

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 systems

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 aquation 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

CAD Computer Aided Design. 6

CDC Communication Device Class. 9

CLI Command Line Interface. 9, 10, 13, 25–27, 33

GPIO General Purpose Input/Output. 18

GUI Graphical User Interface. 25

HAL Hardware Abstraction Layer. 21

HTML Hypertext Markup Language. 34, 35

I2C Inter-Integrated Circuit. 5, 8

IC Integrated Circuit. 5

IDE Integrated development environment. 30, 34

IP Internet Protocol. 23

LUT Lookup Table. 8

MRP MagneticReadoutProcessing. 4, 10, 12, 14, 17, 20, 25–27, 30–33, 35

PC Personal Computer. 7, 9, 10, 17, 18, 20–23, 26, 31

PDF Portable Document Format. 34

PIP Python Package Installer. 32

PPS Puls Per Second. 23

PTP Precision Time Protocol. 23

REST Representational State Transfer. 20, 22

SBC Single Board Computer. 11, 18, 20, 23

UART Universal Asynchronous Receiver / Transmitter. 8, 18

USB Universal Serial Bus. 9, 10, 12, 23

Contents

1	Introduction	1
1.1	Background and Motivation	1
1.1.1	Low-Field MRI	1
1.1.2	Shimming procedure	1
1.2	Aim of this Thesis	1
1.3	Structure	1
2	State of the art	2
2.1	Opensource projects	2
2.2	Conceptual design	2
3	Use cases	3
4	Unified Sensor	4
4.1	Sensor selection	4
4.2	Mechanical Structure	5
4.3	Electrical Interface	6
4.4	Firmware	7
4.4.1	Communication interface	10
4.4.2	Sensor syncronisation interface	10
4.5	Example sensors	14
4.5.1	1D: Single Sensor	14
4.5.2	3D: Fullsphere	15
4.5.3	Integration of an industry teslameter	17
5	Software readout framework	19
5.1	Library requirements	19
5.1.1	Concepts	19
5.1.2	User interaction points	19
5.1.3	Export	19
5.1.4	Multible sensor setup	20
5.1.5	Examples	24

6 Usability improvements	25
6.1 Command Line Interface	25
6.2 Programmable data processing pipeline	27
6.3 Tests	30
6.4 Package distribution	32
6.4.1 Documentation	34
7 Evaluation	36
7.1 Prequesites for evaluation	36
7.2 Evaluation confiugration	36
7.2.1 Sensor readout	36
7.2.2 Processing pipeline	36
7.3 Test scenarios	36
7.4 Results	36
8 Conclusion and dicussion	37
8.1 Conclusion	37
8.2 Problems	37
8.3 Outlook	37
Bibliography	38
List of Figures	40
List of Tables	41
Listings	42

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

- Entwicklung eines hardware und software framework zur einfachen Aquirierung von Magnetfelddaten
- Analysetools und Funktionen

3 Usecases

4 Unified Sensor

The main aim of this project is to develop a low-cost Hall sensor interface that is also universally expandable. The focus is on mapping different sensors and being compatible with different magnet types and shapes. This ensures broad applicability in different scenarios. Another goal is reproducibility to ensure consistent results. Easy communication with standard PC hardware maximises the user-friendliness of the interface. The flexibility to support different sensors and magnets makes the system versatile and opens up possibilities for different applications. This low-cost Hall sensor interface will therefore not only be economically attractive, but also facilitate the integration of Hall sensors in different contexts. In addition, the low-cost Hall sensor interface will serve as a development platform for the data evaluation MagneticReadoutProcessing (MRP) library and provide real measurement data from magnets. In addition, the interface firmware creates a basis for the development of a data protocol for exchanging measured values. This makes it easy to integrate your own measuring devices into the MRP ecosystem at a later date. This is only possible with a minimal functional hardware and firmware setup and was developed for this purpose first.

4.1 Sensor selection

The selection process for possible magnetic field sensors initially focussed on the most common and cost-effective ones, especially those that are already used in smartphones and are therefore widely available. A key aspect of this selection was the preference for sensors with digital interfaces to facilitate implementation in the circuit layout. The integration of integrated temperature sensors represents a significant enhancement that will later enable precise temperature compensation.

Linear analogue Hall sensors were deliberately omitted, although they are suitable for more precise measurements and extended measuring ranges. They were excluded due to the more complex circuit layouts required and more complex power management. In

the context of the desired goal of developing a cost-efficient and universally expandable Hall sensor interface, the decision in favour of digital sensors seems appropriate.

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

Focussing on digital interfaces not only facilitates implementation, but also contributes to overall cost efficiency. At the same time, the integration of temperature sensors enables precise measurements under varying environmental conditions. This strategic choice forms the basis for a flexible, universally applicable Hall sensor interface that can be seamlessly integrated into various existing systems.

The table 4.1 shows a selection of sensors for which hardware and software support has been implemented. The resolution of the selected sensors covers the expected range of values required by the various magnets to be tested.

4.2 Mechanical Structure

The mechanical structure of a sensor is kept very simple for the following tests. The focus was on providing a stable foundation for the sensor Integrated Circuit (IC) and an exchangeable holder for different magnets.

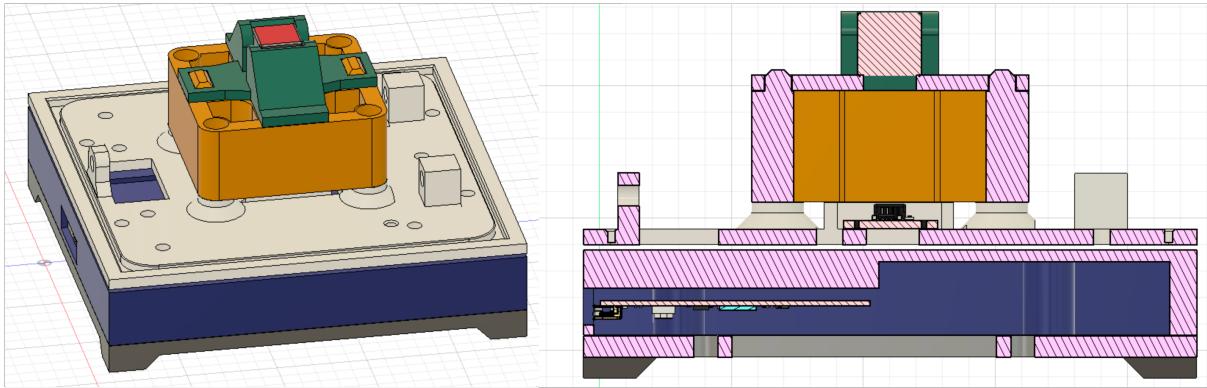


Figure 4-1: 1D sensor mechanical 3D printed structure

The following figure 4-1, shows a sectional view of the Computer Aided Design (CAD) drawing of the 1D: Single sensor 4.5.1.

All parts were produced using the 3D printing process. The sensor circuit board was glued underneath the magnet holder. This is interchangeable, so different distances between sensor and magnet can be realised.

The exchangeable magnetic holder (shown in green) can be adapted to different magnets. It can be produced quickly due to the small amount of material used. The two recesses lock the holder with the inserted magnet over the sensor. Due to predetermined tolerances, the magnet can be inserted into the holder with repeat accuracy and without play. This is important if several magnets have to be measured, where the positioning over the sensor must always be the same.

