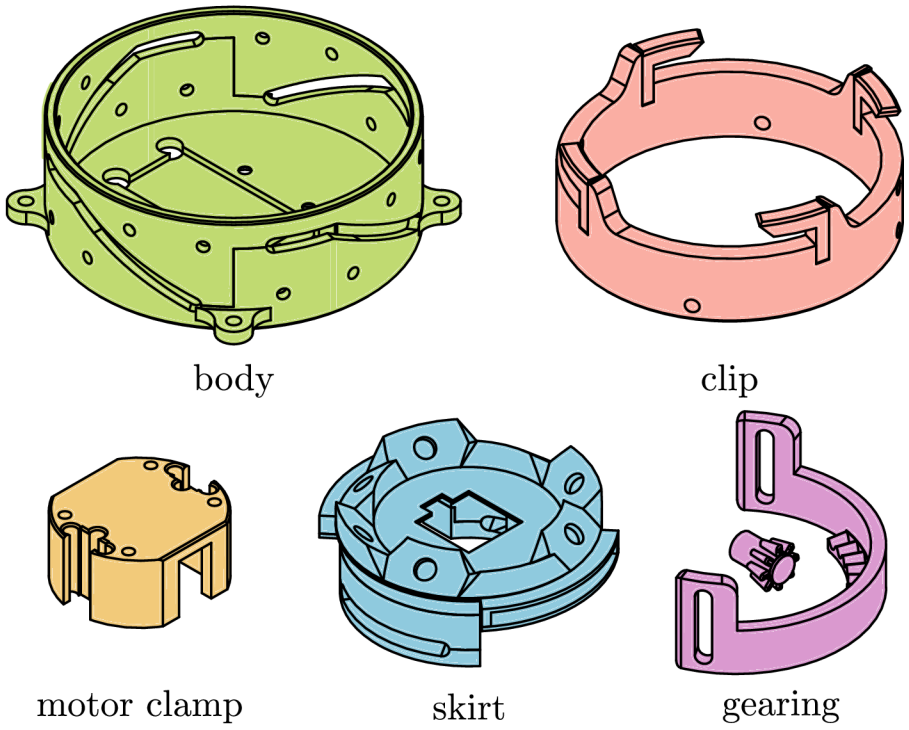

Figure 2.3: The components of the connector from the paper "RoFICoM" [3]

The motor that rotates the clip and the skirt is at the body's bottom. The rotation of the clip and the skirt is accomplished by another component, the gearing, which is connected to both the skirt and clip with a small steel pin that spans from the holes in the body up to the skirt. When the motor activates, it rotates the gearing with the two components and the steel pins move the interconnected components among the hole path in the body, elevating the components.

The Figure 2.4 shows the composed components of the RoFICoM, the color of each component matches colors in Figure 2.3. The outmost ring part is the body in the color green. Next in the body, the pink circle is the clip. The gearing is the purple half-circle connected with the clip by the steel pin.

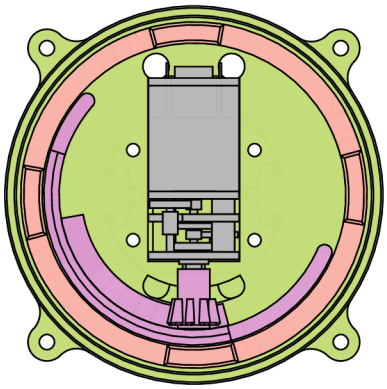

Figure 2.4: "Internal arrangement of the connector — top view without skirt. The motor is shown gray." from the paper [3]

2.2.2 Connector-to-Connector Interface

A small printed circuit board (PCB) inside the skirt with spring-loaded pins realizes the connector-to-connector connection. This PCB provides both data and power-sharing to the connected module. RoFICoM uses *UART* transferring frames with specified format and CRC for data communication. Those frames hold "blobs" — binary data transferred between modules.

2.2.3 Connector-to-Module Interface

RoFICoM provides a *SPI* interface for communication between the module and RoFICoM. The communication on the bus is performed in transactions. Transaction format consists of a *command*, the first 8 bits of a transaction, followed by a *command data*, then at least 8 bytes pause with connector response. Commands 0 to 127 are reserved for the base connector and must be supported by all connector types. For example, some connector commands are: version command, status command, and send and receive earlier mentioned blob.

2.2.4 Hardware details

This section describes the hardware details of the RoFICoM. The schematics and the board design can be found in the project under the path `RoFI/hardwareModules/RoFICoM/hardware/control_board/`.

The RoFICoM is actually split into two boards: skirt connector and system board, which are connected with a cable connecting the pins between them. The skirt connector board holds the LiDAR and pins for connector-to-connector connection, where the pins are:

- UART pins (TX — transmit pin and RX — receive pin),
- EXT and INT powerlines, and
- "sense" pins for detecting the orientation of the connection.

The internal (INT) powerline can be used to charge the current module or connected modules from the current one. On the other hand, the external powerline is used only to connect the powerlines of other modules. There are four used pins for the LiDAR: two pins for I^2C (SDA and SCL explained in subsection 4.1.1), the LiDAR's enable pin and the LiDAR's interrupt pin. The skirt connector is placed in the middle of the skirt, thus, moving together with the skirt.

The system board contains all other necessary circuits for RoFICoM to function. It consists of the board's Microcontroller unit (MCU), motor, power connections, power-sharing, reading current and voltage from power lines, and SPI interface needed for the connector-to-module interface. That being said, the circuit for measuring the current is not attached at the moment.

The magnetic sensors for the motor's retraction and expansion limit could not detect and stop the movement of the motor fast enough, which damaged the RoFICoM's case. As for the magnetic sensors, instead of two pieces, they got increased to an array of ten pieces along the motor's movement path to determine the motor's speed and position more precisely.

The RoFICoM has microcontroller STM32G071CBUx from the STMicroelectronics. This microcontroller from the STM32G0 family is a high-performance ARM Cortex-M0+ processor.

The main features of this microcontroller are:

- a maximum clock frequency of 64 MHz,

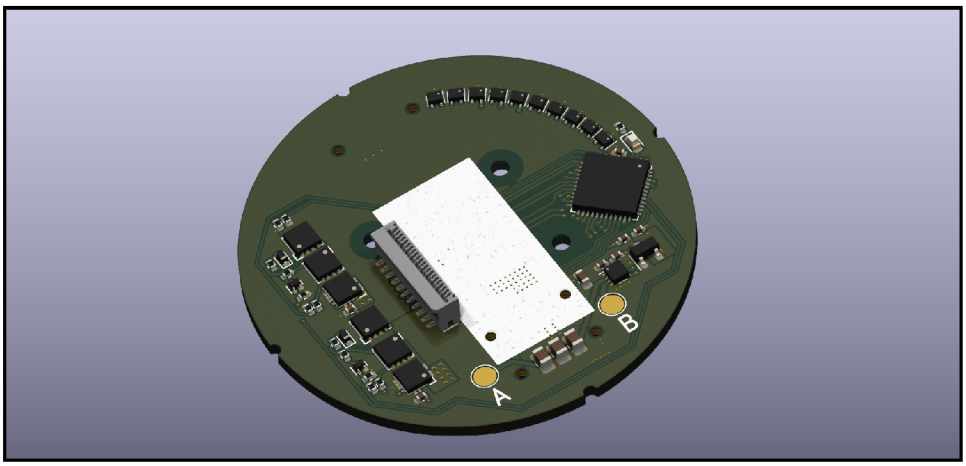

Figure 2.5: 3D render of the system board without the motor. At the right side of the system board is the MCU next to the arc of ten magnetic sensors. The white space in the middle is the place where the motor is soldered.

