



# RAM

● ROBOTICS  
AND  
MECHATRONICS

## DESIGNING AN EMBEDDED SOFTWARE ARCHITECTURE FOR A MOBILE EDUCATION ROBOT WITH REAL-TIME CONTROL ON A RASPBERRY PI 4 WITH FPGA-BASED I/O

J.T. (Jasper) Vinkenvleugel

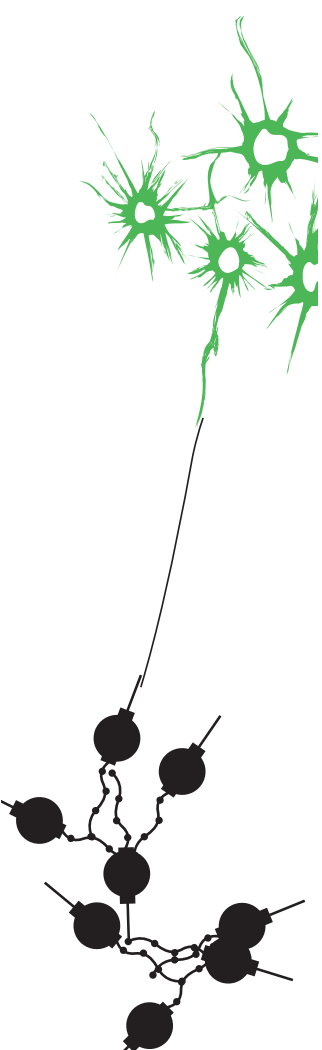
BSC ASSIGNMENT

**Committee:**

dr. ir. J.F. Broenink  
dr. ir. G. van Oort  
ing. M.H. Schwirtz  
dr. ir. M Ottavi

July, 2022

025RaM2022  
Robotics and Mechatronics  
EEMCS  
University of Twente  
P.O. Box 217  
7500 AE Enschede  
The Netherlands



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Requirements . . . . .	1
1.2	Overview . . . . .	2
<b>2</b>	<b>Background</b>	<b>3</b>
2.1	Hardware platform . . . . .	3
2.2	Real-time software . . . . .	3
<b>3</b>	<b>Analysis</b>	<b>5</b>
3.1	Real-time software . . . . .	5
3.2	Communication protocol . . . . .	5
<b>4</b>	<b>Design</b>	<b>10</b>
4.1	Hardware platform . . . . .	10
4.2	Real-time software . . . . .	10
4.3	Communication protocol . . . . .	10
4.4	Block-level overview . . . . .	12
<b>5</b>	<b>Validation</b>	<b>13</b>
5.1	Real-time software . . . . .	13
5.2	Communication protocol . . . . .	14
5.3	Hardware platform . . . . .	16
<b>6</b>	<b>Conclusion</b>	<b>19</b>
6.1	Conclusions . . . . .	19
6.2	Recommendations . . . . .	19
<b>A</b>	<b>Software</b>	<b>21</b>
A.1	Userspace . . . . .	21
A.2	Verilog . . . . .	22
<b>B</b>	<b>Toolchain</b>	<b>23</b>
B.1	icoBoard . . . . .	23
B.2	Raspberry Pi 4 . . . . .	23
	<b>Bibliography</b>	<b>24</b>

# 1 Introduction

Commercially available robotic platforms for education usually come with software systems such as ROS that hide real-time control from the user. For courses related to embedded control, a new platform has to be designed that enables students to work on a level much closer to the hardware, while also being cost effective and easy to expand based on different teaching requirements.

The complete project is divided in separate parts that have been executed in three different bachelor assignments: the bachelor assignments of Rob Binnenmars and Aron Boerkamp. In this bachelor project, a further look is taken at the software architecture of such a robotic platform, taking a Raspberry Pi 4 single-board computer and an icoBoard containing an FPGA, to expand on existing work to create a robust software architecture that can work with different types of sensors. This work is to be integrated in a mobile robot that can be used or adapted for educational purposes.

## 1.1 Requirements

For this project, several requirements have been given. Based on this, the requirements for the project have been drawn up. Some of these requirements have to do with the whole robotic platform, while some of the requirements have to do with only the part that are discussed in this report: the embedded software architecture.

### 1.1.1 Robotic platform

As stated before, the goal of the robot is to have an expandable robotic platform that can be used by students. This means that the robot needs to be used during practical sessions. The robot also need to be mobile: it needs to be able to move in a certain direction. The following are the requirements that have been set for the entire robotic platform:

- Rectangular base for ease of construction with approximately the size of A4 paper.
- Two wheels with encoders.
- Motors that can achieve a velocity of approximately 0.3 m/s.
- Two casters for stability of the robot.
- A weight of around 10 kg.
- Rechargeable batteries that can last for at least one lab session.
- The ability to expand the robot with different extra sensors.
- A Raspberry Pi 4 as a main computing unit with an FPGA-based I/O board in the form of an icoBoard.

The construction of this robot is mostly executed in the bachelor thesis project of Rob Binnenmars, which is done in parallel to this project. Both projects work on independent parts of the overall project: the division is made between the embedded software and the robot itself. Next, the requirements of the embedded software are discussed.

### 1.1.2 Embedded software

This project focuses on the embedded software architecture, which has a different set of requirements that expands the previously created list:

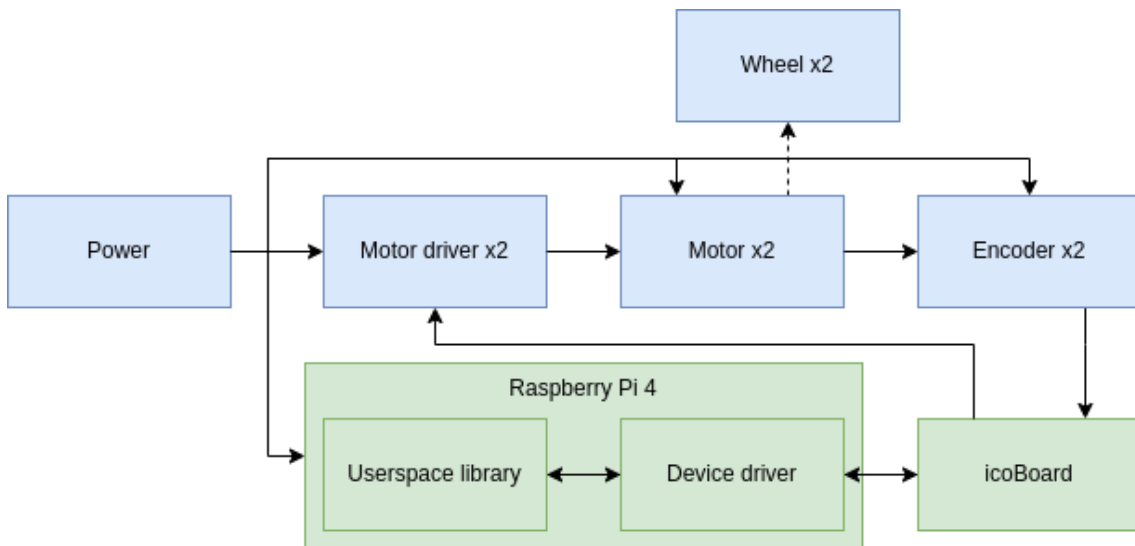
- A Raspberry Pi 4 as a main computing unit with an FPGA-based I/O board in the form of an icoBoard (this requirement was given at the start of this project)

- A software architecture that can easily be expanded based on different needs and the user has access to the complete software stack.
- Precise real-time behaviour with as little latency as possible between the Raspberry Pi 4 and the FPGA.
- The ability to add extra hardware parts on request without having to completely rewrite the software.

## 1.2 Overview

The design objectives of the project are then: to design an embedded software architecture for an education robot that is expandable in both software and hardware features, provides open access to the complete software stack and utilizes a Raspberry Pi 4 and an icoBoard. The device requires real-time behaviour to work with time constraints and has to be able to be easily incorporated in the work of Rob Binnenmars.

In Figure 1.1 a complete block-level overview of the robotic platform is shown. In this figure, the blue pieces cover the part of the device that is purely hardware while the green pieces cover the part of the device that relates to the embedded software architecture.



**Figure 1.1:** Block-level overview of the robotic platform, green is covered in this project.

In Chapter 2, background information on the discussed topics is given, in Chapter 3 analysis is done on the different aspects of the software architecture and then design decisions are described in Chapter 4 to implement an architecture which corresponds with these requirements. After this, the work that was executed is validated according to the requirements that have been set up. A conclusion is drawn up in Chapter 6 and further recommendations are discussed.

---

## 2 Background

In this chapter necessary background information for reading the report is described. In Section 2.1 the used hardware platform is described and in Section 2.2 a brief explanation of real-time software is given.

### 2.1 Hardware platform

The goal of the complete project is to create a mobile education robot, but in this report the focus is on creating an embedded software architecture. For this, a Raspberry Pi 4 (Raspberry Pi, 2022a) and an icoBoard (icoBoard, 2022) are used as the main embedded computing units. The Raspberry Pi 4 is an Arm-based single-board computer with the following specifications:

- **CPU:** Broadcom BCM2711 (4x Cortex-A72 at 1.5 GHz)
- **RAM:** 8 GB LPDDR4-3200 SDRAM
- **I/O:** USB 2.0 and USB 3.0, ethernet, 40-pin GPIO header

Besides this, the board is relatively low cost, it has a large developer and enthusiast community and Linux is supported very well out of the box. The BCM2711 provides a reasonable amount of processing power for running a real-time control loop. These factors, and the fact that this particular single-board computer is already being used for other projects at the Robotics and Mechatronics research group (RaM), means that this hardware platform was chosen before the start of the project.

