Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institute for
Dynamic Systems and Control
IDSC
Institut für Dynamische Systeme
und Regelungstechnik

Florian Wenk

Sigi2.0

Documentation

Technical Report

Institute for Dynamic Systems and Control
Swiss Federal Institute of Technology (ETH) Zurich

June 28, 2021

Abstract

The Sigi2.0 is a inverted pendulum on wheels developed for teaching purposes at the Institute for Dynamic Systems and Control (IDSC) at ETH Zurich. It aims to provide a possibility for the students of control to gain some hands-on experience and a playful insight into control systems. For this purpose a inverted pendulum on wheels was chosen because it is a well-known system for which many different control algorithms can be applied, making it possible to use this as an example for a broad range of lectures.
This report is the documentation to this setup, providing an brief overview of the hardware, the provided interaction software and on how to set the Sigi2.0 up.

Keywords: Sigi2.0, Inverted Pendulum, Inverted Pendulum on Wheels, Documentation.

Contents

Nomenclature

Acronyms and Abbreviations

ADC	Analog Digital Converter
I^2C	Equivalent to TWI, Interface using two wires to communicate with hardware
ISR	Interrupt Service Routine, Piece of code executing when a microprocessor receives an interrupt.
PWM	Pulse width modulation

Non-SI units

$^\circ$	Degree	$360\,^\circ = 2\pi\,\mathrm{rad}$
dps	Degrees per second	$1\,\mathrm{dps} = \frac{\pi}{180}\,{}^{\mathrm{rad}}/_{\mathrm{s}}$
gauss	Gauss; Magnetic induction / Magnetic flux density	$1\,\mathrm{gauss} = 10^{-4}\,\mathrm{T}$
g	Earths gravitational acceleration	$1\,\mathrm{g} \approx 9.81\,{}^{\mathrm{m}}/_{\mathrm{s}^2}$
h	Number in hexadecimal format	
LSB	Least significant bit, smallest step of an integer number	$1\,\mathrm{LSB} = 1$

Chapter 1

Introduction

The inverted pendulum on wheels Sigi2.0 was developed at ETH Zurich to present control systems to the students. This is because the inverted pendulum is a well known system with various aspect making it possible to compare different control strategies. The Sigi2.0 bases on the `Balboa32U4` platform by Pololu, to which a Raspberry Pi is attached for running the control algorithm. To this setup additional hardware can be added to examine also different aspects of the pendulum. E.g. in [1] additional encoders on the wheels were added to an instance to gain a deeper insight on the backlash between the motors and wheels. The Sigi2.0 is depicted in figure 1.1.

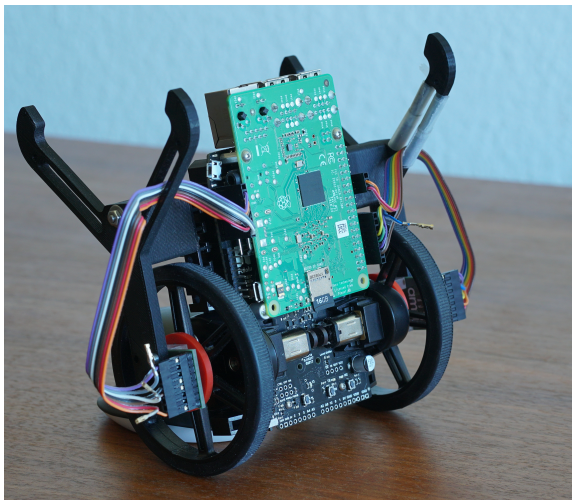

Figure 1.1: The Sigi2.0 with additional encoders attached to measure the rotation of the wheels

This setup has several advantages and disadvantages. The advantages include that all parts are commercially available, making it possible to obtain the parts at relatively low costs, and that the control algorithms can be programmed in Mathworks Simulink, providing a graphical way for programming and an easy possibility to see the control flow. This makes it easy to for the students to use the program without needing a knowledge of "classical" programming. The disadvantages are among others that due to the usage of the Raspberry Pi and Simulink no real-time guarantees can be made and that the system is slower than if the control algorithm would be implemented directly on the microprocessor on the Balboa32U4. However, for this purpose the advantages are highly desirable, while these disadvantages are not very important.

In chapter 2, a short overview of the hardware on the Sigi2.0 is given. Then, chapter 3 describes the provided software to interact from Simulink with the hardware. There, the part written in Simulink as well as the part on the microprocessor are explained and it is detailed how to use the software. Finally, chapter 5, summarizes how to install the needed software on the Sigi2.0.

Chapter 2

Hardware

The Sigi2.0 is an inverted pendulum on wheel based on the Balboa32U4 balancing robot kit by Pololu. The kit provides all the hardware needed to balance the pendulum. However, to be able to improve the controller and easily interact with Simulink with the robot a Raspberry Pi Model B+ is attached. This further adds WIFI capabilities, such that the full access to the device can be done wireless. Additionally, wheel encoders were added to the outside of the wheel which allow further investigation of the backlash occurring due to the gearing. In the following sections the individual hardware components will be further discussed.

2.1 Pololu Balboa 32U4

The Pololu Balboa32U4 consists mainly of an control board, a connected battery case and optionally some bumpers which protect the hardware when falling. The control board includes an Micro-USB B port which can be used to connect to a computer. Through this connection it is possible to communicate with the boards microcontroller as well as provide power to the board. For the programming of the board one can connect a computer to the USB port and then use the Arduino IDE, as the Arduino-compatible A-Star 32U4 USB bootloader is loaded on the microcontroller. Alternatively an 6-Pin ISP header is provided by the board which allows to program the board with an external programmer. This is needed to flash the bootloader if it isn't written correctly by the manufacturer. For programming the Arduino library Balboa32U4[1] is provided, which configures the hardware as needed.
The following hardware components are included on the board:

- 1 Atmel Atmega32U4 microcontroller
- 2 Texas Instruments DRV8838 motor driver
- 2 Quadrature encoder
- 1 STMicroelectronics LSM6DS33 inertial module
- 1 STMicroelectronics LIS3MDL magnetometer
- 5 Buttons
- 5 LEDs
- 1 Buzzer
- 1 ID EEPROM

[1] Available at https://github.com/pololu/balboa-32u4-arduino-library

Furthermore 2 Pololu Micro Metal Gearmotors are attached to the motor drivers as well as 2 wheels which are driven by the motors trough some gearing. It is also possible to add custom hardware, provided by Pololu or any other manufacturer, to the board using the provided pinouts. In the following sections the single components (present on the Sigi2.0 configuration of the Balboa32U4) and the power management will be explained.

2.1.1 Atmega 32U4

The Atmega32U4 by Atmel is an USB enabled 8-Bit AVR microcontroller, which is clocked at 16MHz and driven by a supply voltage 5V. This chip contains 32KBytes of Flash memory used for the program, which can be flashed by USB. It is fully integrated onto the board and its pins are connected to the hardware which are controlled by it. As the main computing power of the board it is theoretically able to balance the balboa on its own. The I^2C (or TWI) interface is the main communication protocol used for data acquisition is also provided by the chip. It's logic voltage is 5V (and therefore also the boards), which is important to consider if additional devices are to be connected. Detailed information is available in the datasheet[2]

2.1.2 Motors and Motor Drivers

Motors Two Pololu Micro Metal Gearmotors are attached to the DRV8838 motor drivers by Texas Instruments. The motors are available with 12 different gear ratios ranging from 5:1 to 1000:1 as well as in 5 motor variations:

Motor Option	Voltage [V]	Power	Brushes
LP 6V	6	Low-power	Metal brushes
MP 6V	6	Medium-power	Metal brushes
HP 6V	6	High-power	Metal brushes
HPCB 6V	6	High-power	Carbon brushes
HPCB 12V	12	High-power	Carbon brushes

All motors are available with an extended rear motor shaft, which is needed for the encoders (see section 2.1.3). However, the 12 V motors might not be suitable for the Balboa, as the motor drivers are only recommended to be used up to 11 V motor power-supply voltage (although 12 V is the absolute maximum rating) and 6 AA NiMH batteries only provide a nominal voltage of 7.2 V. The performance data of the motors including charts with the motor characteristics can be found in the datasheet[3]

Motor Drivers Each of the installed Texas Instruments DRV8838 motor drivers are usually controlled using 3 input pins: nSLEEP (Sleep mode input), PH (Phase) and EN (Enable). The outputs (OUT1 and OUT2) are to be connected to the motor. Then, the control logic is as presented in the following table:

nSLEEP	PH	EN	OUT1	OUT2	Motor Function
0	X	X	Z	Z	Coast
1	X	0	L	L	Brake
1	1	1	L	H	Reverse
1	0	1	H	L	Forward

Here, X stands for either 0 or 1, Z for high impedance, H for high output and L for low output. Both outputs being low in brake mode means that the motor is short circuit and the full energy is dissipated over the motor resistance and therefore the motor can get quite hot.
On the board, the phase input of the left motor is connected to the digital pin 16 (PB2) of the Atmega32U4 and the enable input to the digital pin 10 (PB6). For the right motor, phase is

[2] Available at http://ww1.microchip.com/downloads/en/devicedoc/atmel-7766-8-bit-avr-atmega16u4-32u4_datasheet.pdf

[3] Available at https://www.pololu.com/file/0J1487/pololu-micro-metal-gearmotors-rev-4-2.pdf

connected to digital pin 15 (PB1) and enable to digital pin 9 (PB5). The sleep mode input pins are not connected to the microcontroller and pulled up to high. Thus the motor cannot go to coast, and the motor speed is set by adding a Pulse width modulated (PWM) signal to enable (which switches the motor between brake and drive mode), and the direction can be controlled by setting the phase input (0 results in forward, 1 in reverse rotation).