- the biggest size of flash memory currently in embedded microcontrollers of 128 kB,
- static RAM of size 36 kB, and
- 48 available pins.

The microcontroller has other functionality, such as CRC calculation, SPI, UART, I^2C.

3 RoFICoM — prototype version and current implementation

This chapter focuses on the software of the RoFI platform and describes both the state of the software in the prototype implementation and the changes in the current implementation.

The RoFI repository is an enormous monolithic (mostly) `C++` project and contains many RoFI related sub-projects. For this thesis, only several sub-projects are essential. Those sub-projects are the RoFICoM's firmware, stm32 drivers, and RoFI HAL.

3.1 RoFICoM's Firmware

RoFICoM's firmware is located in the repository at the path `RoFI/hardwareModules/RoFICoM/software/control_board/`.

The prototype version had several needed features implemented. Those features were the `spiInterface` (the connector-to-module interface), `connInterface` (the connector-to-connector interface), and motor control. However, since the motor control could not stop the motor fast enough and the magnetic sensors for detecting the motor position changed, the firmware for motor control also required an update. However, the `connInterface` is not fully implemented in the prototype and is missing sending all the information defined by the protocol. Due to the addition of lidar, the protocol had to be extended.

3.1.1 Connector-to-Module Protocol

The protocol for communication had to be revised because of the addition of distance measurement to the RoFICoM and the need to transfer the distance to the control unit. After the revision, it was decided that only the status command would be extended by needed information (distance status, measured distance, and distance mode).

As previously mentioned, the protocol was established in the thesis [2, p. 19–22], and the following text only describes the final look the status command (not the other commands).

The status command (command of value 1) is used to request and change the status of the connector, such as expanding or retracting the connector. The module's control unit initiates the communication by sending the command data and waiting for the response.

The format of command data is shown in Figure 3.1, where the upper left corner is the sent data's first bit (bit 0). Each row represents two bytes sent in the transaction, and the rows are sent from top to bottom. The numbers above the format specifies the bit of the row in that column.

The first row of the command data represents the request state, whereas the second row is a bit mask for the previous row. The mask is used to signal which values of the first row to apply. The meaning of the status bits is the following:

- P (position) — 0 = retracted connector, 1 = extended connector,
- I (internal powerline) — 0 = internal powerline is disconnected, 1 = internal powerline is connected,
- E (external powerline) — 0 = external powerline is disconnected, 1 = external powerline is connected, and
- DM (distance mode) — 0 = autonomous distance mode, 1 = short distance mode, 2 = long distance mode.

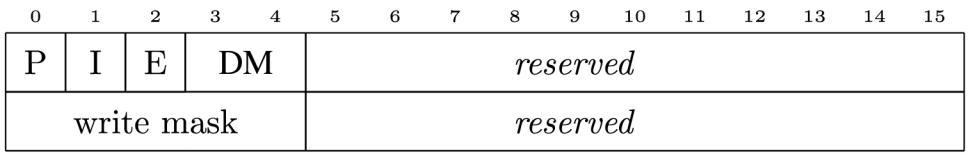

Figure 3.1: The new format of the status command data

The prototype version of the protocol already contained the bits P, I and E. The current version added the 2-bits for setting the distance mode. The autonomous distance mode makes the RoFICoM handle the distance mode of the LiDAR and is explained in subsection 4.4.3. The manual handling of the distance mode is done via setting either the short or the long mode, described in subsection 4.3.2.

Figure 3.2 shows the format of the status command response.

- C (connected) — 0 = not connected to mating side, 1 = connected to the mating side,

0	1	2	3 4	5 6 7	8	9 10	11	12 13 14 15
P	I	E	DM	*reserved*	C	O	L	*reserved*
blobs pending to send					blobs pending to retrieve			
INT line voltage (8.8 fixed-point)								
INT line current (8.8 fixed-point)								
EXT line voltage (8.8 fixed-point)								
EXT line current (8.8 fixed-point)								
LiDAR distance								

Figure 3.2: The new format of the status command response

- O (orientation) — 0 = north, 1 = east, 3 = south, 4 = west, and
- L (LiDAR status) — 0 = error, 1 = not measured, 2 = out of range, 3 = fully valid.

Compared to the prototype version of the protocol, the new version added the L bit for the LiDAR status and two bytes for the measured distance in millimeters.

3.2 Stm32 Drivers

The `stm32cxx` sub-project abstracts and wraps around the bare `C` stm32 drivers into nicer `C++` classes. This abstraction allows using the drivers without explicitly handling family-specific code and provides easier and prettier usage rather than using the `C` drivers.

The port files of the `stm32cxx` handle the portability or the MCU family-specific code. Folders with the family-specific abstractions are under the path `RoFI/softwareComponents/stm32cxx/port/`. The `port` files are in the format `{driver}.port.hpp`. For example, the port file for I^2C for the microcontroller of RoFICoM is in `.../stm32cxx/stm32g0xx/i2c.port.hpp`.

Port files follow the same guidelines, which is that the specific code is under `namespace detail`, and it is implemented via *mixins*. However, since `C++` does not natively support mixins, this is done

using the C++ technique called CRTP. CRTP [1] (Curiously recurring template pattern) is a templated class, which takes the class we wish to extend with our mixin as a template parameter, and implements the mixin functionality on the template class. Such functionality might be enabling a clock for the periphery or setting an alternative function on periphery pins.

The prototype version had many drivers implemented (that were needed), such as drivers for General-purpose input/output (GPIO), UART, SPI and more. The current implementation mainly added the driver for I^2C and some smaller additions, namely timer counter, blocking wait for time, and missing family-specific definitions.

3.3 RoFI HAL

Hardware Abstraction Layer, or HAL for short, is part of the RoFI platform that provides an abstraction over the running platform, which can be real hardware or a simulator. It can also be used as an abstraction over different RoFI hardware — modules with motors, sensors and more — without changing the user firmware. Therefore, RoFI HAL aims to define a solid and compact interface to make both — the implementation and the usage simpler.

The RoFI HAL is divided into *interface* and *implementation*. Every change that breaks the previous interface version is costly because that change needs to be reflected in every implementation, which can be a lot. For example, there can be an implementation for each module with a specific sensor, motor, and multiple simulators. However, due to the addition of LiDAR, the interface needs an update to reflect this addition.

The prototype had implementations for the simulator and esp32. The esp32 is the implementation for the Universal module.

The actual changes to the prototype HAL are described later in chapter 7.

1. https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern

4 Distance Measurement — LiDAR

This chapter describes LiDAR and its capabilities, how to use the VL53L1X LiDAR, which includes communicating with the VL53L1X over the bus and using the VL53L1X driver provided by the manufacturer. The last section describes the current implementation of the LiDAR driver, a `C++` class abstracting the usage of the VL53L1X driver.

LiDAR, which stands for Light Detection and Ranging, is a device that measures range. LiDAR uses laser and time, in which the light travels to an object or a surface and bounces back to determine the distance. Thus, the results of this technique are very dependent on the ambient light in the measured environment. As such, LiDAR is used for measuring the distance from an object and *3-D scanning*. 3-D scanning is collecting data about the real-world environment into a digital representation. Many industries use LiDAR — one of the most known use cases might be in NASA's Mars helicopter Ingenuity [4].

RoFICoM uses VL53L1X LiDAR from STMicroelectronics. This specific LiDAR has the following main features: (All features and description can be found in the part's datasheet [5].)