The second piece of hardware that is used is an FPGA-based I/O board in the form of an icoBoard. This board is used for connecting to peripheral hardware such as encoders, motors and sensors. The icoBoard is distributed in the form of a "HAT", which means it can be mounted on top of the Raspberry Pi 4. The icoBoard has the following specifications:

- **FPGA:** Lattice iCE40-HX8K (8k LUT, 100 MHz clock)
- **RAM:** 8 Mbit SRAM
- **I/O:** Up to 200 GPIO pins or 20 PMOD connector modules

Verilog (Wikipedia Authors, 2022) is used to program this FPGA, and a fully open-source toolchain is available in the form of Yosys (an open source synthesis suite (YosysHQ, 2022c)), nexprn (an open source place and route tool that is the successor of Arachne-pnr (YosysHQ, 2022a)) and Icestorm (an open source project that works on the iCE family of FPGAs from Lattice (YosysHQ, 2022b)).

By combining these two pieces a relatively powerful platform with a sizable amount of I/O is available for this education robot.

### 2.2 Real-time software

On the previously described hardware platform, a real-time operating system is used. In embedded systems, it is often critical to execute tasks within a certain time constraint. Normal operating systems such as Linux do not explicitly guarantee being able to handle these time constraints and are as such not real-time, so either a dedicated real-time operating system (RTOS) needs to be used, or the Linux-kernel needs to be modified in such a way that it is able to handle time constraints.

In the case of using Linux several different options exist: the PREEMPT\_RT-patch for the Linux kernel (described in McKenney (2005)) or Xenomai Authors (2022), which provides a real-time core called Cobalt which uses a separate, high-priority execution stage (I-pipe) that is added

to the Linux kernel. According to Meijer (2021b) out of these two options, the latter has much better latency behaviour than the former. Previous work done by (Hofstede, 2022) on adding a real-time communication protocol between the Raspberry Pi 4 and icoBoard based on SPI also used Xenomai. This work included an SPI driver for the Linux kernel based on RTDM (an API provided as part of Xenomai for real-time device drivers) adapted from Schurando (2016). This driver was made compatible with the Raspberry Pi 4. The work done by Hofstede (2022) also includes a userspace demonstration program on the Raspberry Pi 4 and a SPI peripheral written in Verilog.

Xenomai provides so-called skins on top of the Cobalt core. These skins allow software that was written for a different RTOSs such as VxWorks to also work on Xenomai. Cobalt also provides a skin that closely resembles a subset of familiar POSIX functions that are available on Linux. This POSIX-compatibility enables comparisons between real-time and non real-time behaviour, as threads can either be scheduled in the high-priority execution stage of I-pipe, or not, using the same mechanisms.

With these patches, it is possible for an operating system to schedule processes in such a way that they can be executed as quickly as possible, minimizing having to wait for any other processes to ensure they are executed within the set time constraints.

---

## 3 Analysis

In this chapter, analysis is done to determine the different options that can be used to achieve the goals that were stated in Chapter 1. In Section 3.1 the real-time software on the Raspberry Pi 4 is discussed and in Section 3.2 different strategies for communication between the Raspberry Pi 4 and icoBoard are discussed.

### 3.1 Real-time software

#### 3.1.1 Kernel space

As discussed in Section 2.2, two different options exist for real-time software on top of the Linux kernel: `PREEMPT_RT` and Xenomai. As was already described, Xenomai offers lower latency combined with the same POSIX interface whereas `PREEMPT_RT` provides a kernel that very closely resembles a vanilla Linux kernel without modifications.

A disadvantage of Xenomai is that it does not have very good support for different hardware platforms. While the userspace part is supported on Arm, a custom kernel build is required for the Raspberry Pi 4. A patch with I-pipe for the Raspberry Pi 4 is provided by (Tam, 2019). Even though Xenomai is supported on 64-bit devices such as the Raspberry Pi 4, this patch was created for the 32-bit version of Linux 4.19. According to the Raspberry Pi 4 documentation, users of a 32-bit kernel on a device with 8GB RAM can address the total amount of RAM using Physical Address Extension (*PAE*) but processes are limited to 3GB of RAM. This should not be a problem in most control loops, but needs to be taken into account for other tasks that need to be executed on the Raspberry Pi 4. A problem with using an older kernel version is that newer developments, such as Dovetail as a replacement for I-pipe, cannot easily be incorporated.

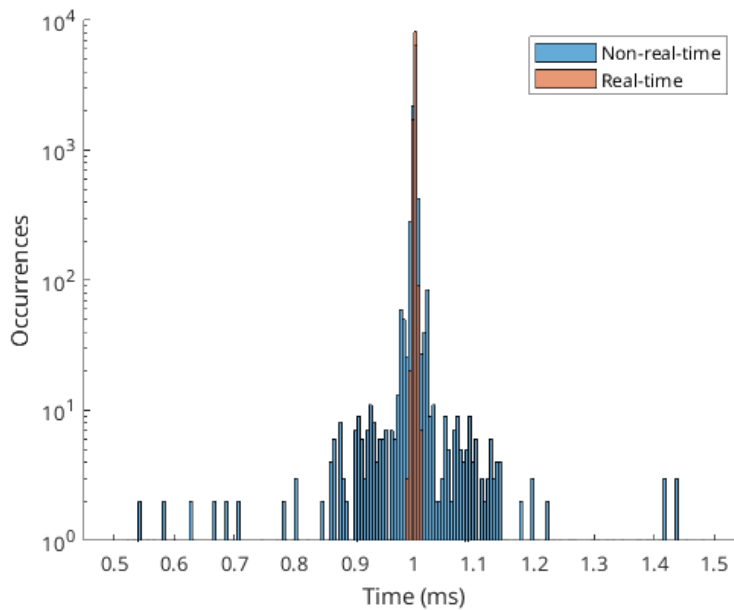
A different concern with using Xenomai is the fact that drivers need to be rewritten to take advantage of the real-time capabilities of the kernel. Section 2.2 already describes a driver that uses Xenomai's RTDM API for real-time SPI, which can be re-used.

#### 3.1.2 User space

In case of `PREEMPT_RT`, no changes to userspace have to be made. Existing code is automatically executed with lower latency. With Xenomai, code has to be adapted to run on the different skins that are described in Section 2.2. With the POSIX-skin, a subset of the features provided by POSIX is available, which means programs need minor adaptations to be able to run in real-time on a Xenomai-enabled kernel.

A question that might be asked is: does such a control loop even need a real-time operating system? As stated in Section 2.2, this can easily be tested by using Xenomai to schedule a normal and a real-time thread and making a comparison. A timer is created with an interval of 1 ms. Taking 10000 samples, the actual interval time is shown in a logarithmic histogram in Figure 3.1. It is clear that a real-time thread results in much less spread of interval times: they are centered very steeply around 1 ms. Taking larger sample counts, it is also not uncommon for non-real-time interval times to be 10 times as long as they actually should be. Something that does not happen for real-time samples. Having a sample time of 10 ms would mean that 9 times the sensors should have been sampled but are not, putting the robot in a potentially unstable state.

### 3.2 Communication protocol



**Figure 3.1:** Comparison between real-time and non-real-time behaviour

### 3.2.1 Protocol options

To make sure the Raspberry Pi 4 can use all of the different peripherals such as PWM drivers and encoders on the icoBoard, communication needs to exist between the two different devices. For the communication protocol, different options exist. Available controller-to-peripheral protocols include:

- **SPI:** a simple protocol that requires at least three wires (SCK (the serial clock), POCI and PICO<sup>1</sup>) as well as an optional CS (chip select) wire for selecting the correct peripheral device. Controller and peripheral can simultaneously transmit and receive when the controller activates CS and SCK (the serial clock), and bit transmissions happen on the rising or falling edge of the clock.
- **I2C:** this protocol only requires two wires, but devices need to present a hardware address for identification. This means it has more overhead than SPI, which means it is slower. It is also a less simple protocol than SPI.
- **SMI:** a parallel interface that is available on the Raspberry Pi 4. Due to its parallelism, it is fast, but it lacks documentation and available real-time drivers.

### 3.2.2 Communication strategies

In this section, a further look is taken at SPI because of the fact that a driver is already available for this protocol (Schurando, 2016). The simplicity of the SPI protocol means that there are a lot of different options for transferring data over the bus. In the existing work that was done by Hofstede (2022), for every port (a PMOD connector with one degree of freedom, so one PWM driver and one encoder) a separate command needs to be sent. Usually though, it happens that all data is needed every interval for the controller to do its work properly (all sensors need to be sampled at very regular intervals). This means that sending a different command per port adds overhead that can be prevented by making the command longer and adding more data to the command. These two different command structures are shown here, where "opcode" means the command that is sent to the FPGA:

<sup>1</sup>POCI: peripheral output, controller input. PICO: peripheral input, controller output. These were formerly called MISO and MOSI, respectively (OSHOWA, 2022).



Per port:	Opcode	Port number	Data		
All ports:	Opcode	Data port 1	Data port 2	...	Data port N

**Table 3.1:** Different communication strategies

If it is said that the opcode and port number are both contained in one byte, this means that the overhead of sending a command per port is the amount of ports times two bytes, while the overhead of sending a command to all ports is just one byte, with the notion that it must indeed be useful to send data to all ports at once. Additionally, extra time overhead comes from the driver that needs to send the command from the userspace to the icoBoard.

### 3.2.3 Controlling strategies

There are several different ways of handling the communication. In case of SPI, one of the devices acts as the controller and one of the devices acts as the peripheral. Though the SoC on the Raspberry Pi 4 does formally support working as a SPI peripheral, this option is undocumented and no drivers exist. It would thus be logical to use the Raspberry Pi 4 as the SPI controller. Then, the question is raised which of the devices should be the clock controller or reference clock: the device that handles sending signals for periodic sampling of the sensors.