Caution: As the nSLEEP pin of the drivers is always pulled up, the motor is, when the Balboa32U4 is powered on, automatically in brake mode and any external turning will heat the motors up! Also due to the setting of the speed with braking, the normal usage of the motors heats them up and when used for a while, they will get hot and one has to make a pause and wait for them to cool down, as the glue in the coil of the motors can melt and if it is melted and then cools down it can glue the rotor which then gets blocked and the motors damaged!

For further information such as protection circuits it is referred to the datasheet[4].

2.1.3 Quadrature Encoders

Both motors are equipped with a quadrature encoder system. Each of these encoders consist of two hall effect sensors, which are directly mounted on the board. These hall sensors then capture the change in the magnetic field caused by the rotation of the magnetic disc attached to the extended shaft of the motor. The two sensors are placed each on one side of the magnetic disc and therefore register the change of the magnetic field shifted. Then, by considering which sensor first recorded the change, one can get the direction of rotation. As there can be several thousand encoder ticks per second on the balboa, the encoders have to be handled by interrupts. The two sensors each yield one channel of the encoder channels (A and B). To reduce the number of required interrupt pins on the Atmega32U4 the two signals get XORed together and this signal (A XOR B) is then used for the interrupt. The channel B is also input to the Atmega32U4, with which then the two original signals can be reconstructed, as (A XOR B) XOR B = A. Both encoders are connected such that channel B leads channel A when the motor is rotating in forward direction (B rises before A rises and B falls ahead of A). Such a encoder signal for the motor turning forward is depicted in figure 2.1. On the Atmega32U4 the digital pin 8 (PB4) has the XORed signal of the left encoder,

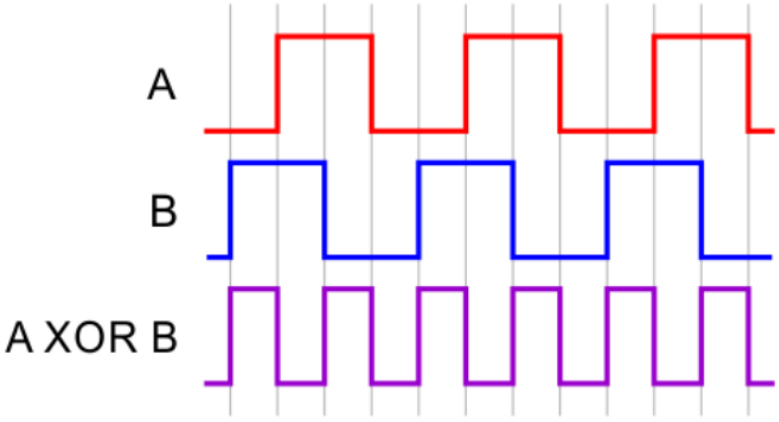

Figure 2.1: Example of the encoder signal and the XORed of it for the motor turning forward. The graphic is taken from Pololu's "Pololu Balboa 32U4 Balancing Robot User's Guide".

and PE2 has the corresponding channel B. For the right encoder the XORed signal is on digital

[4]Available at https://www.ti.com/lit/ds/symlink/drv8838.pdf?ts=1588175657093

pin 7 (PE6) and channel B on digital pin 23 (PF0).
The encoder register 12 counts per revolution of the magnetic disc, thus one tick corresponds to an angle of $30\,^\circ \equiv 0.5236\,\text{rad}$. This is quite a large angle, but to get the wheel angle the gearing still has to be considered which usually results in a higher resolution for the wheel angle.

2.1.4 Wheels and Gearing

Wheels The wheels used on the Sigi2.0 are Pololu Wheels $80 \times 10\,\text{mm}$ which have a diameter of 80 mm and a thickness of 10 mm. They consist of an ABS (Polylac) PA-747[5] hub and a silicone rubber tire. A D-shaped hole with an diameter of 3 mm, which matches the shaft of the Pololu Micro Metal Gearmotor and the shaft of the additional gearing coming with the Balboa32U4. The mass evaluates to approximately 19.84 g and the moment of inertia along its axis is approx. $22.49 \cdot 10^{-6}\,\text{kg}\,\text{m}^2$, which is calculated from the model provided as step-files by Pololu[6] using an uniform density of $1030\,\text{kg}/\text{m}^3$ from the ABS specifications, as there is no exact value for the density of the silicone rubber found.

Gearing There are two different parts of the gearing on the Sigi2.0. On one hand, there is the metal gearing already mounted on the motors and on the other hand the external plastic gearbox which is chosen at the time of assembly of the Balboa32U4. The Pololu Micro Metal Gearmotors are available with 12 different gear ratios ranging from $5:1$ up to $1000:1$. However, there are 3 versions using the HPCB 6V motor especially recommended by Pololu:

Nominal Ratio	Exact Ratio
$30:1$	$86955/2912$
$50:1$	$3344/65$
$75:1$	$38437/507$

The external gearbox ratio can be chosen from the 5 possibilities in the following table by using different gears in the assembly:

Gear Ratio
$49:17$
$47:19$
$45:21$
$43:23$
$41:25$

Using these two gearboxes the overall gear ratio (which results from a multiplication of the two independent ones) can be adjusted to 15 different values from $45.9:1$ up to $218.5:1$. However it should be noted, that higher ratios introduce more backlash which makes the Sigi2.0 harder to stabilize.

2.1.5 Inertial Module

The STMicroelectornics LSM6DS33 inertial module consists of a 3-axis accelerometer and a 3-axis gyroscope. It is connected to the Atmega32U4's and Raspberry Pi's I^2C (or TWI) ports through which it is controlled and read out. The axes of the sensor are oriented as depicted in figure 2.2.

Here only some key values will be shown, for further information e.g. the temperature dependency of the values and the control registers that are to be set for configuration it is referred to the datasheet[7].

[5] Specifications available at http://www.chimeicorp.com/upload/att/2013-05/20130516135948246872l.pdf
[6] Available at https://www.pololu.com/file/0J1294/pololu-wheel-80%C3%9710mm.zip
[7] Available at https://www.st.com/resource/en/datasheet/lsm6ds33.pdf

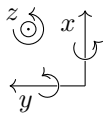

Figure 2.2: Orientation of the axes of the sensors on the Balboa32U4. The z-axis points out of the control board towards the mounted Raspberry Pi and the x-axis away from the motors.

Accelerometer The linear accelerometer is constructed to measure the acceleration in multiples of earths gravitational acceleration $1\,\mathrm{g} = 9.81\,\mathrm{m/s^2}$ and can measure DC accelerations. It outputs three 16-bit integers in two's complement, each corresponding to the acceleration along one axis. Multiplying this value with the sensitivity then results in the measurement in physical units. The unit of the sensitivity is thus denoted as $\mathrm{g/LSB}$ where LSB stands for least significant bit and corresponds to a value of 1 in the `int16` for the axis value. The sensitivity is coupled to the full-scale (maximal measurable values) and can be set to the following:

Full-Scale [g]	Sensitivity [mg/LSB]
±2	0.061
±4	0.122
±8	0.244
±16	0.488

These values are the typical values at a temperature $T = 25\,°\mathrm{C}$ and input voltage $Vdd = 1.8\,\mathrm{V}$. However these values might be inaccurate, as the sensor is connected to 3.3 V. Additionally several filters are already implemented in the accelerometer chain. These filter's bandwidths and types can be configured using the registers. By default there is a anti-aliasing filter with bandwidth of 400 Hz and an digital low-pass filter where the bandwidth is dependent on the selected output data rate (ODR). The following table shows the possible ODRs:

Accelerometer ODR [Hz]
13
26
52
104
208
416
833
1666
3332
6664

Gyroscope The gyroscope measures the angular velocity in degrees per second $180\,\mathrm{dps} = \pi\,\mathrm{rad/s}$. As the accelerometer it also outputs its data in three 16-bit integers in two's complement which can be calculated to the physical value using the sensitivity, which is dependent on the selected full-scale. The typical values for sensitivity and full-scale at $T = 25\,°\mathrm{C}$ and $Vdd = 1.8\,\mathrm{V}$ are:

Full-Scale [dps]	Sensitivity [mdps/LSB]
±125	4.375
±245	8.75
±500	17.5
±1000	35
±2000	70

Similar to the accelerometer, some filters are already available on the chip. By default an analog anti-aliasing filter and a digital low-pass filter, with an output data rate dependent bandwidth. Additionally an digital high-pass filter can be configured. The available ODRs are:

Gyroscope ODR [Hz]
13
26
52
104
208
416
833
1666

2.1.6 Magnetometer

An STMicroelectronics `LIS3MDL` 3-axis magnetometer is also available on the control board. As the inertial module it is connected to the `Atmega32U4`'s and Raspberry Pi's I^2C port for communication. The axes point in the same directions as those on the inertial module, which are shown in figure 2.2. The magnetometer measures the magnetic flux density in $1\,\text{gauss} = 10^{-4}\,\text{T} = 10^{-4}\,{}^{\text{kg}}/_{\text{A}\,\text{s}^2}$. The 16-bit data output is given in two's complement and can be read out as a `int16` number which then can be multiplied by the sensitivity to obtain the value in the physical unit. The sensitivity depends, similar to the inertial module, on the selected full-scale and from the following typical values (at $T = 25\,°\text{C}, Vdd = 2.5\,\text{V}$) can be selected:

Full-Scale		Sensitivity	
[gauss]	[T]	$[{}^{\text{LSB}}/_{\text{gauss}}]$	$[{}^{\text{LSB}}/_{\text{T}}]$
± 4	$\pm 4 \cdot 10^{-4}$	6842	$6842 \cdot 10^4$
± 8	$\pm 8 \cdot 10^{-4}$	3421	$3421 \cdot 10^4$
± 12	$\pm 12 \cdot 10^{-4}$	2281	$2281 \cdot 10^4$
± 16	$\pm 16 \cdot 10^{-4}$	1711	$1711 \cdot 10^4$

The magnetometer does not include any filtering of the signals and the following ODR's are available:

Magnetometer ODR [Hz]
0.625
1.25
2.5
5
10
20
40
80
155
300
560
1000

However, the ODR's cannot be freely selected as some of them are only available in certain operating modes. For further information and the configuration it is referred to the datasheet[8].
It is important to mention that the magnetometer on the `Balboa32U4` is not very accurate, as there are many other sources of magnetic fields, such as currents and hard iron distortion. However, a post in the Pololu Forum[9] shows how to correct this.

2.1.7 Buttons

Five Pushbuttons are found on the `Balboa32U4` control board:

[8] Available at `https://www.st.com/resource/en/datasheet/lis3mdl.pdf`
[9] `https://forum.pololu.com/t/correcting-the-balboa-magnetometer/14315`

- 1 Power button
- 1 Reset button
- 3 User pushbuttons (A, B and C)

The power button is located in the bottom left corner right above the power switch and is used for turning the board on/off. The reset button in the top right corner resets the `Atmega32U4` and is also used to start the bootloader for programming. The three user pushbuttons are labeled `A`, `B` and `C` respectively. They are connected to the `Atmega32U4`'s pins 14 (`PB3`) for button `A`, 30 (`PD5`) for button `B` and 17 (`PB0`) for button `C`. If configuring the buttons one has to be cautious, as the pins have multiple uses for e.g. the LED's, the LCD or some lines for the SPI interface. Although the lines have multiple uses, the hardware is designed such that the buttons can be pressed at any time without disrupting any other communications nor damaging the control board. The functions in the `Balboa32U4` arduino library take care of configuring these pins and can be of help if one wants to reconfigure them.

2.1.8 LEDs

The `Balboa32U4` features five LEDs. Two of these are power indicators:

- A blue power LED, indicating when the `Balboa32U4` is powered by the batteries.
- A green power LED, indicating the presence of the USB bus voltage (VBUS).

These two power indicator LEDs are located in the top left corner of the control board.
The remaining three are user-controllable and located near the bottom of the board:

- A yellow LED, connected to the digital pin 13 (`PC7`). Driving the pin high, the LED is turned on. Additionally, the bootloader fades the LED on and off, while waiting to be flashed.
- A green LED, connected to pin 30 (`PD5`). Driving the pin low enables the LED. This LED is also flashing when the `Balboa32U4` transmits data via USB.
- A red LED, connected to pin 17 (`PB0`). Driving the pin low enables the LED. This LED is also indicating, when data is received via USB.

It is important to note that:

1. The user LED control lines are also data lines of the LCD (if connected). Therefore the LEDs will flicker when the LCD is updated.
2. The green and red user LEDs share the IO lines with the pushbuttons (see section 2.1.7).

2.1.9 Buzzer

The Buzzer, that has to be soldered manually onto the control board, is used to generate sounds and music. It is connected to digital pin 6 (which is also `OC4D`, a hardware PWM output from the `Atmega32U4`'s 10-bit timer). If the pin is driven high and low at a specific frequency, the buzzer generates a sound at that frequency.

2.1.10 Power

The `Balboa32U4` has generally two different power supplies from which one is selected:

- The USB Bus, and
- the battery.

The voltage from the USB is denoted as `VBUS` and the battery voltage as `VBAT`. The battery voltage undergoes several steps before the decision which supply to use:

I It passes the Power switch and a reverse voltage protection. This leads to the voltage `VSW`.

II Then, a 5 V power regulator is reducing the voltage to the needed 5 V, which is called `VREG`.

Afterwards, the logic power selection selects the power supply as follows:
If the regulated battery voltage `VREG` is bigger than 5 V, then this one is used. Otherwise, the higher voltage of `VREG` and `VBUS` is selected. The result of this power selection results in the final used voltage `5V`, which is then again regulated down to 3.3 V denoted as `3V3`. This power conversion and selection circuit is depicted in figure 2.3.

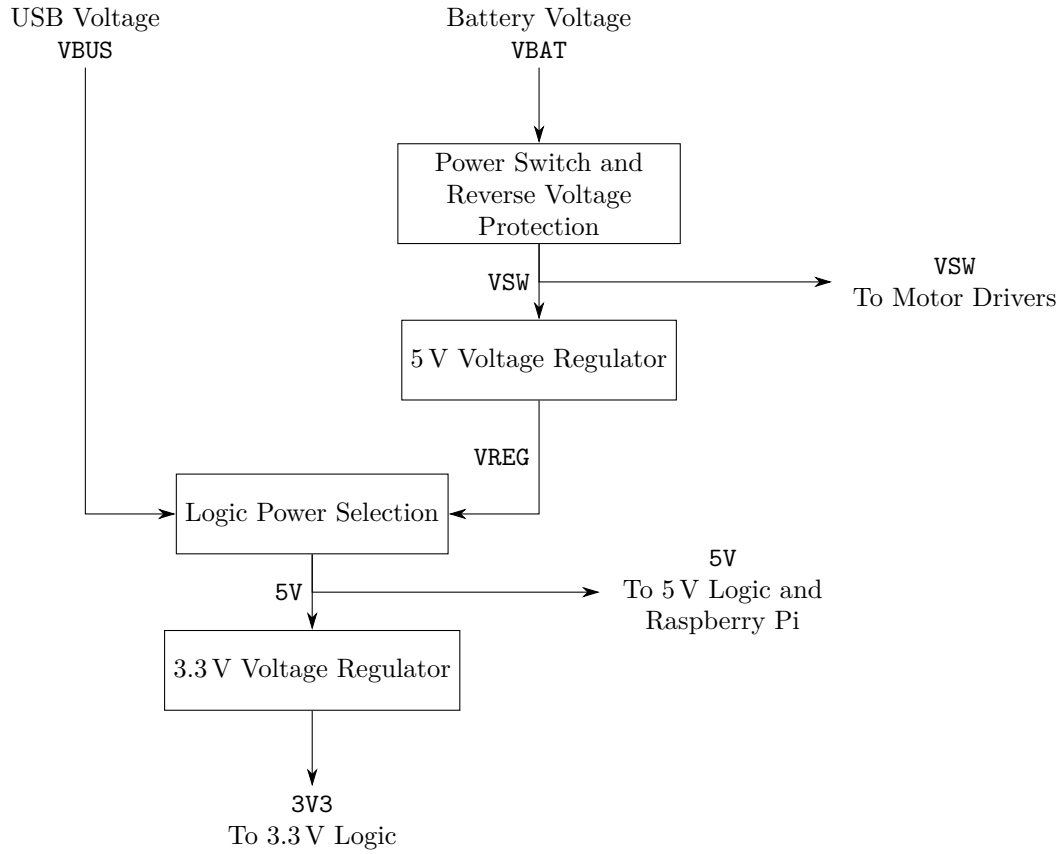

Figure 2.3: Power conversion and selection procedure

As indicated in figure 2.3, the `5V` voltage powers all the 5 V logic on the control board as well as the Raspberry Pi, while the `3V3` voltage powers all the 3.3 V logic on the control board.
The motor drivers and through them the motors themselves, are directly powered by `VSW`, resulting in the motors not being powered by the USB power and therefore, only working with the battery.
The description above is only a short summary on how the power is supplied to the hardware. More detailed information is available in the `Balboa32U4` documentation and on the schematics.

2.1.11 ID EEPROM

The board further includes a 32-kilobit (4069-byte) EEPROM directly attached to the Raspberry Pi's `ID_SD` and `ID_SC` pins. It thus is not connected to the `Atmega32U4` and can only be accessed by the Raspberry Pi (if one is connected). The EEPROM can be programmed according to the Raspberry Pi HAT specification [10].

[10] See `https://github.com/raspberrypi/hats`.

The EEPROM is not write-protected by default, however bridging the jumper labeled WP next to the EEPROM, write protection can be enabled.

