FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

FH Aachen

Faculty Electrical Engineering and Information Technology

Information Systems Engineering

Master Thesis

Development of a hardware and software framework for the automated characterization of permanent magnets for low-field MRI systemsg

Submitted by **Marcel Werner Heinrich Friedrich Ochsendorf**

Matriculation number: **3120232**

Examiner: Prof. Dr.-Ing. Thomas Dey

Examiner: Prof. Dr.-Ing. Volkmar Schulz

Date: 01.01.2024

Erklärung

I hereby declare that I have prepared this thesis independently and without outside assistance. Text passages, which are based literally or in the sense on publications or lectures of other authors, are marked as such. The work has not yet been submitted to any other examination authority and has not yet been published.

Aachen, ___________, ________________________

Abstract

In the construction of low-field MRI devices based on permanent magnets, a large number of magnets are used. In order to realize a homogeneous B0 field with these magnets, which is necessary for many setups, the magnetic properties of these magnets have to be as similar to a certain degree. Due to the complex manufacturing process of neodymium magnets, the different properties, the direction of magnetization, can deviate from each other, which affects the homogeneity of the field. To adjust the field afterwards, a passive shimming process is typically performed, which is complex and time-consuming and requires manual corrections to the magnets used. To avoid this process, magnets can be systematically measured in advance. In this methodology, the recording, data storage and subsequent evaluation of the data play an important role. Various existing open-source solutions implement individual parts, but do not provide a complete data processing pipeline from aqusation to analysis and the data storage formats of these are not compatible to each other. For this use case, the MagneticReadoutProcessing library was created, which implements all major aspects of acquisition, storage, analysis, and each intermediate step can be customized by the user without having to create everything from scratch, favoring an exchange between different user groups. Complete documentation, tutorials and tests enable users to use and adapt the Framework as quickly as possible. The framework for the characterisation of different magnets, which requires the integration of magnetic field sensors, was used for the evaluation.

List of abbreviations

CDC Communication Device Class. 7

CLI Command Line Interface. 7–9, 12, 23–25, 31

GPIO General Purpose Input/Output. 15

GUI Graphical User Interface. 23

HAL Hardware Abstraction Layer. 19

HTML Hypertext Markup Language. 32, 33

I2C Inter-Integrated Circuit. 4, 7

IDE Integrated development environment. 28, 32

IP Internet Protocol. 21

LUT Lookup Table. 7

MRP MagneticReadoutProcessing. 8–11, 15, 18, 23–25, 28–31, 33

PC Personal Computer. 6–9, 15, 18–21, 24, 29

PDF Portable Document Format. 32

PIP Python Package Installer. 30

PPS Puls Per Second. 21

PTP Precision Time Protocol. 21

REST Representational State Transfer. 18, 20

SBC Single Board Computer. 15, 18, 21

UART Universal Asynchronous Receiver / Transmitter. 7, 15

USB Universal Serial Bus. 7, 9, 21

Contents

1 Introduction

1.1 Background and Motivation

1.1.1 Low-Field MRI

1.1.2 Shimming procedure

1.2 Aim of this Thesis

1.3 Structure

2 State of the art

2.1 Opensource projects

2.2 Conceptual design

- etnwicklung eines hardware uns software framework zur einfachen aquirierung von Meagnetfelddaten
- analysetools und funktionen

3 Usecases

4 Unified Sensor

- ziel ist es einen low cost hallsensor-interface zu entwickeln welcher möglichst
- universell
- verschienee sensoren abbilden kann
- mit verschienden magneten typen und formen nutzbar
- reproduzierbar
- 1d, 2d, 3d
- integration

4.1 Sensor selection

Tabelle 4.1: Implemented digital halleffect sensors

	TLV493D-A1B6	HMC5883L	MMC5603NJ	AS5510
Readout-Axis	3D	3D	3D	1D
Temperature-Sensor	yes	no	yes	no
Resolution [uT]	98	0.2	0.007	48
Range [mT]	±130.0	±0.8	±3	±50
Interface	Inter-Integrated Circuit (I2C)	I2C	I2C	I2C

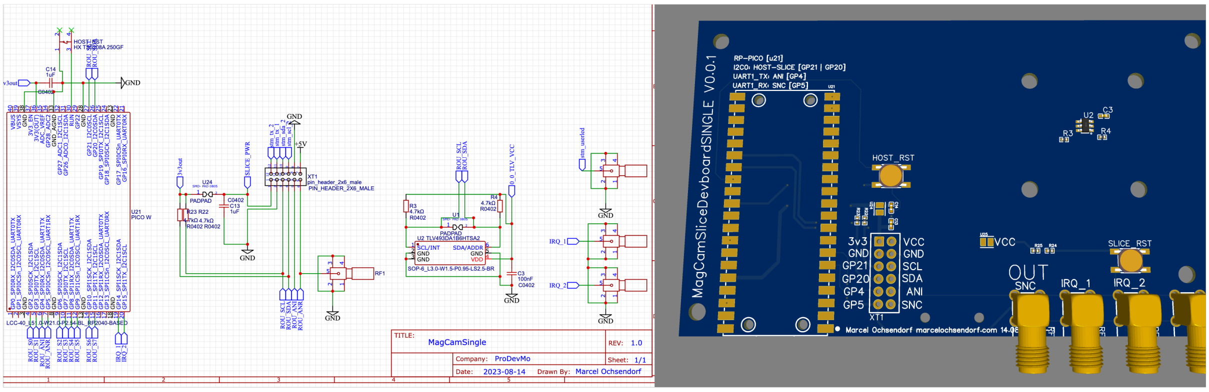


Figure 4-1: 1D sensor schematic and circuit board

- for higher ranges analog sensoren nutzen welche jedoch eine aufwändigere schaltung erfodern
- datenblatte links ergänzen
- alle i2c in der regel, welches eine einfache integration ermöglicht
- eingebauter temperatursensor ermöglicht temperaturkompensation
- conrad teslameter mit seperaten temperatursensor

4.2 Mechanical Structure

- 3d druck toleranztest
- magnet halterung mit kraftloser arretierung
- motoren und andere unter umstaänden magnetische teile in der nähe des sensors
- nylon schrauben, 3d druck, 3d gedruckte klemmverbindungen
- später rausrechnen durch kalibierung

4.3 Electrical Interface

4-1

- usb
- multiplexer for i2c sensors alredy implemented
- imput / output for sync

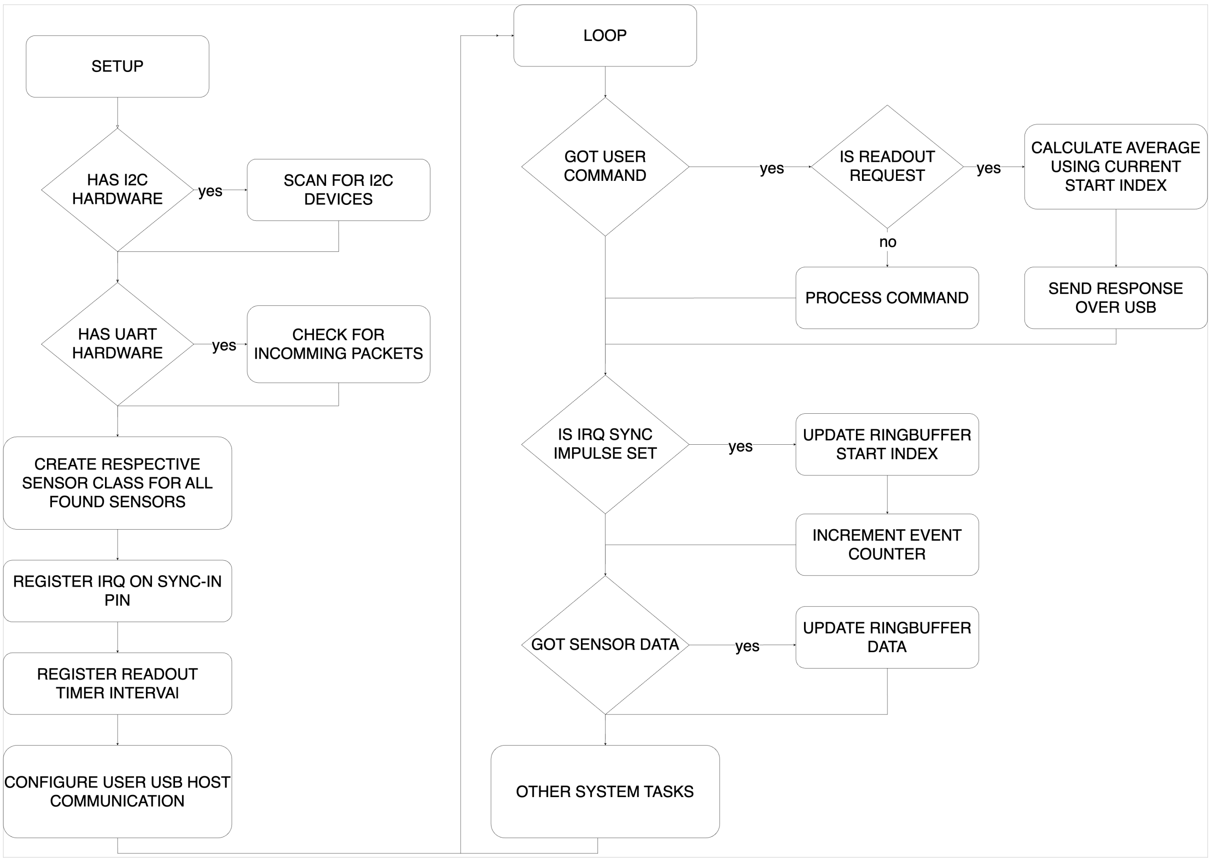


Figure 4-2: Unified sensor firmware simplified program strucutre

4.4 Firmware

Microcontroller firmware is software that is executed on a microcontroller in embedded systems. It controls the hardware and enables the execution of predefined functions. The firmware is used to process input data, control output devices and fulfil specific tasks according to the program code. It handles communication with sensors, actuators and other peripheral devices, processing data and making decisions. Firmware is critical to the functioning of devices.