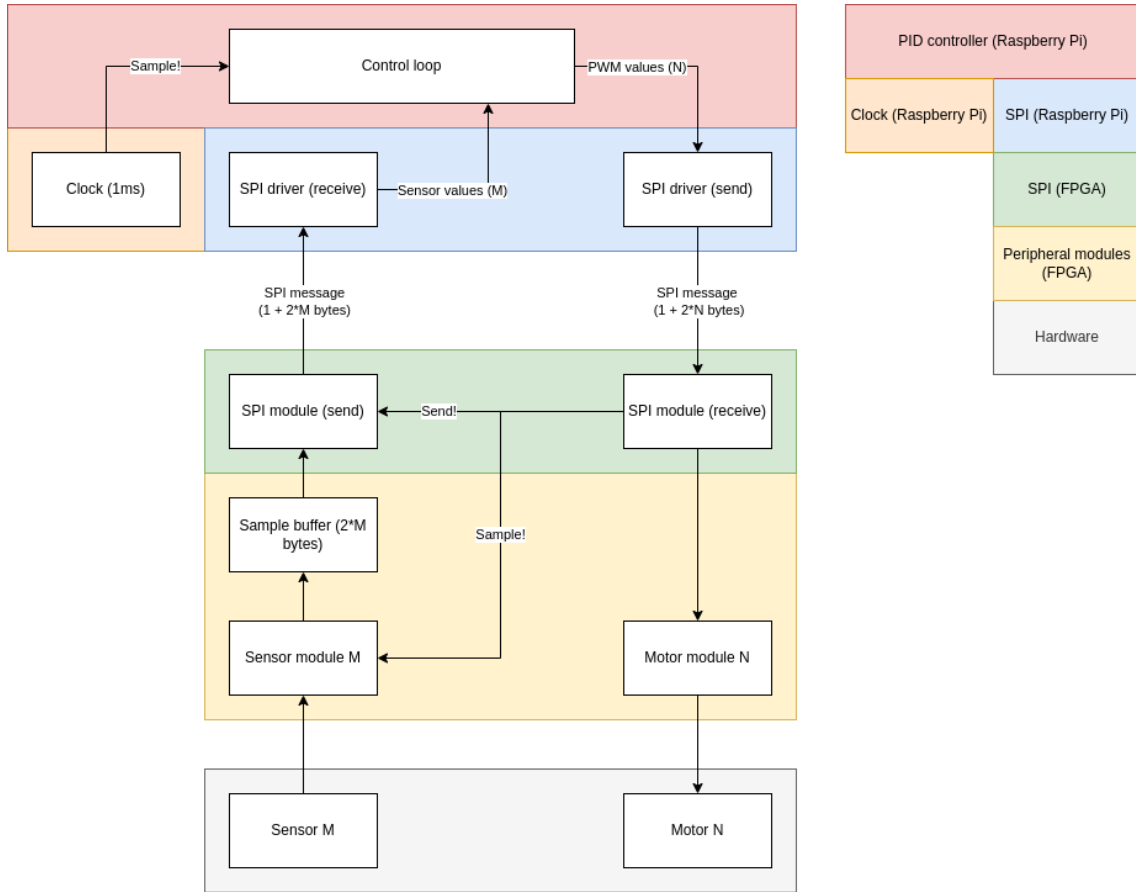
- **Method 1:** the Raspberry Pi 4 is the clock controller. A real-time clock is created as was previously done in Section 3.1, at every interval an SPI message is sent to the FPGA which subsequently samples the sensors, ready to return the data to the Raspberry Pi 4 on the next interval. An advantage is that a timer is easy to implement in software and the Raspberry Pi 4 is in control of the complete chain. A disadvantage is the fact that the Raspberry Pi 4 has more jitter in its clock than the FPGA has.
- **Method 2:** the FPGA is the clock controller. A clock is created in hardware (Verilog), which then raises a GPIO pin after every interval. This enables the Raspberry Pi 4, which starts the SPI transmission. An advantage is the fact that the clock is much more precise than the software clock on the Raspberry Pi 4. A disadvantage is the fact that it adds extra complexity to the Verilog code on the FPGA.

Block diagram overviews of both of these methods for the whole system are shown in Figure 3.2 and Figure 3.3, which includes the controller, the SPI sender and receiver for both devices and the modules that deal with connecting to the hardware peripherals. In this block diagrams, the assumption is made that the SPI message data length per peripheral is two bytes, but this might not be true in a real-world situation.

From these methods it is possible to come up with a diagram showing a timeline of what an SPI command should do, from the starting phase going into execution of the control loop. These are shown in Figure 3.4 for having the Raspberry Pi 4 as a clock controller and Figure 3.5 for having the FPGA as a clock controller. In this figure, the colors of the different parts correspond to the colors that are defined in in Figure 3.2 and Figure 3.3. In this figure, the delays for the individual actions are only displayed for illustrative purposes and do not reflect any real-world measurements. The figure also includes the initialization step that will be discussed at a later stage. Periodical work starts from the third SPI transmission, so the third blue and green blocks. The dotted arrow between the two rows indicates that a data transfer has occurred, and in which way the data has been sent.

### 3.2.4 Sample-to-action delay

In Figure 3.4 and Figure 3.5, the time it takes for a sensor to be sampled until a PWM value for this sensor is returned is indicated. Note that, because the FPGA receives a command to sample the sensors every 1 ms, two of these sample-to-action cycles can be active at a given



**Figure 3.2:** Raspberry Pi 4 as the clock controller

moment: during the PWM delay of the previous cycle, the sensor delay of the next cycle has already started. Even though the length of the segments do not reflect measured values, from this it can be determined that the following holds for the sample-to-action delay  $T_{\text{total}}$  of the two different methods for clock controllers:

$$T_{\text{total,raspberry}} = 2T_{\text{period}} - T_{\text{sample}} \quad (3.1)$$

$$T_{\text{total,fpga}} = T_{\text{period}} + T_{\text{send}} + T_{\text{gpio}} \quad (3.2)$$

In these equations,  $T_{\text{total}}$  is the time it takes from the moment a sensor was sampled until the calculated PWM value for this sensor is returned to the FPGA. If both of these equations are combined, it can be said that method 1 (so having the Raspberry Pi 4 as a clock controller) is the faster of the two options in terms of time from sample until return of a PWM value if the following holds:

$$T_{\text{send}} + T_{\text{gpio}} > T_{\text{period}} - T_{\text{sample}} \quad (3.3)$$

In this equation,  $T_{\text{send}}$  depends on the frequency of the SPI clock,  $T_{\text{gpio}}$  and  $T_{\text{sample}}$  depend on the frequency of the FPGA clock and  $T_{\text{period}}$  depends on the frequency of the timer. Because in general it can be said that the frequency of the FPGA clock (100 MHz, so 10 ns) is much higher than the SPI clock (31 MHz, so 32.25 ns) which is much higher than the frequency of the timer (1 kHz, so 1 ms), this gives the impression that it is likely that out of these two methods having the FPGA as a clock controller gives the least amount sample-to-action delay.

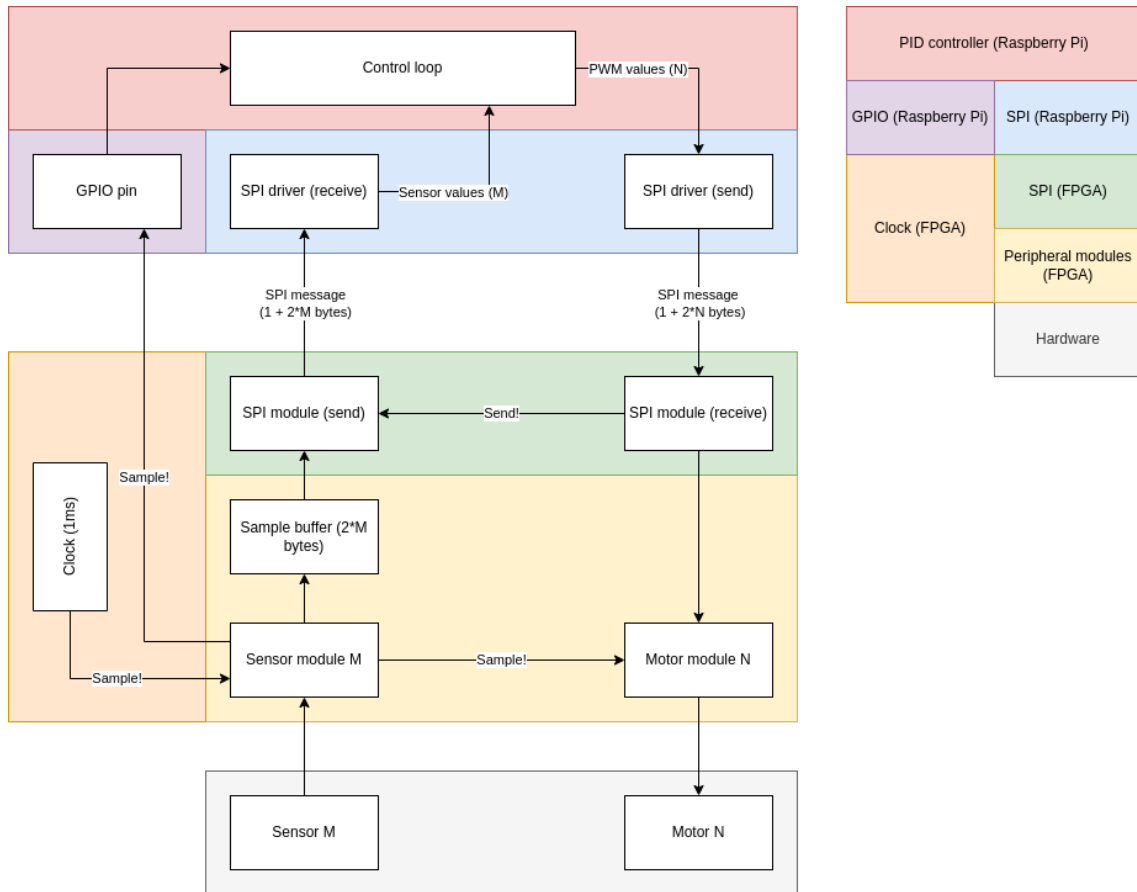


Figure 3.3: FPGA as the clock controller

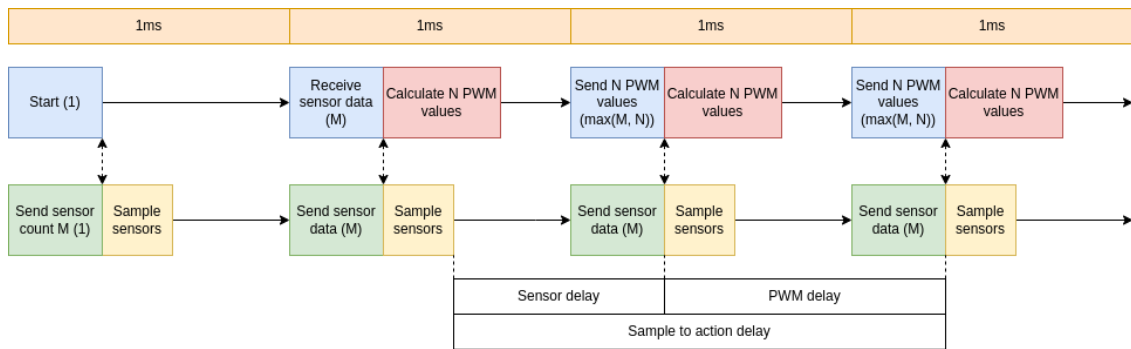


Figure 3.4: SPI communication timeline starting from initialization with the Raspberry Pi 4 as the clock controller.