4.3 Electrical Interface

4-2

- usb
- multiplexer for i2c sensors already implemented
- input / output for sync

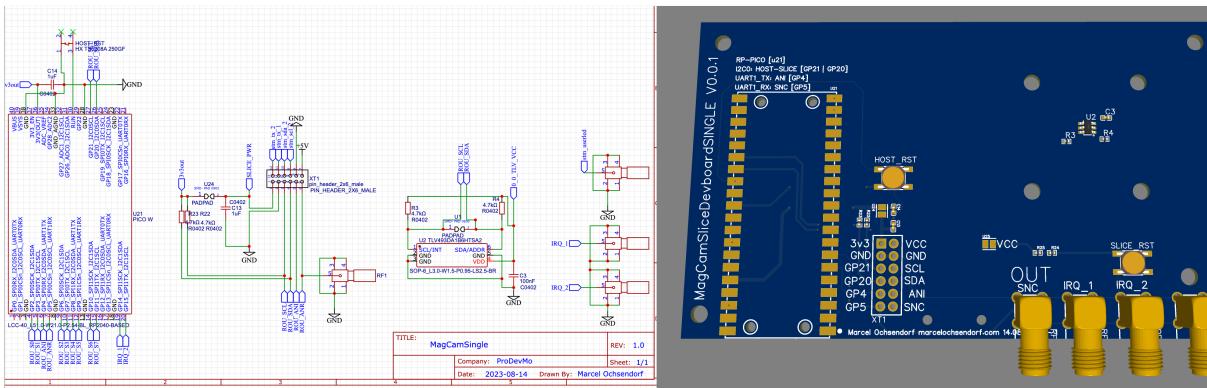


Figure 4-2: 1D sensor schematic and circuit board

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 sensors^{4.1} 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 functions^{4.1} need to be implemented.

```

1 #ifndef __CustomSensor_h__
2 #define __CustomSensor_h__
3 // register your custom sensor in implemented_sensors.h also
4 class CustomSensor: public baseSensor
5 {
6 public:
7     CustomSensor();
8     ~CustomSensor();
9     // implement depending sensor communication interface
10    bool begin(TwoWire& _wire_instance); // I2C
11    bool begin(HardwareSerial& _serial_instance); // UART
12    bool begin(Pin& _pin_instance); // ANALOG or DIGITAL PIN like

```

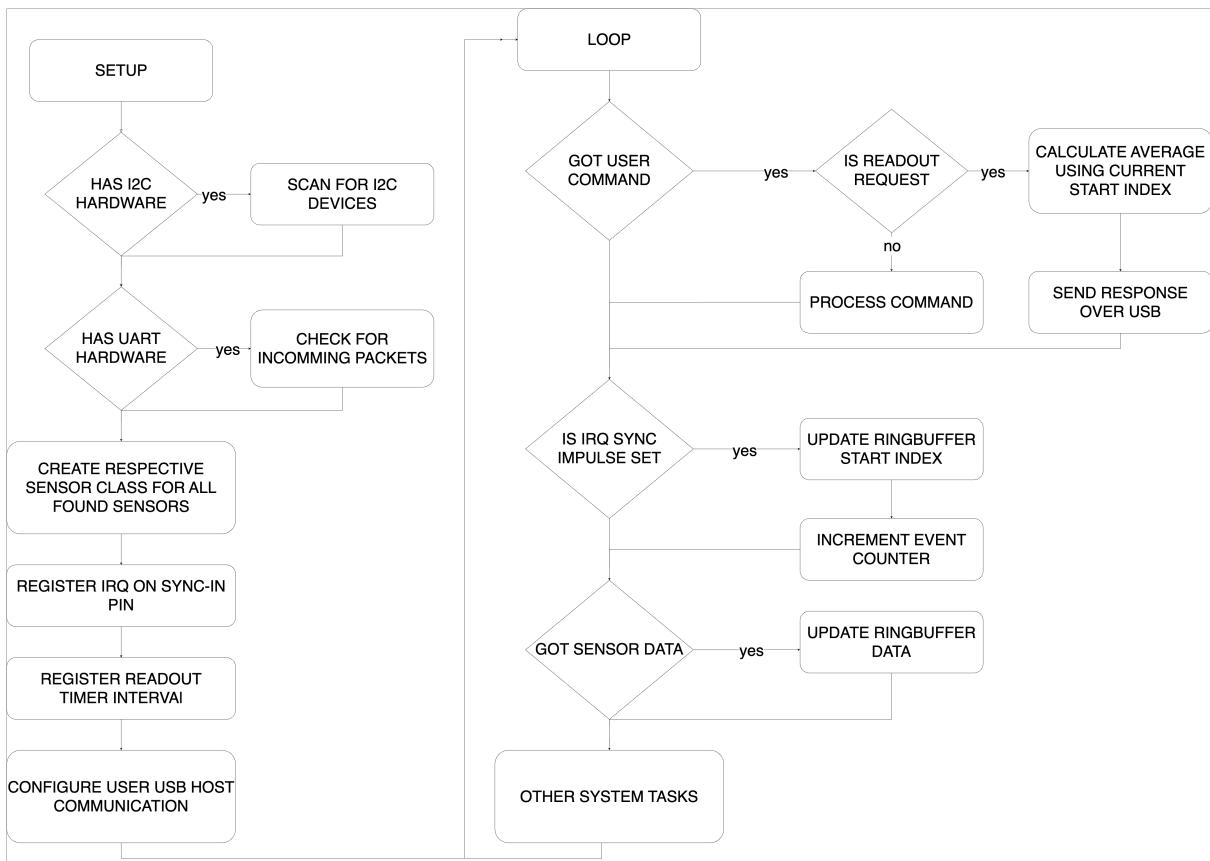


Figure 4-3: Unified sensor firmware simplified program structure

```

onewire
13 // FUNCTIONS USED BY READOUT LOGIC
14 bool is_valid() override;
15 String capabilities() override;
16 String get_sensor_name() override;
17 bool query_sensor() override;
18 sensor_result get_result() override;
19 };
20 #endif
  
```

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

The flow chart 4-3 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 synchronisation 4.4.2 is set up. The last step in

```
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 4-4: Sensors CLI

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 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 synchronized^{4.4.2} in time and are not out of sequence or drift.