The firmware is responsible for detecting the possible connected sensors4.1 and reading them out. This measured data can be forwarded to a host Personal Computer (PC) via a user interface and can then be further processed there. An important component is that as many common sensors as possible can be easily connected without having to adapt the firmware. This modularity was implemented using **abstract** class design. These are initiated according to the sensors found at startup. If new hardware is to be integrated, only the required functions4.1 need to be implemented.

```
#ifndef __CustomSensor_h__
#define __CustomSensor_h__
```

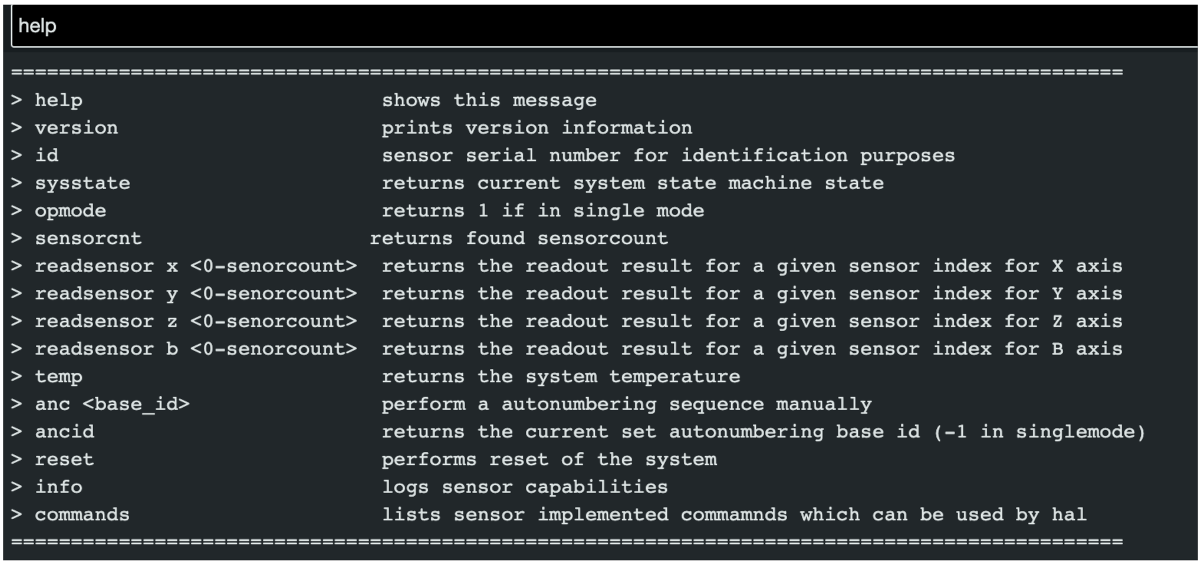
```
// register your custom sensor in implemented_sensors.h also
class CustomSensor: public baseSensor
{
public:
    CustomSensor();
    ~CustomSensor();
    // implement depending sensor communication interface
    bool begin(TwoWire& _wire_instance); // I2C
    bool begin(HardwareSerial& _serial_instance); // UART
    bool begin(Pin& _pin_instance); // ANALOG or DIGITAL PIN like
    onewire
    // FUNCTIONS USED BY READOUT LOGIC
    bool is_valid() override;
    String capabilities() override;
    String get_sensor_name() override;
    bool query_sensor() override;
    sensor_result get_result() override;
};
#endif
```

Listing 4.1: CustomSensor-Class for adding new sensor hardware support

The flow chart**4**-2 shows the start process and the subsequent main loop for processing the user commands and sensor results. When the microcontroller is started, the software checks whether known sensors are connected to I2C or Universal Asynchronous Receiver / Transmitter (UART) interfaces. If any are found (using a dedicated Lookup Table (LUT) with sensor address information), the appropriate class instances are created and these can later be used to read out measurement results.

Next, the subsystem for multi-sensor synchronisation4.4.2 is set up. The last step in the setup is to set up communication with the host or connected PC. All microcontroller platforms used here have a Universal Serial Bus (USB) slave port. The used usb descriptor is `Serial over (+usb)`- ((+usb)Communication Device Class (CDC)). This is used to emulate a virtual RS232 communication port using a USB port on a PC and usually no additional driver is needed on modern systems.

Now that the setup process is complete, the system switches to an infinite loop, which processes several possible actions. One task is, to react to user commands which can be sent to the system by the user via the integrated Command Line Interface (CLI). All sensors are read out via a timer interval set in the setup procedure and their values are stored in a ringbuffer. Ring buffers, offer efficient data management in limited memory. Its cyclic structure enables continuous overwriting of older data, saves memory space and facilitates seamless processing of real-time data. Ring buffers



```
help
==========================================================================================
> help                         shows this message
> version                      prints version information
> id                           sensor serial number for identification purposes
> sysstate                     returns current system state machine state
> opmode                       returns 1 if in single mode
> sensorcnt                   returns found sensorcount
> readsensor x <0-senorcount>  returns the readout result for a given sensor index for X axis
> readsensor y <0-senorcount>  returns the readout result for a given sensor index for Y axis
> readsensor z <0-senorcount>  returns the readout result for a given sensor index for Z axis
> readsensor b <0-senorcount>  returns the readout result for a given sensor index for B axis
> temp                         returns the system temperature
> anc <base_id>                perform a autonumbering sequence manually
> ancid                        returns the current set autonumbering base id (-1 in singlemode)
> reset                        performs reset of the system
> info                         logs sensor capabilities
> commands                     lists sensor implemented commamnds which can be used by hal
==========================================================================================
```

Figure **4**-3: Sensors CLI

are well suited for applications with variable data rates and minimise the need for complex memory management. The buffer can be read out by command and the result of the measurement is sent to the host. Each sensor measurement result is transmitted from the buffer to the host together with a time stamp and a sequential number. This ensures that in a multi-sensor setup with several sensors. The measurements are synchronized4.4.2 in time and are not out of sequence or drift.

4.4.1 Communication interface

Each sensor that has been loaded with the firmware, registeres on to the host PC as a serial interface. There are several ways for the user to interact with this:

- Use with MagneticReadoutProcessing (MRP)-library5
- Stand-alone mode via sending commands using built-in CLI

The CLI mode is a simple text-based interface with which it is possible to read out current measured values, obtain debug information and set operating parameters. This allows you to quickly determine whether the hardware is working properly after installation. The CLI behaves like terminal programmes, displaying a detailed command reference**4**-3 to the user after connecting. The current measured value can be output using the readout command**4**-4.

The other option is to use the MRP-library. The serial interface is also used here.

Figure 4-4: Query sensors b value using CLI

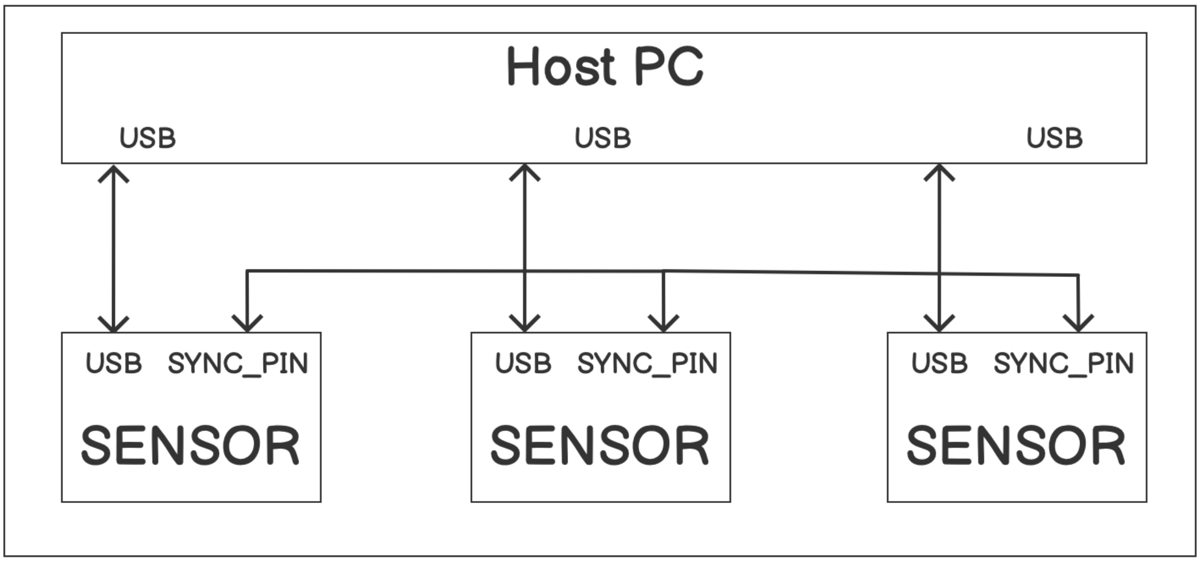


Figure 4-5: Multi sensor synchronisation wiring example

However, after a connection attempt by the MRPHal module5.1.2 of the MRP-library, the system switches to binary mode, which is initiated using the sbm command. However, the same commands are available as for CLI-based communication.

4.4.2 Sensor syncronsisation interface

One problem with the use of several sensors on one readout host PC is that the measurements may drift over time. On the one hand, USB latencies can occur. This can occur due to various factors, including device drivers, data transfer speed and system resources. High-quality USB devices and modern drivers often minimise latencies. Nevertheless, complex data processing tasks and overloaded USB ports can lead to delays.

The other issue is sending the trigger signal from the readout software5. Here too, unpredictable latencies can occur, depending on which other tasks are also executed on this port.

In order to enable the most stable possible synchronisation between several sensors, an option has already been created to establish an electrical connection between sensors. This is used together with the firmware to synchronise the readout intervals. The schematic**4**-5 shows how several sensors must be wired together in order to implement this form of synchronisation.