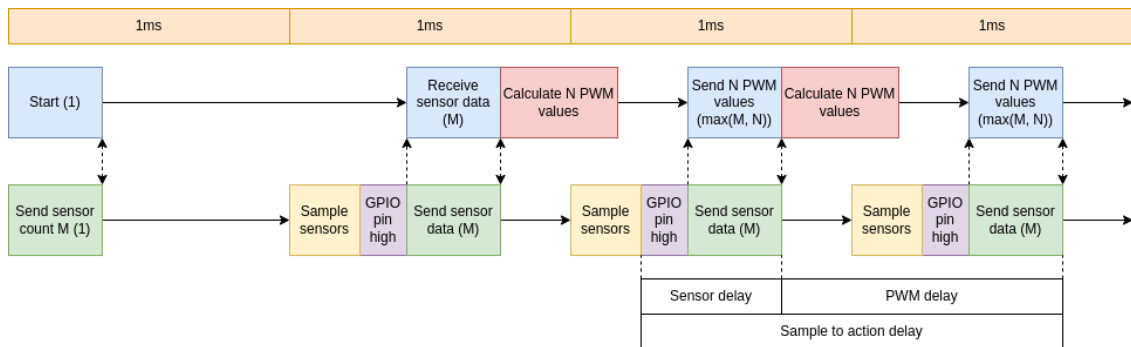


Figure 3.5: SPI communication timeline starting from initialization with the FPGA as the clock controller.

## 4 Design

In this chapter the information from Chapter 2 and Chapter 3 is combined to design a robotic platform that adheres to the requirements that were set in Chapter 1. First, the hardware platform is discussed in Section 4.1, then the software is discussed in Section 4.2, the communication protocol implementation is discussed in Section A.1 and at the end a block-level overview is presented in Section 4.4.

### 4.1 Hardware platform

In Chapter 1 the overall requirements of the robotic platform have been discussed. A Raspberry Pi 4 is combined with an icoBoard, connecting the FPGA to the GPIO pins on the Raspberry Pi 4. The actual physical construction of the mobile robot is described in the bachelor assignment of Rob Binnenmars (Binnenmars, 2022).

### 4.2 Real-time software

From the analysis in Chapter 3, several different options for the real-time software stack are available:

- **Option 1:** a Linux kernel that has been patched with PREEEMPT\_RT and a standard POSIX userspace. Advantages: supported upstream, does not require any modifications to the userspace software, has drivers available for several communication protocols. Disadvantages: slower than the other available options.
- **Option 2:** a Linux kernel that has been patched with Xenomai and a POSIX skin on top of the Cobalt core. Advantages: has the lowest latency, a driver is available for SPI. Disadvantages: requires a modified kernel that is not supported upstream, needs modifications to the userspace for the provided subset of POSIX, needs new drivers for other communication protocols.
- **Option 3:** no modifications to the Linux kernel and a standard POSIX userspace. Advantages: no requirements to the kernel and userspace are necessary, all usual tools are available. Disadvantages: very high latency in scheduled threads.

From these three different options, it is clear that the last is not a good option as the latency is not low enough for real-time behaviour, as described in Section 3.1. This leaves the first and second options as viable alternatives. In this case the lower latency of the Xenomai kernel and the fact that an SPI driver already exists makes it the most compelling option.

The choice to go for the POSIX skin for Cobalt comes from the fact that POSIX is documented very well and it enables the option to schedule regular non real-time threads from within the same software. For the validation in Chapter 5 a userspace program is written in C that uses the available subset of POSIX that is available in Cobalt. A further explanation of this program can be found in Appendix A.

### 4.3 Communication protocol

#### 4.3.1 Data transmission

The previous discussion in Section 4.2 already hints at the choice being SPI, and the advantages and disadvantages that have been listed in Section 3.2 support this decision. The overhead of I2C is much higher than SPI leading to slower speed, and the more high-speed option (SMI) has poor driver and documentation availability. SPI also has the simplest implementation of these protocols, the user is free to implement whatever communication protocol on top of the bus. This is why the decision is made to use SPI.

In previous chapters it has been concluded that sending all information in a single transmission greatly reduces overhead, so this is the approach that is taken. Using this protocol in a way similar to what has been visualised in Figure 3.4, several different operation codes are necessary. The following operation codes need to be implemented on both the Raspberry Pi 4 and the icoBoard:

Code	Value	Description
OP0 / SPI_EMPTY	0	Empty command: does nothing.
OP1 / SPI_START	1	Start command: tells the FPGA to start execution.
OP2 / SPI_RECEIVE	2	Receive command: receives sensor data from the FPGA, transmits nothing.
OP3 / SPI_TRANSMIT	3	Transmit command: simultaneously receives sensor data from the FPGA and transmits PWM data to the FPGA.
OP4 / SPI_EXIT	4	Exit command: stops execution on the FPGA and resets the PWM devices (duty cycle is 0).
OP5 / SPI_RESET	5	Reset command: resets the PWM devices and encoder.

**Table 4.1:** Operation codes for the SPI protocol.

In the next table, a regular SPI transmission can be shown, based on Figure 3.4:

Count	Bytes	Raspberry Pi 4	icoBoard
1	1	OP1 (start)	Sensor count ( $M$ )
2	$1 + B * M$	OP2 (receive)	OP3 (transmit) + data
3	$1 + B * \max(M, N)$	OP3 (transmit) + data	OP3 (transmit) + data
...	...	...	...
End - 1	$1 + B * \max(M, N)$	OP3 (transmit) + data	OP3 (transmit) + data
End	1	OP4 (exit)	OP3 (transmit)

**Table 4.2:** Operation codes for a regular SPI transmission

In this table, count means the operation number from the start, and the columns below Raspberry Pi 4 and icoBoard describe what has been transmitted by each of these boards over the SPI bus. The amount of sensors connected to the icoBoard is denoted as  $M$ , the amount of PWM devices that are connected to the icoBoard is denoted as  $N$ . The amount of bytes that each sensor gets is denoted as  $B$ . As is shown in the table, the last response of the icoBoard is cut, because the Raspberry Pi 4 is sending the exit operation code.

With the robot that is designed in the project of Rob Binnenmars (Binnenmars, 2022), two sensors are connected in the form of the two encoders of the wheels of the robot, and two different PWM devices are connected in the form of the motor drivers. This means that in the previous paragraph and table, both  $M$  and  $N$  are equal to 2, meaning that the largest length of an SPI command for this particular setup will be 5. Of course, this can be extended as extra peripheral devices are added.  $B$ , the amount of bytes necessary per peripheral for sending the data, is dependent on the connected peripherals. This is set to be 2, which gives 65536 possible values.

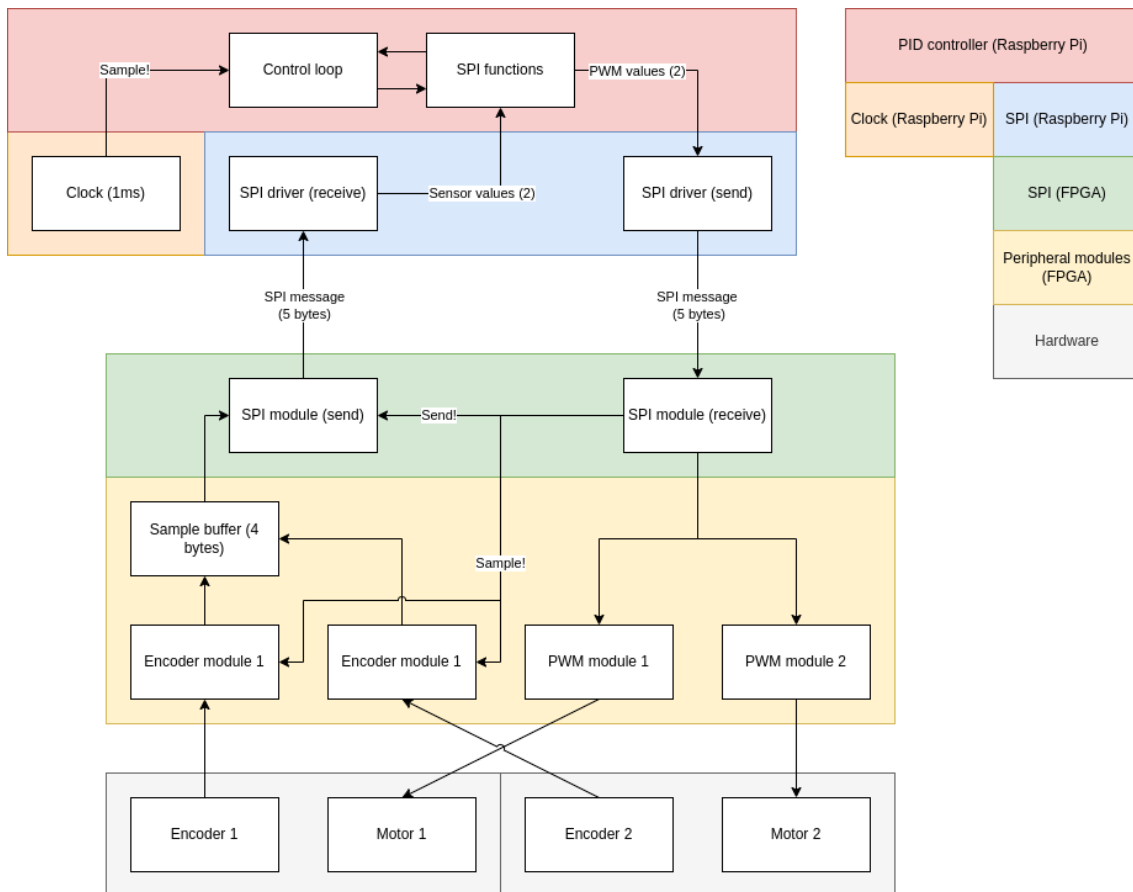
### 4.3.2 Clock controller

The next design discussion involves the decision whether to make the Raspberry Pi 4 or the icoBoard the clock controller of the software architecture. According to the discussion in Section 3.2 having the icoBoard be the clock controller would be the preferable option, as this would theoretically have the lowest sample-to-action delay combined with the fact that the

