This excellent tutorial by Marcel Dür tells how to set up and use the Raspberry Pi computer with ooRexx and BSF4ooRexx (which enables you to use Java libraries without having to learn Java) –

The goal of this thesis is to show that a Raspberry Pi can be programmed using ooRexx and BSF4ooRexx. Practical Nutshell examples are used to illustrate what needs to be considered and how programs can be implemented. The required software is explained and how to use it. ooRexx is a simple programming language that allows the user to use Java libraries with the help of BSF4ooRexx and to program with them without having Java knowledge. Raspberry Pi is a small computer, which was developed to give especially young people access to a computer and the possibility to program something. In combination, a broad mass with little expertise in ooRexx and a small budget can develop programs themselves and use this as an entry into the world of programming.

WIRTSCHAFTSUNIVERSITÄT WIEN
Vienna University of Economics and Business

Bachelor Thesis

Titel of Bachelor Thesis (english)	pi4oorexx - An introduction to programming the Raspberry Pi with ooRexx and BSF4ooRexx
Titel of Bachelor Thesis (german)	pi4oorexx - Eine Einführung zur Programmierung des Raspberry Pi mit ooRexx und BSF4ooRexx
Author (last name, first name):	Dür Marcel
Student ID number:	00925194
Degree program:	Bachelor of Science (WU), BSc (WU)
Examiner (degree, first name, last name):	ao.Univ.Prof. Mag.Dr.rer.soc.oec. Rony G. Flatscher

I hereby declare that:

1. I have written this Bachelor thesis myself, independently and without the aid of unfair or unauthorized resources. Whenever content has been taken directly or indirectly from other sources, this has been indicated and the source referenced.

2. This Bachelor Thesis has not been previously presented as an examination paper in this or any other form in Austria or abroad.

3. This Bachelor Thesis is identical with the thesis assessed by the examiner.

4. (only applicable if the thesis was written by more than one author): this Bachelor thesis was written together with

   The individual contributions of each writer as well as the co-written passages have been indicated.

Date

Unterschrift

Contents

List of Figures

List of Tables

Listings

List of Abbreviations

API	Application Programming Interface
ARM	Acorn RISC Machines
BSF4ooRexx	Bean Scripting Framework for Open Object Rexx
CLI	Command Line Interface
CRC	Cyclic Redundancy Check
DIMM	Dual Inline Memory Module
EEPROM	Electrically Erasable Programmable Read-Only Memory
GND	Ground
GPIO	General Purpose Input/Output
GPS	Global Positioning System
HAT	Hardware Attached on Top
HDMI	High Definition Multimedia Interface
I^{2}C	Inter-Integrated Circuit
ID	Identity document
JAR	Java Archive
JDK	Java Development Kit
JRE	Java Runtime Environment
JVM	Java virtual machine
LCD	Liquid-crystal display
LED	Light-emitting diode
MOSI	Master Output, Slave Input
MISO	Master Input, Slave Output
NMEA	National Marine Electronics Association
ooRexx	Open Object Restructured Extended Executor
PWM	Pulse Width Modulation
RAM	Random-Access Memory
RFID	Radio-frequency identification
SAA	Systems Application Architecture)
SoC	System on a Chip
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver/Transmitter
VCC	Voltage common collector
WPI	Wiring Pi

Abstract

The goal of this thesis is to show that a Raspberry Pi can be programmed using ooRexx and BSF4ooRexx. Practical Nutshell examples are used to illustrate what needs to be considered and how programs can be implemented. The required software is explained and how to use it. ooRexx is a simple programming language that allows the user to use Java libraries with the help of BSF4ooRexx and to program with them without having Java knowledge. Raspberry Pi is a small computer, which was developed to give especially young people access to a computer and the possibility to program something. In combination, a broad mass with little expertise in ooRexx and a small budget can develop programs themselves and use this as an entry into the world of programming.

1 Introduction

This chapter briefly introduces the topic, presents the objective and research question, and outlines the methodological approach. Furthermore, an overview of the present bachelor thesis is given.

1.1 Initial Situation and Problem Definition

The Raspberry Pi was developed in 2012 by the Raspberry Pi Foundation to offer young people an inexpensive computer with which they can learn programming and experiment. Since programming was the main focus during the development of Raspberry Pi, there is a GPIO (General Purpose Input Output) strip, which can be used to read out or control sensors, displays, motors or LEDs.[bit19]
Through the Business Programming I and II lecture with Professor Flatscher at the WU Vienna, the author got to know the programming language ooRexx. Since the author has already programmed in other languages, this simpler use of Java through ooRexx or BSF4ooRexx was very interesting. The lecture aroused the author's interest, so the author attended a symposium of the Rexx Language Association in November 2021. Especially the contribution of Tony Dycks "Stable RPM Based Linux Distros for the Raspberry Pi 4" stood out. Among other things, he discussed how ooRexx and BSF4ooRexx can be installed and used on the Raspberry Pi 4.[Dyc21]

1.2 Target Setting

The author does not only want to install ooRexx or BSF4ooRexx on the Raspberry Pi, as presented in the symposium, but also to make it useful for everyday life. For this reason, examples are shown in this bachelor thesis, which are useful in everyday life, such as an RFID Attendance System. However, the basis for the control of various sensors is also shown in order to offer a variety of possible use cases.
In this bachelor thesis the following research question is answered:
"Is it possible to use ooRexx and BSF4ooRexx to program and control sensors via the GPIO interface on the Raspberry Pi?"
The research question is answered using programs that the author has programmed.

1.3 Methodical Approach

The presented bachelor thesis is, in the broadest sense, a literature work. It is quoted from relevant literature and journals. Also significant websites were consulted as a source. However, the core of the thesis is the programming of programs on the Raspberry PI, which is why only a small part of the thesis is dedicated to theory. For the programs, data sheets are quoted, which can be read online at any time. All sources are traceable and verifiable. The author pays attention to a proper reproduction of the contents for all sources, which also means a complete bibliography as well as a correct citation.

1.4 Structure of the Bachelor Thesis

At the beginning of this bachelor thesis an introduction is given. This consists of the initial situation, in which the author explains his motives for choosing the topic of this thesis, an objective, which presents the goal of this thesis and the research question, the methodological approach and the structure of the thesis. In chapter 2, the required software is discussed. A short overview of the respective software is given and advantages and/or disadvantages are dealt with. The third chapter deals in detail with Raspberry Pi. The different versions and interfaces are discussed. The next chapter shows how to install and configure the required software. It is explained step by step what has to be done. Chapter 5 shows the examples worked out by the author, so called nutshell examples, and thus forms the main part of the thesis. Each example is described exactly and shown with pictures and source code. The last chapter is the Conclusio. The author summarizes the findings of the thesis and his "lessons learned" during the programming. Furthermore the author gives an outlook on further possible applications of BSF4ooRexx on the Raspberry Pi.

2 Required Software

To implement the examples presented in this bachelor thesis, the use of various software is necessary. This chapter goes into more detail about the individual software components and gives a brief overview of them.

2.1 Rexx and ooRexx

Rexx is a programming language designed by Mike Cowlishaw. It is purpose was to make programming easier because the high-quality programs are written in a simple and understandable language.[Cow90]
Algorithms written in Rexx are written in a structured and clear way. Everybody should be able to use it. Furthermore, Rexx is independent, which means that is works with several system software.[Cow90]
It was designed to make the manipulation of objects, such as numbers and words, easier. Rexx can call programs and functions which are written in other languages although it was designed to be independent of its supporting system software. Rexx can be used to write anything from simple to large programs. [Ass21]
Rexx offers an English-like language which makes it easy to learn and use it, fewer rules, built-in functions and methods, typeless variables as variables can hold any kind of object, string handling, decimal arithmetic instead of binary arithmetic, clear error messages and very good debugging[Ass15]
Rexx was developed in two phases. The first was about a rapid evolution in an experimental environment and the second was about cautious enhancements. Mike Cowlishaw worked on phase one for several years and finished it in 1982 at IBM. After that, numerous implementations from different suppliers followed. Rexx became the producers language for the SAA (Systems Application Architecture) at IBM, which was a big milestone. As Rexx was used for commercial purposes a stable language definition was necessary. Therefore, the characteristically radical changes from the beginning were no longer possible. Luckily due to the rapid evolution in the first years such radical changes are no longer needed because of small adjustments and changes. Over the years Rexx stayed approachable and small.[Cow90]

In Listing 1 you can see a short example of the, already mentioned, simple and understandable language. In this example we iterate through a loop, and on each pass we output the variable ”i” with the ”say ” command. This output would result in 1,2,3,4

```
do i=1 to 4
  say i
end
```

Listing 1: Rexx Sample

ooRexx is short for Open Object Restructured Extended Executor. It adds object orientation to Rexx, it is Open Source and it is managed by the Rexx Language Association (RexxLA). It is an enhancement of the classic Rexx as it is a “full-featured programming

language which has a human-oriented syntax". Due to the Open Object Rexx-Interpreter programs can be written procedurally as well as object oriented. Some of its main benefits are the easy usage and learnability, the ability to issue commands to multiple environments, the full object orientation, and the powerful functions.[Ass15]
It was first published in spring 2005 and it is actively developed since then. Today it is available for the most important operating systems in a precompiled version, e.g. Linux, Windows and MacOSX[Fla12]
Problems are solved by identifying and classifying objects that are related to the problem. Actions and behaviours of these objects are determined. Finally, the instructions to generate classes are written, the objects are generated and the actions are implemented. "The main program consists of instructions that send messages to objects." Many advantages are associated with the object-oriented technology:

- Objects are modeled to simplify the design
- Code can be reused more often
- Rapid prototyping
- Higher quality of components
- Reduced and easier maintenance
- Less costs
- Increased adaptability and scalability

[Ass15]

2.2 BSF4ooRexx

BSF4ooRexx stands for Bean Scripting Framework for ooRexx and it extends the functionality of ooRexx, so that it can interact directly with Java objects. Everything that is available in Java is also available for ooRexx.[Fla13]
It is an external Rexx function package. This means that its functions can be accessed from Rexx programs. To simplify the interaction with Java the entire Java class library and the interaction with Java objects were masked. This created the impression that these are ooRexx class libraries and ooRexx objects and programmers only have to send ooRexx messages. Access to Java using an external Rexx function package was the original intention. It should be executable on all platforms. BSF4oorexx provides several advantages for programmers. The author lists a few of them here:

- There is no need for additional external function packages if the function is already available in the Java runtime environment(JRE)
- All JRE Java classes, which are executable unchanged on all operating systems supported by Java, can be used

- Any Java class library can be used directly

- Also abstract methods can be used when the implementation is done in ooRexx which realizes a callback mechanism from Java to ooRexx.

- All information systems can be controlled via ooRexx as long as they have Java programming interfaces. This is independent of the installed operating system.

- Java applications can execute Rexx scripts under their own control.

[Fla12]

2.3 Java

Java is a object-oriented programming language. It was first released by Sun Microsystems in 1995. It can be downloaded for free and numerous applications and websites only work if Java is installed on the end device.[Java]
Originally it was designed for interactive television, but it was too advanced at that time. Java is named after Java coffee. It intends to let programmers “write one, run anywhere (WORA)”, which means that compiled Java code can run on all Java supporting platforms without recompiling. Typically, Java applications are compiled to bytecode. The syntax is similar to C and C++, but has fewer low-level facilities. In May 2007 Java released its Java virtual machine (JVM) as an open-source software. In the years 2009/2010 Oracle acquired Sun Microsystems.[Javb]
Java has five principles:

- Simplicity, object-orientation, familiarity

- Robustness and security

- Architecture-neutral and portable

- Execute with high performance

- Interpretable, dynamic and threaded

[Oraa]
A Java platform facilitates the developing and running of programs written in Java language. The platform includes an execution engine (which is a virtual machine - JVM), a compiler and some libraries. The virtual machine is the heart of the Java platform. It executes Java bytecode programs. These are the same, independent of the hardware or the operating system. JVM contains a compiler which translates Java bytecode intro native processor instructions just in time and catches the native code during execution. This makes the Java application run as fast as the native programs. Because of the bytecode Java programs run on any platform that has a virtual machine. The Java platform provides a set of libraries which contains many of the standard functions of other operating systems. The

Java libraries provides a common set of functions, an abstract interface to tasks that depend on the hardware and operating system, and it provides a substitute or a consistent way to check for the presence of a specific feature if the underlying platform does not support all features. The Java Development Kit (JDK) contains a Java compiler, the Java Runtime Environment (JRE), and many important development tools. The JRE contains a JVM, the standard library, a configuration tool, and a browser plug-in. It is the most installed Java environment. [Orab]

2.4 Raspberry Pi OS

An own operating system for the Raspberry Pi was developed, which is based on Debian Linux and optimized for the Raspberry Pi. The name of the operating system is Raspberry Pi OS or was formerly called Raspbian. The latest version of Raspberry Pi OS is based on Debian 11 Bullseye and was released in August 2021. Other operating systems for the Raspberry Pi are also offered, such as Ubuntu.[Lon21]

2.5 Pi4J

Pi4j is an open source project started by Robert Savage and Daniel Sendula in 2012.[Del20] The goal of this project is to develop a user-friendly Java library to provide low level access to the hardware of the Raspberry Pi and other hardware. For this, the library handles the communication with the native low level libraries to simplify programming.(see Figure 1) There are currently two versions. Version 1.4 was released on 3/3/2021 and is also the last release of this version. Version 1.4 uses the WiringPi scheme for pinout. The more current version is version 2, which is still under development. The goal of the second version is to focus on the communication with the hardware of the Raspberry Pi. No drivers for specific sensors are provided. However, these can be embedded as plug-ins in version 2. Version

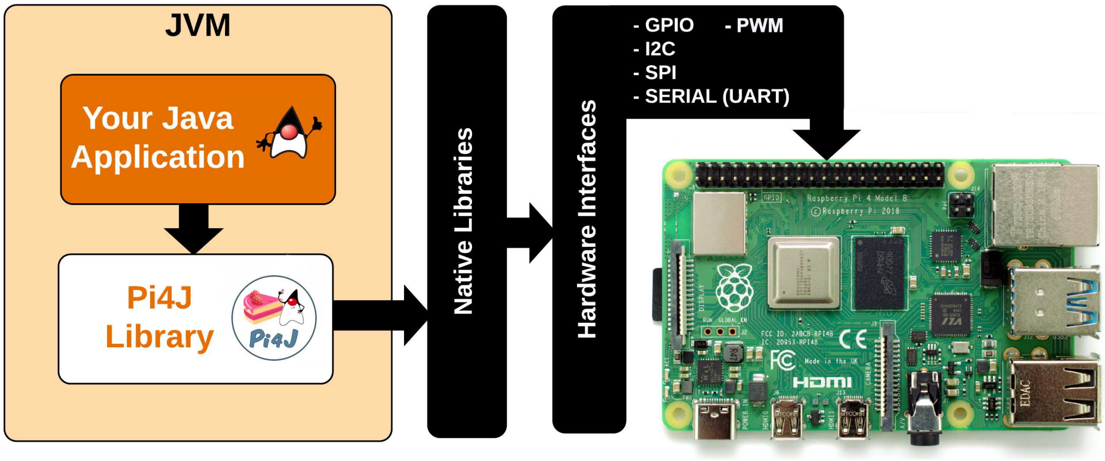


Figure 1: Pi4J API [Del20]

2 uses the BCM scheme, as from this version pigpio is used to drive the GPIO instead of WiringPi. Since the WiringPi project has been discontinued. [Pi4] However, this bachelor thesis will continue to use version 1.4, since version 2 has no advantage at the time of writing. Version 1.4 will provide the best support for connecting devices until a new Raspberry Pi is released (the latest version is the Raspberry Pi 4 or the Pi Zero 2).

2.6 WiringPi

WiringPi is a library developed in the C programming language for the Raspberry Pi to provide access to the GPIO interface. The library was developed by Gordon Henderson, who however stopped the development of the library in August 2019. In this work the open source version from Github(`https://github.com/WiringPi/WiringPi`) is used, because the used Pi4j version 1.4 is based on this library. The WiringPi library supports all Raspberry Pi models up to and including the Raspberry Pi Model 4B and Pi Zero 2.[Hen22]

2.7 pigpio

Pigpio is like WiringPi a library to get access to the GPIO interface of the Raspberry Pi. The pigpio library, unlike the WiringPi library, is still supported. The Pi4j version 2 is now based on this library. About the developer of this library is not known except for his nickname ”joan”.[Pi4]

2.8 I^{2}C Tools for Linux

The I^{2}C tools for linux package is a collection of programs that can be used for communication over the I^{2}C bus. For this the tools i2cdetect, i2cset and i2cset are used.[Del21]

2.9 pi4oorexx

The author has created his own JAR archive, which consists of ready-made Java classes that provide ready-made drivers of various used components. Drivers for the following components are included in the pi4oorexx.jar file:

- BME280 - temperature/pressure/humidity sensor
- BH1750 - ambient light sensor
- LC-Display I2C - hd44780 Driver
- DS18B20 - 1-wire temperature sensor
- MFRC522 - RFID module
- MAX7219 - LED Matrix driver

For each of these classes a help function has been implemented, which can be output with "∼help" via ooRexx, listing all available functions and the wiring.

By using this archive the access to the above mentioned sensors is possible with a few lines of code. For the use of this archive BSF4ooRexx and Pi4J are necessary.

3 Raspberry Pi

The Raspberry Pi is a single-board computer in credit card format developed by the Raspberry Pi Foundation. The first Raspberry pi was released in 2012. The idea behind this development was to provide people with affordable hardware to encourage them to learn a programming language.[Ras]

3.1 Versions

The first version of the Raspberry Pi was the Model 1B which was released in 2012. This had a single core CPU with 700 MHZ clock speed and 256MB RAM. The difference to the 2013 released version Model 1A is that the Model B has an Ethernet connector and two USB ports unlike the Model 1A which has no Ethernet and only one USB port. However, due to the lack of these components, the Model 1A was cheaper to purchase.
In 2014, an improvement of the Model 1B was released, the Model 1B+. The Model 1B+ was equipped with more RAM and the GPIO pins were increased from 26 to 40. The number of USB ports was doubled from two to four, and the composite video output was replaced with an HDMI output.
2014 also saw the introduction of a compute module in the form of a DDR2 SODIMM and an improved version of the 1A, the 1A+. The 1A+ was again downsized and equipped with HDMI.
Then in 2015, the Model 2B was introduced. This model shipped with quad-core CPU and had a 200MHz increase in clock speed(from 700MHz to 900MHz) and instead of 512MB RAM now 1024MB RAM.
The Raspberry Pi Zero was also introduced in 2015, this has the features of a Model 1B+ with single core CPU and 512MB Ram. The Zero's size was again smaller than the Model 1A+. The Zero was sold around 5 USD. However, it did not have an Ethernet port and instead of USB ports, it only had Micro-USB ports.
Then in 2016, the Raspberry Pi Model 3B was introduced. This was the first Raspberry Pi to have a WiFi and Bluetooth chip on board. The clock frequency of the CPU was increased by a new CPU from 900 MHz to 1200 MHz. With the new CPU it was also possible to run 64bit operating systems.
In 2017 a new Compute Module was introduced. The Compute Module 3 had the hardware of the Model 3B in the form of a DDR2 SODIMM module. The Raspberry Pi Zero W was introduced in the same year. This was equipped with a new chip to make WiFi and Bluetooth available.
Then in 2018, an improved version of the Model 3B was released. The 3B+. The Model 3B+ was equipped with increased CPU clock speed and the Ethernet port was swapped from 100Mbit to 1Gbit. A new Model 3A+ was also released. However, this had only 512MB RAM instead of the 1024MB of the Model 3B+.
In 2019, the Model 4B was released. This was equipped with a stronger CPU and the possibility to equip with up to 8GB RAM.

In 2020, the new Compute Module 4 and the Raspberry Pi 400 were released. The new Compute Module was no longer delivered in the form of a DDR2 SODIMM bar but in a new form. The Raspberry Pi 400 was shipped as a "keyboard". In this "keyboard" a Raspberry Pi was integrated. All connections were led out on the back of the keyboard. In 2021 the Pi Zero 2 was released. This now has a Quad-Core CPU instead of a Single-Core CPU.[pim21]

In this bachelor thesis the Raspberry Pi 4B with the following specifications is used:

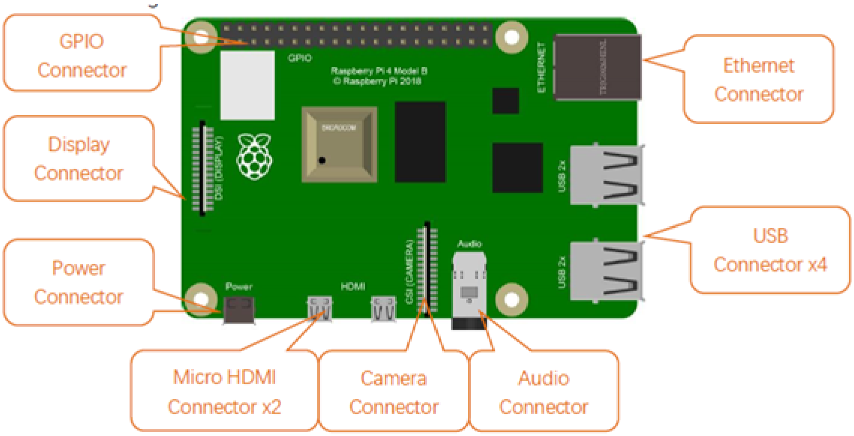


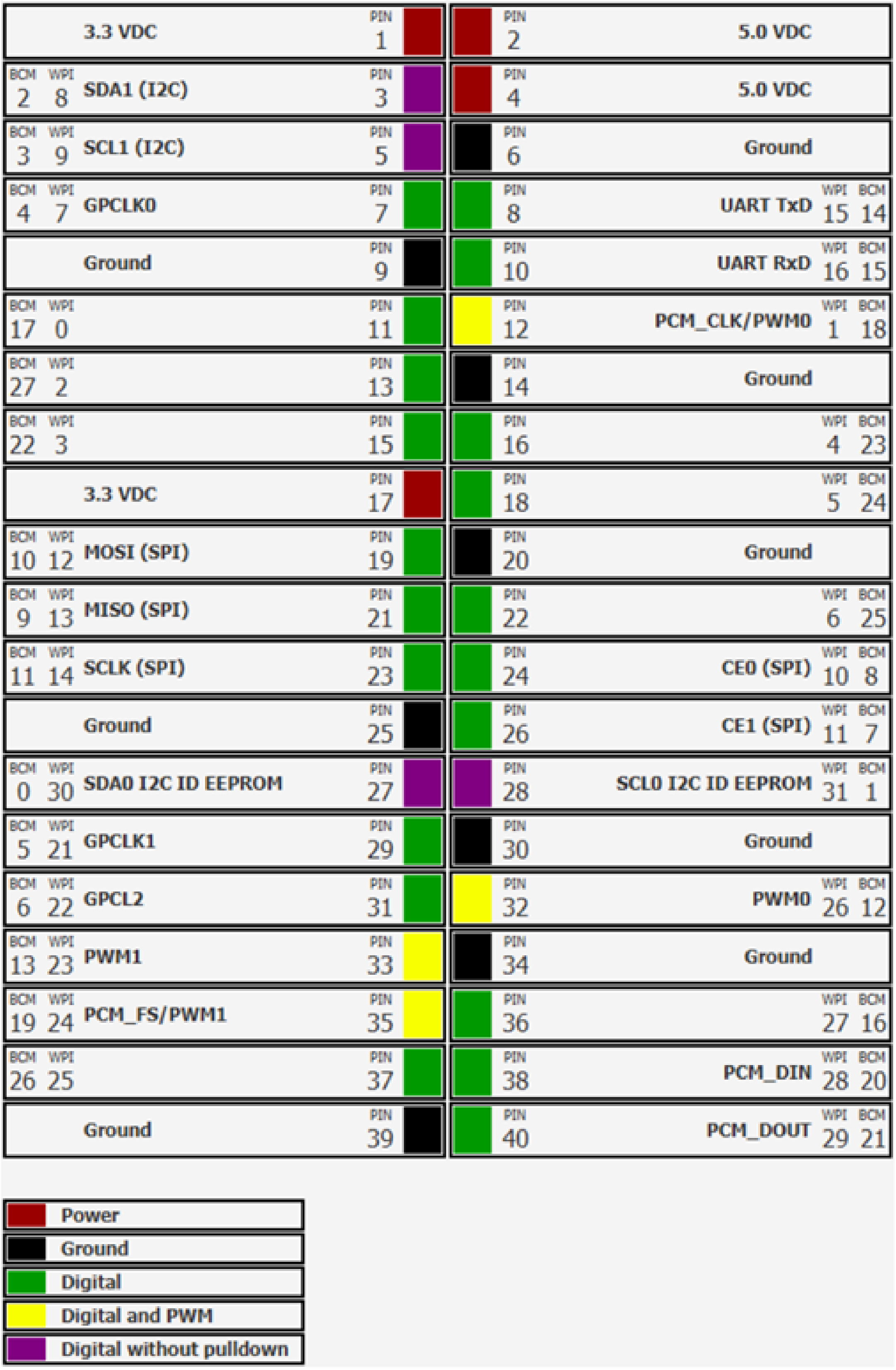
Figure 2: Raspberry Pi 4B [fre21]

- SoC: Boardcom BCM2711, Quad core Cortex A72 (ARM V8)
- RAM: 8GB LDDR4-3200 SDRAM

All examples were also tested and worked on a Pi 3B,3B+ and Zero 2W.

3.2 Pinout

The pinout in this work is based on the previously mentioned device (Raspberry 4B). The Raspberry 4B has a 40 pin GPIO header with the following pinout



BCM	WPI		PIN			PIN		WPI	BCM
		3.3 VDC	1	🟥	🟥	2	5.0 VDC		
2	8	SDA1 (I2C)	3	🟪	🟥	4	5.0 VDC		
3	9	SCL1 (I2C)	5	🟪	⬛	6	Ground		
4	7	GPCLK0	7	🟩	🟩	8	UART TxD	15	14
		Ground	9	⬛	🟩	10	UART RxD	16	15
17	0		11	🟩	🟨	12	PCM_CLK/PWM0	1	18
27	2		13	🟩	⬛	14	Ground		
22	3		15	🟩	🟩	16		4	23
		3.3 VDC	17	🟥	🟩	18		5	24
10	12	MOSI (SPI)	19	🟩	⬛	20	Ground		
9	13	MISO (SPI)	21	🟩	🟩	22		6	25
11	14	SCLK (SPI)	23	🟩	🟩	24	CE0 (SPI)	10	8
		Ground	25	⬛	🟩	26	CE1 (SPI)	11	7
0	30	SDA0 I2C ID EEPROM	27	🟪	🟪	28	SCL0 I2C ID EEPROM	31	1
5	21	GPCLK1	29	🟩	⬛	30	Ground		
6	22	GPCL2	31	🟩	🟨	32	PWM0	26	12
13	23	PWM1	33	🟨	⬛	34	Ground		
19	24	PCM_FS/PWM1	35	🟨	🟩	36		27	16
26	25		37	🟩	🟩	38	PCM_DIN	28	20
		Ground	39	⬛	🟩	40	PCM_DOUT	29	21

🟥	Power
⬛	Ground
🟩	Digital
🟨	Digital and PWM
🟪	Digital without pulldown

Figure 3: Pinout [Del20]

All examples illustrated in the next chapter use the WiringPi Scheme (WPI), if it is not explicitly mentioned otherwise. The author has chosen this scheme, because the used Pi4J version 1.4 is based on this scheme. The BCM scheme is only used from Pi4J version 2, since here, as already mentioned in chapter 2, WiringPi was changed to pigpio as low level integration.