Once the hardware has been prepared, the task of the firmware of the various sensors is to find a common synchronisation clock. To do this, the register irq on sync pin is overwritten. To set one primary and several secondary sensors, each sensor waits for an intial pulse on the sync-pin**4**-6. Each sensor starts a random timer beforehand, which sends a pulse on the sync line. All others receive this and switch to secondary mode and synchronise the measurements based on each sync pulse received. Since the presumed primary sensor cannot register its own sync pulse (because the pin is switched to output), there is a timeout got pulse within 1000ms and this becomes the primary sensor. This means that in a chain of sensors there is exactly one primary and many secondary sensors. In single-sensor operation, this automatically jumps to primary sensor operation through the got impulse within 1000ms query. The synchronisation status can be queried via the user interface4.4.1 using the opmode**4**-7 command. An important aspect of the implementation here was that there is no numbering or sequence of the individual sensors. This means that for the subsequent readout of the measurements, it is only important that they are taken at the same interval across all sensors. The sensor differentiation takes place later in the readout software5 by using the sensor identification number.

4.5 Example sensors

Two functional sensor platforms4.2 were built in order to create a solid test platform for later tests and for the development of the MRP library with the previously developed sensor concepts.

Tabelle 4.2: Build sensors with different capabilities

	1D4.5.1	1D: dual sensor	3D: Fullsphere4.5.2
Maximal magnet size	Cubic 30x30x30	Cubic 30x30x30	Cubic 20x20x20

	1D4.5.1	1D: dual sensor	3D: Fullsphere4.5.2
Sensor type	MMC5603NJ	TLV493D	TLV493D
Sensor count	1	2	1
Scanmode	static (1 point)	static (2 points)	dynamic (fullsphere)

These cover all the required functions described in the Usecases3. The most important difference, apart from the sensor used, is the scan mode. In this context, this describes whether the sensor can measure a **static** fixed point on the magnet or if the sensor can move dynamically around the magnet using a controllable manipulator.

In the following, the hardware structure of a **static** and dynamic sensor is described. For the **static** sensor, only the 1D variant is shown, as this does not differ significantly from the structure of the 1D: dual sensor, except it uses two TLV493D sensors, mounted above and on top of the magnet.

4.5.1 1D: Single Sensor

The 1D sensor**4**-8 is the simplest possible sensor that is compatible with the Unified Sensor Firmware4.4 platform and also with the MRP-library5.

The electrical level here is based on a Raspberry–Pi Pico together with the MMC5603NJ magnetic sensor. The mechanical setup consists of four 3D-printed components, which are fixed together with nylon screws to minimise possible influences on the measurement.

Since the MMC5603NJ only has very limited measuring range, even small neodymium magnets already saturate this range, it is possible to use 3D-printed spacers above the sensor.

The standard magnet holder can be adapted for different magnet shapes and can be placed on the spacer without play in order to be able to perform a repeatable

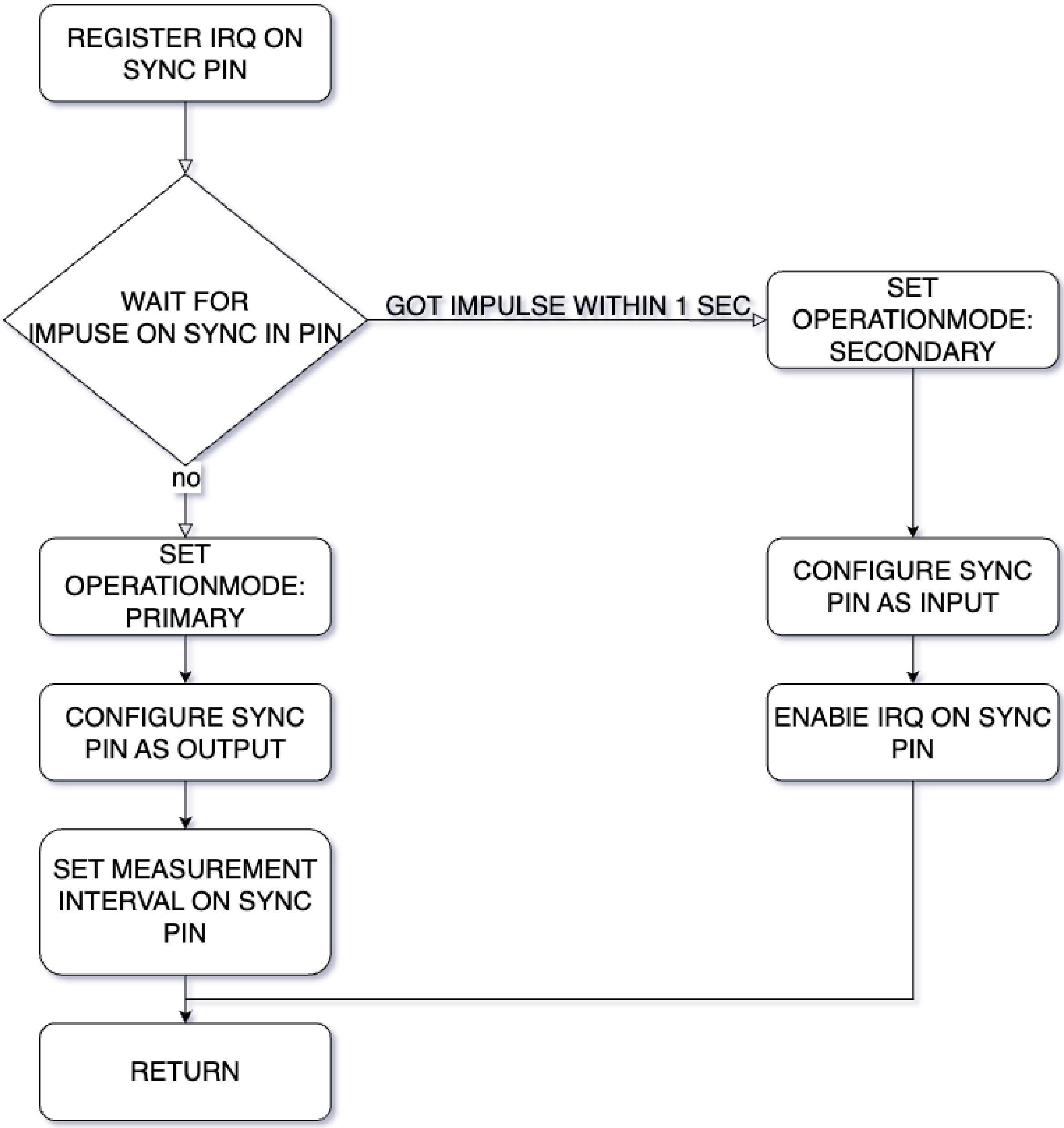


Figure **4**-6: Unified sensor firmware multi sensor synchronisation procedure

Figure **4**-7: Query opmode using CLI

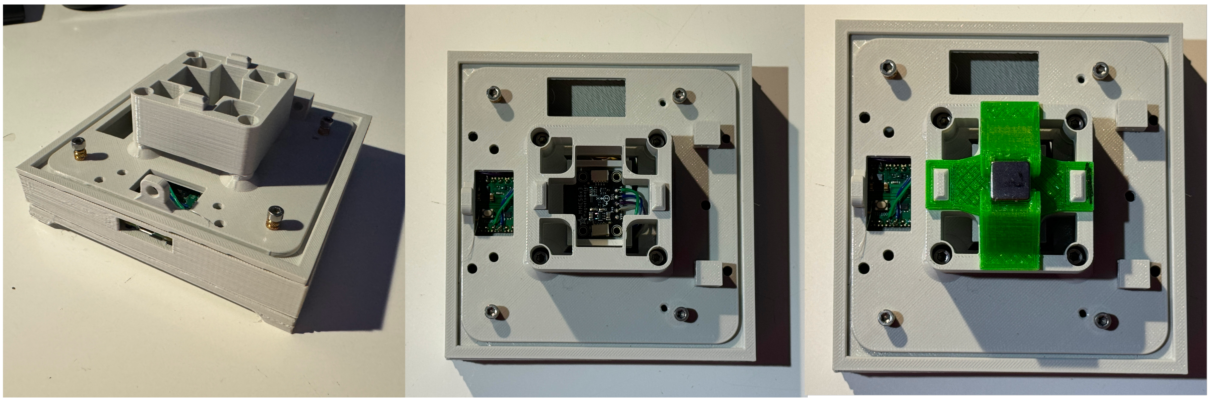


Figure 4-8: 1D sensor construction with universal magnet mount

measurement without introducing measurement irregularities by mechanically changing the magnet.

4.5.2 3D: Fullsphere

The 3D Fullsphere**4**-9 sensor offers the possibility to create a 3D map**4**-10 of the magnets. The magnet sensor is mounted on a movable arm, which can move 180 degrees around the magnet on one axis. In order to be able to map the full sphere, the magnet is mounted on a turntable. This permits the manipulator to move a polar coordinate system.

As the magnets in the motors, as with the screws used in the 1D sensor, can influence the measurements of the magnetic field sensor, the distance between these components and the sensor or magnets was increased. The turntable and its drive motor are connected to each other via a belt.

On the electrical side, it also consists of a SKR Pico stepper motor controller, together with the TLV493D magnetic field sensor. This was chosen because of its larger measuring range and can therefore be used more universally without having to change the sensor of the arm.