clock of the icoBoard should be more consistent than the clock on the Raspberry Pi 4. However, in practice, the architecture of the software is made a lot easier if the Raspberry Pi 4 is the clock master as there is no need of writing extra Verilog for the timer and having to handle interrupts on a pin going high in the userspace software. It can also be argued that the consistency of the clock on the Raspberry Pi 4 is good enough (see Figure 3.1). This is why the decision is made to have the Raspberry Pi 4 act as the clock controller, and a real-time timer is implemented in the userspace program.

#### 4.4 Block-level overview

Taking the decisions that are made in this chapter, a revised version of the block diagram that is shown in Section 3.2 can be made. This block diagram is shown in Figure 4.1. In this diagram, the actual connected peripheral devices are shown: two motors with two encoders, with their respective SPI command lengths. The control loop is split up according to Appendix A. This diagram also shows the Raspberry Pi 4 as the clock controller and integrates the SPI protocol visualized in Figure 3.4.



**Figure 4.1:** Block level overview of the complete system

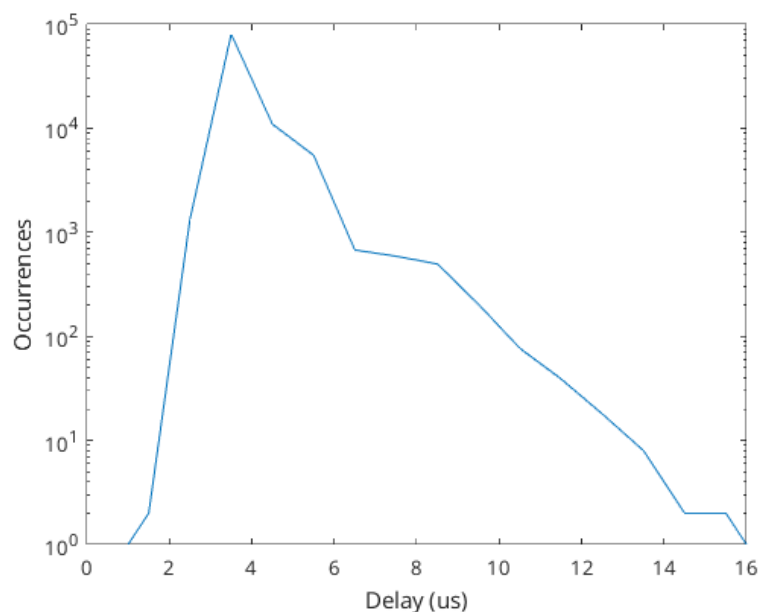
## 5 Validation

In this chapter, the design that is described in Chapter 4 is implemented and then validated with the requirements that were set in Chapter 1. First, a look is taken at the software in Section 5.1, then a look is taken at the communication protocol in Section 5.2 and at the end a look is taken at the complete education robot in Section 5.3.

### 5.1 Real-time software

#### 5.1.1 Xenomai validation

First, a Raspberry Pi 4 with Xenomai installed on it is taken. This includes the I-pipe high-priority execution stage on the Linux kernel as well as the userspace tools. The userspace tools from Xenomai provide a piece of software that can be used to test the real-time behaviour (latency, as described in Appendix B). In Figure 5.1 a graph from this tool is shown that displays the latency in  $\mu\text{s}$  of a periodic user-mode task with a period of 1ms. This test ran for 100 s. From this graph it is shown that the latency peak is between  $3\ \mu\text{s}$  and  $4\ \mu\text{s}$ , meaning that a period of  $1000\ \mu\text{s}$  usually takes  $3\ \mu\text{s}$  to  $4\ \mu\text{s}$  longer than  $1000\ \mu\text{s}$ . Within this testing period, no samples were over time (or, longer than 1 ms) which is reflected in the graph.



**Figure 5.1:** Latency test from Xenomai

These results differ from the results that were found in Figure 3.1: while in the previous graph some period times are shown to be slightly lower than 1 ms, this graph shows all period times to be slightly higher than 1 ms. An explanation can be that this tool has a different execution priority or other scheduling-related differences, or that it implements a timer in a slightly different way. It is also possible that the current time is recorded differently.

#### 5.1.2 Userspace program

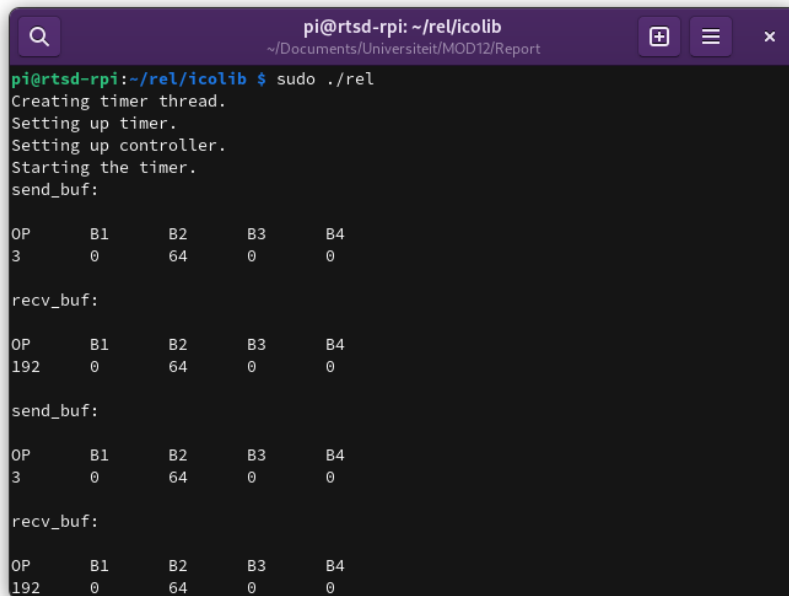
After this, work is started on the new implementation of the userspace software. This software consists of a userspace timer, a framework in which a controller can be implemented that also implements the SPI protocol that is designed in Section A.1 and functions that can be used to write to the SPI device. A further description of this program is shown in Appendix A. A timer

is implemented that runs at 1000 Hz and writes the time each sample takes to a CSV file for verification. This validation is shown in Figure 3.1 in Chapter 3. Every time the timer fires a function is executed that runs the control loop. Within the control loop the communication with the icoBoard is handled.

## 5.2 Communication protocol

### 5.2.1 Setup validation

In the userspace program the communication with the SPI driver is implemented. For this to work the required `ioctl`<sup>1</sup> settings from Hofstede (2022) are taken. Initially, to ensure the communication between the Raspberry Pi 4 and icoBoard is working, code is written to enable LEDs on the icoBoard with a command that is sent from the userspace program. To verify whether returning a value over SPI works, a value is sent using the Raspberry Pi and then the same value is returned by the icoBoard. This is shown in Figure 5.2, where `send_buf` is the buffer that is sent to the icoBoard and `recv_buf` is the buffer that is returned from the icoBoard. Here, `OP` is the operation code and `B2` is the byte that was set and thus the byte that should be equal, which is indeed the case.



```

pi@rtsd-rpi: ~/rel/icolib
~/Documents/Universiteit/MOD12/Report
pi@rtsd-rpi:~/rel/icolib $ sudo ./rel
Creating timer thread.
Setting up timer.
Setting up controller.
Starting the timer.
send_buf:
OP      B1      B2      B3      B4
3       0      64      0       0

recv_buf:
OP      B1      B2      B3      B4
192    0      64      0       0

send_buf:
OP      B1      B2      B3      B4
3       0      64      0       0

recv_buf:
OP      B1      B2      B3      B4
192    0      64      0       0

```

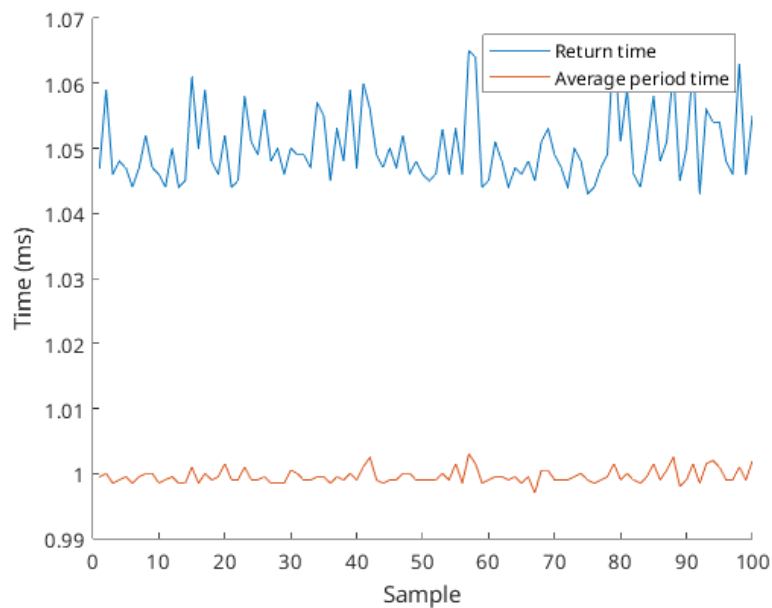
Figure 5.2: Sending and returning the same value over SPI

### 5.2.2 Testing the round trip latency

To test the round trip latency of the SPI protocol without peripheral hardware, an SPI command of 8 bytes is sent to the icoBoard and the same command is then sent back to the Raspberry Pi 4 on the next period of the timer. For this test the SPI clock speed is lowered from 31 MHz, the frequency the other tests have been executed at, to 2 MHz to be able to record the values of the pins using the logic analyzer. The time is recorded before the message is sent and then the time is recorded after the correct return message is received on the Raspberry Pi 4. While this is done, the time period of the timer is also recorded and averaged. Doing this 100 times results in the graph that is shown in Figure 5.3.