3.3 GPIO

The GPIO interface of the Raspberry Pi is a very versatile interface that offers several purposes. The Raspberry Pi 4B has headers with 40 GPIO pins, but not all of them can be used for data exchange. Only 26 of these pins can be used for data exchange. The remaining pins are divided as follows. Four pins are used for power supply of the connected devices, two each with 3.3V and 5V. Eight pins are used as ground pins. The remaining two pins are used to expand the Raspberry Pi with a so-called HAT(Hardware At Top) module.[Mon16] These HAT modules must have certain characteristics to be considered as such modules. The plug-on board must have a size of 65x56 mm and it must also have the possibility to feed through the ribbon cables for a display and a camera. The modules are automatically recognized via the I^{2}C line.[Dem19] In the following the possible applications of the GPIO interface are described.

3.3.1 Digital Input/Digital Output

All 26 GPIO pins can be defined as input or output. If the pins are used as outputs, care must be taken that a maximum output current of 16mA per pin may flow.[Dem19] It should also be noted that a maximum output current of 50mA is available for all GPIO pins. The GPIO pins can also be used as input. Here you should take care that the input voltage does not exceed 3.3V, because this can destroy the Raspberry Pi.[Eleb] The pins WPI 8 and WPI 9 are equipped with an external pullup resistor to fulfill the specification for I^{2}C bus (chapter 3.3.6). All other pins can be connected with an internal pullup or pulldown resistor by software. These internal pullup or pulldown resistors have the purpose to set the input of a pin to a predefined state. The pullup resistor sets the input to "high" the pulldown resistor to "low". The voltages required to set the state can be found in Table 1. Defining a defined state is important because the GPIO pins tend to change their state without external influence[Elea] The GPIO pins can be used, for example, to switch an LED or to receive a signal from a pushbutton. Some applications of the GPIO pins are illustrated in chapter 5.

Parameter	Volts	Description
V_{IL}	≤ 0.8	Voltage, input low
V_{IH}	≥ 1.3	Voltage, input high

Table 1: Logic Levels [Gay18]

3.3.2 PWM

Pulse width modulation (PWM) can be used to simulate voltages between 0V and 3.3V. Since the Raspberry Pi can only output 0V or 3.3V this technique is used. The two important parameters are the pulse duration t1 and the period duration T. Thereby a so called duty-cycle can be generated. This can be in the range of 1-100 percent.[Tut] The Raspberry Pi has two PWM pins. These are WPI 26(PWM0) and WPI23(PWM1). These provide hardware PWM with a minimum frequency of 10Hz and a maximum frequency of 8KHz.[Mat21] However, it is also possible to simulate a PWM pin by a fast level change from 0V to 3.3V. The software-driven PWM is possible with WiringPi on all pins. However, only a maximum frequency of 100 Hz can be achieved.[Hen]

3.3.3 1-Wire

1-wire or one Wire is the name of a serial bus system developed by Dallas Semiconductor. This bus system was developed for devices with low data rates, such as the DS18B20 temperature sensor which is also presented in the following examples in the next chapter.[Gay18] For the 1-wire-bus only a data line, which also acts as a supply line, and a ground line are necessary. This type of supply is also called parasitic supply and uses a capacitor, which charges itself via the data line and supplies the sensor with power.[Dem19]
However, to realize a longer data line and to reduce data loss it is possible to supply the sensors directly with power.[sci]
The 1-wire bus is a master-slave bus, which means that the slaves only send data to the master when the master makes a request. The slaves cannot communicate with each other. The half-duplex method is used. The master is formed by the Raspberry Pi.[Dem19]
Since several devices can be used simultaneously on the 1-wire bus, the devices (slaves) must have a unique identifier so that the master can identify them unambiguously. This identifier consists of a 64bit long data set which is composed as follows: The first eight bits represent the "family code". The "family code" is a predefined code of the manufacturer, which identifies the type of the sensor (e.g. DS18B20 Temperature Sensor). The next 48 bits represent a unique serial number of the sensor. And the last 8 bits are the CRC code to verify that the type and serial number match.[Hor13]

Family Code	ID	CRC
8 bits	48 bits(unique to family)	8 bits
10	0008027e34	ca

Table 2: Example of a 1-wire address [Hor13]

The table given here(see Table 2) illustrates an example how this code can look like. A table listing the family codes is available at `https://owfs.org/`.
The pin intended for 1-wire devices is the WPI 7 pin.

3.3.4 UART

The Universal Asynchronous Receiver Transmitter (UART) is an interface for serial data exchange. The UART interface is directly integrated in the CPU of the Raspberry Pi. Asynchronous means that the sender does not have to send a clock signal to synchronize the transmission. This is achieved with UART with start bits and stop bits.[Mol16] UART exchanges the data via two lines. These are called RXD and TXD. RXD stands for Receive Data and TXD stands for Transmit Data. To exchange data between two UART interfaces the RXD and TXD lines must be crossed out. This means that the TXD line of the transmitter must be connected to the RXD line of the receiver and the RXD line of the transmitter must be connected to the TXD line of the receiver.[mik] The UART interface can only exchange data between two Raspberry Pi without further hardware, because they only work with an operating voltage of 3.3V. Connecting the UART interface of the Raspberry Pi with e.g. a UART interface of an Arduino would cause damage to the Raspberry Pi, because the Arduino has an operating voltage of 5V.[Ard] However, to enable communication with standardized interfaces such as the RS-232 interface, a MAX3232 chip must be used. This converts the signal into RS-232 compliant signal.[Hor13]
Figure 4 shows the exchange of data between an RS-232 interface and a Raspberry Pi using a MAX2323 chip. The output voltage of the RS-232 ranges from -15 to +15 volts, which would destroy the CPU of the Raspberry PI, but the MAX2323 chip reduces the voltage to the voltage needed by the Raspberry PI, which is about 3V. The transmission starts with sending the start bit. On this start bit the eight data bits are sent immediately. The following parity bit (used to detect transmission errors) and the stop bits (so that the receiver knows that the transmission has ended and it waits again for a new start bit) are optional.

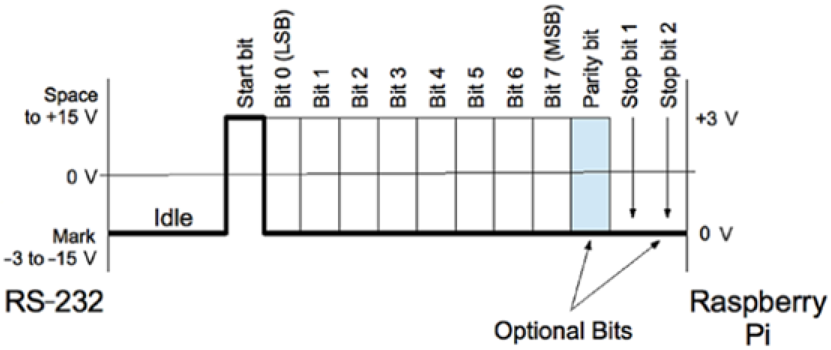


Figure 4: data exchange RS-232/UART [Gay18]

Another term that appears when using the UART interface is the baud rate. The baud rate, also called symbol rate, indicates how often the state (e.g. voltage level) of the communication channel changes in a given time unit (usually seconds).[ABa90] With the baud rate, care must be taken that the transmitter and receiver are set to the same baud rates.
The flow control regulates the data transmission. If the receiver takes longer to process the

data than the transmitter sends it, the receiver informs the transmitter to stop transmission in the meantime. Only when the receiver is ready to receive new data again does it inform the transmitter to resume transmission[Lab]
The pins intended for the UART interface are WPI 15(TXD) and WPI16(RXD)

3.3.5 SPI

The Serial Peripheral Interface (SPI) is a serial synchronous data bus which was developed by Motorola in 1987. The purpose is the communication between the microcontroller and the devices connected to the bus. However, the SPI bus is only designed for short data lines but higher transfer rates of up to 20Mbyte/s can be achieved. The data transfers take place in full duplex mode. This means that data can be sent and received simultaneously. SPI is based on the master-slave concept. Only the master has the possibility to start the communication between the master and the slaves. The slave devices cannot communicate with each other. The Raspberry Pi acts as master. Different devices can act as slaves, e.g. the MAX7219 chip presented in chapter 5.5.[Dem19]
Four lines are required by default. Two lines for data transmission (MOSI [Master Out Slave In] , MISO[Master In Slave Out]) and two control lines(SS[Slave Select],SCLK[Serial Clock]). The data is transmitted via the MOSI and MISO lines. The SCK line is used to transmit the clock signal generated by the master. The SS line is used to select the desired slave device.[Mic20]
The names of the data lines can vary, because we try to get away from the master-slave terminology. MOSI is also called POCI(Peripheral Out Controller In) and MISO is also called PICO(Peripheral In Controller Out). Likewise SS is now called CS(Chip Select). [Ass]
On the Raspberry Pi you have to take care that only two SS(CS) lines (CE0: WPI10 and CE1: WPI11) are available at the GPIO header. This means that without further manual configuration, only two slave devices can be used simultaneously in normal operation. In figure 5 you can see the necessary cabling[Pla]

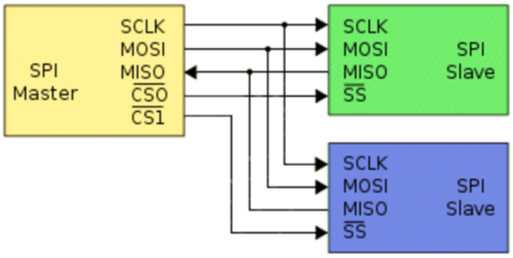


Figure 5: SPI cabling [Pla]

However, there is also a possibility to address several chips with one line. This is called "daisy chaining". The difference to normal operation is that the slave devices are not connected to a common data line but in series as shown in Figure 6. This means that the data sent by the master is passed from slave 1 to slave 2 and slave 2 then sends the response data back to the master.

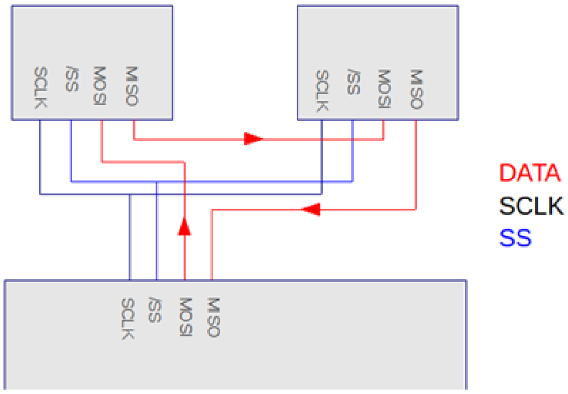


Figure 6: SPI Daisy Chaining[Mikb]

With this implementation it must be noted that as many commands are sent to the slaves as there are slaves. Otherwise the slaves could interpret the commands incorrectly.[Mikb]
The pins intended for the SPI interface are WPI12(MOSI), WPI13(MISO), WPI14(SCLK),WPI10(CE0) and WPI11(CE1)

3.3.6 I²C

The Inter-Integrated Circuit(I²C) or two-wire interface(TWI) was developed by Philips in 1982 and is, like SPI, an available BUS on the Raspberry Pi. On the basis of the I²C bus other bus systems were developed, e.g. the SMBus. The I²C bus is still widely used because it is cheap and easy to use.[Gay18]
The I²C bus has several operating modes which differ in the data rate. The different operating modes can be seen in table 3.[Dem19] Like the SPI bus, the I²C bus is a master-slave bus. The lines necessary for operation are called SDA (Serial Bus Data) and SCL (Serial Bus Clock). The SDA line is responsible for data transmission and the SCL for the clock. Figure 7 shows an example of how the slave devices are connected to the bus. The Raspberry Pi acts as the master and there are three slave devices connected to the bus.

Mode	Datarate
Standard Mode	100 kBit/s
Fast Mode	400 kBit/s
Fast Mode Plus	1 MBit/s
High Speed Mode	3.4 MBit/s
Ultra Fast Mode	5 MBit/s

Table 3: I²C operating modes [Dem19]

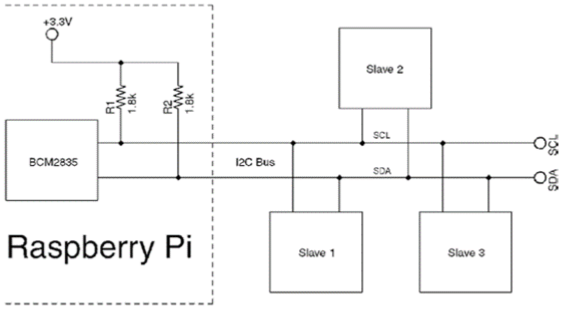


Figure 7: I^{2}C example [Gay18]

With the I^{2}C bus it is possible that the devices can also take on other roles. The Raspberry Pi could also be operated as a slave, in which case the role of the master could be taken over by a microcontroller, for example. However, there must always be one master, but there can also be several masters. This is then called MultiMaster.[Gay18]
Without further integration of range extenders[Moh17], the I^{2}C bus is limited to a spatial extension of a few meters, otherwise transmission errors may occur.[Dem19]
So that the I^{2}C bus master can distinguish between the slave devices, each slave device has its own address. These addresses identify the devices unambiguously. A maximum of 127(at 7 bit) addresses or connected devices are possible on one bus. For some devices it is possible to change the address by separate pins. This is shown in chapter 5.6. As with the SPI bus, with the I^{2}C bus it is only possible for the master to start communication.[Mika]
The pins intended for the I^{2}C Bus are WPI8(SDA) and WPI9(SCL).

4 Installation and Configuration

This chapter explains the installation of all necessary software components and the configuration of the Raspberry Pi. For this bachelor thesis the following software versions were used.

- Raspberry Pi OS Bullseye 32 Bit Kernel 5.10, Release Date: 28.01.2022
- ooRexx: Open Object Rexx Version 5.0.0 r12352 32 Bit
- Java: Bellsoft openjdk version ”11.0.14” 2022-01-18 LTS
- BSF4ooRexx: V641-20220131 32 Bit
- WiringPi: Version 2.70
- Pi4J Version 1.4

4.1 ooRexx

The first step is to install ooRexx on the Raspberry Pi. The procedure is based on the installation guide by Tony Dycks, which was presented at the 32nd Annual Rexx Symposium in 2021.[Dyc21] Before installing ooRexx, all required software packages must be installed.(see Listing 2)

```
sudo apt install subversion -y
sudo apt install cmake -y
sudo apt install g++ -y
sudo apt install libncurses5-dev -y
```

Listing 2: additionally required software packages

The next step is now the installation of ooRexx. First a folder for ooRexx is created. Then ooRexx is downloaded from suversion and compiled.(see Listing 3)

```
cd /home/pi
sudo mkdir oorexx
cd oorexx
sudo mkdir build
cd build
svn checkout svn://svn.code.sf.net/p/oorexx/code-0/main/trunk oorexx-code-0
cd oorexx-code-0
sudo cmake .
sudo make install
```

Listing 3: install ooRexx

ooRexx uses the appropriate version by compiling on the device. Here it will be the 32 bit version. This process will take some time.

4.2 Java

The next step is to install the Java Development Kit.(see Listing 4). The software is downloaded with the help of the command line program wget and is installed immediately afterwards. The latest version of Java11 JDK 32bit for ARM is used at the time of writing this bachelor thesis. The JDK used is created by BellSoft. The respective commands for download and installation have been split into two lines for readability. In line 1 and 2 Java is downloaded and in line 3 and 4 it is installed.

```
1 sudo wget "https://download.bell-sw.com/java/11.0.14+9/\
2 bellsoft-jdk11.0.14+9-linux-arm32-vfp-hflt-full.deb"
3 sudo apt install /home/pi/oorexx/build/oorexx-code-0/\
4 bellsoft-jdk11.0.14+9-linux-arm32-vfp-hflt-full.deb
```

Listing 4: install Java JDK

4.3 BSF4ooRexx

In this step BSF4ooRexx is downloaded and installed(see Listing 5)

```
sudo wget "https://sourceforge.net/projects/bsf4oorexx/files/latest/download" -O bsf.zip
sudo unzip bsf.zip
sudo rm bsf.zip
cd bsf4oorexx
cd install
cd linux
sudo yes | sh install.sh
```

Listing 5: install BSF4ooRexx

Just like ooRexx, BSF4ooRexx automatically uses the appropriate version. The 32Bit version is installed

4.4 WiringPi

Next step is the installation of the WiringPi software package. This is downloaded from Github and installed afterwards.(see Listing 6)

```
cd /home
sudo git clone "https://github.com/WiringPi/WiringPi.git"
cd WiringPi
sudo ./build
```

Listing 6: install WiringPi

4.5 Jar Archives

If all previous steps have been completed successfully, the next step can be started. This step explains how to download all the required JAR archives.

4.5.1 Pi4J

For the two still needed JAR archives a folder jar is created in the directory /home/pi/. The version 1.4 of Pi4J is then loaded and unpacked into this folder (see Listing 7).

```
cd /home/pi
mkdir jars
cd jars
sudo wget "https://pi4j.com/download/pi4j-1.4.zip"
sudo unzip pi4j-1.4.zip
```

Listing 7: download ad unzip Pi4J

4.5.2 pi4oorexx

The pi4oorexx jar archive must be loaded by hand from `https://github.com/pi4oorexx/pi4oorexx/blob/6dd1fd72928f0a516725bce7be26c16103a3d105/pi4oorexx.jar` and moved to the /home/pi/jar directory.

4.6 Set Java Classpath

The next step is to set the Java Classpath. The following three JAR archives must be placed in the Classpath

- pi4j-core.jar
- bsf4ooRexx-v641-20220131-bin.jar
- pi4oorexx.jar

To enter the classpath into the bashrc file, the command from Listing 8 must be executed.

```
sudo nano ~/.bashrc
```

Listing 8: open bashrc file

Then you should go to the end of the file and extend the Classpath with the following command.(see Listing 9) With this command the JAR archives of Pi4J, pi4oorexx and BSF4ooRexx are entered into the classpath The "$CLASSPATH" variable causes any values that were already in the classpath to be taken over. The second and third "export" command was added only because of readability.

```
export CLASSPATH=$CLASSPATH:"/home/pi/jars/pi4j-1.4/lib/pi4j-core.jar"
export CLASSPATH=$CLASSPATH:"/home/pi/jars/pi4oorexx.jar"
export CLASSPATH=$CLASSPATH:"/opt/BSF4ooRexx/bsf4ooRexx-v641-20220131-bin.jar":.
```

Listing 9: set Classpath

The paths to the individual JAR archives should be placed under quotation marks. The colon serves as a separator between the individual paths. The dot at the end of the Classpath is also intentional, because it includes the current directory in the Classpath.[del]

Afterwards with CTRL+O the entry is stored and confirmed with RETURN. With CTRL+X the editor is left again. The command line interface must be restarted after setting the classpath.

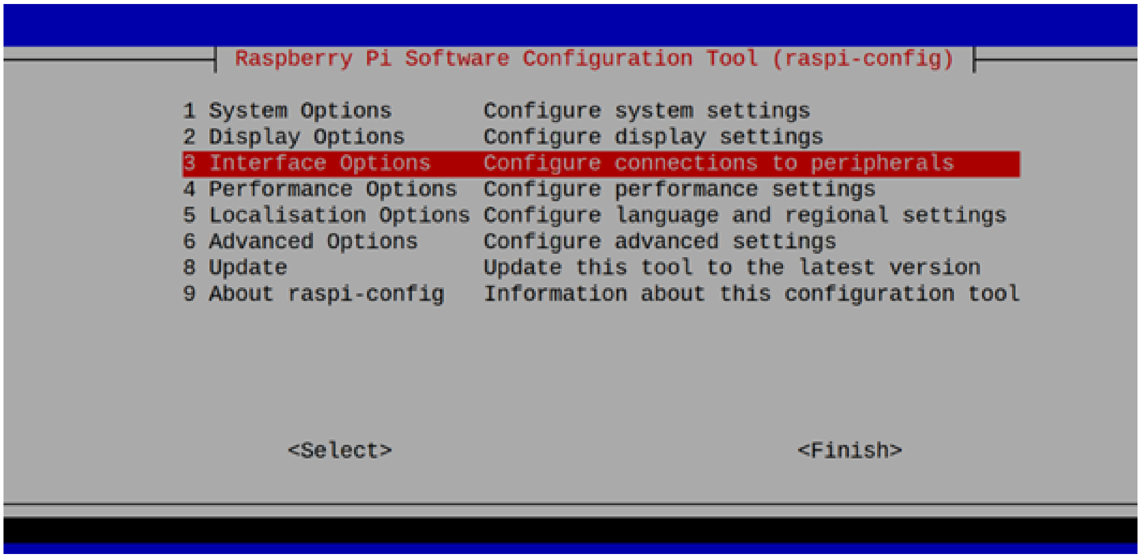
4.7 Enable Interfaces

The interfaces presented in chapter 3 are deactivated by default and must be activated before they can be used for the first time. The activation of the interfaces can be done with the following command (see Listing 10).

```
sudo raspi-config
```

Listing 10: raspi-config

After that, point "3 interface options" will be selected and confirmed by return key.(see Figure 8)



```
Raspberry Pi Software Configuration Tool (raspi-config)

1 System Options       Configure system settings
2 Display Options      Configure display settings
3 Interface Options    Configure connections to peripherals
4 Performance Options  Configure performance settings
5 Localisation Options Configure language and regional settings
6 Advanced Options     Configure advanced settings
8 Update               Update this tool to the latest version
9 About raspi-config   Information about this configuration tool

           <Select>                          <Finish>
```

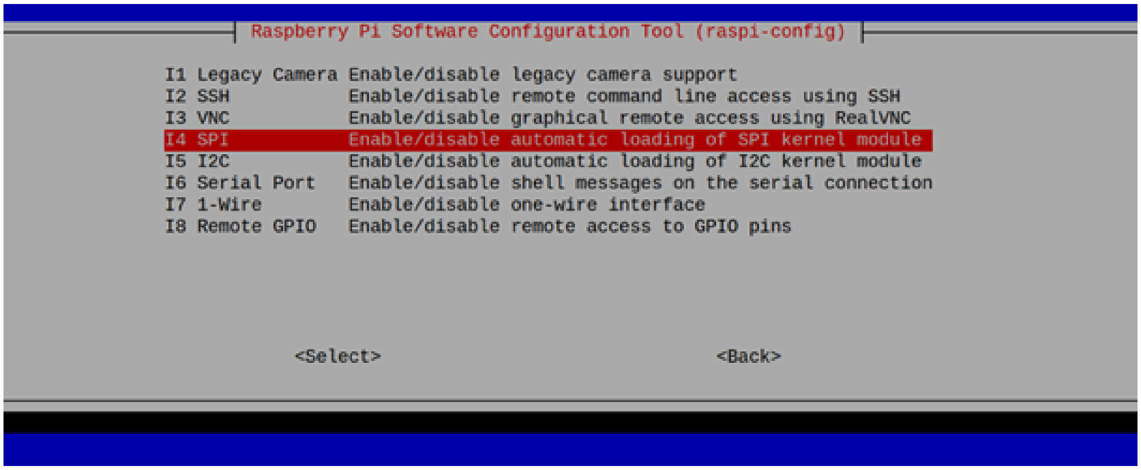
Figure 8: raspi-config

After that the desired interfaces can be activated. To be able to test all examples of this bachelor thesis the following interfaces have to be activated(see Figure 9)

- I4 SPI
- I5 I2C
- I6 Serial Port (UART)

- I7 1-Wire

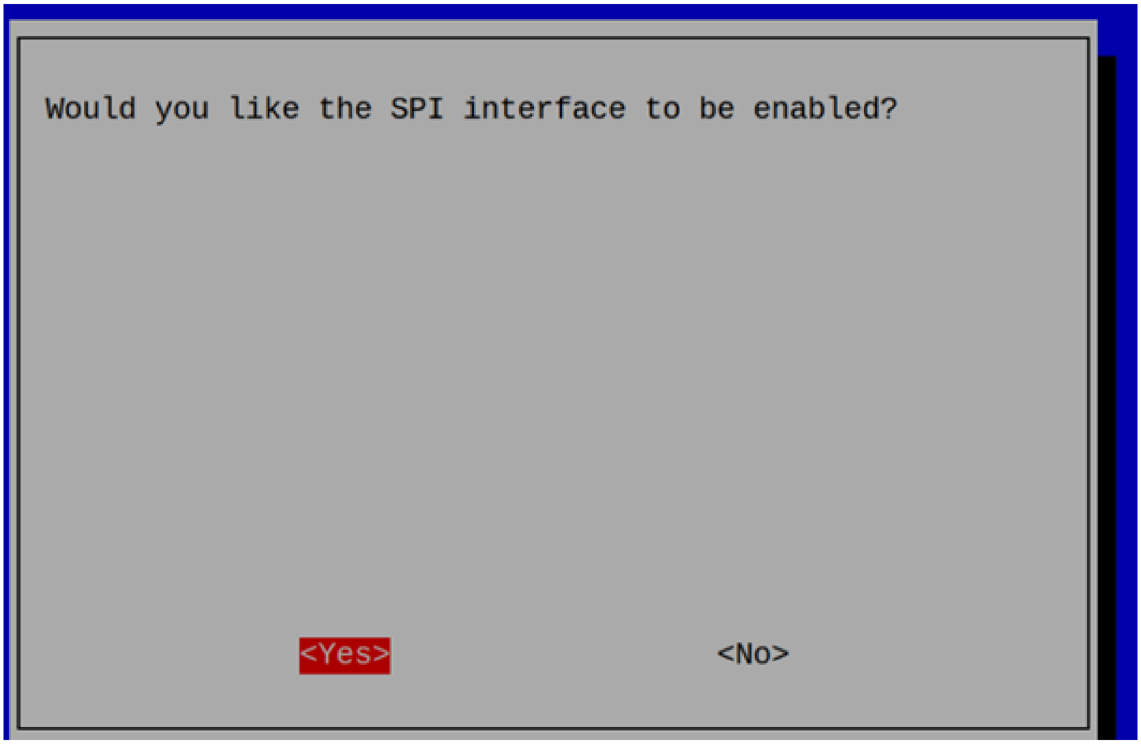
- I8 Remote-GPIO (the activation of this interface is only necessary for the demonstration of pigpio in example 5.1.3)



```
              Raspberry Pi Software Configuration Tool (raspi-config)

        I1 Legacy Camera Enable/disable legacy camera support
        I2 SSH           Enable/disable remote command line access using SSH
        I3 VNC           Enable/disable graphical remote access using RealVNC
        I4 SPI           Enable/disable automatic loading of SPI kernel module
        I5 I2C           Enable/disable automatic loading of I2C kernel module
        I6 Serial Port   Enable/disable shell messages on the serial connection
        I7 1-Wire        Enable/disable one-wire interface
        I8 Remote GPIO   Enable/disable remote access to GPIO pins




              <Select>                               <Back>
```

Figure 9: raspi-config interfaces

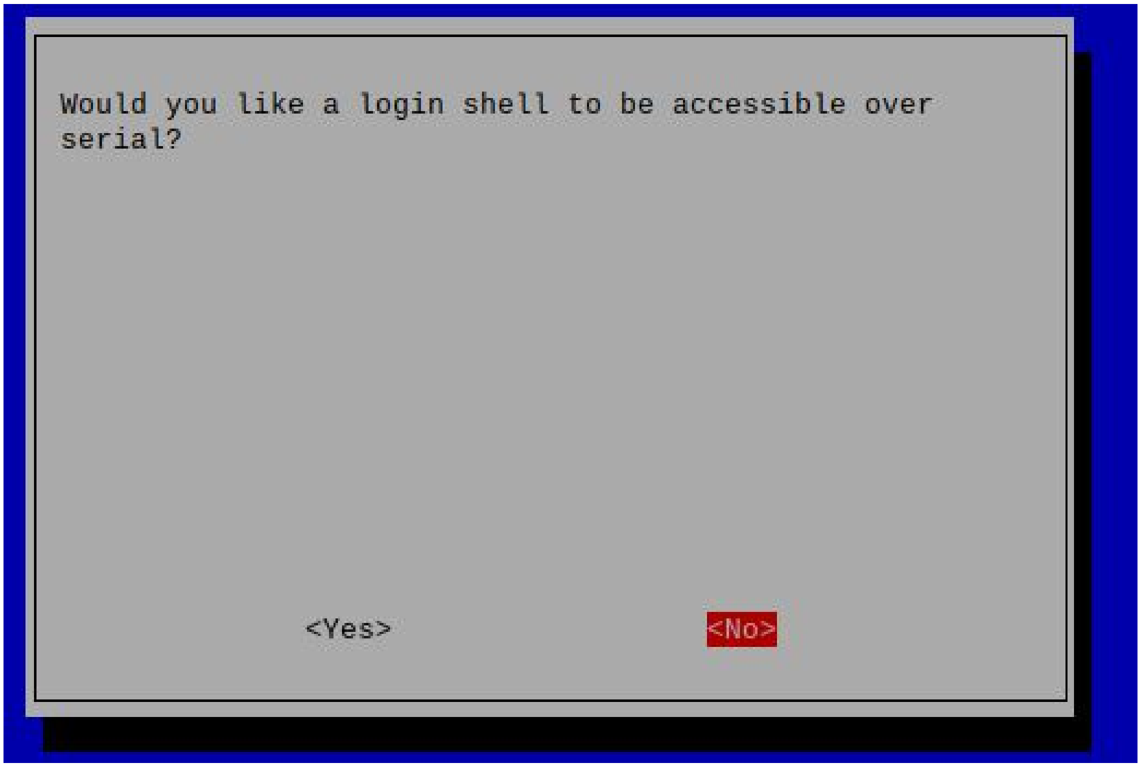
As a demonstrative example the activation of the SPI interface is shown. This procedure can also be used for the I2C and the 1-Wire interface. Only for the serial port further steps have to be done.
To activate the SPI interface ”I4 SPI ” must be selected and confirmed with the return key. Afterwards ”Would you like the SPI interface to be enabled” must be confirmed with ”yes ”.(see Figure 10)



Would you like the SPI interface to be enabled?