4.5.3 Integration of an industry teslameter

As the sensors shown so far relate exclusively to self-built, low-cost hardware, the following section shows how existing hardware can be integrated into the system. This

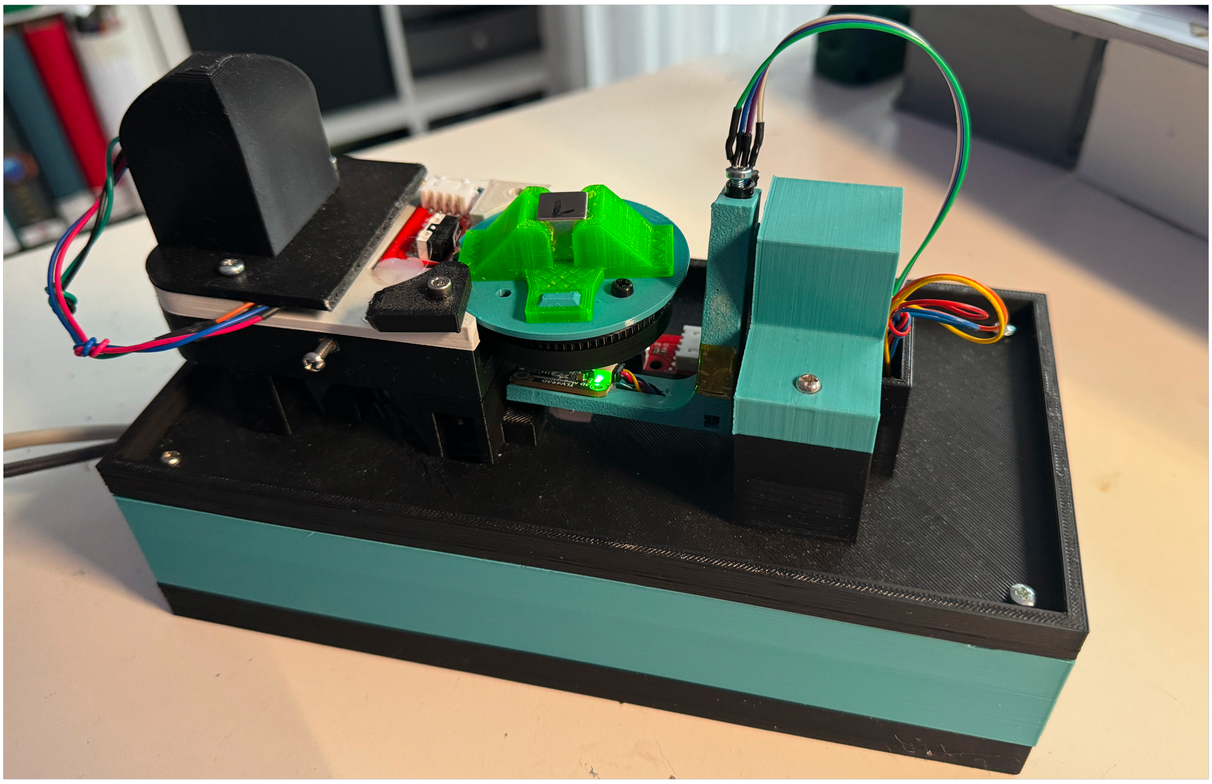


Figure 4-9: Full-Sphere sensor implementation using two Nema17 stepper motors in a polar coordinate system

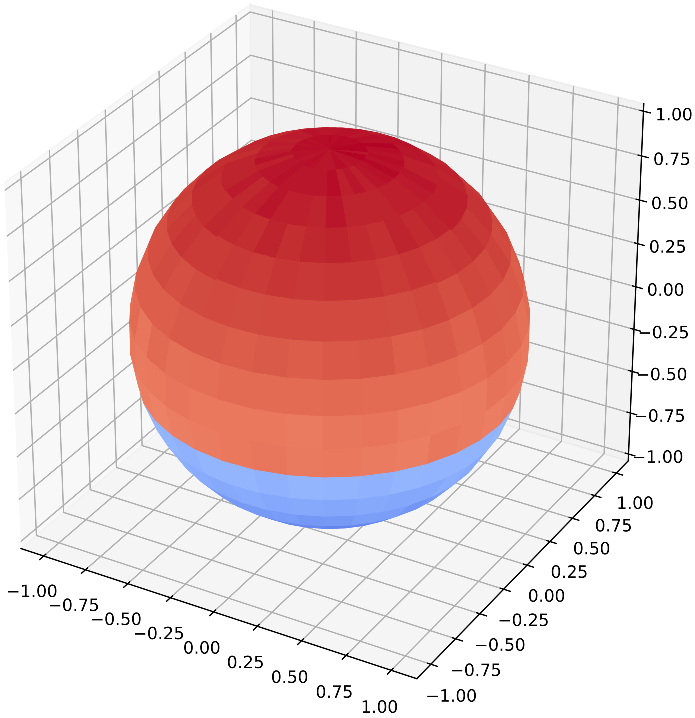


Figure 4-10: 3D plot of an N45 12x12x12 magnet using the 3D fullsphere sensor

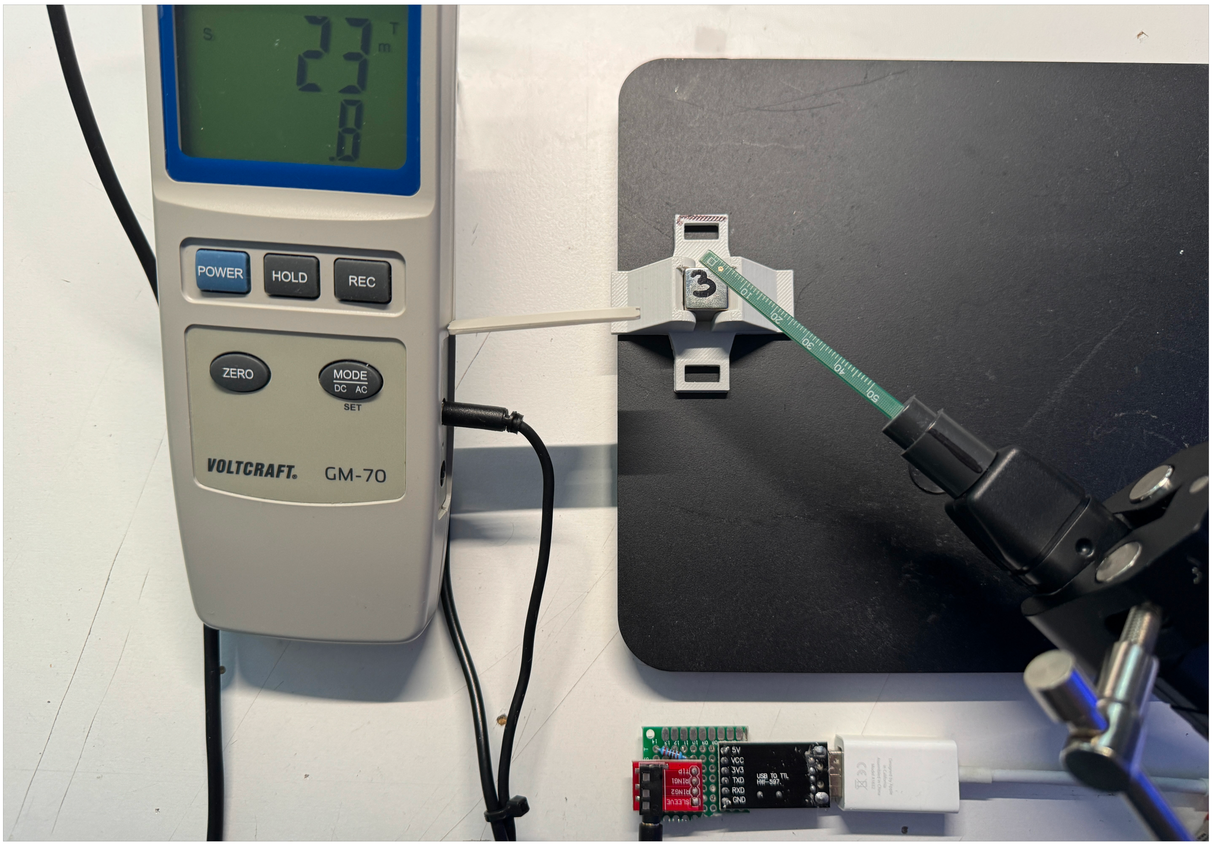


Figure 4-11: Voltcraft GM70 teslameter with custom PC interface board

is shown here using a temperature-compensated Voltcraft GM–70 telsameter4-11, which has a measuring range of 0–3T with a resolution of 0.1mT. This offers an RS232 interface with a documented protocol4.3 for connection to a PC. This connectivity makes it possible to make the device compatible with the MRP library using interface software [10] executable on the host PC. However, it does not offer the range of functions that the Unified Sensor Firmware4.4 offers.

Another option is a custom interface board between the meter and the PC. This is a good option as many modern PCs or Single Board Computer (SBC)s no longer offer an RS232 interface. As with the other sensors, this interface consists of your RaspberryPi Pico with an additional level shifter. The Teslameter is connected to the microcontroller using two free General Purpose Input/Output (GPIO)s in UART mode. The Unified Sensor Firmware4.4 was adapted using a separate build configuration and the protocol of the measuring device was implemented.

Tabelle 4.3: Voltcraft GM70 serial protocol

BYTE-INDEX	REPRESENTATION	VALUE
0	PREAMBLE	0x2
1		0x1
2		0x4
3	UNIT	‘B’ => Gauss ‘E’ => mT
5	POLARITY	‘1’ => *0.1 ‘2’* => 0.01
6	value MSB	0x-0xFF
13	value LSB	0x-0xFF
14	STOP	0x3

This software or hardware integration can be carried out on any other measuring device with a suitable communication interface and a known protocol thanks to the modular design.