<sup>1</sup>A Linux system call for setting the correct options for opening a file, in this case to set the SPI polarity, phase and serial clock frequency.





**Figure 5.3:** Latency test for 100 samples (plot)

A histogram of the same data is shown in Figure 5.4. From this graph, it becomes clear that the average time spent sending and returning the same message is higher than the average sample time: this difference is the time that is actually spent on the SPI transmission. These data points also have a larger spread.

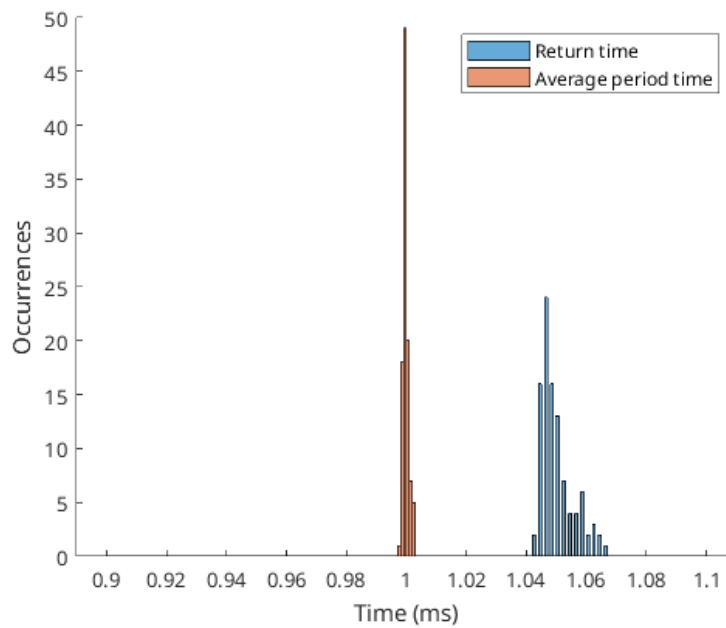
### 5.2.3 Determining the SPI transmission duration

From the recorded data in the previous section, the average difference in time between the average period time and the round trip time is  $51 \mu\text{s}$  with a standard deviation of  $4.8 \mu\text{s}$ . This means that in practice when the Raspberry Pi 4 requests data from the sensors on the icoBoard, the data is available for processing one period (approximately 1 ms) plus  $51 \mu\text{s}$  later. Taking into account the duration of the next period, this leaves 0.949 ms for any calculations that need to be done in the control loop.

To verify whether this corresponds to the actual time that is spent on the SPI communication, the logic analyzer can be used to get the values of the pins during this SPI cycle. This timing diagram is shown in Figure 5.5. In this particular recording the time spent sending the 8 bytes over SPI is approximately  $45 \mu\text{s}$ , which is slightly lower than the average that was found from calculating the time from the userspace software. A possible explanation of this extra delay is the overhead of the kernel module.

### 5.2.4 Implementing the protocol

The protocol that is described in Section A.1 is taken and implemented in the userspace program and the Verilog code running on the icoBoard. Unfortunately, currently the communication protocol as described in Section A.1 and shown in Figure 3.4 cannot be implemented directly. This is because the Verilog code on the FPGA cannot immediately send the newly-sampled sensor data: it can only fill a buffer and set a flag such that it is sent over the bus in the next communication cycle. This means that the sample-to-action delay is 1ms longer than it should be according to the design. This, and a suggestion to solve this problem, is further discussed in Section 6.2.



**Figure 5.4:** Latency test for 100 samples (histogram)

### 5.3 Hardware platform

In this section various hardware connection features are validated. Because the complete robotic platform is to be integrated in the form of a mobile robot that can drive using two motors, the features that are chosen for this validation are reading an encoder and setting a pulse-width modulation (PWM) signal.

#### 5.3.1 Encoder validation

For testing purposes, an HEDM-5540 encoder (Avago Technologies, 2014) is connected to the icoBoard. The C and Verilog code are set up to count the encoder pulses on the icoBoard and then directly send the value of the encoder over the SPI bus. When the shaft connected to the encoder is turned, the resulting encoder count is written to the terminal. Doing this for approximately a quarter of a rotation results in the plot that is shown in Figure 5.6.

In this plot, it is shown that the maximum counter value is  $2^{16} - 1$ , which is the expected value for 2 bytes. It is also shown that the counted amount of pulses is much higher than the amount of pulses the encoder should give per rotation according to the datasheet. This means that even though the shape of the graph somewhat resembles the expected values from an encoder (gradual decline because of the counterclockwise rotation, overflow at 0 and 65535), the actual numerical output does not: more than one rotation is counted while in reality only a quarter of a rotation has been done.

The expected output is a line that starts at 0, then immediately underflows to the maximum value because of the counterclockwise rotation and has a downward slope. To enable a simple form of localization, this complete rotation should fit multiple times in the maximum value, meaning that no new underflow should occur in this quarter of a rotation. This means that in the current implementation either the output values of the pins of the encoder do not correspond with the expected values in the Verilog code, or the counter is running too fast.

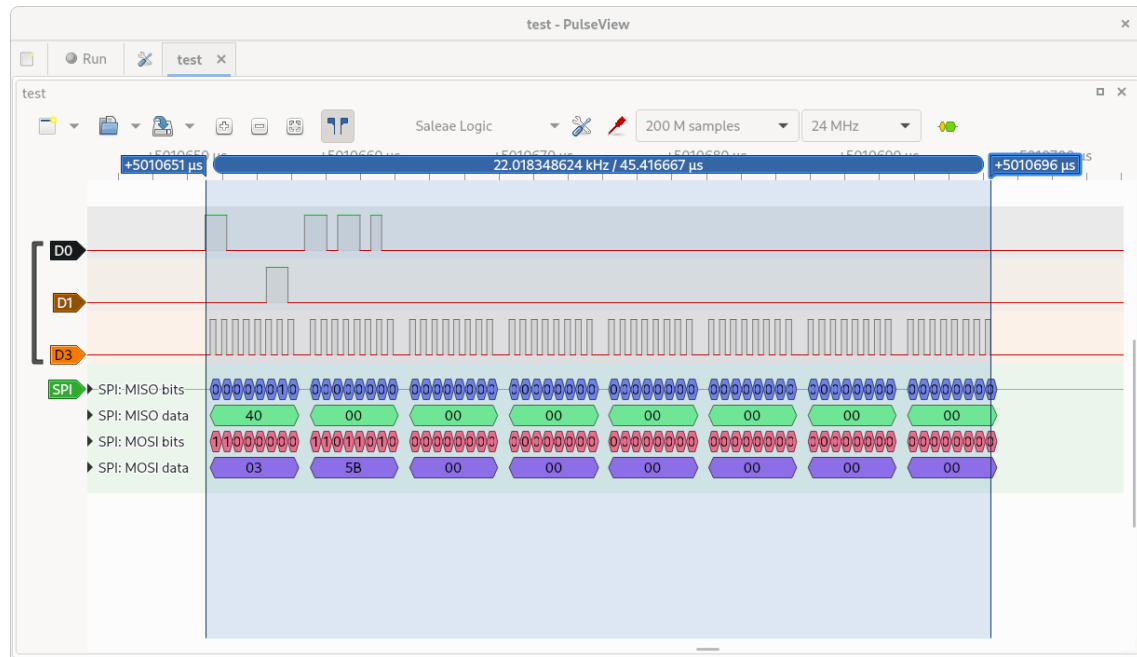


Figure 5.5: Logic analyzer recording of an SPI command

### 5.3.2 PWM validation

To validate the PWM module of the Verilog program, PWM signals with a set duty cycle of 50% and 25% are generated on the icoBoard by sending the duty cycle over SPI. Then, a logic analyzer is used to plot the PWM signal. These are respectively shown in Figure 5.7 and Figure 5.8.