Figure 10: raspi-config activate SPI

For the Serial port one more step is necessary, because the Raspberry Pi redirects the serial port to the console (see Figure 11).



Would you like a login shell to be accessible over serial?

Figure 11: raspi-config activate Serial Port

After activating the interfaces, the Raspberry Pi should be restarted.

5 Examples

This chapter demonstrates how various electrical components can be controlled with ooRexx or with the help of BSF4ooRexx and the pi4j or pi4oorexx library. For simplification all components were used on so-called breakout boards[Hus21]. These breakout boards already have all the additional components required, such as series resistors for LEDs.
For all examples a so called T-Board with cable extension is used which is plugged into a breadboard. This has the advantage that all sensors do not have to be connected directly to the Raspberry Pi. Due to the breadboard there are also several connection possibilities per pin available[Del20]

5.1 Digital Input/Digital Output

The digital inputs and outputs can be used in many different ways. from simple switching of an output or reading in a pushbutton to controlling an LC-Display (example 5.7.1).

5.1.1 gpioOut1.rex

This example demonstrates how to define a GPIO pin as an output pin and switch it on and off. To see the status of the output pin it is connected to a LED. Pi4J is used to define the pin as output.

5.1.1.1 Circuit

The anode(A) of the LED is connected to the GPIO pin 29 which is defined as output. The cathode of the LED(K) is connected to Ground(GND).(see Figure 12)

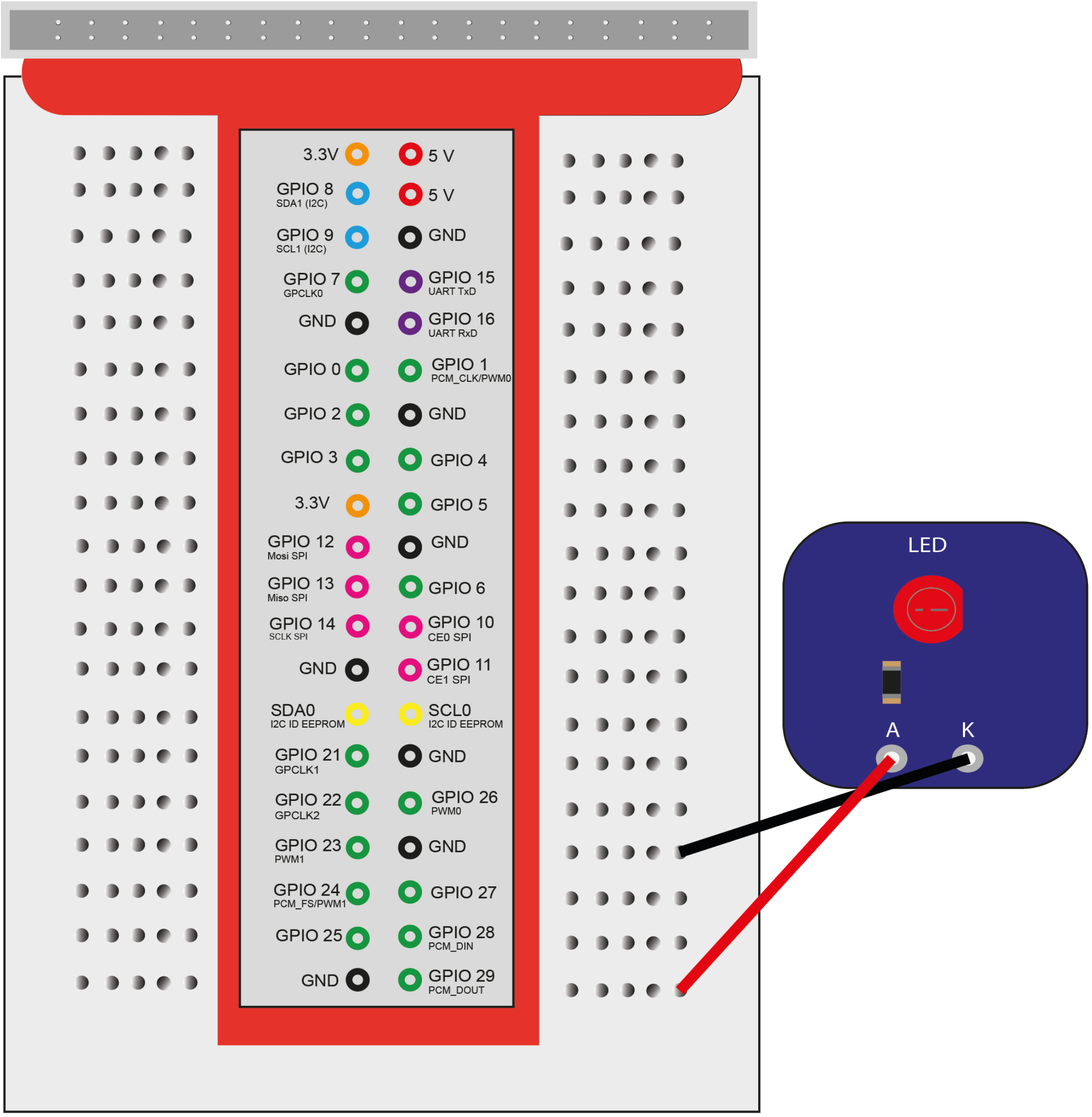


Figure 12: gpioOut circuit

5.1.1.2 Sourcecode

At the beginning the required Java classes have to be loaded once via BSF4ooRexx.The needed Java classes are in the Pi4J JAR archive. In line 14 an instance of this class is created immediately (see Listing 11).

```
14 gpio=bsf.loadClass("com.pi4j.io.gpio.GpioFactory")~getInstance
15 RaspiPin=bsf.loadClass("com.pi4j.io.gpio.RaspiPin")
16 pinstate=bsf.loadClass("com.pi4j.io.gpio.PinState")
```

Listing 11: gpioOut1 Load requires classes

After loading the Java classes, a digital output is defined and the default state low is assigned to it. The GPIO pin 29 is selected as output.(see Listing 12)

```
LED=gpio~provisionDigitalOutputPin(RaspiPin~GPIO_29,pinstate~low)
```

Listing 12: gpioOut1 define output

The actual program (see Listing 13) runs in an endless loop and switches the previously defined GPIO, here called LED, high and after one second low again. Thereby the LED flashes in a second rhythm. The code in line 33 is necessary so that the program can access the Java classes via BSF4ooRexx.

```
24 say "program started"
25 do forever
26     LED~high
27   call syssleep 1
28   LED~low
29   call syssleep 1
30 end
31 exit
32
33 ::requires BSF.CLS  -- get Java support
```

Listing 13: gpioOut1 main program

5.1.2 gpioOut2.rex

The example gpioOut2 offers exactly the same functions as gpioOut1 but it does not use Pi4J and instead uses WiringPi to control the pins. This way Java can be omitted completely.

5.1.2.1 Circuit

This program uses the same circuit as gpioOut1.(see Figure 12)

5.1.2.2 Sourcecode

First the GPIO pin must be defined as output again (Listing 14). This is done in this example with the ooRexx command ”address system”.

```
address system "gpio mode 29 out"
```

Listing 14: gpioOut2 define pin as output

After that the output pin GPIO 29 gets the default state low. WiringPi uses 1 and 0 instead of the terms low and high. 1 symbolizes high and 0 symbolizes low (see Listing 15)

```
address system "gpio write 29 0"
```

Listing 15: gpioOut2 define default state

The main program switches an LED on and off for one second, as in gpioOut1. The switching on of the output is done with the code from line 24 and the switching off is done in line 26 (see Listing 16).

```
say "program started"
do forever
  address system "gpio write 29 1"
  call syssleep 1
  address system "gpio write 29 0"
  call syssleep 1
end
exit
```

Listing 16: gpioOut2 main program

5.1.3 gpioOut3.rex

gpioOut3 offers exactly the same functions as gpioOut1 and gpioOut2. This example was chosen for a one-time demonstration of the pigpio library. The pipgio library is only used in version 2 of Pi4J. Up to version 1.4 WiringPi is used.

5.1.3.1 Circuit

This program uses the same circuit as gpioOut1 and gpioOut2.(see Figure 12) However, it should be noted that pigpio uses the BCM scheme and not the WiringPi scheme. This means that the wiring remains the same, but the GPIO pin is no longer 29 but 21. (see chapter 3.2 Pinout)

5.1.3.2 Sourcecode

In this example the support of Java is omitted and the commands are executed by ooRexx ”address system” command. Before you can access the GPIO pins with pigpio you have to start the pigpiod daemon (see Listing 17).

```
address system "sudo systemctl start pigpiod"
```

Listing 17: gpioOut3 start daemon

The next step is to define pin 21 (BCM scheme) as output pin. This is defined by the letter ”w” at the end of the command. the letter ”r” would define the pin as an input pin.(see listing 18)

```
address system "pigs modes 21 w"
```

Listing 18: gpioOut3 define output

After that the default state is set to low or 0 again. pigpio uses like WiringPi 0 and 1 for the states high and low(see Listing 19)

```
address system "pigs w 21 0"
```

Listing 19: gpioOut3 define default state

The main program is structured as in the previous two programs. In an endless loop the output is switched on and off every second. (see Listing 20)

```
say "program started"
do forever
    address system "pigs w 21 1"
  call syssleep 1
  address system "pigs w 21 0"
  call syssleep 1
end
exit
```

Listing 20: gpioOut3 main program

In this bachelor thesis no further example will appear using the pigpio library. This was only demonstrated once so that you can see the strong similarities to the WiringPi syntax.

5.1.4 pinListener1.rex

In the three previous examples the use of the GPIO pins as output was shown. However, it is also important to be able to process user input. Therefore, the next three examples demonstrate the query of a GPIO pin as input. This input is generated by a push button. In the example pinListener1 the GPIO pin 24 is used as input pin. The pin registers the input based on the applied voltage. The threshold values that must be applied to define the logic level are explained in chapter 3.3.1,Table 1.The pins are controlled by the pi4j library

5.1.4.1 Circuit

The output, which is equipped with an LED, is on GPIO pin 29 as in the gpioOut examples. The input is GPIO pin 24.

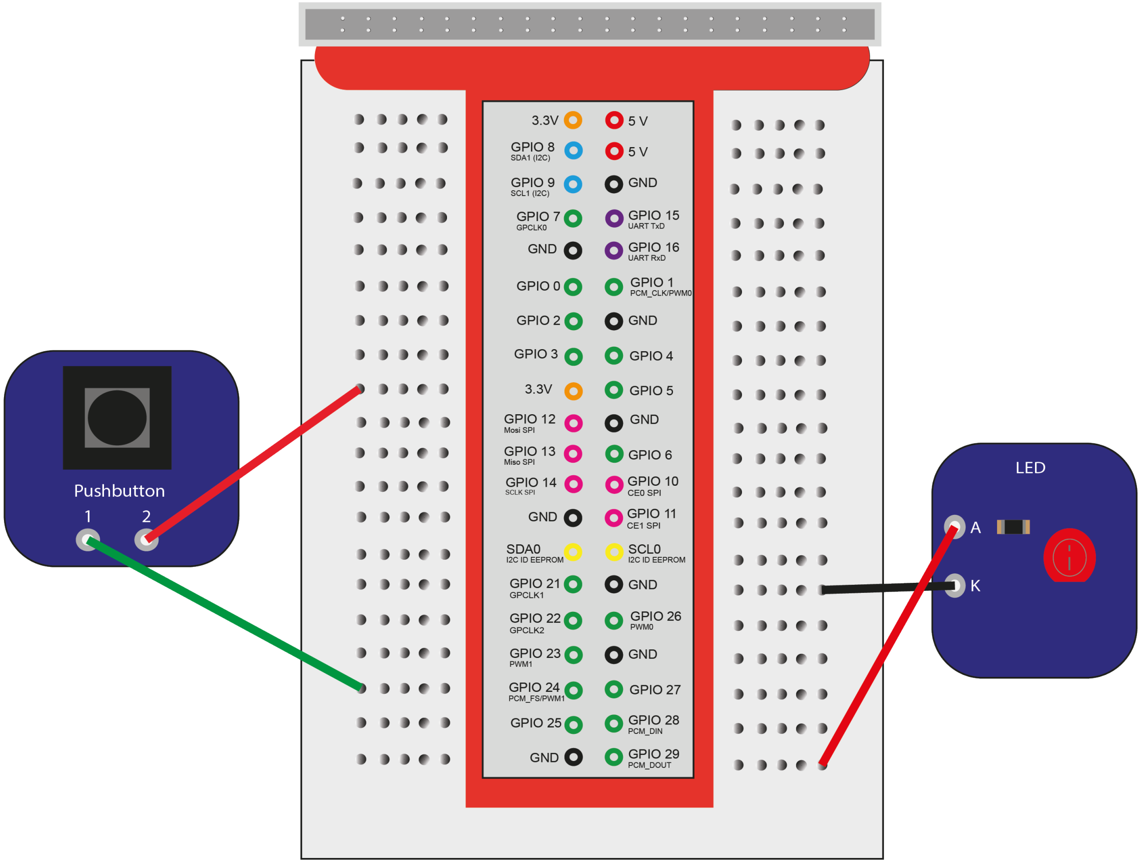


Figure 13: pinListener circuit

5.1.4.2 Sourcecode

At the beginning again all needed Java classes have to be loaded by BSF4ooRexx. In line 17 the class "PinPullResistance" is loaded, which can be used to activate the internal PullUp or PullDown resistors. This is especially necessary for the pushbutton, because its state is not clearly defined and could otherwise change without external influences.(see Listing 21)

```
15 gpio=bsf.loadClass("com.pi4j.io.gpio.GpioFactory")~getInstance
16 clzRaspiPin=bsf.loadClass("com.pi4j.io.gpio.RaspiPin")
17 clzPinPullResistance=bsf.loadClass("com.pi4j.io.gpio.PinPullResistance")
18 pinstate=bsf.loadClass("com.pi4j.io.gpio.PinState")
```

Listing 21: pinListener1 required Java classes

In Listing 22 the definition of GPIO pin 24 as input is shown. Also this input is set to low or 0 with the internal PullDown resistor.

```
PushButton = gpio~provisionDigitalInputPin(clzRaspiPin~GPIO_24,-
clzPinPullResistance~PULL_DOWN)
```

Listing 22: pinListener1 define Pushbutton

In line 26 the output is defined and set to low at the same time. This means that the LED does not light up when the program is started, but only after the button is pressed. The variable LEDstate stores whether the variable is on or off.(see Listing 23)

```
26 LED =gpio~provisionDigitalOutputPin(clzRaspiPin~GPIO_29,pinstate~low)
27 LEDstate = 0
```

Listing 23: pinListener1 define output and LED state

The code in Listing 24 continuously queries the state of the input. It is distinguished whether the LED is on or off. If the LED is off and the state of the input pin is high then the LED is switched on (see line 32-36). In line 37, the program is stopped briefly after a status change of the LED, since pressing the button too long would immediately switch it on again in the next run. Therefore the user is given 0.4 seconds to release the button again. From line 41 onwards, when the LED is on and the button is pressed, the LED is switched off again. The delay in line 49 causes the button to be polled for its state 10 times per second. Line 51 delivers the Java support per BSF4ooRexx

```
29 say "Listening started - press the pushbutton to change the status of the LED"
30 do forever
31
32   if LEDstate = 0 then do
33     if PushButton~getState~toString = "HIGH" then do
34       say "LED on"
35       LEDstat = 1
36       LED~high
37       call syssleep 0.4
38     end
39   end
40
41   else do
42     if PushButton~getState~toString = "HIGH" then do
43       say "LED OFF"
44       LEDstat = 0
45       LED~low
46       call syssleep 0.4
47     end
48   end
49 call syssleep 0.1
50 end
51 ::Requires BSF.CLS
```

Listing 24: pinListener1 main program

5.1.5 pinListener2.rex

This example shows the Pinlistener1 example without using Java and the Pi4J library. For this the WiringPi library is used again.

5.1.5.1 Circuit

The wiring has not changed in this example compared to pinListener1 (see Figure 13)

5.1.5.2 Sourcecode

In line 17 the GPIO pin 24 is defined as input and in the following line the internal PullDown resistor is activated.(see Listing 25)

```
17 address system "gpio mode 24 in"
18 address system "gpio mode 24 down"
```

Listing 25: pinListener2 define input pin and PullDown

In line 23 and 24 the GPIO pin is defined as output and its state is set to 0 or low(see listing 26)

```
23 address system "gpio mode 29 out"
24 address system "gpio write 29 0"
```

Listing 26: pinListener2 define input pin and PullDown

The main program is the same as pinListener1 but all Pi4j commands have been replaced by WiringPi commands. Therefore also in line 30 an array is created, which stores the current state of the input pin. The query for this takes place in line 34. As with pinListener1, it is queried here whether the LED is on or off. If it is off and the button is pressed, it is activated with the command in line 40. If the LED is on and the button is pressed, the LED is switched off with the command in line 49.(see Listing 27)

```
29 say "Listening started - press the pushbutton to change the status of the LED"
30 currentState= .array~new
31 do forever
32
33 /*check status of input pin*/
34 address command "gpio read 24" with output using (currentState)
35
36   if LEDstat = 0 then do
37     if currentState[1] = 1 then do
38       say "LED on"
39       LEDstat = 1
40       address system "gpio write 29 1"
41       call syssleep 0.4
42     end
43   end
44
45   else do
46     if currentState[1] = 1 then do
47       say "LED OFF"
48       LEDstat = 0
49       address system "gpio write 29 0"
50       call syssleep 0.4
51     end
52   end
```

```
call syssleep 0.1
end
```

Listing 27: pinListener2 main program

5.1.6 pinListener3.rex

The example pinListener3 now uses the Pi4J library again via BSF4ooRexx. In this example the state of the input pin is not made via a continuous query but via a so-called ActionListener. This example is based on the Pi4J demo example ListenGpioExample.java by Robert Savage.[Sav21]

5.1.6.1 Circuit

The wiring has not changed in this example compared to pinListener1 (see Figure 13)

5.1.6.2 Sourcecode

At the beginning the needed Java classes have to be loaded.(see Listing 28)

```
gpio=bsf.loadClass("com.pi4j.io.gpio.GpioFactory")~getInstance
clzRaspiPin=bsf.loadClass("com.pi4j.io.gpio.RaspiPin")
clzPinPullResistance=bsf.loadClass("com.pi4j.io.gpio.PinPullResistance")
pinstate=bsf.loadClass("com.pi4j.io.gpio.PinState")
```

Listing 28: pinListener3 required Java classes

After that the same GPIO input 24 is used as in the other two examples (see Listing 29).

```
PushButton = gpio~provisionDigitalInputPin(clzRaspiPin~GPIO_24,-
clzPinPullResistance~PULL_DOWN)
```

Listing 29: pinListener3 define input

The output is again connected to GPIO pin 29 and defined as start state ”off”. This time, however, as can be seen in line 25, the pin must be placed in the locale package so that the pin can be used from the entire program. In line 27 the state of the LED is stored.(see Listing 30)

```
25 pkgLocal=.context~package~local  -- get package local directory
26 pkgLocal~LED = gpio~provisionDigitalOutputPin(clzRaspiPin~GPIO_29, clzPinstate~low)
27 LEDstate = 0
```

Listing 30: pinListener3 define output

In Listing 31 first a new .evl object is created and then the listener is created and in line 38 this is assigned to the button.

```
34 rexxObj = .evl~new
35 clzGpioPinListenerDigital = -
36 bsf.loadClass("com.pi4j.io.gpio.event.GpioPinListenerDigital")
37 javaObj = bsfCreateRexxProxy(rexxObj, , clzGpioPinListenerDigital)
```

```
listeners=bsf.createJavaArrayOf(clzGpioPinListenerDigital, javaObj)
PushButton~addListener(listeners)
```

Listing 31: pinListener3 create and register listener

The code in Listing 32 makes the program run until CTRL + C is pressed.

```
signal on syntax name syntax_but_ok
do forever
  call syssleep .5
end
exit

syntax_but_ok:
  say "syntax_but_ok"
  exit
```

Listing 32: pinListener3 run main program

The unknown method in the class evl (see Listing 33) is used to catch all messages from Java. To get the state of the pin, the value of the slotdir argument is queried, as seen in line 58. If this is high and the LED is off, then it is switched on or vice versa.

```
52 ::class evl
53 ::method unknown
54   expose LEDstate
55   use arg eventObject, slotDir
56
57   if LEDstate = 0 then do
58     if slotdir[1]~getState~toString = "HIGH" then do
59       say "LED on"
60       LEDstate = 1
61       .LED~high
62
63     end
64   end
65
66   else do
67     if slotdir[1]~getState~toString = "HIGH" then do
68       say "LED OFF"
69       LEDstate = 0
70       .LED~low
71
72     end
73   end
```

Listing 33: pinListener3 class evl

Listing 34 shows how to get Java support via BSF4ooRexx, without this support this would not be possible.

```
::requires BSF.CLS
```

Listing 34: pinListener3 get Java Support

5.2 PWM.rex

The Raspberry Pi offers the possibility to control devices via its hardware PWM pin. This is demonstrated in the example servo.rex.

5.2.1 servo

In this example a SG90 servo is controlled by CLI commands. The servo can rotate about 180 degrees. A numeric input in the range between 50 and 250 is expected, an input value of 150 represents the home position of the servo (0 degrees). An input of 50 will move the servo to -90 degrees and an input of 250 will move the servo to +90 degrees. For this example the PWM mode of Wiring Pi is used.

5.2.1.1 Circuit

The servo must be supplied with 5V voltage(VCC). Therefore it is connected to one of the two 5V pins. It also needs a connection to Ground (GND) so that the circuit is closed. The signal pin is connected to pin 26, because this is one of the two PWM pins of the Raspberry Pi.(see Figure 14)

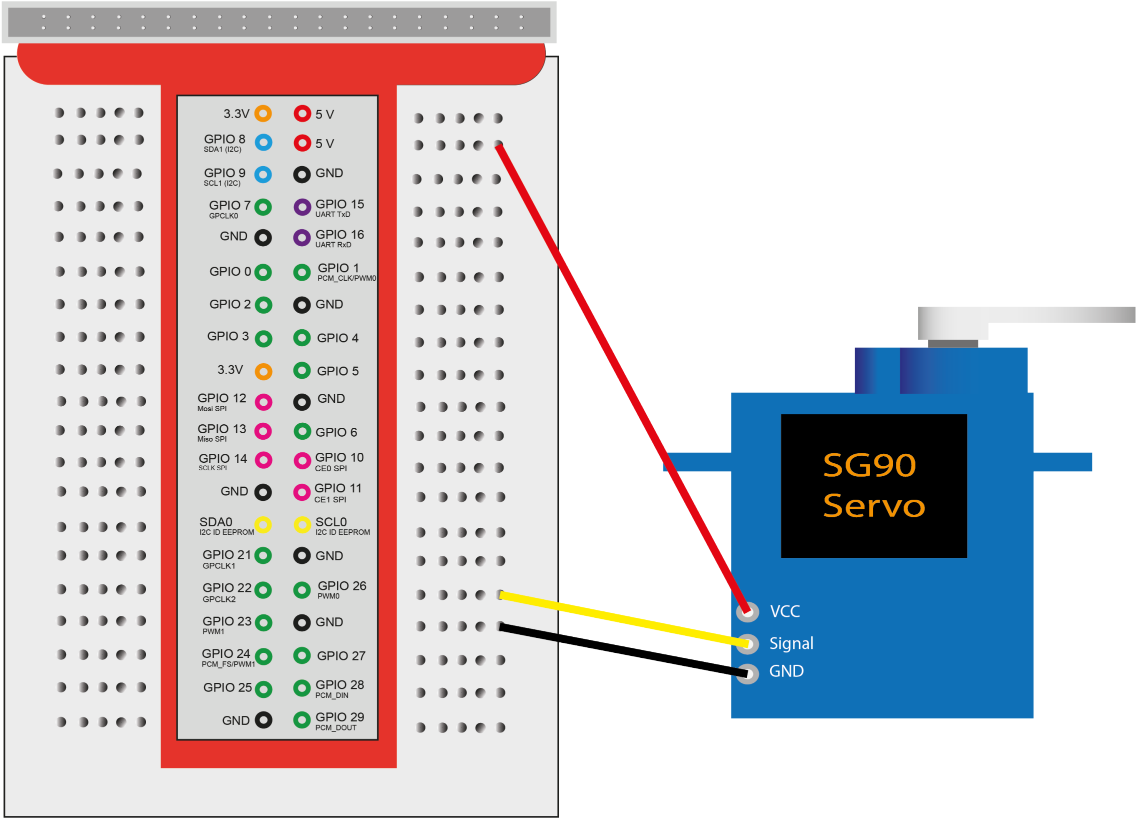


Figure 14: Servo circuit

5.2.1.2 Sourcecode

In line 13 the GPIO pin is set to PWM mode. Line 14 sets the mark space mode for the PWM pin. In line 15 the PWM clock divider is set and Line 16 sets the range of the PWM. The servo needs a frequency of 50 Hertz and this is achieved by dividing the PWM frequency of the Rapsberry Pi (19.2MHz) by the clock divider and by the range. (see Listing 35) this results in $\frac{19,200,000Hz}{192*2000} = 50Hz$

```
13 address system "gpio mode 26 pwm"
14 address system "gpio pwm-ms"
15 address system "gpio pwmc 192"
16 address system "gpio pwmr 2000"
```

Listing 35: servo define pin and state

Before the user can make any entries, the servo is moved to its home position of 0 degrees.(see Listing 36)

```
address system "gpio pwm 26 150"
```

Listing 36: servo 0 degree

In the main program the user has the possibility to set the servo to his desired position between -90 and +90 degrees via CLI input. The default position is at a value of 150. As mentioned above, with an input value of 50 the servo is set to -90 degrees and with 250 to +90 degrees. With each input the entered value is checked for validity. this means that the inputs may only be between 50 and 250, otherwise the servo comes into an undefined state.(see Listing 37)

```
do forever
say " Enter a value between 50 and 250 "
parse pull angle
  if angle >49 & angle < 251 then do
    command = "gpio pwm 26 " angle
    address system command
    end
  else say "wrong input: must be between 50 and 250"
end
exit
```

Listing 37: servo main program

5.3 1-Wire

The 1-Wire bus provides the Raspberry Pi with a very easy to program interface. This is shown with the DS18B20 temperature sensor. Only a few lines of code are needed to implement these sensors.