2.2 Raspberry Pi

The Raspberry Pi 3 Model B+ is directly attached to the control board which features a mounting location for a connector, directly matching the Raspberry Pi's GPIO layout. This connects the I^2C lanes of the Raspberry Pi IO^{2}C Bus 1 with the corresponding ports of the Atmega32U4, the LSM6DS33 inertial module and the LIS3MDL magnetometer. As the Atmega32U4 works with a 5 V logic level, while the logic level of the other three components is 3.3 V, a bidirectional level-shifting circuit is employed to enable the communication.
In this way, the Raspberry Pi is used to expand the processing power while all communication works over I^2C.
As the Raspberry Pi is a fully equipped single-board computer, it is not only used for higher processing power, but also to be able to program the Sigi2.0 using Mathworks Simulink and for wireless communication.

2.3 AMS AS5048B Wheel Encoders

To measure the rotation angle not only of the motor but also of the wheels, two AMS AS5048B wheel encoders can be added on the wheel side facing away from the Sigi2.0's body. This can be used for e.g. examining the backlash.
The AMS AS5048B is a magnetic rotary encoder with an input voltage of either 3.3 V or 5 V and communicating over I^2C. It outputs the absolute rotation of the magnet in degrees of which the zero position can be programmed over I^2C. However, this programming can only be done once. It has a full-scale of 360 ° and a resolution of 14-bit, resulting in a sensitivity of 0.0219 °/LSB.
The 7-bit I^2C slave address is variable and set mostly over I^2C itself, while the two least significant bits depend on the hardware input pins A2 and A1.
Alternatively the measured angle can also be read out using the PWM signal at the sensors PWM output pin.
Further information is provided in the datasheet[11].

[11] Available at https://ams.com/documents/20143/36005/AS5048_DS000298_4-00.pdf/910aef1f-6cd3-cbda-9d09-41f152104832

Chapter 3

Interaction Software

The interaction software is used for communication with the different sensors and actuators. The hardware on the Pololu `Balboa32U4`, except the inertial module and the magnetometer, are connected to the Atmel `Atmega32U4` microprocessor on the board. The microprocessor itself as well as the inertial module, the magnetometer and the wheel encoders are connected to the Raspberry Pi over I^2C. As the control algorithm is running on the Raspberry Pi, interaction software to get the sensor data and to transmit signals to the actuators, is available. This software is mainly running on the Raspberry Pi itself, where it has been implemented in Simulink, but there is also a part on the `Atmega32U4` for the hardware not directly connected to the Raspberry Pi.
Most of the software running on the `Atmega32U4` is provided by Pololu in their *balboa-32u4-arduino-library*[1] and the configuration as I^2C slave is done by the *pololu-rpi-slave-arduino-library*[2]. This code can be found in appendix A.1.
Also, the configuration of the Raspberry Pi's I^2C module is all done by Simulink, which provides blocks for reading and writing over I^2C. For the description of the I^2C interface, see section 4.2.
In the following, the reading and writing of the sensors and actuators on the `Balboa32U4` and the Raspberry Pi is described. However, all the configuration of the I^2C interface is left out, as it is all done by the *pololu-rpi-slave-arduino-library* or Simulink respectively. It is only mentioned that, according to the Mathworks Support, the I^2C bus in Simulink is clocked at 400 kHz by default and this cannot be changed (Although according to the configuration file in the default Mathworks image for the Raspberry Pi, the I^2C is configured to clock at 100 kHz). Also, only the interaction software to interact with the hardware needed to stabilize the Sigi2.0 is described, thus the buzzer and magnetometer are left out and for the interaction with them it is referred to the respective documentation of the magnetometer and the `Balboa32U4`. Finally, the last section of this chapter provides an overview of the values that can be sent to the microprocessor, if the provided file is running.

3.1 Set Motor Speed

The motor speeds are sent from Simulink to the `Atmega32U4`, where the signal is transformed in Simulink to the right format for the program on the `Atmega32U4`, which then processes it and sends the corresponding PWM signal to the motor drivers which then put the right voltage onto the motors. Here, the description will be split into the part done by Simulink and the part done on the `Atmega32U4`. The part of how the signal is sent from the motor drives to the motors is already described in section 2.1.2.

[1] Available at `https://github.com/pololu/balboa-32u4-arduino-library`
[2] Available at `https://github.com/pololu/pololu-rpi-slave-arduino-library`

3.1.1 Simulink

The Simulink block `setMotors` is used to set the speed and direction of the two motors on the Sigi2.0. A schematic representation of it is depicted in figure 3.1. As seen in the scheme, this block

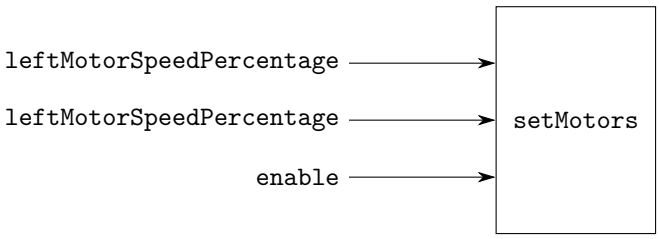

Figure 3.1: Inputs and outputs of the `setMotors` Simulink block.

takes three inputs:

1. `leftMotorSpeedPercentage` The percentage of the maximum motor speed at which the left motor is supposed to turn. A negative value corresponds to turning in the backwards direction:
2. `rightMotorSpeedPercentage` Analogous to the `leftMotorSpeedPercentage`, only for the right Motor.
3. `enable` Either 1 or 0. If it is 0 the motors are disabled and stand still.

These inputs are further described in table 3.1.

Number	Input Name	Datatype	Value	Effect
1	`leftMotorSpeedPercentage`	Double	$[-1, 1]$	1 Left motor full speed forwards 0 Left motor off -1 Left motor full speed backwards
2	`rightMotorSpeedPercentage`	Double	$[-1, 1]$	1 Right motor full speed forwards 0 Right motor off -1 Right motor full speed backwards
3	`enable`	Double	$\{0, 1\}$	1 Motors on 0 Motors off

Table 3.1: Inputs of the `setMotors` Simulink block

These inputs are then processed in Simulink to the format which is accepted by the program on the microprocessor. This is done by multiplying the speed percentages by 300 and rounding down (with the floor function) to the next integer. This is done, because the program on the microprocessor accepts only integer values in the range of $[-300, 300]$. Furthermore the signals are multiplied by the `enable` input and with the respective `*_motor_direction` variable which can be set to change the motor turning direction. Hereby `*` stands for `left` resp. `right` depending on the motor. This variables have to be loaded into the workspace and are needed because the motor direction changes if it is mounted inversely.

3.1.2 Microprocessor

The managing of the motor speeds on the microprocessor is mostly done by the code provided by Pololu.

To set the speed of the motors, a timer (`timer1`) is configured with a prescaler of 1, phase-correct PWM and a `TOP` value of 400. This means that the timer is counting from 0 up to the `TOP` value and then back to 0. When up-counting, the output is set low on the compare match, and during the down-counting the output is set high on the match. If the compare register is set to 0, the output is always low and if it is set to `TOP`, the output is always high. The compare register is set depending on the wanted speed and the output is directly connected to the `enable`-pin of the motor driver. Such a timer and the output is depicted in figure figure 3.2. As the `Atmega32U4` is

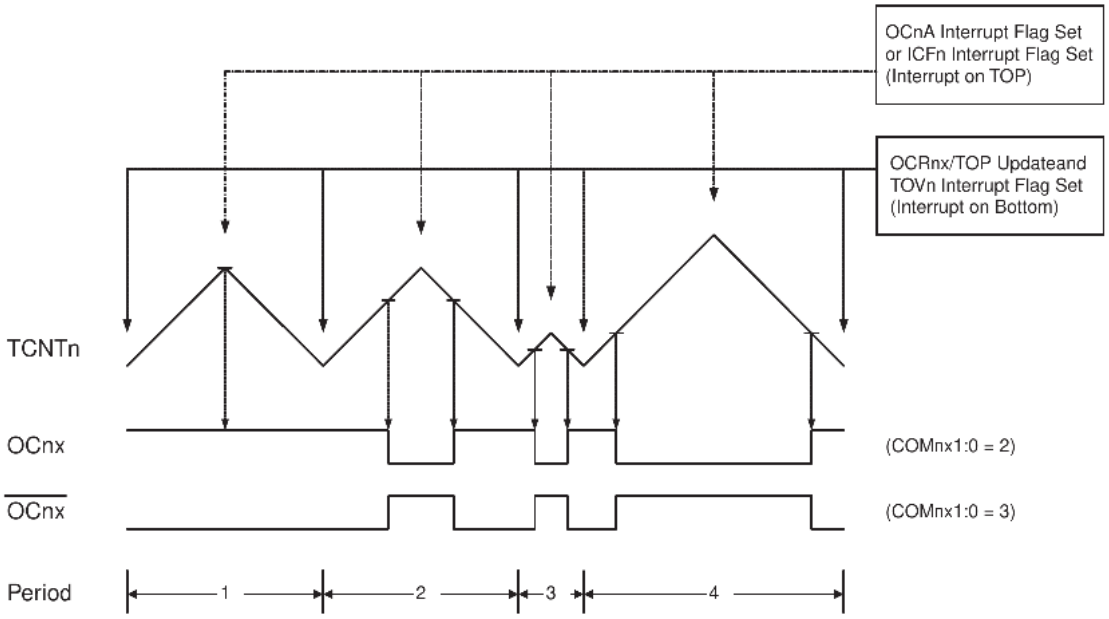

Figure 3.2: Example of the timer used for setting the motor speeds. `TCNTn` is the counter of the timer `n`, while `OCnx` is the output of timer `n` with the compare register `x`. The compare register is always reset at the lower turning point. This picture is taken from the `ATmega32U4` documentation by ATmel.