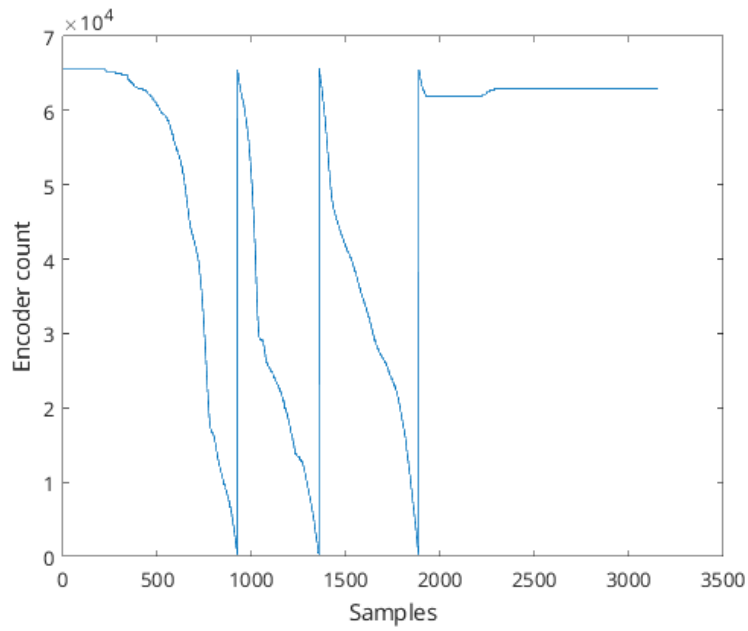
These figures show the expected result. The signal was sampled at 24 MHz and according to the logic analyzer it has a frequency of 762.970 Hz. In Verilog, PWM is implemented by setting a duty cycle value and taking a counter that increments every clock cycle. If the counter is lower than or equal to the duty cycle, the output pin is set to 1, and if the counter is higher than the duty cycle the output is set to 0. To validate whether the measured results are equivalent to what was written in code, the following equation can be used:

$$b = \log_2 \left( \frac{f_{CLK}}{f_{PWM}} \right) = \log_2 \left( \frac{100,000,000}{762.970} \right) \approx 17 \quad (5.1)$$

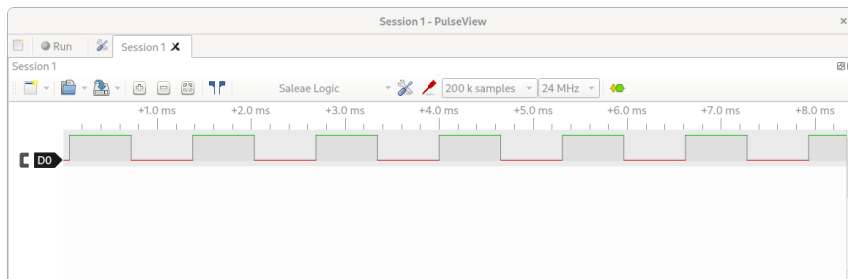
In this equation,  $f_{CLK}$  is equal to the clock speed of the FPGA, in this case 100 MHz, and  $f_{PWM}$  is equal to the frequency that was measured using the logic analyzer. The calculated value for  $b$ , the number of bits which are needed to represent the maximum value of the counter, is correct as the number of bits of the register used for the counter in Verilog is indeed equal to 17.

### 5.3.3 Complete system

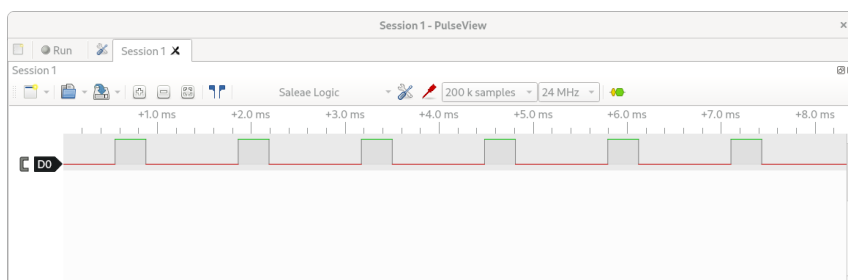
With the previously mentioned peripheral devices it should be possible to integrate this work in the complete robotic platform. A simple test could be to have the robot drive at a constant speed. Due to differences in motor specifications, for the robot to be able to drive in a straight line a simple controller needs to be implemented in the framework that is written in C. This can be done by taking the values from both encoders and calculating the difference between these, which can then be fed into a PID controller to set the PWM values such that this difference in encoder values is zero. Unfortunately, it is not possible to execute this integration successfully due to a lack of available time. It should also be noted that the values that are read from the encoder are not correct, which means that this also requires more investigation before a proper test can be done.



**Figure 5.6:** Encoder values for turning the shaft approximately a quarter of a rotation.



**Figure 5.7:** PWM signal with a duty cycle of 50%



**Figure 5.8:** PWM signal with a duty cycle of 25%

---

## 6 Conclusion

### 6.1 Conclusions

The goal of this thesis is to design an embedded software architecture for an education robot that is expandable in both software and hardware features, provides open access to the complete software stack, utilizes a Raspberry Pi 4 and an icoBoard and has real-time behaviour to work with time constraints.

With the architecture that is designed in Chapter 4 and the results that are presented in Chapter 5, an architecture is realized that includes a real-time operating system, a communication layer between the two different hardware platforms and support for different peripheral devices. The following can be said about the design objectives stated in Chapter 1:

- The architecture includes a Raspberry Pi 4 and an icoBoard: C code is written for the former and Verilog is written for the latter, and a communication protocol is implemented between these two devices.
- The complete software stack is accessible for the end user: all code is openly available and can be extended based on different sets of requirements.
- The architecture requires real-time behaviour: from the figures that are shown in Chapter 5, it can be concluded that real-time behaviour is indeed achieved.
- The architecture needs to be expandable for different requirements: the SPI protocol follows a modular approach where new sensors can easily be added.
- The device needs to be easily incorporated in the complete education robot: encoder and PWM modules are integrated to allow the hardware of the robot to be utilized.

Unfortunately, due to time constraints and difficulties getting the software to work, not everything that was originally set out was actually completed. Some recommendations are given in the next section.

### 6.2 Recommendations

In this section, several recommendations are discussed that can be used to improve the embedded software architecture as presented in this report. These have not been implemented in the current platform.

#### 6.2.1 Lower sample-to-action delay

A way to decrease the sample-to-action delay would be to set up a second process in Verilog that sets the newly sampled sensor data before the next SPI command arrives instead of during receipt of the next command. In Section 5.2 it is stated that currently the communication protocol that was designed in Section A.1 cannot be implemented directly. This is because the current setup in Verilog does not allow the write buffer to be filled asynchronously (meaning independently from when a new SPI command is handled; currently it can only be done after a new command has been received). Because of this, sampled sensor data is sitting in a buffer waiting until the Raspberry Pi 4 initiates the next command, adding 1ms of delay before the data is sent.

#### 6.2.2 More expandable Verilog architecture

To make the Verilog architecture more expandable, buffers can be tied to each other. SPI works by shifting the bits from a register onto a pin. To tie together these different buffers, the data from one buffer can be passed on to the next buffer, combining these separate buffers into one large buffer in the form of a shift register. A reasonable approach would then be to have one

buffer per peripheral device. Such an approach would make it easier to expand the Verilog code to support more different peripheral devices. Currently, the Verilog code that was written assumes a fixed amount of 4 peripheral devices: two PWM modules and two encoders. The SPI command, which contains the data and commands that are sent to these devices, needs to be as long as the complete sum of all this data.

### 6.2.3 Peripheral validation

To validate whether the correct peripheral devices are connected to the icoBoard, a function can be added where the sensor configuration is hashed, sent back on initial configuration and checked against the same hash on the Raspberry Pi 4. If the hashes are equal, the system can then begin execution. If there is a mismatch in the hash, the system can shut down or not proceed with execution. To improve this even further the system needs to know every single sensor that is connected and based on that decide what data has to go in what order in the SPI command. This requires a much more elaborate handshake in the initial phase of setting up the SPI connection.

When connecting the Raspberry Pi 4 and the icoBoard, the software on both sides assumes a certain amount of sensors and actuators to be present. In the current design this is validated by returning the amount of sensors from the icoBoard to the Raspberry Pi 4 during the initial connection, adding the ability to shut down the software on the Raspberry Pi 4 if the amount of sensors does not match. In practical situations however, not always the same peripheral devices are connected to a system. If, for example, one device has a PWM peripheral and another has a distance sensor, the sensor count is still the same while the overall system configuration is not.

### 6.2.4 Large amounts of peripherals

An extension can be added to the SPI protocol to determine what sensors need to be sampled at what time instance. In case a very large amount of peripherals is connected, it might be possible that the time duration per sample of the current SPI algorithm is too high relative to the period time of one cycle. In that case it might be possible to add a feature where not every sensor is polled at every sample time, but only when strictly required. The amount of peripherals in a single SPI command that fits in 1 ms can be calculated like this:

$$N_{\text{peripherals}} = \frac{T_{\text{period}} f_{\text{SPI}}}{8N_B} \quad (6.1)$$

With the values for  $N_B$  (the number of bytes per sensor),  $T_{\text{period}}$  (the period interval time) and  $f_{\text{SPI}}$  (the SPI frequency) that are currently being used, the filled in equation becomes the following:

$$N_{\text{peripherals}} = \frac{T_{\text{period}} f_{\text{SPI}}}{8N_B} = \frac{0.001 \cdot 32,000,000}{8 \cdot 2} = 1937.5 \quad (6.2)$$

That means that with the current setup this would never become a problem as 1937 is much higher than the amount of peripheral pins that are available on the icoBoard. However, with for example a more elaborate SPI protocol with more overhead to check available devices, more bytes per sensor for better accuracy, a lower SPI frequency for better reliability, a larger data overhead for error correction or other features that decrease the amount of peripherals that fit in a 1 ms signal, this could potentially become a problem. It should also be noted that if the total time of 1 ms is filled, no room for calculations is left on the Raspberry Pi 4, so in practice this value of 1937 peripherals can never be reached.

---

## A Software

In this appendix, an overview of the software that is written for this project is given. The source of these programs can be found on the RaM GitLab server (Vinkenvleugel, 2022). In Appendix B more information is given about how to set up this software and what packages are required to compile or synthesize the source code.