5 Software readout framework

5.1 Library requirements

5.1.1 Concepts

- beispiele für projekte welche nur einzelne schnritte implementieren
- so kann man sich auf die implementierung

5.1.2 User interaction points

- grafik zeigen
- einzelne module erleutern

HAL

- aufbau hal im grunde wird nur ein die commandos an das sensor cli weitergegeben
- alle sensoren implementieren mehr oder weniger die gleichen befehle
- hal gibt nur weiter und ist “dumm”

Visualisation

5.1.3 Export

- format import export

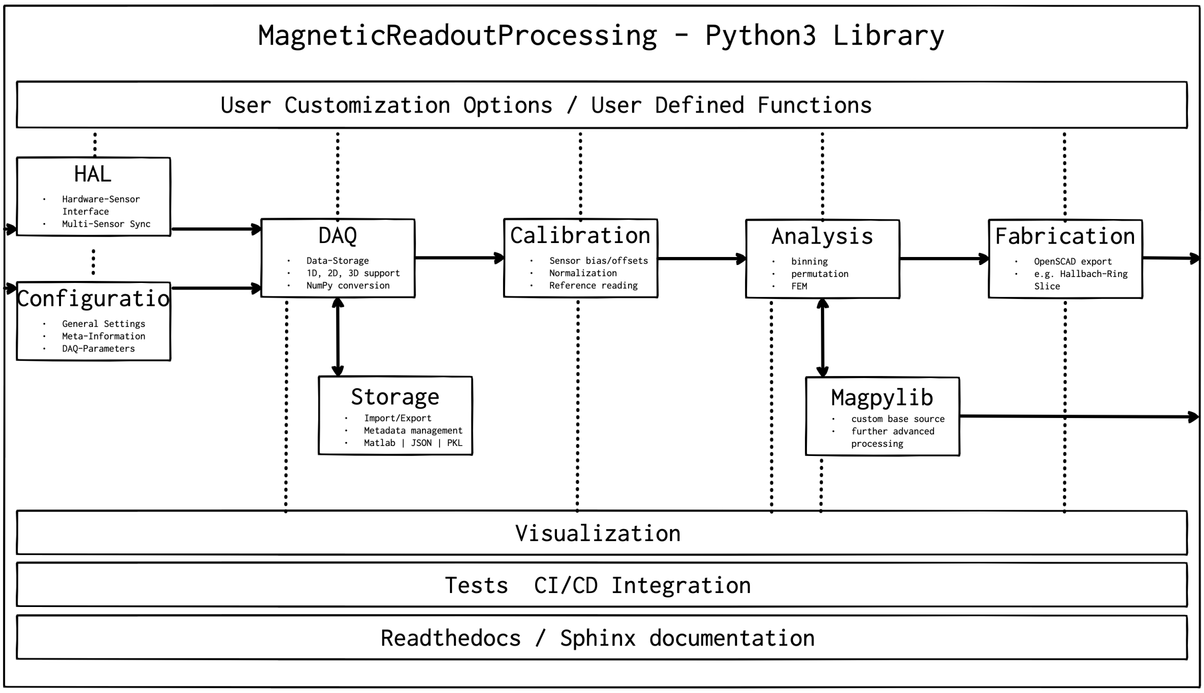


Figure **5**-1: MRPlib COMPLETE FLOW

- matlab

Meta-Data

5.1.4 Multible sensor setup

At the current state of implementation, it is only possible to detect and use sensors that are directly connected to the PC with the MRP-library. This has the disadvantage that there must always be a physical connection. This can make it difficult to install multiple sensors in measurement setups where space or cable routing options are limited. To make sensors connected to a small remote PC available on the network, the Proxy**5**-2 module has been developed. This can be a SBC (e.g. a Raspberry Pi). The small footprint and low power consumption make it a good choice. It can also be used in a temperature chamber. The approach of implementing this via a Representational State Transfer (REST) interface also offers the advantage that several measurements or experiments can be recorded at the same time with the sensors.

Another application example is when sensors are physically separated or there are long distances between them. By connecting several sensors via the proxy module, it is possible to link several instances and all sensors available in the network are available

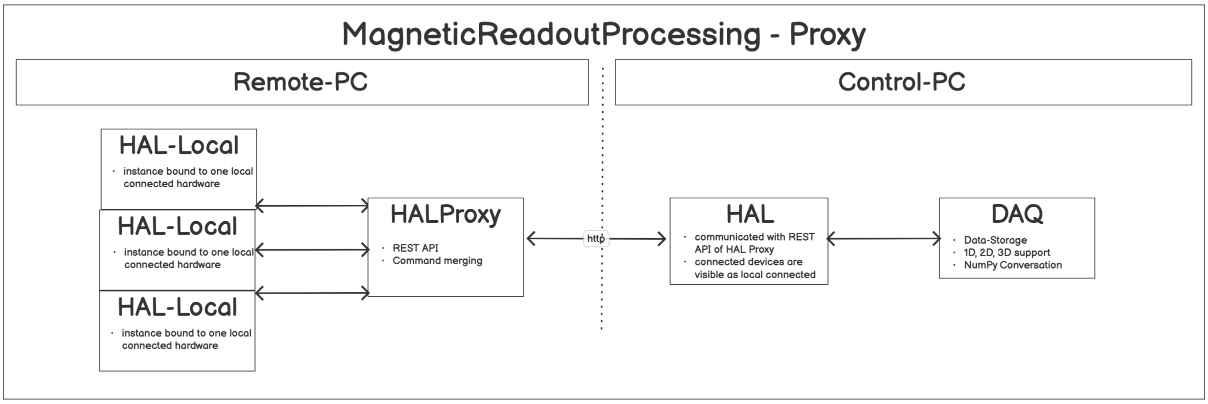


Figure 5-2: MRPlib Proxy Module

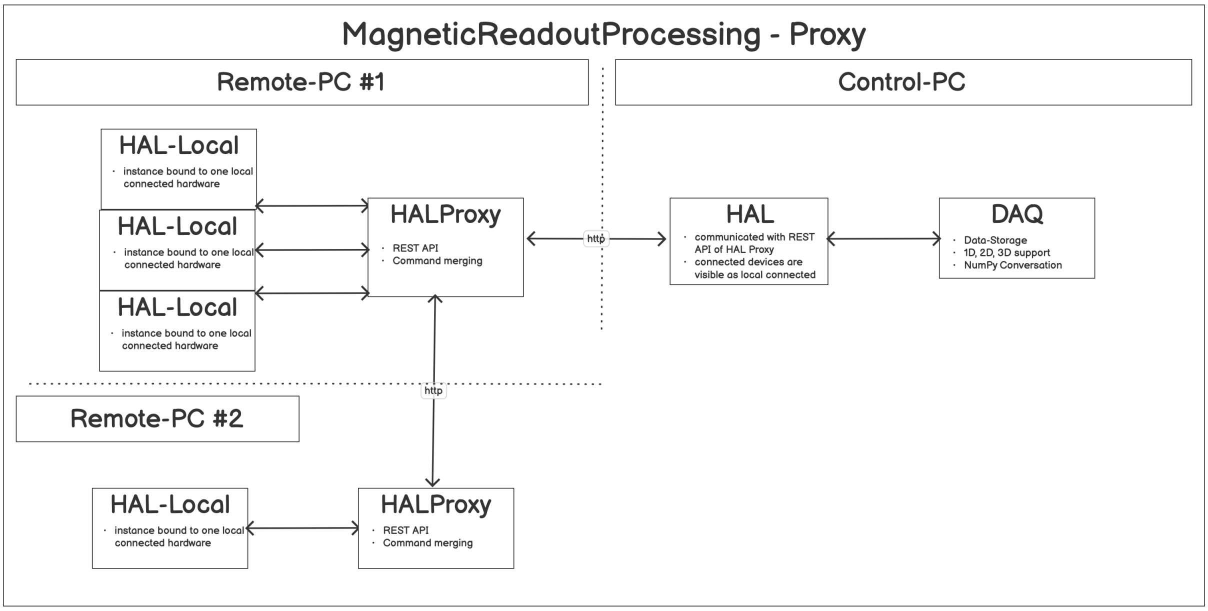


Figure 5-3: mrp proxy multi

to the control PC.

The figure 5-3 shows the modified multi–proxy – multi–sensor topology. Here, both proxy instances do not communicate directly with the control PC, but remote (+pc) #2 is connected to remote (+pc) #1. This is then visible as a sensor opposite the Control PC, even if there are several proxy instances behind it.

Network-Proxy

The figure 5-2 shows the separation of the various Hardware Abstraction Layer (HAL) instances, which communicate with the physically connected sensors on the remote-PC and the control-PC side, which communicates with the remote side via the network. For the user, nothing changes in the procedure for setting up a measurement. The proxy application must always be started5.1 on the remote PC side.

```
# START PROXY INSTNACE WITH TWO LOCALLY CONNECTED SENSORS
$ python3 mrpproxy.py proxy launch /dev/ttySENSOR_A /dev/ttySENSOR_B # add another proxy instance http://proxyinstance_2.local for multi-sensor, multi-proxy chain
Proxy started. http://remotepc.local:5556/
PRECHECK: SENSOR_HAL: 1337 # SENSOR A FOUND
PRECHECK: SENSOR_HAL: 4242 # SENSOR B FOUND
Terminate Proxy instance [y/N] [n]:
```

Listing 5.1: MRPproxy usage to enable local sensor usage over network

After the proxy instance has been successfully started, it is optionally possible to check the status via the REST interface:5.2

```
# GET PROXY STATUS
$ wget http://proxyinstance.local:5556/proxy/status
{
"capabilities":[
   "static",
   "axis_b",
   "axis_x",
   "axis_y",
   "axis_z",
   "axis_temp",
   "axis_stimestamp"
],
"commands":[
   "status",
   "initialize",
   "disconnect",
   "combinedsensorcnt",
   "sensorcnt",
   "readsensor",
   "temp"
]

# RUN A SENSOR COMMAND AND GET THE TOTAL SENSOR COUNT
$ wget http://proxyinstance.local:5556/proxy/command?cmd=combinedsensorcnt
{
"output":[
   "2"
]
```

```
}
}
```

Listing 5.2: MRPproxy REST enpoiint query examples

The query result shows that the sensors are connected correctly and that their capabilites have also been recognised correctly. To be able to configure a measurement on the other, only the Internet Protocol (IP) address or host name of the remote PC is required5.3.

```
# CONFIGURE MEASUREMENT JOB USING A PROXY INSTANCE
$ MRPcli config setupsensor testcfg --path http://proxyinstance.local
    :5556
> remote sensor connected: True using proxy connection:
> http://proxyinstance.local:5556 with 1 local sensor connected
```

Listing 5.3: MRPcli usage example to connect with a network sensor

Sensor Syncronisation

Another important aspect when using several sensors via the proxy system is the synchronisation of the measurement intervals between the sensors. Individual sensor setups do not require any additional synchronisation information, as this is communicated via the USB interface. If several sensors are connected locally, they can be connected to each other via their sync input using short cables. One sensor acts as the central clock as described in**??**. this no longer works for long distances and the syncronisation must be made via the network connection.