clocked at 16 MHz, the PWM frequency is calculated as

$$f = \frac{16\,\text{MHz}}{2 * 1 * 400} = 20\,\text{kHz} \tag{3.1}$$

where the 2 comes from the fact that the counter is going up and down, the 1 is the prescaling factor and the 400 is the `TOP` value.
As mentioned, the compare register is set depending on the wanted speed, meaning that the absolute value of the speed to set is written to the compare register, which results in the output being changed always when the timer equals the speed value.
For example if the speed value to set is 250, the output is high for the first 250 cycles of the period, then low for the next 150 cycles, then the counter changes to counting down and the next 150 cycles the output is low and for the last 250 the output is again high. This results in a ratio of

$$\frac{500}{800} = \frac{250}{400} \tag{3.2}$$

and thus the motor turns at $\frac{25}{40}$ of the maximally possible speed.
It is important to note that the timer has a top value of 400 but the maximum value to set the speed is only 300. This results in the maximum settable speed being only $\frac{3}{4}$ of the absolute maximum speed. This is intended to increase the lifetime of the motors. However, it is possible to change this behavior to allow values from -400 up to 400 and thus achieving higher motor speeds.

The procedure above only explains how the motor speed is set but not the direction. The direction is directly set by setting the phase input (`PH`) of the motor driver. This is done depending on the sign of the value to set. Additionally the library provides functions to flip the direction of the motors, for example if one is mounted the wrong direction. A rough description of the provided functions is summarized in table 3.2, detailed information is provided in the official documentation[3].

Output	Name	Arguments	Description
`static void`	`flipLeftMotor`	`(bool flip)`	Flips the direction of the left motor
`static void`	`flipRightMotor`	`(bool flip)`	Flips the direction of the right motor
`static void`	`setLeftSpeed`	`(int16_t speed)`	Sets the speed for the left motor
`static void`	`setRightSpeed`	`(int16_t speed)`	Sets the speed for the right motor
`static void`	`setSpeeds`	`(int16_t leftSpeed, int16_t rightSpeed)`	Sets the speed for both motors
`static void`	`allowTurbo`	`(bool turbo)`	Turns the turbo mode on or off.

Table 3.2: Available functions of the motor library provided by Pololu.The syntax is as used in C.

3.2 Set LEDs

The LEDs are directly connected to an output pin of the `Atmega32U4` and controlled by driving this pin high and low. As this is also done by the microprocessor itself, there is a Simulink-Block to send the data as well as a program part on the `Atmega32U4` which sets the LEDs after receiving the data.

3.2.1 Simulink

The states of the LEDs are set by the Simulink block `LEDs`. A scheme of this block is depicted in figure 3.3. The three inputs to this block are further described in table 3.3.

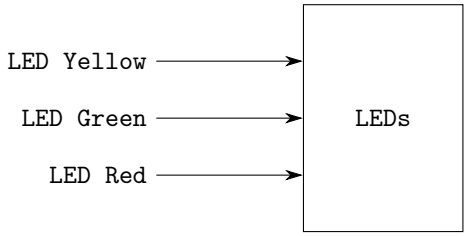

Figure 3.3: Inputs and outputs of the `LEDs` Simulink block.

The processing in Simulink is limited to converting the three boolean values to integers, where 1 represents true and 0 false. Then, these three values are sent to the microprocessor.

3.2.2 Microprocessor

Having received the values from the Raspberry Pi, the microcontroller sets the LEDs by putting a high or low output on the corresponding pins. This is done by three provided functions summarized in table 3.4.

[3] Available at `https://pololu.github.io/balboa-32u4-arduino-library/class_balboa32_u4_motors.html`

Number	Input Name	Datatype	Effect
1	`LED Yellow`	Boolean	**True** LED on **False** LED off
2	`LED Green`	Boolean	**True** LED on **False** LED off
3	`LED Red`	Boolean	**True** LED on **False** LED off

Table 3.3: Inputs of the `LEDs` Simulink block

Output	Name	Arguments	Description
`void`	`ledRed`	`(bool on)`	Turns the red led (`RX`) on or off
`void`	`ledYellow`	`(bool on)`	Turns the yellow led (on pin 13) on or off
`void`	`ledGreen`	`(bool on)`	Turns the green led (`TX`) on or off

Table 3.4: Available functions for setting the LEDs in the library provided by Pololu. The syntax is as used in C.

Hereby it is important to now, that turning on *does not* correspond to pulling the pin high for all leds. This is because the pins are also used for other purposes which is why the wiring on the board is different and the leds have to be accessed differently.

3.3 Read Encoders

The encoder signals are read in the `readEncoders` Simulink block, which reads the values from the microprocessor over I^2C. Most of the calculations from the encoder signal to the counter is done by the `Atmega32U4`.

3.3.1 Simulink

The Simulink block `readEncoders` reads the values from the microcontroller and outputs them directly. These values are formatted as `uint16` represent the number of encoder ticks which occur 12 times per revolution of the motor. A tick backwards counts negative and the counter wraps around if it exceeds the range of the datatype ($[0, 2^{16} - 1]$). The number of ticks starts at 0 at boot.
As the encoders register 12 ticks per revolution of the motor, one tick corresponds to an angle of

$$\frac{2\pi}{12}\,\text{rad} = 0.5236\,\text{rad} \tag{3.3}$$

which is quite a large angle. However, to get the wheel angle instead, the gearing of the motor itself and the mounted gearbox have to be considered leading to a wheel angle of

$$\frac{2\pi}{12 \cdot i_{\text{motor}} \cdot i_{\text{gearbox}}}\,\text{rad} \tag{3.4}$$

per tick, where i_{motor} and i_{gearbox} represent the gear ratio of the motor or gearbox respectively.
A schematic showing the outputs of this block is shown in figure 3.4 and the outputs are further described in table 3.5.

3.3.2 Microprocessor

The microprocessor is doing the calculations to get the ticks from the signals depicted in figure 2.1. This procedure is the same for both encoders and is described in the following:

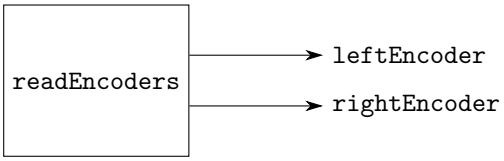

Figure 3.4: Inputs and outputs of the `readEncoders` Simulink block.

Number	Output Name	Datatype	Effect
1	`leftEncoder`	uint16	The number of ticks of the left encoder, wraps around if the range of the datatype is exceeded
2	`rightEncoder`	uint16	The number of ticks of the right encoder, wraps around if the range of the datatype is exceeded

Table 3.5: Outputs of the `readEncoders` Simulink block

The `Atmega32U4` receives two signals, the signal `B` and the `XOR`ed signal `A XOR B`. As the `XOR`ed signal has the property to change if either `A` or `B` changes. Therefore an interrupt is attached to the input pin of the `XOR`ed signal to capture a change of only one of the signals by using only one interrupt pin.
When the interrupt occurs, the first thing is to recover the value of the original signal `A`, which is done using the relation

$$\texttt{(A XOR B) XOR B = A}. \tag{3.5}$$

Then it is determined if the count was in the forward or backward direction. This requires some bookkeeping of the last values of both signals `A` and `B`, denoted as $\texttt{A}_{\text{last}}$ and $\texttt{B}_{\text{last}}$ respectively. The new values are named $\texttt{A}_{\text{new}}$ and $\texttt{B}_{\text{new}}$. These values are combined as

$$(\texttt{A}_{\text{last}}\ \texttt{XOR}\ \texttt{B}_{\text{new}}) - (\texttt{A}_{\text{new}}\ \texttt{XOR}\ \texttt{B}_{\text{last}}) \tag{3.6}$$

which yields 1 if `B` is rising before `A` and -1 if it is the other-way round, if the outputs of both `XOR`s are treated as 1 if `true` and 0 if `false`. As `B` leads `A` if the motor is turning forwards this value directly corresponds to the signed tick of the encoder.
Furthermore, the interrupt service routine (ISR) keeps track of an error, if both signals changed since the last execution of the ISR. This can happen if the ISR was not started soon enough because interrupts were disabled for a too long time.
There are several functions concerning the encoders provided by the class. They are described in table 3.6.

3.4 Read Wheel Encoders

The wheel encoders are directly attached to the Raspberry Pi using the I^2C bus. However, both sensors have initially the same address (`0x40`) resulting in an addressing conflict. To solve this issue, it is possible to change the 7 bit address of the sensors. This can be done in two ways:

1. The two least significant bits of the address are controlled by two `GPIO` input pins(`A1` and `A2`), if they are pulled up, the corresponding bit in the address is put to `1`. This allows to select out of four different addresses.