5.3.1 ds18b20stream.rex

This example shows how to read a DS18B20 temperature sensor using the ooRexx Stream class. With the help of the stream object the data of the sensor are read.

5.3.1.1 Circuit

Even if the sensor is called 1-wire, at least 2 cables must be connected. The term 1-wire comes from the fact that there is only one data line. In this example, however, a breakout board with a 1-wire sensor is used, so even three wires must be connected here. The VCC connection must be connected to the power supply pin of 3.3V. The GND connection must be connected to Ground and the Data connection must be connected to the GPIO 7 pin.(see Figure 15)

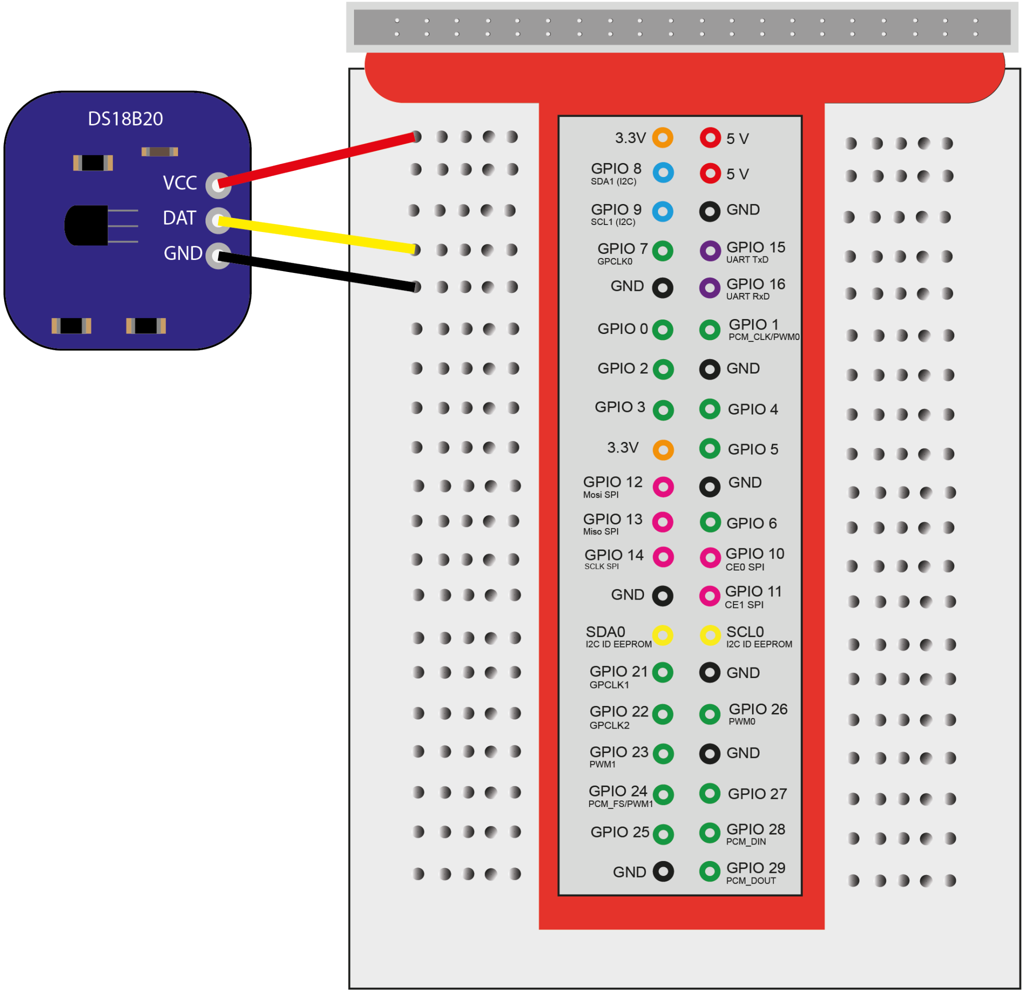


Figure 15: ds18b20 circuit

5.3.1.2 Sourcecode

With the code from Listing 38 the sensor is read out and simultaneously output to the console.

```
say "current Temp: "ds18b20GetTemp()
exit
```

Listing 38: ds18b20 output temp

The routine ds18b20GetTemp is used to return the read temperature value of the DS18B20 sensor. In line 17 the command to read out all 1-Wire sensors is stored in the variable ”Sensors”. This command is then executed in line 19 and the result is stored in an array named ”sen”. The name of the first sensor is stored in line 20 in the variable ”ds18b20”. After that a new stream object is created in line 21 which accesses the previously read sensor. The routine returns the read temperature value in line 22. The value must be divided by 1000, because e.g. a value of 22 degrees is returned as 22000 (see Listing 39).

```
16 ::Routine ds18b20GetTemp public
17 Sensors = "ls /sys/bus/w1/devices"
18 sen=.array~new
19 address system sensors with output using (sen)
20 ds18b20 = sen[1]
21 stream=.stream~new("/sys/bus/w1/devices/"||ds18b20||"/driver/"-
22 ||ds18b20||"/temperature")~~open
23 return stream~lineIn/1000
```

Listing 39: ds18b20 routine ds18b20GetTemp:stream

5.3.2 ds18b20unix.rex

There is also the possibility to read out the temperature measured by the sensor with the Unix command ”tail”. Through the ”address system” command of ooRexx it is very easy to use functions of the operating system.

5.3.2.1 Circuit

The wiring has not changed in this example compared to ds18b20stream(see Figure 15)

5.3.2.2 Sourcecode

The code to output the temperature is identical to the code of example ds18b20stream.rex because only the code in the routine has changed.(see Listing 40)

```
say "current Temp: "ds18b20GetTemp()
exit
```

Listing 40: ds18b20 output temp

In line 17 to 20, as in the ds18b20stream.rex example, first all available sensors are stored in an array and the first sensor is selected. In line 21 the tail command is stored in a variable so that it can be executed in line 23. The tail command reads the last line of the "w1_ slave" device. This line is then parsed and only the part after "t=" is stored in the variable grad. This value, as mentioned before, is too high by a factor of 1000, so the read value is divided by 1000 before it is returned.(see Listing 41)

```
16 ::Routine ds18b20GetTemp public
17 Sensors = "ls /sys/bus/w1/devices"
18 sen=.array~new
19 address system sensors with output using (sen)
20 ds18b20 = sen[1]
21 temp = ("tail -n 1 /sys/bus/w1/devices/"ds18b20"/driver/"ds18b20"/w1_slave")
22 value = .array~new
23 address system temp with output using (value)
24 parse var value "t=" grad
25 return grad/1000
```

Listing 41: ds18b20 routine ds18b20GetTemp:unix

5.3.3 ds18b20pi4oorex.rex

This example shows the third way how to access the 1-Wire temperature sensor DS18B20 and read out the temperature. For this purpose the pi4oorexx JAR library mentioned in chapter 2.9 is used, which uses a modified Java class of the diozero project to read the values of the sensor[JLe21]

5.3.3.1 Circuit

The wiring has not changed in this example compared to ds18b20stream(see Figure 15)

5.3.3.2 Sourcecode

The difference to the two previous examples of the DS18B20 temperature sensor is that Java is required for this example. The access to Java is implemented via BSF4ooRexx. This makes it possible to access the pi4oorexx library.
In line 17 you call the routine setupDs18b20. This is absolutely necessary otherwise the sensor is not available. (see Listing 42) Line 19 shows the return value of the routine ds18b20gettemp.

```
17 call setupDs18b20
18 /*outputs the currently measured temperature*/
19 say ds18b20GetTemp()
20 exit
```

Listing 42: ds18b20 output temp

The public routine returns the temperature of sensor 0. with the "Get" command the desired sensor is selected and the command "getTemperature" returns the measured value. (see Listing 43)

```
::routine ds18b20GetTemp public
return .s~get(0)~getTemperature
```

Listing 43: ds18b20 routine ds18b20GetTemp:pi4oorexx

In line 32 a local package is created. With this you have the possibility to make objects locally accessible. In line 33 the required Java class is loaded from the pi4oorexx library and line 34 the available sensors are made available locally via the variable ”s”. In line 36 the access to the Java classes is established via BSF4ooRexx.(see Listing 44)

```
31 ::routine setupDs18b20 public
32 pkgLocal=.context~package~local
33 sensor = bsf.loadClass("at.pi4oorexx.ds18b20.W1ThermSensor")
34 pkgLocal~s= sensor~getAvailableSensors
35 return
36 ::requires BSF.CLS
```

Listing 44: ds18b20 routine ds18b20setupDs18B20

5.4 UART

For the demonstration of the UART interface the author has chosen the control of a GPS sensor.

5.4.1 gps.rex

A Neo-6M GPS sensor is used. The Neo-6M sensor continuously polls the GPS data. The data is transmitted in NMEA format, then processed and output. The connection to the sensor is established with the help of Pi4J and BSF4ooRexx.

5.4.1.1 Circuit

The Neo-6M sensor requires a supply voltage of 5V. This is applied to the VCC pin. The GND pin is connected to one of the ground pins on the breadboard. When connecting the data line, please note that the RXD and TXD lines must be crossed out. As described in chapter 3.3.4.(see Figure 16)

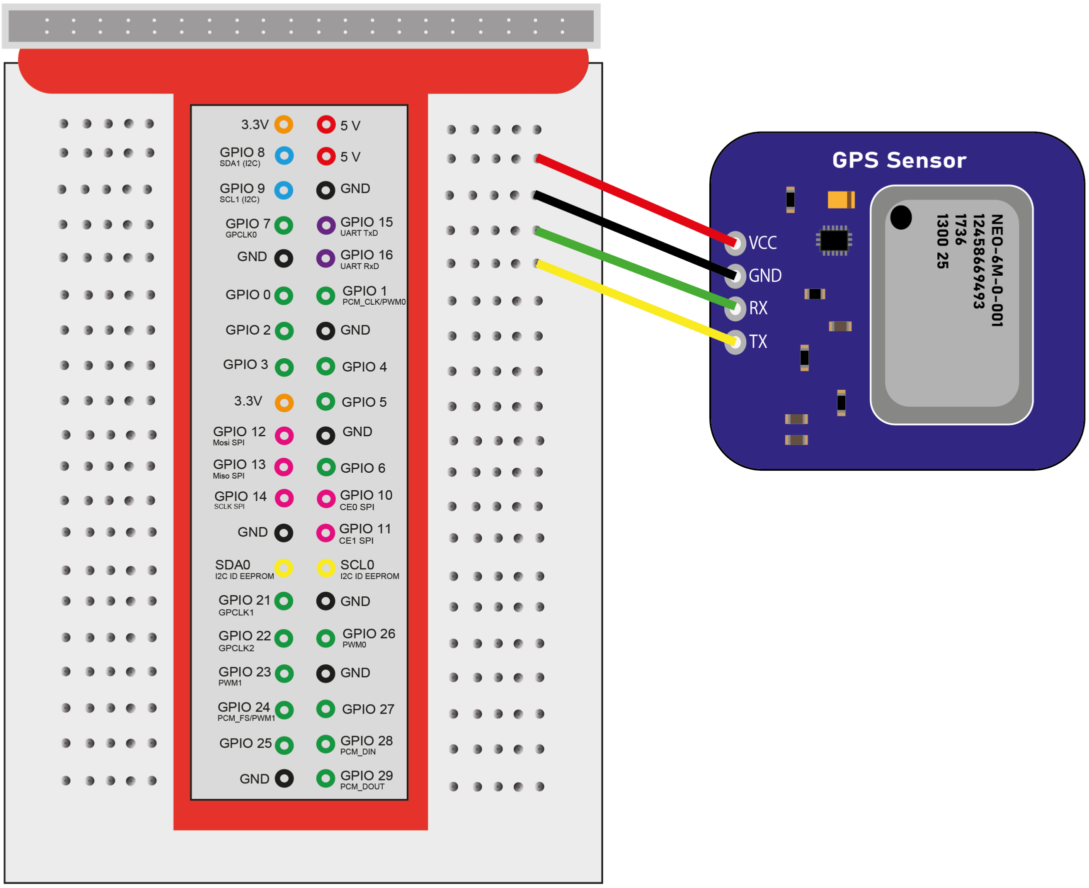


Figure 16: GPS Sensor circuit

5.4.1.2 Sourcecode

To connect to the Neo-6M sensor via the UART interface, some Java classes have to be loaded via BSF4ooRexx and a new object of class "SerialConfig" has to be created(see Listing 45)

```
serial = bsf.loadClass("com.pi4j.io.serial.SerialFactory")~createInstance
port = bsf.loadClass("com.pi4j.io.serial.SerialPort")
baud = bsf.loadClass("com.pi4j.io.serial.Baud")
dataBits = bsf.loadClass("com.pi4j.io.serial.DataBits")
parity= bsf.loadClass("com.pi4j.io.serial.Parity")
stopBits= bsf.loadClass("com.pi4j.io.serial.StopBits")
flowControl= bsf.loadClass("com.pi4j.io.serial.FlowControl")
config = .bsf~new("com.pi4j.io.serial.SerialConfig")
```

Listing 45: gps load requires Java classes

In Listing 46 the serial port is initialized. The GPS sensor uses the default serial port (ttyS0). The baudrate is set to 9600 and 8 databits are used. No parity bit is used and one stop bit. The flowcontrol is not used The individual parameters are explained in chapter

3.3.4. After all parameters are set, the config object created in Listing 45 are assigned parameters and then the serial connection is opened.

```
po=port~getDefaultPort
b=baud~_9600
d= dataBits~_8
pa = parity~NONE
st=stopBits~_1
f=flowControl~NONE
config~device(po)~baud(b)~dataBits(d)~parity(pa)~stopBits(st)~flowControl(f)
serial~open(config)
```

Listing 46: gps init serial port

The routine "readData" is used to read the data of the GPS sensor. For this the routine is passed the object of the serial interface in line 113. In line 114 a variable is created which stores all received data. In the loop in line 115 to 118 a total of 300 characters from the data stream are stored. The code shown in line 116 receives character by character, but in ascii format. However, this is also immediately converted from ascii format to character in line 117 when the string is assembled. Since each character is followed by a space, all spaces are removed from the string in line 120. Then, in line 122, the characters are split into two parts. The part desired for the output is the one after the string "$GPRMC". Then, in line 123, the string is split again. All characters after the string "$GPVTG" are truncated. Thus the string now consists of the desired characters of the NMEA coding [Ray22]. However, it must be noted that although the string consists of the desired characters, this string still contains the control characters for "carriage return" and "line feed". These are removed in line 124. If these characters are not removed, the CRC check will not match. At the end the desired data is returned by return command.(see Listing 47)

```
112 ::routine readData
113 use arg serial
114 datastring = ""
115 do i=1 to 300
116   a=serial~getInputStream
117   datastring = datastring a~read~d2c
118 end
119
120 datastring = space(datastring,0)
121 /*Cut out desired part (GPRMC) from the string*/
122 parse var datastring useless "$GPRMC," firstcut
123 parse var firstcut receivedDataRaw "$GPVTG" .
124 parse var receivedDataRaw receivedData +59
125 return receivedData
```

Listing 47: gps routine readData

Listing 48 shows the validateData routine, which is responsible for ensuring that the data received is valid. The data passed to the routine is first checked for its length. If no data is contained, the routine returns a .FALSE. this would mean that the data is not valid. If data is present, it is split in line 141. The desired $GPRMC NMEA data set consists of the GPS data followed by an asterisk (*) and the CRC checksum. This variable with the GPS data is then linked bitwise by XOR operation. At the end the calculated values from the data and the read checksum must match to confirm valid data. The verification of the data starts in line 148. Here a variable is assigned the character ”g”. The character ”g” is the result of the XOR operations of ”GPRMC” which was precalculated because this part was truncated in the routine ”readData” and was not appended separately. It should also be noted that the dollar sign and the asterisk character may still be used for the calculation of the checksum. In the loop between line 149 and 152 the individually characterized characters are XORed character by character. In line 153 the result of all XOR operations of a character is transferred to the hexadecimal notation. In line 155 it is then checked whether the calculated checksum matches the received checksum. If this is the case a .TRUE is returned, if not a .FALSE is returned.

```
136 ::routine validateData
137 parse arg checkData
138 /* check if data are available */
139 if checkData~length > 0 then do
140    /*read checksum from string */
141   parse var checkData data "*" checksum
142
143   /*Since "GPRMC," also belongs to the calculation of the checksum,
144     but this value has already been cut out to facilitate
145     the search process, the value precalculated value is
146     inserted into the variable a
147     the precalculated value results in a = "g" */
148   a="g"
149     do i=1 to data~length
150     erg = bitxor(a,data[i])
151     a=erg
152     end
153   checksum_calc = c2x(a)
154   /*check if the received checksum is equal to the calculated one*/
155   if checksum = checksum_calc then do
156     return .True
157   end
158   else do
159     return .False
160   end
161 end
162 else do
163   return .False
164 end
```

Listing 48: gps routine validateData

The routine ”prepareData” prepares the verified data in a predefined[Nov21] form. The following data can be output:

- Date

- Time
- Latitude
- Longitude

(see Listing 49)

```
::routine prepareData
parse arg gps
DataColl = .directory~new
parse var gps utc +6 13 lat +10 24 latDir +1 26 lon +11 38 lonDir +1 47 dat +6 .
--prepare Time
time =substr(utc,1,2) || ":"|| substr(utc,3,2) || ":" || substr(utc,5,2) "UTC"
DataColl ~~time = time
--prepare latitude
latitude = substr(lat,1,2) || "." || substr(lat,3,2) || substr(lat,6,5) || latDir
DataColl ~~latitude = latitude
--prepare longitude
longitude = substr(lon,1,3) || "." ||  substr(lon,4,2) || substr(lon,7,5) || lonDir
DataColl ~~longitude = longitude
--prepare Date
date = substr(dat,1,2) || "." || substr(dat,3,2) || "." || substr(dat,5,2) -- dd/mm/yy
DataColl ~~date =date
return dataColl
```

Listing 49: gps routine prepareData

The main program(Listing 50) starts with receiving the GPS data via readData routine. The data is stored in the variable GpsData. After that a counter is created in line 43 and initialized with the value 1. The while loop starting in line 44 is used to check the data for validity (per validateData routine). If the data is valid, the loop is immediately exited and the data is output (lines 60 - 63) and the serial connection is terminated (line 66). If the data is not valid, a warning is issued. After that, line 46 checks how many times the loop has already been run. The counter created before is used for this. If there are more than 10 attempts to read in data, the program is terminated with an error message. If it is not yet the tenth run, in line 50 a second is waited and then in line 51 a new reading attempt is started. With each run the counter (line 53) increases by one.

```
39 GpsData = readData(serial)
40
41 /*If no data is received more than ten times, the program is aborted
42 and an error message is displayed.*/
43 count = 1
44 do while (validateData(Gpsdata) = .False)
45     say "no valid Data Receiveid Nr." count
46   if count >= 10 then do
47     say "no valid data received- check antenna"
48     exit
49   end
50   call syssleep 1
```

```
  GpsData = readData(serial)
  --say GpsData               --debug
  count = count + 1
end

/* valid values were received and were checked by checksum*/

preparedData = prepareData(GpsData)

say preparedData~date
say preparedData~time
say preparedData~latitude
say preparedData~longitude

/*close the serial port*/
serial~close
exit 0
```

Listing 50: gps main program

5.5 SPI

In this chapter the control of devices via the SPI bus is explained.

5.5.1 LEDMatrixDriver.rex

The program LEDMatrixDriver.rex was developed to control a MAX7219 8x8 matrix LED display driver via SPI interface. The SPI interface is implemented via Pi4j using BSF4ooRexx. The driver provides routines for the output at the display and for the control of the display. There are the following routines:

- write: writes a Byte to the Display.
  The write command absolutely needs single char as values for register and value. e.g. write 1~x2c , 255~d2c
  register:1 ... 8 values from 0-255 as Char

- printString: outputs the entered string as a ticker.

- setup: loads all needed components, must be called first at the beginning of the program.

- init: ust be called after setup. Activates the chip so that it accepts commands

- clear: deletes the display content

- intensity: set brightness of the display. e.g. call intensity ”12”x , Min: ”01”x , Max:”15”x (in hexadecimal notation)

- close: switches off the MAX7219 chip and clears the display.

This example was inspired by `github.com/sharetop/max7219-java`

5.5.1.1 Circuit

The LED Matrix breakout board must be connected as follows. The VCC contact must be connected to the power supply of 3.3V. GND must be connected to a ground pin. DIN(Data In [from the chip's point of view]) must be connected to GPIO 12. CLK must be connected to GPIO 14 and CS must be connected to GPIO 10. This chip uses only three lines because the chip does not return a response to the Raspberry Pi.(see Figure 17)

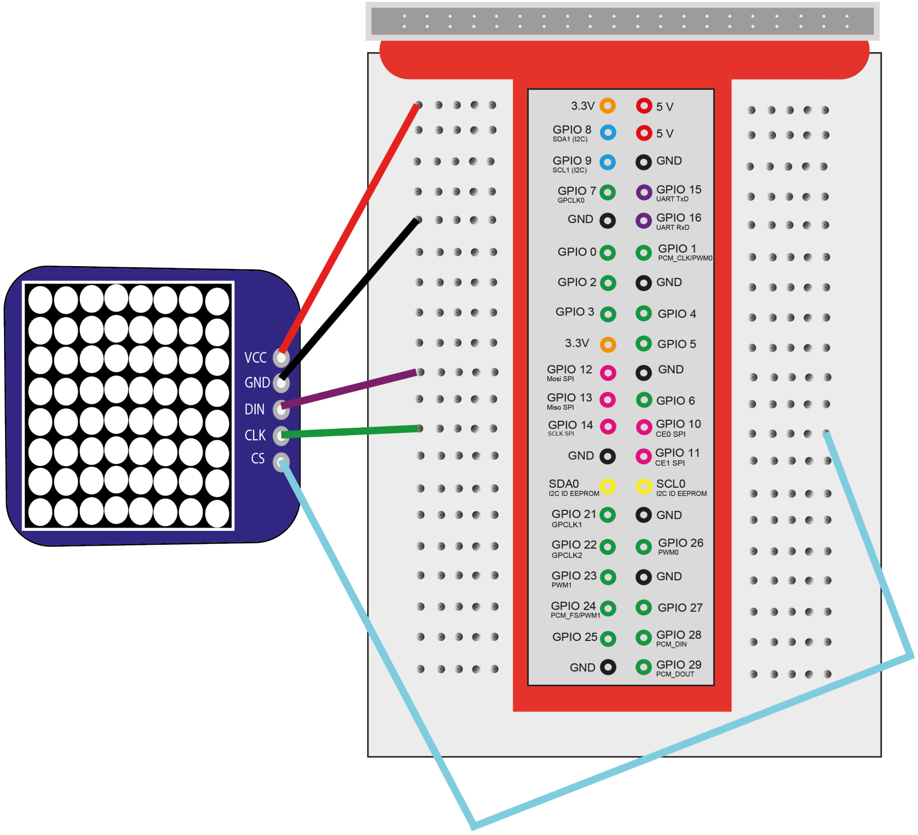


Figure 17: LED Matrix driver circuit

5.5.1.2 Sourcecode

The routine "setupLED"(Listing 51) is necessary to load all required Java classes, per BSF4ooRexx, for the SPI connection (see lines 221-225). These are then made available program-wide by the command in line 219. In line 232-236 the commands for the MAX7219 chip are made available in the local package. Line 239 shows the required registers for the clear sequence. Line 242 provides the required bytes for the representation of the characters which are loaded from lines 249 -377. For better readability, only one line of the resource ascii127fontHex is shown in this code snippet. The complete resource can be found in the source code in the provided file.

```
217 ::routine setupLED public
218
219    pkgLocal=.context~package~local  -- get package local directory
220
221    pkgLocal~clzSpiChannel=bsf.loadClass("com.pi4j.io.spi.SpiChannel")
222    pkgLocal~clzSpiDevice=bsf.loadClass("com.pi4j.io.spi.SpiDevice")
223    pkgLocal~spi = bsf.loadClass("com.pi4j.io.spi.SpiFactory")~getInstance(-
224    .clzSpiChannel~CS0,-
225    .clzSpiDevice~DEFAULT_SPI_SPEED,.clzSpiDevice~DEFAULT_SPI_MODE)
226    -- define some constants
227    /*Addres  Register*/
228    /* DATASHEET MAX7219
229    https://datasheets.maximintegrated.com/en/ds/MAX7219-MAX7221.pdf 06.02.2022
230     */
231
232    pkgLocal~MAX7219_REG_DECODEMODE ="09"x
233    pkgLocal~MAX7219_REG_INTENSITY  ="0a"x
234    pkgLocal~MAX7219_REG_SCANLIMIT  ="0b"x
235    pkgLocal~MAX7219_REG_SHUTDOWN   ="0c"x
236    pkgLocal~MAX7219_REG_DISPLAYTEST="0f"x
237
238    /* clear sequence */
239    pkgLocal~clearSequence= "01 02 03 04 05 06 07 08"x
240
241    /* define font: eight bytes per char */
242    pkgLocal~font=.resources~ascii127fontHex~makeString("line"," ")~space(1)~strip~x2c
243
244
245 /* Ascii 127 font
246   -->https://github.com/sharetop/max7219-java/blob/master/src/main/java/cn/sharetop/
        max7219/Font.java
247   (higher 128 code places seem to be defined for Codepage 437 (CP437) inferring from
        comments?)
248 */
249 ::resource ascii127fontHex
250    00 00 00 00 00 00 00 00
```

Listing 51: LEDMatrixDriver routine setupLED

The routine ”init” must be executed only at the first program call. As long as the routine close is not used, the chip remains ready for use. The command shutdown - ”01”x switches the chip active. After the close routine has been executed (shutdown - ”00”x), the ”init” routine must also be executed again. The calls of the write routine use the values defined before in setup and thereby initialize the chip. The values can be taken from the data sheet.[int21] (see Listing 52)

```
167 ::routine init public
```

```
   call write .MAX7219_REG_SCANLIMIT   , "07"x
   call write .MAX7219_REG_DECODEMODE  , "00"x
   call write .MAX7219_REG_DISPLAYTEST , "00"x
   call write .MAX7219_REG_SHUTDOWN    , "01"x
   call intensity "12"x
   return
```

Listing 52: LEDMatrixDriver routine init

The routine "write" in Listing 53, is the only routine that communicates directly with the Max7219 chip. "write" needs two values as character. The register of the chip and the value(line 204). In line 206 the values are converted to a Java byte array and sent to the chip.

```
203 ::routine write public
204    parse arg register , value
205        --combine both characters and convert them into a Java byte array
206    .spi~write(BsfRawBytes(register || value))
207    return
```

Listing 53: LEDMatrixDriver routine write

The routine speed is used to specify the speed of the ticker.(see Listing 54)

```
::routine speed
   use arg s
   call syssleep s
   return
```

Listing 54: LEDMatrixDriver routine speed

the routine "close" is used to clear the output on the display and to switch off the MAX7219 chip. For this the routine "clear" is called in line 180 and in line 181 the routine "write" is used to send the command to switch off the chip.(see Listing 55)

```
179 ::routine close public
180    call clear
181    call write .MAX7219_REG_SHUTDOWN , "00"x
182    return
```

Listing 55: LEDMatrixDriver routine close

The routine "clear" in Listing 56 writes zero values to the registers previously defined in "setup". This clears the output. Here it is to be noted that the first register is not, as usual, 0 but begins with 1.

```
::routine clear public
   do i=1 to .clearSequence~length
    call write .clearSequence[i], "00"x
   end
   return
```

Listing 56: LEDMatrixDriver routine clear

The "intensity" routine(Listing 57) is used to set the brightness of the display. For this a hexadecimal value between "00 "x and "15 "x must be passed. This value is sent to the chip in line 136.

```
134 ::routine intensity public
135    parse arg value
136    call write .MAX7219_REG_INTENSITY, value
137    return
```

Listing 57: LEDMatrixDriver routine intensity

the routine "ticker" (listing 58) is used to output the text on the display as a ticker. For this the ticker gets a character from the resource ascii127fontHex. This is already divided into eight bytes. In line 115 and 116 a prolog and epilog are appended to the character to be output. This consists of also byte with the value 0. This serves that the characters run in starting from the right edge and that the characters also disappear again completely from the matrix. in the loop in line 118 - 121 the characters are then sent byte by byte to the chip. In line 122 there is the possibility to set the speed of the ticker with the "speed" routine.

```
112 ::routine ticker
113    use arg charsAsFontBytes   -- already has 8 font bytes per character to be output
114
115    epilog=.font~substr(1,8)
116    prolog = epilog
117    charsAsFontBytes= prolog || charsAsFontBytes || epilog
118    do i=1 to (charsAsFontBytes~length-8)
119       do idx=1 to 8
120          call write idx~x2c, charsAsFontBytes[i+idx-1]
121       end
122    call speed 0.05
123    end
124    return
```

Listing 58: LEDMatrixDriver routine ticker

The routine "printString"(Listing 59) is used to split an input text into single characters, then transform them into 8 byte characters using the resource ascii127fontHex(lines 90-95) and assemble them into a mutable buffer and pass them to the routine "ticker"(line 96)

```
87 ::routine printString public
88    use arg string
89    mb=.mutableBuffer~new
90    do i=1 to string~length
91     decCode   = c2d(String[i])
92       pos=decCode*8+1
93       fontBytes = .font~substr(pos,8)
94       mb~append(fontBytes)
95    end
96    call ticker mb~string
97    return
```

Listing 59: LEDMatrixDriver routine printString

Because in this example the Pi4J library is used, it has to be loaded by BSF4ooRexx and therefore the command in Listing 60 is necessary

```
::requires "BSF.CLS"
```

Listing 60: LEDMatrixDriver java support

This program does not have its own main program. When this program is called, it only outputs that the driver has been loaded. However, it is possible to try out the driver. The driver contains a short example which requires an input from the user and then outputs this to the matrix display. However, this example is deactivated. To activate it, only the comment characters in lines 46 and 76 must be removed.