```
readsensor b 0  
3279.99
```

Figure 4-5: Query sensors b value using CLI

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 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 reference⁴⁻⁴ to the user after connecting. The current measured value can be output using the `readout` command⁴⁻⁵.

The other option is to use the MRP-library. The serial interface is also used here. However, after a connection attempt by the `MRPHal` module^{5.1.2} 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 syncronisation 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.

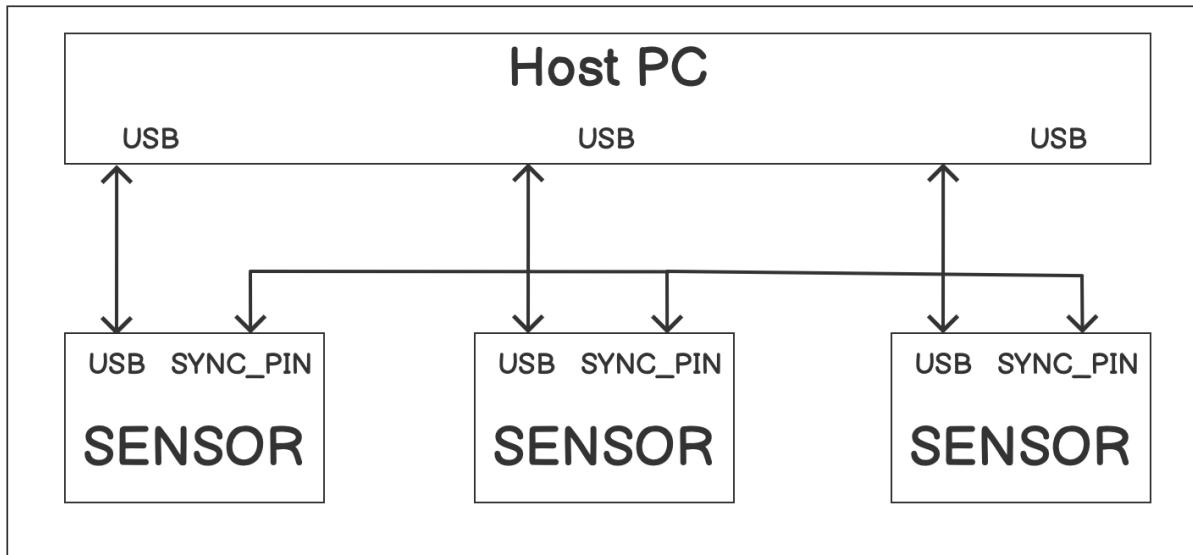


Figure 4-6: Multi sensor synchronisation wiring example

Tabelle 4.2: Measured sensor readout to processing using host software

	Host software	Average Sensor communication time per reading [ms]	Communication jitter time [ms]	Comment
Run	Data point	time [ms]		
1	1	9453	1.44	0
2	1	9864	1.5	0
3	10	12984	1.22	0.9
4	10	12673	1.13	1.1
5	10	43264	2.19	4.2
				High host system cpu load

The table shows various jitter measurements. These were performed on a Raspberry Pi 4 4GB-Single Board Computer (SBC) together with an [1D: Single Sensor](#) 4.5.1 and the following software settings:

- Raspberry Pi OS Lite - debian bookworm x64,

- MRP5-library - version 1.4.1
- Unified Sensor Firmware⁴ - version 1.0.1

It can be seen that a jitter time of up to an additional 1ms is added between the triggering of the measurements by the host system and the receipt of the command by the sensor hardware. If the host system is still under load, this value increases many times over. This means that synchronising several sensors via the USB connection alone is not sufficient.

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 schematic⁴⁻⁶ 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 initial pulse on the sync-pin⁴⁻⁷. 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 interface^{4.4.1} using the `opmode`⁴⁻⁸ 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.

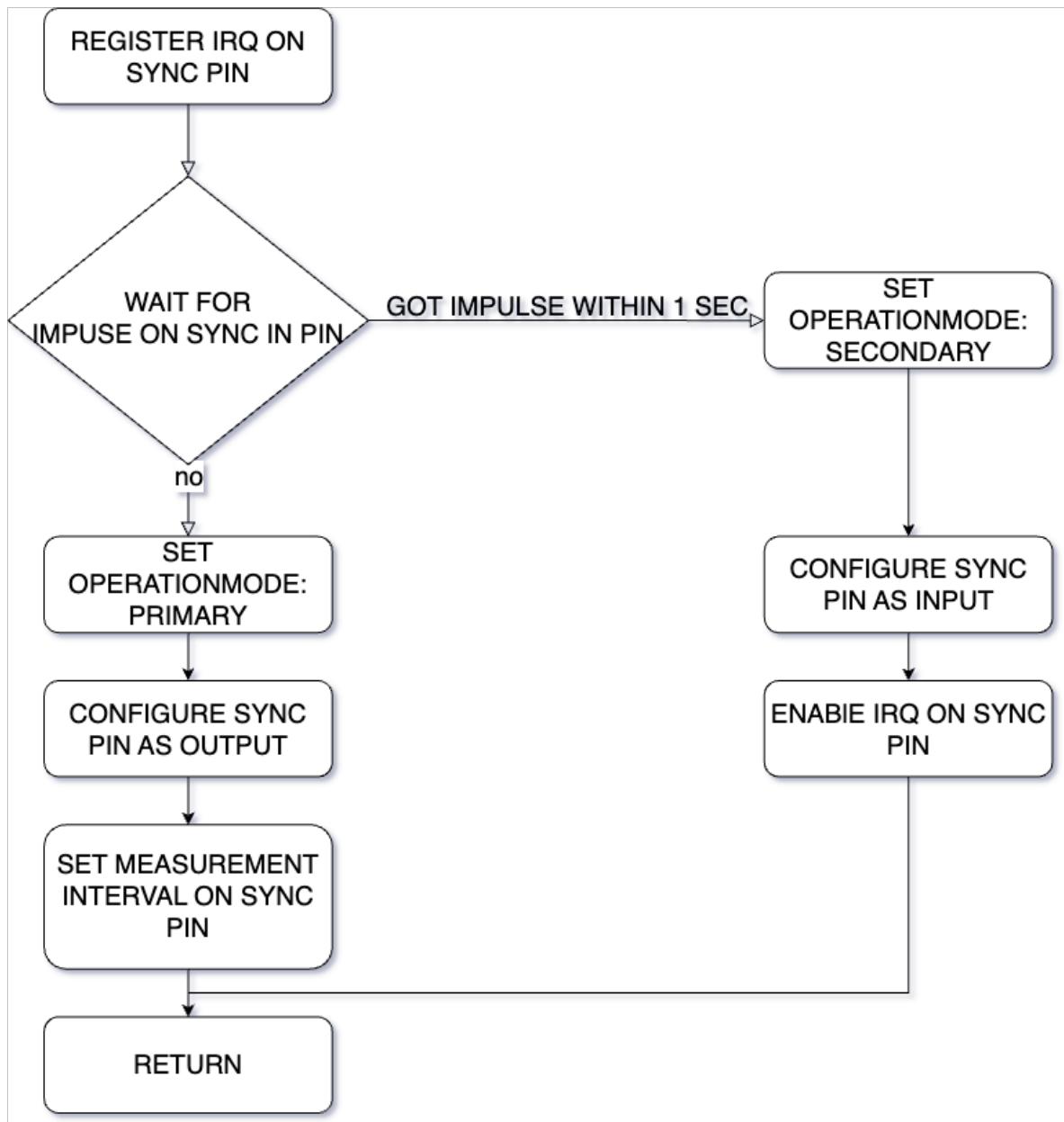


Figure 4-7: Unified sensor firmware multi sensor synchronisation procedure