- the ability to accurately measure the range between 4 cm and 4 m,
- up to 50 Hz ranging frequency,
- single 2.8 V power supply,
- I^2C communication interface with up to 400 kHz speed, and
- shutdown and interrupt pins. (However, these pins are not necessary.)

Even though VL53L1X accurately measures the range between 4 cm and 4 m, it also measures distances below and above this range, although the measured distance does not have to be accurate. The lidar library, which RoFICoM uses, deals with inaccuracy by reporting the status of measured data, which is explained later.

The shutdown pin is used to reset the VL53L1X by software. That being said, on the actual hardware, the pin is inverted, which results in behavior more similar to an enable pin, i.e., the device is enabled when the bus with a shutdown pin is driven high, and the device shuts down

when the bus is driven low. This pin does not have to be controlled by software. However, in that case, it still has to be connected to the power supply through a pull-up connector.

On the other hand, the interrupt pin does not require to be connected. This pin signals that the new distance was measured, it is only an output pin. Thus, this pin cannot be used from the controller, and the controller must signal the successful data transfer through I^2C.

The alternative to the interrupt signal is polling. Polling means that the firmware asks the VL53L1X periodically whether the measurement data are ready to be transmitted. However, if the period is too long, it is possible to miss some measured data. For example, if the inter-measurement timing (the time between two measurements) was 100 ms, and the firmware's asking period was 500 ms, the firmware could miss four measurements from the VL53L1X.

In both cases, the controller should signal the successful read as soon as possible; otherwise, the VL53L1X would be waiting for the data read, unable to change the measured data, thus, missing newly measured data.

4.1 Communication

This section describes details of communication with the VL53L1X — i.e., the protocol used for communication, how the protocol works, usage of the protocol in VL53L1X, and the current I^2C implementation for the VL53L1X driver.

VL53L1X uses the I^2C communication protocol. The next subsection describes this protocol. If you are familiar with the protocol, you can continue to subsection 4.1.2.

4.1.1 Inter-Integrated Circuit Protocol

I^2C, which stands for Inter-Integrated Circuit, is a protocol designed for communication between a controller (or more controllers) and multiple peripheral devices. The protocol requires only two lines: SDA (serial data) and SCL (serial clock). Both lines are open-drain with pull-up resistors, which means that the bus driver can drive the bus

line low but cannot drive it high. This constraint prevents the collision driving the line both high and low, which would damage the drivers.

The protocol's communication is done in 8-bit packets. The transaction starts with the start condition. After that, every packet is confirmed from the receiving side with one acknowledgment bit, and the transaction ends with the stop condition. Since more peripheral devices can be connected to the I^2C bus, the controller must first address the peripheral to communicate.

The address of a peripheral is a 7-bit (in rare cases 10-bit number). The reason why the address is only 7-bit and not the full 8-bits of the packet is because the last bit indicates the transaction type. The transaction type is either read transaction (the bit is set to 0) or write transaction (the bit is set to 1). The read/write bit is the least significant bit of the packet, and the address is contained in the seven highest bits of the packet (i.e., the address is shifted left by one bit). After the address is sent and met with acknowledgment from the peripheral, the data begins to be transmitted.

The start condition is when the controller drives the SDA line low before the SCL goes down. The stop condition is when the controller releases SDA (the line goes from low to high) **after** the SCL line changes to high. It is suggested not to change the SDA line while SCL is high to avoid signaling false stop conditions during data transmission. Note that the acknowledgment bit is negated, i.e., logical 0 on the bus line signals that the peripheral received the packet successfully, whereas logical 1 signals the opposite.

4.1.2 Addressing VL53L1X's Registers

VL53L1X not only uses an I^2C address but also requires the address of its register, which is a 16-bit number. However, the I^2C protocol only addresses the device. In order to address the device's register, a normal transaction is started, where the first two data packets are the two bytes of the register address. The most significant byte of this index (register address) is sent first, followed by the least significant byte. Sending the index applies to both types of transactions (read and write). That means that to read data from the VL53L1X, the controller needs to make two transactions: a write transaction with register index, and the actual read. However, writing data to the VL53L1X makes

only one write transaction because the index and data **must** be written together in one transaction.

4.2 VL53L1X Interface

Even though VL53L1X has clearly defined read and write operations, there is no public specification of how to control the VL53L1X, i.e., how to initialize the VL53L1X, start measuring and request the measured data. STMicroelectronics provides and recommends using their software API, in the form of a software driver, to control the VL53L1X. The driver is a bare `C` library (not a `C++` library) and is dual licensed under STMicroelectronics' proprietary license and under BSD 3-clause license, which is compatible with RoFI's MIT license.

STMicroelectronics provides three driver versions: Full API (standard), ultra lite driver (ULD) and Ultra Low Power (ULP). The full API, or standard, driver implements a plethora of functionality together with errors and error messages. The ultra lite driver strips down the API only to necessary functionality; however, it contains calibration, which can be removed if the space is needed. The Ultra Low Power driver aims for the lowest resource usage possible.

The ultra lite driver was selected for the RoFICoM because the full API was too large for the firmware. The prototype firmware already used a lot of the flash memory, around 80% in debug build and above 60% in release build, making the usage of the full API driver unfeasible. The size of the full API driver's release build took up to 19 kB of the MCU's Flash. In comparison, the release build of ultra lite driver takes only 1 kB.

The use of ULD resulted in missing errors and error messages previously used with the full API driver. The ULD only reports failures either caused by lidar (signified as value 1) or from the under-laying layer (I^2C communication). Nevertheless, in both cases, we lose the error message. The precise design of the error handling of the I^2C class (subsection 4.1.1) together with the examination of the error handling of the ULD (subsection 4.3.1) resulted in some preservation of error information.

4.3 API of the ultra lite driver

Because the ULD API is platform independent, it requires the user to implement I^2C communication for the platform (the implementation of I^2C in RoFICoM is described in subsection 4.1.1). The UM2510 [6] manual explains how to use the ULD API and will be described in this section.

The communication flow is following:

1. Wait for a successful boot of the VL53L1X.
2. Initialize the VL53L1X.
3. Set optional settings for the VL53L1X, such as changing the VL53L1X's I^2C address, changing the distance mode, changing timings for measurements.
4. Start ranging, i.e., start measuring the data.
5. Wait for the data to be ready — either by polling or by interrupt.
6. Receive the data from VL53L1X.
7. Prepare for the next ranging — clear the interrupt flag.
8. Either repeat from step 5 or stop the measuring.

4.3.1 Error Handling

Since the ULD is a `C` library, the error is signaled by the returned value from every API function. When results other than errors are needed, they are provided as an "out" argument, i.e., a pointer to the given variable.

The error type (`typedef int8_t VL53L1X_ERROR;`) for the ULD is an 8-bit signed integer. The full API driver defines error values among the definition of the error type; however, this is not the case for the ULD. Even though the ULD does not specify error codes, the code for no error is defined to be 0. Although the ULD usually returns errors from the underlying I^2C communication, in some cases, the ULD sets the error to value 1. Thus, the typical error handling of the ULD API calls looks like the code in Listing 4.1.