5.5.2 LEDMatrixPi4oorexx.rex

The example LEDMatrixPi4oorexx demonstrates the same functions but this example is based on a ready to use Java driver [sha] which is inserted into the pi4oorexx library. In the following source code all available routines are described.

5.5.2.1 Circuit

The wiring has not changed in this example compared to LEDMatrixDriver(see Figure 17)

5.5.2.2 Sourcecode

This code demonstrates the routine of the pi4oorexx matrix LED class. At the beginning(line 12) the class MATRIX must be loaded from the pi4oorexx library via BSF4ooRexx. After loading a new object of this class must be created (line 14). Line 16 demonstrates the execution of the ”Help” routine. This is used, as mentioned in chapter 2.9, to list all routines and to get more information about this class. The code in line 19 activates the MAX7219 chip. The ”clear” routine clears the content on the display. With the code in line 23 it is possible to change the orientation of the ticker text and in line 25 an entered text is output on the display by the ”showMessage” routine. In line 29 - 35 it is demostrated how to output arbitrary characters on the matrix display. For this an array with all registers is created in line 29. (The numbering of the registers goes from 1 to 8) The code in the loop outputs a triangle on the display starting with a dot. This number of characters then increases per loop pass until eight dots are displayed at the end. After a three-second program pause, the output is cleared again by the command in line 38.(see Listing 61)

```
11 -- Import Class
12 matrix = bsf.importClass("at.pi4oorexx.matrix.MATRIX")
13 -- create new Object
14 m = matrix~new
15 /*use the help function to learn more about the available routines*/
16 m~help
17 /* initialize the MAX7219 driver to be able to send data to the
18 display*/
19 m~open
20 /*clear display*/
```

```
m~clear
/*change orientation of scrolling Text */
m~orientation(0)       --0,90,180,270
/*print scrolling Text*/
m~showMessage("Hello ooRexx")

/*write customdata to the Matrix Display*/
-- all eight register of the Display
reg= .array~of(1~x2c,2~x2c,3~x2c,4~x2c,5~x2c,6~x2c,7~x2c,8~x2c)
-- draw a rectangle
do i = 1 to 8
  val = (2**i-1)~d2c
  m~_write(BSFRawBytes(reg[i]||val))
  call syssleep 0.5
end
call syssleep 3
--clear screen
m~clear

::requires BSF.CLS   -- get java Support
```

Listing 61: LEDMatrixPi4oorexx program

5.5.3 binaryclock.rex

The binaryClock example demonstrates the application of the LEDMatrixDriver from chapter 5.5.1 With the help of this driver a binary clock is generated on the display (see Figure 18). The time shown on the display is 14:06:47

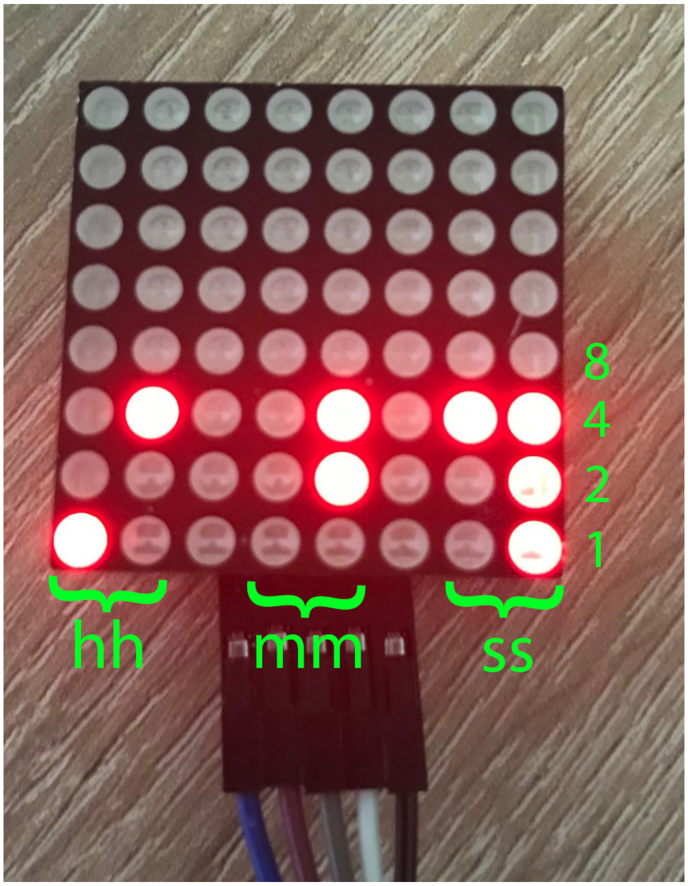


Figure 18: binary clock

5.5.3.1 Circuit

The wiring has not changed in this example compared to LEDMatrixDriver(see Figure 17)

5.5.3.2 Sourcecode

The program binaryClock(Listing 62) begins with the loading of all required classes per "setup "routine.(Line 12) Afterwards in line 15 the "init" routine is loaded. In line 19 an array with all register addresses is created. In the endless loop from line 21 to line 34 the time read in by the system is then prepared for the output. In line 22 the system time is loaded into the variable "time". Afterwards in line 25 all colons are replaced by zeros, because the binary clock should not display at the position of the colons. Further the generated string is inverted. This has the sense, so that the output on the display begins with hh:mm:ss at the left lower edge. In line 29 to 32 the individual characters are then output by "write" routine. In line 37 the driver for the LED matrix module is loaded.

```
11 -- get SPI Support
12 call setupLED
13 call syssleep 0.5
14 -- initialize the chip so that it can accept data
```

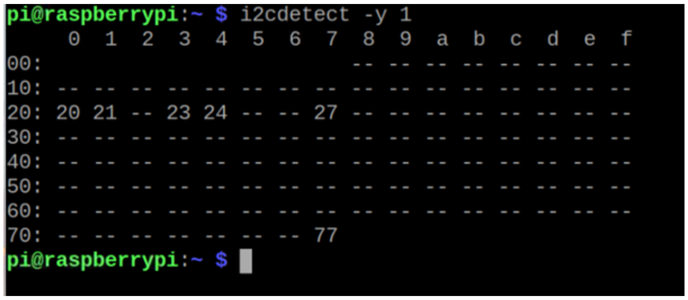
```
call init
call syssleep 0.5
/* create an array with the available registers and
 convert the numbers into characters*/
reg= .array~of(1~x2c,2~x2c,3~x2c,4~x2c,5~x2c,6~x2c,7~x2c,8~x2c)

do forever
  time = TIME()     --get system Time
  /*formats the read-in time for the output on the display
  colons are replaced by zeros and the string is inverted*/
  time = time~replaceAT("0",3)~replaceAT("0",6)~reverse
  /* console output in the usual format*/
  say time~reverse~replaceAT(":",3)~replaceAT(":",6)
  /*output the current time on the display. Byte by byte*/
  do i = 1 to 8
    call write reg[i] , time[i]~d2c
    call syssleep 0.02
  end
  call syssleep 0.1
end
exit

::requires "LEDMatrixDriver.rex"   --load Driver fir LED Matrix Modul
```

Listing 62: binary clock

5.6 I^{2}C

The I^{2}C interface offers a wide range of sensors and devices. From sensors that measure temperature, pressure, humidity or illuminance to extensions of GPIO ports and analog to digital converters. There are also devices that provide a clock or output devices such as an LC display. These devices will be discussed in more detail in this chapter.
Before the first devices and sensors are described, the tool ”I^{2}C Tools for Linux” mentioned in chapter 2.8 is explained. In this bachelor thesis three program from this tool are used.

1. **i2cdetect**
   This program can be used to display all available devices connected via I2c bus. In the given example (Figure 19) all devices connected to the I2C bus 1 are displayed.
   The ”-y” flag after the command means that you don’t have to confirm again and the ”1” means that the I^{2}C bus 1 is used.[Jar+a]



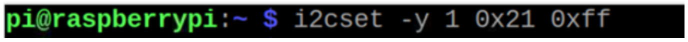
```
pi@raspberrypi:~ $ i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:                         -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: 20 21 -- 23 24 -- -- 27 -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- 77
pi@raspberrypi:~ $
```

Figure 19: i2cdetect

After a successful scan, the device address is displayed in the console. The device addresses are displayed in hexadecimal notation.

In Figure 19 shown, six devices are connected to I²C bus 1. The devices are also called slaves[Rat21] Address 20,21,24 are the address of the PCF8574 remote 8-bit I/O expander. address 23 is a digital ambient light sensor. address 27 is a LC display with PCF8574 backpack. Address 77 is a BME280 temperature/pressure/humidity sensor.

2. **i2cset**
   i2cset can be used to send data to the I2c device[Jar+c] In the given example(Figure 20) the value 0xff(= 255 decimal) is sent via i2cset to the device on bus 1 and the address 0x21.

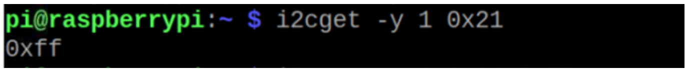


```
pi@raspberrypi:~ $ i2cset -y 1 0x21 0xff
```

Figure 20: i2cset

When using i2cset, care should be taken that only commands that can also be found in the data sheet of the I2C device are sent to the device. Improper use of i2cset could destroy the device, since it is possible to write to areas of the device that are not designed for this purpose, e.g. the memory DIMM of a serial EEPROM.[Jar+c]

3. **i2cget**
   i2cget can be used to read data from an i2cdevice.[Jar+b] The example(Figure 21) demonstrates the reading of data from device 0x21, which was previously sent to the device with i2cset.



```
pi@raspberrypi:~ $ i2cget -y 1 0x21
0xff
```

Figure 21: i2cget

5.6.1 bme280.rex

This example demonstrates the control of a BME 280 temperature/air pressure/humidity sensor via I^{2}C bus. The address of the device was determined with i2cdetect. The connection to the I^{2}C bus is established by Pi4J and BSF4ooRexx. Four public routines are provided so that they can also be called from other programs.

- SetupBme: is necessary to establish a connection to the I^{2}C bus. must be executed only once.
- getTemp: returns temperature in °C
- getPressure: returns pressure in hPa
- getHumidity: returns humidity in

The I^{2}C address of the BME280 can be read out via i2cdetect. In these examples it is 0x77 hex or 119 decimal.

5.6.1.1 Circuit

4 lines are required for the wiring. VCC is connected to the supply voltage of 3.3V. GND is connected to Ground. Data line SDA is connected to GPIO pin 8 and the signal line SCL is connected to GPIO pin 9.(see Figure 22)

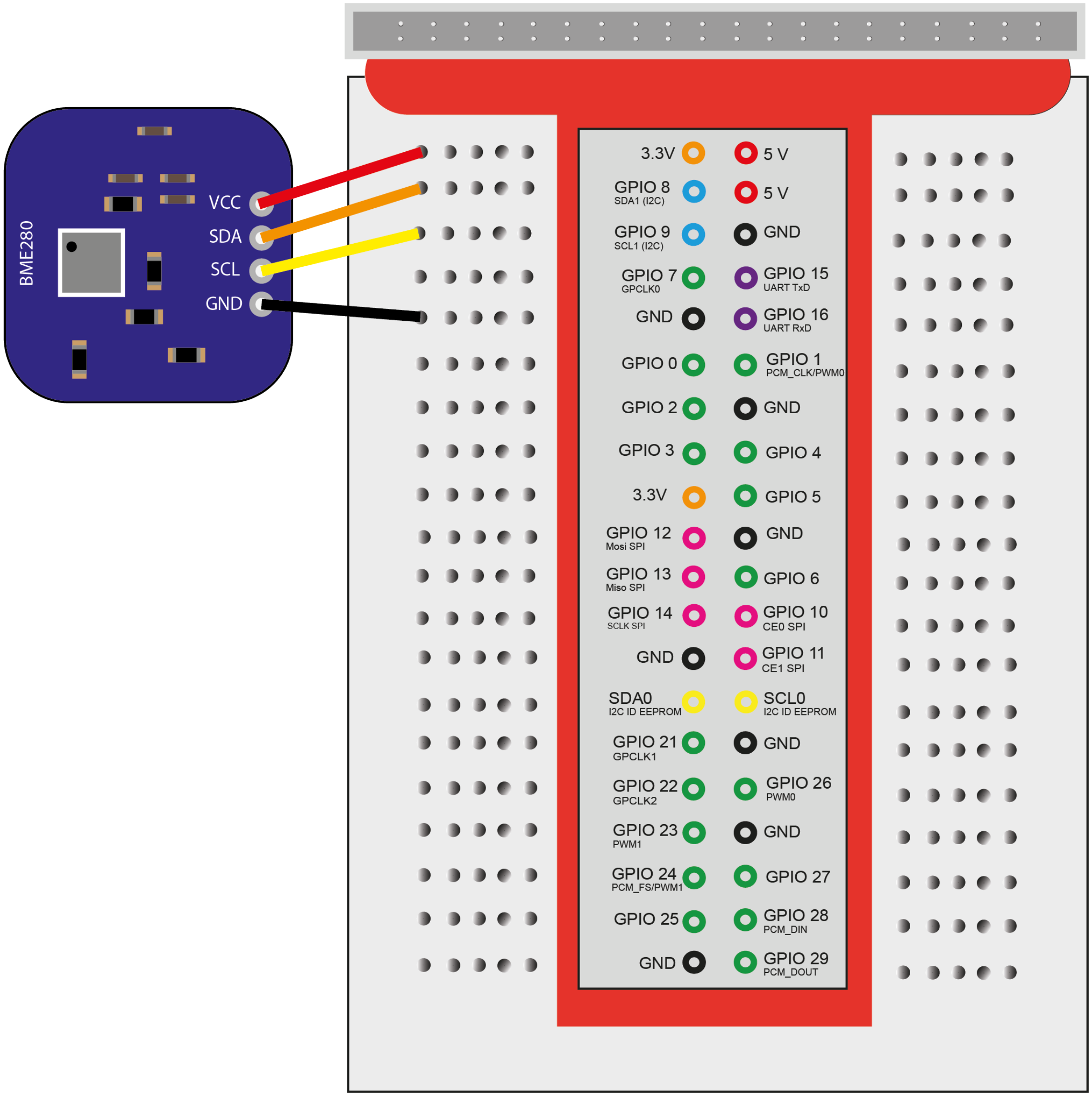


Figure 22: BME 280 circuit

5.6.1.2 Sourcecode

As already seen in the previous examples, the routine ”setupBme” loads the required Java classes and provides the required I²C connection in local package(see Listing 63)

```
::routine setupBme public
pkgLocal=.context~package~local  -- get package local directory
device = bsf.loadClass("com.pi4j.io.i2c.I2CDevice")
i2cbus = bsf.loadClass("com.pi4j.io.i2c.I2CBus")
bus = bsf.loadClass("com.pi4j.io.i2c.I2CFactory")~getInstance(i2cbus~BUS_1)
pkgLocal~device = bus~getDevice(119) --0x77 =    hex -> int
```

```
return
```

Listing 63: bme280 routine setupBme

The routine "0xFF" Converts the measured byte from a signed byte(-128 ... 127) to an unsigned byte(0 ... 255) (see Listing 64)

```
::routine 0xFF
use arg v

if v < 0 then
  do
    v= v+256
    return v
  end
else return v
```

Listing 64: bme280 routine 0xFF

The routine "0xF" the 0xf routine returns a value between 0-15.(see Listing 65)

```
::routine 0xF
use arg v
return 0xff(v)//16
```

Listing 65: bme280 routine 0xF

The routine "calcValues"(see Listing 66) is the part of the program that reads the data from the sensor and subsequently calculates and returns the temperature, air pressure and humidity. The calculations were done based on the datasheet or the Java driver from Naoki Ikeguchi. [Ike18] In line 52 a Java byte array is created by BSF4ooRexx with a capacity of 24 bytes, here you have to pay attention if a primitive type (byte.class) or a reference type (Byte.class) is needed.[pro] In line 53 data are read from the sensor of address 136, these are then stored in the line 52 generated byte array, the 0 means that there is no offset when reading, so should be read from the beginning and the 24 indicates that 24 bytes should be read.

Then the coefficients for temperature and air pressure are calculated (lines 58 - 92). In line 96 again a Java byte array is created and a byte is read from the sensor(address 161) (line 97) In line 104, 7 bytes are read from the sensor into a Java byte array created in the previous line. All these values are necessary for the calculation.

Between lines 108 and 120 the calculation of the coefficients of humidity is done.

After that, three commands are sent to the BME chip in lines 124 - 138 which trigger the chip to start the measurement. The measured values are then read out in line 146. The values are then calculated from line 150 - 185. In line 187 an ooRexx directory is created, into which the measured values are fed (lines 188-190) and then returned (line 192).

```
50 ::routine calcValues
51 -- read compesations parameter
52 data = bsf.createJavaArray("byte.class",24)
53 .device~read(136,data,0,24)
```

```
-- convert Data     Datasheet Table 16
-- temperature coeffocients

dig_T1 = (0xff(data[1])) + ((0xff(data[2]))*256)

dig_T2 = (0xff(data[3])) + ((0xff(data[4]))*256)
if dig_T2 > 32767 then dig_T2 = dig_T2 - 65536

dig_T3 = (0xff(data[5])) + ((0xff(data[6]))*256)
if dig_T3 > 32767 then dig_T3 = dig_T3 - 65536

-- pressure coefficients

dig_P1 = (0xff(data[7])) + ((0xff(data[8]))*256)

dig_P2 = (0xff(data[9])) + ((0xff(data[10]))*256)
if dig_P2 > 32767 then dig_P2 = dig_P2 - 65536

dig_P3 = (0xff(data[11])) + ((0xff(data[12]))*256)
if dig_P3 > 32767 then dig_P3 = dig_P3 - 65536

dig_P4 = (0xff(data[13])) + ((0xff(data[14]))*256)
if dig_P4 > 32767 then dig_P4 = dig_P4 - 65536

dig_P5 = (0xff(data[15])) + ((0xff(data[16]))*256)
if dig_P5 > 32767 then dig_P5 = dig_P5 - 65536

dig_P6 = (0xff(data[17])) + ((0xff(data[18]))*256)
if dig_P6 > 32767 then dig_P6 = dig_P6 - 65536

dig_P7 = (0xff(data[19])) + ((0xff(data[20]))*256)
if dig_P7 > 32767 then dig_P7 = dig_P7 - 65536

dig_P8 = (0xff(data[21])) + ((0xff(data[22]))*256)
if dig_P8 > 32767 then dig_P8 = dig_P8 - 65536

dig_P9 = (0xff(data[23])) + ((0xff(data[24]))*256)
if dig_P9 > 32767 then dig_P9 = dig_P9 - 65536

-- Read dig_H1 from 0xA1 -> 161

data_H1 = bsf.createJavaArray("byte.class",1)
.device~read(161,data_H1,0,1)

dig_H1 = (0xff(data_H1[1]))

-- Read 7 Bytes from 0xE1  -> 225

data2 = bsf.createJavaArray("byte.class",7)
.device~read(225,data2,0,7)
```

```

-- humidity coefficients

dig_H2 = (0xff(data2[1]) + (data2[2]*256))
if dig_H2 > 32767 then dig_H2 = dig_H2 - 65536

dig_H3 = 0xff(data2[3])

dig_H4 = ((0xff(data2[4])*16) +(0xf(data2[5])))       --- 0xff 0xf
if dig_H4 > 32767 then dig_H4 = dig_H4 - 65536

dig_H5 = ((0xff(data2[5])/16) +(0xff(data2[6])*16))       --- 0xff 0xff
if dig_H5 > 32767 then dig_H5 = dig_H5 - 65536

dig_H6 = (0xff(data2[7]))
if dig_H6 > 127 then dig_H6 = dig_H6 - 256

--select control humidity register

com1 = bsf.createJavaArray("byte.class",1)
com1~put(box("byte.class",1),1) -- 0x01 = 1
.device~write(242,com1)     --0xF2 = 242

--select control measurement register

com2 = bsf.createJavaArray("byte.class",1)
com2~put(box("byte.class",39),1)  --0x27 = 39
.device~write(244,com2)     --0xF4 = 244

--select config register

com3 = bsf.createJavaArray("byte.class",1)
com3~put(box("byte.class",-96),1)     --- 0xA0 -> -96 (byte)  --->java Byte
.device~write(242,com3)     --0xF5 = 245


call syssleep 1       -- pause

--read measured data from 0xF7 -> 247    8 Byte

meas = bsf.createJavaArray("byte.class",8)
.device~read(247,meas,0,8)

-- convert pressure and temp

adc_p = ((0xff(meas[1])*65536) + (0xff(meas[2])*256) + (0xff(meas[3])))/16
adc_t = ((0xff(meas[4])*65536) + (0xff(meas[5])*256) + (0xff(meas[6])))/16

-- convert humidity data

adc_h = (0xff(meas[7])*256)+(0xff(meas[8]))
```

```

--Temp offset calculation

var1 = ((adc_t / 16384) - (dig_T1 / 1024)) * dig_T2
var2 = (((adc_t / 131072) -(dig_T1 / 8192)) * ((adc_t / 131072) - (dig_T1 / 8192)))-
*  dig_T3 --131072 = 2^16
t_fine = var1 + var2
temp = (t_fine) / 5120

-- pressure offset calculation

var3 = (t_fine / 2) - 64000
var4 = var3 * var3 * dig_P6 / 32768
var4 = var4 * var3 * dig_P5 * 2
var4 = (var4 / 4) + (dig_P4 * 65536)
var3 = (dig_P3 * var3 * var3 / 524288 + dig_P2 * var3) / 524288
var3 = (1 + var3 / 32768) * dig_P1

p = 1048576 - adc_p
p = (p-(var4/4096))*6250 / var3
var3 = dig_P9 * p * p / 2147483648
var4 = p * dig_P8 / 32768

pressure = (p+ (var3 + var4 + dig_P7) / 16 ) / 100

-- humidity offset calculation

var_H = t_fine - 76800
var_H = (adc_h - (dig_H4 * 64 + dig_H5/16384 * var_H))* (dig_H2 / 65536 * -
( 1 + dig_H6 / 67108864 * var_H * (1 + dig_H3/67108864 * var_H)))
humidity = var_H * ( 1 - dig_H1 * var_H / 524288)

DataColl = .directory~new
DataColl~~temp = temp
DataColl~~pressure = pressure
DataColl~~humidity = humidity

return DataColl
```

Listing 66: bme280 routine calcValues

The routine "getTemp" , "getPressure" and "getHumidity" return the measured values(see Listing 67).

```
::routine getTemp public
temp = calcValues()
return temp~temp
---------------
---------------
/*return Pressure*/
::routine getPressure public
p = calcValues()
```

```
return p~pressure
---------------
---------------
/*return humidity*/
::routine getHumidity public
h = calcValues()
return h~humidity
```

Listing 67: bme280 get routines

Because in this example the Pi4J library is used, it has to be loaded by BSF4ooRexx and therefore the command in Listing 68 is necessary.

```
::requires "BSF.CLS"
```

Listing 68: BME java support

To demonstrate the sensor, a short example has been included in the code (see listing 69) which first executes the ”setupBME” routine and then outputs temperature, pressure and humidity(see Listing 69).

```
call setupBme

say getTemp()
say getPressure()
say getHumidity()

exit 0
```

Listing 69: bme280 example program

5.6.2 bme280pi4oorexx.rex

The example bme280pi4oorexx shows how to read the sensor with only a few lines of code using BSF4ooRexx and the pi4oorexx library.

5.6.2.1 Circuit

The wiring has not changed in this example compared to BME280(see Figure 22).