2. Furthermore, the other 5 bits can be set using the I^2C address register 21. Hereby, the 5 most significant bits of the address are equal to the value of the bits $0 \ldots 4$ of the address register. However, the most significant bit of the address is internally inverted, such that the initial register value `00000` results in the address part `10000`. However, if the register is read, still the stored value `00000` is returned.

Output	Name	Arguments	Description
`static void`	`init`	`()`	Initializes the encoders if necessary. Is called before each function making it unnecessary to call this function manually.
`static int16_t`	`getCountsLeft`	`()`	Returns the counts of the left encoder.
`static int16_t`	`getCountsRight`	`()`	Returns the counts of the right encoder.
`static int16_t`	`getCountsAndResetLeft`	`()`	Returns the counts of the left encoder and resets them.
`static int16_t`	`getCountsAndResetRight`	`()`	Returns the counts of the right encoder and resets them.
`static bool`	`checkErrorLeft`	`()`	Returns if an error occurred on the left encoder.
`static bool`	`checkErrorRight`	`()`	Returns if an error occurred on the right encoder.

Table 3.6: Available functions for setting the LEDs in the library provided by Pololu. The syntax is as used in C.

The second one especially helpful if many sensors are to be connected or if the pin on the microprocessor is used otherwise afterwards. But here the first approach is used to increase the I^2C address of the right wheel encoder by one. All other communication with the sensor is done directly by Simulink on the Raspberry Pi.

3.4.1 Simulink

The Simulink block `readWheelEncoders` directly reads the wheel angle from the wheel encoders without any configuration being necessary. The block then calculates the angle in radians from the data and outputs the values for both the left and right wheel encoder. A schematic of this block is represented in figure 3.5 and the outputs are further described in table 3.7. The calculation from

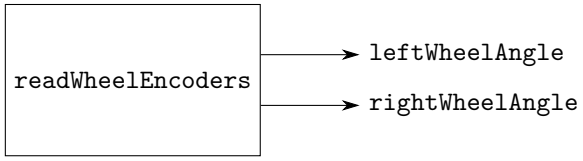

Figure 3.5: Outputs of the `readWheelEncoders` Simulink block.

Number	Output Name	Datatype	Effect
1	`leftAngle`	double	The absolute angle measured by the left encoder, wraps around after 2π
2	`rightAngle`	double	The absolute angle measured by the right encoder, wraps around after 2π

Table 3.7: Outputs of the `readWheelEncoders` Simulink block

the output of the sensors to the final angle proceeds as follows:

The data first gets read as two bytes and then the `14-bit` number representing the angle is extracted. This number is then converted to a double and multiplied by the sensitivity of 0.0219 /LSB. Then, the angle in degrees is converted to radians and a correction is done, which allows to set the positive angle direction. This final angle is then the output of the block.

3.5 Read Inertial Data

Similar to the reading of the wheel encoders in section 3.4, the inertial data gets read directly by the Raspberry Pi from the sensors. Here, the sensors are configurated by Simulink over I^2C and the microprocessor does nothing with this sensor.

3.5.1 Simulink

The inertial data is obtained using the Simulink block `readAccelGyro`, which has no input and two outputs as depicted in figure 3.6 and the outputs are described in table 3.8. This block configures

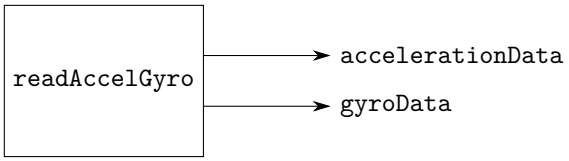

Figure 3.6: Outputs of the `readAccelGyro` Simulink block.

Number	Output Name	Datatype	Effect
1	`accelerationData`	Array of `int16`	The measurements from the accelerometer in the x,y and z direction as well as the full scale in this order in an array. The results are *not* in physical units, they first have to be multiplied by the sensitivity.
2	`gyroData`	Array of `int16`	The measurements from the gyroscope in the x,y and z direction as well as the full scale in this order in an array. The results are *not* in physical units, they first have to be multiplied by the sensitivity.

Table 3.8: Outputs of the `readWheelEncoders` Simulink block

the sensor such that it outputs both measurements with a frequency of 1.66 kHz. The full-scale of the gyroscope is set to ±2000 dps resulting in a sensitivity of 70 mdps/LSB and the accelerometer is configured to have a full-scale of ±2 g and a sensitivity of 0.061 mg/LSB. Additionally, the accelerometer is filtered with an anti-aliasing filter with a bandwidth of 50 Hz while the gyroscope measurement is output directly without any filtering.

3.6 Read Buttons

To get the information of the buttons, the microprocessor reads the button data and forwards them to the Raspberry Pi. Then, the values get processed and filtered in Simulink before they are output.

3.6.1 Simulink

The Simulink block `readButtons` is responsible for obtaining the button presses. It does not get any inputs and outputs three booleans representing the states of each button. These outputs are drawn in figure 3.7 and described in table 3.9. The processing in Simulink is as follows: First, the

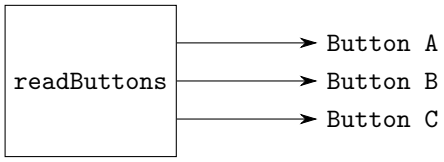

Figure 3.7: Outputs of the `readButtons` Simulink block.

Number	Output Name	Datatype	Effect
1	`Button A`	`boolean`	`True` if the button is pressed, `False` otherwise. Exception: Also `False` if all buttons are pressed.
2	`Button B`	`boolean`	`True` if the button is pressed, `False` otherwise. Exception: Also `False` if all buttons are pressed.
3	`Button C`	`boolean`	`True` if the button is pressed, `False` otherwise. Exception: Also `False` if all buttons are pressed.

Table 3.9: Outputs of the `readButtons` Simulink block

data from the microprocessor gets read over I^2C and the integer converted to a boolean. Then, the case where all three buttons are pressed is filtered out and the buttons are set to false in this case. This is done because such a measurement is often falsely happening when the motors are turning. Then, the buutton press is further filtered using a low-pass filter with a time constant of the variable `buttonFilterTau`, which has to be defined outside. This filtered signal is then compared to the value 0.5, if it is bigger, the button is considered as pressed and the output is `True`, otherwise the output is `False`.

3.6.2 Microprocessor

The microprocessor uses the library function by Pololu to read the state of the buttons and makes it available to be read over I^2C. The library function simply reads if the pin corresponding to the button is `high` or `low` and uses this to determine if the button is pressed or not. If `high` or `low` corresponds to the button being released is done depending on the button, as it is not the same for all buttons on the `Balboa32U4`.

3.7 Read Power Status

The power status is read by the `Atmega32U4` on an analog input. The measured value is then forwarded to the Raspberry Pi where the Simulink block simply reads the value and puts it out.

3.7.1 Simulink

The Simulink block `readBatteryMillivolts` reads the value provided by the microprocessor and outputs it without further calculations. A schematic is shown in figure 3.8 and the output is described in table 3.10.

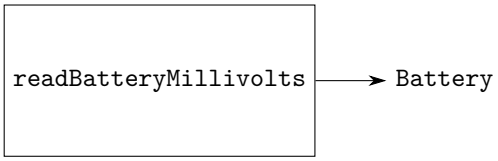

Figure 3.8: Outputs of the `readBatteryMillivolts` Simulink block.

Number	Output Name	Datatype	Effect
1	`Battery`	`uInt16`	The battery voltage in millivolts (mV). This is measured by an ADC on the `Atmega32U4` and if the reading is below 5500, the actual voltage might be significantly lower.

Table 3.10: Outputs of the `readBatteryMillivolts` Simulink block

3.7.2 Microprocessor

The `Atmega32U4` reads the voltage on an analog input pin which is connected to an ADC. To get the voltage, the microprocessor measures the voltage 8 times and takes the average of it. The measured voltage is a third of `VSW`, meaning that the pin is connected through a voltage divider to `VSW`. However, if the reading is below 5500 mV it might be significantly lower. This is because the reading is done relative to the potential difference between ground and the `Atmega32U4`'s input voltage and when the reading is below 5500 mV the voltage supply of the microprocessor might be lower than the usual 5 V.

Chapter 4

I²C Interface Description

The provided firmware for the `Atmega32U4`[1] can be used to provide an interface to the Raspberry Pi, forwarding the information of the connected sensors and actuators (without the I²C sensors which are directly accessible by the Raspberry Pi). This is helpful if the main controls are to be run on the Raspberry Pi.
The provided interface is using the I²C protocol and the `Atmega32U4` is configured as a slave device and accessible under the address 14h (00010100b). It uses the Little Endian byte ordering (Most significant bit first). In the following part of this chapter, the available registers are explained.

4.1 Register Mapping

The table below provides an overview of the 8-bit registers and their addresses. A description of the used and tested registers is provided in section 4.2.