If time-critical synchronisation is required, Precision Time Protocol (PTP) and Puls Per Second (PPS) output functionality can be used on many SBC, such as the Raspberry−Pi Compute Module 4.

- was ptp, bild pps output
- alle clients über ptp verbunden
- dso bild von jeff gerling über rpi4 ptp

Command-Router

- nummerieierung zuerst lokale sensoren dann weitere proxy sensoren

- commando templating
- einzelne sensor capabilites werden gemerged
- ids werden aufsuimmiert
- spätere sensor identifikation geschieht über die jeweilige sensor id, welche über diesen index abgefragt werden kann
- tablle mit mergin algorithmus

5.1.5 Examples

6 Usability improvements

Usability improvements in software libraries are crucial for efficient and user-friendly development. Intuitive API documentation, clearly structured code examples and improved error messages promote a smooth developer experience. Standardised naming conventions and well thought-out default values simplify the application. A Graphical User Interface (GUI) or CLI application for complex libraries can make it easier to use, especially for developers with less experience. Continuous feedback through automated tests and comprehensive error logs enable faster bug fixing. The integration of community feedback and regular updates promotes the adaptability of the MRP-library to changing needs. Effective usability improvements help to speed up development processes and increase the satisfaction of the developer community. In the following, some of these have been added in and around the MRP-library, but they are only optional components for the intended use.

6.1 Command Line Interface

In the first version of this MRP-library, the user had to write his own Python scripts even for short measurement and visualisation tasks. However, this was already time-consuming for reading out a sensor and configuring the measurement parameters and metadata and quickly required more than 100 lines of new Python code. Although such

```
CONFIGURE READING
READING-NAME: [read_dualsensor_normal]: >? new_measurement
OUTPUT-FOLDER [./readings/tlv493d_N45_12x12x12/]: >? ./
final output path for reading /Users/marcelochsendorf/Downloads/MagneticReadoutProcessing/src/MagneticReadoutProcessing
SUPPORTED MAGNET TYPES
0 > NOT_SPECIFIED
1 > RANDOM_MAGNET
2 > N45_CUBIC_12x12x12
3 > N45_CUBIC_15x15x15
4 > N45_CUBIC_9x9x9
5 > N45_CYLINDER_5x10
6 > N45_SPHERE_10
Please select one of the listed magnet types [0-6] [2]:
>?  3
```

Figure **6**-1: MRP CLI output to configure a new measurement examples are provided in the documentation, it must be possible for programming beginners in particular to use them. To simplify these tasks, a CLI**6**-2 was implemented around this MRP-library. The (+mrp)-library-CLI implements the following functionalities:

- Detection of connected sensors
- Configuration of measurement series
- Recording of measured values from stored measurement series
- Simple commands for checking recorded measurement series and their data.

Thanks to this functionality of the CLI, it is now possible to connect a sensor to the PC, configure a measurement series with it and run it at the end. The result is then an exported file with the measured values. These can then be read in again with the MRP-library and processed further. The following bash code6.1 shows the setup procedure in detail:

```
# CLI EXAMPLE FOR CONFIGURING A MEASUREMENT RUN
## CONFIGURE THE SENSOR TO USE
$ MRPcli config setupsensor testcfg
> 0 - Unified Sensor 386731533439 - /dev/cu.usbmodem3867315334391
> Please select one of the found sensors [0]:
> sensor connected: True 1243455
## CONFIGURE THE MEASUREMENT
$ MRPcli config setup testcfg
> CONFIGURE testcfg
> READING-NAME: [testreading]: testreading
> OUTPUT-FOLDER [/cli/reading]: /tmp/reading_folder_path
> NUMBER DATAPOINTS: [1]: 10
> NUMBER AVERAGE READINGS PER DATAPOINT: [1]: 100
# RUN THE CONFIGURED MEASUREMENT
$ MRPcli measure run
> STARTING MEASUREMENT RUN WITH FOLLOWING CONFIGS: ['testcfg']
> config-test: OK
> sensor-connection-test: OK
> START MEASUREMENT CYCLE
> sampling 10 datapoints with 100 average readings
> SID:0 DP:0 B:47.359mT TEMP:23.56
> ....
> dump_to_file testreading_ID:525771256544952_SID:0_MAG:
    N45_CUBIC_12x12x12.mag.json
```

Listing 6.1: CLI example for configuring a measurement run

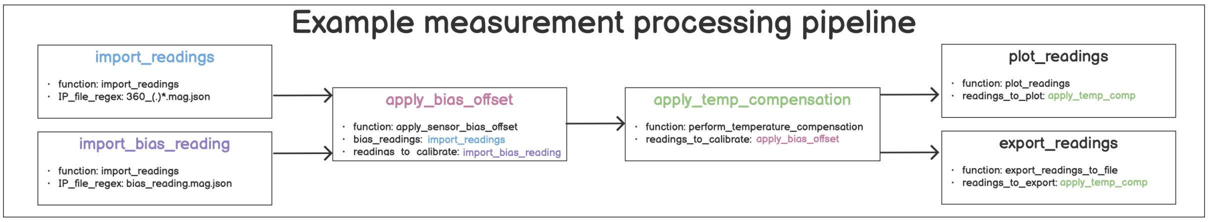


Figure 6-2: example measurement analysis pipeline

6.2 Programmable data processing pipeline

After it is very easy for users to carry out measurements using the CLI, the next logical step is to analyse the recorded data. This can involve one or several hundred data records. Again, the procedure for the user is to write their own evaluation scripts using the MRP-library. This is particularly useful for complex analyses or custom algorithms, but not necessarily for simple standard tasks such as bias compensation or graphical plot outputs.

For this purpose, a further CLI application was created, which enables the user to create and execute complex evaluation pipelines for measurement data without programming. The example**6**-2 shows a typical measurement data analysis pipeline, which consists of the following steps:

- Import the measurements
- Determine sensor bias value from imported measurements using a reference measurement
- Apply linear temperature compensation
- Export the modified measurements
- Create a graphical plot of all measurements with standard deviation

In order to implement such a pipeline, the yaml file format was chosen for the definition of the pipeline, as this is easy to understand and can also be easily edited with a text editor. Detailed examples can be found in the documentation[2]. The pipeline definition consists of sections which execute the appropriate Python commands in the background. The signatures in the yaml file are called using reflection and a real-time search of the loaded global() symbol table[6]. This system makes almost all Python functions available to the user. To simplify use, a pre-defined list of tested MRP library functions for use in pipelines is listed in the documentation[2]. The following pipeline definition6.2 shows the previously defined steps**6**-2 as yaml syntax.

```
stage import_readings:
  function: import_readings
  parameters:
    IP_input_folder: ./readings/fullsphere/
    IP_file_regex: 360_(.)*.mag.json

stage import_bias_reading:
  function: import_readings
  parameters:
    IP_input_folder: ./readings/fullsphere/
    IP_file_regex: bias_reading.mag.json

stage apply_bias_offset:
  function: apply_sensor_bias_offset
  parameters:
    bias_readings: stage import_bias_reading # USE RESULT FROM FUNCTION import_bias_reading
    readings_to_calibrate: stage import_readings

stage apply_temp_compensation:
  function: apply_temperature_compensation
  parameters:
    readings_to_calibrate: stage import_readings # USE RESULT FROM FUNCTION import_readings

stage plot_normal_bias_offset:
  function: plot_readings
  parameters:
    readings_to_plot: stage apply_temp_compensation
    IP_export_folder: ./readings/fullsphere/plots/
    IP_plot_headline_prefix:  Sample N45 12x12x12 magnets calibrated

stage export_readings:
  function: export_readings
  parameters:
    readings_to_plot: stage apply_temp_compensation
    IP_export_folder: ./readings/fullsphere/plots/
```

Listing 6.2: Example User Defined Processing Pipeline

Each pipeline step is divided into `stages`, which contain a name, the function to be executed and its parameters. The various steps are then linked by using the `stage <name>` as the input parameter of the next function to be executed (see comments in 6.2). It is therefore also possible to use the results of one function in several others without them directly following each other. The disadvantages of this system are the following:

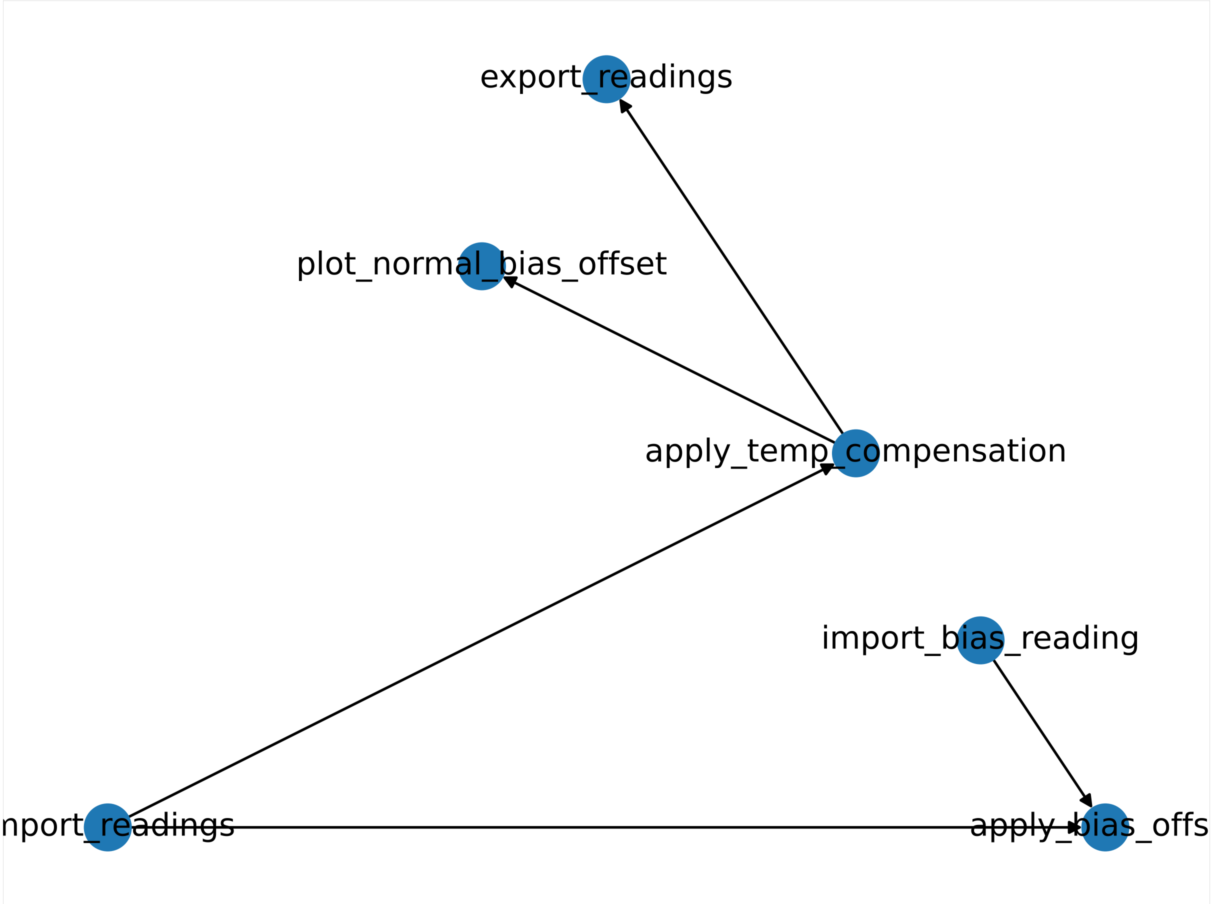


Figure **6**-3: Result step execution tree from user defined processing pipeline example

- No circular parameter dependencies
- Complex determination of the execution sequence of the steps

To determine the order of the pipeline steps, the parser script created converts them into one problem of the graph theories. Each step represents a node in the graph and the steps referred to by the input parameter form the edges.

After several simplification steps, determination of possible start steps and repeated traversal, the final execution sequence can be determined in the form of a call tree**6**-3. The individual steps are then executed along the graph. The intermediate results and the final results**6**-4 are saved for optional later use.

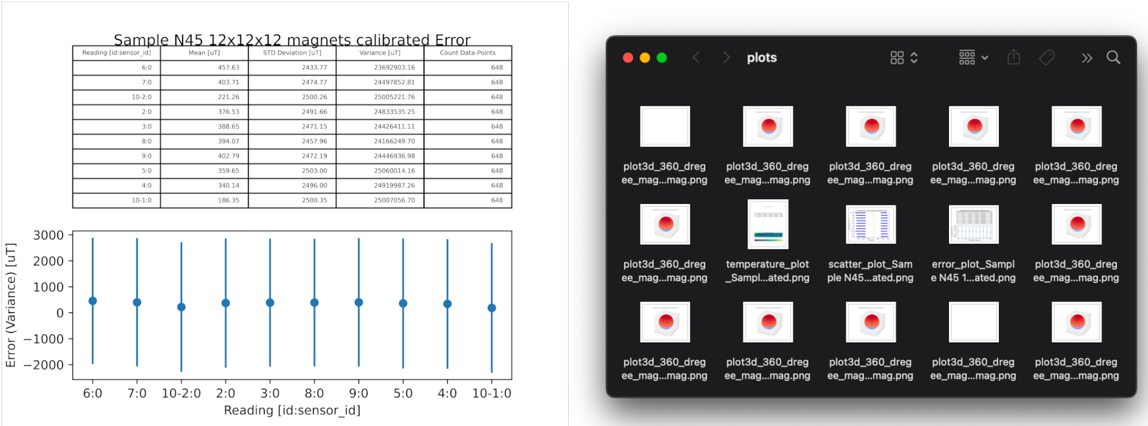


Figure 6-4: pipeline output files after running example pipeline on a set of readings

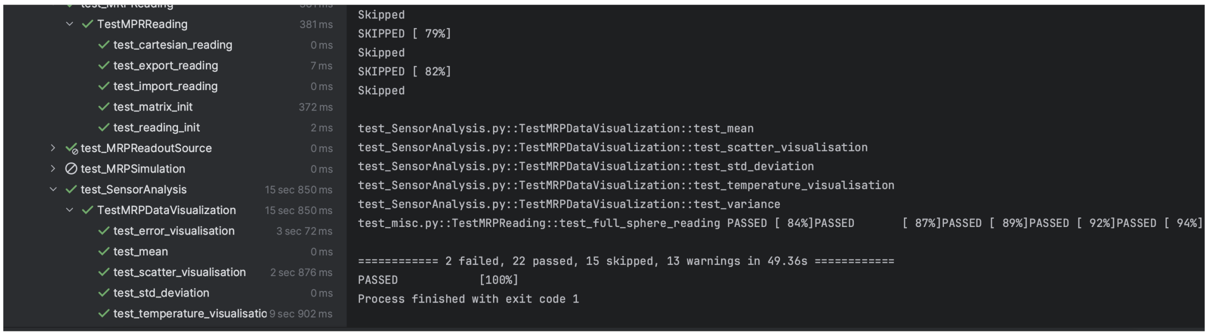


Figure 6-5: MRP library test results for different submodules executed in PyCharm IDE

6.3 Tests

Software tests in libraries offer numerous advantages for improving quality and efficiency. Firstly, they enable the identification of errors and vulnerabilities before software is published as a new version. This significantly improves the reliability of MRP-library applications. Tests also ensures consistent and reliable performance, which is particularly important when libraries are used by different users and for different usecases.

During the development of the MRP-library, test cases were also created for all important functionalities and use cases. The test framework PyTest[9] was used for this purpose, as it offers direct integration in most IDEs (see **6**-5) and also because it provides detailed and easy-to-understand test reports as output in order to quickly identify and correct errors. It also allows to tag tests, which is useful for grouping tests or excluding certain tests in certain build environment scenarios. Since all intended use cases were mapped using the test cases created, the code of the test cases could later be used in slightly simplified variants6.3 as examples for the documentation.

```python
class TestMPRReading(unittest.TestCase):
    # PREPARE A INITIAL CONFIGURATION FILE FOR ALL FOLLOWING TEST CASES IN THIS FILE
    def setUp(self) -> None:
        self.test_folder: str = os.path.join(os.path.dirname(os.path.abspath(__file__)), "tmp")
        self.test_file:str = os.path.join(self.import_export_test_folderpath, "tmp")

    def test_matrix(self):
        reading: MRPReading = MRPSimulation.generate_reading()
        matrix: np.ndarray = reading.to_numpy_matrix()
        n_phi: float = reading.measurement_config.n_phi
        n_theta: float = reading.measurement_config.n_theta
        # CHECK MATRIX SHAPE
        self.assertTrue(matrix.shape != (n_theta,))
        self.assertTrue(len(matrix.shape) <= n_phi)

    def test_export_reading(self) -> None:
        reading: MRPReading = MRPSimulation.generate_reading()
        self.assertIsNotNone(reading)
        # EXPORT READING TO A FILE
        reading.dump_to_file(self.test_file)

    def test_import_reading(self):
        # CREATE EMPTY READING AND LOAD FROM FILE
        reading_imported:MRPReading = MRPReading.MRPReading(None)
        reading_imported.load_from_file(self.test_file)
        # COMPARE
        self.assertIsNotNone(reading_imported.compare(MRPSimulation.generate_reading()))
```

Listing 6.3: Example pytest class for testing MRPReading module functions

One problem, however, is the parts of the MRP-library that require direct access to external hardware. These are, for example, the MRPHal and MRPHalRest modules, which are required to read out sensors connected via the network. Two different approaches were used here. In the case of local development, the test runs were carried out on a PC that can reach the network hardware and thus the test run could be carried out with real data.

In the other scenario, the tests are to be carried out before a new release in the repository on the basis of Github Actions[8]. Here there is the possibility to host local runner software, which then has access to the hardware, but then a PC must be permanently available for this task. Instead, the hardware sensors were simulated by software and executed

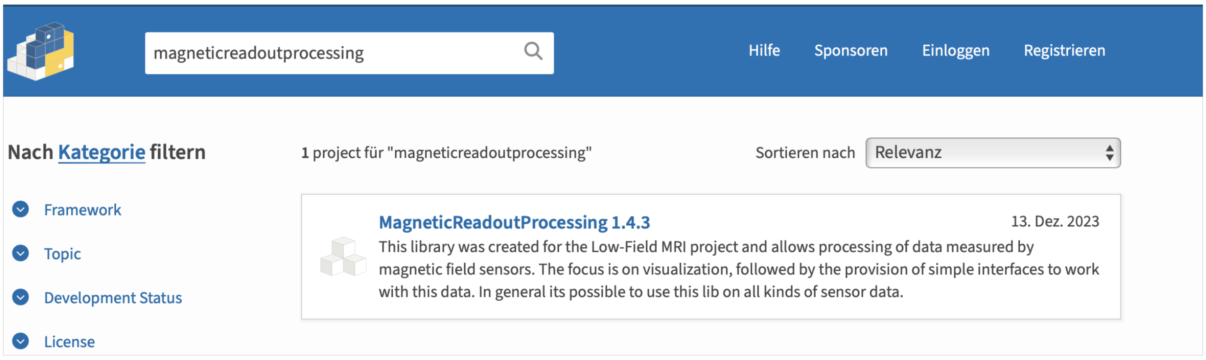


Figure 6-6: MagneticReadoutProcessing library hosted on PyPi

via virtualisation on the systems provided by Github Actions[8].

6.4 Package distribution

One important point that improves usability for users is the simple installation of the MRP-library. As it was created in the Python programming language, there are several public package directories where users can provide their software modules. Here, PyPi [5]6-6[4] is the most commonly used package directory and offers direct support for the package installation programm Python Package Installer (PIP)6.4.