5.6.2.2 Sourcecode

The source code for the BME280 sensor is very short thanks to the pi4oorexx library, because the calculations are done in the implemented Java class. In line 11 the necessary Java class is loaded from BSF4ooRexx. In line 12 the ”help” routine is called, which prints information about the class on the console. In line 13 a measurement is performed. It is important that each time before the temperature, pressure or humidity are retrieved (line 14-17) a new measurement must be performed. Line 18 loads the required Java support via BSF4ooRexx.

```
11 b = bsf.loadClass("at.pi4oorexx.bme280.BME280")
12 b~help
```

```
b~measure
say b~getPressure
say b~getTempCelcius
say b~getTempFahrenheit
say b~getHumidity
::requires bsf.cls
```

Listing 70: bme280pi4oorexx example program

5.6.3 pcf8574.rex

This example shows the control of the PCF8574 I/O port expander via I^{2}C bus. Thus it is possible to switch eight outputs(LED) with two lines. For this, data or a byte is sent to the PCF8574 via i2cset, which converts this into individual bits and switches the outputs accordingly. i2cset expects the address of the PCF8574 as well as the byte to be transmitted in hexadecimal notation, as shown in Figure 20. The address is determined by i2cdetect(see Figure 19). the byte to be transmitted is randomly generated by ooRexx. For this you should be familiar with the conversion from binary to hexadecimal number system.
In table 4 a byte(8 bit) is shown as binary number. The first line shows the digit value in the decimal system and the third line shows a random byte with the decimal value of 186. This can also be verified by adding the decimal digit values: $128 + 32 + 16 + 8 + 2 = 186$. The I^{2}C address of the display can be read out via i2cdetect. In these examples it is 0x21 hex.

128	64	32	16	8	4	2	1
Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
1	0	1	1	1	0	1	0

Table 4: One Byte in Bits

In Table 5, the decimal numbers from 0 - 15 are output as binary and hexadecimal numbers, but since a byte consists of 8 bits and not four, it is composed of two half-bytes or nibbles. For this, two hexadecimal characters are simply strung together. In this way, it is possible to generate a byte with a decimal value range of 0 - 255, as shown in Table 4.

decimal	binary				hexadecimal
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	2
3	0	0	1	1	3
4	0	1	0	0	4
5	0	1	0	1	5
6	0	1	1	0	6
7	0	1	1	1	7
8	1	0	0	0	8
9	1	0	0	1	9
10	1	0	1	0	A
11	1	0	1	1	B
12	1	1	0	0	C
13	1	1	0	1	D
14	1	1	1	0	E
15	1	1	1	1	F

Table 5: Dec vs Bin vs Hex

Another function of the program is to count binary from 0 to 255. This is then output on the eight LEDs binary coded.

5.6.3.1 Circuit

The wiring is the same as for any I^{2}C device. VCC belongs to the 3.3V power supply. Gnd is connected to Ground. The I^{2}C line SDA belongs to GPIO pin 8 and SCL to GPIO pin 9. To control the eight contacts P1 - P8 must be connected to the contacts on the LED board L1 - L8. The separate GND pin on the LED board still has to be connected to ground. (see Figure 23)

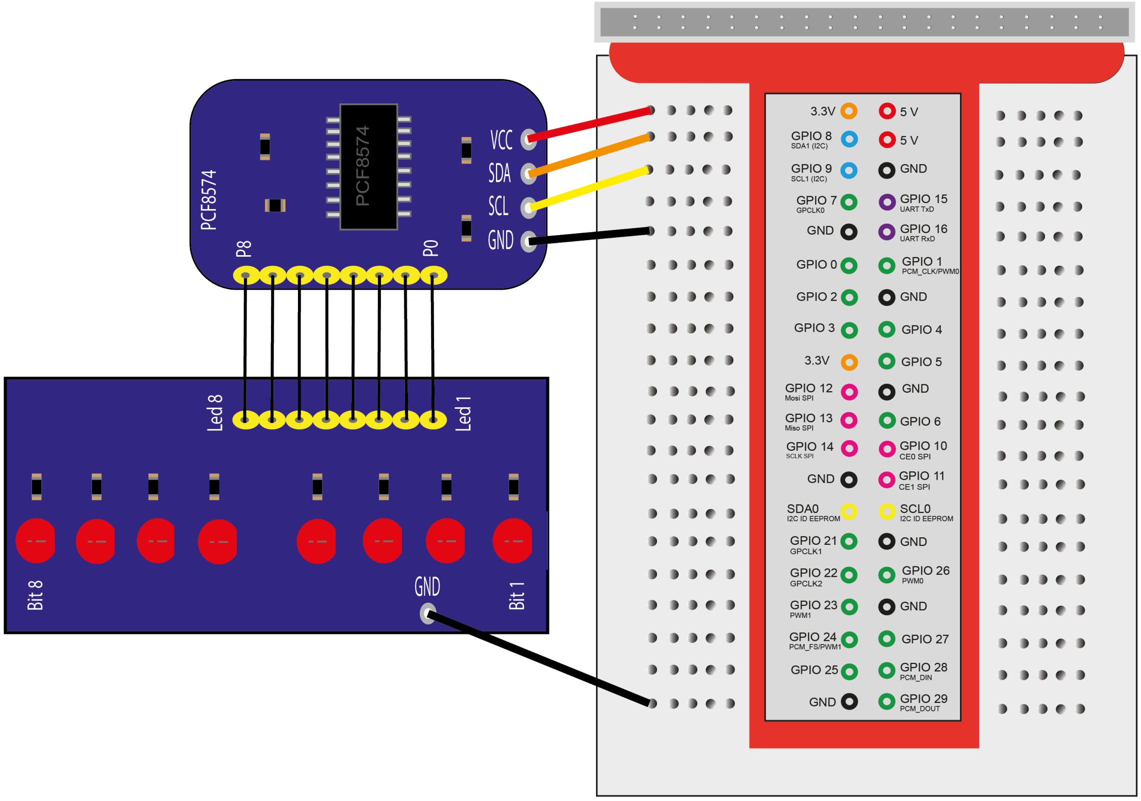


Figure 23: PCF 8574 circuit

5.6.3.2 Sourcecode

In line 11 the address of the PCF8574 is entered in hexadeximal notation. The address can be determined with i2cdetect (see Figure 19). In line 12 a binary number is declared which represents decimal 0 to switch all outputs to 0 or low. In line 13 this command is executed by i2cset. Here it must be noted that i2cset expects a hexadecimal number in the form of 0x00 to 0xff and therefore the previously defined binary number is converted to a hexadecimal number. ooRexx does not use the prefix "0x" in the hexadecimal notation therefore this must be added by hand. In line 16 to 23 8 random bits are generated which are assembled to a byte in line 25. The eighth bit with the highest value comes first. As last the first bit with the lowest place value is appended. This generated byte is then sent to the PCF8574 by i2cset in line 27. In line 29 the program is stopped for 5 seconds to see the output.
After that the second program section(line 32 to 37) starts. Here is counted binary from 0 to 255 and this is shown by LED. First the number to be counted away from is defined in line 32. Then in the loop the respective number is sent via i2cset to the PCF8574. For this, however, the decimal number must first be converted into a hexadecimal number. In line 35 the number is increased by one. The loop runs up to 256 so that all numbers from 0 to 255 can be output on the LED board.

```
11 addr = "0x21"    -- get address with i2cdetect
12 clear = "00000000"b
```

```
address system "i2cset -y 1 " addr "0x"clear~c2x  -- set all output to 0

--create random Bits
bit1= random(0,1)
bit2= random(0,1)
bit3= random(0,1)
bit4= random(0,1)
bit5= random(0,1)
bit6= random(0,1)
bit7= random(0,1)
bit8= random(0,1)

byte = bit8|| bit7 || bit6 || bit5 || bit4 || bit3 || bit2 || bit1 -- create byte

address system "i2cset -y 1 " addr "0x"byte~b2x --displays the randomly generated byte

call syssleep 5

-- count binary from 0 to 255
val = 0
do i = 1 to 256
  address system "i2cset -y 1 " addr "0x"val~d2x
  val = val+1
  call syssleep 0.3
end
exit
```

Listing 71: pcf8574 program

5.6.4 pcf8591.rex

Since the Raspberry Pi has no analog inputs, the pcf8591 example uses an analog to digital converter (ADC). The PCF8591 ADC offers four analog inputs with a resolution of 8 bit. This means that values between 0 and 255 (decimal) can be displayed. The program has a routine ”getAnalogInput(n)” with which the four analog inputs can be queried. The analog inputs are numbered from n= 0 ... 3. In this example, for illustration, a potentiometer is connected to the analog input 2 to perform measurements. However, all four inputs can be used simultaneously. The I^2C address of the PCF8591 can be read out via i2cdetect. In these examples it is 0x48 hex.

5.6.4.1 Circuit

The wiring of the PCF8591 ADC is like all I^2C devices very simple because only four lines are needed. VCC must be connected to the power supply of 3.3V. GND must be connected to Ground. The data line SDA must be connected to GPIO pin 8 and the clock line SCL must be connected to GPIO pin 9.

Since a measurement is also to be shown, the potentiometer must be connected as follows. VCC must be connected to the power supply of 3.3V and GND must be connected to Ground.

The analog output signal SIG must be connected to the PCF8591 on AI2. It would also be possible to use one of the other analog inputs.(see Figure 24)

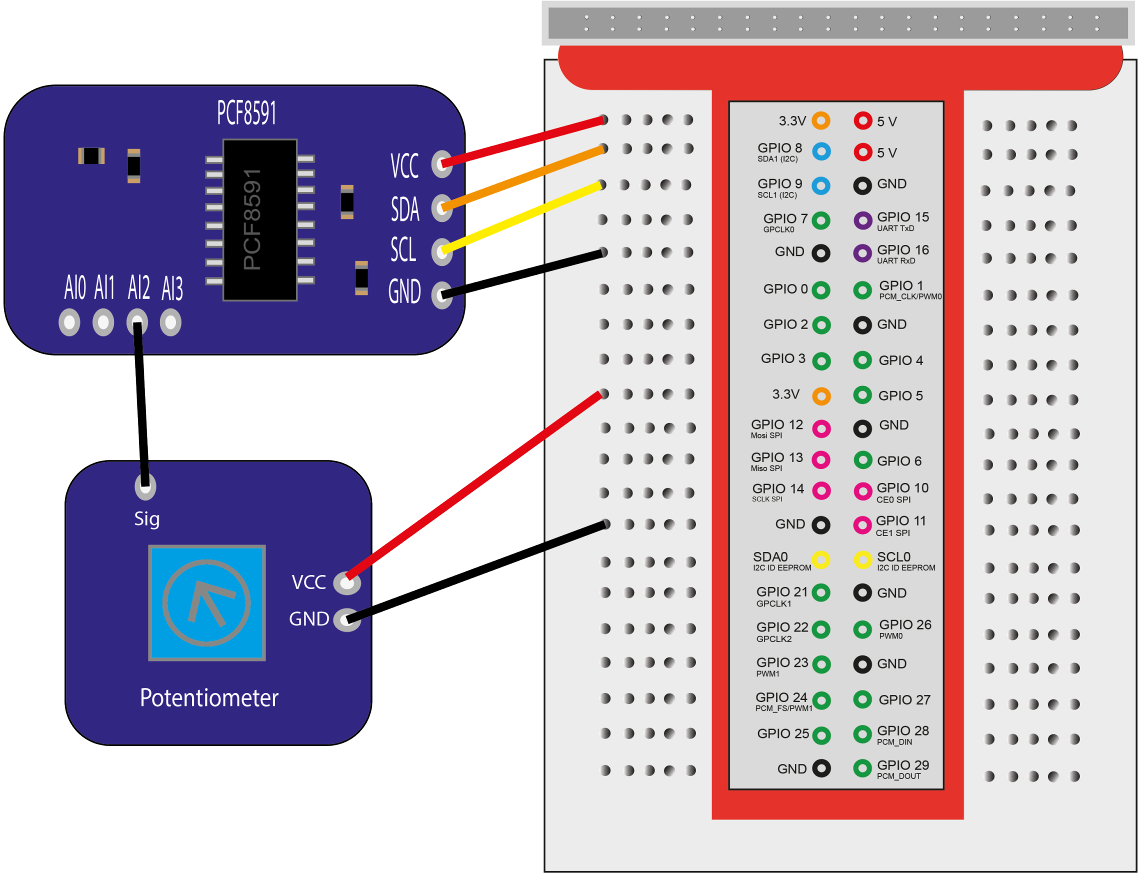


Figure 24: PCF 8591 circuit

5.6.4.2 Sourcecode

The routine "getAnalogInput"(see Listing 72) expects a parameter with the desired analog input (line 24) After that the input is checked for correctness and the correct chip address (e.g. for input 0 the address is 0x40) is stored in the variable "addr". In line 32 an array is created which stores the value received by i2cget command. In line 33 the analog input is queried for the first time, but this value contains the value of the previous measurement, so in line 34 a new query is made and stored in the array. In line 36 the value of the second measurement is returned, but before that it is converted from hexadecimal to decimal. "substr" is used because i2cget returns a value in the form of e.g. "0xf5" and ooRexx needs hexadecimal numbers in the form "f5", therefore the first two characters "0x" of the received value are truncated.

```
23 ::routine getAnalogInput public
24 use arg input
25
```

```
if input = 0 then addr = "0x40"
else if input = 1 then addr = "0x41"
else if input = 2 then addr = "0x42"
else if input = 3 then addr = "0x43"
else say "falsche eingabe"

value = .array~new
address system "i2cget -y 1 0x48 " addr with output append using(value)  -- dummy
address system "i2cget -y 1 0x48 " addr with output append using(value)

return x2d(substr(value[2] ,3))  -- 0xf5 -> f5
```

Listing 72: pcf8594 routine getAnalogInput

Listing 73 shows the value of the potentiometer connected to Analog Input 2 which is read out and displayed.

```
say getAnalogInput(2)
exit
```

Listing 73: pcf8594 main program

5.6.5 ds3231.rex

This example shows the use of a DS3231 RTC module, which provides time and date via I^{2}C bus. This module is useful when the Raspberry Pi is operated without an active internet connection, because the Raspberry Pi does not have a backup battery and therefore does not save the time when it is switched off. This module has a buffer battery that saves the time and date even after it is switched off.
There are two routines available to recover the time and date from the DS3231. The routine ”getTime” returns the time in the format HH:MM:SS. The routine ”getDate” returns the date in the format YY:MM:DD. There are also two routines to set the time and date. With the routine ”setTime HH ,MM ,SS” the time can be set and with the routine ”setDate YY, MM, DD” the date can be set.
It is still to be mentioned that the Real time clock does not count decimally but hexadecimally. However, this hexadecimal value represents the time decimally. This means for example that seconds are returned in the hexadecimal notation ”0x00” to ”0x59”. Also there is the ”setupDS3231” routine, which loads all needed classes via BSF4ooRexx and makes them available in the local package. The I^{2}C address of the DS3231 can be read out via i2cdetect. In these examples it is 0x68 hex or 104 decimal.

5.6.5.1 Circuit

The wiring is the same as in all previous I^{2}C examples. VCC is connected to the power supply of 3.3V, GND is connected to Ground and the two I^{2}C lines are connected to SDA on GPIO pin 8 and SCL on GPIO pin 9.(see Figure 25)

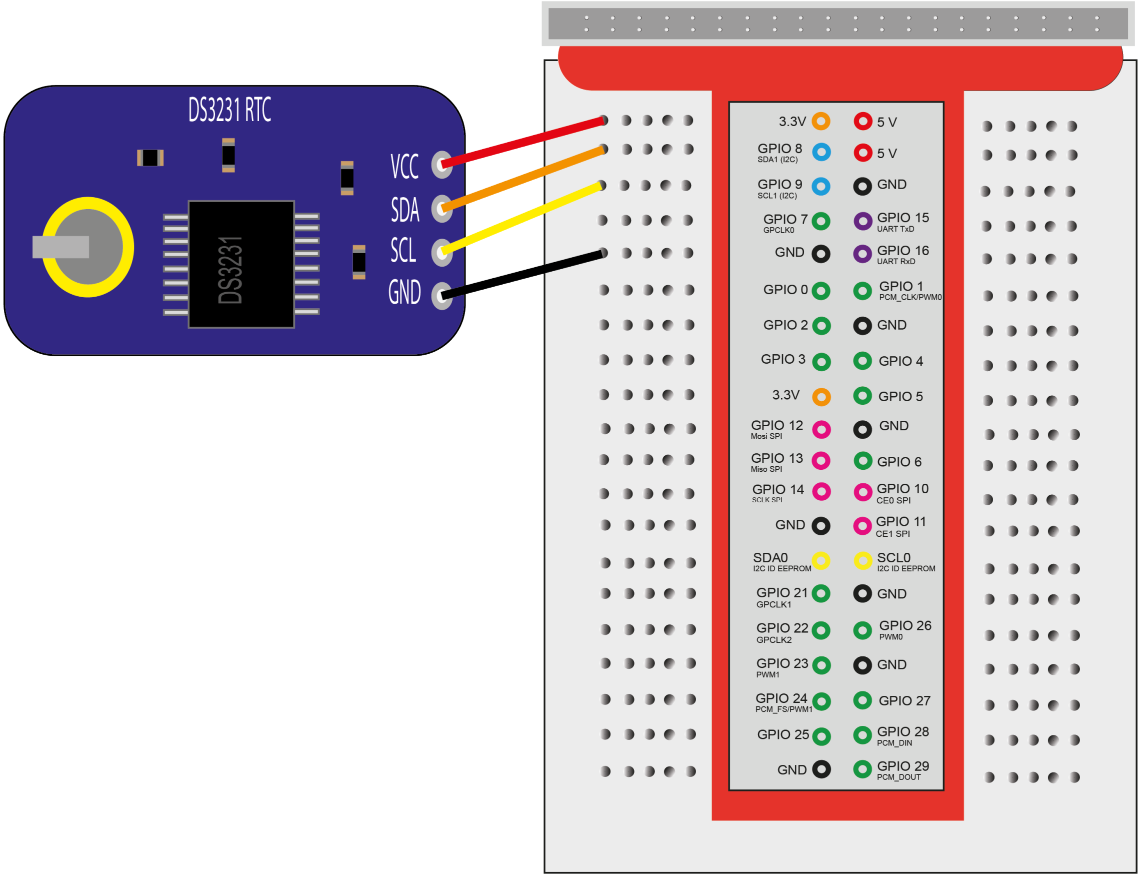


Figure 25: DS3231 Real Time Clock circuit

5.6.5.2 Sourcecode

The routine ”getTime” (see Listing 74) reads the data from the individual registers of the chip which store the time. Register 0 contains the seconds, register 1 the minutes and register 2 the hours. In line 34 a Java byte array is created which caches the read values. In line 35, register 0 is read and the received data is stored in an array created in line 34. In line 36, this value is then read and converted from decimal to hexadecimal and stored in variable ”s”. Without the conversion, ooRexx would interpret the read hexadecimal value as a decimal number and output it incorrectly. This process is repeated in lines 37 - 40 also for minutes and hours. In line 41 the time is then stored in the variable ”time” in the usual form and in line 42 this value is returned as a string.

```
33 ::routine getTime public
34 read = bsf.CreateJavaArray("byte.class",2)
35 .device~read(0,read,0,1)
36 s= read[1]~d2x
37 .device~read(1,read,0,1)
38 m= read[1]~d2x
39 .device~read(2,read,0,1)
40 h= read[1]~d2x
```

```
time = h":"m":"s
return time
```

Listing 74: ds3231 routine getTime

The routine "getDate" (see Listing 75) does the same as the routine "getTime" only for "getDate" the registers 4 for days, 5 for month and 6 for year are read and returned.

```
::routine getDate public
read = bsf.CreateJavaArray("byte.class",1)
.device~read(4,read,0,1)
d= read[1]~d2x
.device~read(5,read,0,1)
mo= read[1]~d2x
.device~read(6,read,0,1)
y= read[1]~d2x
date = y"."mo"."d
return date
```

Listing 75: ds3231 routine getDate

The routine "setTime"(see Listing 76) can be used to set the time. Here three parameters for the desired time in hours(hh), minutes(mm) and seconds(ss) must be passed to the routine (line 61). These are then written to the respective registers in lines 62-64. Since the routine BSFRawBytes is used, which only accepts character, the passed value must be converted from hexadecimal notation to a character.

```
60 ::routine setTime public
61 use arg hh,mm,ss
62 .device~write(0,BSFRawBytes(ss~x2c))
63 .device~write(1,BSFRawBytes(mm~x2c))
64 .device~write(2,BSFRawBytes(hh~x2c))
65 return
```

Listing 76: ds3231 routine setTime

The routine "setDate" (see Listing 77) is changed to the routine "setTime" again only in the registers to be written. For this the passed parameters are written into the respective registers. The values must be passed in the format year(yy), month(momo) and day(dd).

```
::routine setDate public
use arg yy, momo,dd
.device~write(4,BSFRawBytes(dd~x2c))
.device~write(5,BSFRawBytes(momo~x2c))
.device~write(6,BSFRawBytes(yy~x2c))
return
```

Listing 77: ds3231 routine setDate

The routine "setupDs3231" (see Listing 78) establishes the connection to the required Java classes and then makes them available in the local package.

```
::routine setupDs3231 public
pkgLocal=.context~package~local  -- get package local directory
device = bsf.loadClass("com.pi4j.io.i2c.I2CDevice")
i2cbus = bsf.loadClass("com.pi4j.io.i2c.I2CBus")
bus = bsf.loadClass("com.pi4j.io.i2c.I2CFactory")~getInstance(i2cbus~BUS_1)
pkgLocal~device = bus~getDevice(104)
```

Listing 78: ds3231 routine setupDs3231

Listing 79 shows a sample output as the main program. In line 16 the SetupRoutine is called to get access to all required classes. In line 18 and 19 the time and date are set. In line 22 to 26 an endless loop is executed, which outputs the date and time every second. Line 29 is necessary to get the possibility to load Java classes with BSF4ooRexx.

```
16 ::routine setupDs3231 public
17 pkgLocal=.context~package~local  -- get package local directory
18 device = bsf.loadClass("com.pi4j.io.i2c.I2CDevice")
19 i2cbus = bsf.loadClass("com.pi4j.io.i2c.I2CBus")
20 bus = bsf.loadClass("com.pi4j.io.i2c.I2CFactory")~getInstance(i2cbus~BUS_1)
21 pkgLocal~device = bus~getDevice(104)
```

Listing 79: ds3231 routine main program

5.6.6 bh1750.rex

This example shows the use of the BH1750 Ambient Light Sensor. This sensor can measure the illuminance in the unit lux. Again a setup routine ”setupBh1750” is provided. Furthermore there is the routine ”readBH1750” to read out the actual measured illuminance. The I^{2}C address of the bh1750 can be read out via i2cdetect. In these examples it is 0x23 hex or 35 decimal.

5.6.6.1 Circuit

The wiring is the same as in all previous I^{2}C examples. VCC is connected to the power supply of 3.3V, GND is connected to Ground and the two I^{2}C lines are connected to SDA on GPIO pin 8 and SCL on GPIO pin 9.(see Figure 26)

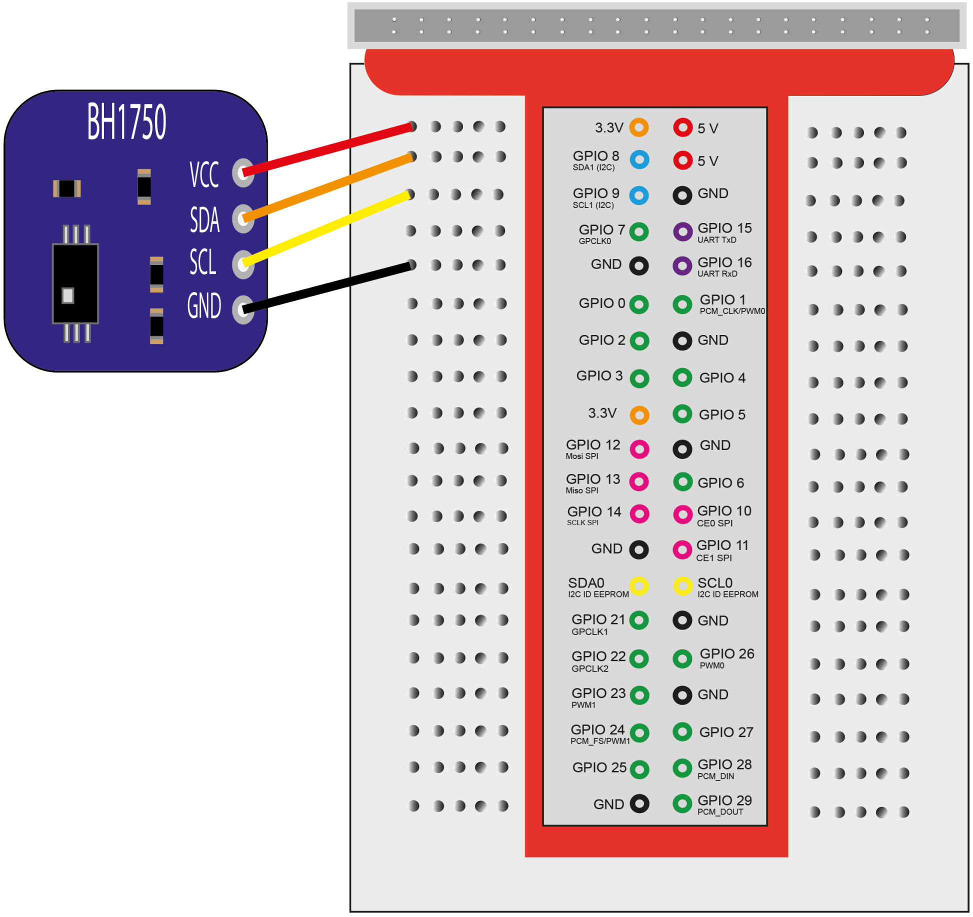


Figure 26: bh1750 Ambient Light Sensor circuit

5.6.6.2 Sourcecode

The routine "setupBh1750" establishes the connection to the required Java classes and makes them available in the local package.(see Listing 80)

```
::routine setupBh1750 public
pkgLocal=.context~package~local  -- get package local directory
device = bsf.loadClass("com.pi4j.io.i2c.I2CDevice")
i2cbus = bsf.loadClass("com.pi4j.io.i2c.I2CBus")
bus = bsf.loadClass("com.pi4j.io.i2c.I2CFactory")~getInstance(i2cbus~BUS_1)
pkgLocal~device = bus~getDevice(35)
return
```

Listing 80: bh1750 routine setupBh1750

The routine "readBh1750"(Listing 81) performs the measurement and returns the measured value. In line 36 the command to activate the chip is stored in the variable "power" and in line 37 the mode is stored in the variable "mode". Mode 20 is used, which means that a single measurement with a resolution of 1 lux is performed. The modes can be read in the data sheet.