### A.1 Userspace

For this project, a userspace program is written in C that borrows from Hofstede (2022). This software constructs a real-time timer that has a period duration of 1 ms, and executed a function at the start of every period. This function sends and receive messages over SPI following the protocol from and it runs a control loop based on the values that have been received over SPI. A list of the files and a description of each file is listed below:

- `src/main.c`: the C file that contains the `main` function that is executed at the start of the program. This function starts a new thread with the scheduling parameters that are required for real-time execution. In this thread, the function `timer_thread` is executed.
- `src/timer.c`: this file contains the functions that deal with setting up and starting the timer:
  - `timer_thread`: function that is started as a separate thread that calls the `setup_timer` function to get a timer ID, then runs the timer using `timer_run` and then after executing calls `timer_delete` to delete the timer.
  - `timer_setup`: creates a new timer that fires on receipt of `SIGALRM` by calling `timer_create`.
  - `timer_run`: sets a new timer interval, sets up the controller using `control_setup` and runs `timer_settime` to start the timer. Then, a loop is started that waits until `SIGALRM` is received, then records the time (for checking how long a period lasts) and then runs `timer_handler`. After the loop has finished, the controller is exited by calling `control_exit`.
  - `timer_handler`: calls `control_run`.
- `src/control.c`: in this file, the setup and execution functions for the controller are located. No actual controller is provided at this point, but a framework for sending and reading SPI commands is, as well as a basic structure of the communication protocol that was described in this report. The following functions are provided:
  - `control_setup`: sets up the SPI connection by calling `spi_setup` and then sends the start command over SPI.
  - `control_run`: runs the controller, also implements the rest of the SPI protocol. Can also be adapted to run various tests.
  - `control_exit`: exits the controller by sending the exit command over SPI and disables SPI using `spi_exit`.
- `src/spi.c`: this file contains the helper functions for writing to and reading from the SPI character device. The included functions are:
  - `spi_setup`: sets up the connection with the SPI kernel driver using the correct settings by calling `ioctl`.
  - `spi_send`: takes an SPI command and sends it to the kernel driver.
  - `spi_recv`: takes a buffer, reads the SPI kernel driver and writes the SPI command to the buffer.

- `spi_exit`: closes the file descriptor of the SPI kernel driver.
- `src/utls.c`: this file contains several helper functions, such as the function that is used to write timing data to a CSV file and the function that is used to print the contents of an SPI command buffer to `stdout`. These functions are mostly used for debugging and testing purposes.

Though some settings for the SPI driver have been taken from Hofstede (2022), the rest of this program is built from the ground up to properly integrate the new SPI protocol from Section A.1. The choice to write this program in C instead of the previously used C++ comes from a personal preference of the author regarding object-oriented programming, but yields no other benefits. In Figure A.1 a flow diagram of the userspace software is shown, based on the functions that are included in the source files that are described above.

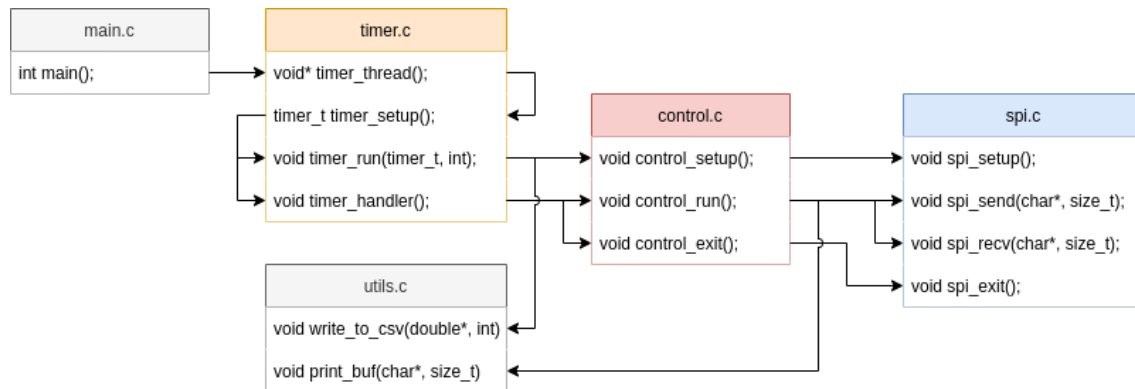


Figure A.1: Flow diagram of the userspace software

## A.2 Verilog

On the `icoBoard`, Verilog has been written for the Lattice `iCE40-HX8K` FPGA. This code is also adapted from Hofstede (2022). A list of the files and a description of each file is listed below:

- `src/top.v`: this file contains the module `top` that is the "main" module of the Verilog code. This connects to all the ports that are specified in `io.pcf`, instantiates the SPI module and connects to the encoder and PWM modules.
- `src/spi.v`: this file contains the module that acts as an SPI peripheral device. All data is read from the pins by this module and written to a buffer that can be accessed from `top`.
- `src/pwm.v`: in this file, the PWM module is defined. It accepts the duty cycle and then writes to the pins that are specified for PWM.
- `src/enc.v`: this file contains the module that is responsible for reading the encoder inputs.

In case of the SPI module, the previously existing code is simplified for clarity, removing features that are unused in the new SPI implementation: for example different SPI states are not used in the new implementation. In `top`, the code has been adapted to handle the use of a long list of peripherals instead of having a command per port. The PWM and encoder modules have largely been taken as-is.



## B Toolchain

In this appendix, the toolchain for setting up the embedded software architecture is discussed. Currently the code can be found on the RaM GitLab server (Vinkenvleugel, 2022). In this section, it is assumed that an icoBoard is connected as a "HAT" on top of the Raspberry Pi 4.

### B.1 icoBoard

For the icoBoard, as described in Section 2.1, the toolchain consists out of three separate parts:

- **Yosys:** an open synthesis suite for Verilog, used to convert the initial Verilog code to an intermediate representation in JSON (YosysHQ, 2022c).
- **nextpnr:** a place and route tool that accepts the JSON file that was generated by Yosys and converts it to a hardware-specific layout in an ASC file (YosysHQ, 2022a).
- **Icestorm:** a package that contains hardware support for the particular Lattice iCE40-HX8K FPGA that is used in the IcoBoard. A program called Icepack is used to convert the ASC file to a binary file that can be flashed on the FPGA chip (YosysHQ, 2022b).
- **Icotools:** tools to flash the binary file on the FPGA, either to flash or to run it as a bitstream (Wolf, 2019).

Additionally, other tools such as Icarus Verilog (Williams, 2022) can be used to simulate Verilog code on a computer. On GitLab, a Makefile is provided that can automatically build and remotely flash the Verilog code on the icoBoard. This build file expects a file called `top.v` that contains a module called `top`.

### B.2 Raspberry Pi 4

For the Raspberry Pi 4, the toolchain is slightly more extensive. First of all, the Linux kernel needs to be built according to the guide in Meijer (2021a). In this project the specific kernel version that is required by the I-pipe patch (Tam, 2019) is taken, the patches are applied and the kernel is then built. After this the kernel can be installed on the Raspberry Pi 4, which should run a 32-bit version of Raspberry Pi OS (Legacy) (Raspberry Pi, 2022b) as its operating system. Then, the driver that is provided (Hofstede, 2022) needs to be inserted in the Linux kernel. For convenience, the repository for this project contains this driver in binary form, but in case of changes in the kernel version this driver needs to be updated as well. After these steps have been executed the userspace Xenomai tools for 32-bit Arm can be downloaded from Xenomai Authors (2022) and installed as usual.

If all of the above steps are completed the resulting operating system can be tested by running `/usr/xenomai/bin/latency` to check whether the latency figures are approximately equal to those found in Figure 5.1.

To then install and test the userspace program, `/usr/xenomai/bin/xeno-config` is required to get the correct compilation flags for the POSIX skin that is used. For this, again a Makefile is provided that can be used to compile the program. If the icoBoard is properly connected and flashed according to the previous section, the program can be executed using `make run`.

## Bibliography

- Avago Technologies (2014), HEDM-55xx/560x HEDS-55xx/56xx.  
<https://docs.broadcom.com/doc/AV02-1046EN>
- Binnenmars, R. (2022), Hardware design of a mobile education robot, that uses Raspberry Pi 4 and FPGA-board for real-time control.
- Hofstede, A. (2022), Drivers and Verilog to control motors using Pi 4 Xenomai.  
<https://git.ram.eemcs.utwente.nl/hofstede/pi4-icoboard>
- icoBoard (2022), icoBoard is a FPGA based IO board for Raspberry Pi.  
<http://icoboard.org>
- McKenney, P. (2005), A realtime preemption overview.  
<https://lwn.net/Articles/146861/>
- Meijer, B. (2021a), Creating a pipeline for integrating 20-sim models into real-time and ROS2 with the help of Xenomai.  
<https://git.ram.eemcs.utwente.nl/meijera/xenomai-ros2-20sim-pipeline>
- Meijer, B. (2021b), Real-time robot software framework on Raspberry Pi using Xenomai and ROS2.
- OSHA (2022), New SPI Terminology.  
<https://www.osha.org/2022/01/10/new-spi-terminology/>
- Raspberry Pi (2022a), Buy a Raspberry Pi 4 Model B.  
<https://www.raspberrypi.com/products/raspberry-pi-4-model-b/>
- Raspberry Pi (2022b), Operating system images.  
<https://www.raspberrypi.com/software/operating-systems/>
- Schurando, N. (2016), Real-Time SPI driver for the Broadcom BCM2835 (BCM2708) and BCM2836 (BCM2709) SoCs using the RTDM API.  
<https://github.com/nschurando/spi-bcm283x-rtm>
- Tam, H. (2019), Xenomai for Raspberry Pi 4.  
<http://www.simplerobot.net/2019/12/xenomai-3-for-raspberry-pi-4.html>
- Vinkenvleugel, J. (2022), Robotics Education Lab.  
<https://git.ram.eemcs.utwente.nl/vinkenvleugeljt/rel>
- Wikipedia Authors (2022), Verilog.  
<https://en.m.wikipedia.org/wiki/Verilog>
- Williams, S. (2022), Icarus Verilog.  
<http://iverilog.icarus.com/>
- Wolf, C. (2019), Icotools.  
<https://github.com/cliffordwolf/icotools>
- Xenomai Authors (2022), Xenomai.  
<https://xenomai.org>
- YosysHQ (2022a), nextpnr – a portable FPGA place and route tool.  
<https://github.com/YosysHQ/nextpnr>
- YosysHQ (2022b), Project IceStorm.  
<http://bygone.clairixen.net/icestorm>
- YosysHQ (2022c), Yosys Open SYnthesis Suite.  
<https://yosyshq.net>