Name	Type	Register Address		Default	Comment
		Hex	Binary		
LED_YELLOW	r/w	00	0000 0000	0000 0000	LED Registers
LED_GREEN	r/w	01	0000 0001	0000 0000	
LED_RED	r/w	02	0000 0010	0000 0000	
BUTTON_A	r	03	0000 0011	Output	Button Registers
BUTTON_B	r	04	0000 0100	Output	
BUTTON_C	r	05	0000 0101	Output	
MOTOR_LEFT_L	r/w	06	0000 0110	0000 0000	Motor Registers
MOTOR_LEFT_H	r/w	07	0000 0111	0000 0000	
MOTOR_RIGHT_L	r/w	08	0000 1000	0000 0000	
MOTOR_RIGHT_H	r/w	09	0000 1001	0000 0000	
BATTERY_L	r	0A	0000 1010	Output	Battery Voltage Registers
BATTERY_H	r	0B	0000 1011	Output	
ANALOG_0_L	r	0C	0000 1100	Output	Analog Input Registers
ANALOG_0_H	r	0D	0000 1101	Output	

[1] See appendix A.1.

ANALOG_1_L	r	0E	0000 1110	Output	
ANALOG_1_H	r	0F	0000 1111	Output	
ANALOG_2_L	r	10	0001 0000	Output	
ANALOG_2_H	r	11	0001 0001	Output	
ANALOG_3_L	r	12	0001 0010	Output	
ANALOG_3_H	r	13	0001 0011	Output	
ANALOG_4_L	r	14	0001 0100	Output	
ANALOG_4_H	r	15	0001 0101	Output	
ANALOG_5_L	r	16	0001 0110	Output	
ANALOG_5_H	r	17	0001 0111	Output	
PLAY_NOTES	r/w	18	0001 1000	0000 0000	Start playing notes
NOTES_0	r/w	19	0001 1001	0000 0000	Set Notes
NOTES_1	r/w	1A	0001 1010	0000 0000	
NOTES_2	r/w	1B	0001 1011	0000 0000	
NOTES_3	r/w	1C	0001 1100	0000 0000	
NOTES_4	r/w	1D	0001 1101	0000 0000	
NOTES_5	r/w	1E	0001 1110	0000 0000	
NOTES_6	r/w	1F	0001 1111	0000 0000	
NOTES_7	r/w	20	0010 0000	0000 0000	
NOTES_8	r/w	21	0010 0001	0000 0000	
NOTES_9	r/w	22	0010 0010	0000 0000	
NOTES_10	r/w	23	0010 0011	0000 0000	
NOTES_11	r/w	24	0010 0100	0000 0000	
NOTES_12	r/w	25	0010 0101	0000 0000	
NOTES_13	r/w	26	0010 0110	0000 0000	
ENCODER_LEFT_L	r	27	0010 0111	Output	Encoders Registers
ENCODER_LEFT_H	r	28	0010 1000	Output	
ENCODER_RIGHT_L	r	29	0010 1001	Output	
ENCODER_RIGHT_H	r	2A	0010 1010	Output	
IS_PLAYING	r	2B	0010 1011	Output	If currently playing
PLAY_FUGUE	r/w	2C	0010 1100	0000 0000	Start playing the fugue

Table 4.1: I^2C Registers Address Map

4.2 Registers Description

In this section the different I^2C registers in the provided firmware for the `Atmega32U4` are described with the values they admit and what effect they have. Only the registers necessary for the stabilization of the Sigi are described. Theoretically further registers are configured for reading of analog inputs or playing notes using the buzzer. These however have not been tested properly and this part should be revised if this functionality is wanted.

The inertial module and magnetometer are also read out using I²C, for their configuration and readout it is referred to the corresponding datasheet (referenced in section 2.1.5, section 2.1.6 respectively).

4.2.1 LED Registers (00h - 02h)

These registers control the state of the three LEDs on board. Each is controlling one LED according to table 4.1. All of them take either 0 or 1 as values as described in table 4.2.

Value	Result
0	LED off
1	LED on

Table 4.2: LED Registers Values

4.2.2 Button Registers (03h - 05h)

The button registers are used to get the states of the buttons and are read-only. Each is providing the state of one button with the buttons assigned according to table 4.1. The registers take values of 0 or 1 according to table 4.3.

Value	Meaning
0	Button Released
1	Button Pressed

Table 4.3: Button Registers Values

4.2.3 Motor Registers (06h-09h)

The motor voltage can be set over I²C by sending a signed `int16` value in $[-300, 300]$. This is done using the registers 06h - 09h as in table 4.1. The combined registers (high and low byte together) can take the values in table 4.4.

Value	Meaning
-300	Max voltage backwards
[-299 ... -1]	Linear voltage decrease
0	No voltage (Stop)
[1 ... 299]	Linear voltage increase
300	Max voltage forwards

Table 4.4: Motor Registers Values (High and low byte together)

4.2.4 Battery Voltage Registers (0Ah-0Bh)

This register is used to read the battery voltage on the Sigi which is split into high and low byte as described in table 4.1. These combined registers contain the value of the battery voltage in millivolts (mV) encoded as `uint16`. It is important to note that if the output is below 5500 mV, the actual voltage might be significantly lower.

4.2.5 Encoder Counts Registers (27h-2Ah)

The encoders can be read using the four registers 27h-2Ah which have the functions according to table 4.1. They are read in pairs of two as signed `int16` value, representing the number of encoder ticks in forward direction. The mounted encoders register 12 counts per revolution of the motor, the wheel rotation can from this be calculated using the gear ratios as in section 2.1.4.

Chapter 5

Setup of the Sigi2.0

To be able to use the Sigi2.0 hardware, it first has to be assembled and set up. The assemby is described in the "Pololu Balboa 32U4 Balancing Robots User's Guide" by Pololu[1]. The setup afterwards is specific to the Sigi2.0 and generally consists of three steps:

1. Load the software image to the Raspberry Pi.
2. Flash the Program onto the Atmel Atmega32U4 on the Pololu Balboa32U4.
3. Deploy Simulink on the hardware.

However, it is important to note, that the first two points is only necessary once per Sigi2.0, while the last one has to be done once for each Simulink model.

5.1 Load the software image to the Raspberry Pi & Configure Simulink

Write the iso-Image onto the SD-Card designed to use in the Sigi2.0. This can be done e.g. with the command line utility GNU dd or balenaEtcher. Then, insert the SD-Card into the Raspberry Pi and the Raspi can be started. From now on, it is possible to connect to the Sigi over WLAN and SSH with the following data:

WLAN: SSID `Sigi#Nr` (Nr represents the 3-digit number of the Sigi at hand)

WLAN: Password — (None)

Raspi: IP-Address `192.168.9.1`

Raspi: Username `pi`

Raspi: Password `raspberry`

Then, install the Simulink Support Package for Raspberry Pi[2] from the Addon-Manager in Matlab/Simulink. This package is necessary for some Raspberry Pi-specific Simulink blocks.

[1] Available at `https://www.pololu.com/docs/pdf/0J70/balboa_32u4_robot.pdf`

[2] Available at `https://ch.mathworks.com/matlabcentral/fileexchange/40313-simulink-support-package-for-raspberry-pi-hardware`

5.2 Flash the Pololu Balboa32U4

5.2.1 Using the Arduino IDE

The easiest way to this is with the Arduino IDE available here. Having installed the software one proceeds as follows:

1. Move the `*.ino`-file to be flashed into an equally named folder (without the `.ino`).

2. Then open the file using Arduino.

3. In Arduino, open the Library management (*Sketch* → *Include Library* → *Manage Libraries*), search for the libraries `Balboa32U4` and `PololuRPISlave` and install them.

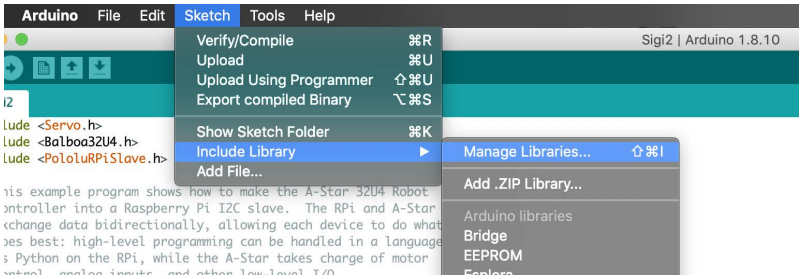

Figure 5.1: Install the libraries

4. Connect the Pololu Balboa32U4 to the PC over USB.

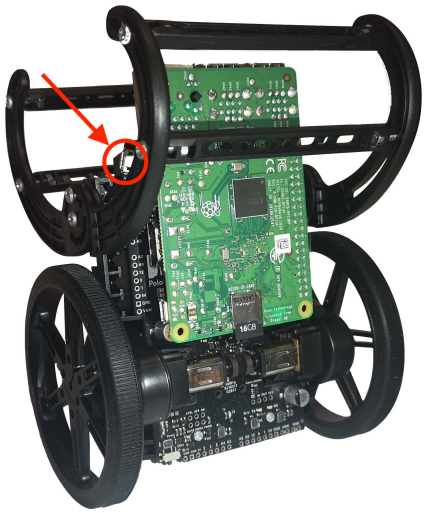

Figure 5.2: USB Location on the Sigi

5. Choose `Board: "Arduino Leonardo"` in Arduino under *Tools* → *Board:'...'*

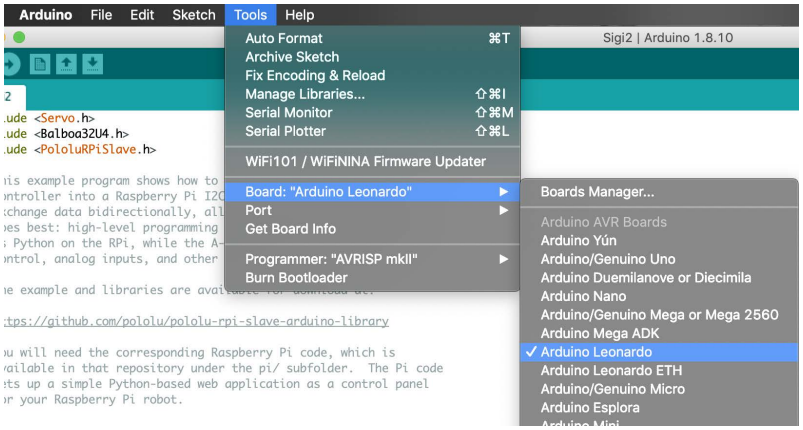

Figure 5.3: Choose the correct board

6. Select the Balboa under *Tools* → *Port*
7. Upload the program by clicking *Upload* (Arrow to the right) in the command bar of the IDE.