4.3.2 Distance Modes

VL53L1X has three distance modes: *short*, *medium* and *long*. Since the LiDAR depends on ambient light, this is when distance modes make

```
VL53L1X_ERROR status;
status = VL53L1X_SensorInit( ... );
if (status != 0) {
    ... // Report error
    return status;
}

status = VL53L1X_StartRanging( ... );
if (status != 0) {
    ... // Report error again
    return status;
}
```

Listing 4.1: Structure of error handling of the ULD function calls.

a difference. Even though short mode can measure only up to 136 cm, it is not as impacted by the ambient light as other modes. The datasheet [5] mentions that the short distance mode can measure 135 cm under strong ambient light, which is nearly the same as without ambient light. On the contrary, the ambient light impacts the medium and long modes. Whereas the long mode can measure up to 360 cm, in the strong ambient light, it only measured 73 cm.

The ULD provides only two distance modes: short and long. The representation for modes is `1` for the short mode and `2` for the long mode. The ULD provides functions `VL53L1X_SetDistanceMode()` and `VL53L1X_GetDistanceMode()` to set and retrieve the current VL53L1X distance mode.

4.3.3 Timing Budget

The *timing budget* is the time that LiDAR measures the distance. By extending the timing budget, the LiDAR is capable of measuring a greater maximum distance, but it comes at the cost of increased power consumption. The user can set the timing budget, which ranges from 20 ms to 1000 ms. There are a few constraints on the timing budget. The minimal budget timing that can be used for all distance modes is 33 ms; however, the short distance can use lower timings. In order to

be able to measure 4 m in the long distance mode, the timing budget must be at least 140 ms.

For the ULD, there is a difference in allowed values compared to the timing budget in the datasheet [5]. The allowed values for timing budget in the ULD are only [15, 20, 33, 50, 100, 200, 500 ms]. The default value is set to 100 ms.

4.3.4 Inter-Measurement Period

The *inter-measurement* period is the time between the consecutive measurements. Therefore the inter-measurement period must be longer than the timing budget. The inter-measurement period is also programmable by the user.

The inter-measurement period must be longer than the timing budget. However, the ULD does not validate this and is left to the user to ensure. Otherwise, the actual inter-measurement period would be double the expected value. The default value is set to 100 ms.

4.3.5 Result Data

The main reason for having LiDAR is the ability to measure the distance from the RoFICoM; thus, receiving the measurements and their correctness is the main focus. The result can be received in two ways, either by requesting the specific result values, such as distance, or by receiving all the values at once. The current implementation only uses measured *distance* and the *status* of the measurement.

The distance is measured in millimeters and is saved as two bytes long unsigned number.

Status On the contrary, the status is only one byte long but has more complex meaning. The ULD manual [6] documents only five statuses, even though the VL53L1X datasheet [5] specifies eight. The five statuses with their corresponding values are:

- Valid (0),
- Sigma failure (1),
- Signal failure (2),
- Out of bounds (4), and

- Wraparound (7).

The range status of 0 indicates successful measurement with the measurement data being ready to use.

The sigma failure with value 1 means that the standard deviation of the measurements had surpassed the expected (normal) values, thus resulting in an error. This error can be fixed by increasing the timing budget.

The signal failure with value 2 is caused by VL53L1X measuring a weak signal. The signal rate is represented as the amplitude of the reflected signal from the target. This failure can happen when the target is too far or is not reflective enough, or there is no target. Since increasing the timing budget increases the maximum measurable distance, it can eliminate this status but does not have to.

The out-of-bounds with value 4 usually occurs when the target is at the maximum possible measurable distance (and the target is very bright). Subsequently, the measured results may be inconsistent.

The wraparound range status of value 7 is reported when the target is very reflective, and the distance is longer than the limited physical distance measurable by the VL53L1X. This distance is 1.3 m for the short distance mode and 5 m for the long mode.

4.4 LiDAR Driver

Because most of the RoFICoM project uses `C++` abstraction, the same was intended for the LiDAR. Moreover, the LiDAR was designed as an independent library from the RoFICoM firmware. However, the LiDAR library depends on the `stm32cxx` drivers. The main goal for this driver is to make pleasant abstraction around the actual `C` driver. Having the LiDAR driver abstraction enables to change or update of the underlying driver without more or less changing the RoFICoM's firmware. The LiDAR driver was designed to keep communication flow similar to the `C` driver (as described in section 4.3).The LiDAR is supposed to be *hot-pluggable*, which means it can be inserted or removed at any time without restarting or crashing the system. Since the LiDAR itself does not have to be plugged into the RoFICoM, it was desired to have the management of the LiDAR state more explicit. The states are: LiDAR is not present, LiDAR is present but not yet

initialized, LiDAR is initialized but not measuring, and last, LiDAR is measuring.

4.4.1 Error handling

Managing the state goes in hand with error handling since every operation can fail by either I^2C failing, `C` driver failing, or simply by the fact that LiDAR is not plugged in. Before considering the error handling, it is important to mention that the operations of VL53L1X differ in the returned value. For example, some function returns the distance mode, the measurement result, or in most cases, they do not return any value; however, all of them can fail.

There are several choices for considering error handling in `C++`. The first choice that comes to mind is exceptions. However, they are not suitable for embedded programming due to the overhead they bring and thus cannot be used. Another solution might be returning the error and passing in other return values as output argument (e.g., pointer or reference). This solution has mainly two issues. The first issue is that every function call needs to be explicitly checked whether it succeeded, significantly cluttering the code. The second issue is that there needs to be a variable in the caller function for each return value. This complicates the code; for example, consider what needs to be done to check whether the measured data are ready to receive from the VL53L1X. Firstly, there needs to a `bool` variable in the current scope that will hold the readiness of the VL53L1X, then the function call to get the readiness; after that, there would be a check whether anything failed, and if not, finally, the check whether the data is ready or not. This produces code that is much more complicated than having a simple check in the form `if (lidar.getMeasurementDataReady()) {...}`.

`C++23` contains a new mechanism for error handling the `std::expected`, similar to Rust's `Result` or Haskell's `Either`. However, at the time of writing, the project is using `C++20` since not all the features of `C++23` are yet implemented in the used compilers. Fortunately, the RoFI project contains implement of the concept under the name `Result` under the Atoms project. Atoms are helper libraries for the RoFI project, and the `Result` can be found under the path `RoFI/softwareComponents/atoms/include/atoms/result.hpp`.

The `Result` contains either *value* or *error*. The `Result` cannot contain both. It is similar to the `std::optional`, although it contains more information about the error. Both the value and error can be of any type. The `Result` provides the ability to check whether the result contains value or error (with either `bool` operator or `has_value()` method), receive the underlying value or error, and *monadic functions*. If you are unfamiliar with monadic functions, they allow you to work on the wrapped value (i.e., value or error in the case of `Result`) rather than the `Result` itself. Those functions save us from the constant checking of whether the result contains value or error. The monadic functions for the `Result` are `and_then()`, `or_else()` and `transform()`. The `and_then()` method applies its argument (callable function) on the value of the `Result` only if the value is present; otherwise, it returns the error. Similarly, the `or_else()` returns value if it contains value; otherwise, returns result of its argument (again callable function). The `transform()` and `transform_error()` are used to transform the type of value or error, respectively.

Even though the return value types differ for many functions, the error type is the same for all the functions; thus, it is important choice. The default error type for the `Result` is `C++`'s `std::string`. However, `std::string` is a very robust and heavy library (in an embedded context), which means it is unsuitable for the firmware. Two alternatives were considered — `std::string_view` and basic number (or an enum). The `std::string_view` was chosen since it allows error detection when developing. However, all functions are implemented in a way to allow for easy error type change. Every function returns the error wrapped by function `errorMessage( error )`. The function `errorMessage()` converts the bare error (in this case, the error returned by the VL53L1X driver) to the LiDAR's error. Thus, when changing the error type for the LiDAR, only the `error_type` member and the `errorMessage()` function must be changed.