In doing so, PIP not only manages possible package dependencies, but also manages the installation of different versions of a package. In addition, the version compatibility is also checked during the installation of a new package, which can be resolved manually by the user in the event of conflicts.

```bash
# https://pypi.org/project/MagneticReadoutProcessing/
# install the latest version
$ pip3 install MagneticReadoutProcessing
# install the specific version 1.4.0
$ pip3 install MagneticReadoutProcessing==1.4.0
```

Listing 6.4: Bash commands to install the MagneticReadoutProcessing (+mrp)-library using pip

To make the MRP-library compatible with the package directory, Python provides separate installation routines that build a package in an isolated environment and then provide an installation wheel archive. This can then be uploaded to the package directory.

Since the MRP-library requires additional dependencies (e.g. numpy, matplotlib), which cannot be assumed to be already installed on the target system, these must be installed prior to the actual installation. These can be specified in the MRP-library installation configuration setup.py6.5 for this purpose.

```python
# dynamic requirement loading using 'requirements.txt'
req_path = './requirements.txt'
with pathlib.Path(req_path).open() as requirements_txt:
    install_requires = [str(requirement) for requirement in
        pkg_resources.parse_requirements(requirements_txt)]

setup(name='MagneticReadoutProcessing',
      version='1.4.3',
      url='https://github.com/LFB-MRI/MagnetCharacterization/',
      packages=['MRP', 'MRPcli', 'MRPudpp', 'MRPproxy'],
      install_requires=install_requires,
      entry_points={
          'console_scripts': [
            'MRPCli = MRPcli.cli:run',
            'MRPUdpp = MRPudpp.uddp:run',
            'MRPproxy = MRPproxy.mrpproxy:run'
          ]
      }
    )
```

Listing 6.5: setup.py with dynamic requirement parsing used given requirements.txt

To make the CLI scripts written in Python easier for the user to execute without having to use the python3 prefix. This has been configured in the installation configuration using the entry_points option, and the following commands are available to the user:

- MRPcli --help instead of python3 cli.py --help
- MRPudpp --help instead of python3 udpp.py --help
- MRPproxy --help instead of python3 proxy.py --help

In addition, these commands are available globally in the system without the terminal shell being located in the MRP-library folder.

6.4.1 Documentation

In order to provide comprehensive documentation for the enduser, the source code was documented using Python-docstrings[7] and the Python3.5 type annotations:

- Function description
- Input parameters - using param and type
- Return value - using returns, rtype

The use of type annotations also simplifies further development, as modern IDEs can more reliably display possible methods to the user as an assistance.**??**

```python
# MRPDataVisualisation.py - example docstring
def plot_temperature(_readings: [MRPReading.MRPReading], _title: str = '', _filename: str = None, _unit: str = "degree C") -> str:
    """
    Plots a temperature plot of the reading data as figure
    :param _readings: readings to plot
    :type _readings: list(MRPReading.MRPReading)
    :param _title: Title text of the figure, embedded into the head
    :type _title: str
    :param _filename: export graphic to an given absolute filepath with .png
    :type _filename: str
    :returns: returns the abs filepath of the generated file
    :rtype: str
    """
    if _readings is None or len(_readings) <= 0:
        raise MRPDataVisualizationException("no readings in _reading given")
    num_readings = len(_readings)
    # ...
```

Listing 6.6: Python docstring example

Since ‘docstrings’ only document the source code, but do not provide simple how-to-use instructions, the documentation framework Sphinx[1] was used for this purpose. This framework makes it possible to generate Hypertext Markup Language (HTML) or Portable Document Format (PDF) documentation from various source code documentation formats, such as the used docstrings. These are converted into a Markdown format in an intermediate step and this also allows to add further user documentation such as examples or installation instructions.

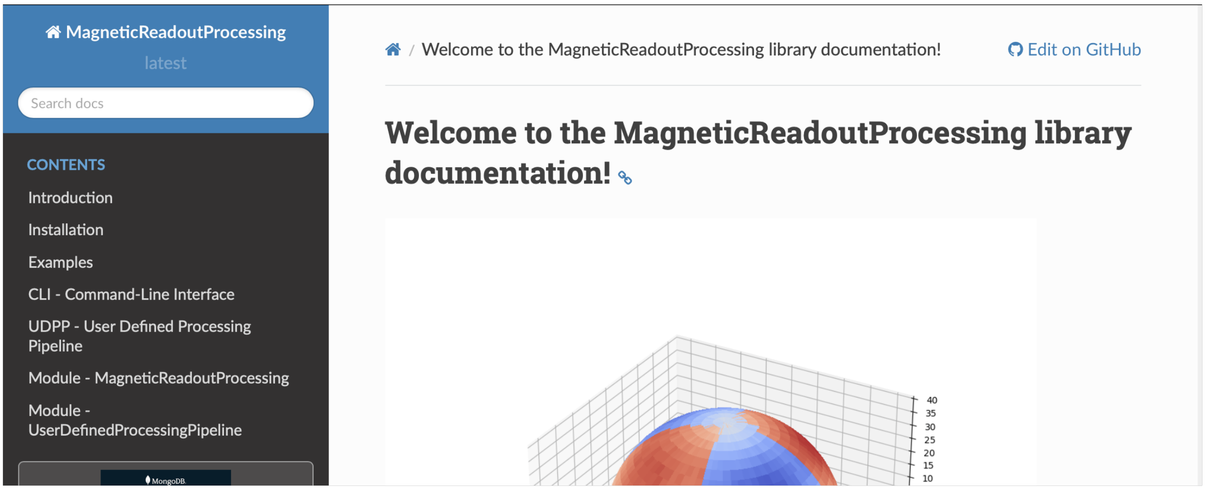


Figure 6-7: MagneticReadoutProcessing documentation hosted on ReadTheDocs

In order to make the documentation created by Sphinx accessible to the user, there are, as with the package management by PyPi services, which provide Python MRP-library documentation online.

Once the finished documentation has been generated from static HTML files, it is stored in the project repository. Another publication option is to host the documentation via online services such as ReadTheDocs[3], where users can make documentation for typical software projects available to others.

The documentation has also been uploaded for ReadTheDocs[2] and linked in the repository and on the overview page6-7 on PyPi.

The process of creating and publishing the documentation has been automated using GitHub Actions[8], so that it is always automatically kept up to date with new features.

7 Evaluation

7.1 Prequesites for evaluation

7.2 Evaluation confiugration

7.2.1 Sensor readout

7.2.2 Processing pipeline

7.3 Test scenarios

7.4 Results

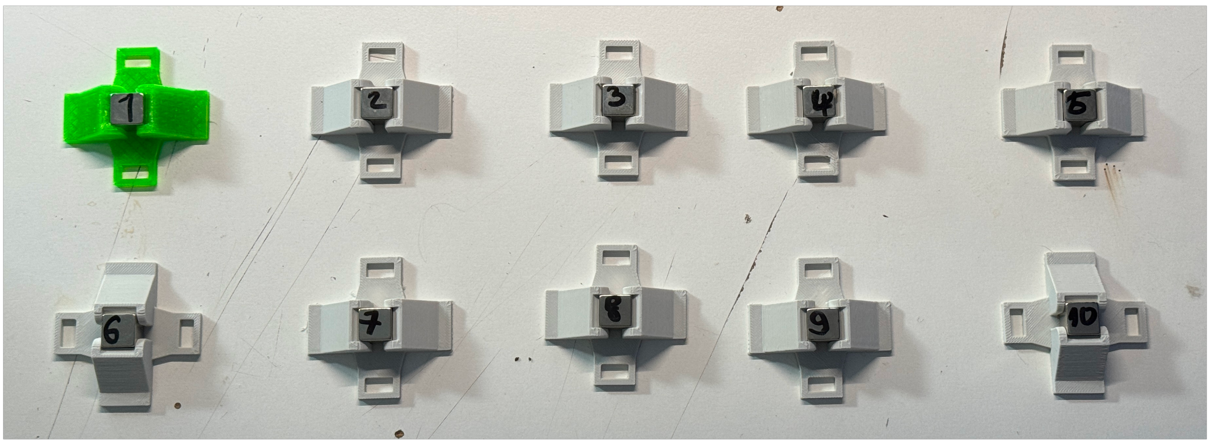


Figure **7**-1: testmagnets in holder

8 Conclusion and dicussion

8.1 Conclusion

8.2 Problems

8.3 Outlook

- magfield camera

Bibliography

[1] DEVELOPERS, Sphinx: *Sphinx makes it easy to create intelligent and beautiful documentation.* https://www.sphinx-doc.org/en/master/. Version: 01.01.2024

[2] DOCS, Inc Read t.: *MagneticReadoutProcessing - ReadTheDocs*. https://magneticreadoutprocessing.readthedocs.io/en/latest/index.html. Version: 01.01.2024

[3] DOCS, Inc Read t.: *ReadTheDocs - Build, host, and share documentation, all with a single platform.* https://readthedocs.com. Version: 01.01.2024

[4] FOUNDATION, Python S.: *MagneticReadoutProcessing - PyPi*. https://pypi.org/project/MagneticReadoutProcessing/. Version: 01.01.2024

[5] FOUNDATION, Python S.: *PyPi - The Python Package Index (PyPI) is a software directory for the Python programming language.* https://pypi.org. Version: 01.01.2024

[6] FOUNDATION, Python S.: *Python Docs - globals*. https://docs.python.org/3/library/functions.html#globals. Version: 01.01.2024

[7] FOUNDATION, Python S.: *Python Enhancement Proposals - Docstring Conventions*. https://peps.python.org/pep-0257/. Version: 01.01.2024

[8] GITHUB, Inc.: *Github Actions - Automate your workflow from idea to production*. https://github.com/features/actions. Version: 01.01.2024

[9] KREKEL, Holger ; TEAM pytest-dev: *pytest: helps you write better programs*. https://docs.pytest.org/en/7.4.x/. Version: 01.01.2024

[10] OCHSENDORF, Marcel: *VoltcraftGM70 REST Interface*. https://github.com/RBEGamer/VoltcraftGM70Rest. Version: 01.01.2024

List of Figures

List of Tables

Listings