```

opmode
SECONDARY (got sync after 823ms)
  
```

Figure 4-8: Query opmode using CLI

4.5 Example sensors

Two functional sensor platforms^{4.3} 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 4.3: Build sensors with different capabilities

	1D4.5.1	1D: dual sensor	3D: Fullsphere4.5.2
Maximal magnet size	Cubic 30x30x30	Cubic 30x30x30	Cubic 20x20x20
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](#) sensor⁴⁻⁹ is the simplest possible sensor that is compatible with the [Unified Sensor Firmware](#)^{4.4} platform and also with the MRP-library⁵.

The electrical level here is based on a [Raspberry-Pi Pico](#) together with the [MMC5603NJ](#)

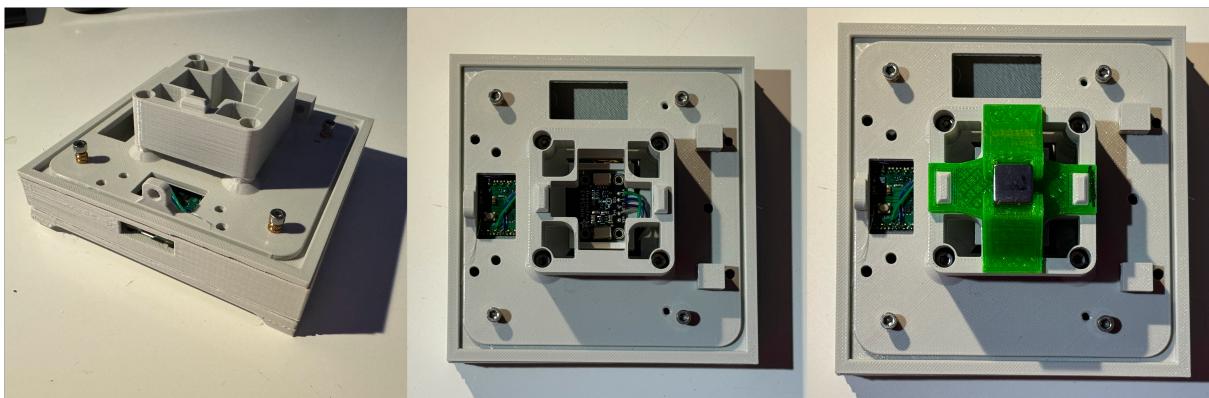


Figure 4-9: 1D sensor construction with universal magnet mount

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 measurement without introducing measurement irregularities by mechanically changing the magnet.

4.5.2 3D: Fullsphere

The 3D [Fullsphere](#) sensor offers the possibility to create a 3D map 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

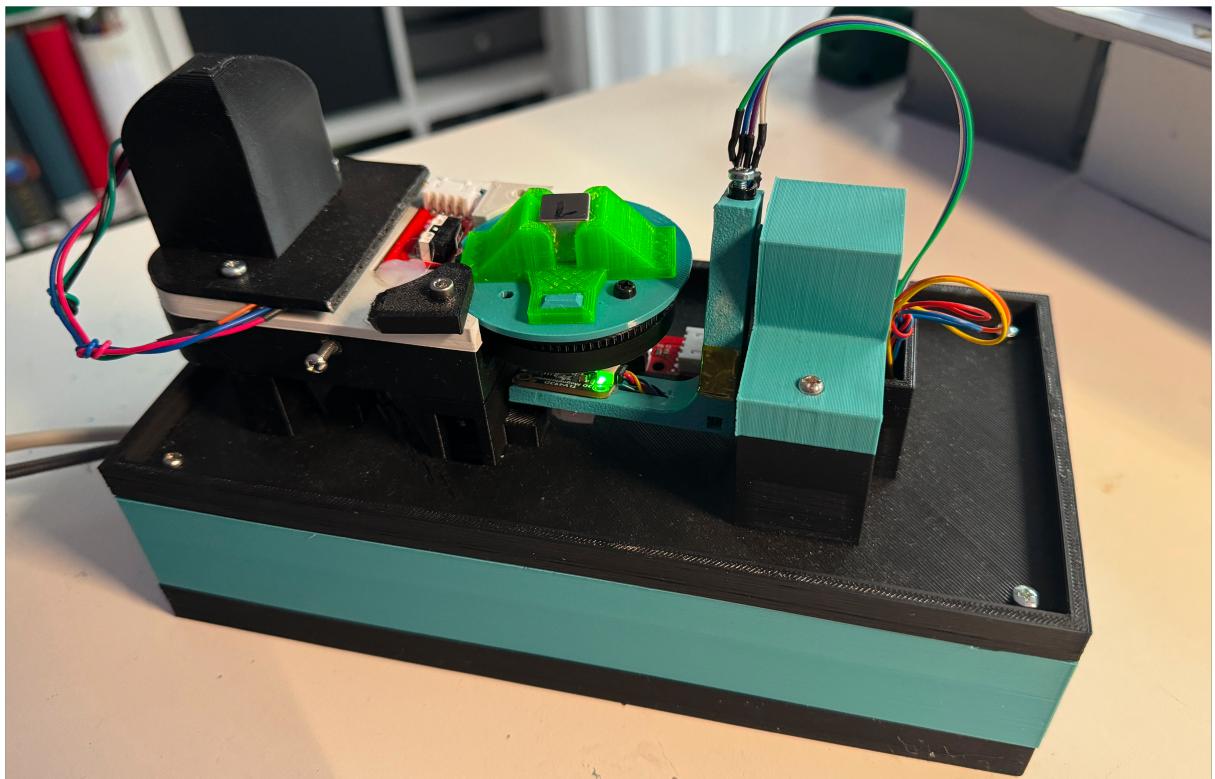


Figure 4-10: Full-Sphere sensor implementation using two Nema17 stepper motors in a polar coordinate system

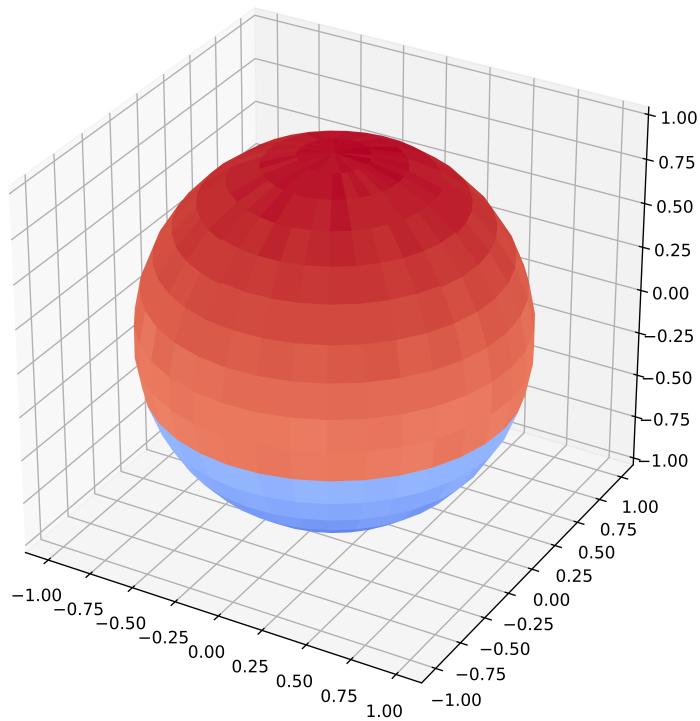


Figure 4-11: 3D plot of an N45 12x12x12 magnet using the 3D fullsphere sensor

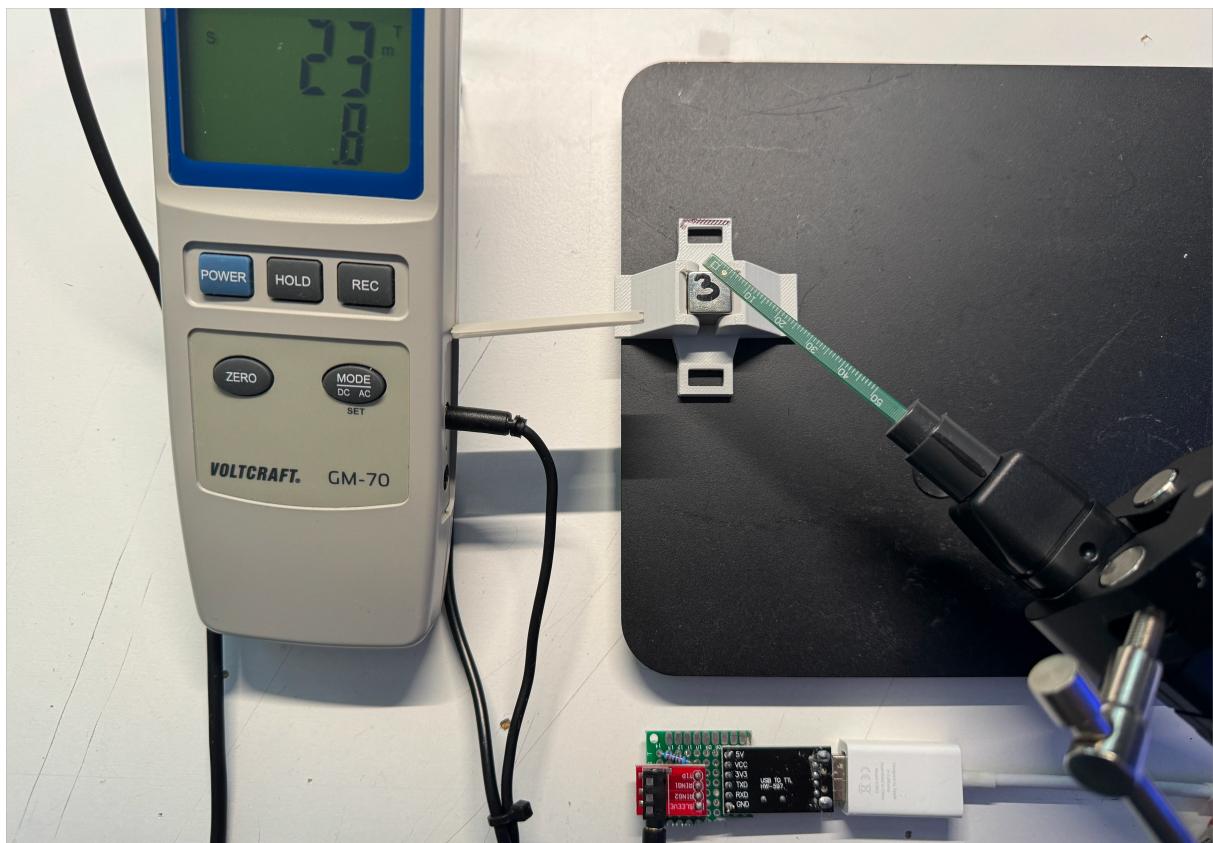