4.4.2 Usage of the driver

The LiDAR's state, visualized in Figure 4.1, consist of four possible values: `Missing`, `Uninitialized`, `Initialized`, and `Measuring`. The state `Missing` represents that the LiDAR is either not present, or the RoFICoM cannot communicate with it. When RoFICoM recognizes that the

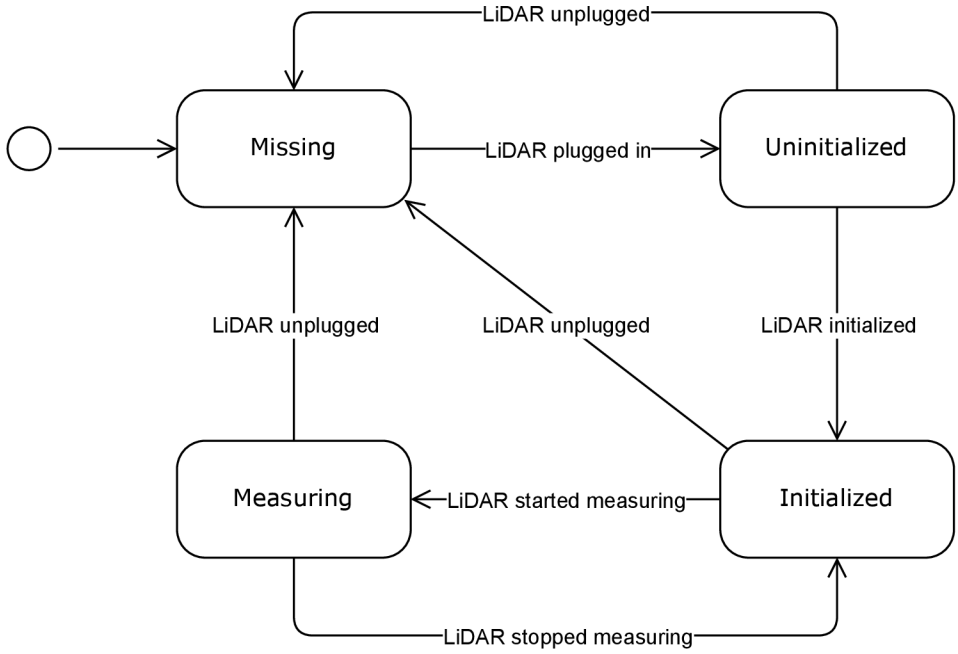

Figure 4.1: Diagram of LiDAR's state.

LiDAR is connected, it is represented as `Uninitialized` state, which means device is connected and running but needs to be initialized by the firmware. After initialization from the firmware, the LiDAR is ready to start measuring, the state `Initialized` can be understood as a software standby. While in `Measuring` state, after measurement, the LiDAR can be stopped in measuring and returned to software standby.

Since the LiDAR's current (only) requirement is always to measure the distance, the firmware accounts for two states — actively measuring and uninitialized (or unable to measure). The `Measuring` in Figure 4.1 stays the same as the actively measuring. However, the state "unable to measure" is combined from states `Missing`, `Uninitialized` and `Initialized`. These two new states make state handling easier in the code. The state "unable to measure" is handled by the function `lidarInit()`, and the state `Measuring` is handled by `lidarGet()`. The main purpose of `lidarInit()` is to correctly initialize the device and start measuring. On the other hand, the function `lidarGet()` expects the device to be correctly initialized and measuring. The purpose of

`lidarGet()` is to check whether the distance is measured, and if it is, save the measurements for the next transfer to the module's control unit.

Those two functions can call each other because if anything fails in the process (in both functions), the device probably needs to be initialized again. On the contrary, when the `lidarInit()` succeeds, it is safe to call `lidarGet()`. `lidarGet()` calls itself to ensure the latest measurements are saved. It is important to note that not receiving the data (i.e., the data is not ready yet) is not a failure and only means it is necessary to check on the data again. Otherwise, the measured value would never be received — the device would get initialized again before the first measurement.

Nevertheless, the previous implementation would result in infinite indirect recursion; to avoid the recursion, the current implementation uses `IdleTask::defer()`. `IdleTask` is a class that provides scheduling function calls; the scheduled functions are called with `IdleTask::run()`. The usage of scheduled function calls allows the firmware to schedule other routines between the two lidar functions, and handle necessary tasks, such as SPI communication.

4.4.3 Autonomous Distance Mode

The LiDAR driver provides one more additional distance mode that is not described in subsection 4.3.2. This mode is completely handled by LiDAR driver and in reality sets the two provided distance modes by the ULD. The autonomous mode does not change the set timings and instead works with the default ones, which were found suitable for most measurements.

The changes in the underlying mode of the autonomous mode are dependent on the status and distance of successful measurement. The autonomous mode starts in the long mode, and if the measurement is valid (signaled by the status value) and the measured is under a specified threshold under its maximum measurable distance, the mode is set to short. If there is invalid measurement while in the short mode, such as sigma failure, out of bounds, or wraparound, the distance is set to the long mode.

5 Board Support Package

Most of the development was done on the prototype board, but it was expected to move to the production-ready (current) version of the hardware board. This was important, especially because the MCU's case changed, which resulted in pin changes and the addition of magnetic sensors for the controlling motor. An easier transition is accomplished by using BSP.

Board Support Package, or BSP for short, is a layer of abstraction specifying minimal needed hardware requirements for the project and procedures for board initialization. For example, needed pins and peripherals, such as I^2C.

The main benefit and use case for BSP is having all board-specific code in one place. The code typically used in BSP is pin definitions, initialization routines and overall board initialization.

BSP is usually used in an embedded environment and provides many benefits. Not only it abstracts hardware details, but it also establishes a single source of truth for pin definitions, which is less error-prone than having them defined at multiple places. BSP assures correct initialization of the board at single place. Having everything at a single point enables easier portability across different board versions. The abstraction allows for using the components defined in BSP in different independent parts of the project.

In RoFICoM, BSP is implemented as two files: `bsp.hpp` and `bsp.cpp`. The `bsp.hpp` header file contains all declarations of pins and peripheral drivers, which live in namespace `bsp` and are exposed for use in RoFICoM. However, the function `bsp::setupBoard()` **must** be called before any pin or driver is accessed. The function `bsp::setupBoard()` initialized the board, pins and drivers provided in the `bsp` namespace. The header file provides the declaration of function `dbgInstance`, which is a function that RoFICoM uses for debugging with a `uart` driver. `bsp.hpp` depends on the function `SystemCoreClockUpdate()`, which is a platform-dependent function that needs to be called after the board clock has been set up. On the contrary, the `bsp.cpp` defines and implements everything declared in `bsp.hpp`.

`bsp.hpp` acts as an interface specifying what drivers and pins RoFICoM needs to function; however, since those pins and drivers are

stored as global variables in namespace `bsp`, the interface should be kept as minimal as possible to save space in the global space. On the other hand, `bsp.cpp` acts as an implementation of this interface and should mainly correctly set up the board, pins, and drivers for usage; that means the `bsp.cpp` can use pins and drivers that are not exposed as it needs to meet the interface specification.

To illustrate the main benefit of the BSP, imagine a scenario where MCU on the board must be replaced with a different family, thus, changing the pin layout. In this scenario, it would be only required to change the `bsp.cpp` file with pin definitions (and if not already present implementation of drivers), but the `bsp.hpp` and components built on top of that would be left unchanged and still working.