In line 40 to 43 the commands are written to the chip and the program is stopped briefly so that the chip has time to process the command. In line 46 a Java byte array with a size of two bytes is created to store the read data in line 47. In line 50 and 51 the read data are processed and prepared according to the data sheet and in line 52 the value is returned.

```
34 ::routine readBh1750 public
35 --values from the data sheet
36 power = 1 --  Power on
37 mode = 20 -- One Time H-Res Mode 1lx Resolution --> Datasheet
38 
39 /* Activate sensor and select mode */
40 .device~write(BsfRawBytes(power~x2c))
41 call syssleep 0.05
42 .device~write(BsfRawBytes(mode~x2c))
43 call syssleep 0.5
44 
45 /* Read out measured data */
46 read = bsf.CreateJavaArray("byte.class",2)
47 .device~read(read,0,2)
48 
49 /* Convert measured values according to data sheet */
50 msb = 0xff(read[1]) * 256
51 lsb= 0xff(read[2])
52 return (msb+lsb)/1.2
```

Listing 81: bh1750 routine readBh1750

The routine "0xFF"(see Listing 82) converts the received signed byte (value range -128 to 127) to an unsigned byte (value range 0 to 255).

```
58 ::routine 0xff
59      use arg v
60      if v < 0 then return v+256
61      return v
```

Listing 82: bh1750 routine 0xFF

Listing 83 shows the main program. In line 15 the "setupBh1750" routine is executed to have access to all required components. After that, in line 16, with the routine "readBH1750", measurement is executed and the value is output. In line 18 the Java support is loaded by BSF4ooRexx.

```
15 call setupBh1750
16 say readBh1750()
17 exit
18 ::requires BSF.CLS  -- get Java support
```

Listing 83: bh1750 main program

5.6.7 bh1750pi4oorexx.rex

The example bh1750pi4oorexx is the same as bh1750.rex only the pi4oorexx library is used for this.

5.6.7.1 Circuit

The wiring has not changed in this example compared to bh1750(see Figure 26)

5.6.7.2 Sourcecode

Listing 84 shows the source code to get access to the bh1750 sensor via pi4oorexx.
In line 12 the required Java class is loaded from the pi4oorexx library via BSF4ooRexx and at the same time a new instance is created with "getInstance(1,35)". "1" represents the number of the I²C bus and "35" represents the address of the chip on the I²C bus. Here it must be noted that by i2cdetect the value is output in hexadeximal notation and here the address is given in decimal. i2cdetect has output the address with 0x20 which corresponds to 35 decimal.
In line 13 the sensor is read out with "getOptical" and the value is output. In line 14 the "help" command is executed to get more information in the console. Line 16 establishes the Java support via BSF4ooRexx.

```
12 call setupBh1750
13 say readBh1750()
14 exit
15 ::requires BSF.CLS  -- get Java support
```

Listing 84: bh1750pi4oorex main program

5.7 HD44780 LC-Display

As the next example the LC display with a HD44780 controller is presented. In this subchapter, the control of the LC display is presented in three different ways. At the beginning the control via the parallel interface is presented. Afterwards the control via the I²C bus is presented. Finally the control with the pi4oorexx library is presented.
This controller is able to control different versions of the display, for example a display with 16 characters per line and two lines. This display is called 1602. There is also a display with 20 characters per line and four lines. This display is then called 2004.

5.7.1 LCDparallel.rex

This example shows the control of the LC display via the parallel interface of the HD-44780 controller. The display offers two modes for the parallel interface. There is an 8-bit mode and a 4-bit mode. In the 8-bit mode eight data lines must be connected between the display and the Raspberry Pi. In addition, 8 further control lines or supply lines must be connected. This would lead to a total of 16 lines. In this example, however, the LC display is controlled in

4-bit mode. In 4-bit mode, only 4 data lines are required, since the character to be displayed is transmitted in two steps. First the upper "half byte" or upper nibble is transmitted and then the lower "half byte" or lower nibble. This leads to a reduction from 16 to 12 lines. There are three public routines which are necessary to output text to the display.

- setup: Builds the connection to the required classes and provides all required GPIO pins in the local package.
- init: initializes the display (according to the datasheet) and sets it to 4-bit mode
- LCDprint: outputs the entered text on the LC display

To show something on the display the methods 1) setup and 2) init have to be executed once at the start of the program this examples based on Java example: `adolf-reichwein-schule.de/bildungsangebote/berufliches-gymnasium/praktische-informatik/newsdetaildvt/news/lcd-display-fuer-den-raspberry-pi/`

5.7.1.1 Circuit

The wiring in 4-bit mode has the advantage of using four data lines less than in 8-bit mode. However, as can be seen in Figure 27, 12 lines still have to be connected.
The VDD pin on the display must be connected to ground. VSS is connected to the 5V power supply. V0 is used to adjust the contrast, so it is connected to SIG from a 10 kiloohm potentiometer. RS is connected to GPIO pin 29. RW is connected to Ground. E is connected to GPIO pin 28.
Data line D4 is connected to GPIO pin 22. D5 is connected to GPIO pin 23. D6 is connected to GPIO pin 24 and D7 is connected to GPIO pin 25. At the end both pins for the backlight have to be connected. A is connected to the 5V power supply and K is connected to ground. For the additional potentiometer VCC must be connected to the 5V power supply and GND must be connected to ground.

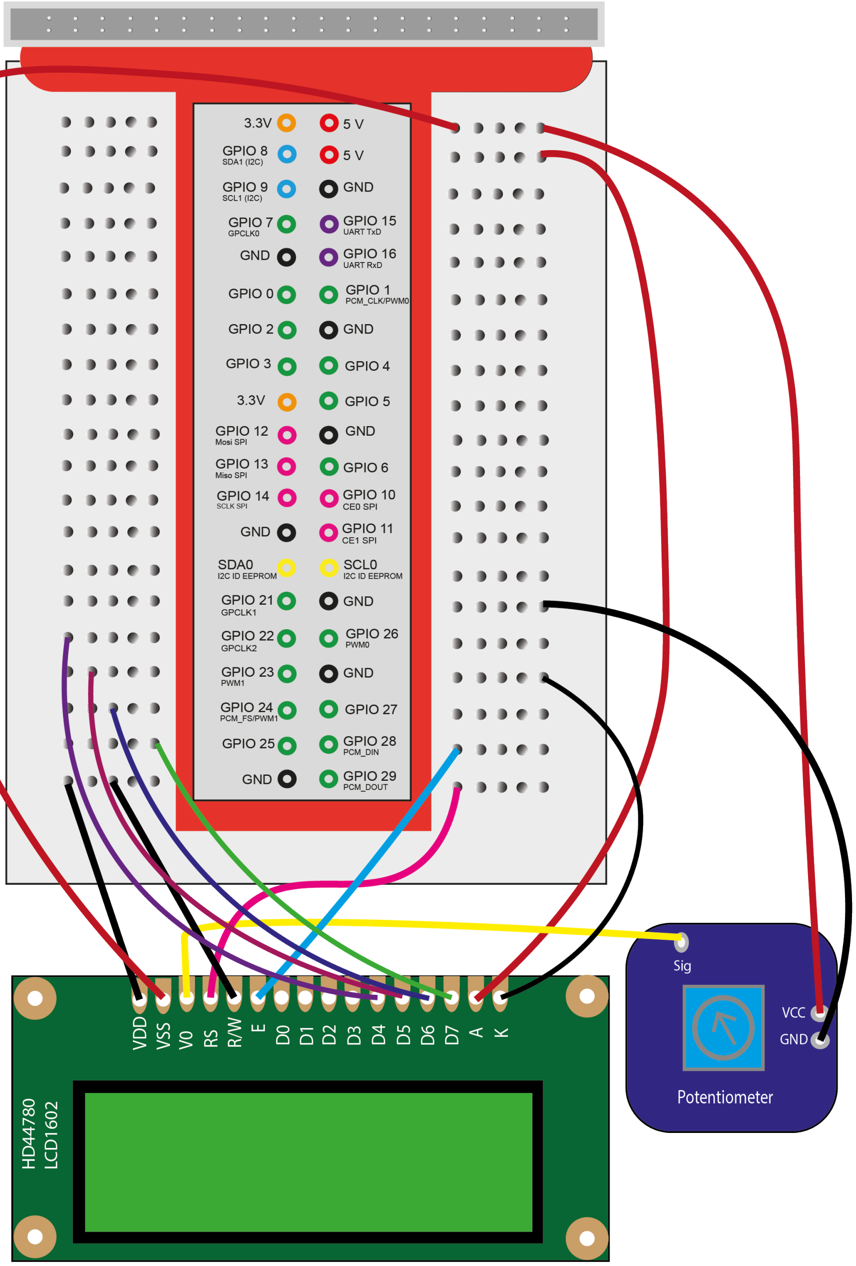


Figure 27: LC-Display parallel Interface 4-Bit Mode circuit

5.7.1.2 Sourcecode

The routine ”setup”(Listing 85) establishes the connection to the GPIO pins and makes them available in the local package (see line 190 - 203). From line 206 on all commands necessary to control the display controller are stored and made available in the local package.

```
188 ::routine setup public
189
```

```
pkgLocal=.context~package~local  -- get package local directory

/*Establish connection to the GPIO pins*/
GpioFactory = bsf.loadClass("com.pi4j.io.gpio.GpioFactory")~getInstance
RaspiPin = bsf.loadClass("com.pi4j.io.gpio.RaspiPin")
pinState = bsf.loadClass("com.pi4j.io.gpio.PinState")

/*raspipin uses the WPI scheme for pinout*/
pkgLocal~rs =GpioFactory~provisionDigitalOutputPin(RaspiPin~GPIO_29,pinState~LOW)
pkgLocal~e =GpioFactory~provisionDigitalOutputPin(RaspiPin~GPIO_28,pinState~LOW)
pkgLocal~d4 =GpioFactory~provisionDigitalOutputPin(RaspiPin~GPIO_22,pinState~LOW)
pkgLocal~d5 =GpioFactory~provisionDigitalOutputPin(RaspiPin~GPIO_23,pinState~LOW)
pkgLocal~d6 =GpioFactory~provisionDigitalOutputPin(RaspiPin~GPIO_24,pinState~LOW)
pkgLocal~d7 =GpioFactory~provisionDigitalOutputPin(RaspiPin~GPIO_25,pinState~LOW)

/* commands for the Lc display */
pkgLocal~LCD_CLEARDISPLAY = "01"x
pkgLocal~LCD_ROW_1 = "80"x
pkgLocal~LCD_ROW_2 = "C0"x
pkgLocal~LCD_ROW_3 = "94"x
pkgLocal~LCD_ROW_4 = "D4"x
pkgLocal~COMMANDREGISTER  = "00"x
pkgLocal~DATAREGISTER     =   "01"x
return
```

Listing 85: LCDparallel setup

The routine ”init”(Listing 86) initializes the display controller and sets it to 4-bit mode. The commands required for this are shown in the data sheet. lcdByte, the routine used in it, is explained in detail in Listing 87.

```
::routine init public

call lcdByte "33"x , .COMMANDREGISTER
call syssleep 0.01
call lcdByte "32"x , .COMMANDREGISTER
call syssleep 0.01
call lcdByte "28"x , .COMMANDREGISTER
call syssleep 0.01
call lcdByte .LCD_CLEARDISPLAY , .COMMANDREGISTER
call syssleep 0.01
call lcdByte "0C"x , .COMMANDREGISTER
call syssleep 0.01
call lcdByte "06"x , .COMMANDREGISTER
call syssleep 0.01
call lcdByte .LCD_CLEARDISPLAY, .COMMANDREGISTER
call syssleep 0.05
return
```

Listing 86: LCDparallel init

The routine ”lcdByte”(Listing 87) is responsible for the output of the character at the display, as well as for the control of the controller.

In line 60 the routine accepts two parameters. First the character to be displayed or the control command (bits) and as second parameter the mode. So if the commandregister or dataregister should be written. This is also queried in line 63. If the command register is to be written, i.e. the command is directed to the display controller, the RS line is set to low and the controller knows that a command follows.

Otherwise, the RS line is set high and the controller knows that data to display follows and it must be written to the data register. In line 71 to 74 all four data lines are set low to ensure the output state of these pins. In line 77 the transmitted character is converted to an 8 bit binary sequence.

In line 80 the preparation of the nibble(half byte) to be transmitted starts. First the upper nibble must be transmitted and then the lower nibble. To determine which GPIO pins must be activated or set high, the binary sequence generated in line 77 is checked for ”0” and ”1”. Since the upper nibble is transmitted first, the first four characters of the binary sequence are considered and the respective GPIO pins are set high if there is a ”1” in the binary sequence.

After the first four characters of the binary sequence have been processed and the GPIO pins have been set high or low, the transmission to the controller follows from line 97. A short program stop is executed in line 97 and 99 so that the program has time to set the pins to the respective state. By switching the E pin on and off (lines 98 and 100) the upper nibble is transferred to the controller.

From line 103 the same follows for the lower nibble. For this the binary sequence from the fifth to the eighth character is checked for ”0” and ”1” and then output.

```
59 ::routine lcdByte
60 use arg bits, mode
61
62 /*Decide whether to write to the command or data register*/
63 if mode = .COMMANDREGISTER then do
64   .rs~LOW
65 end
66 else do
67   .rs~HIGH
68 end
69
70 /*Set data lines to low*/
71 .d4~LOW
72 .d5~LOW
73 .d6~LOW
74 .d7~LOW
75
76 /*Convert the passed character into an 8 bit string*/
77 bits = bits~c2x~x2b
78
79 /*create upper nibble*/
80 do i=1 to 4
81
```

```
if i = 1 then do
    if bits[i] = 1 then .d7~high
  end

else if i= 2 then do
    if bits[i] = 1 then .d6~high
  end
else if i = 3 then do
    if bits[i] = 1 then .d5~high
  end
else
    if bits[i] = 1 then .d4~high
end

/*Write data to the register*/
call syssleep 0.001
.e~high
call syssleep 0.001
.e~low

/*Set data lines to low*/
.d4~LOW
.d5~LOW
.d6~LOW
.d7~LOW

/*create lower nibble*/
do i=5 to 8

if i = 5 then do
    if bits[i] = 1 then .d7~high
  end

else if i= 6 then do
    if bits[i] = 1 then .d6~high
  end
else if i= 7 then do
    if bits[i] = 1 then .d5~high
  end
else
    if bits[i] = 1 then .d4~high
end

/*Write data to the register*/

call syssleep 0.001
.e~high
call syssleep 0.001
.e~low
call syssleep 0.001

```

```
133 return
```

Listing 87: LCDparallel lcdByte

The routine ”LCDprint” (Listing 88) requires two parameters which can be seen in line 172. First, the desired line of the display to be written and the string to be output.
. In line 174 to 177 it checks which line should be selected and sends a command to the display controller to select the line. In line 178 the string is shortened to 16 characters in case it would be longer and in line 179 to 181 each character of the string is written individually into the data register of the display controller to be output.

```
171 ::Routine LCDprint public
172 use arg zeile, String
173 
174 if zeile = 1 then call lcdByte .LCD_ROW_1 , .COMMANDREGISTER
175 else if zeile = 2 then call lcdByte .LCD_ROW_2 , .COMMANDREGISTER
176 else if zeile = 3 then call lcdByte .LCD_ROW_3 , .COMMANDREGISTER
177 else call lcdByte .LCD_ROW_4 , .COMMANDREGISTER
178 string = substr(string,1,16)
179 do j=1 to string~length
180   call lcdByte String[j] , .DATAREGISTER
181 end
182 return
```

Listing 88: LCDparallel LCDprint

In Listing 89 we present a short example which displays the time in line 1 of the display and the current date in line 2. First, however, the routine ”setup” must be called in line 31 and the routine ”init” in line 32 for the output to work. In line 40 the Java support is provided by BSF4ooRexx. This is needed in this example to set the state of the GPIO pins to high or low.

```
31 call setup        --load all required connections
32 call init       --Initialize display
33 
34 say "Clock is running"        --outputs time and date
35 do forever
36   call lcdprint 1 ,TIME()
37   call lcdprint 2 , DATE()
38 end
39 exit
40 ::requires BSF.CLS  --Get Java Support
```

Listing 89: LCDparallel main program

5.7.2 LCDi2c.rex

In the first example of the LC display, the connection via parallel interface was demonstrated. In this example the connection of the display via the I²C interface is presented. For this a special version of the PCF8574 I/O port expander from chapter 5.6.3 is used. This is specially

modified for the control of LC-Displays. It still has only 8 outputs but a potentiometer was added so that the contrast can be adjusted without an additional potentiometer.
This version has the big advantage that the complicated wiring is omitted. The control is again in 4-bit mode.
The connection to the I²C bus is established via Pi4J and BSF4ooRexx The I²C address of the display can be read out via i2cdetect. In these examples it is 0x27 hex or 39 decimal.

5.7.2.1 Circuit

As known from the other I²C examples, the wiring is done with only 4 lines. The LC-Display needs in contrast to the other I²C examples a power supply of 5V, therefore the VCC pin is also connected to it. The GND pin is connected to Ground, SDA to GPIO pin 8 and SCL to GPIO pin 9. By using the I²C bus eight lines are saved compared to the 4-bit mode and 12 lines compared to the 8-bit mode.(see Figure 28)

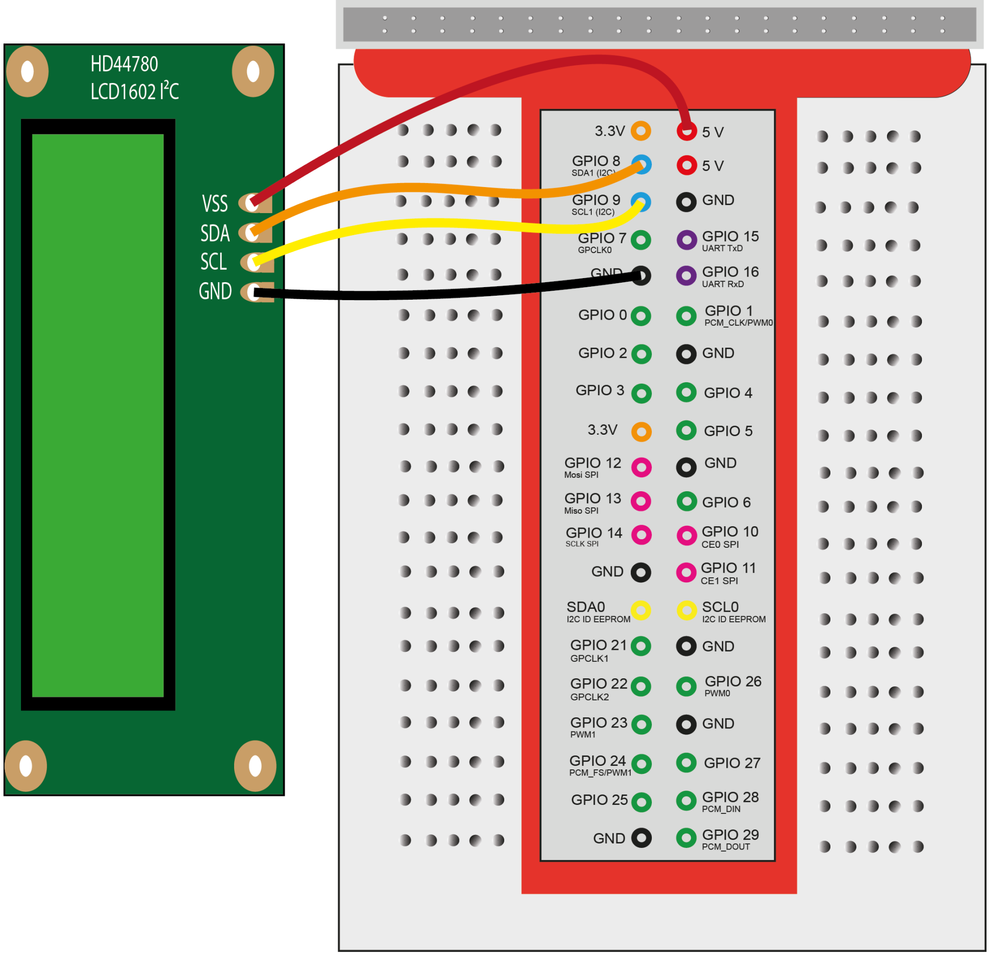


Figure 28: LC-Display I²C 4-Bit Mode circuit

The routine ”setup” establishes the connection to the required I²C bus via BSF4ooRexx and the Pi4J library and makes the required class available in the local package so that it can be accessed program-wide.(see Listing 90)

```
::routine setup public
 pkgLocal=.context~package~local  -- get package local directory
/*load required classes*/
  i2cbus = bsf.loadClass("com.pi4j.io.i2c.I2CBus")
/*get Instance of I2CFactory on I2C Bus 1 */
  bus = bsf.loadClass("com.pi4j.io.i2c.I2CFactory")~getInstance(i2cbus~BUS_1)
/*get Device on I2C Address 0x27 -> 39 decimal*/
  pkgLocal~device = bus~getDevice(39)
return
```

Listing 90: LCDi2c routine setup

5.7.2.2 Sourcecode

The routine ”toByte” (see Listing 91) is used to convert a passed value(line 163) into the form of a Java byte. A Java byte has a value range from -128 to +127. This is achieved with the code in lines 165-168.

```
155 ::routine toByte
156 use arg value
157
158 if value > 127 then do
159     value = value-256
160     return value
161   end
162 return value
```

Listing 91: LCDi2c routine toByte

The routine ”write” (Listing 92) is responsible for sending the commands to the display driver, via I^{2}C bus.
In line 118 the routine gets two values. The byte to be output in binary representation and the desired mode, i.e. whether it is intended for the instruction register or the data register. In line 120 the passed value of the mode is checked, if the mode is ”1”, the passed byte is intended for the data register, if it is ”0”, it is intended for the command register. For this the corresponding commands are stored either in line 121,122 or in line 125,126 in the variables ”mode” and ”mode_”.
The variable ”mode” or ”mode_” are there to send the control commands to the display. So that it knows whether it is either in command or data mode, and whether the values on the data lines are to be read in or not.

This is structured as follows (see Table 6) ,
Bit 1 : Register Select (RS) is responsible for the controller to know if it is a command or data to be output. In this example it is ”1” therefore the data to be transmitted is data to be output and not in the control data.
Bit 2: Read/Write(RW) is used to tell the controller whether to read or write data from the

register selected in bit 1. In this example the "0" is transmitted, that means it is written into the register. Bit 3: With Enable(E) the controller is given the command to read the data bits on the data lines. In this example Enable is "1" the controller reads the data from the data lines. However, so that the controller knows that the transmission is finished, E must be set to "0" again. Therefore there are also the two variables "mode" and "mode_" which differ only in bit 3.
Bit 4 : is responsible for the activation or deactivation of the backlight of the display. The transmission of "1" means that the backlight is switched on.

Backlight	E	RW	RS
1	1	0	1
Bit 4	Bit 3	Bit 2	Bit 1

Table 6: LCDi2c Explanation of the mode variable

After that, starting from line 131, the data from the variable "Byte" is prepared for transmission to the display.
As in the previous LCD example, the data is transferred in 4-bit mode. This means that the values from the variable "Byte" (example see Table 7) must be divided into an upper and lower nibble. First the upper nibble is prepared for the transmission (line 131). For this the bit 5 to bit 8 of the variable "Byte" are taken and stored in the variable "upper_Nibble".
In line 132 the byte to be transmitted is generated. For this the upper nibble(4 bit length) is combined with the "mode"(4 bit length) variable to a byte(8 bit length) and converted into a decimal number. With the routine "toByte" this decimal number is adapted in such a way that it corresponds to a Java byte.
In line 133 the same is done only with the difference that now not the variable "mode" but "mode_" is used. The reason for this is that the Enable(E) input at the controller is switched on and off for a short time and thus reads the data.
After that the same process is repeated for the lower nibble.

Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
0	1	0	0	1	1	0	1
upper Nibble				lower Nibble			

Table 7: LCDi2c byte example letter "M"

After that, in line 143 to 147, a Java byte array is created by BSF4ooRexx and all four commands are put into this array. In line 150 the commands are written to the controller one after the other.
As an example, Table 8 lists all four commands necessary to output the letter "M" on the display.

register	Nibble				Backlight	Enable	RW	RS
Bits	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
enable 1	0	1	0	0	1	1	0	1
Command 1	upper Nibble				control data			
enable 0	0	1	0	0	1	0	0	1
Command 2	upper Nibble				control data			
enable 1	1	1	0	1	1	1	0	1
Command 3	lower Nibble				control data			
enable 0	1	1	0	1	1	0	0	1
Command 4	lower Nibble				control data			

Table 8: LCDi2c Example print ”M” at Display

After these four commands have been transmitted to the display controller, the letter ”M” is shown on the display.

```
::routine write
use arg byte, m

if m = 1 then do         --Datamode
  mode  = 1101
  mode_   = 1001
end
else do              --Commandmode
  mode  = 1100
  mode_ = 1000
end

-- create upper Nibble

upper_Nibble = substr(byte,1,4)
by_un_en  =   toByte((upper_Nibble||mode)~b2x~x2d)
by_un_en_   = toByte((upper_nibble||mode_)~b2x~x2d)

-- create lower Nibble
lower_Nibble = substr(byte,5,4)

by_ln_en  =   toByte((lower_Nibble||mode)~b2x~x2d)
by_ln_en_   = toByte((lower_nibble||mode_)~b2x~x2d)

--put the bytes into a Java Byte array

out=bsf.createJavaArray("byte.class", 4)
out~put(by_un_en,1)
out~put(by_un_en_,2)
out~put(by_ln_en,3)
out~put(by_ln_en_,4)

/*write byte by byte to the controller*/
```

```
143 do i= 1 to 4
144   .device~write(out[i])
145 end
146 return
```

Listing 92: LCDi2c routine write

The routine "printline"(Listing 93) is used to output text to the display. For this two parameters are needed (line 92), these are the string to be printed and the desired line to be written to. In line 93 the mode is set to "1" so that the data mode is selected. In line 95 the string is shortened to a maximum length of 16 characters. In line 99 and line 100 it is checked which line was selected. In line 105 the string is split into its individual characters, then each individual character is converted into a binary sequence and passed to the routine "write", so that it outputs the character on the display.

```
84  ::routine printline public
85  use arg string, line
86  mode = 1             -- mode 1 = Datamode
87
88  if string~length > 16 then string = substr(string,1,16)
89  /*                     -- for a 2004 Display
90  if string~length > 20 then string = substr(string,1,20)
91  */
92  if line = 1 then call printcmd "80"x
93  if line = 2 then call printcmd "C0"x
94  /*                     --for a 2004 Display
95  if line = 3 then call printcmd "94"x
96  if line = 4 then call printcmd "D4"x
97  */
98  do i=1 to string~length
99    call write String[i]~c2x~x2b, mode
100 end
101
102 return
```

Listing 93: LCDi2c routine printline

The routine "printcmd"(Listing 94) is used to send commands to the display controller. To do this, mode "0" is selected in line 82 and the command passed is sent to the controller via the "write" routine in line 83.

```
73 :routine printcmd
74 use arg cmd
75 mode = 0       -- Mode 0 = CMD mode
76 call write cmd~c2x~x2b , mode
77 return
```

Listing 94: LCDi2c routine printcmd

The routine "init" (Listing 95) is necessary to put the display controller into the desired mode.

```
::routine init public
--4 Bit Mode
call printcmd "33"x
--4 Bit Mode
call printcmd "32"x
-- 4Bit 2 Line
call printcmd "28"x
--Turn display on
call printcmd "0C"x
-- move cursor right
call printcmd "06"x
return
```

Listing 95: LCDi2c routine init

The routine ”clear” (Listing 96) can be used to clear the display content.

```
::routine clear public

call printcmd "01"x
return
```

Listing 96: LCDi2c routine clear

Listing 97 shows an example program, which first executes the routine ”setup” in line 24 to get access to the desired Java classes. Afterwards the display is initialized with the routine ”init” and in line 28 the output on the display is deleted.