Figure 4-12: Voltcraft GM70 teslameter with custom PC interface board

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 is shown here using a temperature-compensated [Voltcraft GM-70](#) teslameter⁴⁻¹², which has a measuring range of 0–3T with a resolution of 0.1mT. This offers an [RS232](#) interface with a documented protocol^{4.4} 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 Firmware](#)^{4.4} offers.

Another option is a custom interface board between the meter and the PC. This is a good option as many modern PCs or SBCs 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 Firmware](#)4.4 was adapted using a separate build configuration and the protocol of the measuring device was implemented.

Tabelle 4.4: 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 erläutern

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

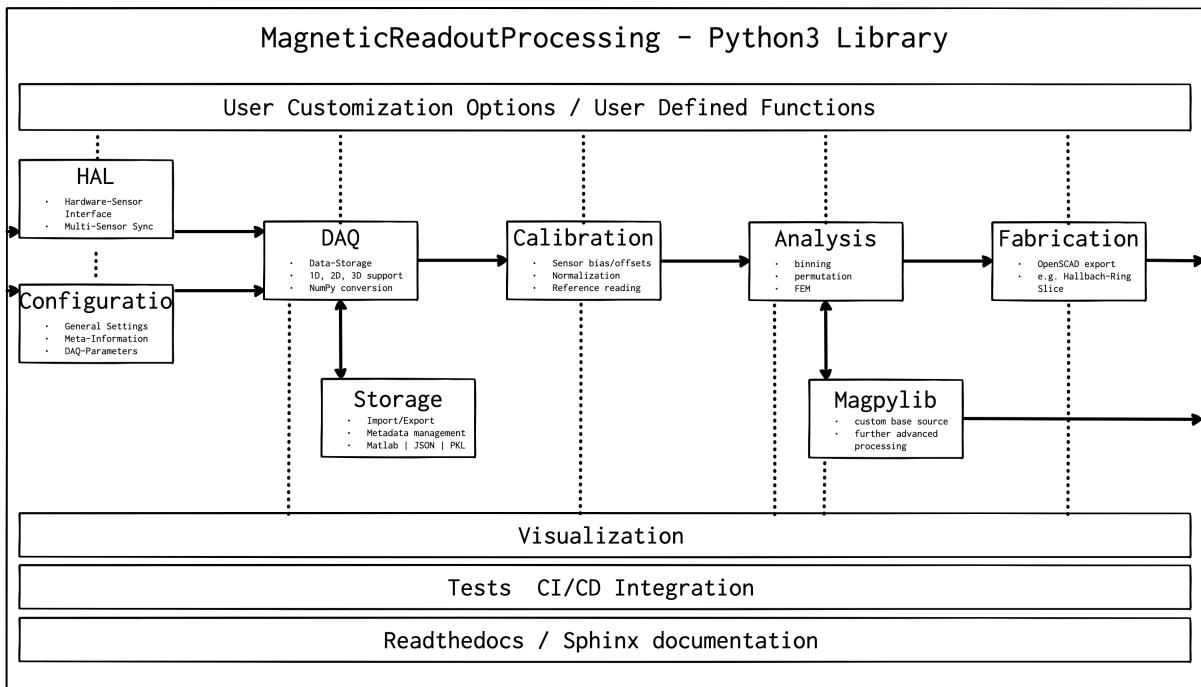


Figure 5-1: MRPlib COMPLETE FLOW

- 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](#) 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

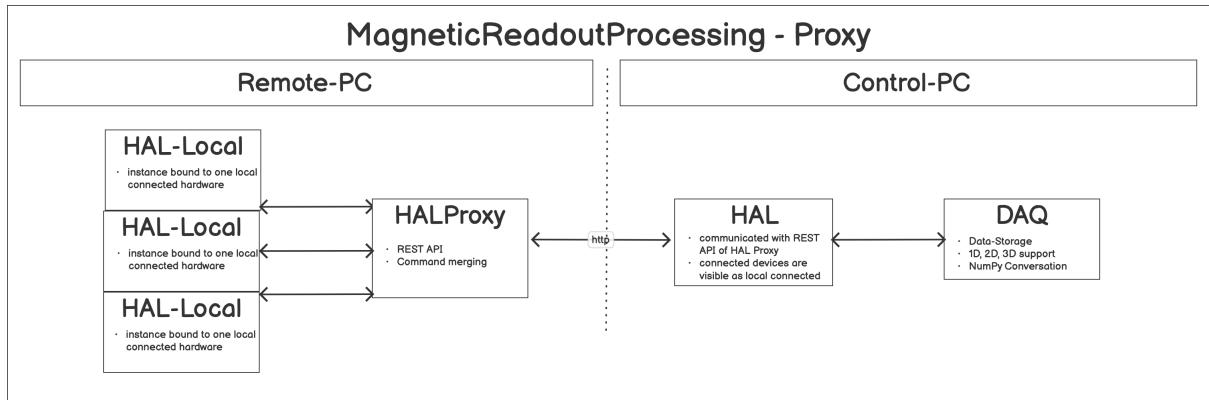


Figure 5-2: MRPlib Proxy Module

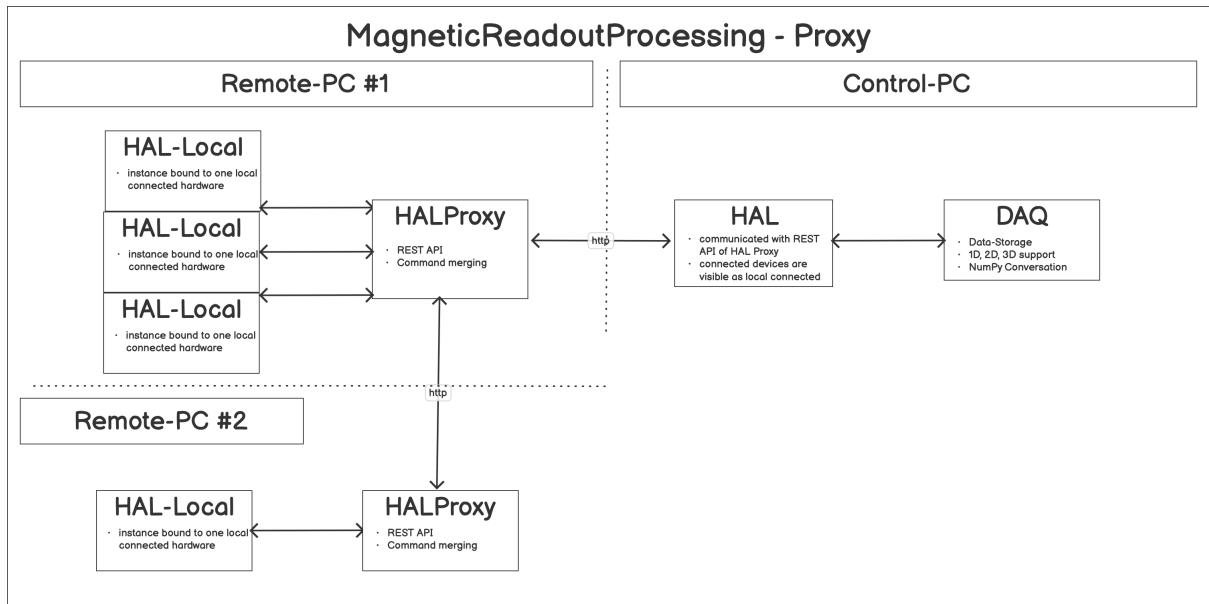


Figure 5-3: mrp proxy multi

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](#)

5 Software readout framework

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 started^{5.1} on the remote PC side.