5.1 `Gpio::Pin` Initialization

`Gpio::Pin` initialization is the most notable and important change in `bsp` that differs from all other uses in RoFICoM. RoFICoM uses the expression `GpioA[ 9 ]` to specify the used pins. `GpioA`, `GpioB` and `GpioC` are global objects provided by the `Gpio` driver. At the time of writing, the initialization order of global variables across multiple translation units is not defined by the language specification. As a result, `bsp` cannot use `GpioA`, ..., for pin definitions and instead uses the `Gpio::Pin` structure initialization, where the first argument is the pin position. The second argument is the GPIO line defined by STMicroelectronics.

6 Implementation — Design Decisions

This chapter discusses interesting design decisions and explains their reasoning. This chapter contains sections discussing the changes made to the build system with the related issue, the implementation of the I^{2}C driver, the implementation of the new motor driver, the implementation of the power-sharing, and the implementation of the connector-to-module protocol in RoFICoM.

6.1 Changes to the Build System

During the development, the firmware started hard faulting on the RoFICoM. Hard faults are exceptions that are typically caused by unrecoverable system errors. Such errors can be an execution of an Undefined instruction, an attempted load or store to an unaligned address, and many more, which are described in more detail in the ARM Devices Generic User Guide. In the debug build of prototype version the static usage of the RAM was very high (around 70%), not leaving enough RAM for the runtime of the firmware, which resulted in in corrupted stack frames. The fix to this issue was reducing the size of statically allocated packet buffer. The count of 1025B buffers was reduced from count of 20 to 5, bringing the static usage of RAM to manageable 30%.

Compiler flags `-fstack-protector-all` and `-fstack-check` were added to the debug build to detect similar issues as soon as possible.

6.2 I^{2}C driver

The VL53L1X requires for its communication the I^{2}C protocol, which is not present in the prototype version of the firmware. This section explains the decisions in designing the implementation for the `stm32cxx` I^{2}C driver.

The I^{2}C driver lies with other `stm32` drivers in a project at path `RoFI/softwareComponents/stm32cxx/src/drivers/`.

The driver contains class `I2C`, `SdaPin` and `SclPin`, to make the pins distinguishable from `Gpio::Pin`, i.e., when constructing the `I2C`

the following construction, where `pinSda` and `pinScl` are `Gpio::Pins`, `I2C(..., pinSda, pinScl)` would fail to compile. On the other hand, the construction `I2C(..., SdaPin( pinSda ), SclPin( pinScl ))` compiles. It also ensures that swapping the two pin classes would not compile, i.e., `I2C(..., SclPin( ... ), SdaPin( ... ))` would not compile.

The I^{2}C driver provides two functions: `read` and `write`. Those functions take the I^{2}C address of the peripheral and data container. There are also `unsafe` variants of these functions that are used in LiDAR I^{2}C implementation that take the raw buffer and its size instead of a container. Since the `read` and `write` are templated functions, they use private functions to implement them. These private functions are not templated and take the raw buffer and size. Using a slim templated function with an inner function that is not templated reduces the size of generated functions from the template but allows the usage of a generalized function.

6.2.1 Error handling

Since the two functions only take the arguments either for reading from or writing to the given argument, the functions are free to return anything. In this case, they return an error. Because the I^{2}C driver is used in LiDAR I^{2}C, the error values are chosen to be compatible with error handling in the ULD subsection 4.3.1. The I^{2}C error enum has the following errors:

- *Valid* value 0 — this is to be compatible with the ULD error reporting,
- *Reserved* 1 — Reserved for the error reported by the ULD,
- *NotAcknowledge* = 2 — The acknowledgment bit was not signaled after the 8 bits were transmitted,
- *BusError* = 3 — Bus error signals misplaced start or stop condition,
- *ArbitrationLoss* — Arbitration loss occurs when two masters start transmitting at the same time,
- *OverrunUnderrun* — Overrun is when the newly sent 8-bits are read before the previous 8-bits were read, thus, resulting in data loss. On the other hand, Underrun is when the 8-bits are sent

from the data register before the new 8-bits are written into the register, resulting in sending the same 8-bits of data a second time, and
- *BufferOverflow* — Buffer overflow happens when the receiving data are longer than the allocated buffer to store those data. The data are read until the end of the buffer and the error is reported.

6.3 Motor driver

As previously mentioned in subsection 2.2.4, the prototype version of the motor's driver could not stop the motor's rotation fast enough and damaged the RoFICoM's case. That resulted in addition of magnetic sensors and the motor driver had to updated. This section explains the current implementation of the driver.

The number of magnetic sensors increased from two to array of ten among the movement path of the gearing. The gearing contains a small magnet that activates the sensors while moving near them. While testing the movement, it was observed that at the same time multiple sensors can detect the magnet, which influenced the way the current position is measured.

The current position is measured as a sum of the positions of active pins divided by the count of active pins, further divided by the position of the last pin. The pins are indexed from 0 to 9, where the gearing in the retracted position activates the sensor pin 0 and the sensor pin 9 is active in an expanded state. Such indexing should mean that the count of active pins should always be at least one; however, due to the movement or malfunction, the count can be zero, resulting in a division by zero and crashing the whole system. The measured position ranges from 0 to 100 (the calculated value times 100) and sets explicitly the value -1 for error in measuring (i.e., zero active sensors). Explicitly setting the error value allows stopping the motor; otherwise, it could run indefinitely, damaging the structure.

The Figure 6.1 shows the possible states and their transition in the motor driver. The most important states are `Retracted` and `Expanded` as they indicate the requested position (the motor's goal). The modeled states in the driver are: `Retracted`, `Expanded`, `Retracting`, `Expanding` and `Unknown`. The `Unknown` state indicates the initial state

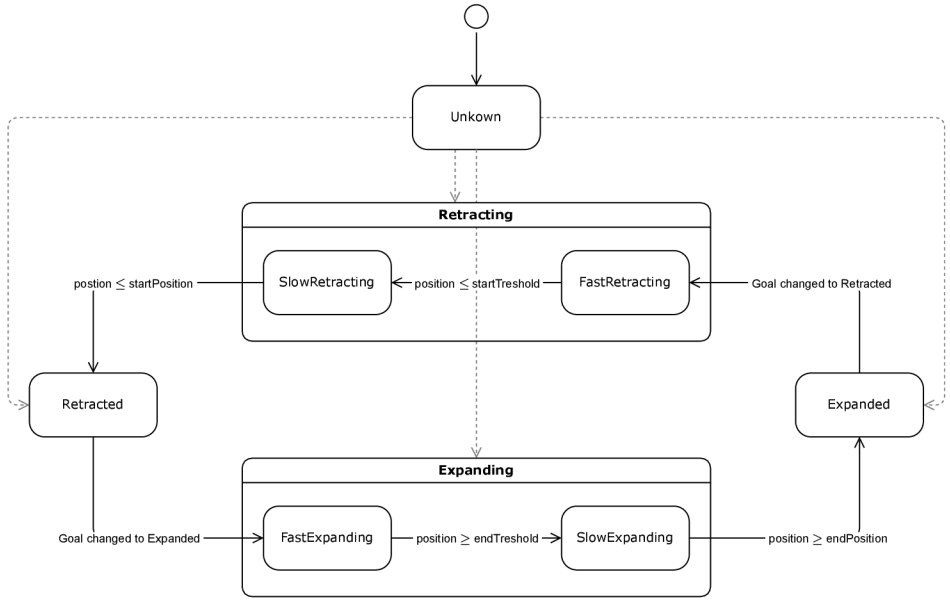

Figure 6.1: Diagram of motor's state

before the position is calculated; this state can change into any other state based on the set goal and current position. The `Expanding` is the transition from the `Retracted` state to the `Expanded` state; similarly, the `Retracting` state is the transaction from the `Expanded` state to the `Retracted` state. Both states (`Expanding` and `Retracting`) can be split into two more states: Fast and Slow. The Fast state utilizes the whole power of the motor, and upon reaching a specified threshold position, the state changes to the Slow one, which, on the contrary, slows the motor in order to stop the movement before reaching the physical end in the mechanical structure.