In line 31 ”Hello World” is output in row one, then one second is waited and in line 33 ”from ooRexx” is output in row two. After that the content stays on the display for five seconds and is cleared with ”clear”. Line 39 establishes the Java support via BSF4ooRexx.

```
24 /*Make i2c device available*/
25 call setup
26 /*initialize the device*/
27 call init
28 /*clear the display*/
29 call clear
30
31 /* Sample Output*/
32 call println "Hello World" , 1
33 call syssleep 1
34 call println "from ooRexx" , 2
35 call syssleep 5
36 call clear
37
38 exit
39
40 ::Requires BSF.CLS    --get Java Support
```

Listing 97: LCDi2c example main programm

5.7.3 LCDpi4oorexx.rex

As a third example, the control of the LC display using the pi4oorexx library is shown. This is done again via the I^2C bus.

5.7.3.1 Circuit

The wiring hast not changed in this example compared to LCDi2c (see Figure 28)

5.7.3.2 Sourcecode

Listing 98 shows an example program how to control a LC display using the pi4oorexx library.
First all needed classes are loaded in line 13 - 16. In line 18 the connection to the device is established and in line 20 a new object of the pi4oorexx class is created.
After that the display can already be controlled. In line 22 the "Help" routine is called, which provides information about the pi4oorexx class and also about the wiring. In line 24 the display content is cleared so that in line 26 and 27 text can be printed on it by the "display_string" routine. After that the content is cleared again after three seconds and text is printed again, but this time with "dispay_string_pos". With this routine the position of the first character can be specified. After that the output is cleared again and at the end the routine "backlight" is demonstrated, which switches the backlight off and on again.
In line 43 the Java support is loaded by BSF4ooRexx.

```
12 /* load all required classes*/
13 lcd = bsf.import("at.pi4oorexx.lcd.I2CLCD")
14 i2cbus = bsf.loadClass("com.pi4j.io.i2c.I2CBus")
15 /*get Instance of I2C Factory: I2C Bus 1   */
16 bus = bsf.loadClass("com.pi4j.io.i2c.I2CFactory")~getInstance(i2cbus~BUS_1)
17 /*get Device on I2C Address 0x27 -> 39 int*/
18 device = bus~getDevice(39)
19 /*build new object*/
20 screen =lcd~new(device)
21 /*use the help function to learn more about the available routines*/
22 screen~help
23 /*clear screen*/
24 screen~clear
25 /*display string on desired line*/
26 screen~display_string("Hello World",1)
27 screen~display_string(" from ooRexx",2)
28
29 call syssleep 3
30 screen~clear
31
32 /*display string on desired line and desired positon*/
33 screen~display_string_pos("Hello from",1,2)
34 screen~display_string_pos("ooRexx",2,3)
35 call syssleep 3
36 screen~clear
```

```
call syssleep 1
/*Backlight off-on*/
screen~backlight(0)
call syssleep 0.5
screen~backlight(1)

::requires BSF.CLS  --get Java Support
```

Listing 98: LCDpi4oorexx main programm

5.8 RFIDAttendance.rex

This example shows a use case that connects several components. As an example, the creation of a simple RFID attendance system was chosen. This includes an MFRF522 RFID reader, an LC display, two push buttons and a buzzer. This makes it possible to record the presence of employees in a file.
The operation starts by pressing the "come" or "go" button, which triggers the RFID reader, which is then ready for reading for about five seconds. If the RFID card is held up to the reader until then, it is read and, if successful, the data record with time and date is written to a file. The status can be read on the LC display. Successful reading is also acknowledged with a "beep". If the reading process is not successful, this is also indicated on the LC display and acknowledged with three "beeps". For the use of this example a file with employee data "emp.dat" must be created which contains the ID of the RFID card and its name. Other data is also possible.
This example uses the Pi4J and the pi4oorexx library. These are loaded by BSF4ooRexx.

5.8.0.1 Circuit

First the LC-Diplay is described, because the wiring of this is connected the same way as all other I^{2}C devices. VCC is connected to the 3.3V power supply, GND is connected to Ground. SDA is connected to GPIO pin 8 and SCL is connected to GPIO pin 9. After that the RFID reader is connected, it is connected via SPI. VCC is connected to the 3.3V power supply. MOSI is connected to GPIO pin 12. MISO is connected to GPIO pin 13. SCLK is connected to GPIO pin 14. GND is connected to Ground. CE0 is connected to GPIO pin 10 and RST is connected to GPIO pin 6. IRQ is not needed in this example therefore it is not connected. the "come" and "go" buttons are connected as follows. Pin 1 is connected to the 3.3V power supply. Pin 2, the "go" button, is connected to GPIO pin 24. Pin 3, the "come" button, is connected to GPIO pin 5.
The buzzer contact SIG is connected to GPIO pin 1 and GND to Ground. (see Figure 29)

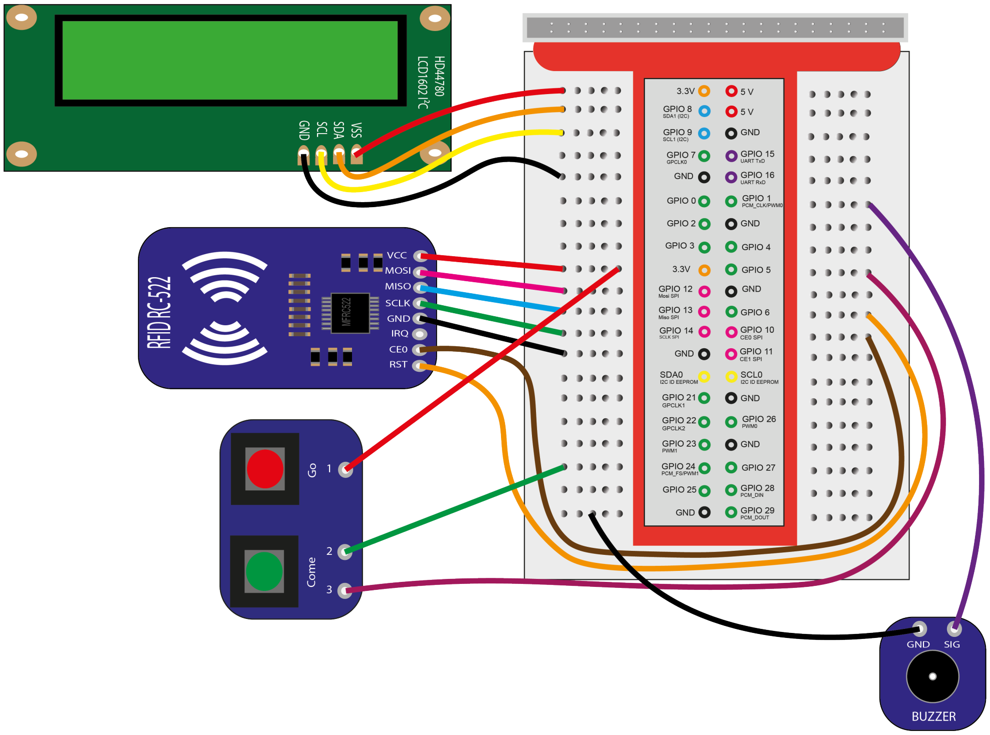


Figure 29: RFID Attendance System circuit

5.8.0.2 Sourcecode

The ”setup” routine(Listing 99) is used to load all required classes and make them available in the local package. Line 114 creates the package. In line 116-118 the RFID reader is initialized. In line 121-127 the two inputs (buttons) and the output (buzzer) are initialized. in line 131-136 the LC-Display is initialized.

```
113 ::routine setup
114 pkgLocal=.context~package~local  -- get package local directory
115 --- initialzie MFRC522
116 rc522clientimpl = bsf.loadClass("at.pi4oorexx.mfrc522.rc522.RC522ClientImpl")
117 card = bsf.loadClass("at.pi4oorexx.mfrc522.model.card.Card")
118 pkgLocal~rc522client = rc522clientimpl~createInstance
119 --- initialize Button and Buzzer
120 
121 gpio = bsf.loadClass("com.pi4j.io.gpio.GpioFactory")~getInstance
122 RaspiPin = bsf.loadClass("com.pi4j.io.gpio.RaspiPin")
123 PinPullDown = bsf.loadClass("com.pi4j.io.gpio.PinPullResistance")~PULL_DOWN
124 pkgLocal~pinCome = gpio~provisionDigitalInputPin(RaspiPin~GPIO_05,PinPullDown)
125 pkgLocal~pinGo = gpio~provisionDigitalInputPin(RaspiPin~GPIO_24,PinPullDown)
126 pinstate = bsf.loadClass("com.pi4j.io.gpio.PinState")
127 pkgLocal~pinBuzzer = gpio~provisionDigitalOutputPin(RaspiPin~GPIO_01,pinstate~high)
128 
129 --- initialize LC- Display
```

```

device = bsf.loadClass("com.pi4j.io.i2c.I2CDevice")
lcd = bsf.import("at.pi4oorexx.lcd.I2CLCD")
i2cbus = bsf.loadClass("com.pi4j.io.i2c.I2CBus")
bus = bsf.loadClass("com.pi4j.io.i2c.I2CFactory")~getInstance(i2cbus~BUS_1)
device = bus~getDevice(box('int',39)) --0x27 = 39   hex -> int
pkgLocal~screen =lcd~new(device)
return
```

Listing 99: RFIDAttendance routine setup

The ”readwrite” routine(Listing 100) is used to write the data read by RFID to a file. In line 23 the paramenter is passed or the person comes or goes. this is necessary so that the correct entry is written to the file.
In line 25 the auxiliary variable ”nocard” is created and assigned to the value ”0”. this is needed so that the LC-Display only displays the output ”Hold card to reader”.
After that, a loop is created in line 27, which generates a total of 10 read attempts. After that in line 28 the value of the RFID card is read.
In line 30 again an auxiliary variable ”readok” is created, which is set to ”1” when the card is read successfully.
In line 31 it is checked if data was read If no data was read, the display shows that the card should be held to the reader and the ”nocard” variable created before in line 25 is set to ”1” so that the output is not updated at each run. If data are read, the code from line 41 is executed. This causes the ID of the card to be read. After that in line 44 the file with the employees is read by stream object. In line 47 a loop starts, which compares all available employees or their ID from the ”emp.dat” file with the read ID. in line 49 the read-in data from the ”emp.dat” file are split into individual variables. e.g. ID. If a matching ID is found, its name with time and date (line 55 or 62) is written into the ”attendance.csv” file. If the value of the variable cg is ”come”, then ”come” is also written to the file, otherwise ”go”. As well as ”card detected” and either ”Hi” or ”bye” + name is output. Depending on the ”cg” variable. Additionally there is an audible confirmation that the data has been read successfully. and the loop counter is incremented(parts 10) to end the loop.additionally the auxiliary variable ”readok” is set. If no data has been read after ten read attempts and readok is ”0”, ”no card detected” is output on the display from line 87 and this is acknowledged with three ”beeps”.

```
22 ::routine readwrite
23 use arg cg
24
25 nocard = 0  --so that display is not constantly updated when no card is present
26
27       do i=1 to 10  -- 10 reading attempts
28             carddata= .rc522client~readCardData
29         call syssleep(0.1)
30         readok=0
31         if carddata == .nil then  do
32
33           say "no card yet"
```

```
          if nocard ==0 then do
              .screen~clear
            .screen~display_string_pos("Hold card",1,0)
            .screen~display_string_pos("to reader",2,0)
            nocard = 1
          end
      end
        else do
        tagID = carddata~getTagIdAsString
        say "card detected" tagID
        emp = .stream~new("emp.dat")
        emp~open("read")

            do while emp~lines<> 0
                    data = emp~linein
            parse var data tag "," name "," perso
                    if tag = tagID then do
               --say tagid passt
               file =.stream~new("attendance.csv")
               file~open("write")
               if cg = "come" then do
               file~lineout(date() ", "Time() "," name "," COME)
                say come
                    .screen~clear
                            .screen~display_string_pos("card detected" ,1,0)
                            .screen~display_string_pos(("Hi" name) ,2,0)
               end
               else do
               file~lineout(date() ", "Time() "," name "," GO)
                   say  go
                    .screen~clear
                            .screen~display_string_pos("card detected" ,1,0)
                            .screen~display_string_pos("Bye" name,2,0)
               end
               file~close
               call beep 1
                       call syssleep (2)
                        i=10  --> end loop
                        readOK = 1
            end
                end
      end

    call syssleep(0.2)

    if (i==10 & readok == 0)  then do
      .screen~clear
      .screen~display_string_pos("no card",1,0)
      .screen~display_string_pos("detected!!!",2,0)
            call syssleep(1)
            call beep 3
```

```
 85     end
 86
 87   end
 88   .screen~clear
 89
 90 return
```

Listing 100: RFIDAttendance routine readwrite

The "beep" routine(Listing 101) is used to generate an acoustic signal with the buzzer.

```
100 ::routine beep
101 use arg n
102
103 do i = 1 to n
104 .pinBuzzer~low
105 call Syssleep(0.05)
106 .pinBuzzer~high
107 call Syssleep(0.05)
108 end
109 return
```

Listing 101: RFIDAttendance routine beep

The main program (see listing 102) first calls the "setup" routine and clears the display contents. After that an endless loop is started which waits for button inputs. If the "come" button is pressed the routine "readwrite" is called with the parameter "come", with the "go" button the same happens only with the parameter "go". The current time is also shown on the display. In line 19 BSF4ooRexx is loaded, which provides Java support.

```
100 call setup
101 call syssleep 0.2
102 .screen~clear
103 call syssleep 0.2
104 say "start"
105 do forever
106 .screen~display_string_pos("TIME"(),1,3)
107
108     if .pinCome~getState~toString =="HIGH" then call readwrite "come" -- come button
109   if .pinGo~getState~toString =="HIGH" then call readwrite "go"     -- go button
110     call syssleep(0.2)
111 end
112
113 exit
114
115 ::requires BSF.CLS -- get Java Support
```

Listing 102: RFIDAttendance main program

6 Conclusio

The conclusion of this bachelor thesis is a conclusion and an outlook from the author's point of view. In this thesis the programming with ooRexx on a Raspberry Pi was presented. The author described the necessary software, showed how to install it and presented examples he developed himself. The author has thus proven that it is possible to program sensors connected to a Raspberry Pi using ooRexx and BSF4ooRexx, which clearly answers the research question. As already explained, BSF4ooRexx provides a simple way to use Java classes without knowing complicated terms. The programming is simplified clearly and can be used so by a broader mass. Especially on the comparatively inexpensive Raspberry Pi many programmers, or just interested in it, have the opportunity to develop their own programs. ooRexx therefore opens a door to develop simple programs with relatively little know-how. The more one deals with programming in general and ooRexx in particular, the more complex programs can be developed.

In the future, from the author's point of view, work will definitely continue with ooRexx, since its ease of use is an enormous advantage over programming in Java. The author is of the opinion that ooRexx can be further developed in the future in such a way that access to Java classes will no longer be necessary. One will program directly with ooRexx and use own, just as easily accessible, classes.

7 Appendix

7.1 Source Codes

The source codes created for the work can be found at the following link `https://github.com/pi4oorexx`

7.2 LaTex

The author used LaTex software package to create this work.
The LaTex package ”Listings” was used to display the source code. This LaTex package has a syntax highlighting for Rexx intigrated. It does not handle the syntax of ooRexx, but you can insert the needed commands by yourself. Listing 103 shows where you can add your own keywords. In line 29 at morekeywords you can add your own keywords. Here, for example, among other things ”myOwnKeyword” was added.The code in line 51 is also very important, otherwise the code from the listings cannot be copied correctly formatted.

```
 1 \lstset{
 2   % normal tilde symbol
 3   literate={~} {$\sim$}{1},
 4   %background color; you must add \usepackage{xcolor}; should come as last argument
 5   backgroundcolor=\color{darkgray},
 6   % the size of the fonts that are used for the code and textcolor
 7   basicstyle=\ttfamily\scriptsize\color{white}
 8   % sets if automatic breaks should only happen at whitespace
 9   breakatwhitespace=false,
10   % sets automatic line breaking
11   breaklines=true,
12    % sets the caption-position to bottom
13   captionpos=b,
14   % comment style
15   commentstyle=\color{red},
16   % if you want to delete keywords from the given language
17   deletekeywords={...},
18   % lets you use non-ASCII characters; for 8-bits encodings only, doesnt work with UTF-8
19   extendedchars=true,
20   % keeps spaces in text, useful for keeping indentation of code
21   keepspaces=true,
22   % keyword style
23   keywordstyle=\color{myorange},
24   % the language of the code
25   language=OORexx,
26   % if you want to add more keywords to the set
27   morekeywords={::requires,::routine,public, myOwnKeyword},
28   % where to put the line-numbers; possible values are (none, left, right)
29   numbers=left,
30    % how far the line-numbers are from the code
31   numbersep=5pt,
32    % the style that is used for the line-numbers
```

```
  numberstyle=\tiny\color{mygray},
  % if not set, the frame-color may be changed on line-breaks within not-black text
  rulecolor=\color{red},
  % show spaces everywhere adding particular underscores; it overrides 'showstringspaces'
  showspaces=false,
  % underline spaces within strings only
  showstringspaces=false,
   % show tabs within strings adding particular underscores
  showtabs=false,
   % the step between two line-numbers. If it's 1, each line will be numbered
  stepnumber=1,
   % string literal style
  stringstyle=\color{green},
   % sets default tabsize to 2 spaces
  tabsize=2,
  % copy text enable
  columns=fullflexible,
  % show the filename of files included with \lstinputlisting; also try caption
  title=\lstname
}
```

Listing 103: lstListings Configuration

Listing 104 shows a short ooRexx program that demonstrates how to display custom keywords. Attention, this program is not executable because it only serves to demonstrate the display of the keyword.

```
do i= 1 to 5
  myOwnKeyword i
end
```

Listing 104: myOwnKeyword

References

[ABa90] Michael A.Banks. *BITS, BAUD RATE, AND BPS , Taking the Mystery Out of Modem Speeds*. 1990. URL: `http://www.textfiles.com/apple/bitsbaud.txt` (visited on 02/14/2022).

[Ard] Arduino. *Serial*. URL: `https://www.arduino.cc/en/reference/serial&gt` (visited on 02/12/2022).

[Ass] Open Source Hardware Association. *A Resolution to Redefine SPI Signal Names*. URL: `https://www.oshwa.org/a-resolution-to-redefine-spi-signal-names/` (visited on 02/04/2022).

[Ass15] Rexx Language Association. *About Open Object Rexx*. 2015. URL: `https://www.oorexx.org/about.html` (visited on 02/04/2022).

[Ass21] Rexx Language Association. *What is Rexx?* 2021. URL: `https://www.rexxla.org/rexxlang/` (visited on 02/07/2022).

[bit19] bitreporter.de. *Alle Raspberry Pi Modelle: Übersicht + Bilder*. Nov. 2019. URL: `https://bitreporter.de/raspberrypi/raspberry-pi-geschichte-modelle-und-bauformen/` (visited on 02/14/2022).

[Cow90] Michael Cowlishaw. *The REXX Language,A Practical Approach to Programming*. 2nd ed. Prentice-Hall, Inc, 1990.

[del] delftstack.com. *Java Classpath*. URL: `https://www.delftstack.com/de/howto/java/java-classpath-/` (visited on 02/13/2022).

[Del20] Frank Delporte. *Getting started with Java on the Raspberry Pi,A lot of small and bigger examples to introduce you to Java(11+), JavaFX (11+), Pi4J, Spring, Queues... with hardwareprojects on the Raspberry Pi*. Leanpub, 2020.

[Del21] John Delvare. *I^2C Tools for Linux*. 2021. URL: `https://i2c.wiki.kernel.org/index.php/I2C_Tools` (visited on 02/12/2022).

[Dem19] Klaus Dembowski. *Raspberry Pi – Dastechnische Handbuch.Konfiguration, Hardware,Applikationsentwicklung*. 3rd ed. Springer Vieweg, 2019.

[Dyc21] Tony Dycks. *Stable RPM Based Linux Distros for theRaspberry Pi 4*. Nov. 2021. URL: `https://www.rexxla.org/presentations/2021/RexxLA2021-StableRPMLinuxDistro-RPi4-TDycks.pdf` (visited on 02/13/2022).

[Elea] Elektronik-Kompendium.de. *Raspberry Pi: Belegung GPIO*. URL: `https://www.elektronik-kompendium.de/sites/raspberry-pi/1907101.htm` (visited on 02/14/2022).

[Eleb] Elektronik-Kompendium.de. *Raspberry Pi: GPIO mit Pullup- oder Pulldown-Widerstand beschalten?* URL: `https://www.elektronik-kompendium.de/sites/raspberry-pi/2006051.htm` (visited on 02/14/2022).

[Fla12] Rony G. Flatscher. “Automatisierung mit ooRexx und BSF4ooRexx”. In: *Proceedings der GMDS 2012 / Informatik 2012* (2012), pp. 1–12.

[Fla13] Rony G. Flatscher. *Introduction to Rexx and ooRexx, From Rexx to Open Object Rexx (ooRexx).* 1st ed. Facultas Verlags- und Buchhandels AG. Wien, 2013.

[fre21] freenove.com. *Freenove Ultimate Starter Kit for Raspberry PI.* 2021. URL: `https://github.com/Freenove/Freenove_Ultimate_Starter_Kit_for_Raspberry_Pi/blob/master/Tutorial.pdf` (visited on 02/11/2022).

[Gay18] Warren Gay. *Advanced Raspberry Pi: Raspbian Linux and GPIO Integration.* 2nd ed. Apress Media LLC, 2018.

[Hen] Gordon Henderson. *Wiring Pi, Software PWM Library.* URL: `http://wiringpi.com/` (visited on 02/12/2022).

[Hen22] Gordon Henderson. *Wiring Pi,GPIO Interface library for the Raspberry Pi.* 2022. URL: `http://wiringpi.com/reference/software-pwm-library/` (visited on 02/12/2022).

[Hor13] Brendan Horan. *Practical Raspberry Pi.* Apress, 2013.

[Hus21] Rebecca Husemann. *Die besten Breakout-Boards für Maker.* Feb. 2021. URL: `https://www.heise.de/news/Die-besten-Breakout-Boards-fuer-Maker-5051937.html` (visited on 02/14/2022).

[Ike18] Naoki Ikeguchi. *BME280.java.* Aug. 2018. URL: `https://github.com/siketyan/TempRa/blob/master/src/main/java/io/github/siketyan/monitor/util/BME280.java` (visited on 02/15/2022).

[int21] maxim integrated. *MAX7219 Datasheet.* 2021. URL: `https://datasheets.maximintegrated.com/en/ds/MAX7219-MAX7221.pdf` (visited on 02/06/2022).

[Jar+a] Aurelien Jarno et al. *i2cdetect(8) - Linux man page.* URL: `https://linux.die.net/man/8/i2cdetect` (visited on 02/15/2022).

[Jar+b] Aurelien Jarno et al. *i2cget(8) - Linux man page.* URL: `https://linux.die.net/man/8/i2cget` (visited on 02/15/2022).

[Jar+c] Aurelien Jarno et al. *i2cset(8) - Linux man page.* URL: `https://linux.die.net/man/8/i2cset` (visited on 02/15/2022).

[Java] Java. *What is Java?* URL: `https://www.java.com/en/download/help/whatis_java.html` (visited on 02/08/2022).

[Javb] JavaTpoint. *History of Java.* URL: `https://www.javatpoint.com/history-of-java` (visited on 02/08/2022).

[JLe21] Matthew J.Lewis. *Diozero.* 2021. URL: `https://github.com/mattjlewis/diozero` (visited on 02/15/2022).

[Lab] Silicon Labs. *AN0059.0: UART Flow Control.* URL: `https://www.silabs.com/documents/public/application-notes/an0059.0-uart-flow-control.pdf` (visited on 02/14/2022).

[Lon21] Simon Long. *Bullseye – the new version of Raspberry Pi OS*. 2021. URL: https://www.raspberrypi.com/news/raspberry-pi-os-debian-bullseye/ (visited on 02/12/2022).

[Mat21] MathWorks. *The Raspberry Pi PWM*. 2021. URL: https://de.mathworks.com/help/supportpkg/raspberrypiio/ug/the-raspberry-pi-pwm.html (visited on 02/14/2022).

[Mic20] Christoph Scherbeck Michael Kofler Charly Kühnast. *Raspberry Pi, Das umfassende Handbuch*. 6th ed. Rheinwerk Verlag GmbH, 2020.

[Mika] Mikrocontoller.net. *I^2C*. URL: https://www.mikrocontroller.net/articles/I%C2%B2C (visited on 02/12/2022).

[Mikb] Mikrocontroller.net. *SPI Daisychain*. URL: https://www.mikrocontroller.net/articles/SPI_Daisychain (visited on 02/12/2022).

[mik] mikrocontroller.net. *AVR-Tutorial: UART*. URL: https://www.mikrocontroller.net/articles/AVR-Tutorial:_UART (visited on 02/12/2022).

[Moh17] Martin Mohr. "I2C-Reichweite steigern mit dem P82B715". In: *Raspberry Pi Geek* (Feb. 2017).

[Mol16] Derek Molloy. *Exploring Raspberry Pi, Interfacing to the Real World with Embedded Linux*. John Wiley and Sons, Inc., 2016.

[Mon16] Simon Monk. *Raspberry Pi Cookbook*. 2nd ed. O'Reilly Media Inc., 2016.

[Nov21] Novatel. *GPRMC, GPS specific information*. 2021. URL: https://docs.novatel.com/OEM7/Content/Logs/GPRMC.htm (visited on 02/15/2022).

[Oraa] Oracle. *Introduction to Java*. URL: https://www.oracle.com/java/technologies/introduction-to-java.html (visited on 02/08/2022).

[Orab] Oracle. *Java Platform Overview*. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/index.html (visited on 02/08/2022).

[Pi4] Pi4J. *About Pi4J*. URL: https://pi4j.com/about/ (visited on 02/12/2022).

[pim21] pimylifeup. *The Different Versions of the Raspberry Pi*. 2021. URL: https://pimylifeup.com/raspberry-pi-versions/ (visited on 02/10/2022).

[Pla] Prof. Jürgen Plate. *Raspberry Pi: SPI-Schnittstelle, Grundlagen*. URL: http://www.netzmafia.de/skripten/hardware/RasPi/RasPi_SPI.html (visited on 02/14/2022).

[pro] programming.guide. *Java: Byte (class) vs byte (primitive)*. URL: https://programming.guide/java/byte-vs-byte.html (visited on 02/16/2022).

[Ras] Pi Foundation Raspberry. *About Us*. URL: https://www.raspberrypi.org/about/ (visited on 02/10/2022).

[Rat21] Sushil Rathore. *I2C Utilities in Linux*. Oct. 2021. URL: https://linuxhint.com/i2c-linux-utilities/ (visited on 02/15/2022).

[Ray22] Eric S. Raymond. *NMEA Revealed*. Feb. 2022. URL: https://gpsd.gitlab.io/gpsd/NMEA.html#_rmc_recommended_minimum_navigation_information (visited on 02/15/2022).

[Sav21] Robert Savage. *ListenGpioExample*. Jan. 2021. URL: https://github.com/Pi4J/pi4j-v1/blob/master/pi4j-example/src/main/java/ListenGpioExample.java (visited on 02/15/2022).

[sci] scienceprog.com. *1-Wire protocol simple and easy*. URL: https://scienceprog.com/1-wire-protocol-simple-and-easy/ (visited on 02/10/2022).

[sha] sharetop. *max7219-java*. URL: https://github.com/sharetop/max7219-java (visited on 02/15/2022).

[Tut] Raspberry Pi Tutorials. *Programmieren lernen am Raspberry Pi – Teil 4: LEDs mit PWM dimmen*. URL: https://tutorials-raspberrypi.de/programmieren-lernen-raspberry-pi-gpio-pwm/ (visited on 02/14/2022).