```
1 # START PROXY INSTNACE WITH TWO LOCALLY CONNECTED SENSORS
2 $ 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
3 Proxy started. http://remote_pc.local:5556/
4 PRECHECK: SENSOR_HAL: 1337 # SENSOR A FOUND
5 PRECHECK: SENSOR_HAL: 4242 # SENSOR B FOUND
6 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}

```
1 # GET PROXY STATUS
2 $ wget http://proxyinstance.local:5556/proxy/status
3 {
4     "capabilities": [
5         "static",
6         "axis_b",
7         "axis_x",
8         "axis_y",
9         "axis_z",
10        "axis_temp",
11        "axis_stimestamp"
12    ],
13    "commands": [
14        "status",
15        "initialize",
16        "disconnect",
17        "combinedsensorcnt",
18        "sensorcnt",
19        "readsensor",
20        "temp"
21    ]
22
23 # RUN A SENSOR COMMAND AND GET THE TOTAL SENSOR COUNT
24 $ wget http://proxyinstance.local:5556/proxy/command?cmd=
   combinedsensorcnt
25 {
26     "output": [
27         "2"
28     ]
```

```
29 }  
30 }
```

Listing 5.2: MRProxy REST endpoint query examples

The query result shows that the sensors are connected correctly and that their capabilities 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 required.

```
1 # CONFIGURE MEASUREMENT JOB USING A PROXY INSTANCE  
2 $ MRPcli config setsensor testcfg --path http://proxyinstance.local  
   :5556  
3 > remote sensor connected: True using proxy connection:  
4 > http://proxyinstance.local:5556 with 1 local sensor connected
```

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

Sensor Synchronisation

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 synchronisation 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

- nummerierung zuerst lokale sensoren dann weitere proxy sensoren

- commando templating
- einzelne sensor capabilites werden gemerged
- ids werden aufsummiert
- spätere sensor identifikation geschieht über die jeweilige sensor id, welche über diesen index abgefragt werden kann
- table mit margin 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

6 Usability improvements

examples are provided in the documentation, it must be possible for programming beginners in particular to use them. To simplify these tasks, a CLI6-2 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:

```
1 # CLI EXAMPLE FOR CONFIGURING A MEASUREMENT RUN
2 ## CONFIGURE THE SENSOR TO USE
3 $ MRPcli config setupsensor testcfg
4 > 0 - Unified Sensor 386731533439 - /dev/cu.usbmodem3867315334391
5 > Please select one of the found sensors [0]:
6 > sensor connected: True 1243455
7 ## CONFIGURE THE MEASUREMENT
8 $ MRPcli config setup testcfg
9 > CONFIGURE testcfg
10 > READING-NAME: [testreading]: testreading
11 > OUTPUT-FOLDER [/ cli/reading]: /tmp/reading_folder_path
12 > NUMBER DATAPOINTS: [1]: 10
13 > NUMBER AVERAGE READINGS PER DATAPOINT: [1]: 100
14 # RUN THE CONFIGURED MEASUREMENT
15 $ MRPcli measure run
16 > STARTING MEASUREMENT RUN WITH FOLLOWING CONFIGS: [ 'testcfg' ]
17 > config-test: OK
18 > sensor-connection-test: OK
19 > START MEASUREMENT CYCLE
20 > sampling 10 datapoints with 100 average readings
21 > SID:0 DP:0 B:47.359mT TEMP:23.56
22 > ....
23 > dump_to_file testreading_ID:525771256544952_SID:0_MAG:
N45_CUBIC_12x12x12.mag.json
```

Listing 6.1: CLI example for configuring a measurement run

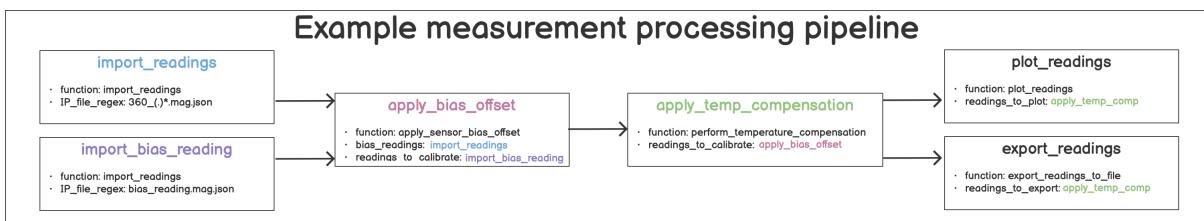


Figure 6-2: example measurement analysis pipeline

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 example6-2 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 steps6-2 as `yaml` syntax.

```
1 stage import_readings:
2   function: import_readings
3   parameters:
4     IP_input_folder: ./readings/fullsphere/
5     IP_file_regex: 360_(.)*.mag.json
6
7 stage import_bias_reading:
8   function: import_readings
9   parameters:
10    IP_input_folder: ./readings/fullsphere/
11    IP_file_regex: bias_reading.mag.json
12
13 stage apply_bias_offset:
14   function: apply_sensor_bias_offset
15   parameters:
16     bias_readings: stage import_bias_reading # USE RESULT FROM FUNCTION
17       import_bias_reading
17     readings_to_calibrate: stage import_readings
18
19 stage apply_temp_compensation:
20   function: apply_temperature_compensation
21   parameters:
22     readings_to_calibrate: stage import_readings # USE RESULT FROM
23       FUNCTION import_readings
24
24 stage plot_normal_bias_offset:
25   function: plot_readings
26   parameters:
27     readings_to_plot: stage apply_temp_compensation
28     IP_export_folder: ./readings/fullsphere/plots/
29     IP_plot_headline_prefix: Sample N45 12x12x12 magnets calibrated
30
31 stage export_readings:
32   function: export_readings
33   parameters:
34     readings_to_plot: stage apply_temp_compensation
35     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:

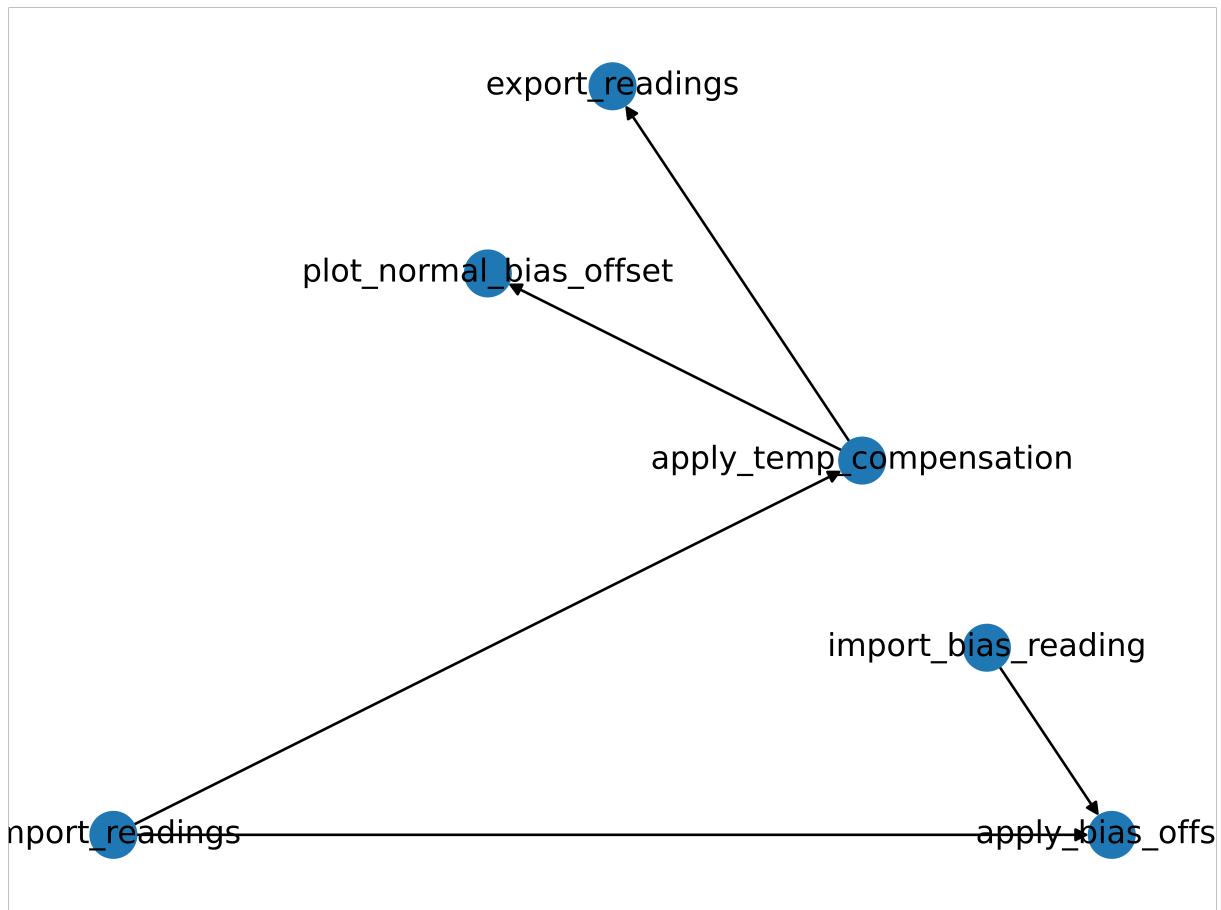


Figure 6-3: Result step execution tree from user defined processing pipeline example

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

To determine the order of the pipeline steps, the parser script creates 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 tree⁶⁻³. The individual steps are then executed along the graph. The intermediate results and the final results⁶⁻⁴ are saved for optional later use.

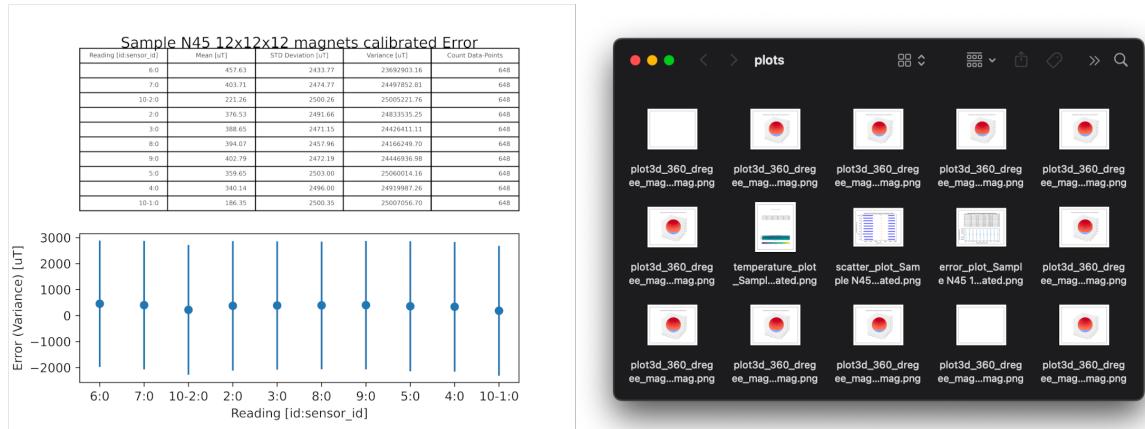


Figure 6-4: pipeline output files after running example pipeline on a set of readings