As soon as *Done uploading* appears above the terminal, the process is finished and the USB connection can be terminated.

5.2.2 Using avr-gcc

Alternatively, e.g. if there is a problem with the Arduino IDE, the board can be programmed directly using the `avr-gcc` toolchain and `AVRDUDE`. For this the following programs have to be installed, depending on the used operating system:

Windows
- WinAVR
- AVRDUDE

Mac OS X CrossPack for AVR Development

Linux `avr-gcc`, `avr-libc` and `AVRDUDE`. In Ubuntu, they are available in the official package repositories.

Then, the Pololu Balboa32U4 is flashed as follows:

1. Open the commandline and navigate to the folder with the `*.hex`-file to be flashed. If this file isn't existing yet, it can be created as follows:
   (a) Open the corresponding `*.ino` file in the Arduino IDE and proceed as in section 5.2.1 but leaving out steps 4 and 7.
   (b) Then, compile the program by clicking *Verify* (the tick mark) in the command bar of the IDE.
   (c) After *Done compiling* appears above the terminal, the compilation is successfully finished and the `*.hex`-file has been created
   (d) Find the `*.hex` file in a temporary folder somewhere on your system. This folder depends on the operating system. In Linux Mint for example, the file is located in `/tmp/arduino_build_xxxxx`, where `xxxxx` is some number assigned by the Arduino IDE. The file inside is named as `*.ino.hex`.
   (e) Copy this file out of this temporary folder into the place where you want to store it.
2. Connect the Balboa32U4 to the computer using USB.

3. Then, put the following command in the commandline and replace `PORT` by the actual "port" on which the Balboa32U4 is connected. (On Windows, `\\\\.\\GLOBALROOT\\Device\\USBSER000` should work as long as the Balboa is the only connected USB-device.) **Do not execute the command yet.**:
   `avrdude -p atmega32u4 -c avr109 -P PORT -D -U flash:w:Sigi2.hex`
4. Start the bootloader on the Balboa32U4 by pressing the reset button twice during 750ms and start the prepared command within 8 seconds. If one misses the 8s interval, the Balboa32U4 exits the bootloader which would have to be reentered for flashing.

5.3 Deploy Simulink on the Hardware

Then the last step is to configure Simulink to start the model on the hardware which is done with the following steps:

1. Open the model to deploy on the hardware in Simulink.
2. Go to *Tools → Run on Target Hardware → Options...*
3. Select the `Raspberry Pi` as Hardware Board in the *Hardware Implementation* tab.
4. Expand the area *Target hardware resources* below and fill in the Sigi parameters in the *Board Parameters*. The values are:

   Device Address `192.168.9.1`

   Username `pi`

   Password `raspberry`
5. Execute the model by either clicking *Deploy to Hardware* or selecting *External Mode* in the dropdown on the right of the execute button and click the usual execute button.

Appendix A

Code

A.1 Code for the Atmega32U4

A.1.1 Sigi2.ino

```
#include <Servo.h>
#include <Balboa32U4.h>
#include <PololuRPiSlave.h>

/* This example program shows how to make the A-Star 32U4 Robot
 * Controller into a Raspberry Pi I2C slave.  The RPi and A-Star can
 * exchange data bidirectionally, allowing each device to do what it
 * does best: high-level programming can be handled in a language such
 * as Python on the RPi, while the A-Star takes charge of motor
 * control, analog inputs, and other low-level I/O.
 *
 * The example and libraries are available for download at:
 *
 * https://github.com/pololu/pololu-rpi-slave-arduino-library
 *
 * You will need the corresponding Raspberry Pi code, which is
 * available in that repository under the pi/ subfolder.  The Pi code
 * sets up a simple Python-based web application as a control panel
 * for your Raspberry Pi robot.
 */

const char fugue[] PROGMEM =
  "! O5 L16 agafaea dac+adaea fa<aa<bac#a dac#adaea f"
  "O6 dcd<b-d<ad<g d<f+d<gd<ad<b- d<dd<ed<f+d<g d<f+d<gd<ad"
  "L8 MS <b-d<b-d MLe-<ge-<g MSc<ac<a ML d<fd<f O5 MS b-gb-g"
  "ML >c#e>c#e MS afaf ML gc#gc# MS fdfd ML e<b-e<b-"
  "O6 L16ragafaea dac#adaea fa<aa<bac#a dac#adaea faeadaca"
  "<b-acadg<b-g egdgcg<b-g <ag<b-gcf<af dfcf<b-f<af"
  "<gf<af<b-e<ge c#e<b-e<ae<ge <fe<ge<ad<fd"
  "O5 e>ee>ef>df>d b->c#b->c#a>df>d e>ee>ef>df>d"
  "e>d>c#>db>d>c#b >c#agaegfe f O6 dc#dfdc#<b c#4";

// Custom data structure that we will use for interpreting the buffer.
```

```
34 // We recommend keeping this under 64 bytes total.  If you change the
35 // data format, make sure to update the corresponding code in
36 // a_star.py on the Raspberry Pi.
37
38 struct Data
39 {
40   bool yellow, green, red;
41   bool buttonA, buttonB, buttonC;
42
43   int16_t leftMotor, rightMotor;
44   uint16_t batteryMillivolts;
45   uint16_t analog[6];
46
47   bool playNotes;
48   char notes[14];
49
50   // Encoders are unused in this example.
51   int16_t leftEncoder, rightEncoder;
52   bool playFugue;
53   bool notPlaying;
54 };
55
56 PololuRPiSlave<struct Data,5> slave;
57 Balboa32U4Buzzer buzzer;
58 Balboa32U4Motors motors;
59 Balboa32U4Encoders encoders;
60 Balboa32U4ButtonA buttonA;
61 Balboa32U4ButtonB buttonB;
62 Balboa32U4ButtonC buttonC;
63
64 void setup()
65 {
66   // Set up the slave at I2C address 20.
67   slave.init(20);
68
69   // Play startup sound.
70   //buzzer.play("v10>>g16>>>c16");
71 }
72
73 void loop()
74 {
75   // Call updateBuffer() before using the buffer, to get the latest
76   // data including recent master writes.
77   slave.updateBuffer();
78
79   // Write various values into the data structure.
80   slave.buffer.buttonA = buttonA.isPressed();
81   slave.buffer.buttonB = buttonB.isPressed();
82   slave.buffer.buttonC = buttonC.isPressed();
83
84   // Change this to readBatteryMillivoltsLV() for the LV model.
85   slave.buffer.batteryMillivolts = readBatteryMillivolts();
86
87
```

```
88    slave.buffer.leftEncoder = encoders.getCountsLeft();
89    slave.buffer.rightEncoder = encoders.getCountsRight();
90
91    slave.buffer.notPlaying = buzzer.isPlaying();
92
93    for(uint8_t i=0; i<6; i++)
94    {
95      slave.buffer.analog[i] = analogRead(i);
96    }
97
98    // READING the buffer is allowed before or after finalizeWrites().
99    ledYellow(slave.buffer.yellow);
100   ledGreen(slave.buffer.green);
101   ledRed(slave.buffer.red);
102   motors.setSpeeds(slave.buffer.leftMotor, slave.buffer.rightMotor);
103
104   // Playing music involves both reading and writing, since we only
105   // want to do it once.
106   static bool startedPlaying = false;
107
108   if(slave.buffer.playNotes && !startedPlaying)
109   {
110     buzzer.play(slave.buffer.notes);
111     startedPlaying = true;
112   }
113   else if (startedPlaying && !buzzer.isPlaying())
114   {
115     slave.buffer.playNotes = false;
116     startedPlaying = false;
117   }
118
119   if(slave.buffer.playFugue){
120     slave.buffer.playFugue = false;
121     buzzer.playFromProgramSpace(fugue);
122   }
123
124   // When you are done WRITING, call finalizeWrites() to make modified
125   // data available to I2C master.
126   slave.finalizeWrites();
127 }
```

Bibliography

[1] W. Florian, "Modeling and control of backlash on an inverted pendulum on wheels," Studies on mechatronics and Bachelor's thesis, ETH Zurich, Institute for Dynamic Systems and Control, Zürich, Switzerland, Jul. 2020, IDSC-CO-RHe-14.