The thresholds were determined by manual testing.

6.4 Power-sharing

The last missing part for the protocol implementation in the RoFICoM is the power-sharing. The RoFICoM has two powerlines internal (INT) and external (EXT), those powerlines are connected to the mating side. The decided features for power-sharing are to charge or discharge the battery from the powerline, additionally, to measure the voltage and

the current on both powerlines. However, the measurement of the current could not be implemented because the circuit for measuring is not soldered in the current board.

The power-sharing is implemented as a `PowerSwitch` class in the file `connInterface.hpp`. The connection of the internal or external powerline is done via circuit switch, which result in controlling the switch by a pin on the MCU — pin with high value means that the switch is connected, pin with low value means the switch is disconnected.

The measurement of the voltage is more complicated and is accomplished with analog-to-digital convertor (ADC for short). ADC allows to read analog values, such as voltage, and convert them into digital data readable by the MCU. The result of the conversion depends on the number of bits the ADC outputs, in the case of RoFICoM, it is 12-bits. The ADC measures the input by comparing in to the reference input and returns the comparison as digital number, where the maximum digital number corresponds to the referential input. The reference voltage in the RoFICoM is 3.3 V.

The following formula $A = \frac{V_{ref}}{2^M} \cdot D$ calculates the conversion from the digital data (integer) to the analog data (float), where V_{ref} is the referential voltage, the D is the digital input from the ADC, M is the count of ADC bits, and A is the measured analog that is received from the conversion.

However, the voltage is connected to the ADC through resistors. The schematics of the connection is shown in Figure 6.2. The voltage is measured between two resistors connected in series, one with 100 kΩ (R_1) and the other with 6.8 kΩ (R_2). To receive the original voltage V from the measured input V_M, the following formula is used: $V = \frac{R_1+R_2}{R_2} \cdot V_M$

Combining the formulas ($V_M = A$) we receive the following calculation in the firmware: $V = \frac{(R_1+R_2)\cdot V_{ref}}{R_1 \cdot 2^M} \cdot D$.

However, the protocol specifies the data of the voltage to be in fixed format `8-bit.8-bit` number, which is converted by multiplying by 255.0 and converted to `uint16_t`.

Each measurement and conversion is done in the `run()` function, the results are obtained by the corresponding "get" function because

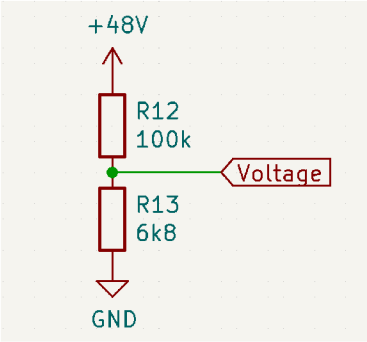

Figure 6.2: Schematics of the powerline voltage wiring

the when the communicating through the SPI RoFICoM cannot measure and must provide response fast enough to the control unit.

6.5 Protocol implementation in RoFICoM

As previously mentioned in section 3.1, the prototype version of the firmware was missing the full implementation of the connector-to-module communication. The unfinished command is the status command, which only implemented expanding and retracting the connector based on the received command and responding with blobs remaining to be sent and received. This section describes finishing the implementation of the status command.

The function `onCmdStatus()` accomplishes the status command parsing, reacting and responding, which is part of the reaction callback function given for the `spiInterface`. The `onCmdStatus()` parameters needed to be extended by the `ConnectorStatus` (manages connection to mating connector), `PowerSwitch` (power-sharing), `Lidar` and `std::optional< LidarResult >`. The LiDAR result is optional since the command status can be received before the LiDAR responds with the first measurement or when the LiDAR is not plugged in. Because of the new addition to the protocol, the structure containing the flags for command status was extended correspondingly (with LiDAR distance mode and result status).

The `onCmdStatus()` function can be virtually split into two halves — the first half parses the requested command and reacts to it, while the

second half creates and sends the response. The interesting part in the first half is that the LiDAR distance mode can contain an invalid value (`0b11`), which is simply ignored because the user's request should not be able to break the firmware.

The most interesting part of the second half is the transformation of LiDAR status into the protocol flag. The values of the flag are specified in the section 7.1. Each possible value for the flag can be derived from the possible value of `std::optional< LidarResult >`. If the optional does not contain a value, the result is *not yet measured* (or the LiDAR is not plugged in). Otherwise, the `Result` class must be present. If the `Result` has the error type set, the flag is set as such (the error value). The two checks handle the validity bit of the flag. After that, the `Result` must contain the value type, which is the VL53L1X result structure described in subsection 4.3.5. The certainty bit is decided by the value of the VL53L1X status from the result structure — value of 0 (i.e., the previously mentioned value of valid status) results in Fully Valid flag. In contrast, other values result in the Out of Range flag since the measurement may not be correct. Consequently, the distance is set only if the validity bit is set to valid.

7 RoFI Hardware Abstraction Layer

This chapter describes changes made to the prototype version of the RoFI HAL in both: the interface and the Universal module implementation. The RoFI HAL has been introduced in section 3.3.

7.1 Interface

The header file for the interface is in file `RoFI/softwareComponents/rofiHalInc/include/rofi_hal.hpp`.

The prototype version of the interface has all components as a *Proxy*. The Proxy object is meant to be an abstraction that does not hold the ownership of the underlying component and is meant to be lightweight, safely copyable and shareable.

The prototype version contains three main components — `RoFI`, `Joint` and `Connector`. The `RoFI` represents the RoFI module, this object does not have a constructor and is instead received with static methods `RoFI::getLocalRoFI()` or `RoFI::getRemoteRoFI(Id remoteId)`. Each module has its shape, consisting of a number of joints and connectors. The `RoFI` allows to get the proxy for the module's joints and connectors. The `Joint` proxy allows for both: controlling the joint and getting the joint information.

The connector was the only component that needed a change due to the LiDAR addition. The `Connector` proxy has the ability to extend and retract the connector, connect and disconnect powerlines (internal or external), react to connection and disconnection from the mating side and powerlines, and send and react to received packets. The part of the `Connector` that needed change was the `ConnectorState`. The prototype version of the `ConnectorState` contains information about the connector's position (extended or retracted), connection on powerlines and to the mating side, the orientation of the connection (North, East, South, West), and the voltage and current of the powerlines.

The state was extended by the measured distance, the measurement status, and setting and receiving distance modes. However, this is different from the LiDAR status from subsection 4.3.5. Whereas, the measured distance is just a 16-bit number with the measured distance in millimeters, the status is more complicated.

The status is a 2-bit number, with each value having assigned a specific meaning. The status values and their meanings are the following:

- `0b00` = Error — This signals an error, which can be an I^2C communication error, a LiDAR error, or that the device got unplugged while communicating.
- `0b01` = Not Measured — This status is when the LiDAR is not plugged in or the request was made before the first LiDAR measurement was completed.
- `0b10` = Out of Range — The measured data was measured; however, they **do not have to be valid**. Usually, this means that the data are outside of the measurable range; however, it can also be caused by wrong settings.
- `0b11` = Fully Valid — The measured distance is fully valid to use.