```

test_MPRReading
  ✓ TestMPRReading           381 ms
    ✓ test_cartesian_reading  0 ms
    ✓ test_export_reading    7 ms
    ✓ test_import_reading   0 ms
    ✓ test_matrix_init       372 ms
    ✓ test_reading_init      2 ms
  > ✘ test_MPReadoutSource  0 ms
> Ø test_MRPSimulation    0 ms
  ✓ test_SensorAnalysis     15 sec 850 ms
    ✓ TestMRPDataVisualization 15 sec 850 ms
      ✓ test_error_visualisation 3 sec 72 ms
      ✓ test_mean               0 ms
      ✘ test_scatter_visualisation 2 sec 876 ms
      ✓ test_std_deviation      0 ms
      ✓ test_temperature_visualisation 9 sec 902 ms
Skipped
SKIPPED [ 7%]
Skipped
SKIPPED [ 8%]
Skipped

test_SensorAnalysis.py::TestMRPDataVisualization::test_mean
test_SensorAnalysis.py::TestMRPDataVisualization::test_scatter_visualisation
test_SensorAnalysis.py::TestMRPDataVisualization::test_std_deviation
test_SensorAnalysis.py::TestMRPDataVisualization::test_temperature_visualisation
test_SensorAnalysis.py::TestMRPDataVisualization::test_variance
test_misc.py::TestMPRReading::test_full_sphere_reading PASSED [ 84%]PASSED [ 89%]PASSED [ 92%]PASSED [ 94%]

=====
===== 2 failed, 22 passed, 15 skipped, 13 warnings in 49.36s =====
PASSED [100%]
Process finished with exit code 1

```

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 ensure consistent and reliable performance, which is particularly important when libraries are used by different users and for different use cases.

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 variants as examples for the documentation.

```

1 class TestMPRReading(unittest.TestCase):
2     # PREPARE A INITIAL CONFIGURATION FILE FOR ALL FOLLOWING TEST CASES
3     # IN THIS FILE
4     def setUp(self) -> None:
5         self.test_folder: str = os.path.join(os.path.dirname(os.path.
6             abspath(__file__)), "tmp")
7         self.test_file:str = os.path.join(self.
8             import_export_test_folderpath , "tmp")
9
10    def test_matrix(self):
11        reading: MRPReading = MRPSimulation.generate_reading()
12        matrix: np.ndarray = reading.to_numpy_matrix()
13        n_phi: float = reading.measurement_config.n_phi
14        n_theta: float = reading.measurement_config.n_theta
15        # CHECK MATRIX SHAPE
16        self.assertTrue(matrix.shape != (n_theta,))
17        self.assertTrue(len(matrix.shape) <= n_phi)
18
19    def test_export_reading(self) -> None:
20        reading: MRPReading = MRPSimulation.generate_reading()
21        self.assertIsNotNone(reading)
22        # EXPORT READING TO A FILE
23        reading.dump_to_file(self.test_file)
24
25    def test_import_reading(self):
26        # CREATE EMPTY READING AND LOAD FROM FILE
27        reading_imported:MRPReading = MRPReading.MRPReading(None)
28        reading_imported.load_from_file(self.test_file)
29        # COMPARE
30        self.assertIsNotNone(reading_imported.compare(MRPSimulation.
31            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

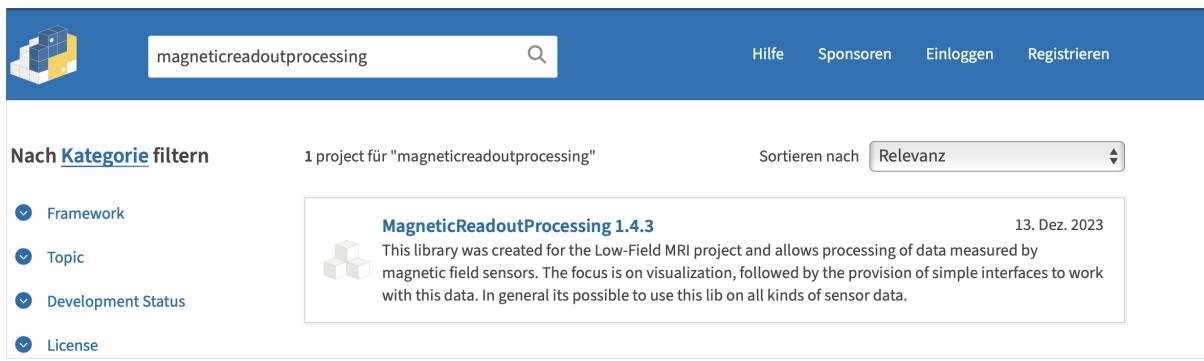


Figure 6-6: MagneticReadoutProcessing library hosted on PyPi

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.

```
1 # https://pypi.org/project/MagneticReadoutProcessing/
2 # install the latest version
3 $ pip3 install MagneticReadoutProcessing
4 # install the specific version 1.4.0
5 $ 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.

6 Usability improvements

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.py` for this purpose.

```
1 # dynamic requirement loading using 'requirements.txt'
2 req_path = './requirements.txt'
3 with pathlib.Path(req_path).open() as requirements_txt:
4     install_requires = [str(requirement) for requirement in
5         pkg_resources.parse_requirements(requirements_txt)]
6
7 setup(name='MagneticReadoutProcessing',
8       version='1.4.3',
9       url='https://github.com/LFB-MRI/MagnetCharacterization/',
10      packages=[ 'MRP', 'MRPcli', 'MRPudpp', 'MRPproxy'],
11      install_requires=install_requires,
12      entry_points={
13          'console_scripts': [
14              'MRPCli = MRPcli.cli:run',
15              'MRPUdpp = MRPUdpp.udpp:run',
16              'MRPproxy = MRPproxy.mrpproxy:run'
17          ]
18      })
19
```

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.??

```
1  # MRPDataVisualisation.py – example docstring
2  def plot_temperature(_readings: [MRPReading.MRPReading], _title:
3      str = '', _filename: str = None, _unit: str = "degree C") -> str:
4      """
5          Plots a temperature plot of the reading data as figure
6          :param _readings: readings to plot
7          :type _readings: list(MRPReading.MRPReading)
8          :param _title: Title text of the figure , embedded into the head
9          :type _title: str
10         :param _filename: export graphic to an given absolute filepath
11         with .png
12         :type _filename: str
13         :returns: returns the abs filepath of the generated file
14         :rtype: str
15         """
16         if _readings is None or len(_readings) <= 0:
17             raise MRPDataVisualizationException("no readings in
18             _reading given")
19             num_readings = len(_readings)
20             # ...
```

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.

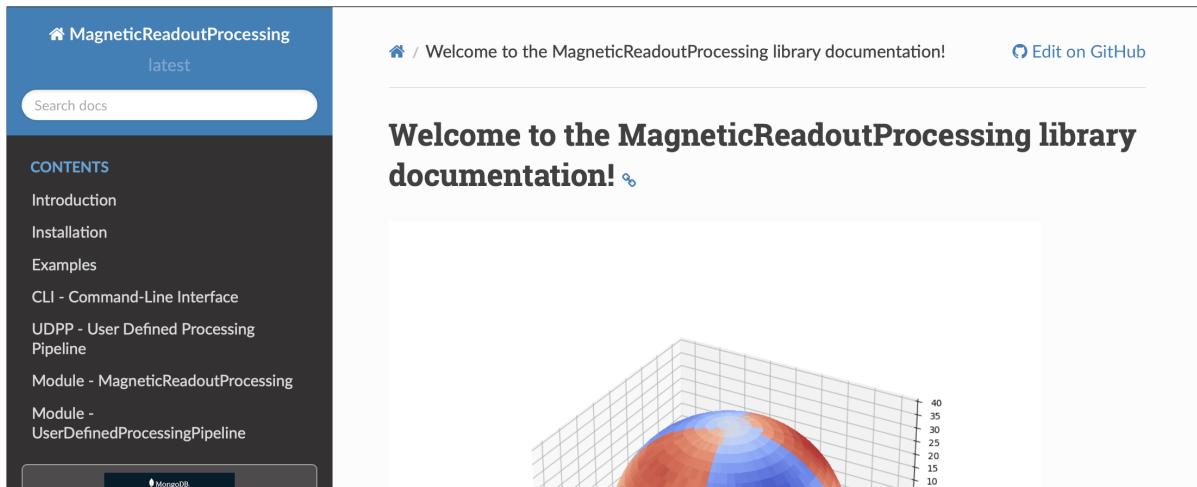


Figure 6-7: MagneticReadoutProcessing documentation hosted on ReadTheDocs

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 page**6-7** 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 configuation

7.2.1 Sensor readout

7.2.2 Processing pipeline

7.3 Test scenarios

7.4 Results

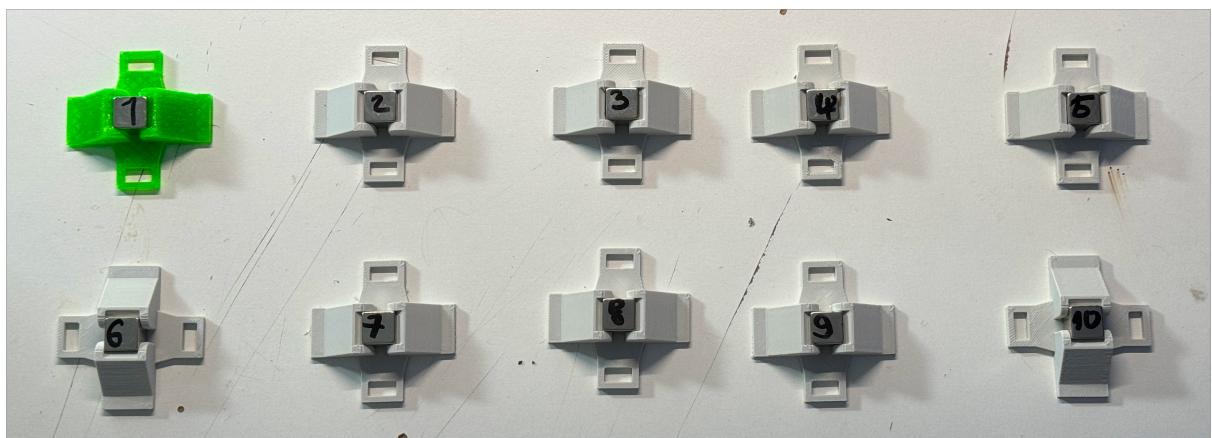


Figure 7-1: testmagnets in holder

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

Bibliography

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

List of Figures

4-1	1D sensor mechanical 3D printed structure	6
4-2	1D sensor schematic and circuit board	7
4-3	Unified sensor firmware simplified program strucutre	8
4-4	Sensors CLI	9
4-5	Query sensors b value using CLI	10
4-6	Multi sensor synchronisation wiring example	11
4-7	Unified sensor firmware multi sensor synchronisation procedure	13
4-8	Query opmode using CLI	13
4-9	1D sensor construction with universal magnet mount	15
4-10	Full-Sphere sensor implementation using two Nema17 stepper motors in a polar coordinate system	16
4-11	3D plot of an N45 12x12x12 magnet using the 3D fullsphere sensor . .	16
4-12	Voltcraft GM70 teslameter with custom PC interface board	17
5-1	MRPlib COMPLETE FLOW	20
5-2	MRPlib Proxy Module	21
5-3	mrp proxy multi	21
6-1	MRP CLI output to configure a new measurement	25
6-2	example measurement analysis pipeline	27
6-3	Result step execution tree from user defined processing pipeline example	29
6-4	pipeline output files after running example pipeline on a set of readings	30
6-5	MRP library test results for different submodules executed in PyCharm IDE	30
6-6	MagneticReadoutProcessing library hosted on PyPi	32
6-7	MagneticReadoutProcessing documentation hosted on ReadTheDocs	35
7-1	testmagnets in holder	36

List of Tables

4.1	Implemented digital halleffect sensors	5
4.2	Measured sensor readout to processing using host software	11
4.3	Build sensors with different capabilities	14
4.4	Voltcraft GM70 serial protocol	18

Listings

4.1	CustomSensor-Class for adding new sensor hardware support	7
5.1	MRPproxy usage to enable local sensor usage over network	22
5.2	MRPproxy REST enpoiint query examples	22
5.3	MRPcli usage example to connect with a network sensor	23
6.1	CLI example for configuring a measurement run	26
6.2	Example User Defined Processing Pipeline	27
6.3	Example pytest class for testing MRPReading module functions	30
6.4	Bash commands to install the MagneticReadoutProcessing (+mrp)-library using pip	32
6.5	setup.py with dynamic requirement parsing used given requirements.txt	33
6.6	Python docstring example	34