The value for the Error was chosen carefully to prevent false signaling of the measured state — if there is an error in communication with the RoFICoM, the read value is 0, and thus the LiDAR status is parsed as an Error. Additionally, the specific choice of values allows for the individual bits of the status to carry their own meaning, i.e., the higher bit acts as a *validity* bit (the value `1` means the measured data can be used). The lower bit acts as a *certainty* bit (the value of 1 means that the measured distance should be correct). Note that the bit can be used as a certainty bit only when the validity bit is set to valid. The LiDAR status is for explicitness represented as a `enum`.

The distance mode is represented as a 2-bit number, and the values correspond to the value of the distance modes for the LiDAR driver. The possible modes for the distance mode are autonomous (with the value of 0), short (1) and long (2). The last possible value (3) is currently unspecified and its usage is ignored. There was a newly added function to set the distance mode for the connector, which takes an `enum class` of the possible modes.

7.2 Implementation — esp32

Since the protocol has changed, the changes must be synchronized to all implementations. Not only does the protocol change need to

be replaced in all of the RoFI HAL implementations, but also in the firmware implementation. The change of firmware implementation that sends the LiDAR status, measured distance, and distance mode is described in subsection 4.3.5. This section describes changes made in the esp32 HAL implementation (for the Universal module). Other implementations are out of the scope of this thesis.

The prototype version has a lot of functionality implemented. Before mentioning the specific changes made, there is a short overview of how the esp32 HAL handles the connector-to-module communication. The `ConnectorBus` handles any command (such as to connect and disconnect) for the local connector. The connector schedules the command to be sent over the bus. The `ConnectorBus` for sending the commands runs on a parallel thread. The send is handled by the function `run()` with a specified command as an argument. After the command is sent and the response is received, it is returned to the connector under the function call `finish()`. The parsing of the response into the `ConnectorState` is done with the function `ConnectorStateImpl::from()`.

The changes to the esp32 implementation include new protocol flags — (LiDAR status) and distance modes. This is followed by parsing the status command response from the connector in the function `ConnectorStateImpl::from()`, that contains the LiDAR status, measured distance and distance mode. The function to set the distance mode sends the status command with flags set accordingly to the chosen value. As last, due to the change in the size of the command, the change needs to be updated in the `ConnectorBus::run()`, but also in the DMA buffer allocation of the `ConnectorBus`.

Those changes are enough to get the status and distance. However, the state is only updated when the status command is issued; this means that the state of status and distance does not represent the current value. Updating the state was solved with the poller. This poller runs in a separate thread and periodically asks the RoFICoM to update the state. There must be a slight delay between commands to allow the RoFICoM to apply the commands and send the response. The poller is done by the FreeRTOS function `xTaskCreate()`, which creates the new thread for polling and constantly issues the command status.

8 Conclusion

This thesis introduces changes and improvements made to the RoFICoM's firmware prototype, making it closer to the production-ready state.

These changes consists of distance measurement using the LiDAR and making abstraction around the specific VL53L1X for easier usage in the firmware. The VL53L1X required I^2C protocol, which was explained and implemented together with some family-specific code in the `stm32cxx` drivers.

The Board Support Package has been added to isolate board-specific code and allow easier transition in different RoFICoM's board versions.

Followed by smaller, but not less important additions to the RoFICoM's firmware, such as new motor driver, implementation for power-sharing and power measuring.

Lastly, this thesis described the update of connector-to-module communication to transfer the status of LiDAR's measurement and measured distance and allow for setting the LiDAR's distance mode. This change was added to the RoFICoM's firmware, RoFI HAL's interface and esp32 implementation for the Universal module.

Even though many functionality works as intended, such as the LiDAR measurement, its propagation to the Universal module's control unit, transition to new board version with Board Support Package, update of the connector-to-module protocol, there is still room for improvement.

Such improvement might be in the motor driver or in the power-sharing. Even though both are implemented, they do not work as intended. The motor movement occasionally moves the gearing beyond the specified position and the magnetic sensors, which prevents the motor from moving back. The power-sharing currently does not measure the correct values, although the formula is correct, and should be further inspect it to find the issue. Both are suspected to be hardware-related problem in the current board and mechanical construction revision and their rework is out of the scope of this thesis.

The final implementation can be found in the github repository of the RoFI project at the address: `https://github.com/paradise-fi/RoFI`. The snapshot of the whole RoFI project containing the changes

made during the development can be found in the thesis attachment. The appendix contains the procedures for building the relevant subprojects of the project.

A Building the Project

The attachment of this thesis contains the snapshot of the whole project that can be used to build and flash the result implementation.

At the root of the project, there is dockerfile, that is used as development environment for building and flashing the firmware.

The build environment is set up with the `setup.sh` script, it is invoked by `source setup.sh {build_mode}`, where the current values for the `build_mode` are: `Debug`, `Release`, `RelWithDebInfo`, `MinSizeRel`. For releasing the environment, use the command `teardown`. Before building the wanted sub-project, the wanted suite for compilation needs to be configuration via the command `rcfg suite [suite1...]`. After that, the sub-project can be build using the command `rmake {target}`. Finally, the firmware can be flashed using the command `rflash {target}`.

RoFiCoM

The suite for the RoFICoM is the `stm32`, and the target is `roficomFw`. After invoking the build environment with desired build mode, the suite can be configured with command `rcfg stm32`, then building the firmware with command `rmake roficomFw`, and finally, flashing the firmware to the connected RoFICoM through st-link device with `rflash roficomFw`.

HAL example

The suite for the Universal module is the `esp32`, and the target example is `hal_example_connector_status` for showcasing the implemented status command. The build environment for the Universal module must be invoked with the `-i` argument to download necessary dependencies — i.e., `source setup.sh -i {build_mode}`. After that, configure the suite with `rcfg esp32` and build the example with `rmake hal_example_connector_status`. The command `rflash hal_example_connector_status` is used to flash the program to the connected esp32.

Bibliography

1. *The RoFI platform* [online]. 2023. [visited on 2023-05-10]. Available from: https://ro.fi.muni.cz/.
2. MRÁZEK, Jan. *RoFI – Distributed Metamorphic Robots*. Brno, 2019. Available also from: https://is.muni.cz/th/y1s7e/. Diplomová práce. Masarykova univerzita, Fakulta informatiky. Supervised by Jiří BARNAT.
3. MRÁZEK, Jan; BARNAT, Jiří. RoFICoM – First Open-Hardware Connector for Metamorphic Robots. In: *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2019, pp. 2720–2725. Available from DOI: 10.1109/IROS40897.2019.8968167.
4. *Garmin on Mars.* [online]. Garmin Blog., 2021/2023 [visited on 2023-03-15]. Available from: https://www.garmin.com/en-US/blog/general/garmin-on-mars/.
5. *A new generation, long distance ranging Time-of-Flight sensor based on ST's FlightSense technology* [online]. STMicroelectronics, 2022. [visited on 2022-06-24]. No. vl53l1x. Available from: https://www.st.com/resource/en/datasheet/vl53l1x.pdf. Rev. 6.
6. *A guide to using the VL53L1X ultra lite driver* [online]. STMicroelectronics, 2021. [visited on 2022-09-21]. No. UM2510. Available from: https://www.st.com/resource/en/user_manual/um2510-a-guide-to-using-the-vl53l1x-ultra-lite-driver-stmicroelectronics.pdf. Rev. 6.